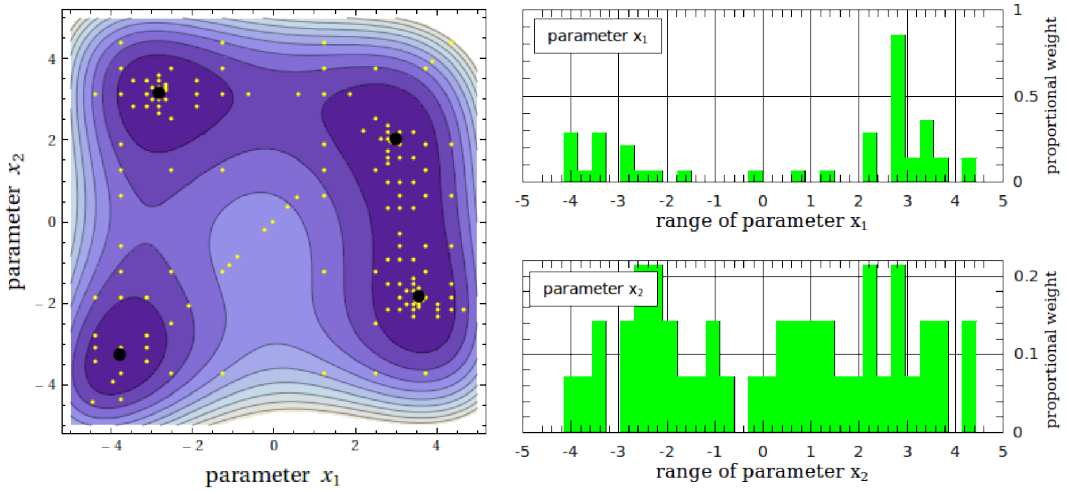
MAGIX manual

T. Möller, D. Panoglou

May 29, 2020



Version 2.1.0

Contents

1 Introduction

1.1 What is MAGIX?

Modeling of astronomical observations requires specialized numerical codes and knowledge about how to use them. MAGIX[1] provides a framework of an easy interface between existing (registered) codes with an iterating engine that attempts to minimize deviations of the model results from available observational data, constraining the values of the model parameters and providing according error estimates.

MAGIX is a model optimizer developed under the framework of CATS, which is a german–french–swedish project aimed to provide common tools and databases for astrophysical applications. (M)any models (and, in principle, not only astrophysical models) can be plugged into MAGIX so as to explore their parameter space and find the set of parameter values that best fits observational/experimental data.

MAGIX compiles with the data structures and reduction tools of ALMA. It aims to be a tool that can be used with observations assembled with the ALMA interferometer, but can be used even with non-astronomical data. MAGIX is under construction, but already operational. At the moment MAGIX is a command line based program; therefore it can additionally be called by other programs, e.g. CASA.

People willing to test it and provide us with feedback on this manual, as well on the actual use of MAGIX, are welcome. In that case, please contact Peter Schilke via email (schilke@ph1.uni-koeln.de).

1.2 How to use this manual

- To begin with, appendix §A.1 includes some basic terminology so that we all understand what the other speaks of, as well as abbreviations referred to within the text, names of software programs etc.
- If one wants to use MAGIX, it is recommended that one reads all of this first chapter (§1). This will allow the user to get to know what MAGIX is about so as to see if s/he needs to use it, why, and what s/he has to expect for as the result (§ 1.1), how to install MAGIX (§ 1.3) and how to make it work (§ 1.4).
- The second section describes the so-called I/O control file: It contains the paths and names of the input XML files and the output log file (§2).
- The third section describes the so-called experimental file: It contains all settings for importing experimental/observational data (§3).
- The fourth section describes the so-called instance: Within the instance file the (starting) values and other properties for all parameters defined in the input file(s) of the external model program are set (§4).
- The fifth section includes the description of the so-called fit control file: It contains the settings and directives for the fit procedure that has to be followed by MAGIX (§5).
- The sixth section contains a detailed description of the output files created by MAGIX during the fit process (§6).
- The seventh section describes the registration process, especially the so-called registration file: It describes the input and output file(s) being used by the external program,

[1] MAGIX home page: http://www.astro.uni-koeln.de/projects/schilke/MAGIX

giving names to the parameters contained in each file, and setting their type (string, integer or real number) and the format with which they should be written (§7)

- Finally, chapter § 8 gives short descriptions of the available algorithms, giving insight of how they work and hints of how to make a better use of them in order to optimize parameters of the model in question.

The above list includes references to the sections of this document, where the format of the XML files is described, while the rules that should be followed when creating/editing such a file are listed.

1.3 Requirements

The following packages are required:

- `python 2.6` (or later) NOTE, Ubuntu users have to install the package `python-dev` as well.
- `numpy 1.3` (or later)
- `scipy 0.9` (or later)
- `subversion` (only for NR version)
- `pyfits 1.0` (or later)
- `gfortran 4.3` (or later)
- `matplotlib 0.99` (or later)
- `OpenMPI 1.5` (or later), only required for MPI version of MAGIX

1.4 Installation and usage

- In order to install the SMP parallelized version of MAGIX, simply download the zip file from CATS web site: https://www.astro.uni-koeln.de/wd-schilke/CATS/ (username `magix`, password `Magix4ever`), extract the zip file, go to the new created directory, and start the installation script from the the command prompt:

```
./install.sh
```

  In order to install the MPI parallelized version of MAGIX, add the MPI flag to the call of the installation script, i.e.

```
./install.sh --mpi
```

- MAGIX can be started by typing

```
./magix_start.py path-to-your-files/io_control.xml
```

  at the command prompt.
- In order to get help, type

```
./magix_start.py --help
```

  or read the `readme.txt` file in the installation directory.
- If no screen output is desired (except error messages), type

```
./magix_start.py --quiet PathToYourFiles/io_control.xml
```

followed by the path of the I/O control file (§2).

- If you do not want to plot the resulting fit function (or if you have problems with the `matplotlib` package), type

```
./magix_start.py --noplot PathToYourFiles/io_control.xml
```

followed by the path of the I/O control file (§2).

Please note, the `--noplot` option prevents the usage of the Internal Nested Sampling as well as the Error Estimation algorithms.

- If you do not want to work with the interactive GUI of `matplotlib` but create all plots and save them into files, type

```
./magix_start.py --plotsaveonly PathToYourFiles/io_control.xml
```

followed by the path of the I/O control file (§2).

- In order to stop MAGIX after the first call of the external model function, set the so-called debug flag `--debug`. This flag can be very helpful to analyze problems occurring with the call of the external model program (§7.3).

1.5 Examples

Note, the MAGIX installation includes the subdirectory `run/examples/`. This subdirectory contains directories for every algorithm included in MAGIX (§ 8) where all xml-files as well as all output files are stored in, which are created during a fit process:

For example, the directory `run/examples/Levenberg-Marquardt/` contains all xml-files and all output files, which are created during a fit process using the Levenberg-Marquardt algorithm. In order to recreate the output files, type

```
./magix_start.py run/examples/Levenberg-Marquardt/io_control.xml
```

1.6 Environment variables

Before you use MAGIX, you might need to set some general environment variables (§ 1.6.1). Those environment variables have to be set by typing the corresponding commands at the command line (or adding the corresponding line in your `~/.bashrc` file, if you want a permanent setting for the corresponding environment variable).

Note, the following environment variables must not be changed during a fit process.

1.6.1 General environment variables

- `MAGIXWorkingDir` is the root directory of MAGIX. Typically, it is the directory where you run the `install_magix.sh` script in order to install MAGIX (§1.4). But in fact, in order that everything is functionable, the MAGIX root directory only needs to contain the directory `Modules` and the script `magic_start.py`.
- During the fit process, MAGIX creates several subdirectories located in a temporary directory (`temp`) which is by default located in the root directory of MAGIX (it is created if it doesn't exist). By setting the environment variable `MAGIXTempDirectory`

```
export MAGIXTempDirectory="temp_somewhere_else"
```

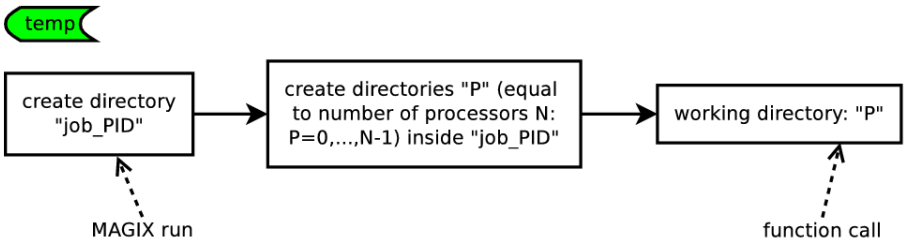


Figure 1: Creation of temporary directories by MAGIX during the fit process. `MAGIXTempDirectory` is the container of the `job_PID` directory. Each run of MAGIX takes place in a `job_PID` directory. Each function call (i.e. each call of the external model program by MAGIX) makes the model program run in one of the `P` directories, which are located inside `job_PID`.

the user can define another location of this temporary directory. It is strongly recommended that the user should use a RAM drive, i.e., set the environment variable to

```
export MAGIXTempDirectory="/dev/shm/MAGIX/"
```

whenever possible. (The RAM drive is a common name for a temporary file storage facility on many Unix-like operating systems. The usage of a RAM drive improve the performance of MAGIX because the input and output file(s) of the external model programs are not written to the hard drive but to the RAM which is orders of magnitude faster.)

The process of the creation of temporary directories is shown in fig. 1. For each run of MAGIX, one directory is created inside `MAGIXTempDirectory`. The name of the newly created directory is the string `job_` with a number appended. This number corresponds to the extended PID[2]. Inner directories are created inside `job_PID`, one for each thread, i.e. one for each processor working on the given optimization process (see also a representation of the directory tree in fig. 6). The same procedure is followed when parallelization is not allowed (`<ParallelizationPossible>` (§7.7) is set to `no`); in that case there will be only one thread and `job_PID` will contain one directory named `0`.

A detailed description of the management of MAGIX can be found in (§7.7).

- In case of programs that deal with large data arrays, the error message `segmentation error` (or `Speicherzugriffsfehler`, in German) may occur. In that case, the user has to unlimit the size of stack, using the following command:

```
ulimit -s unlimited
```

If the user activates the parallelization options (§7.7) of MAGIX the size of the OMP stack has to be increased as well by writing the following command:

```
export OMP_STACKSIZE='999M'
```

Here the size of the stack depends on the memory that is available. If `OMP_STACKSIZE` is not set by the user, the default is `999M`.

[2]**Extended PID:** Something like the commonly used internal PID, but it is unique in that it is possible to distinguish all processes/threads that started at the same time – excluding all other processes – and at the same time each of the selected processes is distinguished from one another.

It was essential to find a way to distinguish the temporary directories of MAGIX instances that start at the same time, for cases that different MAGIX instances start on the same machine at the same time. This would be particularly useful in cases that you want to speed up your task and get a fast solution for a problem, by running different MAGIX instances for the same model and changing e.g. the starting value of some parameter (see example in §8.1).

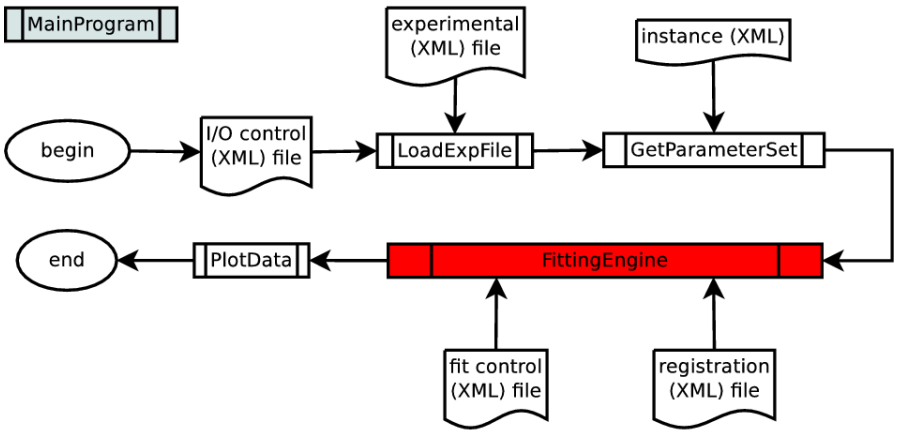


Figure 2: The flow chart of MAGIX. The flow chart of the `FittingEngine` module which actually performs the optimization process is shown in fig. 3.

- For the plot for each iteration daemon, the user can define a time interval by using the following command:

```
export MAGIXTimePlotIter="5000"
```

  For further informations see (§5.3).

1.7 Modules of MAGIX

MAGIX is basically written in Python, with some algorithm packages written in FORTRAN. `MainProgram` is the central module of MAGIX: it controls the whole process and calls all other modules, managing the communication between them. All necessary python and FORTRAN modules are located in the directory `Modules` within the root directory of MAGIX; do not rename, move or remove this directory and any of its contents.

The flow chart of MAGIX is shown in fig. 2. It consists of four basic modules:

- `LoadExpFile`: contains subroutines for loading/reading experimental data files, using appropriate filters depending on the format of the files (§3)
- `FittingEngine`: performs the fitting process using the algorithms specified in the fit control file (§8)
- `GetParameterSet`: reads the model parameters from the model instance, which defines model ranges, starting values and `fit` attributes (information about the `fit` attribute in §4)

It is envisioned that the final version of MAGIX will also include a heuristics module that will be able to choose the best algorithm or combination of algorithms, based on user–defined priorities.

1.8 Abort MAGIX

In order to abort a MAGIX process, press “Crtl”+“z” (or “Crtl”+“c”) followed by the command

```
kill %1
```

Due to the cancellation of the MAGIX process, MAGIX is not able to clean up the temporary directory (`temp`). As described in one of the next sections, MAGIX creates a temporary directory for each MAGIX process (job). The name of the so-called job directory is composed of two components: The first component is the phrase "(`job_`)" and the second component is a 8 digit integer number, called job ID, e.g. (`job_84376543`). In order to free memory on the hard disk, the user has to remove this job directory, manually.

2 I/O control file

MAGIX requires a so-called I/O control file which is located in the same directory where the file `magix_start.py` is. If you want to use an I/O control file located somewhere else, type both the path and the file name of the I/O control file after the `magix_start` command. For example:

```
./magix_start.py PathToYourFiles/io_control_test.xml
```

This command makes MAGIX run using the I/O control file `io_control_test.xml` located in directory `PathToYourFiles`.

The I/O control file contains the path and file name of the following files:

- `<experimental_data>`: experimental (XML) file (§ 3); this tag may not point to an XML file: it may point directly to a `.dat` or `.fits` file. In such a case MAGIX uses the default settings for importing all data contained in an ASCII or FITS file, respectively.
- `<parameter_file>`: instance XML (§4)
- `<fit_control>`: fit control XML file (§5)
- `<fit_log>`: log file of the fitting process (§6)
- `<model_description>`: registration XML file (§7)

Sample 1: Example of an I/O control file

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<ioControl>
  <title>io control</title>
  <description>paths and the file names used by MAGIX</description>
  <PathFilename>

    <experimental_data>
      <filename>test/TwoOscillators_RefFit_R.xml</filename>
      <description>path and file name of the experimental file</description>
    </experimental_data>

    <parameter_file>
      <filename>test/parameters.xml</filename>
      <description>path and file name of the instance file</description>
    </parameter_file>

    <fit_control>
      <filename>test/Levenberg-Marquardt_Parameters.xml</filename>
      <description>path and file name of the fit-control xml file</description>
    </fit_control>

    <fit_log>
      <filename>test/fit.log</filename>
      <description>path and file name of log file(s)</description>
    </fit_log>

    <model_description>
      <filename>test/model_registration.xml</filename>
      <description>path and file name of the registration file</description>
    </model_description>

  </PathFilename>
</ioControl>
```

- The four XML files (experimental file, fit control files, instance and registration) are used by MAGIX as input, therefore MAGIX will check their existence and will cancel if one of these files does not exist.
- The fifth (non-XML) file specified in the tag `<fit_log>` needs not exist but the existence of its path will be checked, since MAGIX will create three files in the given path. A non-existence of this path will cancel MAGIX. If `logs` is the name given under the `<filename>` tag and `alg` is the name of the algorithm chosen, then during the fitting process three log files will be created:
  - `logs_alg`: For every iteration step, this file contains a line with the running number of the step, the value of χ^2 at current step, and the current value of all parameters that are being optimized. (For the Levenberg-Marquardt (§ 8.1) as well as for the Simulated annealing algorithm (§8.2) additional informations are also printed.
  - `logs_alg.param`: The contents of all input files – with the new optimized values of the parameters to be fitted – of the external program are written in this file for every iteration step.
  - `logs_alg.chi2`: Similar to `logs`. For each function call of every iteration step, this file contains a line with the value of χ^2 and the value of all parameters that are being optimized.

  If a path precedes the string `logs` in the `<filename>` tag, then the three new files will be located in this path.

  The string `alg` is every time substituted with the name of the algorithm chosen; i.e. if the algorithm were PSO, then the three log files would be `logs_PSO`, `logs_PSO`, `logs_PSO`. In cases of algorithms that are used two or more times for a single MAGIX run, then the string of the algorithm's name will be followed by a serial number. For further details see (§6).
- Note, the tags `<title>` and `<description>` are optional and are not read by MAGIX.

3 Experimental data

Before the experimental files can be imported, the XML file containing all settings required for the import of the experimental data has to be loaded. More specifically, the experimental XML file defines the number of experimental files, it describes and gives the path and file name for each one of them.

3.1 General tags

All tags are necessary (except for `<MinExpRange>` and `<MaxExpRange>` when `<NumberExpRanges>` is set to 0, see §3.2).

- MAGIX is able to read files in a variety of experimental data formats. At the moment it can load experimental data stored in ASCII, or FITS format. The user can select the format of the experimental data file with the `<ImportFilter>` tag:
  - If the user sets the content of the tag `<ImportFilter>` to `automatic` then MAGIX chooses the correct import filter depending the ending of the file. For that setting to function properly, the files have to end with either one of `.dat` and `.fits` for ASCII and FITS files respectively.
  - The user can also specify the correct import filter, by setting the tag `<ImportFilter>` to `ASCII` or `FITS`.
- The names of the tags `<ExpFiles>`, `<NumberExpFiles>` etc. have to be written in the same way as presented in the example above (file sample 1).
- The tag `<file>` must occur as many times as the number of experimental files defined in the tag `<NumberExpFiles>`.
- Note, the order of the different experimental data files in the xml-file becomes important if the external model program creates for each function call more than one output file. MAGIX assumes that the first experimental data file is described by the first output file of the model program etc. The order of the output files is defined in the registration file (§7). Therefore, the user has to declare the different experimental data files in the correct order:

  For example, the external model program creates two output files: The first file contains the transmission as a function of frequency and the second file describes the velocity as a function of frequency. If the first experimental data file (1) includes the measured transmission as a function of frequency, and the second file (2) the corresponding data for the velocity, the user has to declare file (1) first.

3.2 Experimental data ranges

- The number of ranges `<NumberExpRanges>` must always be given! If no range is desired (i.e. all data contained in the file are to be included in the fitting process), then the number of ranges must be set to 0.
- If the number of ranges is set to 0, then the tags `<MinExpRange>` and `<MaxExpRange>` need not be given.
- If the user does not want to use all data (number of ranges > 0), then the tags `<MinExpRange>` and `<MaxExpRange>` have to occur as many times as defined by `<NumberExpRanges>`.

- Note that the XML description of experimental data files for ASCII, and FITS files differ by some tags.

3.3 X and Y columns

- The expression X column refers to an independent variable of the model. The X columns are the columns of an array defining the X position of the experimental data point.

Example A: For a function $f(x_1, x_2, x_3)$, the number of X columns is 3; for a function $f(x_1, x_2)$, the number of X columns is 2.

- The tag `<NumberColumnsX>` defines the number of columns (starting from the left-most column) that belong to the X points of each experimental data file.

Example B: If the user wants to import an ASCII file containing 3D data, then `<NumberColumnsX>` has to be set to 3. The first 3 columns will then define the X, Y and Z position.

- If the number of X columns is > 1, then the min and max of X columns of the ranges have to be separated by the comma (,) character.

Example C: That's for one X column:

```
<NumberColumnsX>1</NumberColumnsX>
<MinExpRange>0</MinExpRange>
<MaxExpRange>2000</MaxExpRange>
```

Example D: For three X columns:

```
<NumberColumnsX>3</NumberColumnsX>
<MinExpRange>0, 0, 0</MinExpRange>
<MaxExpRange>2000, 100, 20</MaxExpRange>
```

- The expression Y column refers to a dependent variable of the model. The `<NumberColumnsY>` is only relevant for ASCII files (§ 3.4). For a given ASCII file, there can be defined exactly one number of X columns and exactly one number of Y columns. This means that the independent variables have to be of equal number for all the dependent variables–functions.

Example E: Imagine an experimental file that includes the values of three independent variables, i.e. some 3D grid coordinates, x_1, x_2, x_3, of two functions, say the temperature $T(x_1, x_2, x_3)$ and the density $n(x_1, x_2, x_3)$. Then the number of X columns is 3 and the number of dependent variables – Y columns – is 2.

3.4 Experimental data from ASCII files

Sample 2: XML structure to import experimental data from ASCII files

```
<?xml version="1.0" encoding="UTF-8"?>
<ExpFiles>


    <!-- define number of experimental data files -->
    <NumberExpFiles>1</NumberExpFiles>


    <!-- define import settings for 1st exp. data file -->
    <file>
```

```
        <!-- define path and name of experimental data file -->
        <FileNamesExpFiles>examples/TwoOscillators_RefFit_R.dat</FileNamesExpFiles>


        <!-- define import filter -->
        <ImportFilter>ascii</ImportFilter>


        <!-- define number of header lines -->
        <NumberHeaderLines>0</NumberHeaderLines>


        <!-- define character, which separate columns -->
        <SeparatorColumns> </SeparatorColumns>


        <!-- define number of X- and Y-columns -->
        <NumberColumnsX>1</NumberColumnsX>
        <NumberColumnsY>1</NumberColumnsY>


        <!-- are errors included? -->
        <ErrorY>no</ErrorY>


        <!-- define number and limits of ranges -->
        <NumberExpRanges>1</NumberExpRanges>
        <MinExpRange>50</MinExpRange>
        <MaxExpRange>1000</MaxExpRange>
    </file>
</ExpFiles>
```

- The tag `<NumberHeaderLines>` defines the number of header lines that must be ignored at import of an ASCII file.
- The user can specify a separator character (the tag `<SeparatorColumns>` defines the character that separates the columns from each other) for each file.
- For ASCII files, it may be necessary to specify the number of Y columns. This means that for a given X position in the experimental data, you can specify several Y values.

Example A: You measured several spectra under different polarization angles at the same frequency. A line in the corresponding ASCII file may look like:

```
100.12,  0.34134,  0.12341,  0.78901,  0.13361
```

Here, the first column describes the frequency and the other columns describe the transmission at different polarization angles. The number of X columns is 1 and the number of Y columns is 4.

- The tag `<NumberColumnsY>` defines the number of columns that belong to the Y points of the experimental data. The Y columns have to be next to the X values!

Example B: If the user wants to import an ASCII file that contains values of four Y points at every given X point then the tag `<NumberColumnsY>` has to be set to 4.

- If the error tag `<ErrorY>` is set to `yes`, then the columns containing the errors have to be next to the Y columns. The number of these error columns have to be equal to the number of Y columns given in the tag `<NumberColumnsY>`.

Note, the error values are used for the calculation of the χ^2 value, see (§5.2).

Sample 3: Example of an ASCII file with 3 Y columns and the corresponding Y errors (`ErrorY="YES"`)

```
NumberColumnsX=2     NumberColumnsY=3       3 <ErrorY> columns
100.2313  20.6578   0.5846  40.1  1.4218   0.020  0.451  0.017
102.2463  21.7548   0.5947  60.3  1.5432   0.039  0.230  0.092
140.5671  21.9998   0.3450  93.0  1.6725   0.091  0.561  0.005
```

3.5 Experimental data from FITS files

Sample 4: XML structure to import experimental data from FITS files

```
<?xml version="1.0" encoding="UTF-8"?>
<ExpFiles>


    <!-- define number of experimental data files -->
    <NumberExpFiles>2</NumberExpFiles>


    <!-- define import settings for 1st exp. data file -->
    <file>


        <!-- define path and name of experimental data file -->
        <FileNamesExpFiles>one_parameter_free/File3.fits</FileNamesExpFiles>


        <!-- define import filter -->
        <ImportFilter>automatic</ImportFilter>


        <!-- define number of HDU -->
        <NumberHDU>0</NumberHDU>


        <!-- define number and limits of ranges -->
        <NumberExpRanges>1</NumberExpRanges>
        <MinExpRange>0</MinExpRange>
        <MaxExpRange>1000</MaxExpRange>
    </file>


    <!-- define import settings for 2nd exp. data file -->
    <file>


        <!-- define path and name of experimental data file -->
        <FileNamesExpFiles>one_parameter_free/File4.fits</FileNamesExpFiles>


        <!-- define import filter -->
        <ImportFilter>automatic</ImportFilter>


        <!-- define number of HDU -->
        <NumberHDU>0</NumberHDU>
```

```
        <!-- define number and limits of ranges -->
        <NumberExpRanges>2</NumberExpRanges>
        <MinExpRange>0</MinExpRange>
        <MaxExpRange>2000</MaxExpRange>
        <MinExpRange>3130</MinExpRange>
        <MaxExpRange>3200</MaxExpRange>
    </file>
</ExpFiles>
```

- For FITS files, the number of Y columns is always 1!
- Although the `<NumberColumnsX>` tag is ignored (if it exists) when importing a FITS file, the content of this tag is defined by the dimension of the FITS file. Thus, the ranges settings, namely the way that the beginning and ending of each range are specified, have to be given as in example D in §3.3, if the dimension of the FITS file is > 1.
- The user has to specify the Header Data Unit (HDU) that should be loaded for each FITS file. This tag is needed only for FITS files.
- MAGIX distinguishes between image and table HDUs.

4 Model instance

The model instance is where the values of all parameters are set. This file also specifies whether each parameter is one to be optimized. If yes, then the starting values, as well as the lower and upper limits are also provided. If the parameter is not one to be optimized, then the lower and upper limits are ignored.

In the rest of this section, the tags of the instance are described with reference to some sample files. More examples are given in §7, together with the corresponding registration.

4.1 Necessary tags in the instance

Sample 5: Example of a parameter XML file

```
<?xml version="1.0" encoding="UTF-8"?>
<ModelParameters>


    <!-- define total number of parameters -->
    <NumberParameters>8</NumberParameters>


    <!-- define parameter "EpsilonInfinity" -->
    <Parameter fit="false">
        <name>EpsilonInfinity</name>
        <value>2.5</value>
        <error> </error>
        <lowlimit>0</lowlimit>
        <uplimit>10</uplimit>
    </Parameter>


    <!-- define parameter "NumberOscillators" -->
    <Parameter fit="false">
        <name>NumberOscillators</name>
        <value>2</value>
        <error> </error>
        <lowlimit>0</lowlimit>
        <uplimit>100</uplimit>
    </Parameter>


    <!-- define 1st parameter "EigenFrequency" -->
    <Parameter fit="false">
        <name>EigenFrequency</name>
        <value>150.0</value>
        <error> </error>
        <lowlimit>0</lowlimit>
        <uplimit>1000</uplimit>
    </Parameter>


    <!-- define 1st parameter "PlasmaFrequency" -->
    <Parameter fit="true">
        <name>PlasmaFrequency</name>
        <value>200.0</value>
        <error> </error>
        <lowlimit>0</lowlimit>
        <uplimit>1000</uplimit>
```

```
    </Parameter>


    <!-- define 1st parameter "Damping" -->
    <Parameter fit="true">
        <name>Damping</name>
        <value>10.0</value>
        <error> </error>
        <lowlimit>0</lowlimit>
        <uplimit>1000</uplimit>
    </Parameter>


    <!-- define 2nd parameter "EigenFrequency" -->
    <Parameter fit="false">
        <name>EigenFrequency</name>
        <value>600.0</value>
        <error> </error>
        <lowlimit>0</lowlimit>
        <uplimit>1000</uplimit>
    </Parameter>


    <!-- define 2nd parameter "PlasmaFrequency" -->
    <Parameter fit="true">
        <name>PlasmaFrequency</name>
        <value>400.0</value>
        <error> </error>
        <lowlimit>0</lowlimit>
        <uplimit>1000</uplimit>
    </Parameter>


    <!-- define 2nd parameter "Damping" -->
    <Parameter fit="true">
        <name>Damping</name>
        <value>10.0</value>
        <error> </error>
        <lowlimit>0</lowlimit>
        <uplimit>1000</uplimit>
    </Parameter>
</ModelParameters>
```

- A parameter name should appear within the instance as many times as it appears in the registration file, i.e. preferably once. Exceptions for this are the following cases:
  - If a parameter belongs to a group, then its name should appear in the instance so many times as the number of replications in this group.
  - If a parameter belongs to a group, its name can also be given to another parameter that does not belong to this group.
  - If a parameter is declared more than once in the same input file (with double square brackets appended in their name) or in different input files (with no double square brackets appended), then it has to be declared only once in the instance, and that is the first time it appears in the registration file.
  - If a parameter is declared more than once in the same input file (with no double square brackets appended), then all occurrences of the name are considered to belong to different parameters and have to exist also in the instance.
- The number of the model parameters defined in this file (`<NumberParameters>`) must be

equal to the number of all the parameters defined in the registration file (for all files and all of their lines – no exact tag exists for the total number in the registration file, but it can be derived summing up the contents of the `NumberParameterLine` of all lines and all files), taking into account all existing replication of lines and parameters. (The total number of parameters is altered by the `group` attribute and line replications; see §7.9).

- Each parameter is described inside the `<Parameter>` tag. There have to occur as many `<Parameter>` tags as defined in the tag `<NumberParameters>` in the same parameter XML file.
- The names of the model parameters defined within the `<name>` tags must be identical with the corresponding names as defined in the registration file of the given model (§7). Otherwise the program stops. The only parameter names that may appear more than once in an instance are those who belong to a group, if the replication number of that group is > 1.
- In order to include a parameter in the fitting process, set the `fit` attribute of the `<Parameter>` tag to `true`.
  For example: In case you want to optimize the value of the parameter named as `EpsilonInfinity` during the fit process:

```
<Parameter fit="true">
            <name>EpsilonInfinity</name>
            <value>3.0</value>
            <error></error>
            <lowlimit>0</lowlimit>
            <uplimit>9999</uplimit>
</Parameter>
```

  If the value of this parameter should not be optimized, then you have to set the `fit` attribute to `False`:

```
<Parameter fit="false">
...
```

- The tags `<uplimit>` and `<lowlimit>` indicate the upper and the lower limits of the model parameters, respectively. If the value of the model parameter runs out of this defined range during the fit process, MAGIX will print out a warning message on the screen and corrects the value of the parameter to the closest value within the range.

  Note, the value of the tag `<uplimit>` has to be greater than the value of the tag `<lowlimit>`. Please avoid using non-number specifications like "±inf", because the range definitions are essential for the swarm algorithms like Bees or PSO. Setting a parameter limit to "±inf" leads to an dramatic enhancement of the computational effort.
- The `<error>` tag must occur for every parameter, even if empty. The content of this tag is replaced by the error of the optimized parameter in the end.

5 Fit control file

Some general directives about how to select the algorithm(s) to use are given in § 8.11. The module `FittingEngine` reads the parameters controlling the fitting process from the fit control file, with the use of the function `read_control_file`. The flow chart of this module is shown in fig. 3.

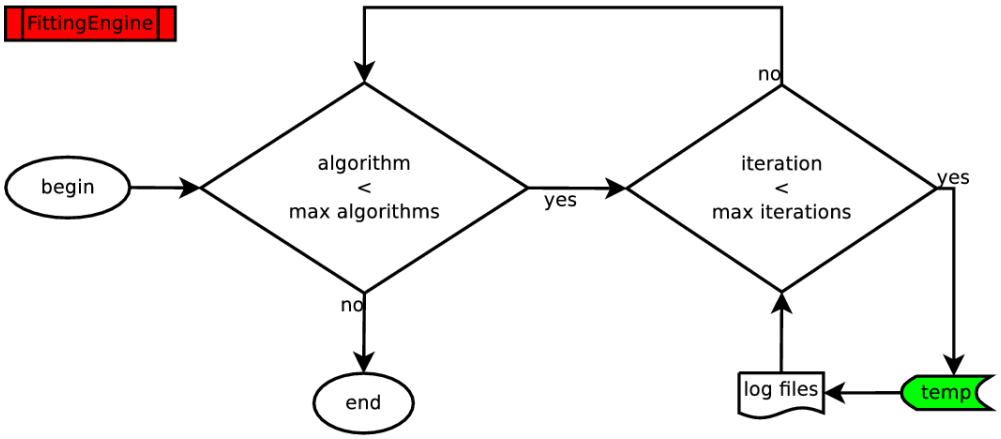


Figure 3: The flow chart of the `FittingEngine` module.

5.1 General information

Sample 6: Example of a fit control file (The expressions "`<!--`" and "`-->`" indicate a remark line in a xml-file.)

```
<?xml version="1.0" encoding="UTF-8"?>
<FitControl>
    <!-- settings for fit process -->


    <!-- set number of used algorithms -->
    <NumberOfFitAlgorithms>1</NumberOfFitAlgorithms>


    <algorithm>
        <!-- define algorithm -->
        <FitAlgorithm>bees</FitAlgorithm>


        <!-- special settings for bees algorithm -->
        <!-- BestSiteCounter (number of best sites) > 0 -->
        <BestSiteCounter>5</BestSiteCounter>


        <!-- set max. number of iterations -->
        <number_iterations>10</number_iterations>


        <!-- set max. number of processors -->
        <NumberProcessors>8</NumberProcessors>
```

```
        <!-- set path and name of host file -->
        <MPIHostFileName>hostfile.txt</MPIHostFileName>


        <!-- settings for chi^2 -->
        <limit_of_chi2>0.001</limit_of_chi2>
        <RenormalizedChi2>yes</RenormalizedChi2>
        <DeterminationChi2>default</DeterminationChi2>
        <SaveChi2>yes</SaveChi2>


        <!-- set plot options -->
        <PlotAxisX>Frequency [Hz]</PlotAxisX>
        <PlotAxisY>Intensity</PlotAxisY>
        <PlotIteration>no</PlotIteration>
    </algorithm>
</FitControl>
```

- The tag `<NumberOfFitAlgorithms>` defines the number of algorithms which should be used within the fit process. A number greater than 1 defines a so-called *algorithm chain* (see example 7).
- The settings for each algorithm are enclosed inside the `<algorithm>` tag. The tag has to occur as many times as specified by the tag `<NumberOfFitAlgorithms>`.
- Each algorithm is described by the tags
  `<FitAlgorithm>`, `<number_iterations>`, `<NumberProcessors>`,
  (`<MPIHostFileName>`),
  `<limit_of_chi2>`, `<DeterminationChi2>`, `<SaveChi2>`, `<RenormalizedChi2>`,
  `<PlotAxisX>`, `<PlotAxisY>`, (`<PlotAxisZ>`) and `<PlotIteration>`.
  Depending on the chosen algorithm, a couple of additional tags have to be added (§5.4).
- The tag `<FitAlgorithm>` defines the algorithm that is used in the fit process. The content of the `<FitAlgorithm>` tag has to be identical with one of the algorithm names (information on the available algorithms in section §8; it does not matter if these words are written in lower or upper case letters):
  - `levenberg-marquardt` (§8.1),
  - `simulated-annealing` (§8.2),
  - `nested-sampling` (§8.3),
  - `pso` (§8.4),
  - `bees` (§8.5),
  - `genetic` (§8.6),
  - `mcmc` (§8.7),
  - `interval-ns` (§8.8),
  - `errorestim_ins` (§8.9).
  - `additionalpackages` (§8.10),
- In order to deactivate the sorting of the chi2 log file the user can set the tag `<SortFortranFlag>` to `False`. If this tag is not defined, the chi2 log file is sorted in ascending order.
- The tag `<number_iterations>` sets the number of iterations for each algorithm and has to be an integer greater zero (N_I, integer > 0).
- The tag `<NumberProcessors>` defines the number of processors used by MAGIX.

  NOTE that a value > 1 can be used only for external model programs that allow parallelized work (see tag `<ParallelizationPossible>` of the registration XML file, §7.7).

- The tag <MPIHostFileName> defines the path and name of a so-called host file required for the MPI parallelized version of MAGIX, see § 5.1.1. The host file contains names of all of computers on which the MPI job will execute. For ease of execution, the user should be sure that all of these computers have SSH access, and that an authorized keys file is defined to avoid a password prompt for SSH. Additionally, the number of cores which should be used for an MPI run on each machine can be limited by using the slots command. For that purpose, the user has to extend the name of each machine by the definition of cores.

```
meslam slots=16
lugal slots=8
anu slots=2
```

  In the example described above, we use 16 cores on "meslam", 8 cores on "lugal" and two cores on "anu".

  NOTE, if the total number of processors defined in the host file is smaller than the number of processors defined by the tag <NumberProcessors> MAGIX will reduce this value.

  For some supercomputers the user does not need to specify a host file, because there is already a host file defined. In order to use a globally defined MPI host file, please insert the phrase MPI_HOSTS into the tag <MPIHostFileName>.

5.1.1 Different parallelization techniques used by MAGIX

MAGIX supports two different parallelization techniques. On the one hand MAGIX provides the algorithms in a SMP (Symmetric multiprocessing) parallelized version using OpenMP. A symmetric multiprocessor system where two or more identical processors are connected to a single, shared main memory, have full access to all I/O devices, and are controlled by a single operating system instance that treats all processors equally. Modern multiprocessors offers up to 32 different processor cores which can be used for a MAGIX run. This parallelization technique is very fast, because all processors use the same memory, but the number of threads is limited by the number of available cores on the current machine.

In contrast to the SMP parallelization, MPI (Message Passing Interface) is a standardized and portable message-passing system, where one or more computers are connected in a so-called cluster. MPI parallelization is somewhat slower than SMP, caused by the network, but it can be used with an (in principle) unlimited number of cores. In order to use the MPI parallelization, the user has to install the OpenMPI package on all computers in the cluster. Additionally, the user has to provide a temp directory which is visible by all computers. Please note, by using the environment variable MAGIXTempDirectory (see § 1.6.1), the user can define different paths for the temp directory on each machine in the cluster. Depending on the external model program, this makes the definition of a directory which is visible by all computers in the cluster dispensable.

5.2 Tags concerning χ^2

MAGIX is able to fit the values of parameters, providing confidence intervals presented by the value of χ^2.

- <DeterminationChi2>: Specifies the method that is used for the determination of χ^2. At the moment the following options are included in MAGIX:

- `default`:

$$\chi^2 = \sum_{i=1}^{N} \left(y_i^{\text{obs}} - y_i^{\text{fit}} \right)^2$$

where y_i^{obs} represents the value of the experimental data at point i, and y_i^{fit} the corresponding value of the fit function.
If the tag `<DeterminationChi2>` is set to `default` or `difference`, then the content of the tag `<SaveChi2>` is read (`yes`/`no`, default value `yes`), which specifies whether the difference $y_i^{\text{obs}} - y_i^{\text{fit}}$ is saved for all experimental points to a file (in the `.chi2` log file).
Note, if the experimental data file(s) includes error values then the χ^2 value is defined as follows:

$$\chi^2 = \sum_{i=1}^{N} \left[\left(y_i^{\text{obs}} - y_i^{\text{fit}} \right)^2 \cdot \frac{1}{\left(\sigma_i^{\text{error}} \right)^2} \right],$$

where σ_i^{error} represents the error of the ith data point.

- ...

- The tag `<limit_of_chi2>` specifies the value of χ^2 where the fitting process stops, i.e. if the value of χ^2 drops below this value the algorithm stops. The limit of χ^2 should be a real number > 0.

- `<RenormalizedChi2>` (`yes`/`no`, default value `yes`): specifies if MAGIX uses a re-normalized value for the limit of χ^2. If you set the flag to `yes` or `y`, then MAGIX determines the limit of χ^2 through the relation

$$\left(\chi^2_{\text{limit}} \right)_{\text{renom}} = \sum_{i=1}^{N_{\text{exp}}} \left(N_Y(i) \cdot N_{\text{points}}(i) - N_{\text{par}} \right) \cdot \left(\chi^2_{\text{limit}} \right)_{\text{orig}}$$

where N_{exp} is the number of observation files `Number_ExpFiles`; $N_Y(i)$ represents the number of Y columns of observation file i; $N_{\text{points}}(i)$ indicates the number of observation data points in the observation file i; N_{par} `NumberParameters` is the total number of all parameters; $\left(\chi^2_{\text{limit}} \right)_{\text{orig}}$ is the original unmodified value of χ^2.

5.3 Tags available only for 2D and 3D plots of 1D functions $y = f(x)$ and $y = f(x, y)$

- The tags `<PlotAxisX>` and `<PlotAxisY>` define the labels for the X and Y axis, respectively.

- The tag `<PlotAxisZ>` is used only for 3D plots.

- The observed data and the fit function are plotted for each iteration step, if the tag `<PlotIteration>` is set to `yes` (default value `no`). Please note, that this option starts a daemon, which refreshes the plot window every 3 sec. By setting the environment variable `MAGIXTimePlotIter`

```
export MAGIXTimePlotIter="5000"
```

the user can define another time interval (in milliseconds). If the time interval is to short, the window is always gray and no function is plotted.

Please take into account, that the creation of the plot window requires time as well.

5.4 Tags required only for certain algorithms

- The following tag is only relevant when the algorithm chosen is **NS**, **PSO**, **Genetic**, **MCMC**, **INS** or **Bees**:

  - `<BestSiteCounter>`: Defines the number of best sites. MAGIX writes the results (parameter set and value of the model function) of these sites to files. A number of best sites greater than one, is especially useful when the χ^2 function has multiple minima, i.e. there is more than one best fit (description) of the experimental data. Additionally, a number greater than one is useful when you want to use a so-called *algorithm chain*, and the current algorithm (§5.5) is not the last one in the chain.

    Imagine you use an algorithm chain (§ 5.5) of two algorithms: The first algorithm is the Bees, with the best site counter set to 2, so MAGIX will search and find the two best parameter sets. Then, if the next algorithm in the chain is the Levenberg-Marquardt, it will find the best fitting parameter sets, starting from the two sites found previously by the Bees. File sample 7 shows the fit control file of a similar scheme with the best site counter for Bees set to 3.

- The following tags are only relevant when the algorithm chosen is **Levenberg-Marquardt** (§8.1):

  - `<VariationValue>` (var, real positive number > 0, typical value 10^{-6}): For each iteration step the Levenberg-Marquardt algorithm has to determine the gradient of the χ^2 function. Due to the fact that MAGIX can not determine the components of the gradient analytically, MAGIX has to use a numerical approximation:

    $$\frac{\partial}{\partial x_i} f(\vec{x}) = \frac{f(x_i + h) - f(x_i)}{h},$$

    where the variation h is defined by

    $$h = x_i \cdot var.$$

    Varying the value of `<VariationValue>` could be very useful if the χ^2 function is not a smooth function and the calculation of the gradient produces awkward results.

- The following tags are only relevant when the algorithm chosen is **Simulated Annealing** (§8.2):

  - `<Temperature>` (T_0, real number > 0, typical value 1000): This tag defines the starting value for the global temperature, which is updated (decreased) at every step of the fit process.
  - `<TemperatureReductionKoeff>` (k, real number > 0 and < 1, typical value 0.8): Defines coefficient for the temperature reduction.
  - `<NumberOfReduction>` (N_R, integer number > 0, typical value 10): The value defines the number of temperature reductions. The number of reductions N_R does not need to be the same as the number of iterations N_I specified by the `number_iterations` tag. If $N_R < N_I$ and the first N_R reductions have been completed, then the global temperature is reset to T_0, but with the configuration being the one that resulted from the last reduction. The procedure continues until the total number of reductions completed is N_I.

- <NumberOfReheatingPhases> (N_{rH}, integer number > 0, typical value 3): The value defines the number of the reheating phases. The Simulated Annealing algorithm "heats" with the temperature T_0 which leads to a modification of the starting values. Then, T_0 is reduced by k for N_R iterations. If the algorithm is not able to find a better χ^2 value after N_R iterations, MAGIX "heats" again with temperature T_0. MAGIX will repeat this process N_{rH} times before it stops the algorithm.
- <ScheduleSA> This tag (only used for scipy version) defines the annealing schedule. Available ones are 'fast', 'cauchy', 'boltzmann'.

► The following tag is only relevant for the **Bees** algorithm (§8.5):

- <NumberBees>: This tag defines the number of the so-called bees, which should be used within the Bees algorithm. (The default setting is automatic.) The user can define a value which has to be larger than

$$N_{\text{Bess}} = N_{\text{site}} \cdot (5 + 11 \cdot N_{\text{site}}) \cdot N_{\text{free}},$$

where N_{site} indicates the number of best sites, see above, and N_{free} the number of free parameters. Otherwise, MAGIX determines the number of bees automatically. Note that a bigger number would lead to an increased computational effort, whereas a smaller number can produce a worse result. But this depends immensely on the model function used in the fitting process.

► The following tags are only relevant for the **Genetic** algorithm (§8.6):

- <NumberChromosomes>: This tag defines the number of the so-called chromosomes, which should be used within the Genetic algorithm. (The default setting is automatic.) The user defined value has to be larger than zero. Note that a bigger number would lead to an increased computational effort, whereas a smaller number can produce a worse result. But this depends immensely on the model function used in the fitting process.
- <UseNewRange>: This tag defines if the algorithm should determine new (shrinked) ranges for each free parameter (yes), or not (no). (The default setting is yes.)

► The following tags are only relevant when the algorithm chosen is **Nested Sampling** (§8.3):

- <NumberObjects>: This tag defines the number of the so-called objects, which should be used within the NS algorithm. A typical value is 100. Note that a bigger number would lead to an increased computational effort, whereas a smaller number can produce a worse result. But this depends immensely on the model function used in the fitting process.

► The following tags are only relevant when the algorithm chosen is **MCMC** (§8.7):

- <NumberMCMCSampler>: This tag defines the number of the so-called walkers, which should be used within the MCMC algorithm. A typical value is 100. Note that a bigger number would lead to an increased computational effort, whereas a smaller number can produce a worse result. But this depends immensely on the model function used in the fitting process.

  Following {8} there is no reason not to go large when it comes to walker number, until you hit performance issues. Although each step takes twice as much compute time if you double the number of walkers, it also returns to you twice as many

independent samples per autocorrelation time. So go large. In particular, we have found that-in almost all cases of low acceptance fraction-increasing the number of walkers improves the acceptance fraction. The one disadvantage of having large numbers of walkers is that the burn-in phase (from initial conditions to reasonable sampling) can be slow; burn-in time is a few autocorrelation times; the total run time for burn-in scales with the number of walkers. These considerations, all taken together, suggest using the smallest number of walkers for which the acceptance fraction during burn-in is good, or the number of samples you want out at the end (see below), whichever is greater. A more ambitious project would be to increase the number of walkers after burn-in; this requires thought beyond the scope of this document; it can be accomplished by burning in a set of small ensembles and then merging them into a big ensemble for the final run. One mistake many users of MCMC methods make is to take too many samples! If all you want your MCMC to do is produce one- or two-dimensional error bars on two or three parameters, then you only need dozens of independent samples. With ensemble sampling, you get this from a single snapshot or single time step, provided that you are using dozens of walkers (and we would recommend that you use hundreds in most applications). The key point is that you should run the sampler for a few (say 10) autocorrelation times. Once you have run that long, no matter how you initialized the walkers, the set of walkers you obtain at the end should be an independent set of samples from the distribution, of which you rarely need many. Another common mistake, of course, is to run the sampler for too few steps. You can identify that you haven't run for enough steps in a couple of ways: If you plot the parameter values in the ensemble as a function of step number, you will see large-scale variations over the full run length if you have gone less than an autocorrelation time. You will also see that if you try to measure the autocorrelation time (with, say, acor), it will give you a time that is always a significant fraction of your run time; it is only when the correlation time is much shorter (say by a factor of 10) than your run time that you are sure to have run long enough. The danger of both of these methods-an unavoidable danger at present-is that you can have a huge dynamic range in contributions to the autocorrelation time; you might think it is 30 when in fact it is 30 000, but you don't "see" the 30 000 in a run that is only 300 steps long. There is not much you can do about this; it is generic when the posterior is multi-modal: The autocorrelation time within each mode can be short but the mode-mode migration time can be long. See above on low acceptance ratio; in general when your acceptance ratio gets low your autocorrelation time is very, very long.

- `<NumberBurnInIter>`: This tag defines the number of iterations (default 50) used for the so-called *burn-in* phase, see (§8.7).
- `<BackendFileName>`: The tag defines the path and name of a so-called backend file used to store and resume an interrupted MCMC run. If the given file does not exists before the MCMC run starts, XCLASS stores all MCMC parameters into a HDF5 file. Therefore, the h5py python package has to be installed. In order to resume an interrupted MCMC run, the path and name of the corresponding HDF5 file has to be defined by this tag.

► The following tags are only relevant when the algorithm chosen is **Additional packages** (§8.10):

- `<minAlgorithm>`: This tag defines the name of the `scipy` algorithm which should be used. The following algorithms are available:

1. “fmin”,
2. “fmin_powell”,
3. “fmin_cg”,
4. “fmin_bfgs”,
5. “brute”.

For further documentation see documentation of `scipy` package.

Note that the tag `<minAlgorithm>` has to include one of the above listed names of algorithms.

- The following tags are only relevant when the algorithm chosen is **Interval-Nested-Sampling** (§8.8):
  - `<vol_bound>`: This tag indicates the critical element of the volume. If the tag is empty, then MAGIX determines the value using the following expression:

$$\text{vol\_bound} = 0.1 \cdot \left(1.0 - \sqrt{\frac{(N_{\text{free}} - 0.75)}{N_{\text{free}}}}\right),$$

    where N_{free} indicates the number of free parameters.
  - `<delta_incl>`: This tag defines the difference between maximal and minimal value of inclusion function. (The default setting is `0.001`.)
- The following tags are only relevant when the algorithm chosen is the **Error estimation** (§8.10):
  - `<ErrorMethod>`: This tag defines the method (“MCMC” (default), “INS”, “Fisher’) which is used for error estimation (§8.9).
  - `<NumberMCMCSampler>`: This tag (relevant only for the MCMC method) describes the number of samples / walkers (default $2N$, where N indicates the number of free parameters) which are used by the MCMC algorithm, see description of tags used by MCMC algorithm above.
  - `<NumberBurnInIter>`: This tag (relevant only for the MCMC method) defines the number of iterations (default 50) used for the so-called *burn-in* phase, see (§8.7).
  - `<UsePrevResults>`: This tag (relevant only for the MCMC method) indicates, if parameter vectors calculated by other algorithms in the algorithm-chain, are used for the burn-in phase (`True`) or not (`False`, default). Using previous calculated parameter vectors reduces the computational effort, but the parameters are not calculated at the position which where generated by the MCMC algorithm. So, the underlying probability distribution might be not well sampled.
  - `<MultiplicitySigma>`: This tag (relevant only for the MCMC method) defines the multiplicity (default 2) of the standard deviation σ, which defines the error bounds of the free parameters. For example, by setting the multiplicity to 2 the error estimation algorithm computes the 2σ errors for the free parameters.
  - `<VariationValue>`: This tag (relevant only for the Fisher method) specifies the variation value (default 10^{-3}, see description of the `<VariationValue>` tag for the Levenberg-Marquardt algorithm described above) used for the computation of the covariance matrix, see (§8.9.1).

5.5 Optimization through an algorithm chain

It is possible to send the results of the optimization process performed by a certain algorithm, to another optimization procedure through some other algorithm. Some directives about how to select the order of the algorithms to use are given in §8.11.

In file 7, the fitting process starts with the Bees algorithm. Thereafter, the Levenberg-Marquardt algorithm is applied to the best three sites found previously by the Bees algorithm.

Sample 7: Example of a fit control file with an algorithm chain

```
<?xml version="1.0" encoding="UTF-8"?>
<FitControl>
    <!-- settings for fit process -->


    <!-- set number of used algorithms -->
    <NumberOfFitAlgorithms>2</NumberOfFitAlgorithms>


    <algorithm>
        <!-- define algorithm -->
        <FitAlgorithm>bees</FitAlgorithm>


        <!-- special settings for bees algorithm -->
        <!-- BestSiteCounter (number of best sites) > 0 -->
        <BestSiteCounter>3</BestSiteCounter>


        <!-- set max. number of iterations -->
        <number_iterations>30</number_iterations>


        <!-- set max. number of processors -->
        <NumberProcessors>8</NumberProcessors>


        <!-- set path and name of host file -->
        <MPIHostFileName>hostfile.txt</MPIHostFileName>


        <!-- settings for chi^2 -->
        <limit_of_chi2>0.001</limit_of_chi2>
        <RenormalizedChi2>yes</RenormalizedChi2>
        <DeterminationChi2>default</DeterminationChi2>
        <SaveChi2>yes</SaveChi2>


        <!-- set plot options -->
        <PlotAxisX>Frequency [Hz]</PlotAxisX>
        <PlotAxisY>Intensity</PlotAxisY>
        <PlotIteration>yes</PlotIteration>
    </algorithm>

    <algorithm>
        <!-- define algorithm -->
        <FitAlgorithm>Levenberg-Marquardt</FitAlgorithm>


        <!-- set max. number of iterations -->
```

```
        <number_iterations>20</number_iterations>


        <!-- set max. number of processors -->
        <NumberProcessors>8</NumberProcessors>


        <!-- set path and name of host file -->
        <MPIHostFileName>hostfile.txt</MPIHostFileName>


        <!-- settings for chi^2 -->
        <limit_of_chi2>0.0008</limit_of_chi2>
        <RenormalizedChi2>yes</RenormalizedChi2>
        <DeterminationChi2>default</DeterminationChi2>
        <SaveChi2>yes</SaveChi2>


        <!-- set plot options -->
        <PlotAxisX>Frequency [Hz]</PlotAxisX>
        <PlotAxisY>Intensity</PlotAxisY>
        <PlotIteration>yes</PlotIteration>
    </algorithm>
</FitControl>
```

6 MAGIX Output files

A series of files are created during a run of MAGIX:

6.1 Log files

MAGIX creates three different log files (see above, § 2) for each application of an algorithm referred to in the fit control file (see §5):

- a "normal" log-file with ending ".log", which corresponds to the screen output. The log file contains for every iteration step the best χ^2 value and the corresponding values of the parameters that are being optimized.
- a file with ending ".log.param" including for every iteration step the best χ^2 value and the corresponding values of the parameters that are being optimized as well as all input files for the external model program. This allows the user to verify that MAGIX writes the parameters at the right positions.
- a file with ending ".log.chi2" including all χ^2 values and the corresponding free parameter values for all calls of the external model program starting with the smallest χ^2 value.

Note, if the user specifies only the path for the log-files, MAGIX creates the files "fit.log", "fit.log.param", and "fit.log.chi2". Otherwise MAGIX creates a log file with filename specified in the I/O control file and extends the filename within ".log.param" and ".log.chi2". For example the user specifies `PathToYourFiles/mylogfile` in the I/O control file. Then MAGIX creates three different log-files:

- "`PathToYourFiles/mylogfile.log`",
- "`PathToYourFiles/mylogfile.log.param`", and
- "`PathToYourFiles/mylogfile.log.chi^2`".

The names of the log-files become more complicate, if the user applies a algorithm chain:

MAGIX extends the name of the log-files by an abbreviation for the algorithm (e.g. 'LM" for Levenberg-Marquardt) and by "__call_" followed by the number of the call of the algorithm.
For example, the Levenberg-Marquardt algorithm is applied to the three best sites of a previous used Bees algorithm, then the names of the log-files ("fit.log" for simplification) are "fit_LM__call_1.log", "fit_LM__call_2.log", "fit_LM__call_3.log",
"fit_LM__call_1.log.param", "fit_LM__call_2.log.param", "fit_LM__call_3.log.param",
"fit_LM__call_1.log.chi2", "fit_LM__call_2.log.chi2", "fit_LM__call_3.log.chi2".

6.2 Files for fit function comparison and χ^2

For each experimental data file MAGIX creates two (three) additional output files in the directory where the experimental data files are located:

(a) **File of optimized parameter values:** After finishing a fit algorithm MAGIX writes all parameter values and the corresponding error values (if calculated) for the best fit to a file which has the same names as the instance file. Additionally, MAGIX extends the ending of the filename with an abbreviation for the algorithm (e.g. "LM" for Levenberg-Marquardt) followed by the phrase ".out.xml".

For example, the instance xml-file is named “parameter.xml”. MAGIX writes the optimized parameter values and the corresponding error values (if calculated) to the file “parameter.LM.out.xml”.

(b) **File of fit function values:** Additionally, MAGIX writes the values of the model function for each data point of the best fit to files which have the same names as the experimental data files. But, MAGIX extends the ending of the filenames with an abbreviation for the algorithm (e.g. “LM” for Levenberg-Marquardt) followed by the phrase “.out.dat” for ASCII files and “.out.fits” for fits files.

For example, the name of the experimental data file is “datafile.dat”. MAGIX writes the values of the model function to the file “datafile.LM.out.dat”.

(c) **File with χ^2 values:** If the user sets the value of the tag `<SaveChi2>` to “`yes`”, MAGIX writes the values of χ^2 for each data point of the best fit to further files. These files have the same names as the experimental data files except that MAGIX extends the ending of the filenames with an abbreviation for the algorithm (e.g. “LM” for Levenberg-Marquardt) followed by the phrase “.out.chi2.dat” for ASCII files and “.out.chi2.fits” for FITS files.

For example, the name of the experimental data file is “datafile.dat”. MAGIX writes the values of χ^2 for each data point to the file “datafile.LM.out.chi2.dat”.

(d) **Error Estimation:** If the user selects the error estimation algorithm MAGIX produces four additional files for each experimental data file. Two of these files contain the model function for each data point where each free parameter is reduced (enhanced) by the corresponding lower (upper) error value. The other two files contain the corresponding χ^2 functions. The filenames corresponding to the reduced (lower) parameter values contain the phrase “LowerErrorValues” the enhanced contain “UpperErrorValues”.

For example, the name of the experimental data file is “datafile.dat”. MAGIX writes the values for the reduced parameters to “datafile.ErrorEstim_INS__LowerErrorValues__call_1.out.dat” and to “datafile.ErrorEstim_INS__LowerErrorValues__call_1.out.chi2.dat” whereas the enhanced parameters are written to “datafile.ErrorEstim_INS__UpperErrorValues__call_1.out.dat” and to “datafile.ErrorEstim_INS__UpperErrorValues__call_1.out.chi2.dat”.

Additionally, MAGIX determines the χ^2 distribution for each free parameter. In order to determine the error of a parameter j at the minimum, MAGIX varies this parameter within the given parameter range, whereas the other parameters are kept constant. MAGIX plots the χ^2 values as a function of the free parameter j and and saves the plot to a file named “ErrorEstim_INS__chi2-distribution_of_free-parameter_parm_” followed by an integer number indicating the free parameter. Additionally, the plotted χ^2 values together with the corresponding parameter values are stored to an ASCII file having the same name as the corresponding png file. For example, the file “ErrorEstim_INS__chi2-distribution_of_free-parameter_parm_1.png” contains the plot of the χ^2 distribution of the first free parameter and the ASCII file “ErrorEstim_INS__chi2-distribution_of_free-parameter_parm_1.dat” contains the data points plotted in the png file.

Furthermore, MAGIX adds further informations to the plot: The value of the parameter which corresponds to the best fit result is indicated by a solid vertical thick black line. The error range determined by the Error Estimation algorithm is marked with two dashed vertical red lines, see Fig. 4.

The names of the output files produced by MAGIX become more complicate, if an algorithm chain is used or if the number of best sites is set to a value greater one:

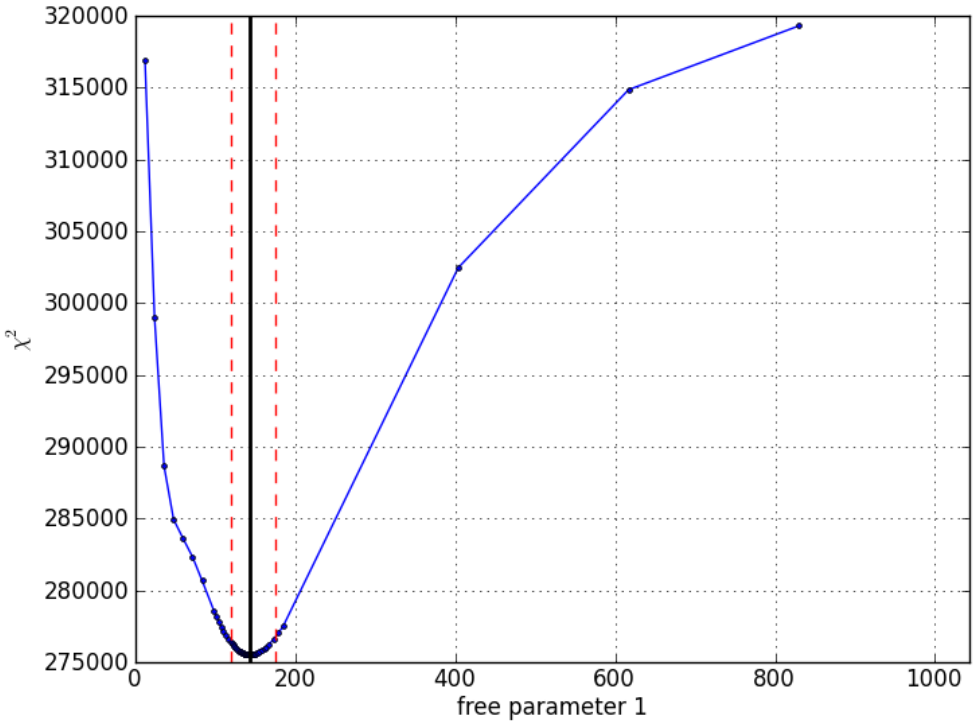


Figure 4: Example of a χ^2 distribution plot for the seventh free parameter. The solid vertical thick black line indicates the optimized parameter value and the two dashed vertical red lines describe the optimized parameter value reduced by the left and increased by the corresponding right errors, respectively.

In addition to the extensions of the file names described above, MAGIX adds in case of an algorithm chain the phrase "__call_" followed by the number of the call of the algorithm as well.

For example, the Levenberg-Marquardt algorithm is applied to the three best sites of a previous used Bees algorithm, then the names of the instance xml-files are named "parameters.LM__call_1.out.xml", "parameters.LM__call_2.out.xml", and "parameters.LM__call_3.out.xml".

In order to distinguish between different "best" sites, MAGIX adds in addition to the extensions described above the phrase "__site_" followed by the number of the best site as well. For example, the user applies the Bees algorithm and sets the number of best sites to 3. The names of the instance xml-files are named "parameters.Bees__call_1__site_1.out.xml", "parameters.Bees__call_1__site_2.out.xml", and "parameters.Bees__call_1__site_3.out.xml".

In cases of special FITS files that are images, the X column is declared in the header through the declarations of the first reference pixel (`CRPIX1`), its value (`CRVAL1`), and the distance between two pixels (`CDELT1`). If the user specified more than one range for the import of experimental data, then we have to additionally specify which one of those ranges is referred to in which one of the aforementioned output files (a+b).

Imagine that the user has specified more than one data range for the experimental FITS file `PathToYourFiles/datafile.fits`. If the beginning and end of range `R` are `begR` and `endR` respectively, then for this experimental file and each application (serial number `N`) of each algorithm (`alg`), MAGIX creates two files for each range `R`:

(a) **File of fit function values:** `PathToYourFiles/datafile_alg_N.out_begR_endR.fits`

(b) **File with χ^2 values:** `PathToYourFiles/datafile_alg_N.out_begR_endR.chi2.fits`

It must be understood that the string strings `begR` and `endR` are substituted by the exact numbers that give the beginning and end of ranges.

6.3 Plots

If the user do not select the `--noplot` option, MAGIX creates a plot containing the experimental data, the model function values and the χ^2 values for each data point. The plot is divided into two parts: The left side contains plots for each experimental data file where the observation data are plotted together with the model function for the best fit result. The right side contains plots for the corresponding χ^2 values for each data point, see Fig. 5. Finally, the plot is saved to a file where the name contains the phrase "final_plot", but become more complicate, if the user applies an algorithm chain or sets the number of best sites to a value greater 1, see the general description of the output file names above. For example, the user applies the Bees algorithm and sets the number of best sites to 3. The names of the plot files are named "final_plot.Bees__site_1.out.xml", "final_plot.Bees__site_2.out.xml", and "final_plot.Bees__site_3.out.xml".

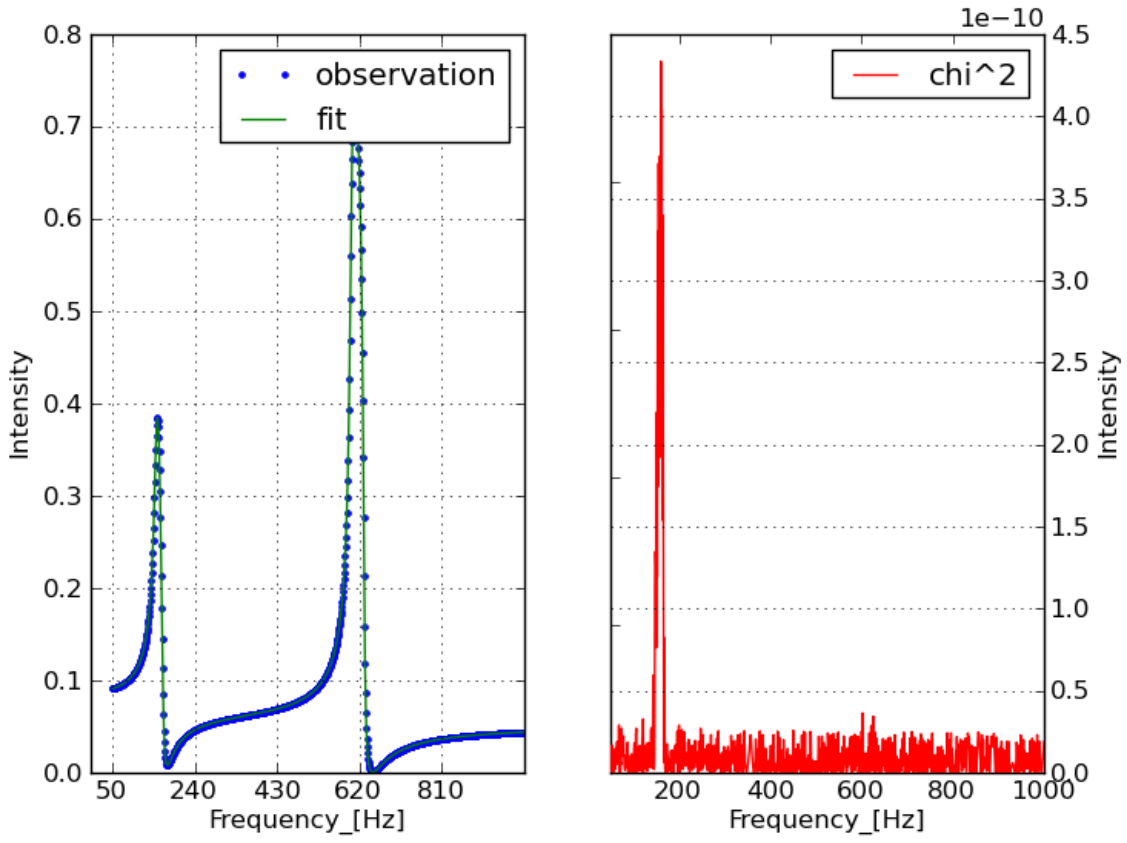


Figure 5: The final plot for the Levenberg-Marquardt example.

7 Model registration

7.1 Scientific rationale

MAGIX must communicate with the external program:

- Before each function call, MAGIX has to provide the starting values of the parameters to the external model program, so that it runs. During one optimization step the values of the parameters that are to be fitted have been modified. So at each next function call, MAGIX has to provide input files which are modified in comparison to the input files provided to the external model program at the previous function call.

  MAGIX knows the new values of the parameters specified in the input files, but does not yet have the actual input files that contain the new parameter values – since the external program can only receive the parameter values (and run) if they are given in the form of the input files that it expects to read.

  Therefore MAGIX has to be given directives about how to create/write the input files that will be used in the next function call.
- After each optimization step, MAGIX compares the experimental data with the value of the dependent variables calculated by the external program with the use of the latest values of the parameters that are being optimized. The comparison between the two takes place through the calculation of χ^2, so with one value per optimization step we can visualize the actual optimizing procedure.

  The new values of the model function have to be read from the output file of the previous function call. After MAGIX has done its job, it has to print the input file for the external program; this file will contain the best fitting values of the model parameters.

  Therefore MAGIX needs also descriptions of the output files of the external model program.

Ideally, the registration of a model should occur only once, i.e. we don't have to create a new registration file for a model more than once or every time that we want to run MAGIX for this model. Whenever one wants to optimize some parameter(s) of the model with MAGIX, it should be enough that one edits the instance.

The descriptions of both the input and the output files are given in XML format in the XML files that we call registration files. The following example helps in making important remarks about the tags of the registration XML files.

Additionally, the script starting the external model program plays an important role. This so-called "*start-script*" has to prepare all file(s) except the input file(s) to start the external model program. For example, the external model program requires some environment variables and some additional files (from a database), the start-script has to copy the files to the designated directory, sets the environment and finally, starts the external model program.

7.2 Organization of MAGIX

Before we start with the description of the registration process, it is important to understand how MAGIX is organized. This is can be roughly made clear by presenting the directory tree, i.e. how and where the temporary directories of MAGIX are created and used, which files are considered temporary, from and to where the temporary files are copied. When we understand this, we will understand where MAGIX is executed and where the partial instances of the external model program are executed, making it more obvious why and how a mistake or system crash might emerge.

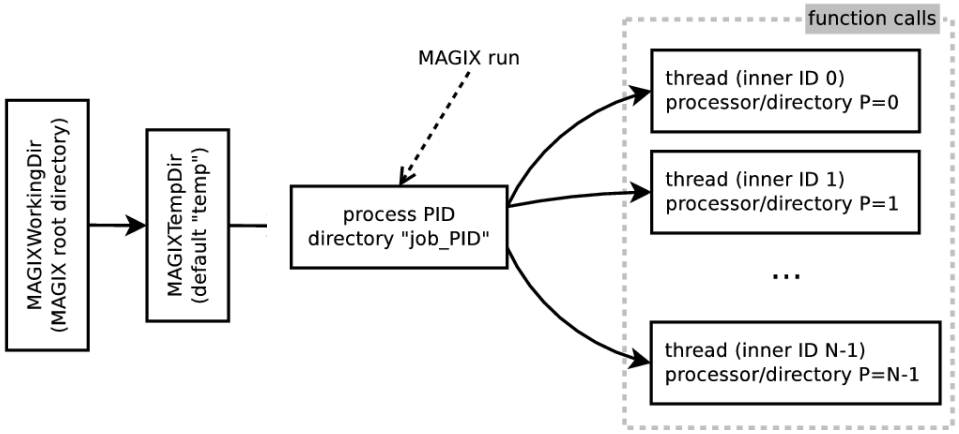


Figure 6: The tree below the MAGIX root directory, together with the N temporary directories created in one MAGIX session.

As MAGIX is a piece of software developed for the optimization of the values of the parameters of a model, it is obvious that the algorithms available to perform the optimization process will work in an iterative way, performing recurrent function calls to the external model program. The results of each function call are compared between each other in order to check which set of parameters best fits our experimental data.

These recurrent calls to the external model program are more quickly performed when executed in parallel. (The gain in computational time is explained in §7.7.) Therefore MAGIX is developed as a parallelized software. This does have some further logical requirements from the user and the programs whose parameters s/he wants to optimize with MAGIX (§7.7). But the external model program does not need to be itself parallelized (but if it is, there is no problem).

The tree from the MAGIX root directory and below is shown in fig. 6. Each processor has its own thread (its own process identification number) and its own running directory.

7.3 Start script

A very important information that is given in the registration file is the path and the name of the start script, which will be described in the following:

The way MAGIX is organized is largely made clear through the start script, the path and name of which are given in the registration file. This is the script that includes the command that actually executes the external model program, all the commands that have to run before the external program runs, as well as all the commands that have to follow after its execution is completed.

In order to debug the start script use the `--debug` flag at the MAGIX start (§1.4).

The start script has to be an executable script, without any constraints on the scripting language (e.g. shell, python etc.). The first line of the start script has to contain a line starting with `#!` (no preceding spaces), which will tell your system that this file is a set of commands to be fed to the command interpreter indicated and where this interpreter is located. For example for a bash shell script, the first line of the start script has to be:

```
#!/bin/bash
```

We forget for a moment this first line that specifies how the script has to be interpreted.

It is absolutely necessary for the external model program that the start script and the input/output file(s) have to be allowed to be copied to arbitrary directories: The external model program has to allow its start script be moved to an arbitrary directory, and the input and the output files have to be read from and written in the directory where the start script is located. This arbitrary directory is the temporary directory that is created for each thread, see fig. 6.

7.3.1 Simple start scripts

The simplest start script contains only the command with which the model program is executed. Imagine that we want to optimize the parameters of a program called `mystery`, whose running command is simply `./mystery_machine`. The following notes all refer to such cases, where only the external program's execution command is included in a bash shell start script, which therefore is a file of only two lines (the interpreter specification and the execution command).

- If the executable is located in MAGIX root directory, and all the input and output files are / will be located in the directory where the executable runs, then the (shell) start script will contain only one line:

  ```
  ./mystery_machine
  ```

  If there is any preparation to be done before the program is executed, the corresponding commands will go before this line. If there is any post-processing to be done after the program is executed, the commands have to go below this line.

- After you have created the start script (and all the XML files needed by MAGIX), you have to define it in the preamble of the registration file (§ 7.6). Then you can run MAGIX with the command

  ```
  ./magix_start.py PathToYourFiles/io_control_test.xml
  ```

  where `PathToYourFiles/io_control_test.xml` is the path and file name of the I/O control file (§ 2), where the registration file is specified. The registration file specifies the start script within the `<PathStartScript>` and the `<ExeCommandStartScript>` tags. In the simplest case the start script is a file that contains only two lines

  ```
  #!/bin/bash
  ./mystery_machine
  ```

- When you run MAGIX (giving the command at the root directory of MAGIX – that is where MAGIX runs), the external model program is not executed in the root directory of MAGIX. Instead, MAGIX copies the start script located in the directory, which is given by the tag `<PathStartScript>`, to the directory of the current thread (the thread directories are visualized on the right side of fig. 6).

  Note, the tag `<PathStartScript>` have to include not only the path but also the name of the start script.

  For example, MAGIX copies the start script `start_mystery.sh` located in the directory `PathToExample` to the temp directory "temp/job_12345678/0/". (Here, we assume that the environment variable `MAGIXTempDirectory` is set to "`temp/`"). The corresponding tags `<PathStartScript>` and `<ExeCommandStartScript>` in the registration file have to be set to

  ```
  <PathStartScript>PathToExample/start_mystery.sh</PathStartScript>
  <ExeCommandStartScript>./start_mystery.sh</ExeCommandStartScript>
  ```

After copying the start script to the current thread directory MAGIX executes the start script within the current thread directory using the command, which is defined by the tag <ExeCommandStartScript>. Therefore, the command to start the start script must not include an absolute path definition.

- If you want to save the screen output of the program's execution to some file, you have to redirect the screen output to some arbitrary file by setting the value of the tag <ExeCommandStartScript> to

```
./start_mystery.sh > screen_output.txt
```

- Additionally, MAGIX writes all input file(s) to the current thread directory. If the external model program expects the input file(s) in a special subdirectory (e.g. "Eingabe-Dateien/"), then the start script has to create this subdirectory within the thread directory and copies the input file(s) to this subdirectory. But this will be described in more detail in the following subsection.

7.3.2 Start scripts that include pre- and post-processing

Imagine that the user scoobydoo wants to optimize the parameters of the same program referred to in the previous section, mystery. Below we will show step by step what kind of commands the start script start_mystery_example.sh located in the directory PathToExample must/may contain. In the text I refer to shell-like commands, while the sample start script 8 is a python script doing more or less the same staff. The corresponding tags <PathStartScript> and <ExeCommandStartScript> in the registration file have to be set to

```
<PathStartScript>PathToExample/start_mystery_example.sh</PathStartScript>
<ExeCommandStartScript>./start_mystery_example.sh</ExeCommandStartScript>
```

- You can print user name, date and time of MAGIX run.

```
echo "user = `whoami` -- time: `date`"
```

- If the program expects to find the input file "input.txt" in a directory called Eingabe-Dateien/ which is located in the directory where the program is executed, then the corresponding commands have to be included in the start script start_mystery_example.sh, before the command that executes the program:

```
#!/bin/bash
mkdir Eingabe-Dateien
cp input.txt Eingabe-Dateien/
./mystery_machine
```

- If the program expects that there is a directory called Ausgabe-Dateien, where all the output files will be stored in, then the command to create this directory has to be also added before the command that executes the program. Otherwise, the program will crash, since it will not be able to find the directory where the output files are expected to be saved. So, the example start script has to be extended in the following way:

```
#!/bin/bash
mkdir Eingabe-Dateien
mkdir Ausgabe-Dateien
cp input.txt Eingabe-Dateien/
./mystery_machine
```

- If you want to make sure that the external program is executable by everybody, you can type the UNIX command

```
chmod 555 mystery_machine
```

  before you execute the program.

- You can combine several commands in the execution command defined by the tag `<ExeCommandStartScript>` using the ";" character:

```
chmod 555 start_mystery_example; sh start_mystery_example.sh
```

The above are possible commands that you may want to execute before the actual execution of the program by adding the line

```
./mystery_machine
```

Afterwards you might want to add some final commands, by the end of the start script:

- If some of the output files are very big, then you may want to compress them

```
gzip -9 Output-Files/filesA*.txt
```

  and/or remove them:[3]

```
rm -f Output-Files/filesA*.txt
```

- If you want to perform some post processing after the external program has run, you will type the necessary commands within the start script after the command that executes the program. For example, the following bash shell segment reads one of the output files that contains energy levels and subtracts the energy value of the first line, which is the base energy.

```
counter=0
while read EnergyLine
do
  let "counter = $counter + 1"
  if [ $counter -gt 3 ]; then
    if [[ $EnergyLine =~ "------------------------------" ]]; then
      break
    else
      line=$(echo $EnergyLine | awk '{print $NF}')
      if [ $counter -eq 4 ]; then
        GSEnergy=line
      fi
      echo `expr $line-$GSEnergy` >> ExcitationEnergies.dat
    fi
  fi
done < Output-Files/energy.txt
```

During the optimization process, MAGIX performs a number of calls to the external model program, in the end of which the parameters have been somewhat optimized. The modified parameters will be written into a new model instance, which will be used as the input instance at the next function call.

[3] Directives for the start script: In the case that you want to add some removal commands, you have to add the force option (`-f`), otherwise MAGIX will stop at the end of every function call and waiting for the user to confirm or deny deletion. In the same logic you have to make sure that certain aliases in your `.bashrc` (or `.bash_aliases` or similar scripts that depend on your OS/shell and presume aliases to commands) do not require confirmation for commands that are used in the start script.

Note, that MAGIX removes all files/subdirectories in the current thread directory after reading in the output file(s). Therefore, the start script has to create the several subdirectories for every function call.

Sample 8: A complete (python) start script. This start script performs quite the same operations as the ones described incrementally in § 7.3.2. Additionally, it copies the executable and some additional system files from directory `Fit-Functions/mystery/bin`, which is located inside the root directory of MAGIX. The executable and the system files will be copied to the thread directory of each function call during the optimization process.

```
#!/usr/bin/python
## example start script in python

## load python packages
import sys    ## load sys package
import os     ## load os package
import string ## load string package
import numpy  ## load numpy

## unlimit UNIX stacksize (different to the OMP_STACKSIZE needed for OpenMP)
os.system("ulimit -s unlimited")

## create output directory within local temp-directory
os.system("mkdir Output-Files/")

## create input directory within local temp-directory
os.system("mkdir Input-Files/")

## copy SystemFiles directory to local temp-file
os.system("cp -r Fit-Functions/mystery/bin/SystemFiles/ .")

## copy mystery execution file
os.system("cp Fit-Functions/mystery/bin/mystery_machine .")

## move input file to input directory
os.system("mv eingabe.txt Input-Files/")

## make the file executable for everybody
os.system("chmod 555 mystery_machine")

## start mystery_machine
os.system("./mystery_machine > screen_output.txt")

## zip basis_angular-momentum file
os.system("gzip Output-Files/basis_angular-momentum*.txt")

## copy energies-file to local temp-directory and create final output file
os.system("cp Output-Files/energy.txt .")

## import energy file
f = open("energy.txt")
contens = f.readlines()

## construct final output array
OutputFile = []
counter = 0
  for EnergyLine in contens:
    counter += 1
      if (counter > 3):
        if (EnergyLine.strip() == "------------------------------"):
```

```
          ## exit loop if end of energies is reached
          break
        else:
          line = EnergyLine.split()
              if (counter == 4):
                  GSEnergy = float(line[1])            ## save ground state energy
          CurrentEnergy = float(line[1])
          OutputFile.append(CurrentEnergy - GSEnergy)  ## create excitation energy

f.close()

## save output array to file
numpy.savetxt('ExcitationEnergies.dat', OutputFile)    ## use exponential notation
```

7.4 Input files of constant content or whose parameters are to be optimized

The registration file describes the format of the input and output files of the external model program. As explained in the previous section, in the end of each iteration step the new optimized values of the parameters will be written in the new instance, which will be used by MAGIX in the next iteration step so that the new ASCII input file is created and given to the external model program to run. All this process takes place in the thread directory, i.e. any thread directory contains an initial instance with the starting values and a new instance with the optimized values. So all input and output files described in the registration file have to be referred to in the program with their relative paths.

When an input file is not included in the registration file, this means that no parameter given in this file has ever to be optimized. In fact, those files are used in a read-only mode and they can also be given in their absolute paths. For example, if we ever want that some file taken from a database would be updated, we would simply substitute the older file with one with the new data. Those files DO NOT need to be included in the registration file.

Concluding, we distinguish the input files in two categories:

- Input files that contain parameters that we might want to optimize. Those files have to be included in the registration file.
- Input files that contain parameters that we know we will never attempt to optimize. Such files contain constants or values/files from a (public) database.

7.5 Examples of input files and their registration

Below there are two examples of registering models with only one input file.

Sample 9: A simple model input file: no replications of parameters or lines

```
100.123
2.2          // This is a remark
45
300.0
```

Sample 10: The registration file of sample input file 9

```
<InOutputFile>
    <!-- settings for call of model program -->
    <ModelProgramCall>
        <CommandLine>Fit-Functions/DL_conv/bin/DrudeLorentzConv.exe</CommandLine>
        <CalculationMethod>AtOnce</CalculationMethod>
```

```
    <ParallelizationPossible>Yes</ParallelizationPossible>
</ModelProgramCall>


<!-- define number of input files -->
<NumberInputFiles>1</NumberInputFiles>


<!-- description of input file 1 -->
<InputFile>


    <!-- define name of input file -->
    <InputFileName>in.txt</InputFileName>


    <!-- define number of unreplicated lines -->
    <NumberLines>4</NumberLines>


    <!-- describe 1st line -->
    <line group="false">
        <NumberReplicationLine></NumberReplicationLine>


        <!-- define number of parameters of current line -->
        <NumberParameterLine>1</NumberParameterLine>


        <!-- define settings for parameter XValue1 -->
        <Parameter group="false">
            <NumberReplicationParameter></NumberReplicationParameter>
            <Name>XValue1</Name>
            <Format>F7.3</Format>
            <LeadingString></LeadingString>
            <TrailingString></TrailingString>
        </Parameter>
    </line>


    <!-- describe 2nd line -->
    <line group="false">
        <NumberReplicationLine></NumberReplicationLine>


        <!-- define number of parameters of current line -->
        <NumberParameterLine>1</NumberParameterLine>


        <!-- define settings for parameter ParameterB -->
        <Parameter group="false">
            <NumberReplicationParameter></NumberReplicationParameter>
            <Name>ParameterB</Name>
            <Format>F4.1</Format>
            <LeadingString></LeadingString>
            <TrailingString>          // This is a remark</TrailingString>
        </Parameter>
    </line>
```

```
        <!-- describe 3rd line -->
        <line group="false">
            <NumberReplicationLine> </NumberReplicationLine>


            <!-- define number of parameters of current line -->
            <NumberParameterLine>1</NumberParameterLine>


            <!-- define settings for parameter ParameterC -->
            <Parameter group="false">
                <NumberReplicationParameter></NumberReplicationParameter>
                <Name>ParameterC</Name>
                <Format>I2</Format>
                <LeadingString></LeadingString>
                <TrailingString></TrailingString>
            </Parameter>
        </line>


        <!-- describe 4th line -->
        <line group="false">
            <NumberReplicationLine>NumberOscillators</NumberReplicationLine>


            <!-- define number of parameters of current line -->
            <NumberParameterLine>1</NumberParameterLine>


            <!-- define settings for parameter ParameterD -->
            <Parameter group="false">
                <NumberReplicationParameter></NumberReplicationParameter>
                <Name>ParameterD</Name>
                <Format>F5.1</Format>
                <LeadingString></LeadingString>
                <TrailingString></TrailingString>
            </Parameter>
        </line>
    </InputFile>


    <!-- xml-description of the 1st output file -->


    <!-- define number of output files -->
    <NumberOutputFiles>1</NumberOutputFiles>


    <!-- description of output file 1 -->
    <OutputFile>


        <!-- define name of output file -->
        <OutputFileName>FitFunctionValues.dat</OutputFileName>


        <!-- define OnlyYColumn flag -->
        <OnlyYColumn>yes</OnlyYColumn>
```

```
        <!-- define format of output file -->
        <OutputFileFormat>ascii</OutputFileFormat>


        <!-- define number of header lines -->
        <NumberHeaderLines></NumberHeaderLines>


        <!-- define character indicating comments -->
        <CharacterForComments></CharacterForComments>
    </OutputFile>
</InOutputFile>
```

Sample 11: ASCII input file for the conventional Drude-Lorentz model (model where one line has to be repeated as many times as given by the parameter `NumberOscillators`

```
                100.123
                3.5
                3
                100.0    456.234    12.120
                250.0    106.641     2.127
                403.0    251.577    30.022
```

Sample 12: The registration file of sample ASCII input file 11

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<InOutputFile>
    <!-- settings for call of model program -->
    <ModelProgramCall>
        <CommandLine>Fit-Functions/DL_conv/bin/DrudeLorentzConv.exe</CommandLine>
        <CalculationMethod>AtOnce</CalculationMethod>
        <ParallelizationPossible>Yes</ParallelizationPossible>
        <InputDataPath>DataIn.dat</InputDataPath>
    </ModelProgramCall>


    <!-- define number of input files -->
    <NumberInputFiles>1</NumberInputFiles>


    <!-- description of input file 1 -->
    <InputFile>


        <!-- define name of input file -->
        <InputFileName>in.txt</InputFileName>


        <!-- define number of unreplicated lines -->
        <NumberLines>4</NumberLines>


        <!-- describe 1st line -->
        <line group="false">
            <NumberReplicationLine> </NumberReplicationLine>


            <!-- define number of parameters of current line -->
            <NumberParameterLine>1</NumberParameterLine>
```

```
        <!-- define settings for parameter XValue1 -->
        <Parameter group="false">
            <NumberReplicationParameter></NumberReplicationParameter>
            <Name>XValue1</Name>
            <Format>F15.8</Format>
            <LeadingString></LeadingString>
            <TrailingString></TrailingString>
        </Parameter>
    </line>


    <!-- describe 2nd line -->
    <line group="false">
        <NumberReplicationLine> </NumberReplicationLine>


        <!-- define number of parameters of current line -->
        <NumberParameterLine>1</NumberParameterLine>


        <!-- define settings for parameter EpsilonInfinity -->
        <Parameter group="false">
            <NumberReplicationParameter></NumberReplicationParameter>
            <Name>EpsilonInfinity</Name>
            <Format>F15.8</Format>
            <LeadingString></LeadingString>
            <TrailingString></TrailingString>
        </Parameter>
    </line>


    <!-- describe 3rd line -->
    <line group="false">
        <NumberReplicationLine></NumberReplicationLine>


        <!-- define number of parameters of current line -->
        <NumberParameterLine>1</NumberParameterLine>


        <!-- define settings for parameter NumberOscillators -->
        <Parameter group="false">
            <NumberReplicationParameter></NumberReplicationParameter>
            <Name>NumberOscillators</Name>
            <Format>I6</Format>
            <LeadingString></LeadingString>
            <TrailingString></TrailingString>
        </Parameter>
    </line>


    <!-- describe 4th line -->
    <line group="false">
        <NumberReplicationLine>NumberOscillators</NumberReplicationLine>


        <!-- define number of parameters of current line -->
        <NumberParameterLine>3</NumberParameterLine>
```

```
        <!-- define settings for parameter EigenFrequency -->
        <Parameter group="false">
            <NumberReplicationParameter></NumberReplicationParameter>
            <Name>EigenFrequency</Name>
            <Format>F8.1</Format>
            <LeadingString></LeadingString>
            <TrailingString></TrailingString>
        </Parameter>


        <!-- define settings for parameter PlasmaFrequency -->
        <Parameter group="false">
            <NumberReplicationParameter></NumberReplicationParameter>
            <Name>PlasmaFrequency</Name>
            <Format>F11.3</Format>
            <LeadingString></LeadingString>
            <TrailingString></TrailingString>
        </Parameter>


        <!-- define settings for parameter Damping -->
        <Parameter group="false">
            <NumberReplicationParameter></NumberReplicationParameter>
            <Name>Damping</Name>
            <Format>F11.3</Format>
            <LeadingString></LeadingString>
            <TrailingString></TrailingString>
        </Parameter>
    </line>
</InputFile>


<!-- xml-description of the 1st output file -->


<!-- define number of output files -->
<NumberOutputFiles>1</NumberOutputFiles>


<!-- define AllInOneOutputFile flag -->
<AllInOneOutputFile>yes</AllInOneOutputFile>


<!-- description of output file 1 -->
<OutputFile>


    <!-- define name of output file -->
    <OutputFileName>FitFunctionValues.dat</OutputFileName>


    <!-- define OnlyYColumn flag -->
    <OnlyYColumn>yes</OnlyYColumn>


    <!-- define format of output file -->
    <OutputFileFormat>ascii</OutputFileFormat>


    <!-- define number of header lines -->
```

```
        <NumberHeaderLines></NumberHeaderLines>


        <!-- define character indicating comments -->
        <CharacterForComments></CharacterForComments>
    </OutputFile>
</InOutputFile>
```

The outer tag `Section` is divided in four main parts:`ModelProgramCall`, `NumberInputFiles`, `InputFile` (as many times as specified by `NumberInputFiles`), `NumberOutputFiles` and `OutputFile` (as many times as specified by `NumberOutputFiles`; see §7.13)

The simplest tags will be explained here, and the more complex ones will be explained in separate sections.

Information about some simple tags:

- The tag `<NumberInputFiles>` defines the number of input files that are described in the registration XML file for the given model. The description of each input file has to be included within a separate `<InputFile>` tag.

```
    <NumberInputFiles>2</NumberInputFiles>
    <InputFile>
        <title>Description of the first file</title>
        <InputFileName>File1.txt</InputFileName>
        <NumberLines>4</NumberLines>
        ...
    </InputFile>
    <InputFile>
        <title>Description of the second file</title>
        <InputFileName>File2.txt</InputFileName>
        <NumberLines>4</NumberLines>
        ...
    </InputFile>
```

- The tags `<InputFileName>` and `<OutputFileName>` have to include the file name (and relative path) of the input and output files of the fit function program, respectively.

7.6 Function calls

The `ModelProgramCall` tag includes the directives concerning how to perform the function calls, or in other words the execution of the external model program.

- As mentioned above, the tag `<PathStartScript>` contains the (relative or absolute) path and the filename of the start script. The relative path has to be defined relative to the MAGIX root directory.
- The command to start the start script is defined by the tag `<ExeCommandStartScript>`.
- `<ParallelizationPossible>`: The tag `<ParallelizationPossible>` defines whether the external program allows parallelized running (`yes`) or not (`no`). Section § 7.7 sets the requirements for a model program to be able to be executed in a parallelized MAGIX session, while how this works is also explained in detail.
- In order to calculate the χ^2 at every one of the X points for which we have a Y value in our experimental data files, we need the value of the fit function at the same X point. Therefore it is often necessary to resample our fit function, i.e. calculate it at the same X points for which we have experimental data.

As the resampling method integrated in MAGIX cannot be exact and it may not meet the requirements of the user, it is highly recommended that the user provides his/her own resampling routines/modules. If s/he has such routines available, then s/he simply has to make sure that those routines are executed at each completion of the external model program has completed, which can be achieved by adding the necessary command in the post-processing part (after calling the model program) of the MAGIX start script (§7.3).

If the fit function program needs the X points of the experimental data where the fit function should be calculated, then the user has to make use of the `<InputDataPath>` tag, in order to define the file name of a ASCII data file that contains only the X points of all experimental data files which are located within the given exp. data ranges. For example, the user would like to use a experimental data file which contains the measured transmission between 1 - 5 MHz. Additionally, the user defines a data range from 3 - 4 MHz. MAGIX will write all X column values between 3 - 4 MHz to the ASCII file with name defined in the tag `<InputDataPath>`. So, MAGIX will write only those X column values to the ASCII file for which the values of the model function are required. If the tag `<InputDataPath>` is not empty, MAGIX writes all X column values to one big ASCII file which is stored to the same MAGIX temp directory where the input file(s) are written to. The first n lines of the ASCII file contain the n X column values of the first experimental data file. The next m lines correspond to the m X column values of the second experimental data file etc. For example, the first experimental data file contains the X column values:

```
100.341234
101.234512
102.342456
```

The second experimental data file contains the X column values:

```
340.123456
341.986453
342.567887
343.103245
```

Then the ASCII file with name defined in the tag `<InputDataPath>` looks like:

```
100.341234
101.234512
102.342456
340.123456
341.986453
342.567887
343.103245
```

If the external model program requires the number of X column values of the ith experimental data file within the defined ranges, the user has to use the MAGIX variable "NumberXValue" followed by the integer number of the ith experimental data file in the registration XML file. In the example described above, the variable "NumberXValue1" is equal to 3 and contains the number of X column values in the first experimental data file. Accordingly, the variable "NumberXValue2" is equal to 4 and contains the number of X column values of the second experimental data file. If the external model program requires the number of experimental data files as well, the user has to use the MAGIX variable "MAGIXImportNumberExpFiles" in the registration XML-file. In the example described above, the MAGIX variable "MAGIXImportNumberExpFiles" is set to 2.

If the external model program requires the number of X column values of each experimental data file, the user has to use the MAGIX variable "NumberXValueAll" in the registration

XML file. The MAGIX variable "NumberXValueAll" is a list where the *i*th element represents the number of X columns of the *i*th experimental data file. In the example described above, the MAGIX variable "NumberXValueAll" has the value "3, 4". Note, the numbers of X column values of each experimental data file are separated by ",".

There are two cases where we need the X points:

- For external model programs that need some kind of sampling, i.e. inter- or extrapolation and perform it with their own resampling routines.
- To avoid resampling by MAGIX, if the external model program reads the experimental data files internally and therefore also the resampling is done within the execution of the model function call.

In those two cases, the content of the `<InputDataPath>` tag is needed in the registration file.

7.7 Parallelization

When we talk about parallelization of a MAGIX run, we never talk about parallelization of the model program. The parallelization of the external program is something that concerns the user of the external program and has nothing to do with MAGIX, except that the total number of processors that are available is not exceeded. For example, the external model program uses 3 cores (processors) and the user set the number of processors defined in tag `<NumberProcessors>` in the fit-control file (§5) to 4, the computer must have at least 12 cores (processors).

Of course it is recommended that the model program is itself parallel, especially in programs that are very time-consuming, but whether the model program is parallel or not is absolutely irrelevant for this model running in MAGIX.

In order that a model program is used in an parallelized MAGIX run, we need to make sure that it is possible to execute two or more instances of the same model on the same machine at the same time. This requirement is not met in the following cases:

- When the input and output files of the external model program are given in absolute paths: If even one of those files is given in its absolute path and more than one calls of the external model program run at the same time, then a file that was created will be overwritten by another call of the model function.
- When the external model program reads data from public databases, which do not allow access of more than one people at the same time: Then if one program accesses the database, and another program asks also for access, then the second program will not get access.
- When the program creates many/big temporary files during the run: Even if those files are deleted in the end of the program's run, when MAGIX executes such a program in a lot of treads, then the size of the files that are temporarily written might become too big for the system to handle.

A very coarse way to check if parallelization is possible is to execute two model runs on the same machine at the same time (not even using MAGIX). If the system does not crash, then it is very probable that parallelization IS possible, unless there is a problem of storage of the temporary files when more than two instances are executed at the same time.

In case that `<ParallelizationPossible>` is set to `yes`, it might be crucial to check out the environment variables `MAGIXTempDirectory` and `OMP_STACKSIZE` (§ 1.6.1). Parallelization is

especially needed in cases that we use algorithms that perform a lot of function calls to the external model program; such algorithms are the swarm algorithms, as function calls of the external model program can always be performed in parallel. If there is only one processor available, we need e.g. $N_c > 1$ function calls, and each function call needs CPU time equal to t_c on average, then a single processor would do it serially in a time period equal to $t_c N_c$. If we were to perform N_c function calls in parallel with p processors, we would roughly need a time period equal to $t_c N_c / p$ (see also § A.2.2). So in swarm algorithms it is better to have a big number of processors and a small size of RAM than the contrary.

7.8 Line description

Each `<InputFileName>` tag describes one input file. The `<InputFileName>` tag contains one occurrence of the following tags: `<InputFileName>` and `<NumberLines>`. Inside `<InputFileName>`, there is also the `<line>` tag occurring so many times as specified by the `<NumberLines>` tag.

- The tag `<NumberLines>` defines the total number of lines (without replications) included in a given input file.
- The description of each line is bracketed inside a `<line>` tag. The number of `<line>` tags is defined by `<NumberLines>` of `<InputFileName>`.
- The tag `<NumberParameterLine>` is given only once within one `<line>` block and defines the number of model parameters (without replications) contained in that line.

7.9 Replication of lines

7.9.1 Basic properties of line replication

The attribute `group` of the `<line>` tag indicates if this line is grouped with other lines forming a block that may be replicated. There are three basic things that are needed for the full and non-ambiguous declaration of a group:

(a) the way to distinguish different groups from one another

(b) how to define the beginning and the end of a group

(c) the number of replications

The attribute `group` may contain several (key)words, the combination of which provides all this information. Those words may indicate the affiliation to a group (`group1`, `group2` etc.), the beginning of a group (`group1: start`), and information about the replication number of the current group.

- Each group corresponds to an integer ID number. This is our way to distinguish different groups from one another. The affiliation to a group is expressed when the line's `group` attribute contains the string `group` followed by the group's ID number and a colon, e.g. if the ID number of the group is 3, the group attribute contains the string `group3:`. All lines that belong to the same group 3 contain the string `group3:`.
- A block of lines that may be replicated is named after the word `group` followed by the number of the block (and is quoted in the `group` attribute).
- If a line does not belong to a group, then its `group` attribute is `false`:

```
<line group="false">
```

- The first line of the input file whose `group` attribute is not `false` is the first line of the first group (`group1:`); therefore the `group` attribute should start with `group1`, followed by a colon, a space and the word `start` (e.g. `group1:␣start`).
- When the `group` attribute contains the word `start`, this means the beginning of a new group.
- The last line of the input file whose `group` attribute contains the string `group1:` is the last line of the group.
- The number of the lines that exist between (and including) the first and the last lines of the group in the registration file is the number of necessary lines that one block of the group can consist of.
- The number of parameters contained in a single block of the group is equal to the number of the parameters contained in all the lines of the group, i.e. the sum of the quantities specified by the `<NumberParameterLine>` tag of all the lines.
- MAGIX stops looking for further groups, if the next group number does not exists. For example, the user defines the groups `group1:` and `group2:`. If a group with number 3 (`group3:`) is not defined, MAGIX stops looking for further groups, even if there is a group with number 4 (`group4:`).

Note, if you would like to use more than 100 groups in the registration file you have to modify the source code, i.e. you have to increase the variable `MaxNumberOfGroups` defined in line 2147 of the file `Modules/magix-parts_python/FittingEngine.py`.

So we have already explained how we give information about points (a) and (b) in the above list of requirements. Point (c), i.e. how we can set the number of line replications, is described in §7.10.

7.9.2 Groups of lines nested innerly of other groups of lines

It is possible that within a group of lines there is another group of lines that may be replicated as a separate group nested inside the outer group.

- The line's `group` attribute has to contain the affiliations to all the groups to which the line belongs. For example, the following line describes a line in an input file, which belongs to the `group1` and describes additionally the start of a second group called `group2`:

```
<line group="group1:; group2: start, replication = Number_MolLine">
```

- The string of the `group` attribute begins with the affiliation to the outermost group. The affiliations to all nested groups follow on the right hand side of the string. The affiliation to the innermost group is the last one within the string of the `group` attribute (right-most), see above.
- When there are nested groups, the affiliations to the different groups are separated between each by a semicolon and a space (`;␣`).

Note, that each group number have to be followed by an colon !

Sample 13: Part of a registration file with nested loops

```
<line group="group1: start, replication = Number_Molecules">
    <NumberParameterLine>2</NumberParameterLine>
    <Parameter group="false">
        ...
    </Parameter>
</line>
<line group="group1:; group2: start, replication = Number_MolLine">
    <NumberParameterLine>15</NumberParameterLine>
    <Parameter group="false">
        ...
    </Parameter>
</line>
```

7.10 Setting the replication number for lines

What's left to explain as far as it has to do with the declaration of a group is the way to set the replication number, i.e. the number of blocks of the same group that are contained in the input file, one below another.

7.10.1 Properties of the replication number for lines

A line can be replicated as many times as set by the replication number, whose value can be set in one of the following ways:

- The number of replications may be set equal to a specific integer number > 0. This way to set the number of replications for a group should be avoided and only be used in the special case where this number is not included in the input file.
- More flexibility is available by setting the number of replications equal to the value of another model parameter which is defined in the input file and included in the registration file of the model. That way you don't have to edit the registration file every time you want to remove or add a block to a specific group. You just have to edit the instance, since the number of replications is actually just another parameter.

  This parameter has to be a non-grouped variable if the current group is not nested. If the group is nested, then this parameter has to be a parameter contained in one of the lines grouped within the current group. In other words, the replication number of any group has be defined outside of this group.

In the following, I almost always refer to setting the replication number automatically equal to one of the parameters included in the parameter list by specifying the parameter name instead of the actual number of replications (except if I want to give examples of exact number of replications of a file's elements). Nevertheless, the replication number of the group is declared in the same way, no matter if it is given by an integer number or by the name of an (integer) parameter.

7.10.2 Specify the groups in the registration files

When a group starts in a given line, then the `group` attribute is a string that contains the following segments:

1. the string `group`

2. the (integer) ID number of the group; this is a successive number, depending on the order with which the group appears in the current file
3. a colon, a space, the word `start`, a comma, a space, the word `replication`, a space, an equal sign, a space (`:␣start,␣replication␣=␣`)
4. the name of the parameter that specifies the number of replications of the group that starts in the current line (or a raw integer number, as explained above)

Example A: If the ID number of the group is N and the name of the parameter that sets the replication number is `pname`:

```
<line group="groupN: start, replication = pname">
```

Example B: If at a given line a group 3 starts and its replication number is given by parameter with name `pname`, but this line already belongs to group 2, which is nested inside group 1, then this line's group attribute would be like:

```
<line group="group1:; group2:; group3: start, replication = pname">
```

We see that the segments for each group contain all necessary information for the given group.
The next line will be:

```
<line group="group1:; group2:; group3:">
```

since it is not in this line that group 3 begins, but this line still belongs to the groups 1, 2, 3.

7.11 Parameter description

7.11.1 Main tags

- The settings for each parameter are given within the `<Parameter>` tags.
- The number of `<Parameter>` tags occurring within a `<line>` tag has to be equal to the value given in the `<NumberParameterLine>` tag WITHOUT counting the replications!
- The `<Format>` tag indicates the format of the parameter value (like FORTRAN notation):

Example A: `I5` for integers with maximum 5 digits, `A4` for strings with maximum 4 characters, `F15.8` for real numbers with 8 decimal places and maximum $15 - 8 - 1 = 6$ integer places.

Make sure that you define enough integer places! During the fitting algorithm, the value of the parameter may increase by many magnitudes. If you do not specify enough integer places, the value of the parameter is replaced by `********` and MAGIX will abort with an error message! Therefore, we strongly recommend to use `ES25.15` for all float numbers.

Due to the fact that MAGIX works in double precision, it is useless to define more than 15 decimal places.

Note, the `<Format>` tag defines the number of decimal places of a real number. Therefore, it is possible to define the step size of the χ^2 function, i.e. the accuracy of the χ^2 determination. For example, the user sets the format tag of a parameter A to `F15.2`. Although, MAGIX works internally with 15 decimal places, the χ^2 value will vary only, if the value of the parameter A changes within the first two decimal places. Otherwise the χ^2 value will be the same (if the call of the external model program produces the same χ^2 value for the same parameter set).

- The <LeadingString> and <TrailingString> tags define the leading and the trailing part of the string that contains the parameter value, respectively.

Example B: If the model parameters are set by command words in the input file of the external fit function program, these command words can be inserted in these tags. E.g. imagine that a quantity, i.e. the frequency, is set in the input file by preceding its value with the command word FREQUENCY =; then this command word has to be inserted in the <LeadingString> tag:

```
<LeadingString>FREQUENCY =</LeadingString>
```

- The (optional, not required) <Visible> tag defines the visibility of the current parameter. If the value of this tag is set to false, the current parameter is not written to the current input file. The value of this tag can be controlled by a single comparison or a sequence of comparisons. Here, the phrase "comparison" means, that the user has to define the name of another parameter called comparison-name defined in a previous line of the input file and the so-called comparison-value. If the user defined parameter contains exactly the comparison-value, the <Visible> tag is set to true and the current parameter is written to the current input file.

  Example:

```
<Visible>ParameterA = branch1</Visible>
```

  In this example, the current parameter will be written to the input file, if the parameter ParameterA contains exactly the phrase branch1, otherwise the current parameter won't be written to the input file. (Leading and trailing blanks of the comparison-value will be removed before the comparison.)

  Please note, that the "=" character has to be always given. NO other comparison is possible.

  Additionally, MAGIX offers the possibility to define a sequence of comparisons. The current parameter will be written to the current input file, if all comparisons are true. The comparisons are separated by the ";" character. So, the comparison-value must not include the ";" character!

  Example:

```
<Visible>ParameterA = branch1; ParameterB = subbranch1</Visible>
```

  Here, the current parameter will be written to the input file, if the parameter ParameterA contains exactly the phrase branch1 and if the parameter ParameterB contains exactly the phrase subbranch1, otherwise the current parameter won't be written to the input file.

  Note, if the <Visible> tag is not given, MAGIX sets this parameter to visible, i.e. the current parameter will be written to the current input file.

7.11.2 Replication of parameters

- The <NumberReplicationParameter> tag defines the number of replications of the parameter in the current line. The default value is 1. As for the replication number of lines, the replication number of parameters can be defined either by an exact integer number or by the name of a parameter whose value is given in the same file (§7.10).

Example A:
```
<NumberReplicationParameter>ParameterA</NumberReplicationParameter>
```

where the value of parameter `ParameterA` defines the number of replications of the current parameter.

- The `<Parameter>` tags contains the attribute `group`, which is obsolete and always set to `false`.

Example B: If you want to repeat the variable `ParameterA` two times, then the XML description in the registration file would look like:

```
<line group="false">
    <NumberReplicationLine> </NumberReplicationLine>


    <!-- define number of parameters -->
    <NumberParameterLine>2</NumberParameterLine>


    <!-- settings for parameter ParameterA -->
    <Parameter group="false">
        <NumberReplicationParameter>2</NumberReplicationParameter>
        <Name>ParameterA</Name>
        <Format>F4.1</Format>
        <LeadingString></LeadingString>
        <TrailingString></TrailingString>
    </Parameter>


    <!-- settings for parameter ParameterB -->
    <Parameter group="false">
        <NumberReplicationParameter></NumberReplicationParameter>
        <Name>ParameterB</Name>
        <Format>F4.1</Format>
        <LeadingString></LeadingString>
        <TrailingString></TrailingString>
    </Parameter>
</line>
```

This XML description in the registration file can describe the second one of the following lines of an input file:

```
                    // ParameterA ParameterA ParameterB
                        4.5          1.3       31.0
```

So replication for parameters is not implemented in the same way as replication for lines. Nested loops are not allowed and the `group` attribute has absolutely no meaning. A first order depth loop is only specified by changing the value of `<NumberReplicationParameter>`. Additionally, a loop over parameters can exist only over parameters of the same line.

- You can combine the tags `<NumberReplicationLine>` and `<NumberReplicationParameter>`.

Example C: If you want to repeat a line as many times as given in the parameter named as `NumParamLine`, and you also want to repeat the parameter in the current line as many times as given by the parameter `NumParamCol` (both `NumParamLine` and `NumParamCol` are given in other lines of the same file), then the corresponding XML description within the registration file would look like:

```
<line group="group1: start, replication = NumParamLine">


    <!-- define number of parameters -->
    <NumberParameterLine>1</NumberParameterLine>


    <!-- settings for parameter NumParamCol -->
    <Parameter group="false">
        <NumberReplicationParameter>NumParamCol</NumberReplicationParameter>
        <ParameterName>TestParameter</ParameterName>
        <ParameterFormat>I5</ParameterFormat>
        <LeadingString></LeadingString>
        <TrailingString></TrailingString>
    </Parameter>
</line>
```

7.12 Parameter names

- The name of the model parameter of the current line is defined inside the `<Name>` tag. The names of the model parameters used in the XML description of the input file MUST BE IDENTICAL to the names of the parameters used within the instance (§4).
- The names of the model parameters MUST NOT contain the square brackets and comma characters (`[ ] ,`). Preferably they should be arbitrary sequences of letter and/or number characters, as well as the underscore character (_). A number digit can as well be the first character of the name, but it is not recommended.

7.12.1 Parameters of the same name

- Parameters appearing more than once in the same file: There is one exception for the square brackets. When the first part of the parameter name complies with the aforementioned rules, but the name string is followed by double square brackets (`[[]]`), this means that the parameter will appear more than once within the same file. It implies that this parameter is repeated in arbitrary places of the same file, but keeping the same value. After the model has been registered, the starting value of such a parameter will be the one that is given in the first occurrence of the parameter in the instance.

  Because of the fact that it is rather inappropriate to set a parameter more than once, the use of the `[[]]` in the name of a parameter is usually used in cases where there are repeated comment lines in the input file with exactly the same content. An example is shown in sample input file 14 and the description of the corresponding lines in the registration file at sample 15. You can see the respective instance in sample 16.

Sample 14: Input file with repeated comment lines of the same content.

```
"# Comment line"
       2.7600000000
"# Comment line"
         1
"# Comment line"
         0
```

Sample 15: The description of the lines of input file 14 (part of the registration file).

```
            <line group="false">
```

```
            <NumberParameterLine>1</NumberParameterLine>
            <Parameter group="false">
                <NumberReplicationParameter> </NumberReplicationParameter>
                <Name>EmptyLine[[]]</Name>
                <Format>A1</Format>
                <LeadingString> </LeadingString>
                <TrailingString> </TrailingString>
            </Parameter>
        </line>
        <line group="false">
            <NumberParameterLine>1</NumberParameterLine>
            <Parameter group="false" type="" tab="">
                <NumberReplicationParameter/>
                <Name>Tbg</Name>
                <Format>F20.10</Format>
                <LeadingString> </LeadingString>
                <TrailingString> </TrailingString>
            </Parameter>
        </line>
        <line group="false">
            <NumberParameterLine>1</NumberParameterLine>
            <Parameter group="false">
                <NumberReplicationParameter> </NumberReplicationParameter>
                <Name>EmptyLine[[]]</Name>
                <Format>A1</Format>
                <LeadingString> </LeadingString>
                <TrailingString> </TrailingString>
            </Parameter>
        </line>
        <line group="false">
            <NumberParameterLine>1</NumberParameterLine>
            <Parameter group="false" type="" tab="">
                <NumberReplicationParameter/>
                <Name>n_shells</Name>
                <Format>I10</Format>
                <LeadingString> </LeadingString>
                <TrailingString> </TrailingString>
            </Parameter>
        </line>
        <line group="false">
            <NumberParameterLine>1</NumberParameterLine>
            <Parameter group="false">
                <NumberReplicationParameter> </NumberReplicationParameter>
                <Name>EmptyLine[[]]</Name>
                <Format>A1</Format>
                <LeadingString> </LeadingString>
                <TrailingString> </TrailingString>
            </Parameter>
        </line>
        <line group="false">
            <NumberParameterLine>1</NumberParameterLine>
            <Parameter group="false" type="" tab="">
                <NumberReplicationParameter/>
                <Name>data_tables</Name>
                <Format>I10</Format>
                <LeadingString> </LeadingString>
                <TrailingString> </TrailingString>
            </Parameter>
        </line>
```

Sample 16: The part of the instance for a model with an input file with the lines shown in sample 14, which are described in the sample of registration file 15.

```
<!-- settings for parameter EmptyLine[[]] -->
<Parameter fit="false">
    <name>EmptyLine[[]]</name>
    <value>"# Comment line"</value>
    <error> </error>
    <lowlimit> </lowlimit>
    <uplimit> </uplimit>
</Parameter>


<!-- settings for parameter Tbg -->
<Parameter fit="false">
    <name>Tbg</name>
    <value>2.7600</value>
    <error> </error>
    <lowlimit> </lowlimit>
    <uplimit> </uplimit>
</Parameter>


<!-- settings for parameter n_shells -->
<Parameter fit="false">
    <name>n_shells</name>
    <value>1</value>
    <error> </error>
    <lowlimit> </lowlimit>
    <uplimit> </uplimit>
</Parameter>


<!-- settings for parameter data_tables -->
<Parameter fit="false">
    <name>data_tables</name>
    <value>0</value>
    <error> </error>
    <lowlimit> </lowlimit>
    <uplimit> </uplimit>
</Parameter>


<!-- settings for parameter r_core -->
<Parameter fit="true">
    <name>r_core</name>
    <value>0.1200</value>
    <error> </error>
    <lowlimit>0.01</lowlimit>
    <uplimit>0.5</uplimit>
</Parameter>


<!-- settings for parameter n_e -->
<Parameter fit="false">
    <name>n_e</name>
    <value>0.0000</value>
    <error> </error>
    <lowlimit> </lowlimit>
    <uplimit> </uplimit>
</Parameter>
```

```
<!-- settings for parameter T_e -->
<Parameter fit="false">
    <name>T_e</name>
    <value>0.0000</value>
    <error> </error>
    <lowlimit> </lowlimit>
    <uplimit> </uplimit>
</Parameter>
```

- It is highly recommended that no parameters are declared with the same name, unless there is a serious reason for doing that, i.e. they are actually the same parameters that should always get the same value. Nevertheless, it is possible to declare two parameters with the same name, even if there is no reason to do that: In that case, if two or more parameters appear in the registration file with the same name without the double square brackets appended, then they are considered as different parameters and have to be declared in the instance as many times as (and in the order with which) they appear in the registration file (plus replications).
- Parameters appearing more than once in different files: Same as in the case where two parameters are declared twice in the same file, with the only difference that no double square brackets are needed in the parameter name. So, when the same parameter (a number with exactly the same value) has to appear more than once in two or more different input files of the same model, then you simply declare them with the same name (with no double square brackets).
- If a name occurs more than once in the registration file, the values of these parameters are assigned to parameter names of the registration file according to the order in which they appear in the instance, i.e. the parameters are filled in from up to down and from left to right.

Example A: A parameter with name `Eigenfrequency` occurs three times in the parameter registration file with values `200`, `400` and `700`. The input file will include these values in the following order: 200, 400, 700:

Example B: Parameter `param` contains the values `1`, `2`, `3`, `4`, `5`, and `6` may appear in the input file as follows:

```
param param param
param param param
```

Their values will be set in the following order:

```
1 2 3
4 5 6
```

7.12.2 Special parameters

In a model we want to register for MAGIX, it is possible that one or more of its input files contain some parameter(s) whose value is implied or set in some other piece (XML file) of MAGIX. We call such parameters as special parameters, which are distinguished by their names, so we refer to them as parameters with special names. The special parameters have the following characteristics, the second of which is partially derived from the first one:

(a) According to the above definition of the special parameters, it becomes obvious the special parameters are parameters for which we simply need and use their value. A special parameter is not a parameter that could ever be needed to be fitted and it could never happen that we want its value optimized.

Therefore for those parameters we don't even need to set neither a `fit` (all parameters with special names are assumed to have their `fit` attribute set to `false`) nor any upper or lower limits for its value. In fact, the inclusion of the special parameters in the model instance really has no meaning.

(b) When registering a model with an input file where the value of a special parameter is set, then of course we include this parameter in the registration file (so that we have a full description of the input file), but we don't include the tag of the respective `<Parameter>` in the model instance (because the value of a special parameter has already been given or calculated elsewhere).

The model instance will contain all other parameters listed in the registration file, but no special parameter will be included in it.

The following parameter names have a special meaning:

- All parameters whose names start with `MAGIXImport` and are completed with the name of one of the tags included in the experimental XML file. See §7.12.3.
- `NumberXValueM`, where the last character `M` of the name has to be substituted by an integer number indicating the order that a given data file is presented in the experimental XML file. If a parameter with that name occurs in the registration file, then MAGIX automatically sets the value of this parameter equal to the number of X points of the experimental file included in the experimental XML file in the order M.

  The value of this parameter is anyway calculated by MAGIX, and is equal to the number of X points of the data file that lies within the ranges specified for this file.

Example A: If you have two or more experimental data files: `NumberXValue1` identifies the number of X points in the first file described in the experimental XML file; `NumberXValue2` identifies the number of X points in the second one; and so on.

Example B: Imagine that the experimental XML file with the settings for the import of two experimental data files looks like:

```
<!-- define number of experimental data files -->
<NumberExpFiles>2</NumberExpFiles>


<!-- define import settings for 1st exp. data file -->
<file>


    <!-- define path and name of experimental data file -->
    <FileNamesExpFiles>File1.dat</FileNamesExpFiles>


    <!-- define import filter -->
    <ImportFilter>ascii</ImportFilter>
    ...
</file>


<!-- define import settings for 2nd exp. data file -->
```

```
<file>


    <!-- define path and name of experimental data file -->
    <FileNamesExpFiles>File2.dat</FileNamesExpFiles>


    <!-- define import filter -->
    <ImportFilter>ascii</ImportFilter>
    ...
</file>
```

Then a parameter named as `NumberXValue1` in the registration file will represent the number of lines in file `File1.dat`, and `NumberXValue2` represents the number of lines in `File2.dat`.

- `NumberXValueAll`, is a comma separated list where each element represents the number of X column values located within the defined ranges of each experimental data file. For example, the first experimental data file has 10 X column values and the second experimental data file has 234 X column data points. The MAGIX variable `NumberXValueAll` contains the list `10, 234`.

7.12.3 Special parameters for the experimental data settings

As special parameters, those parameters who define settings for the data files also comply with the general characteristics listed in the previous section (§7.12.2).

There are models that use experimental data within their programs. In such a case, the external model program has to be given some directives in order to import the experimental data. But for MAGIX this information has already been given in the experimental XML file (§3). We would like to avoid specifying the settings for importing experimental data twice for similar cases. With this in mind, a new series of parameters was invented.

- The names of those parameters all begin with the string `MAGIXImport`, and the corresponding tag of the experimental XML file follows (and closes the parameter name).
- Those parameters do not appear in the instance, but are instead declared and their values set in the experimental XML file.
- If we have declared some parameter with a name `MAGIXImportX`, where `X` is a string identical to a tag name that is not expected in the experimental XML file, then an error terminates the program.

For example: Let's say we have a model's input file which contains a line with some density value, the number of observational data files and for each observational data file the size of the telescope and finally the temperature:

```
   1.41e5   2     3.50  10.43   100.3
```

The description of this line would look like:

```
<line group="false">


  <!-- define number of replications -->
  <NumberReplicationLine>1</NumberReplicationLine>
```

```
  <!-- define number of parameters -->
  <NumberParameterLine>1</NumberParameterLine>


  <!-- settings for parameter density -->
  <Parameter group="false">
    <NumberReplicationParameter></NumberReplicationParameter>
    <Name>density</Name>
    <Format>es10.2</Format>
    <LeadingString></LeadingString>
    <TrailingString></TrailingString>
  </Parameter>


  <!-- settings for parameter MAGIXImportNumberExpFiles -->
  <Parameter group="false">
    <NumberReplicationParameter></NumberReplicationParameter>
    <Name>MAGIXImportNumberExpFiles</Name>
    <Format>i3</Format>
    <LeadingString></LeadingString>
    <TrailingString></TrailingString>
  </Parameter>


  <!-- settings for parameter MAGIXImportNumberExpFiles -->
  <Parameter group="false">
    <NumberReplicationParameter>MAGIXImportNumberExpFiles</NumberReplicationParameter>
    <Name>MAGIXImportTelescopeSize</Name>
    <Format>F6.2</Format>
    <LeadingString></LeadingString>
    <TrailingString></TrailingString>
  </Parameter>


  <!-- settings for parameter temperature -->
  <Parameter group="false">
    <NumberReplicationParameter></NumberReplicationParameter>
    <Name>temperature</Name>
    <Format>F7.1</Format>
    <LeadingString></LeadingString>
    <TrailingString></TrailingString>
  </Parameter>
</line>
```

The model instance would specify the values of the parameters of this line as:

```
<!-- parameter density -->
<Parameter fit="false">
    <name>density</name>
    <value>1.4e5</value>
    <error></error>
    <lowlimit>0</lowlimit>
    <uplimit>1e99</uplimit>
</Parameter>


<!-- parameter temperature -->
<Parameter fit="false">
    <name>temperature</name>
    <value>100.3</value>
    <error></error>
```

```
    <lowlimit>0</lowlimit>
    <uplimit>1e99</uplimit>
</Parameter>
```

We see that there is no reference to the number of observational files or the telescope size, even if they are supposed to exist in the input file that contains this line. Instead of the model instance, this information is given in the experimental XML file. If the model instance were to refer to two observational data files, then MAGIX would copy this information from the experimental XML file, where it has already been specified:

```
<!-- define number of experimental data files -->
<NumberExpFiles>2</NumberExpFiles>


<!-- define import settings for 1st exp. data file -->
<file>

    <!-- define path and name of experimental data file -->
    <FileNamesExpFiles>test/345.cso</FileNamesExpFiles>


    <!-- define import filter -->
    <ImportFilter>automatic</ImportFilter>


    <!-- define number of data ranges -->
    <NumberExpRanges>2</NumberExpRanges>


    <!-- define parameters for each data ranges -->
    <FrequencyRange>
        <MinExpRange>326001</MinExpRange>
        <MaxExpRange>350000</MaxExpRange>
        <StepFrequency>1.0</StepFrequency>
        <BackgroundTemperature>2.7</BackgroundTemperature>
        <TemperatureSlope>0.0</TemperatureSlope>
    </FrequencyRange>

    <FrequencyRange>
        <MinExpRange>350001</MinExpRange>
        <MaxExpRange>360000</MaxExpRange>
        <StepFrequency>0.8</StepFrequency>
        <BackgroundTemperature>2.8</BackgroundTemperature>
        <TemperatureSlope>1.0</TemperatureSlope>
    </FrequencyRange>


    <!-- define size of telescope -->
    <TelescopeSize>3.5</TelescopeSize>
</file>


<!-- define import settings for 2nd exp. data file -->
<file>


    <!-- define path and name of experimental data file -->
    <FileNamesExpFiles>magix_import_vars/345.cso</FileNamesExpFiles>
```

```
    <!-- define import filter -->
    <ImportFilter>automatic</ImportFilter>


    <!-- define number of data ranges -->
    <NumberExpRanges>2</NumberExpRanges>


    <!-- define parameters for each data ranges -->
    <FrequencyRange>
        <MinExpRange>326001</MinExpRange>
        <MaxExpRange>350000</MaxExpRange>
        <StepFrequency>1.0</StepFrequency>
        <BackgroundTemperature>2.7</BackgroundTemperature>
        <TemperatureSlope>0.0</TemperatureSlope>
    </FrequencyRange>

    <FrequencyRange>
        <MinExpRange>350001</MinExpRange>
        <MaxExpRange>360000</MaxExpRange>
        <StepFrequency>0.8</StepFrequency>
        <BackgroundTemperature>2.8</BackgroundTemperature>
        <TemperatureSlope>1.0</TemperatureSlope>
    </FrequencyRange>


    <!-- define size of telescope -->
    <TelescopeSize>10.4</TelescopeSize>
</file>
```

7.13 Output file settings

In order to determine the value of χ^2 the user has to define how MAGIX should read in the output file(s) of the external model program. This is done in the registration xml-file as well.

The tag `<AllInOneOutputFile>` indicates if all output of the external model program is stored in one big output file ”yes“ or not ”no“. So, if the user sets the tag `<AllInOneOutputFile>` to ”yes“ MAGIX assumes that the external model program produces only one output file independent of the number of experimental data files. MAGIX expects that the values of the model function are written into one file where the first line corresponds to the first line of the first experimental data file etc. For example, the user would like to use two experimental data files which contain the transmission for n data points from 1 - 3 MHz in the first file and for m data points from 5 - 6 MHz in the second experimental data file, respectively. If the user sets the tag `<AllInOneOutputFile>` to ”yes“, MAGIX expects that the external model program produces one file where the first n lines corresponds to the transmission from 1 - 3 MHz and the next m lines corresponds to the transmission from 5 - 6 MHz.

Note, if the user sets the tag `<AllInOneOutputFile>` to ”yes“ the INTERPOLATION routine which is included in MAGIX can not be used!

MAGIX can handle more than one output files. The number of output files produced by the external model program is given by the tag `<NumberOutputFiles>`, where the content has to be always an integer > 0. If the user sets the tag `<AllInOneOutputFile>` to ”yes“ the number of output files is automatically set to 1.

Note, if the content of the tag `<NumberColumnsY>` is larger than 1 the external model program has to produce ALWAYS output file(s) which has (have) the same number of Y columns. For

example, the user sets the tag <NumberColumnsY> to 2 and the external model program produces one output file. Then the output file of the external model program has to contain two Y columns as well.

If the external model program produces always the same number of output files than experimental data files, the tag <NumberOutputFiles> has to be set to "MAGIXImportNumberExpFiles". In this case, the user has to define only one output file where MAGIX expects that the name of the output file is expanded by "_" followed by the number of the output file. For example, the user would like to use two experimental data files. The name of the output file is set to "output.dat". MAGIX expects that the output of the external model program is written to the files "output_1.dat" and "output_2.dat", where the file "output_1.dat" describes the first experimental data file and "output_2.dat" the second data file.

In any other case, the tag <OutputFile> has to occur as many times as the number given by <NumberOutputFiles>.

Each <OutputFile> tag contains the following settings for each output file.

- The tag <OnlyYColumn> defines if the output file(s) include(s) only the values of the model function (Y column) without the corresponding X column values "yes" or if the output file(s) contain(s) the X column values as well "no". If the tag <OnlyYColumn> includes a content unequal to yes, MAGIX reads in the X column as well as the Y column values of the model function and checks if the X column values in the output file(s) of the external model program correspond to the given X column values defined in the observation file(s). If the X column values are not identical, MAGIX performs always a simple interpolation which is controlled by the tags <InterpolationMethod> and <NormalizationFlag>, see below. In order to prevent an interpolation of the model function the user has to set the tag <OnlyYColumn> to "yes".

  Please note, interpolation is neither possible if all output files are stored in one big output file, i.e., the tag <AllInOneOutputFile> is set to "yes".

- The tag <OutputFileFormat> defines the format of the output file. The user can choose one of the following commands: ascii (default) and dat (for an ASCII file) and fits (for a FITS file). Additionally, the command auto and automatic selects the format of the current output file by the ending of the file name; i.e. if the file name ends in .fits, MAGIX expects a FITS file; otherwise MAGIX expects an ASCII file.

- If the tag <OutputFileFormat> is set to ascii or dat MAGIX reads in the contents of the tag <NumberHeaderLines> which defines the number of header lines. These lines at the beginning of the output file(s) are ignored during the import of the output file(s). If the tag is empty or not defined (default), MAGIX expects no header lines.

- If the tag <OutputFileFormat> is set to ascii or dat MAGIX reads the tag <CharacterForComments> which defines a character, with which a comment begins in the lines of the output file. If the tag is empty or not defined (default), MAGIX expects no comments in the output file(s).

- The tag <ValueEmptyOutputFile> defines the value of the model function if the output file(s) is empty. If the tag is empty or not defined (default), MAGIX sets this value to 0.

- The tag <LSRAdjustement> defines a global shift of the X column values. The value defined in the tag <LSRAdjustement> is subtracted from each X column value. For example, the user would like to model the transmission between 1 - 3 MHz, but the external model program produces the transmission between 5 - 7 MHz. By setting the tag to

<LSRAdjustement> to 4, MAGIX subtracts 4 from each X column value. If the tag is empty or not defined (default), MAGIX sets this value to 0.

Note, this option is only available, if tag <OnlyYColumn> is set to ”no“. If the number of X columns is greater than 1, MAGIX uses only one lsr adjustment value for all X columns!

- The tag <LSRAdjustementFlag> defines if the value of the tag <LSRAdjustement> should be fitted ”yes“ or not ”no“. If the tag is empty or not defined (default), MAGIX sets this value to ”no“.
- The tag <InterpolationMethod> defines the method of interpolation. The user can select ”nearest-neighbour“ or ”linear“. For ”linear“, MAGIX makes a linear interpolation. For ”nearest-neighbour“, MAGIX uses the model function value of the nearest neighboured point. If the tag is empty or not defined (default), MAGIX sets this value to ”nearest-neighbour“.
- If the tag <NormalizationFlag> is set to ”yes“, MAGIX normalizes the interpolated model function, i.e. MAGIX guarantees that the integral of the interpolated model function over the whole experimental data range is equal to the integral of the experimental data over the the whole experimental data range. If the tag is empty or not defined (default), MAGIX sets this value to ”yes“.

8 Algorithms implemented

MAGIX provides optimization through one of the algorithms described below, or via a combination of more than one algorithms (algorithm chain).

8.1 Levenberg–Marquardt algorithm (LM)

advantages	disadvantages
fast	derivative method
easy to parallelize	finds only local minima

The Levenberg–Marquardt (LM) algorithm is a kind of interpolation between the Gauss–Newton algorithm and the method of gradient descent. The Gauss–Newton algorithm (GN) is a method used to solve non-linear least square problems; it is a modification of Newton's method to find the minimum of a function, but it is constrained so that it can only MINIMIZE a sum of square function values. The gradient descent algorithm searches for a local minimum of a function going towards the direction that the gradient decreases most quickly. More information under the corresponding lemmas in Wikipedia {1}.

It is possible that LM can find the local minimum even if it starts very far from it, but that depends on the landscape of the parameters. On the other hand, for functions and starting values of parameters that lie very close to the final minimum, LM tends to be a bit slower than GN. In any case LM is an algorithm that depends very much on the starting values of the parameters that are to be optimized, and the user should choose the starting values very carefully. If we have no idea of a starting value of a parameter, then we can do one of the following:

- With the use only of LM: We can run more than one MAGIX instances, giving to the parameter starting values, each one very far away from each other.
- With an algorithm chain: We use a fit control file with e.g. two algorithms, the first of which is the bees (§ 8.5) that will find, say, three best sites. The results for these three best sites found by the bees, will be used as starting values for the second algorithm, LM.

Additionally, MAGIX determines the variance (squared uncertainties) of the fitted parameters from the diagonal elements of the covariance matrix. The square roots of each diagonal element of the covariance matrix are written to the error tags of the instance file. For more information, see the corresponding lemma in Wikipedia {1} and Numerical Recipes {11}.

Please note, the LM algorithm requires at least N+1 data points, when N represents the number of free parameters.

8.2 Simulated Annealing (SA)

advantages	disadvantages
non-derivative method	many function calls
easy to parallelize	
can find global minimum	

Simulated annealing (SA) is a generic probabilistic meta-heuristic computational method that is used for the problem of global optimization, i.e. to find a good approximation to the global optimum of a given function in a large search space. For certain problems, simulated annealing may be more effective than exhaustive enumeration – provided that the goal is merely to find an acceptably good solution in a fixed amount of time, rather than the best possible solution {1}.

In comparison to LM, SA is more robust. Its result does not depend so much on the neighborhood of the starting point. LM would search for the highest (negative) gradient (in absolute values), and when it detects a local minimum it stops there. On the contrary, SA can detect if the gradient around this local minimum is small (low perturbation), and will continue the procedure in order to find a better minimum, i.e. lower value than the one previously found.

The name and inspiration of this algorithm come from the process of annealing in metallurgy. This technique involves heating and controlled cooling of a material, with the aim to gradually increase the size of its crystals and reduce their defects. The heat provides energy and causes the atoms to move from their initial position (which was a local minimum of internal energy) and wander randomly through states of higher energy. Then, a slow cooling gives them more chances of finding configurations of lower internal energy than the initial one.

By analogy to the physical process, each step of the SA replaces the current solution by a random nearby solution. This nearby solution is chosen with a probability that depends both on the difference between the corresponding function values and also on a global temperature, T. Temperature T is gradually decreased (in fact, it is multiplied by the temperature reduction coefficient, k, which therefore has to be < 1) during the process.

The dependency of the temperature difference between two subsequent steps is such that the current solution changes almost randomly when T is large, but it is modified increasingly downhill as T goes to zero[4]. Allowing for uphill moves (greater changes of the value of the global temperature) prevents the method from becoming stuck at local optima – which are the burden of greedier methods.

The subsequent points of the currently available SA algorithm follow a perpendicular direction. More information in Numerical Recipes {11, §10.9} and {16}.

8.3 Nested Sampling algorithm (NS)

The Nested Sampling (NS) algorithm is a combination of a Monte Carlo method and Bayesian statistics. The Monte Carlo (MC) methods is a family of computational algorithms that perform repeated random sampling and compute the results for every sample. The Bayesian statistics is a statistical inference technique, i.e. attempts to draw conclusions from data subjected to random variation. In particular, the Bayesian inference calculates the probability that a hypothesis is true, i.e. how probable it is that the newly calculated value of a parameter is closer to the real one (the value that best fits experimental data); if it is more probable than the previously calculated, then the parameter value is updated.

The actual algorithm is based on the Markov chain Monte Carlo (MCMC) methods. Those are a family of algorithms that sample from probability distributions, with the aim to construct a (Markov) chain with the desired distribution (a distribution with the desired properties) as the equilibrium distribution. The quality (how well the parameter sets fit the data) of the sample (and the distribution that makes up the chain) is a monotonically increasing function of the number of steps.

An MCMC algorithm is used in order to reduce the dimensionality of the parameter space through integration. The Bayesian

advantages	disadvantages
finds multiple minima	difficult to parallelize
probabilities for parameter sets	dependence on random generator

[4] **Temperature reduction in simulated annealing:** The temperature changes increasingly downhill as $T \to 0$, i.e. in absolute numbers; this means that the reduction coefficient remains constant, but e.g. when $k = 0.8$ and the start temperature is $T_0 = 1,000$, then $T_1 = 800$ and $\Delta T_1 = 200$; $T_2 = 640$ and $\Delta T_2 = 160 < \Delta T_1$ etc.

approach to this technique allows not only to find multiple solutions, but also to estimate the confidence intervals of parameters values and to evaluate the Bayesian evidence[5].

In general, our NS algorithm requires fewer samples of standard MCMC methods, while also provides posterior probabilities for each one of the best parameter vectors. For more information see also {1, 12}.

8.4 Particle Swarm Optimization (PSO)

The Particle Swarm Optimization (PSO) algorithm implemented in MAGIX is a hybrid between a particle swarm optimization algorithm and a Nelder–Mead simplex search method. Particle swarm optimization is called a computational method that optimizes a problem by iteratively trying to improve a candidate solution according to some measure of quality; the particles are sent towards a better solution flying so much faster according to the technique's performance in last step. The Nelder–Mead technique or downhill simplex search method is a traditional technique for direct search of function minima; it is easy to use, does not need calculation of derivatives and therefore can converge even to non-stationary solutions.

Both of the two techniques have been modified by Fan and Zahara and their combination {6} is the hybrid algorithm that is available in MAGIX. The computational cost of the particle swarm optimization in slow convergence cases is paid back by the use of a local simplex search when the particle is found close to a solution. So the PSO algorithm performs a kind of heuristics: It starts from an initial random population of particles and searches in the neighborhood for a global optimum. Actually, the particles with the best fit function values are updated with the simplex method, while the particles with the worst function values are updated with particle swarm optimization.

advantages	disadvantages
non-derivative method	many function calls
finds global minimum	
easy to parallelize	

The whole procedure prevents the algorithm from being trapped locally, and at the same time allows it to find the global minimum. It repeats until a termination criterion or the maximal number of iterations is reached.

8.5 Bees algorithm

The bees algorithm {3} is a swarm algorithm that performs a kind of neighborhood search combined with random search. This name was given to the algorithm because it tries to mimic the collection of nutriments by bees, in that there is always a part of the population that perform the role of scouts, traveling far away towards random directions in order to detect new nutrition sources.

advantages	disadvantages
non-derivative method	many function calls
finds multiple minima	dependence on random generator
easy to parallelize	

The algorithm starts with an initial set of parameter vectors (a collection of particles or

[5] **Bayesian interpretation of probability:** As the number of steps increases, we collect evidence with regard to the consistency or inconsistency of that evidence with a given hypothesis. More specifically, as the evidence accumulates we tend to believe less or more to the given hypothesis, depending on the degree that the increasing evidence agree with that hypothesis. This is particularly interesting and different to other optimization methods, because we are used to presume that our data is true, while the Bayesian interpretation allows us also to evaluate the credibility of our data.

scout bees, i.e. the hive of bees), randomly selected and such that it spreads in all parameter space. After the fitness of each bee is evaluated in terms of the quality ranking it just visited, then the bees with the highest fitnesses visit the neighborhoods of the sites they currently are. From the rest of the bees, some are sent away to random sites and some are sent to also search in the neighborhood of the very best sites (so that there are more bees searching for food in places where it is more probable to find nutrition sources). In the end of the step the fitness of each visited site is evaluated, and the bees who just visited them move away or search more in the vicinity of the best sites.

The result is the bees algorithm finds areas of local minima.

8.6 Genetic algorithm (GA)

The Genetic algorithm is a probabilistic search algorithm that mimics the process of natural evolution. It iteratively transforms a set of parameter vectors (population), each with an associated fitness value, into a new population of objects.

advantages	disadvantages
non-derivative method	many function calls
finds multiple minima	dependence on random generator
easy to parallelize	
converges quickly	
constrains parameter ranges	

The procedure takes place using the Darwinian principle of natural selection, with operations that are patterned after naturally occurring genetic operations such as recombination and mutation. Recombination is the joined process of reproduction and crossover, i.e. mixing the genetic matter of the parents. Mutation is the complete disappearance of specific genetic material and its transformation to a completely different material; this happens especially in combinations of genetic material (parameter sets) that are not fit.

The evolution starts from a randomly selected population (of parameter sets) and proceeds in evolutionary generations (stages / iteration steps). When a generation step is completed, the fitness of all the members of the population is evaluated. Then a number of parameter sets is stochastically selected to remain as it is; the fittest members are more probable to be kept unmodified. The rest of the population members are modified; the modifications they go through are more intense as less fit they are. The modified and unmodified parameter sets make up the new population whose fitness will be evaluated in the end of the step.

For more information see also the corresponding Wikipedia lemma under Genetic algorithm {1}.

8.7 Markov chain Monte Carlo (MCMC)

We use the `emcee`[6] package {8}, which implements the affine-invariant ensemble sampler of {9}, to perform a full-parallelized MCMC algorithm. An additional installation of the `emcee` package is not necessary.

The following discussion is largely taken from {8}: The general goal of MCMC algorithms is to draw M samples $\{\Theta_i\}$ from the posterior probability density

$$p(\Theta, a|D) = \frac{1}{Z} p(\Theta, a)\, p(D|\Theta, a)\,, \tag{1}$$

[6] http://dan.iel.fm/emcee

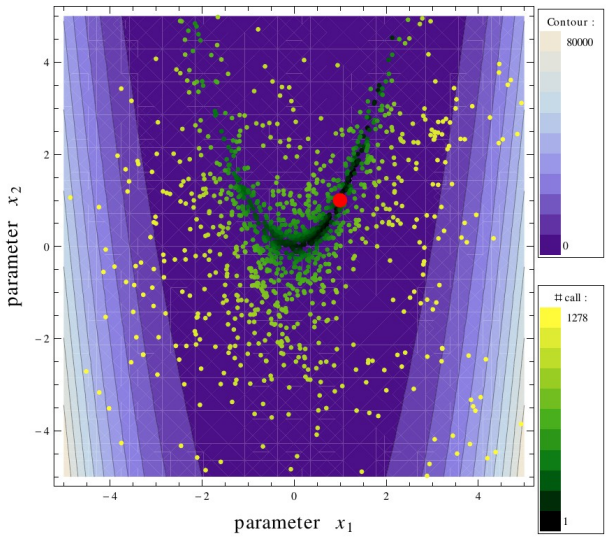


Figure 7: Results for the Rosenbrock function using MCMC: The distribution of the parameter values after 1278 function calls ($\chi^2 = 2.35 \cdot 10^{-2}$) is indicated by green-yellow points, where the dark green points indicate function calls that are made at the beginning of the fit process and light yellow points represent function calls at the end of the fit process. The red dot denotes the global minimum of the Rosenbrock function.

where the prior distribution $p(\Theta, a)$ and the likelihood function $p(D|\Theta, a)$ can be relatively easily (but not necessarily quickly) computed for any particular value of (Θ_i, a_i). The normalization $Z = p(D)$ is independent of Θ and a once we have chosen the form of the generative model. This means that it is possible to sample from $p(\Theta, a|D)$ without computing Z – unless one would like to compare the validity of two different generative models. This is important because Z is generally very expensive to compute.

Once the samples produced by MCMC are available, the marginalized constraints on Θ can be approximated by the histogram of the samples projected into the parameter subspace spanned by Θ. In particular, this implies that the expectation value of a function of the model parameters $f(\Theta)$ is

$$\langle f(\Theta)\rangle = \int p(D|\Theta) f(\Theta)\, d\Theta \approx \frac{1}{M}\sum_{i=1}^{M} f(\Theta_i). \tag{2}$$

Generating the samples Θ_i is a non-trivial process unless $p(\Theta, a, D)$ is a very specific analytic distribution (for example, a Gaussian). MCMC is a procedure for generating a random walk in the parameter space that, over time, draws a representative set of samples from the distribution. Each point in a Markov chain $X(t_i) = [\Theta_i, a_i]$ depends only on the position of the previous step $X(t_i - 1)$.

The simplest and most commonly used MCMC algorithm is the Metropolis-Hastings (M-H) method. The iterative procedure is as follows: (1) given a position $X(t)$ sample a proposal position Y from the transition distribution $Q(Y; X(t))$, (2) accept this proposal with probability

$$\min\left(1, \frac{p(Y|D)}{p(X(t)|D)}\,\frac{Q(X(t); Y)}{Q(Y; X(t))}\right). \tag{3}$$

The transition distribution $Q(Y; X(t))$ is an easy-to-sample probability distribution for the proposal Y given a position $X(t)$. A common parameterization of $Q(Y; X(t))$ is a multivariate Gaussian distribution centered on $X(t)$ with a general covariance tensor that has been tuned

for performance. It is worth emphasizing that if this step is accepted $X(t+1) = Y$; Otherwise, the new position is set to the previous one $X(t+1) = X(t)$ (in other words, the position X(t) is repeated in the chain). The M-H algorithm converges (as $t \to \infty$) to a stationary set of samples from the distribution but there are many algorithms with faster convergence and varying levels of implementation difficulty. Faster convergence is preferred because of the reduction of computational cost due to the smaller number of likelihood computations necessary to obtain the equivalent level of accuracy. The inverse convergence rate can be measured by the autocorrelation function and more specifically, the integrated autocorrelation time. This quantity is an estimate of the number of steps needed in the chain in order to draw independent samples from the target density. A more efficient chain has a shorter autocorrelation time.

The stretch move {9} proposed an affine-invariant ensemble sampling algorithm informally called the "stretch move." This algorithm significantly outperforms standard M-H methods producing independent samples with a much shorter autocorrelation time. This method involves simultaneously evolving an ensemble of K walkers $S = \{X_k\}$ where the proposal distribution for one walker k is based on the current positions of the $K-1$ walkers in the complementary ensemble $S_{[k]} = \{X_j, \forall_j \neq k\}$. Here, "position" refers to a vector in the N-dimensional, real-valued parameter space. To update the position of a walker at position X_k, a walker X_j is drawn randomly from the remaining walkers $S_{[k]}$ and a new position is proposed:

$$X_k(t) \to Y = X_j + Z[X_k(t) - X_j] \tag{4}$$

where Z is a random variable drawn from a distribution $g(Z = z)$. It is clear that if g satisfies

$$g(z^{-1}) = zg(z), \tag{5}$$

the proposal of Eq. (4) is symmetric. In this case, the chain will satisfy detailed balance if the proposal is accepted with probability

$$q = \min\left(1, Z^{N-1}\frac{p(Y)}{p(X_k(t))}\right), \tag{6}$$

where N is the dimension of the parameter space. This procedure is then repeated for each walker in the ensemble in series. {9} advocate a particular form of $g(z)$, namely

$$g(z) \propto \begin{cases} \frac{1}{\sqrt{z}} & \text{if } z \in \left[\frac{1}{a}, a\right] \\ 0 & \text{otherwise} \end{cases} \tag{7}$$

where a is an adjustable scale parameter that {9} set to 2. It is tempting to parallelize the stretch move algorithm by simultaneously advancing each walker based on the state of the ensemble instead of evolving the walkers in series, see Fig. 8. Unfortunately, this subtly violates detailed balance. Instead, we must split the full ensemble into two subsets ($S(0) = \{X_k, \forall_k = 1, \ldots, K/2\}$ and $S(1) = \{X_k, \forall_k = K/2+1, \ldots, K\}$) and simultaneously update all the walkers in $S(0)$ based only on the positions of the walkers in the other set ($S(1)$). Then, using the new positions $S(0)$, we can update $S(1)$. In this case, the outcome is a valid step for all of the walkers. The performance of this method – quantified by the autocorrelation time – is comparable to the serial stretch move algorithm but the fact that one can now take advantage of generic parallelization makes it extremely powerful.

The autocorrelation time is a direct measure of the number of evaluations of the posterior PDF required to produce independent samples of the target density. {9} show that the stretch-move algorithm has a significantly shorter autocorrelation time on several non-trivial densities. This means that fewer PDF computations are required – compared to a M-H sampler – to

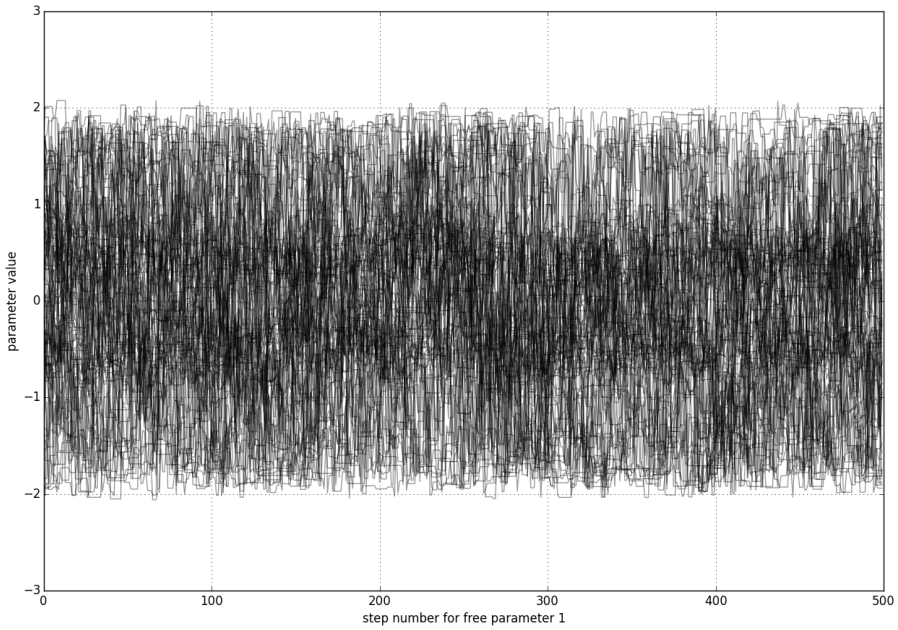


Figure 8: The figure shows the positions of each walker as a function of the number of steps in the chain.

produce the same number of independent samples. The autocovariance function of a time series $X(t)$ is

$$C_f(T) = \lim_{t\to\infty} \mathrm{cov}[f(X(t+T)), f(X(t))]. \tag{8}$$

This measures the covariances between samples at a time lag T. The value of T where $C_f(T) \to 0$ measures the number of samples that must be taken in order to ensure independence. In particular, the relevant measure of sampler efficiency is the integrated autocorrelation times

$$\tau_f = \sum_{T=-\infty}^{\infty} \frac{C_f(T)}{C_f(0)} = 1 + 2\sum_{T=1}^{\infty} \frac{C_f(T)}{C_f(0)}. \tag{9}$$

In practice, one can estimate $C_f(T)$ for a Markov chain of M samples as

$$C_f(T) \approx \frac{1}{M-T} \sum_{m=1}^{M-T} \left[f(X(T+m)) - \langle f\rangle\right]\left[f(X(m)) - \langle f\rangle\right]. \tag{10}$$

We advocate for the autocorrelation time as a measure of sampler performance for two main reasons. First, it measures a quantity that we are actually interested in when sampling in practice. The longer the autocorrelation time, the more samples that we must generate to produce a representative sampling of the target density. Second, the autocorrelation time is affine invariant. Therefore, it is reasonable to measure the performance and diagnose the convergence of the sampler on densities with different levels of anisotropy.

8.8 Interval-Nested-Sampling (INS)

advantages	disadvantages
non-derivative method	difficult to parallelize
investigates the landscape of optimization function	
provides posterior proportional weights for parameter vectors	
finds global minimum	
finds multiple minima	

The Interval-Nested sampling algorithm is efficient technique for estimation of parameter values and their uncertainties which based on Nested sampling algorithm and uses an interval method for definition of a next part of prior volume. The INS algorithm implements the branch-and-bound algorithm {13} to find the next part of prior volume. The main principle of the interval method is a division of the parameter space on interval boxes and an estimation of the optimization function value over the boxes. The estimation is called *inclusion function* and can be calculated by various methods. The centered form of inclusion with slopes is used in the current version of the INS algorithm {14}. The interval method assists to find the next part of prior volume by current calculation of the ratio of volume of the working interval box to the whole volume of the parameter space. The INS algorithm is capable of handling the following complications: multimodality of optimization function, phase transitions, and strong correlations between model parameters.

The algorithm uses the critical element of the volume to form a list with best interval boxes for output results. If the volume of interval box is less than the value defined in tag `<vol_bound>` then the interval box is added to the list with best boxes. The difference between maximal and minimal value of inclusion function plays the role of the stopping criterion.

8.9 Error estimation

MAGIX provides an error estimation for each single parameter at the point of minimum using different methods:

8.9.1 Error estimation using Fisher matrix

This error estimation technique is based on the central-limit theorem which assumes that any well-behaved likelihood function is asymptotically Gaussian near its minimum. Assuming that $\log \mathcal{L} = \log \mathcal{L}(\vec{\vartheta})$ is a function of P parameters $\vec{\vartheta} = \{\vartheta_1, \ldots, \vartheta_P\}$ we expand this function in a Taylor series around the maximum $\vec{\vartheta}_{\max}$ to second order and get

$$\log \mathcal{L}(\vec{\vartheta}) \approx \log \mathcal{L}(\vec{\vartheta}_{\max}) + \frac{1}{2} \left. \frac{\partial^2 \log \mathcal{L}}{\partial\vartheta_i \partial\vartheta_j} \right|_{\vec{\vartheta}_{\max}} (\vartheta - \vartheta_{\max})_i \, (\vartheta - \vartheta_{\max})_j \, . \tag{11}$$

Close to the maximum $\vec{\vartheta}_{\max}$, the linear term vanishes, so that $\log \mathcal{L}(\vec{\vartheta})$ is approximately quadratic in $\vec{\vartheta}$, i.e. $\mathcal{L}(\vec{\vartheta}) = e^{\log \mathcal{L}(\vec{\vartheta})}$ is a Gaussian. The goodness of this approximation scales with the distance to the maximum. But for error estimation we cannot go arbitrarily close to the maximum. Fig. 9 describes two cases: In panel (a) the central-limit theorem gives a reasonable approximation of the likelihood and the error estimation is useful. In panel (b) the Gaussian is not a good approximation and the corresponding error estimates are useless.

In the following we will assume, that we can apply the central-limit theorem and express the likelihood $\mathcal{L}(\vec{\vartheta})$ as described by {10} as

$$\mathcal{L}(\vec{\vartheta}) = \frac{1}{(2\pi)^{P/2} \sqrt{\det \hat{\Sigma}^{-1}}} \exp\left[-\frac{1}{2} \left(\vec{\vartheta} - \vec{\vartheta}_0\right)^T \cdot \hat{\Sigma}^{-1} \cdot \left(\vec{\vartheta} - \vec{\vartheta}_0\right) \right], \tag{12}$$

where $\vec{\vartheta}$ describes a parameter vector for a given model function and $\vec{\vartheta}_0$ the vector of the (local) minimum of the χ^2 function. Eq. (12) describes a P-dimensional (P-variate) Gaussian with

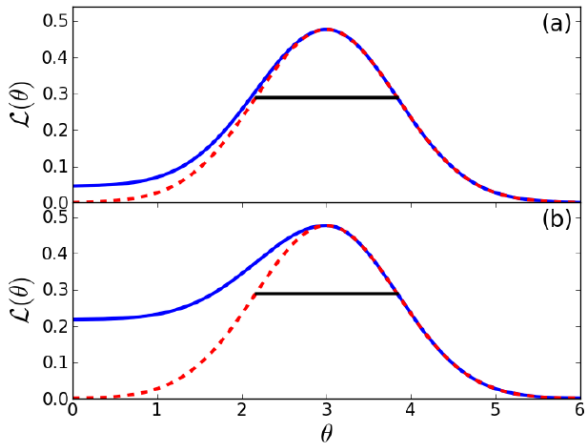


Figure 9: (Taken from {2}): Possible failures of the central-limit theorem. This figure shows an example likelihood function $\mathcal{L}(\vartheta)$ (solid blue curves), its Gaussian approximation at the maximum (dashed red curves), and the widths of these Gaussian (horizontal solid black lines). In panel (a) the Gaussian approximation and the corresponding error estimate is useful, whereas in panel (b) the error estimate is substantially underestimated.

mean $\vec{\vartheta}_0$ and covariance matrix $\hat{\Sigma}$. This matrix describes the desired error estimates of $\vec{\vartheta}_0$. The variance estimates of each parameter ϑ_0^p is described by the diagonal entries of $\hat{\Sigma}$, whereas the off-diagonals are the estimates of the covariances. Comparing Eqs. (11) and (12) we see

$$\hat{\Sigma} = \left(-\frac{\partial^2 \log \mathcal{L}}{\partial \vartheta_i \partial \vartheta_j} \right)^{-1} . \tag{13}$$

The matrix of second derivatives of $\log \mathcal{L}$ is called *Fisher-matrix* or *Fisher information matrix*. If the second derivatives of $\log \mathcal{L}$ can be calculated, this method is order of magnitudes faster than all the other methods, described below, and for some high-dimensional problems the only applicable error estimation method. But this method can only describe elliptical error contours, i.e. it is not possible to obtain banana-shaped error contours. Additionally, this method assumes that the second-order Taylor expansion of Eq. (11) is a good approximation, see Fig. 9. A valid covariance-matrix $\hat{\Sigma}$ has to be positive definite, i.e. $\vec{x}^T \cdot \hat{\Sigma} \cdot \vec{x} > 0$, for any nonzero vector $\vec{x}$. In order to check this, one can use the following tests:

1. Compute the determinant $\det \hat{\Sigma}$. If $\det \hat{\Sigma} \leq 0$, $\hat{\Sigma}$ is not valid.
2. Compute the eigenvalues of the matrix $\hat{\Sigma}$. If any eigenvalue is negative or zero, $\hat{\Sigma}$ is not valid.

Unfortunately, these tests are only rule-out criteria. If $\hat{\Sigma}$ fails any of these two tests, it is clearly ruled out, i.e. the central limit theorem can not be applied. But even if $\hat{\Sigma}$ passes both tests, the Gaussian might not be a good description of the likelihood around the maximum.

Finally, the error (or *marginal error*) $\Delta\vartheta_i$ of a parameter ϑ_i is than given as

$$\Delta\vartheta_i = \sqrt{\left(\hat{\Sigma}\right)^{-1}_{i,i}}, \tag{14}$$

where $\left(\hat{\Sigma}\right)^{-1}_{i,i}$ describes the ith diagonal element of the inverse of the covariance matrix $\hat{\Sigma}$.

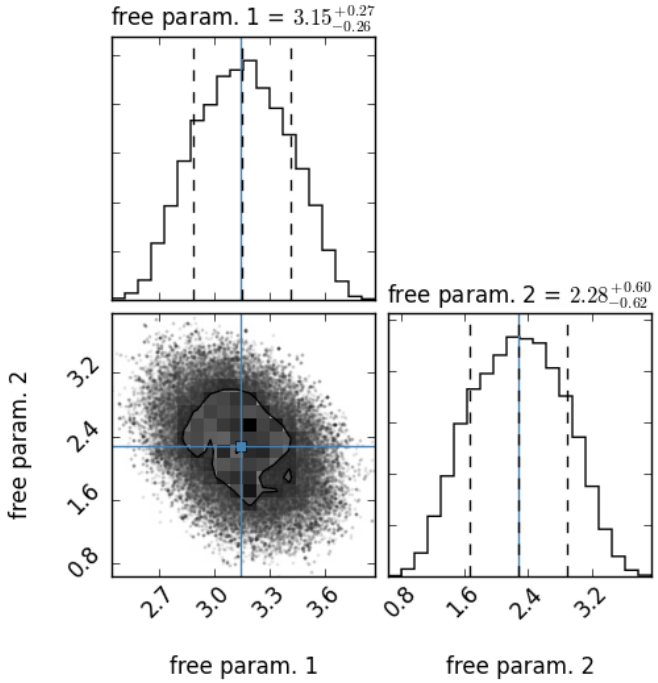


Figure 10: Example of a so-called *corner plot* created with the `corner` package. On top of each column the probability distribution for each free parameter is shown. The left and right dashed lines indicates the lower and upper limit of the corresponding HPD interval, respectively. The dashed line in the middle indicates the mode of the distribution. The blue lines indicate the parameter values of the best fit, calculated in previously applied optimization algorithm(s). The plot in the lower left corner describes the projected 2D histograms of two parameters. The contours indicate the HPD region.

8.9.2 Error estimation using Markov chain Monte Carlo (MCMC)

By choosing the MCMC method, the error estimation algorithm starts an MCMC algorithm at the estimated maximum $\vec{\vartheta}_0$ of the likelihood function and draws M samples of model parameters $\left\{\vec{\vartheta}_1, \ldots, \vec{\vartheta}_M\right\}$ from the likelihood function in a small ball around the a priori preferred position. After finishing the algorithm the probability distribution and the corresponding highest posterior density (HPD) interval of each free parameter is calculated. Additionally, the *corner* package[7] is used to plot each one- and two-dimensional projection of the sample to reveal covariances, see Fig. 10.

Before we describe the calculation of the HPD intervals in detail, we briefly summarize some important statistical expressions.

We start with the cumulative distribution function (CDF) of the standard normal distribution which is given as

$$\Phi(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} e^{-t^2/2}\, dt. \tag{15}$$

In statistics one often uses the related error function, or $\mathrm{erf}(x)$, defined as the probability of a random variable with normal distribution of mean 0 and variance $1/2$ falling in the range $[-x, x]$ that is

$$\mathrm{erf}(x) = \frac{1}{\sqrt{\pi}} \int_{-x}^{x} e^{-t^2}\, dt. \tag{16}$$

[7] https://pypi.python.org/pypi/corner

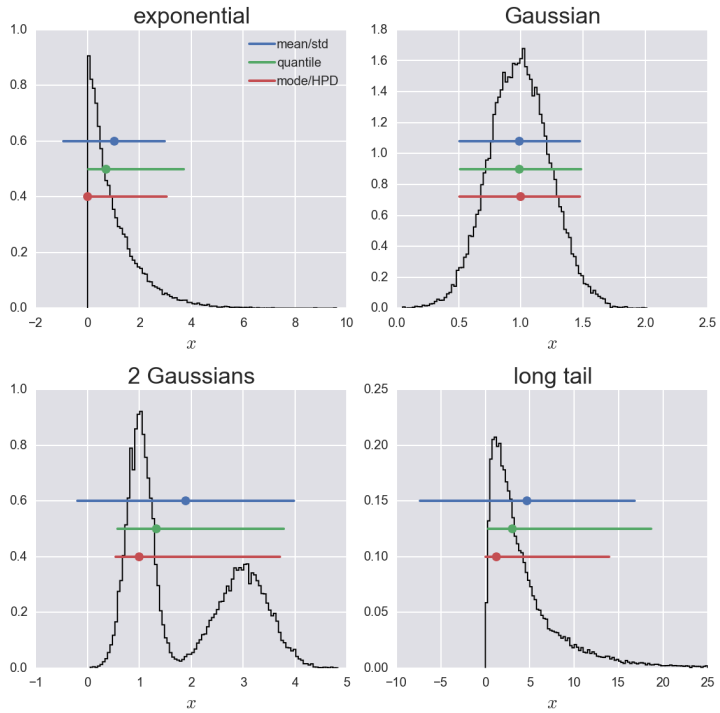


Figure 11: Three commonly used ways of plotting a value plus error bar for a 95 % credible region for four different probability distributions[8]: 1.) *Mean ± standard deviation (blue line)*: The most commonly used confidence interval is $\mu \pm k\sigma$, where k is chosen to give the appropriate confidence interval, assuming the posterior is Gaussian. Here, $k = 1.96$. 2.) *Median with quantile (green line)*: The posterior need not be Gaussian. If it is not, we would like a more robust way to summarize it. A simple method is to report the median, and then give lower and upper bounds to the error bar based on quantile. For a 95 % credible region we would report the 2.5th percentile and the 97.5th percentile. 3.) *Mode with HPD (red line)*: This method uses the highest posterior density (HPD) interval. If we're considering a 95 % confidence interval, the HPD interval is the shortest interval that contains 95 % of the probability of the posterior.

The two functions are closely related, namely

$$\Phi(x) = \frac{1}{2}\left[1 + \mathrm{erf}(\frac{x}{\sqrt{2}})\right]. \tag{17}$$

For a generic normal distribution f with mean μ and deviation σ, the cumulative distribution function (CDF) for standard score $z = \left(\frac{x-\mu}{\sigma}\right)$ is

$$F(x) = \Phi\left(\frac{x-\mu}{\sigma}\right) = \frac{1}{2}\left[1 + \mathrm{erf}\left(\frac{x-\mu}{\sigma\sqrt{2}}\right)\right]. \tag{18}$$

Following the so-called 68-95-99.7 (empirical) rule, or 3-sigma rule, about 68 % of values drawn from a normal distribution are within one standard deviation σ away from the mean; about 95.4 % of the values lie within two standard deviations; and about 99.7 % are within three standard deviations.

More precisely, the probability that a normal deviate lies in the range $\mu - n\sigma$ and $\mu + n\sigma$ is

[8] Taken from http://bebi103.caltech.edu/2015/tutorials/l06_credible_regions.html

given by

$$P(n) = F(\mu + n\,\sigma) - F(\mu - n\,\sigma) = \Phi(n) - \Phi(-n) = \mathrm{erf}\left(\frac{n}{\sqrt{2}}\right) \approx \begin{cases} 0.682, & n = 1 \\ 0.954, & n = 2 \\ 0.997, & n = 3 \end{cases} . \tag{19}$$

As mentioned above, the error estimation algorithm calculates the highest posterior density (HPD) interval of each parameter, respectively. A HPD interval is basically the shortest interval on a posterior density for some given confidence level, i.e. 68 % for 1σ, 95.4 % for 2σ etc. For example, if we're considering a 95.4 % (or 2σ) confidence interval, the HPD interval is the shortest interval that contains 95.4 % of the probability of the posterior. Mathematically a $100 \cdot (1 - P(n))$ % HPD interval (or region) for a subset $C \in \Theta$ defined by

$$C = \{\partial : \pi(\partial|x \geq k\}\,, \tag{20}$$

where k is the largest number such that

$$\int_{\{\partial:\pi(\partial|x\geq k\}} \pi(\partial|x)d\partial = 1 - P(n). \tag{21}$$

The value k can be thought of as a horizontal line placed over the posterior density whose intersection(s) with the posterior define regions with probability $1 - P(n)$, see Fig. 11. In order to compute a HPD interval we rank-order the MCMC trace. We know that the number of samples that are included in the HPD is 0.954 (or another confidence level) times the total number of MCMC sample. We then consider all intervals that contain that many samples and find the shortest one. In the case of a normal distribution an HPD interval coincides with the usual probability region symmetric about the mean, spanning the $\frac{n\sigma}{2}$ and $1 - \frac{n\sigma}{2}$ quantiles[9]. The same is true for any unimodal, symmetric distribution, see Fig. 11.

So, the error estimation algorithm reports the (first) mode[10] and then the bounds on the HPD intervals, see Fig. 10. Note, the mode must not coincidence with the best fit result of the previously applied algorithms!

Please note, the calculated HPD intervals are credible intervals not confidence intervals. In Bayesian statistics, a credible interval is an interval in the domain of a posterior probability distribution or predictive distribution used for interval estimation. The generalization to multivariate problems is the credible *region*. Credible intervals are analogous to confidence intervals in frequentist statistics, although they differ on a philosophical basis; Bayesian intervals treat their bounds as fixed and the estimated parameter as a random variable, whereas frequentist confidence intervals treat their bounds as random variables and the parameter as a fixed value. (Taken from https://en.wikipedia.org/wiki/Credible_interval).

8.9.3 Error estimation using Interval Nested Sampling (INS)

The error estimation module of MAGIX is based on Interval Nested Sampling (INS) algorithm. The INS algorithm uses Bayesian approach with interval methods to obtain a set of physical

[9] In statistics and the theory of probability, quantiles are cut points dividing the range of a probability distribution into contiguous intervals with equal probabilities, or dividing the observations in a sample in the same way. There is one less quantile than the number of groups created. Thus quartiles are the three cut points that will divide a dataset into four equal-size groups. q-Quantiles are values that partition a finite set of values into q subsets of (nearly) equal sizes. There are q - 1 of the q-quantiles, one for each integer k satisfying 0 < k < q. In some cases the value of a quantile may not be uniquely determined, as can be the case for the median (2-quantile) of a uniform probability distribution on a set of even size. (Taken from https://en.wikipedia.org/wiki/Quantile)

[10] Generally, the *mode* is the value that appears most often in a set of data. For a normal distribution the numerical values of mode, mean, and median are identical, and may be very different in highly asymmetric distributions.

parameters $\Theta = (\partial_1; \partial_2, \ldots, \partial_n)$ of the model M which attempts to describe the experimental data D. The assumption of the Bayesian analysis is to incorporate the prior knowledge with a given set of current observations in order to make statistical inferences. The prior information $\pi(\Theta)$ could come from observational data or from previous experiments. Bayes' theorem states that the posterior probability distribution of the model parameters is given by:

$$\Pr(\Theta) = \frac{L(\Theta\pi(\Theta}{Z},$$

where $\Pr(\Theta)$ is the posterior probability distribution of the model parameters; $L(\Theta)$ is the likelihood of the data for the given model and its parameters; $\pi(\Theta)$ is a prior information, and Z is Bayesian evidence. The Bayesian evidence is the average likelihood of the model in the parameter space. It is given by the following integral over the n-dimensional space:

$$Z = \int L(\Theta)\pi(\Theta)d(\Theta). \tag{22}$$

The Nested Sampling (NS) algorithm (§8.3) transforms the integral (22) to the single dimension by re-parametrization to a new variable - a *prior volume* X. The volume of parameter space can be divided into elements $dX = \pi(\Theta)d\Theta$. The prior volume X can be accumulated from its elements dX in any order, so we construct it as a function of decreasing likelihood:

$$X(\lambda) = \int_{L(\Theta)>\lambda} \pi(\Theta)d(\Theta).$$

That means that the cumulative prior volume covers all likelihood values greater than λ. As λ increases, the enclosed volume X decreases from $X(0) = 1$ to $X(1) = 0$. If the prior information $\pi(\Theta)$ is uniformly distributed in the parameter space, then (22) the equation for the evidence transforms into

$$Z = \int_0^1 L(X)dX.$$

One can calculate the partial likelihood as $L_i = L(X_i)$, where the X_i is a sequence of decreasing values, such that

$$0 < X_m < \ldots < X_2 < X_1 < 1.$$

Then the evidence is defined by the trapezoid rule

$$Z = \sum_{i=1}^{m} Z_i, \quad \text{where} \quad Z_i = L_i \frac{X_{i-1} - X_{i+1}}{2}.$$

The Nested Sampling algorithm is targeted at the calculation of Bayesian evidence, but it assists to obtain a posterior sample of points from which one can estimate uncertainties of parameter values. Inferences of classical NS algorithm are calculated in probabilistic sense, because the prior volume X_i, which corresponds to the likelihood contour L_i, is usually evaluated with a random number generator.

The Interval Nested Sampling algorithm included in MAGIX is an implementation of the branch-and-bound algorithm {13} to construct a sequence of decreasing prior volume X_i. The main principle of the interval method is a division of the parameter space into interval boxes and an estimation of the optimization function value over the boxes. The estimate is called

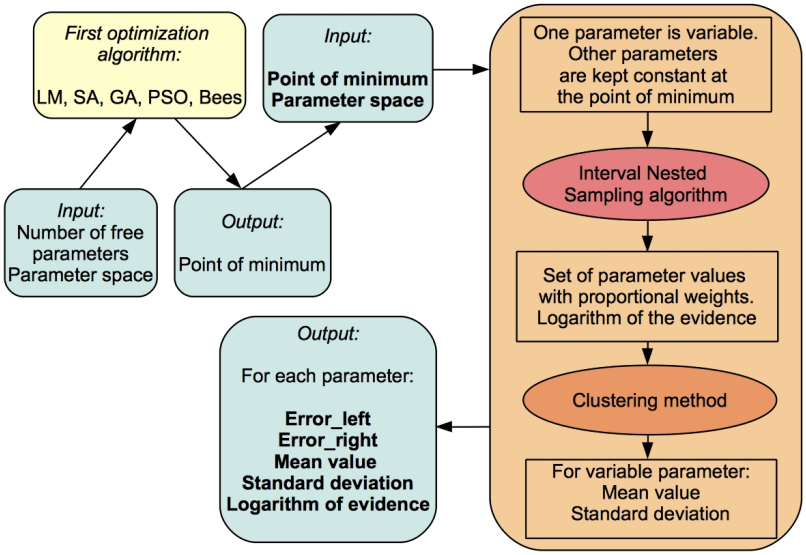


Figure 12: Schematic diagram of error estimation module of MAGIX.

inclusion function and can be calculated by various methods. The centered form of inclusion with slopes is used in the current version of the Interval Nested Sampling algorithm {14}. The interval method assists in finding the subsequent prior volume X_i by determining the ratio of the volume of the working interval box Z_i to the whole volume Z of the parameter space. Posterior weights of points after the Nested Sampling process are calculated using Bayes' theorem:

$$w_i = \frac{Z_i}{Z}. \tag{23}$$

Using the sequence of posterior samples of parameter vectors we are able to determine the reliability of model parameters such as standard deviations or to construct posterior distributions of parameter values. The mean value $\mu(\partial_j)$ of each model parameter $\partial_j, j = (1, \ldots, n)$ and its standard deviation $\sigma(\partial_j)$ are given as:

$$\mu(\partial_j) = \sum_{i=1}^{k} w_i \partial_j \tag{24}$$

and

$$\sigma(\partial_j) = \sqrt{\sum_{i=1}^{k} w_i \left(\partial_j - \mu(\partial_j)\right)^2}, \tag{25}$$

where k indicates the number of points in the sample. The Interval Nested Sampling algorithm is capable of handling multi-modality of the optimization function, phase transitions, and strong correlations between model parameters.

The error estimation module of MAGIX is assigned in a manner such that an error estimation is calculated for each single parameter at the point of minimum. The schematic diagram of the error estimation module is shown in fig. 12. After some optimization procedure MAGIX determines a point of minimum. The input values for the error estimation are given by the point

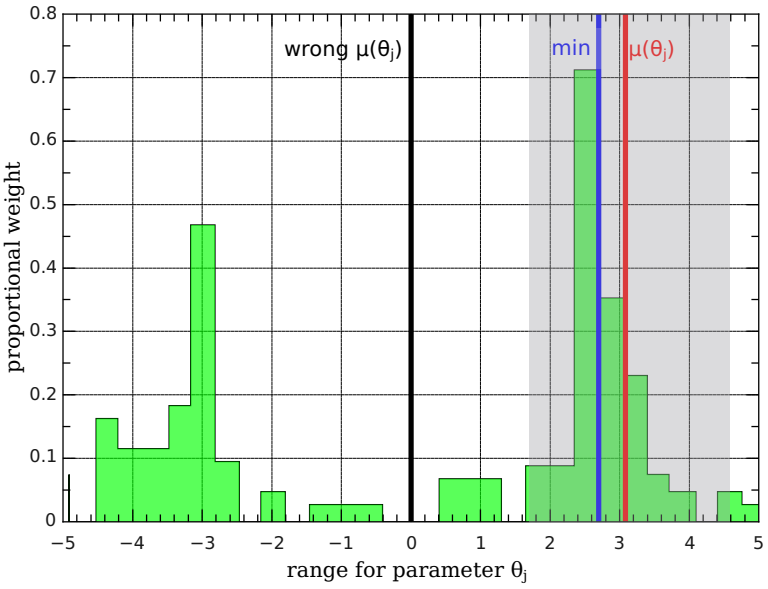


Figure 13: Distribution of parameter values with several minima where a direct application of Eqn. (24) - (25) is not possible. (Here, $\mu(\partial_j)$ indicates the mean value $\mu(\partial_j)$, $\sigma(\partial_j)$ is the standard deviation $\sigma(\partial_j)$, and "min" represents the value of the parameter ∂_j of the best fit result).

of minimum and the parameter space. In order to determine the error of a parameter ∂_j at the minimum, MAGIX varies this parameter within the given parameter range, whereas the other parameters are kept constant. The Interval Nested Sampling algorithm is applied and returns a set of parameter values with proportional weights and the logarithm of the Bayesian evidence for parameter ∂_j for a sequence of parameter values distributed over the whole parameter space. If the distribution of parameter values has only one minimum, then Eqn. (24) - (25) can be applied to calculate the mean value and the standard deviation. Sometimes there are several minima in the sequence, hence these formulas can not be used directly, because the resulting estimation of the mean value and standard deviation produces meaningless results (see fig. 13).

We are interested in the uncertainty around the considered minimum. Therefore, we need to estimate the mean value and the standard deviation in the minimum. This is done using a clustering method:

- Calculate the distances from the minimum to all points of the sample.
- Sort the points depending on their distances (ascending order).
- Select the points with function value less than the χ^2 boundaries for the 99 percent confidence region $\Delta\chi^2_{a,n}$ (a = 0.99).

Finally, the new sample of m points ($m < k$) is distributed around the minimum point (Fig. 13, gray box) and the mean value of the parameter ∂_j is calculated as follows:

$$\mu\left(\partial_j\right) = \frac{\sum_{i=1}^{m} w_i \partial_j}{\sum_{i=1}^{m} w_i}.$$

In Bayesian statistics one usually applies a credible interval (or Bayesian confidence interval) to the parameter value. In the case of a single parameter and a sample of points which can

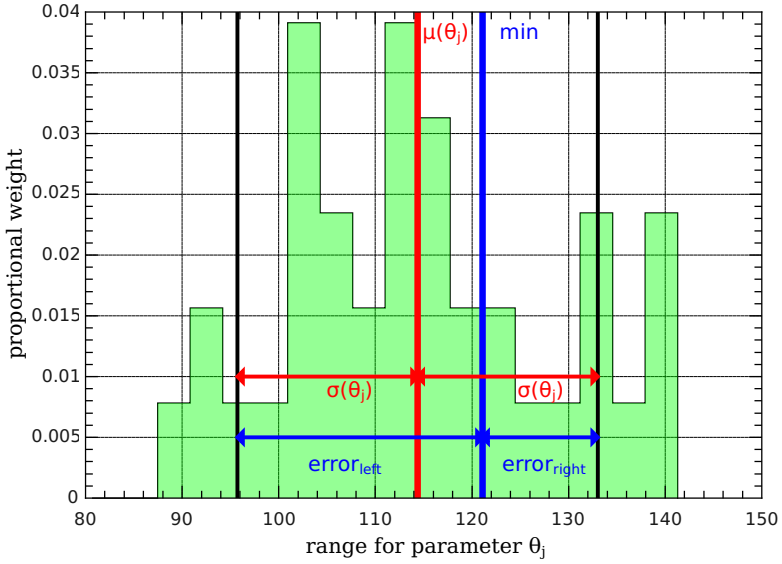


Figure 14: Schematic diagram of error estimation module of MAGIX. (Here, $\mu(\partial_j)$ indicates the mean value $\mu(\partial_j)$, $\sigma(\partial_j)$ represents the standard deviation $\sigma(\partial_j)$, and "min" the value of the parameter ∂_j of the best fit result).

be summarized in a single sufficient statistic, it can be shown {15} that the credible interval and the confidence interval coincide, if the unknown parameter is a location parameter (i.e. the forward probability function has the form $Pr(\partial_i|\mu(\partial_i)) = f(\partial - \mu)$) with a uniform prior distribution. Hence credible intervals in our case are analogous to confidence intervals in frequentist statistics. Using the mean value $\mu(\partial_j)$ and the standard deviation $\sigma(\partial_j)$, the 3σ confidence interval of the parameter ∂_j is given by:

$$Pr(\mu(\partial_j) - \sigma(\partial_j) < \partial_j < \mu(\partial_j) + \sigma(\partial_j)) \approx 0.99.$$

In Fig. 14 the example of a histogram of distribution of parameter values after error estimation is shown. Hence for the parameter value at the point of minimum (using the error left and the error right):

$$Pr(\partial_i(\text{min}) - \text{error}_{\text{left}} < \partial_j < \partial_i(\text{min}) + \text{error}_{\text{right}}) \approx 0.99.$$

The Bayesian evidence plays usually an important role in model selection but in parameter estimation the evidence factor is ignored because it is an integrate value over the whole parameter space. The Interval Nested Sampling algorithm calculates the logarithm of the evidence and it can be used to estimate the quality of the fitting procedure. A large absolute value of the logarithm of the evidence indicates a big uncertainty of the parameters.

8.10 Additional packages

This package makes the following algorithms included in the scipy package {16} available in MAGIX:

1. "fmin": Minimize a function using the downhill simplex algorithm. This algorithm only uses function values, not derivatives or second derivatives.

2. “fmin_powell”: Minimize a function using modified PowellâĂŹs method. This method only uses function values, not derivatives.
3. “fmin_cg”: Minimize a function using a nonlinear conjugate gradient algorithm.
4. “fmin_bfgs”: Minimize a function using the BFGS algorithm.
5. “fmin_ncg”: Unconstrained minimization of a function using the Newton-CG method.
6. “fmin_l_bfgs_b”: Minimize a function func using the L-BFGS-B algorithm.
7. “fmin_tnc”: Minimize a function with variables subject to bounds, using gradient information in a truncated Newton algorithm.
8. “anneal”: Minimize a function using simulated annealing.
9. “brute”: Minimize a function over a given range by brute force.

8.11 Conclusions

It is possible to give some general directives of how and when to use each algorithm and a combination of more than one.

- The algorithms that result to a number of best sites (NS, PSO, bees; they include a `BestSiteCounter` tag) can be used as the first algorithm in a chain of algorithms specified in the fit control file. The second algorithm in the chain could be one of those algorithms that are not global optimizers (LM, SA), so that the best parameter sets found by the first algorithm is better worked on locally.
- MC external model programs should avoid being used with derivative algorithms. (The only derivative algorithm currently available is LM.) Actually, the real problem is more general and has to do with programs that use random number generators. Randomly selected numbers could produce very high derivatives, and might therefore make a derivative algorithm diverge.
- If we have at our disposal an adequate number of processors and we want to use a swarm algorithm, it is desirable that the number of processors is set proportional to the number of particles.

  For example: If we use the bees algorithm with 6 bees, the ideal would be that there were 6 processors to follow the path of each bee. If the available processors are 4, we don’t gain anything in real time if we use 4 processors instead of 3: In the first round the 4 processors will be used for the first 4 bees. In the second round 2 processors will be used for the last 2 bees; during the second round the 2 remaining processors will be doing nothing. The second round will need the same time to finish as if it were performed by 3 processors and 3 bees.

  But the number of particles is not something that the user can define before MAGIX starts running, as it is internally calculated depending on the case. The only thing the user can do in this direction is to use as many processors as possible.
- It is possible that an algorithm appears more than once in a chain, but it’s not advisable. We recommended that a algorithm chain does not contain any algorithm more than once: It makes no sense, and additionally the output files of each algorithm would be overwritten.

If we want to debug such cases, a number could be appended to file name even if no sites exist. Then for a new file name, MAGIX would search if such a file exist; if yes, number should be increased by 1 and MAGIX should check again; if not, MAGIX should create such a file (see §6).

- The Error-Estimator using the Interval-Nested sampling algorithm has to be used in a chain, after another algorithm.

A Appendix

A.1 Terminology

- names referred to in this text:

CATS	Coherent set of Astrophysical Tools for Spectroscopy
MAGIX	Model and Analysis Generic Interface for eXternal numerical codes
GIMP	GNU Image Manipulation Program
GTK	GIMP ToolKit
GUI	Graphical User Interface

- terms concerning MAGIX and the fitting process:

external (model) program: The program whose parameters are to be fitted by MAGIX.
model (fit) function: The function that comes along for each one of the dependent variables of the program, after the parameters have been optimized
function call: The action of – the numerical algorithm(s) used by – MAGIX calling the external program
description: The registration file contains descriptions of input and output files. The XML description of each input file of the external program includes descriptions of all the lines of the given input file. The XML description of each line contains descriptions of the parameters declared in the given line. So, the word description is not identical to the registration file, but, instead the registration file consists of XML descriptions of files, lines and parameters. Note that within this text, there is another description that is mentioned: the descriptions of experimental/observational data files within the experimental XML file.
ranges: The word ranges always refers to ranges of the independent variable(s), i.e. the X column(s) (§3.3).

We refer to data ranges to mean ranges of experimental/observational data that will be loaded, i.e. ranges of data from each one of the experimental data files that are specified in the experimental XML file with the use of the `<NumberExpRanges>`, `<MinExpRange>` and `<MaxExpRange>` tags (§3.2).

limits: We use the word limits to mean upper and lower limits for the parameters whose values are to be fitted.

- algorithm names and abbreviations:

Levenberg–Marquardt	algorithm (LM):	§8.1
simulated annealing	algorithm (SA):	§8.2
nested sampling	algorithm (NS):	§8.3
particle swarm optimization	algorithm (PSO):	§8.4
bees	algorithm:	§8.5
genetic	algorithm (GA):	§8.6
interval-nested sampling	algorithm (INS):	§8.8
Gauss–Newton	algorithm (GN):	§8.1
Newton's	method:	§8.1
gradient descent	algorithm:	§8.1
Monte Carlo	(MC) methods:	§8.3
Markov chain Monte Carlo	(MCMC) methods:	§8.7

- other abbreviations

Figure 15: The symbols used in the flow charts included in this document.

CDMS	Cologne Database for Molecular Spectroscopy
JPL	Jet Propulsion Laboratory
ALMA	Atacama Large Millimeter Array
CASA	Common Astronomy Software Applications
API	Application Program Interface
OMP, OpenMP	Open Multi-Processing (API for explicit multi-threaded shared memory parallelism)
MPI	Message Passing Interface (API for processes that communicate with each other by sending and receiving messages)

- models that have already been registered in MAGIX are:

SimLine	model that computes profiles of molecular lines
RATRAN	model that computes profiles of molecular lines
myCloud	model that computes 3D data cubes of spectral ranges with arbitrary input geometries and calculates the corresponding radiation transfer

Meanings of symbols used in flow charts are explained in fig. 15; those symbols comply with the standard typology of design.

A.2 XML rules (that's a noun, not a verb!)

A.2.1 General rules for the tags of the XML files

- The XML files should better start with the following line:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
```

  This is a directive to show that this file is an XML file, and should better be there when the file is to be read e.g. by a browser.
- Do not mix upper and lower case of the characters contained in the names of the XML tags. E.g. `<MinExpRange>` and NOT `<minexprange>`.
- MAGIX does not perform any validation of the XML files with the corresponding schema. Additionally, the tags that are expected to be read are searched for by their exact name. So it should be made sure that the tag names are correct and their values as expected (integer, float or string) depending on the case (although all the numbers are all kind of transformed to float numbers in MAGIX, and are rounded if their are supposed to be integers).
- The `<description>` tags are not read by MAGIX and can contain arbitrary information.

A.2.2 Parallel computation

In a program there is always a part of it that cannot but run serially. This part of the code will always be processed by one processor, no matter the number of processors that are available.

If there is only one processor available, then the whole program would run by this single processor in $t(1)$. Let f be the fraction of $t(1)$ that cannot be performed in parallel, so it has to be performed serially, no matter the number of processors available. Then the fraction of $t(1)$ that can be performed in parallel will be $1-f$ (but it is still performed serially if we have only one processor).

If we have p processors, then the time that will need to run the whole program consists of two parts:

- the part that anyway has to be performed serially by one processor; corresponds to CPU time equal to $ft(1)$
- the part that can be performed in parallel, with the use of p processors; this corresponds to CPU time equal to $(1-f)t(1)$; this has to be divided by the number of processors, if we want to see how much time we gain with the use of more processors

So the use of p processors would require time equal to:

$$t(p) = ft(1) + \frac{(1-f)t(1)}{p} \quad \Rightarrow \quad \frac{t(p)}{t(1)} = \frac{(p-1)f+1}{p}$$

We see that as $f \to 0$, we tend to have a 1:1 gain in time as we increase the number of processors. On the other hand, as $f \to 1$, we tend to not gain anything by increasing the number of processors.

Of course, the estimation of the f factor is the most difficult part in calculating if it really makes a difference to run our program in parallel, and which is the ideal number of processors to use. The estimation of the f factor requires a very good knowledge of the program's structure. In any case the accuracy with which f is estimated decreases with the complexity of the structures contained in the code, especially with `if` or `switch` blocks that do not consist of the same computational needs.

References

[1] Wikipedia. "" http://en.wikipedia.org/wiki/Main_Page, call December 30th 2011.

[2] Andrae, R. 2010 (arXiv: 1009.2755)

[3] Bees algorithm: The official web site. http://www.bees-algorithm.com, call December 30th 2008.

[4] Allan Birnbaum. On the foundations of statistical inference. Journal of the American Statistical Association, 57(298):269–306, June 1962. ISSN 01621459. URL http://www.jstor.org/stable/2281640, call December 30th 2011.

[5] George Casella. Bayesians and frequentists: Models, assumptions and inference. 2008. ACCP 37th Annual Meeting, Philadelphia, PA.

[6] Shu-Kai S. Fan and Erwie Zahara. A hybrid simplex search and particle swarm optimization for unconstrained optimization. European Journal of Operational Research, 181(2): 527–548, 2007.

[7] F. Feroz and M. P. Hobson. Multimodal nested sampling: an efficient and robust alternative to Markov Chain Monte Carlo methods for astronomical data analyses. MNRAS, 384: 449–463, February 2008.

[8] Foreman-Mackey, D., Hogg, D.W., Lang, D. & Goodman, J. 2012 (arXiv: 1292.3665)

[9] Goodman, J. & Weare, J., 2010, Comm. App. Math. Comp. Sci., 5, 65

[10] Heavens, A. 2009 (arXiv: 0906.0664v3)

[11] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. Numerical recipes in FORTRAN. The art of scientific computing. Cambridge University Press, 2nd edition, 1992.

[12] John Skilling. http://www.inference.phy.cam.ac.uk/bayesys, call December 30th 2004.

[13] K. Ichida, Y. Fujii. An interval arithmetic method for global optimization. Computing **23**, 85-97 (1979)

[14] R. Krawczyk. Centered forms and interval operators. Computing **34**, 243-259 (1985)

[15] E. T. Jaynes. Confidence Intervals vs Bayesian Intervals, in Foundations of Probability Theory, Statistical Inference, and Statistical Theories of Science, (W. L. Harper and C. A. Hooker, eds.) Dordrecht: D. Reidel, p. 175 (1976)

[16] http://docs.scipy.org/doc/scipy/reference/optimize.html, call January 23rd 2012.