*my*Fitter Manual

for version 0.2, 28 June 2013

Martin Wiebusch (`martin.wiebusch@kit.edu`)

This manual is for *my*Fitter (version 0.2, 28 June 2013), a C++ class library for maximum likelihood fits and numerical computation of p-values.

Table of Contents

1 Getting Started

The concepts and algorithms implemented in *my*Fitter are described in

> M. Wiebusch, "Numerical Computation of p-values with *my*Fitter," Comput. Phys. Commun. (2013), DOI: 10.1016/j.cpc.2013.06.008, [arXiv:1207.1446v2]

If you use the *my*Fitter package for a scientific publication, please cite this paper. In addition, please follow the citation guidelines of the Dvegas package, which *my*Fitter links to.

1.1 Requirements

The *my*Fitter source code can be obtained from `http://myfitter.hepforge.org` (but since you're reading this, you probably got there already). To compile it, you need

- a standard-compliant C++ compiler like the GNU compiler,
- the GNU Scientific Library version 1.13 or later,
- the Boost C++ Libraries version 1.40 or later and
- the Dvegas library version 2.0.2 or later.

1.2 Installation

The *my*Fitter package uses the standard GNU build system and `pkg-config` to keep track of the various flags you need to link your own programs to *my*Fitter. To install the package, unpack the tarball with

```
tar -zxvf myfitter-x.y.tar.gz
```

(*x.y* being the version of the package). If you have root access on your system you can simply install the package with

```
cd myfitter-x.y
./configure
make
make install
```

This will put all the headers, library, documentation and '`pkg-config`' files in standard directories (usually '`/usr/local/include/myfitter`', '`/usr/local/lib`', '`/usr/local/share/info`' and '`/usr/local/lib/pkgconfig`') where your compiler and other tools will automatically find them. You only need root access for the last command.

If you don't have the root password or don't want to install the package system-wide, you can run

```
cd myfitter-x.y
./configure --prefix=myprefix
make
make install
```

where *myprefix* is some directory you have write access to. This will put the headers in '*`myprefix`*`/include/myfitter`', the library in '*`myprefix`*`/lib`', the documentation in '*`myprefix`*`/share/info`' and the `pkg-config` file in '*`myprefix`*`/lib/pkgconfig`'. If you use `pkg-config` when compiling your own programs, you only have to make sure that `pkg-config` finds the file '`myFitter.pc`' in '*`myprefix`*`/lib/pkgconfig`'. You can do this by setting the environment variable `PKG_CONFIG_PATH`. In `bash`, simply call

```
     export PKG_CONFIG_PATH=$PKG_CONFIG_PATH:myprefix/lib/pkgconfig
```

You may want to add this line to your '`~/.bashrc`'.

If you want to be thorough, you can also run the unit tests for the package. To do this, simply call `make check` in the *my*Fitter source directory. These tests involve a few numerical integrations and may take a couple of minutes, so feel free to grab a cup of coffee while they run. If anything goes wrong with the tests please let the authors know about it.

You can also generate the documentation in some formats other than '`.info`' files. Try running `make html` or `make pdf` in the *my*Fitter source directory and then look at '`doc/myFitter.html`' or '`doc/myFitter.pdf`'.

1.3 Compiling the Example

When everything is installed properly, copy the contents of '`doc/examples`' to some other directory and run

```
     g++ -o linear_example linear_example.cpp $(pkg-config --cflags --libs myFitter)
```

If you don't use the GNU compiler (but why wouldn't you?) replace `g++` by whatever your compiler is called. The command `pkg-config --cflags --libs myFitter` prints out the flags needed to link your program to *my*Fitter, and the `$()` construct substitutes the output on the command line. When you write bigger programs you'll want to do compiling and linking in two steps. Then use `pkg-config --cflags myFitter` for compilation and `pkg-config --libs myFitter` for linking.

If everything worked, you now have an executable called '`linear_example`' in your working directory. Try to run it. It fits a simple model to some data and computes the p-value in two different ways. It takes a couple of minutes to run. To get a first idea of how to implement your own models and do your own fits, have a look at the source files '`linear_example.cpp`' and '`linearmodel.hpp`'. They have lots of comments that explain what's going on.

2 Implementing Models

All models in *my*Fitter are represented by classes derived from the base class `myfitter::Model` defined in '`myfitter/model.hpp`'. Note that all classes of the *my*Fitter library live in the namespace `myfitter`. To implement your own model, you have to write your own derived class. An example can be found in '`doc/examples/linearmodel.hpp`' in the source distribution.

2.1 The `Model` Base Class

The `Model` class provides the following functionality:

Constructors

The `Model` provides a copy constructor and an assignment operator, but no default constructor. The default constructors of derived classes should call the constructor

```
Model(int npar, int nobs)
```

which initialises a model with *npar* parameters and *nobs* observables. The values of the parameters and observables as well as their scales and upper and lower limits (see below) are initialised to NaN. This way, if you forget to initialise a parameter correctly, you will probably notice it.

Parameters

An object of type `Model` stores the "current" values of the parameters of the model. The number of parameters is fixed in the constructor and returned by the function `nparameters()`. Parameter values can be read out and set with the methods

```
double parameter(int ipar)
const ParameterVector& parameters()
void parameter(int ipar, double value)
```

where the parameter index *ipar* runs from zero to `nparameters()-1`. The type `ParameterVector` is a synonym for `boost::numeric::ublas::vector<double>`. You can find more information about uBLAS vectors in the documentation of the Boost uBLAS library.

Observables

A `Model` object stores the "current" values of the observables. The number of observables is fixed in the constructor and returned by the function `nobservables()`. The values of the observables can be accessed with the methods

```
double observable(int iobs)
const ObservableVector& observables()
```

where the index *iobs* runs from zero to `nobservables()-1`. The type `ObservableVector` is another synonym for `boost::numeric::ublas::vector<double>`.

Scales of Parameters

You can read and set the *scale* of each parameter with

```
double scale(int ipar)
void scale(int ipar, double value)
```

For all internal computations, parameters are normalised to their scale. In particular, the chi-square function is not minimized with respect to the parameters p_i set by the user via `Model::parameter`, but with respect to p_i/s_i, where s_i are the scales of the parameters. Ideally, the scales of the parameters should be chosen so that the second derivative of the chi-square function with respect to p_i at the minimum is of the same order as s_i. In other words, they should be your best guess for the errors of the parameters when fitting the model to data. You can set the value and scale of a parameter in one go with

```
void parameter(int ipar, double value, double scale)
```

Bounded Parameters

You can set the ranges in which parameters are allowed to float with

```
void upper_limit(int ipar, double value)
void lower_limit(int ipar, double value)
void set_range(int ipar, double upper_lim, double lower_lim)
```

and obtain the current upper and lower limit of a parameter with

```
double upper_limit(int ipar)
double lower_limit(int ipar)
```

Initially, all parameters are unbounded. In this case, the two functions above return NaN. Conversely, you can remove an upper or lower limit on a parameter by setting the limit to NaN. To do this, you should use the static method

```
static double Model::nan()
```

which just returns NaN. To check if a certain parameter currently has an upper or lower limit you can use the methods

```
bool has_upper_limit(int ipar)
bool has_lower_limit(int ipar)
```

You can set the value, scale, lower and upper limit of a parameter in one go with

```
void parameter(int ipar, double value, double scale,
               double lower_limit, double lower_limit)
```

Fixing Parameters

You can fix a parameter to its current value or release it with the methods

```
void fix(int ipar)
void release(int ipar)
```

A fixed parameter does not float in a fit. To check if a parameter is currently fixed, use the method

```
bool fixed(int ipar)
```

Calculating Derivatives

The derivatives at the current point in parameter space can be calculated with

```
virtual int calc_deriv()
```

which returns zero if the calculation was successful and a non-zero value otherwise. The derivative matrix can be accessed with the method

```
const Matrix& derivatives()
```

The type `Matrix` is a synonym for `boost::numeric::ublas::matrix<double>`. You can find more information about uBLAS matrices in the documentation of the Boost uBLAS library. To access the elements of a `Matrix` object, just call the object with two integer arguments. For the matrix returned by `derivatives()`, the first index is a parameter index and the second an observable index.

The derivatives are calculated numerically by varying the parameters by small amounts proportional to their scale (as returned by `scale(`*`ipar`*`)`). The proportionality factor can be read and modified with the methods

```
double derivative_epsilon()
void derivative_epsilon(double value)
```

Derived classes may overload the `calc_deriv()` method, for example to implement analytical formulae for the derivatives with respect to some parameters. Your own implementation should assign the values of the derivatives to the protected member `derivatives_`, which is of type `Matrix`. If you do not want to implement all derivatives yourself the derivatives with respect to a certain parameter *ipar* can be be calculated numerically with the protected method

```
int numerical_derivative_(int ipar)
```

This method fills the corresponding row of `derivatives_` and returns zero on success and a non-zero value on failure. Finally, you can check your own implementation of derivatives with the method

```
bool check_derivatives(double rel_prec, double abs_prec)
```

This method checks if your results for the derivatives agree with the numerical derivatives with a relative precision *rel_prec*. Any derivatives which are smaller than *abs_prec* in magnitude are regarded as exactly zero.

The `smallrange` Flags

You can read and set the *smallrange* flag for each parameter with the methods

```
bool smallrange(int ipar)
void smallrange(int ipar, bool value)
```

When the smallrange flag is set, the parameter is considered fixed for the purpose of determining the model's hyperplane before a p-value integration (see [arXiv:1207.1446v2] for details), but floats in any fits performed during the p-value integration. The right combination of smallrange flags can significantly increase the efficiency of p-value integrations. The flag should be set if a parameter is only allowed to vary in a small range or if the dependence of all observables on that parameter is very weak.

Sampling the Parameter Space

You can randomly sample the parameter space and build up a dictionary of parameter values and the corresponding observable values. This dictionary can then be used by the fit functions to find good starting points for minimising the chi-square. The ranges in which the parameters are scanned can be read with the methods

```
double scan_min(int ipar)
double scan_max(int ipar)
```

and set with

```
void scan_min(int ipar, double value)
void scan_max(int ipar, double value)
void scan_range(int ipar, double min_val, double max_val)
```

If you are lazy you can also set the scan ranges of all parameters at once with

```
void set_scan_ranges(double factor)
```

which sets the scan range of each parameter *i* to the interval from `parameter(i)-factor*scale(i)` to `parameter(i)+factor*scale(i)`. To sample the parameter space with *n* points, call

```
void scan(int n)
```

If you want more sample points in a specific part of the parameter space you can change the scan ranges and call `scan()` again. The old data will be kept. To clear the dictionary, call

```
void clear()
```

Chi-square values and constraint penalty

Each `Model` object stores a chi-square value which can be accessed with the methods

```
double chisquare()
void chisquare(double chisq)
```

When you fit your model to some data using the `Fitter::global_fit` or `Fitter::local_fit` methods (see Section 3.3 [Minimising the chi-square Function], page 13) they will set the `chisquare` member of your `Model` object to the minimum chi-square value found in the fit. Usually you should not have to modify the `chisquare` member yourself.

Non-linear constraints on your model's parameter space are handled in *my*Fitter by adding penalty terms to the chi-square function which become large when the constraints are violated (see Section 3.2 [Non-linear Constraints], page 12). After calling `Fitter::global_fit` or `Fitter::local_fit` the contribution of these penalty terms is stored in the `constraint_penalty` member of the `Model` class:

```
double constraint_penalty()
void constraint_penalty(double p)
```

As with the `chisquare` member, you will usually not have to modify the `constraint_penalty` member yourself. The value stored in the `chisquare` member *never* includes the contribution from the penalty terms.

2.2 Writing your own Model Class

The only thing the `Model` class does not do for you is calculating the observables for given values of the parameters. Derived classes must implement this by overloading the virtual method

```
virtual int calc()
```

This method should use the current values of the parameters, as returned by `Model::parameter(int i)`, compute the observables and assign them to the elements of the protected member

```
ObservableVector observables_
```

The type `ObservableVector` is a synonym for `boost::numeric::ublas::vector<double>` and documented in the Boost uBLAS library. For most purposes, you only need to know that the elements of `observables_` can be accessed like those of a normal C array or `std::vector<double>`, i.e. with `observables_[i]`. The `calc()` method should return zero when the computation was successful and a non-zero value otherwise. To avoid computing the observables repeatedly with the same parameter values, the `Model` class has a protected boolean member

```
bool needs_update_
```

which is set to `true` whenever a parameter value is changed. You can check the value of `needs_update_` at the start of your `calc()` implementation and you *should* set it to `false` after a successful computation of the observables. The default implementation of `Model::calc()` only sets `needs_update_` to `false` and returns zero.

When you write your own `Model` classes, you should observe few guidelines which will make your life easier in the long run. A typical declaration of a model class will look like this:

```
#include <myfitter/model.hpp>
using namespace myfitter;

class MyModel : public Model {
private:
    ...

protected:
    MyModel(int npar, int nobs) : Model(npar, nobs) { ... }
    ...

public:
    // parameters
    enum {P_FIRSTPAR, P_SECONDPAR, ... , P_LASTPAR};
    static const int NPAR = P_LASTPAR+1;

    // observables
    enum {O_FIRSTOBS, O_SECONDOBS, ... , O_LASTOBS};
    static const int NOBS = P_LASTOBS+1;

    // constructors
    MyModel() : Model(NPAR, NOBS) { ... }
    MyModel(const MyModel& m) : Model(m) { ... }

    // assignment operator
```

```
        virtual const MyModel& operator=(const MyModel& m) {
            Model::operator=(m);
            ...
            return *this;
        }

        ...

        virtual int calc();
    };
```

First of all, note that parameters and observables are identified in *my*Fitter by integer numbers starting from zero. Since you probably don't want to remember the integer assignments for all parameters and observables in all your models, you should define `enum` types that give intuitive names to the integers associated with your parameters and observables. As the assignments are specific to each model class, you should make the `enum` types members of the corresponding model class. Using prefixes '`P_`' and '`O_`' for the names of parameters and observables, respectively, will avoid name clashes in cases where a parameter is at the same time an observable. In addition, each model class should define static integer constants `NPAR` and `NOBS` which keep information about the number of parameters and observables defined in that model.

It is also a good idea to define a default constructor, a copy constructor and a virtual assignment operator for your model. This will allow you to save and restore the state of a model object in a single line. Furthermore, you should re-define the constructor with the two integer arguments from the `Model` class as a protected constructor. This will allow you to write derived classes of `MyModel` with additional parameters or observables. This is how the declaration of a derived class could look like:

```
    class MyOtherModel : public MyModel {
    private:
        ...

    protected:
        MyOtherModel(int npar, int nobs) : MyModel(npar, nobs) { ... }
        ...

    public:
        // parameters
        enum {P_FIRSTNEWPAR=MyModel::NPAR, ... , P_LASTNEWPAR};
        static const int NPAR = P_LASTNEWPAR+1;

        // observables
        enum {O_FIRSTNEWOBS=MyModel::NOBS, ... , O_LASTNEWOBS};
        static const int NOBS = P_LASTNEWOBS+1;

        // constructors
        MyOtherModel() : MyModel(NPAR, NOBS) { ... }
        MyOtherModel(const MyOtherModel& m) : MyModel(m) { ... }
        // assignment operator
        virtual const MyOtherModel& operator=(const MyOtherModel& m) {
            MyModel::operator=(m);
            ...
            return *this;
        }

        ...

        virtual int calc() {
            int status;
            if((status = MyModel::calc())) return status;

            // calculate new observables
            ...
        }
    };
```

Note that the definitions for `NPAR` and `NOBS` in `MyOtherModel` shadow the ones in `MyModel`.

3 Fitting a Model

To fit a given model you first have to specify the experimental inputs for your fit and then determine the best-fit parameters by minimising the chi-square function. Both can be done with the `myfitter::Fitter` class, which is declared in '`myfitter/fitter.hpp`'.

3.1 Specifying Experimental Inputs

To fit a model to some data you first have to instantiate the `myfitter::Fitter` class, which is declared in '`myfitter/fitter.hpp`'. Each `Fitter` object contains an object of type `myfitter::InputFunction` (defined in '`myfitter/inputfunction.hpp`'), which can be accessed through the method

```
InputFunction& Fitter::input_function()
```

The experimental inputs are specified by adding objects of type `InputComponent` to the `input_function()` member of the `Fitter` class. This is done with the method

```
int InputFunction::add(const InputComponent&)
```

The return value does (currently) not serve any purpose. An object of type `InputComponent` represents a term in the input function D (see [arXiv:1207.1446v2] for details) and stores default values for the measured observables (x_0 in the notation of [arXiv:1207.1446v2]). The most common types of input components are defined in the header '`myfitter/inputcomponents.hpp`'. Here is an example:

```
#include <myfitter/fitter.hpp>
#include <myfitter/inputcomponents.hpp>
using namespace myfitter;

...

int main()
{
    MyModel mymodel;

    // initialise mymodel
    ...

    Fitter fitter(MyModel::NOBS);
    fitter.input_function().add(
        GaussianIC(MyModel::O_MYSECONDOBS, 3.0, 1.2, 0.4, 0.6));
    ...
```

The constructor of the `Fitter` class takes an integer value as argument, which specifies the number of observables. Only models which provide that exact number of observables (as returned by `Model::nobservables()`) can be fitted with the created `Fitter` object. The last line adds a Gaussian contribution with systematic errors to the input function. Specifically, it states that the observables associated with the index `MyModel::O_MYSECONDOBS` (see [`MyModel` example], page 7) has a measured value of 3.0 with a Gaussian statistical error of 1.2 and systematic errors of +0.4 and −0.6. In a scientific text, this might be written as 3.0 ± 1.2 (stat.) $^{+0.4}_{-0.6}$ (syst.). When you perform the actual fit you can still change the central values of the observables. The methods `Fitter::local_fit` and `Fitter::global_fit` (see Section 3.3 [Minimising the chi-square Function], page 13) have optional arguments which let you override the default values stored in the `input_function()` member. However, the (different types of) errors can only be specified by passing correctly initialised `InputComponent` objects to the `InputFunction::add` method.

The same observable may contribute to several terms in the input function. For instance, following the last line of the previous example, you may add a second gaussian input for `O_MYSECONDOBS`:

```
fitter.input_function().add(
    GaussianIC(MyModel::O_MYSECONDOBS, 10.0, 0.1, 0.02, 0.03));
```

The contributions of the two terms to the input function will simply be added. However, the `InputFunction` object allows only one central value for each observable. The line above will therefore replace the previous central value of 3.0 with 10.0, so that the gaussian contributions from *both* terms will be computed with a central value of 10.0. Generally, you should avoid adding multiple gaussian input components for the same observable to your input function. If you have several measurements for the same observable you should instead combine the central values and errors correctly and only add one input component for the combination. However, for non-gaussian inputs or non-linear constraints (see Section 3.2 [Non-linear Constraints], page 12) this combination is not necessarily possible and it is for this reason only that *my*Fitter allows multiple inputs for the same observable.

The header '`myfitter/inputcomponents.hpp`' provides the following subclasses of `InputComponent`:

`GaussianIC`

This represents a single observable with a Gaussian error and (possibly) systematic errors. The constructor

```
GaussianIC(int iobs, double value, double error,
           double syst_plus=0., double syst_minus=0.)
```

creates an input component for observable *iobs* with central value *value*, Gaussian error *error* and systematic errors *syst_plus* and *syst_minus*. The values of *error*, *syst_plus* and *syst_minus* should all be positive. If the last two arguments are omitted, the default value 0 is used.

`AsymmetricGaussianIC`

This represents a single observable with asymmetric Gaussian errors and (possibly) systematic errors. The constructor

```
AsymmetricGaussianIC(int iobs, double value,
                     double error_plus, double error_minus,
                     double syst_plus=0.,
                     double syst_minus=0.)
```

creates an input component for observable *iobs* with central value *value*, asymmetric Gaussian errors *error_plus* and *error_minus* and systematic errors *syst_plus* and *syst_minus*. The values of *error_plus*, *error_minus*, *syst_plus* and *syst_minus* should all be positive. If the last two arguments are omitted, the default value 0 is used.

`CorrelatedGaussianIC`

This represents the contribution from several observables with Gaussian errors and a non-diagonal correlation matrix. The class can not be fully initialised in the constructor. The constructor

```
CorrelatedGaussianIC(int n)
```

creates an "empty" input component for n observables. Note that n is the number of observables provided by the model (i.e. the same number that is passed to the `Fitter` constructor) and *not* the number of observables contributing to the input component. To initialise a `CorrelatedGaussianIC` object, you first have to specify the contributing observables and their central values and errors with the `add()` method and then set the elements of the correlation matrix with the `cor()` method. Here is an example:

```
CorrelatedGaussianIC ic(MyModel::NOBS);
ic.add(MyModel::O_MYFIRSTOBS, 2.3, 0.4);
ic.add(MyModel::O_MYSECONDOBS, 5.6, 0.7);
ic.cor(MyModel::O_MYFIRSTOBS, MyModel::O_MYSECONDOBS, 0.23);
```

This means that the two observables associated with the integers `MyModel::O_MYFIRSTOBS` and `MyModel::O_MYSECONDOBS` (see [`MyModel` example], page 7) have measured values of 2.3 and 5.6, standard deviations of 0.4 and 0.7 and a correlation of 0.23. Note that only off-diagonal elements of the correlation matrix in one triangle have to be set. To add this input component to the input function of a `Fitter` instance `fitter`, call

```
fitter.input_function().add(ic);
```

All the information of the `ic` object is copied by the `InputFunction::add()` method and the `lc` object is no longer needed afterwards.

3.2 Non-linear Constraints

In some cases you may need to minimize the chi-square function under constraints of the form $g_i(\xi) = c_i$ where ξ are the parameters of your model, i is an index, g_i are some real-valued functions and c_i constants. One possibility to implement such constraints is to add *penalty terms* to the chi-square function which become large for $g_i(\xi) \neq c_i$. Of course the chi-square function then looses its statistical interpretation, and one should remove the penalty terms from the chi-square value when computing p-values and confidence intervals.

*my*Fitter supports this approach to non-linear constraints by keeping the penalty terms in the chi-square separate from the ordinary terms. To implement non-linear constraints $g_i(\xi) = c_i$ you first have to implement the functions g_i as an observables in your `Model` class. Then you add inputs for these observables to the `input_function()` member of your `Fitter` object with the method

```
int InputFunction::add_constraint(const InputComponent&)
```

You can use any of the `InputComponent` classes discussed above. Usually `GaussianIC` instances with central values c_i small errors will do the job. Calls to `Fitter::global_fit` or `Fitter::local_fit` (see Section 3.3 [Minimising the chi-square Function], page 13) will then minimize the value of $\chi^2 + f\chi_c^2$, where χ^2 is the sum of all contributions added via `InputFunction::add`, χ_c^2 is the sum of all contributions added via `InputFunction::add_constraint` and f is the *constraint penalty factor* which can be accessed with the methods

```
double InputFunction::constraint_penalty_factor()
void InputFunction::constraint_penalty_factor(double f)
```

Thus, by varying the constraint penalty factor you can change the weight of the constraint penalties in the overall fit. The factor must be tuned on a case by case basis. If it is too small the fit will violate the constraints in order to decrease the value of χ^2. If it is too big your fits will not converge because the curvature of the objective function is too large. The default setting is 1.

Instead of changing the constraint penalty factor you can also change the errors of the input components added via `LikelihoodFunction::add_constraint`. In this sense, the constraint penalty factor is just a convenient way of scaling the contributions of all constaint penalties simultaneously. Note, however, that the constraint penalty value written to `Model::constraint_penalty` by the functions `Fitter::global_fit` and `Fitter::local_fit` (see Section 3.3 [Minimising the chi-square Function], page 13) is χ_c^2, i.e. does *not* include the constraint penalty factor.

To implement constraints of the form $c_1 < g(\xi) < c_2$ you can pass a `GaussianIC` instance with non-zero systematic errors to `InputFunction::add_constraint`. If `MyModel::O_G` is the index associated with the observable g, *c1* and *c2* the lower and upper bounds and *dg* the small error you can do the following:

```
fitter.input_function().add_constraint(
    GaussianIC(MyModel::O_G, dg, (c1+c2)/2, (c2-c1)/2));
```

Alternatively, you can define an observable $h(\xi)$ in such a way that it is only non-zero when $g(\xi) < c_1$ or $g(\xi) > c_1$ and add a Gaussian constraint on h with a central value of 0.

3.3 Minimising the chi-square Function

Once the input function of your `Fitter` object is properly initialised, you can fit a model object *model* to some input data *idata* with the methods

```
int Fitter::local_fit(Model& model, const ObservableVector& idata)
int Fitter::global_fit(Model& model, const ObservableVector& idata)
```

These methods take the values in *idata* as measured values of the observables and then minimise the input function. They return zero if the fit was successful and a non-zero value otherwise. If the last argument is omitted, the default input values stored in the `Fitter::input_function()` object are used. After a successful fit, the parameters of the *model* object (as returned by the `Model::parameter` method) are the best-fit parameters, the observables (as returned by the `Model::observable` method) are the values of the observables at the best-fit point, the chi-square value (as returned by `Model::chisquare()`) is set to the minimal chi-square value and the constraint penalty value (as returned by `Model::constraint_penalty()`) is the value of the constraint penalty at the best-fit point (without the constraint penalty factor, see Section 3.2 [Non-linear Constraints], page 12). In the notation of [arXiv:1207.1446v2], and for given input data x_0 and parameters ξ, the chi-square function is related to the input function D by

$$\chi^2(\xi) \;=\; D(\tilde{x}(\xi), x_0) \;-\; D(x_0, x_0)\ ,$$

where $\tilde{x}(\xi)$ are the observables predicted by the model for the parameters ξ. The same relation holds for the constraint penalty χ_c^2 (see Section 3.2 [Non-linear Constraints], page 12).

The difference between the two fitting methods is that `local_fit` uses the current parameter values of the *model* object as starting point for the minimisation while `global_fit` looks through the dictionary of the *model* object (see Section 3.4 [Finding the Right Starting Point], page 15) to find the best starting point.

To only calculate the chi-square value for the current parameter values (without any minimisation) you can use the method

```
int Fitter::calc(Model& model, const ObservableVector& cvals)
```

It sets `model.chisquare()` to the computed chi-square value, `model.constraint_penalty()` to the computed constraint penalty value and returns zero if the calculation was successful and a non-zero value otherwise. The values in *cvals* are used as experimental inputs for the observables. If the *cvals* argument is omitted the default values in `input_function().central_values()` are used. The `Fitter` class also provides methods to compute the chi-square and constraint penalty values due to some subset of observables. The corresponding methods are

```
int Fitter::calc_contrib(const IndexVector& iv, Model& model,
                         const ObservableVector& cvals)
int Fitter::calc_contrib(int i, Model& model,
                         const ObservableVector& cvals)
int Fitter::calc_contrib(int i1, int i2, Model& model,
                         const ObservableVector& cvals)
```

In the first prototype the observables whose contributions should be included in the computation of the chi-square and the constraint penalty can be specified with an `IndexVector` object containing the corresponding observable indices. If the contribution of only one observable ist needed you can supply the index of that observable to the second prototype. The third prototype calculates the contributions of all observables with index between (and including) *i1* and *i2*. Like the `Fitter::calc` method, the above methods set `model.chisquare()` to the computed chi-square value, `model.constraint_penalty()` to the computed constraint penalty value, use *cvals* as central values for the experimental observables or `input_function().central_values()` if the *cvals* argument is omitted and return zero if the calculation was successful and a non-zero value otherwise. Note that, if your input function contain a component which depends on several observables (like `CorrelatedGaussianIC` components) *my*Fitter has no way of separating the contributions from these observables. In this case the *full* contribution of that component is included as soon as at least *one* of these observables is requested in the call to `Fitter::calc_contrib`.

For the actual minimisations *my*Fitter uses a custom implementation of the BFGS algorithm. You can tune the parameters for this algorithm with several `Fitter` methods:

`int minimizer_verbosity()`
`void minimizer_verbosity(int n)`

These methods return or set the verbosity level for minimisations. The default value is zero, in which case no information is displayed during a minimisation. Values of 1 to 3 will print increasing amounts of information to `std::cout`.

`double minimizer_line_search_precision()`
`void minimizer_line_search_precision(double p)`

These methods return or set the precision for one-dimensional minimsations. The default setting of 0.1 is usually sufficient.

`double minimizer_precision()`
`void minimizer_precision(double p)`

These methods return or set the precision of the minimiser. If the norm of the gradient of the chi-square function drops below `minimizer_precision()` the minimisation is considered successful. Remember that internally all parameters are normalised to their *scale* (see [`Model::scale`], page 3), so that derivatives of the chi-square function with respect to the parameters are multiplied with the scale of the corresponding parameter. Thus, unreasonably small values for scales of the parameters can lead to a premature termination of the minimisation. The default setting is 0.001.

`int minimizer_iterations()`
`void minimizer_iterations(int n)`

These methods return or set the maximum number of iterations for a chi-square minimisation. If the maximum number of iterations is exceeded, the minimisation is aborted and the status `GSL_EMAXITER` (defined in the header '`gsl/gsl_errno.h`') is returned. The default setting is 200.

`bool minimizer_keep_hessian()`
`void minimizer_keep_hessian(bool b)`

If this is set to `true` the BFGS estimate for the Hessian matrix from the previous minimisation is used as starting point for the next minimisation. A common application for this feature is minimisation with non-linear constraints (see Section 3.2 [Non-linear Constraints], page 12). In this case convergence problems can be dealt with by first doing a minimisation with a low constraint penalty factor, then increasing the factor and re-starting the minimisation in the state where it terminated. For the default setting of `false` the Hessian matrix is set to the identity matrix at the start of each minimisation.

3.4 Finding the Right Starting Point

To successfully minimise a function one should start the iteration as close as possible to the global minimum. The search for the best starting point is usually done by randomly sampling the parameter space. In *my*Fitter each `Model` object can store a dictionary of such sample points (see Section 2.1 [The Model Base Class], page 3). To build this dictionary the `Model` class provides the `scan` method for performing simple flat scans of the parameter space. However, for particularly difficult objective functions with narrow valleys a flat scan might not be enough to resolve all relevant features. For such cases the `Fitter` class allows you to scan the parameter space *adaptively*. The adaptive scan works as follows: let Ω be the parameter space volume to be scanned and let $\chi^2(\xi)$ be the chi-square function (including constraint penalties). Then *my*Fitter uses VEGAS to compute the integral

$$\int_\Omega d\xi \; (\chi^2(\xi) \; + \; c)^{-a} \; ,$$

where a and c are positive constants to be configured by the user. The value of this integral is completely meaningless, but the adaptive property of the VEGAS algorithm will eventually sample the regions with a small chi-square with a higher density. After giving VEGAS some time to adapt one then starts filling the dictionary of the `Model` class with all the sample points tried by VEGAS. This automatically leads to a higher resolution in the interesting regions with small chi-square values.

The adaptive scan can be done with the methods

```
void Fitter::adaptive_scan(Model& model,
                           const ObservableVector& cvals,
                           HepSource::Int64 nfirst,
                           HepSource::Int64 n,
                           int niter,
                           HepSource::Int64 nlast)
void Fitter::adaptive_scan(Model& model,
                           HepSource::Int64 nfirst,
                           HepSource::Int64 n,
                           int niter,
                           HepSource::Int64 nlast)
```

(The 64 bit integer type `HepSource::Int64` is provided by the Dvegas package.) The adaptation phase of the VEGAS integration consists of a first iteration with *nfirst* shots and then *niter* iterations with *n* shots each. Then a final iteration is done with *nlast* sample points. All the points tried in that last iteration are written to the dictionary of the *model* argument. The chi-square function in the integrand is computed with the `input_function()` member of the `Fitter` object, using the central values *cvals* for the observables if provided and `input_function().central_values()` otherwise.

The constant c in the integrand can be configured with the methods

```
double Fitter::scans_chisquare_offset()
void Fitter::scans_chisquare_offset(double)
```

The default value is 1. The exponent a can be accessed with

```
double Fitter::scans_chisquare_power()
void Fitter::scans_chisquare_power(double)
```

The default value is also 1. Finally, the number of bins used for the VEGAS adaptation can be set with

```
int Fitter::scans_nbins()
void Fitter::scans_nbins(int)
```

The default setting is 50.

4 Calculating p-values

The main feature of the *my*Fitter library is the numerical computation of p-values in likelihood ratio tests of nested and non-nested models. As discussed in [arXiv:1207.1446v2], p-values in likelihood ratio tests have to be computed numerically in cases where Wilks' theorem is not applicable. Such cases include bounded parameters and models which are not *nested*, meaning that one model can not be obtained from the other by fixing some of its parameters.

In a likelihood ratio test one compares the performance of two models A and B in desribing observed data. The test is performed under the *null hypothesis* that one of the models, say, model B, is realised with certain parameters (usually its best-fit parameters for the *measured data* x_0. Then one considers a large ensemble of "toy measurements" which are randomly distributed about their true values (as predicted by model B) according to their experimental errors and uses the difference $\Delta\chi^2 = \chi^2_B - \chi^2_A$ of the minimal chi-square values of the two models as test statistic. The p-value is the probability that a toy measurement leads to a $\Delta\chi^2$ value which is bigger (i.e. more in favour of model A) than the $\Delta\chi^2$ value obtained from the measured data x_0.

4.1 Methods for Computing p-values

The numerical computation of p-values is done by the `Fitter` class via the methods

```
double calc_nested_lrt_pvalue(Model& fullmodel,
                              Model& constrainedmodel)
double calc_lrt_pvalue(Model& model1, Model& model2,
                       int argc=0, char** argv=0)
```

Both methods return the computed p-value. More information about the last p-value computation can be obtained with the methods

```
double Fitter::pvalue()
double Fitter::pvalue_error()
bool Fitter::reached_precision_goal()
HepSource::Int64 Fitter::number_of_shots()
double Fitter::failed_shot_ratio()
```

The `pvalue` method simply returns the result of the last p-value computation. Since the p-value is calculated by numerical Monte Carlo integration it has a statistical error, which is returned by the `pvalue_error` method. The method `reached_precision_goal` returns `true` if the desired precision goal (as set by `dvegas_precision`, see Section 4.2 [Options for p-value Computations], page 18) has been reached in the last p-value integration and `false` otherwise. The total number of shots used to obtain the result is returned by the `number_of_shots` method. The return type is a 64-bit integer and defined by the `Dvegas` package. Each shot requires two minimisations, and a small fraction of these minimisations will usually fail. The fraction of "failed shots" in the last p-value computation is returned by the `failed_shot_ratio` method. If you use parallelised integration (see Section 4.3 [Parallel Computation], page 20) the counting of failed shots does not work and `failed_shot_ratio` will return -1.

The `calc_nested_lrt_pvalue` method is used for comparing nested models, and *constrainedmodel* must be a constrained version of *fullmodel*. If you defined copy constructors and virtual assignment operators for your model classes (see Section 2.2 [Writing your own Model Class], page 6) you will probably want to construct *constrainedmodel* like this:

```
MyModel constrainedmodel = fullmodel;
constrainedmodel.fix(MyModel::P_MYSECONDPAR);
constrainedmodel.fix(MyModel::P_MYLASTPAR);
...
```

The `calc_nested_lrt_pvalue` method checks if the two models have the same number of parameters (as returned by `Model::nparameters()`) and if all the parameters fixed in *fullmodel* are also fixed in *constrainedmodel*, but otherwise it is your responsibility to ensure that the two models are indeed nested. The `calc_nested_lrt_pvalue` method the performs a likelihood ratio test, using the model `constrainedmodel` with its current parameters as the null hypothesis. The difference

```
constrainedmodel.chisquare() - fullmodel.chisquare()
```

is taken as the $\Delta\chi^2$ value obtained from the measured data. So, usually, you will want to fit `fullmodel` and `constrainedmodel` to your data (see Chapter 3 [Fitting a Model], page 10) before passing them to `calc_nested_lrt_pvalue`.

The method

```
double calc_lrt_pvalue(Model& model1, Model& model2,
                       int argc=0, char** argv=0)
```

is used for comparing unrelated models. Here, the only limitation is that both models must provide the same number of observables. It is your responsibility to ensure that the integer values used to identify the observables are the same in both models. The order of the first two arguments is not important in this case: the model with the bigger chi-square value (as returned by `Model::chisquare()`) and its current parameter values is used as null hypothesis. As before, the difference of the two `chisquare()` members is assumed to be the $\Delta\chi^2$ value obtained from the measured data. The last two arguments of `calc_lrt_pvalue` are for parallelising the *p*-value computation. They are discussed in Section 4.3 [Parallel Computation], page 20. If they are omitted, no parallelisation is used.

4.2 Options for *p*-value Computations

The methods `calc_nested_lrt_pvalue` and `calc_lrt_pvalue` use the Dvegas library to compute *p*-values by numerical Monte Carlo integration. The details of the numerical integration and other operations related to the computation of *p*-values can be configured with the following `Fitter` methods:

`double inner_region_fraction()`
`void inner_region_fraction(double f)`

These methods return or set the initial fraction of sample points that are thrown in the "inner region" of the integral, where no non-zero contributions are expected (within some approximation). For details see the discussion in [arXiv:1207.1446v2]. The default setting is 0.1.

`double inner_region_power()`
`void inner_region_power(double` *`alpha`*`)`

These methods return or set the exponent α appearing in the probability density function for the inner region. For details see the discussion in [arXiv:1207.1446v2]. The default setting is 1.

`int dvegas_verbosity()`
`void dvegas_verbosity(int` *`n`*`)`

These methods return or set the verbosity level of the numerical integration. If set to 0, no output is produced. Values of 1 to 3 print increasing amounts of information on `std::cout`. The default setting is 0.

`int dvegas_nbins()`
`void dvegas_nbins(int` *`n`*`)`

These methods return or set the number of bins used in the VEGAS adaptation. More information can be found in the Dvegas documentation. The default setting is 50.

`int dvegas_nfirstshots()`
`void dvegas_nfirstshots(int` *`n`*`)`

These methods return or set the number of sample points (*shots*) for the first iteration of the VEGAS algorithm. The default setting is 1000.

`int dvegas_nshots()`
`void dvegas_nshots(int` *`n`*`)`

These methods return or set the number of sample points (*shots*) for all subsequent iterations of the VEGAS algorithm. The default setting is 500.

`int dvegas_niterations()`
`void dvegas_niterations(int` *`n`*`)`

These methods return or set the maximum number of VEGAS iterations which are performed before terminating the calculation and returning the result. The default setting is 20.

`double dvegas_precision()`
`void dvegas_precision(double` *`p`*`)`

These methods return or set the desired relative precision of the p-value computation. If this relative precision is reached the integration is terminated and the result is returned. The default setting is 0.01.

`double orthogonalization_tolerance()`
`void orthogonalization_tolerance(double` *`t`*`)`

To construct the model hyperplane (see [arXiv:1207.1446v2]) from the derivatives of the observables with respect to the parameters, the `Fitter` class uses the Gram Schmidt orthogonalisation algorithm. These methods return or set the tolerance for the orthogonalisation, i.e. the minimum length of a vector for which it is considered unequal to zero. Remeber that, internally, all derivatives are multiplied with the scale of the corresponding parameter. The default setting is 10^{-6}.

4.3 Parallel Computation

If you installed the Dvegas library with parallelisation features enabled, you can use the last two arguments of `calc_lrt_pvalue` to parallelise your *p*-value computation. Details on the parallelisation Monte Carlo integrations can be found in the Dvegas documentation. Here we just discuss the simplest case.

The arguments *argc* and *argv* are the ones you would pass to the `HepSource::OmniComp` constructor if you were using Dvegas directly. They should (usually) contain the command line arguments passed to your program and thus be identical to the ones system passes to your `main` function. Let's say you have written some program '`myprog`' which calculates the *p*-value (but too slowly) and several computers with access to a network directory *workdir*. On each extra computer available to you, you start a *worker* processes with

```
cd workdir
myprog -w
```

These processes will dump some information about themselves in a common file '`myprog.workers`' in '*`workdir`*' and then wait for the "master" process to show up. Start that one on the last computer with

```
cd workdir
myprog
```

The master process should then connect to the workers and distribute the numerical integration among the workers. It will also do some work itself. Needless to say, if your computers have more than one core, it makes sense to start more than one worker process on each computer.

Appendix A GNU Free Documentation License

ADDENDUM: How to use this License for your documents