Fittino Documentation

Release 2-0-0

Mathias Uhlenbrock

April 23, 2016

CONTENTS

CHAPTER ONE

INTRODUCTION

Fittino is a software framework written in C++, whose main focus is to extract information about physical model parameters that are not directly accessible through experimental observation. The underlying working principle is to find the set of parameters for which the model makes the “best” predictions about the outcome of a number of physics experiments. For that, the model parameters are varied and the model predictions are compared with the measured values repeatedly, until at some point the optimal matching set can be derived. To this end, Fittino offers various powerful optimization and sampling algorithms, next to a number of tools to assist with the statistical interpretation. The main task of the framework, however, is to provide simple and flexible interfaces in order to allow the user to include a great number of third-party theory packages in his or her analysis.

The code is largely developed by a team at Bonn University, but has ever since benefited from contributions by members of other universities and institutes as well. Historically, the physics program of Fittino has concentrated on determining the parameters of supersymmetric models such as the constrained Minimal Supersymmetric Standard Model (cMSSM). In recent times, however, detailed studies of the parameters of the Higgs boson sector have been performed in addition.

Since its first release, the software has been subjected to major adoptions and expansions which altogether justify the assignment of a new version number. It is the purpose of this document to outline the most relevant changes to the experienced user and at the same time provide newcomers with enough information to get quickly acquainted.

The document’s body is divided into three main parts. The *Installation guide* informs the reader about how to obtain the Fittino source code and which additional programs and libraries are necessarily required in order to build and link the individual software components. It also contains a list of third-party software which is already supported to work together with Fittino. The *User guide* gives instructions on the configuration and the basic usage of the program. Finally, the developer-guide is intended to describe the software architecture of Fittino and discusses the most important design decisions in order to help the reader to integrate his or her own custom extension into the framework.

For more information, e.g. about the most recent developments in connection with Fittino, it is recommended to subscribe to one of the available mailing lists. If, for example, you would like to be notified about new code commits, subscribe to the fittino-dev mailing list. In case you are more interested about the physics aspects of Fittino, you can subscribe to the fittino mailing list.

CHAPTER

TWO

INSTALLATION GUIDE

At the time of the writing of this guide, no precompiled versions of Fittino are available. With other words, in order to make use of the framework, it has to be built from its sources in advance.

This document provides instructions on how to obtain the Fittino source code from its repository and how to build and install the program. Whereas the core functionality of Fittino depends only on relatively few external packages, one of the framework's main functions is to provide simple and flexible interfaces to a number of third-party theoretical physics programs and make these work together seamlessly. A list of third-party software which is already supported to run with Fittino can be found in a dedicated section. This list is by no means exhaustive and people are encouraged to write custom extensions in order to include their own favorite code. More information on how this is done is compiled in the developer-guide. Lastly, a final section deals with common issues the reader might be faced with in the course of the installation process and provides some troubleshooting guidance.

Fittino is being developed in a Unix-like environment, therefore a certain familiarity with this class of operating systems and the usage of a command line shell is assumed in the following.

2.1 Getting the sources

The Fittino source code can be obtained from its public subversion repository. To check available releases, run

```
svn ls https://svn.physik.uni-bonn.de/basic/fittino/tags
```

To obtain Fittino version X-Y-Z, execute

```
svn export https://svn.physik.uni-bonn.de/basic/fittino/tags/fittino-X-Y-Z
```

To get the latest development version, run

```
svn export https://svn.physik.uni-bonn.de/basic/fittino/trunk
```

Note that this document is compatible only to Fittino version 2-0-0 or higher.

2.2 Prerequisites

The following external programs/libraries are mandatory in order to compile Fittino

- The build system generator CMake (version 3.2 or higher)
- A build tool supported by CMake, e.g. GNU make or Xcode
- A C++ compiler, e.g. provided by the GNU Compiler Collection (gcc) or the Clang project
- The Boost C++ libraries (version 1.47.0 or higher)

- The data analysis framework ROOT (version 5.20.00 or higher)

A number of optional tools is used to simplify working with Fittino. Fittino can be built without these but in that case some functionality is restricted. These include

- The source code formatter astyle (needed to ensure that the code is in accordance with the Fittino coding style, which is specified in the style-guide)
- The documentation generation system doxygen (used to automatically generate a Fittino code reference)
- The XML C parser Libxml2 (needed to validate Fittino input files)

For information on how to install and setup these programs/libraries, please consult the documentation at the individual project's websites. If a program is installed on your system but cannot be found nevertheless, you should check if the corresponding CMake variables are correctly set as described in the *CMake variables* section.

2.3 Building and installing Fittino

The setup of the installation process is managed by CMake. Its main task is to identify and find the installed required and optional software packages so that the Fittino code can be linked to the corresponding libraries. For that, CMake consults dedicated `CMakeList.txt` files in which the dependencies of the individual Fittino modules are specified.

The system paths pointing to the installation of the external software packages are set by CMake's `Find*.cmake` modules. CMake comes already with a number of `Find*.cmake` modules for many widely used programs and libraries (a list of available CMake modules can be obtained by typing `cmake --help-module-list`). The `Find*.cmake` modules that come with Fittino can be found in the `CMakeModules/` directory. How to write a custom `Find*.cmake` module to link Fittino to yet unsupported code is explained in the developer-guide.

It is recommended to build Fittino in a dedicated directory. This helps keeping a clean separation between the source code and the built objects. So e.g. from Fittino's root directory change to

```
$ cd <build_directory/>
```

In order to configure the build process type

```
$ cmake ..
```

To build Fittino type

```
$ make
```

If the compilation is successful, an executable called "fittino" is produced in `<build_directory>/sources/kernel`.

To install Fittino type

```
$ make install
```

This copies the executable to `bin/fittino` and the example input files to `share/fittino/input` in the Fittino installation directory. The default installation directory is `/usr/local/`, which can be changed by setting the `CMAKE_INSTALL_PREFIX` variable (see the *CMake variables* section).

2.4 Third-party software

The following list contains a number of theoretical physics programs which are supported by Fittino. Within the framework, each program corresponds to a (set of) so-called `Calculator (s)`, which can be understood as (a) wrapper class(es) allowing for a unified integration of the different codes and presenting a unified interface. If a user wants to make a program not on this list work together with Fittino, all he or she has to do is to write a `Find*.cmake` module and a corresponding `Calculator` class.

- FeynHiggs: Fortran code for the diagrammatic calculation of the masses, mixings and much more of Higgs bosons in the MSSM at the two-loop level
  Requires:
  - The Fortran compilers gfortran or ifort
- GM2Calc: Precise calculation of the muon anomalous magnetic moment in the MSSM
  Requires:
  - Eigen: A C++ template library for linear algebra
- HepMC: A C++ Event Record for Monte Carlo Generators
- HiggsBounds: HiggsBounds takes a selection of Higgs sector predictions for any particular model as input and then uses the experimental topological cross section limits from Higgs searches at LEP, the Tevatron and the LHC to determine if this parameter point has been excluded at 95% C.L.
  Requires:
  - The Fortran compilers gfortran or ifort
- HiggsSignals: HiggsSignals performs a statistical test of the Higgs sector predictions of arbitrary models (using the HiggsBounds input routines) with the measurements of Higgs boson signal rates and masses from the Tevatron and the LHC
  Requires:
  - The Fortran compilers gfortran or ifort
- Micromegas: A code for the calculation of Dark Matter Properties including the relic density, direct and indirect rates in a general supersymmetric model and other models of New Physics
  Requires:
  - X11: The X Window System
- SuperIso: Calculation of flavour physics observables

The following programs need not to be present at compile-time in order to build Fittino. However, Fittino needs the individually required packages to be present at compile-time. For example, in order to build Fittino with support for SPheno, SLHAea is mandatory. In order to use one of these program at run-time, it needs to be installed and (a pointer to) the executable must exit in the directory from where Fittino is invoked.

- CheckMATE: A tool to easily test simulated event files against LHC results
- Herwig++: Herwig++ is a particle physics event generator, written in C++
- MadGraph: A framework that aims at providing all the elements necessary for SM and BSM phenomenology, such as the computations of cross sections, the generation of hard events and their matching with event generators, and the use of a variety of tools relevant to event manipulation and analysis
- NLLFast: A computer program which computes the squark and gluino hadroproduction cross sections including next-to-leading order (NLO) supersymmetric QCD corrections and the resummation of soft gluon emission at next-to-leading-logarithmic (NLL) accuracy
- SmodelS: A tool for interpreting simplified-model results from the LHC
  Requires:
  - The Python programming language
- SPheno: SPheno stands for S(upersymmetric) Pheno(menology). The code calculates the SUSY spectrum using low energy data and a user supplied high scale model as input
  Requires:

  - SLHAea: An easy to use C++ library for input, output, and manipulation of data in the SUSY Les Houches Accord (SLHA)

A number of smaller projects, to which also members of the Fittino development team are contributing, is hosted in the same repository as Fittino and can be found in the `external/` directory. These are

- CheckVacuum
- HDim6
  Requires:
  - LHAPDF: A general purpose C++ interpolator, used for evaluating PDFs from discretised data files
  - GSL: The GNU Scientific Library

Again, for information on how to install and setup these programs/libraries, it is recommended to consult the documentation at the individual project's websites.

2.5 CMake variables

The behavior of CMake can be influenced by setting dedicated CMake variables. CMake variables can be set using CMake arguments in two ways:

- Definition of the variables on the command line (`-D<VARIABLE_NAME>=<VARIABLE_VALUE>`)
- Specifying an initial cache file (`-C<PATH_TO_FILE>`) in which the variables are set

Many of the available CMake variables such as e.g. software installation paths are set automatically when invoking CMake, without the need of user interaction. If this fails, however, the user might be urged to set the variables by hand (see next section). On the other hand, the user might want to install Fittino at a location different from the default. This can be achieved by setting `CMAKE_INSTALL_PREFIX` to the desired location, e.g. by typing

```
cmake -DCMAKE_INSTALL_PREFIX=/custom/installation/path ..
```

2.6 Troubleshooting

If the installation of Fittino fails, it is most likely due to a mandatory library which is not found by CMake. In this case, it might be imperative to manually modify the CMake search paths. The CMake search paths for external software can be influenced by setting the following dedicated CMake variables

- BOOST_ROOT
- ROOT_CONFIG_DIR
- GSL_CONFIG_DIR
- HIGGSBOUNDS_INSTALLATION_PATH
- HIGGSSIGNALS_INSTALLATION_PATH
- FEYNHIGGS_INSTALLATION_PATH
- SLHAEA_INSTALLATION_PATH
- LHAPDF_INSTALLATION_PATH

and/or via the usual general CMake and environmental variables.

2.7 Outlook

Things

- Even more modular installation
- Fittino SuperBuild

CHAPTER

THREE

USER GUIDE

In the following, the usage of Fittino is explained. It is assumed at this point that Fittino is already successfully built and installed on your system. If this is not the case, you might have a look at the *Installation guide* first.

It is not assumed that you have added the directory containing the built executable to your system search `PATH` (if you want to do this and do not know how, please check the documentation of your operating system). So, in order to use Fittino, simply change to the directory containing the built executable (e.g. `bin/`).

In the first section of this document the invocation of Fittino from the command line and the available command line options are explained. The majority of configuration options, however, is specified in so-called Fittino input files, which are covered thoroughly in the second section. At last, a final section walks through a simple working example, explaining all the details of its configuration and encourages the reader to experiment and manipulate some of the configuration options on his or her own.

Note: The main purpose of Fittino is to provide tools to analyze highly complex and specialized physics models. If you are about to fully exploit the capabilities of Fittino, you probably want to write your own extensions to the program in addition to what is described here. If this is the case, you also might have a look at the developer-guide.

3.1 Basic usage

The basic command structure of Fittino is

```
$ ./fittino [OPTION(S)]
```

If Fittino is run without options, a help text providing short information about all available options is shown.

The individual options are

`-h, --help`
A help text providing short information about all available options is shown

`-i` `FILE`, **`--input-file`**`=FILE`
Fittino uses the input file `FILE`

This is probably the most important option, since without specifying an input file, Fittino does not do anything useful. A Fittino input file is an XML file, which allows the detailed configuration of the program's behavior. For further information on input files, have a look at *Input files*. To help you getting started, a number of annotated example input files can be found in the `input/` directory

`-l` `FILE`, **`--lock-file`**`=FILE`
Fittino uses the file `FILE` for inter process locking

`-v, --validate`
Fittino validates the input file using the libxml2 library

This option is supposed to help you identifying possible issues with your custom input file. A warning or even an error is produced if you have specified an unsupported configuration item or if a required option is missing or incorrectly set

Example:

The command

```
$ ./fittino -v -i ../input/Example.Rosenbrock.Simple.in.xml
```

runs Fittino using the input file `Example.Rosenbrock.Simple.in.xml`. In addition the input file is validated.

3.2 Input files

The functionality of Fittino is controlled using an XML input file whose path is passed to the program as a parameter. In the input file the user can specify the model to be analyzed and the tool the model should be analyzed with. An example for a complete input file can be found in the *Getting started* section. In addition, a number of annotated example input files can be found in the `input/` directory.

3.2.1 Why XML?

The XML format has been chosen because it is widely used in software configuration and many reliable parsers are readily available. In XML, configuration items can be nested infinitely, so that options can be easily extended. In addition, it is possible to exactly define the structure of an XML document for a specific application, which means the content can be validated. In this way, the validation provides invaluable help, especially for the less experienced user, to prevent configuration errors at the earliest possible stage. The corresponding XML Schema Definition files (XSD) can be found in the `input/definitions/` directory. For simplicity, in Fittino input files only sub-nodes (and no attributes) are allowed to specify configuration options.

Complex XML files, on the other hand, are considered to be tedious to read and write. As a countermeasure, Fittino comes with a set of dedicated style definitions, which allow the user to look at well-arranged configuration options with a web browser. The corresponding XML Stylesheet Language files (XSL) can be found in the `input/style/` directory. For the future, however, it might be worthwhile to also consider alternative input file formats for Fittino, e.g. the rather recent JavaScript Object Notation (JSON), which, despite being less powerful than XML, provides all the aforementioned features and is well readable and writable at the same time.

3.2.2 Input file structure

The basic structure of a Fittino input file looks like this

```
<?xml version="1.0" encoding='UTF-8'?>
<?xml-stylesheet type="text/xsl" href="style/InputFile.xsl"?>

<InputFile>
  <GlobalOption>Value</GlobalOption>
  ...
  <Model>
    ...
  </Model>
  <Tool>
    ...
  </Tool>
</InputFile>
```

The root node (or root element) is `<InputFile>`. The model of interest, the analysis tool and several global configuration options, which affect the general behavior of Fittino, are then specified as first-level sub-nodes. How this is done is explained in detail in the following sections.

3.2.3 Global configuration options

Find below a list of all input file configuration options which affect the global behavior of Fittino.

`<VerbosityLevel>`

> The verbosity level configuration option specifies the amount of output that is produced in the course of the execution of Fittino. Three levels are available, `ALWAYS`, `INFO` and `DEBUG`. In this manner, `ALWAYS` produces only minimal output while `DEBUG` provides the most detailed information about the current status of the active tool and the model being analyzed. The default verbosity level is `ALWAYS`.

Example:

```
<VerbosityLevel>INFO</VerbosityLevel>
```

sets the verbosity level to `INFO`, which produces a bit more output than the default `ALWAYS`.

`<RandomSeed>`

> The random seed configuration option specifies the seed for the random number generators used within Fittino. The choice of a fixed random seed ensures that the result of consecutive Fittino runs is reproduced. If, however, statistically independent results are intended, the random seed has to be changed for each individual run. The random seed configuration option takes an integer number as an argument and defaults to 0.

Example:

```
<RandomSeed>3141</RandomSeed>
```

sets the random seed to 3141.

In the next section the configuration of *Models* for Fittino is explained in full detail.

3.2.4 Models

The `<Model>` element specifies the (physical) model being analyzed by Fittino. In physics, a model is understood as a collection of mathematical prescriptions ("formulas") which map a set of parameters onto a set of predictions about the outcome of a set of experimental measurements. By finding models where the adaption of a few parameters lead to accurate predictions about a wealth of experimental measurements one hopes to gain deeper insight into the underlying principles of Nature. Often, however, one is faced with the situation that the model parameters themselves are not directly accessible by experiments. Fittino has been developed to obtain information about the model parameters in this case, nevertheless.

The level of accuracy between a theoretical prediction and a measurement is quantified as the so-called χ^2 value, which is the error-weighted squared difference between prediction and measurement. A number of tools in Fittino try to minimize the χ^2 value which is equivalent to finding the set of parameters that provides the best description of Nature (within the limits of the model). The determination of the χ^2 value from a set of model parameters is accomplished by a collection of dedicated so-called `Calculators`. With other words, in order to configure a model, the user mainly has to specify the list of model parameters and choose a collection of `Calculators`. In the following, the list of all model configuration options is presented.

`<Name>`

> The name of the model. This name appears at various locations in the output of the program and hereby helps with the organization of output files.

Example:

```
<Name>CMSSM</Name>
```

sets the model name to `CMSSM`.

`<Tag>`

The tag of the model.

Example:

```
<Tag>CMSSM</Tag>
```

sets the model tag to `CMSSM`.

`<ModelParameter>`

Specifies a new model parameter. A model parameter can further be configured with the following options:

`<Name>`

The name of the parameter.

`<PlotName>`

The ROOT markup string for the name of the parameter. Since Fittino uses ROOT to make plots, the ROOT markup conventions for formatted labels are simply adapted. For more information on how formatted labels are marked in ROOT, please have a look at the documentation at the ROOT homepage.

`<Value>`

The initial value of the parameter. This option is ignored if an analysis tool calculates the starting value of the parameter on its own.

`<Error>`

This value serves as a measure for the scale on which the parameter is allowed to vary. Depending on the used analysis tool this value is interpreted differently. Possible interpretations are e.g. a step width or a standard deviation in a Gaussian smearing.

`<Unit>`

The physical unit of the parameter.

`<PlotUnit>`

The ROOT markup string for the physical unit of the parameter.

`<LowerBound>`

A parameter cannot assume a value below this limit.

`<UpperBound>`

A parameter cannot assume a value above this limit.

`<Fixed>`

If set to `true` the parameter is kept fixed on its starting value specified by the `<Value>` element. The default value for this option is `false`.

Example:

```
<ModelParameter>
  <Name>Mass_h</Name>
  <PlotName>M_{h}</PlotName>
  <Value>125.5</Value>
  <Error>0.3</Error>
  <Unit>GeV</Unit>
  <PlotUnit>GeV</PlotUnit>
  <LowerBound>120</LowerBound>
  <UpperBound>130</UpperBound>
  <Fixed>false</Fixed>
</ModelParameter>
```

specifies a new model parameter named `Mass_h`. The parameter can be varied between 120 and 130 GeV, with step width 0.3 GeV starting at 125.5 GeV.

`<Calculator>`

One or more calculators can be specified with this element. Detailed information about the available calculators and their individual configuration options can be found in the *Calculators* section.

Note: The calculators are executed in the same order as they are specified, which means that **order matters**!

`<Chi2Contribution>`

The `<Chi2Contribution>` takes the name of a quantity as an argument. The value of this quantity is added to the global χ^2 value.

Example:

```
<Chi2Contribution>Penalty</Chi2Contribution>
```

adds the value of `Penalty` to the global χ^2 value. `Penalty` needs to be calculated by a dedicated calculator.

In conclusion, the overall structure of a model looks like this

```
<Model>
  <Name>ModelName</Name>
  <Tag>ModelTag</Tag>
  <ModelParameter>
    ...
  </ModelParameter>
  ...
  <Calculator>
    ...
  </Calculator>
  ...
  <Chi2Contribution>
    ...
  </Chi2Contribution>
  ...
</Model>
```

3.2.5 Calculators

`Calculators` play a central role within the Fittino framework. They can be thought of as computing units which transform a set of input variables into a set of output variables. As such, the `Calculators` are ultimately responsible to calculate the model predictions from a given set of input parameters.

There exist already a number of theory programs, calculating predictions for physics experiments, which are developed and maintained by experts of their respective fields. These programs, however, are usually highly specialized and concentrate solely on the calculation of a single aspect of more comprehensive models. In High Energy Physics, for example, separate programs for the computation of the masses of Supersymmetric Particles, the Dark Matter Relic Density and the outcome of Flavor Physics experiments are available. The reason for this separation is that the relationships between the different kinds of phenomena are unclear. To shed light on the possible connections within a more comprehensive model, however, is exactly among the tasks of programs such as Fittino, which provide common interfaces in order to combine the output of codes from very different sources. The `Calculators` can now be understood as wrapper classes around third party software, implementing the interfaces. Furthermore, Fittio comes with a number of built-in `Calculators` serving various purposes. Apart from calculating additional physical model predictions, these `Calculators` may be applied to very general (intermediate) processing tasks, such as the calculation of a formula or the interpolation of a parameter grid.

To conclude, instead of defining a single `Calculator`, a physics model usually constitutes a chain of `Calculators`, meaning a set of `Calculators` which are executed successively. This approach allows for a flexible combination of `Calculators` and enables more sophisticated use cases such as the parallel execution of the same `Calculator` with different configurations.

The various available `Calculators` and their configuration options are described in the following. One configuration option is common to all calculators:

`Tag`

> The tag of the calculator. If specified, the tag is appended to all quantities that are computed by the `Calculator`. This is especially useful if the same quantity should be computed multiple times by different `Calculators` or by several instances of the same `Calculator` with different configuration options.

Example

By specifying

```
<Tag>MyCalculatorTag<Tag>
```

any `Quantity` is renamed to `MyCalculatorTag_Quantity`.

Chi2Calculator

`<Observable>`

> `<Name>`
>
> > The name of the observable.
>
> `<Measurement>`
>
> > The measured value of the observable.
>
> `<Prediction>`
>
> `<Uncertainty>`
>
> > `<Name>`
> >
> > `<Value>`
>
> `<IsLowerLimit>`
>
> `<IsUpperLimit>`
>
> `<LowerBound>`
>
> `<UpperBound>`

`<Correlation>`

`<Name>`

`<Value>`

`<Observable>`

FormulaCalculator

The `FormulaCalculator` can be used to calculate a novel quantity from existing quantities using a mathematical formula. Since Fittino uses the ROOT class `TFormula` for formulas, the ROOT conventions for formula strings are simply adapted. Within the formulas, existing quantities are referenced by name. For more information on how formula strings are written in ROOT and which mathematical expressions are available, please have a look at the documentation at the ROOT homepage.

The `FormulaCalculator` can be configured using the following options:

`Name`

The name of the calculated quantity.

`Formula`

The ROOT markup string of the formula.

Example:

```
<ModelParameter>
  <Name>X1</Name>
  <Error>0.01</Error>
  <LowerBound>-2</LowerBound>
  <UpperBound>2</UpperBound>
</ModelParameter>

<ModelParameter>
  <Name>X2</Name>
  <Error>0.01</Error>
  <LowerBound>-2</LowerBound>
  <UpperBound>2</UpperBound>
</ModelParameter>

<FormulaCalculator>
  <Name>Parabola</Name>
  <Formula>[X1]^2+[X2]^2</Formula>
</FormulaCalculator>
```

calculates a two-dimensional parabola from the model parameters `X1` and `X2`.

LinearInterpolationCalculator

`File`

`Histogram`

`Variable`

RegressionCalculator

`Variable`

  `Name`

  `NameInWeightFiles`

`MVA`

  `WeightFile`

  `Target`

RosenbrockCalculator

The `RosenbrockCalculator` evaluates the Rosenbrock function

$$f(\mathbf{x}) = \sum_{i=1}^{N} (1 - x_{i-1})^2 + 100 \cdot \left(x_i - x_{i-1}^2\right)^2$$

for an arbitrary number of arguments N. The Rosenbrock function is a simple but not trivial optimization problem which can be used as a test model for the validation of Fittino's functionality, especially for the optimization and sampling algorithms. For low dimensionality, the Rosenbrock function can also be computed by the `FormulaCalculator` but at some point the explicit notation becomes tedious. In addition, the `RosenbrockCalculator` might be a helpful simplistic example for developers who want to implement their own custom `Calculator` class.

TreeCalculator

The `TreeCalculator` comes into play when Fittino data should be reprocessed. For that, the `TreeCalculator` simply loops over the entries of the ROOT tree stored in the specified file.

The options of the `TreeCalculator` are:

`InputFileName`

  The name of the input file.

`InputTreeName`

  The name of the input tree.

3.2.6 Tools

Now that Fittino models are established and the desired predictions can be calculated from a set of model parameters it is time to ask how to analyze these. Tools are algorithms. Analysis such as optimization and sampling algorithms. There are also plotting devices. As usual, the descripition of the available tools starts with an overview of the configuration options common to all tools:

`Name`

  The tool's name.

`OutputFile`

  The name of the output file. All data produced by the tool is stored in this file. The default output file name is `Fittino.out.root`.

`WriteAllModelQuantities`

> If set to `true` all quantities are written to the output file.

`Chi2Name`

> The name of the total χ^2 value.

`InitialChi2Value`

> The total χ^2 is initialized with this value.

`IterationCounterName`

> The name of the iteration counter.

`InitialIterationCounterValue`

> The iteration counter is initialized with this value.

`OutputTreeName`

> The name of the output data tree.

`MetaDataTreeName`

> The name of the meta data tree. Meta data such as error codes.

In the following sections the available (classes of) tools are described in full detail.

HistogramMakers

`HistogramMakers` are supposed to help the user to convert the data produced by Fittino into histograms. Among other things, histogrammed data has the advantage of requiring fewer storage space and being easier to process. This is the reason why many `Plotters` operate only on histogrammed data.

The configuration options common to all `HistogramMakers` are:

`AxisMaxDigits`

> Specifies the nuber of digits of the numeric tic labels.

`Histogram`

> `Name`
>
> > The name of the histogram.
>
> `PlotName`
>
> > The name of the plot.
>
> `LogScale`
>
> > Calculates and fills logarithmic bins.
>
> `NumberOfBins`
>
> > The number of histogram bins.
>
> `LowerBound`
>
> > The lower bound of the histogram.
>
> `UpperBound`
>
> > The upper bound of the histogram.

`Plotter`

The `Plotter` used to render a graphic representation of the histogram. The available plotters and their configuration options are described in more detail in the section.

They have a separate section because they also can be used standalone.

ContourHistogramMaker

Prepares the histograms for the contour plots. It should be used together with `ContourPlotter`. The execution of `ContourHistogramMaker` may take some time, depending on the number of data points that have been sampled. It is therefore recommended to separate the histogram filling from the actual plotting, in order not to have to re-run `ContourHistogramMaker` if only changes to the graphical representation need to be applied.

ProfileHistogramMaker

Prepares the histograms for the χ^2 profile plots. It should be used together with `ProfilePlotter`.

Simple1DHistogramMaker

Fills a one-dimensional histogram for each specified quantity.

Simple2DHistogramMaker

Fills a two-dimensional histogram for each combination of two specified quantities.

Simple3DHistogramMaker

Fills a three-dimensional histogram for each combination of three specified quantities.

SummaryHistogramMaker

The `SummaryHistogramMaker` fills histograms needed for a plot showing summarized results of the specified quanitites in one plot.

`LogScale`

Calculates and fills logarithmic bins.

`NumberOfBins`

The number of histogram bins.

`Spacing`

`Borders`

`AxisMaxDigits`

Specifies the nuber of digits of the numeric tic labels.

`LowerBound`

The lower bound of the histogram.

`UpperBound`

The upper bound of the histogram.

`Quantity`

`Name`

The name of the quantity.

`PlotName`

The name of the plot.

`Plotter`

The `Plotter` used to render a graphic representation of the histogram. The available plotters and their configuration options are described in more detail in the section.

They have a separate section because they also can be used standalone.

Optimizers

The Fittino `Optimizers` are algorithms that try to find the best matching set of model parameters based on a criterion that is usually a measure of agreement between the model predictions and the corresponding experimental measurements.

The following configuration options are common to all Fittino `Optimizers`:

`MaxNumberOfIterations`

The maximal number of iterations to be processed.

`AbortCriterion`

If the quantity that serves as a measure of agreement between the model predictions and the corresponding experimental measurements becomes smaller than that number, the algorithm aborts.

GeneticAlgorithmOptimizer

`MutationRate`

`SizeOfPopulation`

MinuitOptimizer

The `MinuitOpitmizer` is merely a wrapper around the `Minuit2` classes shipped with the ROOT data analysis framework. It is taken ‘as is’ and no further configuration options are supported by Fittino at the moment. For more information on `Minuit2`, please have a look at the corresponding documentation website.

ParticleSwarmOptimizer

`NumberOfParticles`

`C1`

`C2`

SimpleOptimizer

SimulatedAnnealingOptimizer

`InitialTemperature`

`TemperatureReductionFactor`

Plotters

Plotters can be used to produce graphic representations of the data created by Fittino. Hereby, the various plotters aim to faciliate daily work standard plots to quickly obtain nicely rendered plots without putting too much thougtht into it. A number of configruation options exist to accomodated the user's needs as as possible.

Configuration options common to all plotters:

`LogScaleX`

> Sets the labels of the x-axis to logscale. To work properly with histograms, make sure that logarithmic bins have been filled.

`LogScaleY`

> Sets the labels of the y-axis to logscale. To work properly with histograms, make sure that logarithmic bins have been filled.

`LogScaleZ`

> Sets the labels of the z-axis to logscale. To work properly with histograms, make sure that logarithmic bins have been filled.

`FileFormat`

> Specifies the output file format. All output file formats that are supported by ROOT are likewise valid in Fittino.

`PageFormat`

> Currently there are two possible options: `Landscape` and `Square`. `Landscape` is a rectangular 8:6 format, which is usually best suited if an even number of plots should be displayed in one row next to each other. `Square` is a quadratic format and should be used for an odd number of plots, respectively.

`Version`

> Specifies the release version of Fittino.

`LogoPath`

> An image whose location is specified by this option is placed at the top right corner of the plot.

The following plotters are available:

ContourPlotter

The contour plotter is used to display the best fit point togehter with the 2-dimensional confidence regions of two parameters.

`LegendFrame`

> This option specifies whether or not the legend is drawn with a black outline.

`LegendPosition`

If the default `BottomRight` location of the legend threatens to hide displayed information, this option can be used to move the legend to a different corner of the plot.

`Style`

Three distinct styles, `Plain`, `Classic` and `Peach`, are available, ranging from very simplistic to more colorful designs.

ProfilePlotter

The `ProfilePlotter` is intended to produce nice displays of the χ^2 value in dependence of a single quantity, e.g. a model parameter. These kind of plots are very important and often looked upon, because they provide valuable information about the location and the width of the minima in the χ^2 space. There are currently no additional configuration options needed for `ProfilePlotter`.

SimplePlotter

The if

`FillColor`

The color code. The default fill color is black.

`FillStyle`

The default fill style is a plain hatched pattern.

`LineColor`

The default line color is black.

`Option`

Option string

SummaryPlotter

`LabelsLeft`

`NDivisions`

`Title`

`LowerBound`

`UpperBound`

ToyFitPlotter

This tool has to be implemented yet.

Samplers

As the name suggests, `Samplers` are supposed to sample, that is to write out, points of the parameter space in a distinguished manner. The main difficulty each efficient sampling algorithm is faced with is to write out relatively more points in the interesting region while covering the whole parameter space at the same time.

HigherOrderMarkovChainSampler

`ScalingFactor`

`MinimalMemorySize`

`MaximalMemorySize`

MarkovChainSampler

`FirstPointScaleFactor`

`StrictBounds`

`NumberOfIterations`

The maximal number of iterations.

`Chi2`

`IterationCounter`

The name of the iteration counter

`Parameter`

`Name`

The name of the parameter.

`Value`

The value of the parameter.

SimpleSampler

The `SimpleSampler` simply samples every point (incremented by the parameter error) within the specified parameter bounds. For that, it loops recursively over the dimensions of the parameter space. Understandably, this simple, brute-force approach is suited only for parameter spaces with low dimensionality, since the computational costs raise with the power of the number of parameters. Nevertheless, the `SimpleSampler` has its applications, e.g. if one wants to do a quick scan of a sub-space (while keeping the other parameters fixed). More importantly, the `SimpleSampler` can be used to produce a (coarse) grid of a parameter (sub-) space, where the computation of individual points takes too much time. The points of the interpolated grid can then be sampled easily, which is in some situations the only way to extract information about the parameter space of interest.

TreeSampler

`NumberOfIterations`

`InputFileName`

The name of the input file.

`InputTreeName`

The name of the input tree.

3.3 Getting started

In this section, the functionality of Fittino is discussed in detail with the help of a simple but not trivial example optimization problem. It is assumed at this point, that Fittino is successfully built and installed on the reader's system and that he or she is equipped with a working knowledge of the basic usage of the software framework. If this is not the case it is recommmended to consult the Installation guide and/or the previous sections of this guide beforehand.

The optimization problem is the 2-dimensional Rosenbrock function.

At first, let's have a look at input file `Example.Rosenbrock.Simple.in.root` which can be found in the `input/` subdirectory

```
<?xml version="1.0" encoding='UTF-8'?>
<?xml-stylesheet type="text/xsl" href="style/InputFile.xsl"?>

<InputFile>

  <VerbosityLevel>ALWAYS</VerbosityLevel>

  <Model>
    <Name>2D Parabola</Name>

    <ModelParameter>
      <Name>X1</Name>
      <Error>0.01</Error>
      <LowerBound>-2</LowerBound>
      <UpperBound>2</UpperBound>
    </ModelParameter>

    <ModelParameter>
      <Name>X2</Name>
      <Error>0.01</Error>
      <LowerBound>-2</LowerBound>
      <UpperBound>2</UpperBound>
    </ModelParameter>

    <FormulaCalculator>
      <Name>Parabola</Name>
      <Formula>[X1]**2+[X2]**2</Formula>
    </FormulaCalculator>
  </Model>

  <Tool>
    <SimpleSampler></SimpleSampler>
  </Tool>

</InputFile>
```

The first line indicates XML file. Second line is a reference to the style sheet files. Input file can be opend with browser the properties are into tables better for overview of complicated files.

The verbosity level is set to always which means that only the most important information is printed out. Then the model is defined. The model consists of two parameters x_1 and x_2.

Output

```
--------------------------------------------------------------------------------------


  Welcome to Fittino
```

```
--------------------------------------------------------------------------------------

  Initializing the Rosenbrock model

  Initializing the list of model parameters

--------------------------------------------------------------------------------------

  Initializing the Simple parameter sampler

   Configuration

    Name                                                Simple parameter sampler
    OutputFileName                                      Fittino.out.root
    WriteAllModelQuantities                             1
    Chi2Name                                            Chi2
    InitialChi2Value                                    inf
    IterationCounterName                                IterationCounter
    InitialIterationCounterValue                        0
    OutputTreeName                                      Tree
    MetaDataTreeName                                    MetaDataTree

--------------------------------------------------------------------------------------

  Running the Simple parameter sampler

--------------------------------------------------------------------------------------

  Terminating the Simple parameter sampler

--------------------------------------------------------------------------------------
```

After displaying a short welcome logo, Fittino initializes the model and its individual components. Then, the analysis tool is initialized which is the `SimpleSampler`. One can see that all data is written out to a ROOT file called `Fittino.out.root`. The data tree is called `Tree` and the meta data tree is called `MetaDataTree`.

If the verbosity level is changed Detailed Output

```
--------------------------------------------------------------------------------------

  Welcome to Fittino

--------------------------------------------------------------------------------------

  Initializing the Rosenbrock model

   Initializing the list of model parameters

    X1                                                  0.00e+00
    X2                                                  0.00e+00

--------------------------------------------------------------------------------------

  Initializing the Simple parameter sampler

   Configuration

    Name                                                Simple parameter sampler
    OutputFileName                                      Fittino.out.root
    WriteAllModelQuantities                                  1
    Chi2Name                                                 Chi2
    InitialChi2Value                                         inf
    IterationCounterName                                     IterationCounter
    InitialIterationCounterValue                             0
    OutputTreeName                                           Tree
    MetaDataTreeName                                         MetaDataTree

--------------------------------------------------------------------------------------

  Running the Simple parameter sampler

--------------------------------------------------------------------------------------

  Iteration step 1

  Set of Rosenbrock model parameters:

    X1                                                      -2.00e+00
    X2                                                      -2.00e+00

    -----------------------------------------------------------

    Total chi2                                               3.61e+03

--------------------------------------------------------------------------------------

...

--------------------------------------------------------------------------------------

  Terminating the Simple parameter sampler

--------------------------------------------------------------------------------------
```

INDEX