MINUIT User's Guide

Fred JAMES
and
Matthias WINKLER
CERN, Geneva

June 16, 2004

Foreword

What MINUIT is intended to do

MINUIT is conceived as a tool to find the minimum value of a multi-parameter function (the "FCN") and analyze the shape of the function around the minimum. The principal application is foreseen for statistical analysis, working on chisquare or log-likelihood functions, to compute the best-fit parameter values and uncertainties, including correlations between the parameters. It is especially suited to handle difficult problems, including those which may require guidance in order to find the correct solution.

What MINUIT is not intended to do

Although MINUIT will of course solve easy problems faster than complicated ones, it is not intended for the repeated solution of identically parametrized problems (such as track fitting in a detector) where a specialized program will in general be much more efficient.

Further remarks

MINUIT was initially written in Fortran around 1975-1980 at CERN by Fred James [1]. Its main field of usage is statistical data analysis of experimental data recorded at CERN, but it is also used by people doing data analysis outside CERN or outside high energy physics (HEP). In 2002 Fred James started a project aiming to re-implement MINUIT in an object-oriented way using C++ .

More information about recent developments, releases and installation can be obtained from the MINUIT homepage [2].

The names of MINUIT applications are written in capital letters (e.g. MIGRAD, MINOS, CONTOURS), the corresponding names of the C++ classes are written using sans-serif font type (MnMigrad, MnMinos, MnContours).

Contents

List of Figures

1 Introduction: MINUIT basic concepts

1.1 The organization of MINUIT

The MINUIT package acts on a multiparameter *objective function* which is called — for historical reasons — the `FCN` function (see 3.1). This function is usually a chisquared or a log–likelihood, but it could also be a mathematical function. The `FCN` function needs to be written in C++ for which MINUIT defines the pure abstract base class FCNBase as interface. The user must define and implement the `FCN` function in a derived class from FCNBase. Sometimes this is done by an intermediate program such as HippoDraw[3], in which case MINUIT is being used under the control of such an intermediate program[1]. The value of the `FCN` function will in general depend on one or more variable parameters whose meaning is defined by the user (or by the intermediate program), but whose trial values are determined by MINUIT .

To take a simple example, suppose the problem is to fit a polynomial through a set of data points. Then the user would write a `FCN` which calculates the χ^2 between a polynomial and the data; the variable parameters of `FCN` would be the coefficients of the polynomials. Using objects for minimization from MINUIT , the user would request MINUIT to minimize the `FCN` with respect to the parameters, that is, find those values of the coefficients which give the lowest value of chisquare.

The user must therefore supply, in addition to the function to be analyzed, via a set or sequence of MINUIT applications the instructions which analysis is wanted. The instructions are coded in C++ in the calling program (main.cpp), which allows looping, conditional execution, and all the other possibilities of C++ , but not interactivity, since it must be compiled before execution.

1.2 Design aspects of MINUIT in C++

What MINUIT is:

- platform independent
- written in an object-oriented way using standard C++
- independent of any external package

The maintainability should be guaranteed with the choice of a modern computer language. Choosing object-oriented technology MINUIT should profit from an increased flexibility and functionality and make it also extendable (recursivness, new algorithms, new functionality).

[1] ROOT [4] uses its own C++ version of the Fortran MINUIT when this manual was written. However an interface for this C++ version exists and the library can be loaded dynamically on demand

What MINUIT does not:

- histogramming
- data handling
- graphics

MINUIT is kept as a low-level package with optimal performance.

The main usages of MINUIT are

- from a user's program (such as int main()...)
- from a graphical data analysis tool such as HippoDraw[3]

The most important goals of MINUIT in C++ are

- its numerical accuracy (equivalent to its Fortran version)
- its computational performance (equivalent to its Fortran version)

For the design of the application programming interface (API) of MINUIT a two-way strategy was imposed:

- a minimal required interface with minimum interaction with MINUIT objects and with appropriate usage of the standard C++ library (STL): the user's implementation of the FCNBase class, initial parameter values and uncertainties are provided by the to MINUIT user via std::vectors.
- a rich interface which provides the user with more functionality such as interaction with parameters.

The core of the minimization functionality and related tools (the kernel of MINUIT) should be clearly separated from the user, who is interfacing via defined user interfaces (the API).

1.3 Internal and external parameters

Each of the parameters to the `FCN` is defined by the user as belonging to one of the following types:

- Freely variable: allowed to take on any value.

- Variable with double sided limits: allowed to vary only between two limits specified by the user.
- Variable with single sided limits: allowed to vary only between one upper or one lower limit specified by the user and unlimited to the other side.
- Fixed: originally defined as variable, but now taking on only the value the parameter had at the moment it was fixed, or a value later assigned by the user.
- Constant: taking on only one value as specified by the user.

When using the minimal required interface, all variable parameters are free and unbound.

The user, in his `FCN`, must of course be able to "see" all types of defined parameters, and he therefore has access to what we call the *external parameter list*, that is, the parameters as he defined them. On the other hand, the internal MINUIT minimizing routines only want to "see" variable parameters without limits, and so they have access only to the *internal parameter list* which is created from the external list by the following transformation:

- Squeeze out all parameters that are not variable.
- Transform all variable parameters with limits, so that the transformed parameter can vary without limits. (See the next section for details concerning this transformation.) Because this transformation is non-linear, it is recommended to avoid putting limits on parameters where they are not needed.

As an example, suppose that the user has defined the following parameters:

- Parameter 0, constant.
- Parameter 1, freely variable.
- Parameter 2, variable with limits.
- Parameter 3, constant.
- Parameter 4, freely variable.

Then the internal parameter list would be as follows:

- Internal parameter 0 = external parameter 1.
- Internal parameter 1 = external parameter 2, transformed appropriately.

- Internal parameter 2 = external parameter 4.

In the above example, MINUIT considers that the number of external parameters is 5, and the number of internal parameters is 3. This is the number which determines, for example, the size of the error matrix of the parameters, since only variable parameters have errors.

An important feature of MINUIT is that parameters are allowed to change types during the MINUIT minimization and analysis of a `FCN` function. Several applications in MINUIT have methods available to make variable parameters fixed and vice-versa; to impose, change, or remove limits from variable parameters; and even to define completely new parameters at any time during a run. In addition, some MINUIT applications (notably the `MINOS` error analysis) cause one or more variable parameters to be temporarily fixed during the calculation. Therefore, the correspondence between external and internal parameter lists is in general a dynamic one, and the number of internal parameters is not necessarily constant.

For more details about parameter interaction see 4.14.

1.3.1 The transformation for parameters with limits

For variable parameters with double sided limits a (lower) and b (upper), MINUIT uses the following transformation:

$$P_{\text{int}} = \arcsin\left(2\,\frac{P_{\text{ext}} - a}{b - a} - 1\right) \tag{1.1}$$

$$P_{\text{ext}} = a + \frac{b - a}{2}\,(\sin P_{\text{int}} + 1) \tag{1.2}$$

so that the internal value P_{int} can take on any value, while the external value P_{ext} can take on values only between the lower limit a and the upper limit b. Since the transformation is necessarily non-linear, it would transform a nice linear problem into a nasty non-linear one, which is the reason why limits should be avoided if not necessary. In addition, the transformation does require some computer time, so it slows down the computation a little bit, and more importantly, it introduces additional numerical inaccuracy into the problem in addition to what is introduced in the numerical calculation of the `FCN` value. The effects of non-linearity and numerical roundoff both become more important as the external value gets closer to one of the limits (expressed as the distance to nearest limit divided by distance between limits). The user must therefore be aware of the fact that, for example, if he puts limits of $(0, 10^{10})$ on a parameter, then the values 0.0 and 1.0 will be indistinguishable to the accuracy of most machines.

For this purpose single sided limits on parameters are provided by MINUIT , with their transformation being:

Lower bound a:

$$P_{\text{int}} = \pm\sqrt{(P_{\text{ext}} - a + 1)^2 - 1} \tag{1.3}$$

$$P_{\text{ext}} = a - 1 + \sqrt{P_{\text{int}}^2 + 1} \tag{1.4}$$

Upper bound b:

$$P_{\text{int}} = \pm\sqrt{(b - P_{\text{ext}} + 1)^2 - 1} \tag{1.5}$$

$$P_{\text{ext}} = b + 1 - \sqrt{P_{\text{int}}^2 + 1} \tag{1.6}$$

The transformation of course also affects the parameter error matrix, so MINUIT does a transformation of the error matrix (and the "parabolic" parameter errors) when there are parameter limits. Users should however realize that the transformation is only a linear approximation, and that it cannot give a meaningful result if one or more parameters is very close to a limit, where $\partial P_{\text{ext}}/\partial P_{\text{int}} \approx 0$. Therefore, it is recommended that:

- Limits on variable parameters should be used only when needed in order to prevent the parameter from taking on unphysical values.
- When a satisfactory minimum has been found using limits, the limits should then be removed if possible, in order to perform or re-perform the error analysis without limits.

Further discussion of the effects of parameter limits may be found in the last chapter.

1.4 MINUIT strategy

At many places in the analysis of the `FCN` (the user provided function), MINUIT must decide whether to be "safe" and waste a few function calls in order to know where it is, or to be "fast" and attempt to get the requested results with the fewest possible calls at a certain risk of not obtaining the precision desired by the user. In order to allow the user to influence these decisions, there is a MINUIT class MnStrategy (see 4.12) which the user can use to put different settings. In the current release, this MnStrategy can be instantiated with three different minimization quality levels for low (0), medium (1) and high (2) quality. Default settings for iteration cycles and tolerances are initialized then. The default setting is set for medium quality. Value 0 (low) indicates to MINUIT that it should economize function calls; it is intended for cases where there are many variable parameters and/or the function takes a long time to calculate and/or the user is not interested in very precise values for parameter errors. On the other hand, value 2 (high) indicates that MINUIT is allowed to waste function calls in order to be sure that all values are precise; it is

intended for cases where the function is evaluated in a relatively short time and/or where the parameter errors must be calculated reliably. In addition all constants set in MnStrategy can be changed individually by the user, e.g. the number of iteration cycles in the numerical gradient.

1.5 Parameter errors

MINUIT is usually used to find the "best" values of a set of parameters, where "best" is defined as those values which minimize a given function, `FCN`. The width of the function minimum, or more generally, the shape of the function in some neighbourhood of the minimum, gives information about the *uncertainty* in the best parameter values, often called by physicists the *parameter errors*. An important feature of MINUIT is that it offers several tools to analyze the parameter errors.

1.5.1 `FCN` normalization and the error definition

Whatever method is used to calculate the parameter errors, they will depend on the overall (multiplicative) normalization of `FCN`, in the sense that if the value of `FCN` is everywhere multiplied by a constant β, then the errors will be decreased by a factor $\sqrt{\beta}$. Additive constants do not change the parameter errors, but may imply a different goodness-of-fit confidence level.

Assuming that the user knows what the normalization of his `FCN` means, and also that he is interested in parameter errors, the user can change the error definition which allows him to define what he means by one "error", in terms of the change in the `FCN` value which should be caused by changing one parameter by one "error". If the `FCN` is the usual chisquare function (defined below) and if the user wants the usual one-standard-deviation errors, then the error definition (return value of the FCNBase::up() method, see 3.1.2) should be 1.0. If the `FCN` is a negative-log-likelihood function, then the one-standard-deviation value for FCNBase::up() to return is 0.5. If the `FCN` is a chisquare, but the user wants two-standard-deviation errors, then FCNBase::up() should return = 4.0, etc.

Note that in the usual case where MINUIT is being used to perform a fit to some experimental data, the parameter errors will be proportional to the uncertainty in the data, and therefore meaningful parameter errors cannot be obtained unless the measurement errors of the data are known. In the common case of a least-squares fit, `FCN` is usually defined as a chisquare:

$$\chi^2(\alpha) = \sum_{i=1}^{n} (\frac{f(x_i, \alpha) - m_i)^2}{\sigma_i^2} \tag{1.7}$$

where α is the vector of free parameters being fitted, and the σ_i are the uncertainties

in the individual measurements m_i. If these uncertainties are not known, and are simply left out of the calculation, then the fit may still have meaning, but not the quantitative values of the resulting parameter errors. (Only the relative errors of different parameters with respect to each other may be meaningful.)

If the σ_i are all overestimated by a factor β, then the resulting parameter errors from the fit will be overestimated by the same factor β.

1.5.2 The error matrix

The MINUIT processors `MIGRAD` (MnMigrad, see 4.6) and HESSE (MnHesse, see 4.4) (normally) produce an error matrix. This matrix is twice the inverse of the matrix of second derivatives of the `FCN`, transformed if necessary into external coordinate space[2], and multiplied by FCNBase::up(). Therefore, errors based on the MINUIT error matrix take account of all the parameter correlations, but not the non-linearities. That is, from the error matrix alone, two-standard-deviation errors are always exactly twice as big as one-standard-deviation errors.

When the error matrix has been calculated (for example by the successful execution of `MIGRAD` (MnMigrad::operator(), see 4.6.4) or HESSE (MnHesse::operator())) then the parameter errors printed by MINUIT are the square roots of the diagonal elements of this matrix. The covariance or the correlations can be printed and shown via std::cout as the ostream operator operator<< is overloaded. The eigenvalues of the error matrix can be calculated using MnEigen, which should all be positive if the matrix is positive-definite (see below on `MIGRAD` and positive-definiteness).

The effect of correlations on the individual parameter errors can be seen as follows. When parameter `n` is fixed (e.g. via the method MnMigrad::fix(n)), MINUIT inverts the error matrix, removes the row and column corresponding to parameter `n`, and re-inverts the result. The effect on the errors of the other parameters will in general be to make them smaller, since the component due to the uncertainty in parameter `n` has now been removed. (In the limit that a given parameter is uncorrelated with parameter `n`, its error will not change when parameter `n` is fixed.) However the procedure is not reversible, since MINUIT forgets the original error matrix, so if parameter `n` is then released (e.g. via the method MnMigrad::release(n)), the error matrix is considered as unknown and has to be recalculated with appropriate commands.

[2]The *internal error matrix* maintained by MINUIT is transformed for the user into *external coordinates*, but the numbering of rows and columns is of course still according to internal parameter numbering, since one does not want rows and columns corresponding to parameters which are not variable. The transformation therefore affects only parameters with limits; if there are no limits, internal and external error matrices are the same.

1.5.3 MINOS errors

The MINUIT processor MINOS (MnMinos, see 4.8) was probably the first, and may still be the only, generally available program to calculate parameter errors taking into account both parameter correlations and non-linearities. The MINOS error intervals are in general assymmetric, and may be expensive to calculate, especially if there are a lot of free parameters and the problem is very non-linear.

MINOS can only operate after a good minimum has already been found, and the error matrix has been calculated, so the MINOS error analysis will normally follow a MIGRAD minimization. The MINOS error for a given parameter is defined as the change in the value of that parameter which causes F' to increase by the amount FCNBase::up(), where F' is the minimum of FCN with respect to all *other* free parameters, and FCNBase::up() is the return value of the error definition specified by the user (default = 1.).

The algorithm for finding the positive and negative MINOS errors for parameter n consists of varying parameter n, each time minimizing FCN with respect to all the other npar - 1 variable parameters, to find numerically the two values of parameter n for which the minimum of FCN takes on the values $F_{\min} + \mathtt{up}$, where $F_{\min}$ is the minimum of FCN with respect to all npar parameters. In order to make the procedure as fast as possible, MINOS uses the error matrix to predict the values of all parameters at the various sub-minima which it will have to find in the course of the calculation, and in the limit that the problem is nearly linear, the predictions of MINOS will be nearly exact, requiring very few iterations. On the other hand, when the problem is very non-linear (i.e., FCN is far from a quadratic function of its parameters), is precisely the situation when MINOS is needed in order to indicate the correct parameter errors.

1.5.4 CONTOURS plotting

MINUIT offers a procedure for finding FCN CONTOURS (provided via the class MnContours, see 4.2).

The contour calculated by MnContours::operator() is dynamic, in the sense that it represents the minimum of FCN with respect to all the other npar - 2 parameters (if any). In statistical terms, this means that MnContours takes account of the correlations between the two parameters being plotted, and all the other variable parameters, using a procedure analogous to that of MINOS. (If this feature is not wanted, then the other parameters must be fixed before calling CONTOURS.) MnContours provides the actual coordinates of the points around the contour, suitable for plotting with a graphics routine or by hand (using MnPlot, see 4.9). The points are given in counter-clockwise order around the contour. Only one contour is calculated per command, and the level is $F_{\min} + \mathtt{up}$. where up is the return value of FCNBase::up() specified by the user (usually 1.0 by default). The number of points to be calculated is chosen by the user (default is 20). As a by-product, CONTOURS provides the MINOS

errors of the two parameters in question, since these are just the extreme points of the contour (use the MnContours::contour(...) method in order to get the points of the contour and the ones of the MINOS errors). MnContours::operator() returns a std::vector<std::pair<double,double>> of (x,y) points. Using MnPlot::operator() will generate a text graphics plot in the terminal.

2 MINUIT installation

2.1 MINUIT releases

To follow the current release process the user is referred to the MINUIT homepage [2].

MINUIT was re–implemented in C++ from 2002–2004, but the functionality is largely compatible with the one of the Fortran-77 version. The usage is different in the sense that the re–write from Fortran-77 to C++ was done by its signification and not literally (with minor exceptions). Applications such as `MIGRAD` have a corresponding C++ class MnMigrad, Fortran-77 MINUIT "commands" became classes or methods of classes according to their purpose. Users familiar with the Fortran-77 version of MINUIT , who have not yet used releases from the C++ version, should however read this manual, in order to adapt to the changes as well as to discover the new features and easier ways of using old features.

2.2 Install MINUIT using autoconf/make

For each release of MINUIT a tar.gz file is provided for downloading from the MINUIT homepage [2]. For non-UNIX platforms please refer to the MINUIT homepage.

The necessary steps to follow are:

1. download the tar.gz by clicking on it from the release page
2. unzip it:

   ```
   $ unzip Minuit-x.x.x.tar.gz
   ```

3. untar it:

   ```
   $ tar xvf Minuit-x.x.x.tar
   ```

4. step down to the created Minuit-x.x.x directory:

   ```
   $ cd Minuit-x.x.x/
   ```

5. run the "configure" script:

   ```
   $ ./configure
   ```

6. run "make" to compile the source code:

   ```
   $ make
   ```

7. run "make check" to create the executable example:

   ```
   $ make check
   ```

8. run the executable example:

   ```
   $ tests/MnTutorial/Quad4FMain.C
   ```

The output should look like that:

```
Minuit did successfully converge.

# of function calls: 74
minimum function value: 1.12392e-09
minimum edm: 1.12392e-09
minimum internal state vector: LAVector parameters:

 -1.82079e-05
 -1.20794e-05
  6.22382e-06
  -3.0465e-05


minimum internal covariance matrix: LASymMatrix parameters:

            4            1            2  2.70022e-18
            1            5            3  1.87754e-17
            2            3            6  2.29467e-17
  2.70022e-18  1.87754e-17  2.29467e-17            1


# ext. ||   name    ||   type  ||   value   ||  error +/-

   0   ||         x ||  free   ||-1.821e-05 ||       2
   1   ||         y ||  free   ||-1.208e-05 ||   2.236
   2   ||         z ||  free   || 6.224e-06 ||   2.449
   3   ||         w ||  free   ||-3.047e-05 ||       1



MnUserCovariance:

            4            1            2  2.70022e-18
```

```
           1            5            3  1.87754e-17
           2            3            6  2.29467e-17
 2.70022e-18  1.87754e-17  2.29467e-17            1

MnUserCovariance parameter correlations:

           1     0.223607     0.408248  1.35011e-18
    0.223607            1     0.547723  8.39663e-18
    0.408248     0.547723            1  9.36796e-18
 1.35011e-18  8.39663e-18  9.36796e-18            1


MnGlobalCorrelationCoeff:

    0.408248
    0.547723
    0.621261
           0
```

2.3 CVS code repository

How to check out (–in) code from the CVS code repository is described at the MINUIT homepage [2]. To get the source code from the CVS repository one needs to do:

Kerberos IV authorization:

```
$ setenv CVSROOT :kserver:SEAL.cvs.cern.ch:/cvs/SEAL

$ cvs co MathLibs/Minuit
```

Anonymous read-only access (if it's enabled by the librarian, see details):

```
$ setenv CVSROOT :pserver:anonymous@SEAL.cvs.cern.ch:/cvs/SEAL

$ cvs login
```

(Logging in to :pserver:anonymous@seal.cvs.cern.ch:2401/cvs/SEAL) CVS password:cvs

```
$ cvs co MathLibs/Minuit
```

(If you want to check out a tagged version SEAL_x_x_x of MINUIT, then do

```
$ cvs co -r SEAL\_x\_x\_x MathLibs/Minuit )
```

2.4 Create a tar.gz from CVS

Once the sources are checked out from the CVS code repository,

1. change to the directory:

   ```
   $ cd MathLibs/Minuit
   ```

2. run autogen:

   ```
   $ ./autogen
   ```

3. create a new directory:

   ```
   $ cd ..
   $ mkdir Minuit-BUILD
   $ cd Minuit-BUILD/
   ```

4. run configure:

   ```
   $ ../Minuit/configure
   ```

5. create the tar.gz:

   ```
   $ make dist
   ```

This will create a Minuit-x.x.x.tar.gz which can be distributed and used as described above.

2.5 MINUIT versions

The version numbers of MINUIT follow the release numbers of the SEAL project [5] at CERN [6].

2.5.1 From Fortran-77 to C++

The program is entirely written in standard portable C++ . MINUIT does not depend on any external library. In its minimal usage the user must only provide an implementation of the FCNBase class to MINUIT and parameters and uncertainties in form of std::vector containers.

2.5.2 Memory allocation and thread safety

Differently to the Fortran-77 version of MINUIT , the C++ version has its own memory manager (StackAllocator. The user can select between the standard dynamic memory allocation and deallocation (default) and performance-optimized stack–like allocation (optional). However, the library is not thread save using stack–allocation.

2.5.3 MINUIT parameters

Differently to the Fortran-77 version of MINUIT there is no limit on the number of parameters, variable or non-variable. Memory allocation is done dynamically according to the actual needs and "on demand". There is no protection against an upper limit on the number of parameters, however the "technological" limitations of MINUIT can be seen around a maximum of 15 free parameters at a time.

2.6 Interference with other packages

The new MINUIT has been designed to interfere as little as possible with other programs or packages which may be loaded at the same time. MINUIT is thread safe by default. Optionally the user can select a different way of dynamically allocating memory in the class StackAllacator for MINUIT , in which case (and after an entire recompiliation of the whole library) the thread safety is lost.

2.7 Floating-point precision

MINUIT is entirely based on C++ double precision. The actual floating point precision of double precision (32–bit or 64–bit) is platform dependent and can even vary on the same platform, depending on whether a floating point number is read from memory a CPU register.

The argument of the user's implementation of FCNBase::operator() is therefore a std:vector<double>. MINUIT expects that the calculations inside `FCN` will be performed approximately to the same accuracy.

The accuracy MINUIT expects is called *machine precision* (MnMachinePrecision, see 4.5) and can be printed on demand using std::cout. If the user fools MINUIT by making internal `FCN` computations in single precision, MINUIT will interpret roundoff noise as significant and will usually either fail to find a minimum, or give incorrect values for the parameter errors.

It is therefore recommended to make sure that all computations in `FCN`, as well as all methods and functions called by `FCN`, are done in double precision. If for some reason the computations cannot be done to a precision comparable with that expected by

MINUIT , the user **must** inform MINUIT of this situation with setting a different machine precision via the MnMachinePrecision::setPrecision(double) method.

With reduced precision, the user may find that certain features sensitive to first and second differences (`HESSE`, `MINOS`, `CONTOURS`) do not work properly, in which case the calculations must be performed in higher precision.

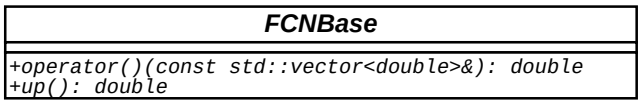

Figure 3.1: The base class FCNBase for the user's FCN.

3 How to use Minuit

3.1 The FCN Function

The user must always implement a derived class of FCNBase (the "FCN") which calculates the function value to be minimized or analyzed.

Note that when MINUIT is being used through an intermediate package such as HippoDraw [3], then the user's FCN may be supplied by the this package.

The name of the user's class to implement the FCNBase interface may be chosen freely (in documentation we give it the generic name FCN).

3.1.1 FCNBase::operator()(const std::vector<double>&)

The meaning of the vector of parameters std::vector<double> in the argument of FCNBase::operator() are of course defined by the user, who uses the values of those parameters to calculate his function value. The order and the position of these parameters is strictly the one specified by the user when supplying the starting values for minimization.

The starting values must be specified by the user, either via an std::vector<double> or the MnUserParameters (see 4.14) supplied as input to the MINUIT minimizers such as VariableMetricMinimizer or MnMigrad (see 4.6). Later values are determined by MINUIT as it searches for the minimum or performs whatever analysis is requested by the user.

3.1.2 FCNBase::up()

Returns the value of up (default value = 1.), defining parameter errors. MINUIT defines parameter errors as the change in parameter value required to change the function value by up. Normally, for chisquared fits up = 1, and for negative log likelihood, up = 0.5.

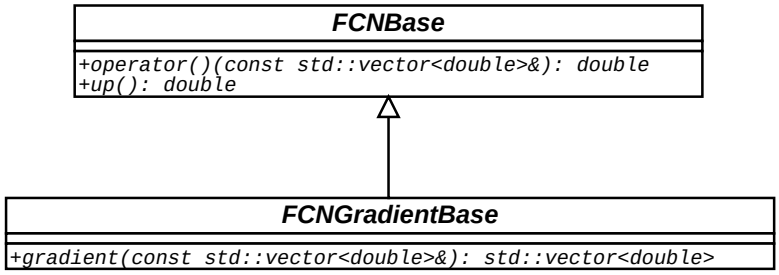

Figure 3.2: The base classe FCNGradientBase for the user's `FCN` with gradient

3.1.3 `FCN` function with gradient

By default first derivatives are calculated numerically by MINUIT . In case the user wants to supply his own gradient calculator (e.g. analytical derivatives), he needs to implement the FCNGradientBase interface.

The size of the output vector is the same as of the input one. The same is true for the position of the elements (first derivative of the function with respect to the n_{th} variable has index n in the output vector).

3.2 MINUIT parameters

Interaction with the parameters of the function are essential both for MINUIT and the user. Different interfaces are provided, depending on the level of interaction.

3.2.1 Minimal required interface

Starting values of parameters and uncertainties can be provided to MINUIT by the user via std::vector<double> vector containers. Any interaction with the parameters before minimization (fix, release, limits, etc.) is not possible then.

Optionally if the user wants to provide starting values for the covariance, he has to provide the values in a std::vector<double> vector container stored in upper triangular packed storage format (see 4.13).

3.2.2 MnUserParameters

A more functional interface to the user parameters is provided through MINUIT via the class MnUserParameters. The user can add parameters giving them a name and starting values. More information can be found in 4.14.

3.2.3 MnUserCovariance

The user can (optionally) provide a covariance matrix as input using the class MnUserCovariance. More information can be found in 4.13.

3.2.4 MnUserParameterState

The MnUserParameterState contains the parameters (MnUserParameters) and covariance (MnUserCovariance). The MnUserParameterState has to main purposes:

- It can be used as input to minimization.
- The result of the minimization is transformed into the user representable format by MINUIT .

For more explanations see 4.15.

3.3 Input to MINUIT

The following input combinations to MINUIT are possible:

- `FCN` + parameters + uncertainties
- `FCN` with gradient + parameters + uncertainties
- `FCN` + parameters + covariance
- `FCN` with gradient + parameters + covariance

For each of these combinations the user can chose between a minimal required interface (using std::vector containers) for the parameters and uncertainties or a more functional one provided by MINUIT . If the user wants to interact with the parameters before minimization (fixing, adding/removing limits), the minimal required interface cannot be used.

3.3.1 What the user must supply

The user must supply to MINUIT

- a valid implementation of the FCNBase base class
- parameters with their starting values
- expected uncertainties on the parameters

The input parameters can be simply defined via an std::vector<double>, which means that all parameters are variables. If the user wants fix a parameter or put limits on it before minimization, he has to instantiate a MnUserParameters object and then add parameters one by one, giving them a name, value, uncertainty. Once all parameters are added to MnUserParameters, he can fix a parameter or put limits on another one before handing them over to Minit for minimization.

3.3.2 What the user can supply

Optionally the user can supply his own gradient calculator by implementing the FCNGradientBase interface or supply a full covariance matrix for input if one is available. The covariance matrix can be supplied in form of a std::vector<double> in packed storage format (upper triangular), or in a more user-friendly way by using the interface provided by the MnUserCovariance.

3.4 Running a MINUIT minimization

Two use cases are addressed for minimization:

- The user just wants the function to be minimized in one go.
- The user wants to minimize the `FCN` in several minimization steps, re-using the result of the preceeding minimization in the next step and change parameters in between (fix/release/put limits on them, etc.).

How MINUIT minimizations can be performed is shown in 6.2.

3.4.1 Direct usage of minimizers

Minimizers such as the VariableMetricMinimizer are designed as state-less minimization engines, which means that they do not depend on the current function and its parameters. Any `FCN` function can be minimized with the same minimizer. The interface is restricted to minimization and no parameter interaction is possible.

3.4.2 Using an application (MnMigrad)

MnMigrad uses the VariableMetricMinimizer for minimization but allows as well for parameter interaction by the user. An instance of MnMigrad is specific to the current `FCN` and user parameters. Any parameter interaction of the user between two minimization steps will make use of the result of the preceeding minimization in an optimal way. The interface for parameters (see 4.14, 4.13 and 4.15) is forwardedin MnMigrad.

3.4.3 Subsequent minimizations

MINUIT takes care that all information is treated in an optimal and correct wayif the user interacts with the parameters between two minimizations. MnMigrad retains the result of the last minimization and uses this as input for the next step. Between two minimization requests to MnMigrad the user can fix or release parameters, put limits on them or remove limits from them. Each instance of a MnMigrad object maintains its own state and one can have multiple instances of MnMigrad objects.

3.4.4 MINUIT fails to find a minimum

If MINUIT fails to find a minimum, the user is notified by a warning message isued by MINUIT when running into troubles. Problems can be:

- a bug in MINUIT
- an error in the `FCN`
- a highly difficult problem (usually strong correlations among parameters)
- floating–point precision

3.5 The output from minimization

3.5.1 The FunctionMinimum

The output of the minimizers is the FunctionMinimum. The FunctionMinimum contains the result of the minimization in both internal parameter representation and external parameter representation.

3.5.2 User representable format: MnUserParameterState

On request, the result of the minimzation is transformed into a user representable format for parameters and errors, the MnUserParameterState.

3.5.3 Access values, errors, covariance

The result can be accessed via methods like MnUserParameterState::value(unsigned int n) and MnUserParameterState::error(unsigned int n), where n is the index of the parameter in the list of parameters defined by the user.

3.5.4 Printout of the result

The FunctionMinimum can be printed on the output simply via std::cout. It will print both the internal and external state, that is parameters, errors and the covariance matrix (if available). It also tells the user if MINUIT did converge or not by issueing an appropriate message. If a covariance matrix is available, the global correlation coefficients are printed as well.

Global correlation coefficients

The global correlation coefficient for parameter n is a number between zero and one which gives the correlation between parameter n and that linear combination of all other parameters which is most strongly correlated with n.

4 MINUIT application programming interface (API)

4.1 FunctionMinimum

The FunctionMinimum is the output of the minimizers and contains the minimzation result. The state at the minimum is available both in internal and external representations. For the external representations the return methods

- FunctionMinimum::userState(),
- FunctionMinimum::userParameters() and
- FunctionMinimum::userCovariance()

are provided. These can be used as new input to a new minimization after some manipulation. The parameters and/or the FunctionMinimum can be printed using std::cout.

4.1.1 isValid()

In general, if the method bool FunctionMinimum::isValid() returns "true", the minimizer did find a minimum without running into troubles. However, in some cases it may happen that a minimum cannot be found, then the return value will be "false". Reasons for the minimization to fail are

- the number of allowed function calls has been exhausted
- the minimizer could not improve the values of the parameters (and knowing that it has not converged yet)
- a problem with the calculation of the covariance matrix

Additional methods for the analysis of the state at the minimum are provided.

4.1.2 fval(), edm(), nfcn()

The method double FunctionMinimum::fval() returns the function value at the minimum, the method double FunctionMinimum::edm() returns the expected *vertical* distance to the minimum EDM and unsigned int FunctionMinimum::nfcn() returns the total number of function calls during the minimization.

4.2 MnContours

4.2.1 MnContours(const FCNBase&, const FunctionMinimum&)

Construct an MnContours object from the user's FCN and a valid FunctionMinimum. Additional constructors for user specific MnStrategy settings are provided.

4.2.2 operator()

The method MnContours::operator()(unsigned int parx, unsigned int pary, unsigned int npoints = 20) const calculates one function contour of `FCN` with respect to parameters `parx` and `pary`. The return value is a std::vector<std::pair<double,double> > of (x,y) points. `FCN` minimized always with respect to all other `n - 2` variable parameters (if any). MINUIT will try to find `npoints` points on the contour (default 20). To calculate more than one contour, the user needs to set the error definition 3.1.2 in its `FCN` to the appropriate value for the desired confidence level and call the method MnContours::operator() for each contour.

4.2.3 contour(...)

MnContours::contour(unsigned int parx, unsigned int pary, unsigned int npoints = 20) causes a `CONTOURS` error analysis and returns the result in form of ContoursError. As a by-product ContoursError keeps the MinosError information of parameters `parx` and `pary`. The result ContoursError can be easily printed using std::cout.

4.3 MnEigen

MnEigen calculates and the eigenvalues of the user covariance matrix MnUserCovariance.

4.3.1 MnEigen()

MnEigen is instantiated via default constructor.

4.3.2 operator()

operator()(const MnUserCovariance&) const will perform the calculation of the eigenvalues of the covariance matrix and return the result in form of a std::vector¡double¿. The eigenvalues are ordered from the smallest first to the largest eigenvalue.

4.4 MnHesse

With MnHesse the user can instructs MINUIT to calculate, by finite differences, the Hessian or error matrix. That is, it calculates the full matrix of second derivatives of the function with respect to the currently variable parameters, and inverts it.

4.4.1 MnHesse()

The default constructor of MnHesse() will use default settings of MnStrategy. Other constructors with user specific MnStrategy settings are provided as well.

4.4.2 operator()

The MnHesse::operator() is overloaded both for internal (MINUIT) and external (user) parameters. External parameters can be specified as std::vector<double> or as MnUserParameters. The return value is always a MnUserParameterState.

The optional argument `maxcalls` specifies the (approximate) maximum number of function calls after which the calculation will be stopped.

4.5 MnMachinePrecision

4.5.1 MnMachinePrecision()

MINUIT determines the nominal precision itself in the default constructor MnMachinePrecision().

4.5.2 setPrecision(double eps)

Informs MINUIT that the relative floating point arithmetic precision is `eps`. The method can be used to override MINUIT 's own determination, when the user knows that the `FCN` function value is not calculated to the nominal machine accuracy. Typical values of `eps` are between 10^{-5} and 10^{-14}.

4.6 MnMigrad and VariableMetricMinimizer

MnMigrad provides minimization of the function by the method of `MIGRAD`, the most efficient and complete single method, recommended for general functions (see also 4.7), and the functionality for parameters interaction. It also retains the result from the last minimization in case the user may want to do subsequent minimization steps with parameter interactions in between the minimization requests. The minimization

is done by the VariableMetricMinimizer. Minimization of the function can be done by directly using the VariableMetricMinimizer if no parameters interaction is required. The minimization produces as a by-product the error matrix of the parameters, which is usually reliable unless warning messages are produced.

4.6.1 MnMigrad(const FCNBase&, const std::vector<double>&, const std::vector<double>&, unsigned int)

Constructor for the minimal required interface: `FCN` and starting values for parameters and uncertainties. Optional the strategy level in MnStrategy can be specified.

4.6.2 MnMigrad(const FCNBase&, const MnUserParameters&, unsigned int)

Constructor for high level parameters interface. Optional the strategy level in MnStrategy can be specified.

4.6.3 MnMigrad(const FCNBase&, const MnUserParameterState&, const MnStrategy&)

Constructor from a full state (parameters + covariance) as starting input plus the desired strategy.

4.6.4 operator()

MnMigrad::operator()(unsigned int maxfcn, double tolerance) causes minimization of the `FCN` and returns the result in form of a FunctionMinimum. The optional argument `maxfcn` specifies the (approximate) maximum number of function calls after which the calculation will be stopped even if it has not yet converged. The optional argument `tolerance` specifies the required tolerance on the function value at the minimum. The default `tolerance` value is 0.1, and the minimization will stop when the estimated vertical distance to the minimum (`EDM`) is less than 0.001∗`tolerance`∗`up` (see 3.1.2).

4.6.5 Parameter interaction

MnMigrad retains the result after each minimization (MnUserParameterStae, see 4.15) and forwards the interface.

4.6.6 VariableMetricMinimizer()

The VariableMetricMinimizer is instantiated using default constructor.

4.6.7 minimize(const FCNBase&, ...)

The VariableMetricMinimizer provides several overloaded methods minimize with return value FunctionMinimum. Together with the user `FCN` (either an implementation of FCNBase or FCNGradientBase) the user has to give as input the parameters with starting values in one of the defined formats (std::vector<double>, MnUserParameters or MnUserParameterState).

4.7 MnMinimize and CombinedMinimizer

Causes minimization of the function by the method of `MIGRAD`, as does the MnMigrad class, but switches to the `SIMPLEX` method if `MIGRAD` fails to converge. Constructor arguments, methods arguments and names of methods are the same as for MnMigrad or MnSimplex and VariableMetricMinimizer or SimplexMinimizer.

4.8 MnMinos

Causes a `MINOS` error analysis to be performed on the parameter whose number is specified. `MINOS` errors may be expensive to calculate, but are very reliable since they take account of non-linearities in the problem as well as parameter correlations, and are in general asymmetric. The optional argument `maxcalls` specifies the (approximate) maximum number of function calls **per parameter requested**, after which the calculation will be stopped for that parameter.

4.8.1 MnMinos(const FCNBase&, const FunctionMinimum&)

Construct an MnMinos object from the user's `FCN` and a valid FunctionMinimum. Additional constructors for user specific MnStrategy settings are provided.

4.8.2 operator()

MnMinos::operator()(unsigned int n, unsigned int maxcalls) causes a `MINOS` error analysis for external parameter `n`. The return value is a std::pair<double,double> with the lower and upper bounds of parameter `n`.

4.8.3 minos(unsigned int n, unsigned int maxcalls)

MnMinos::minos(unsigned int n, unsigned int maxcalls) causes a `MINOS` error analysis for external parameter `n` and returns a MinosError with the lower and upper bounds of parameter `n` and additional information in case that one bound could not be found. The result MinosError can be easily printed using std::cout.

4.8.4 Other methods

Additional methods exist to ask for one side of `MINOS` errors only.

4.9 MnPlot

MnPlot prints the result of `CONTOURS` or `SCAN` on a text terminal.

4.9.1 MnPlot()

The default constructor initializes default settings for the text window size.

4.9.2 operator()

MnPlot::operator()(const std::vector<std::pair<double,double> >&) prints a vector of (x,y) points on the text terminal. operator()(double xmin, double ymin, const std::vector<std::pair<double,double> >&) prints in addition the coordinates of the (x,y) values at the minimum.

4.10 MnScan and ScanMinimizer

MnScan scans the value of the user function by varying one parameter. It is sometimes useful for debugging the user function or finding a reasonable starting point. Constructor arguments, methods arguments and names of methods are the same as for MnMigrad and VariableMetricMinimizer.

4.10.1 scan(unsigned int par, unsigned int npoint, double low, double high)

Scans the value of the user function by varying parameter number `par`, leaving all other parameters fixed at the current value. If `par` is not specified, all variable parameters are scanned in sequence. The number of points `npoints` in the scan is 40 by default, and cannot exceed 100. The range of the scan is by default 2 standard deviations on each side of the current best value, but can be specified as from `low`

to high. After each scan, if a new minimum is found, the best parameter values are retained as start values for future scans or minimizations. The curve resulting from each scan can be plotted on the output terminal using MnPlot 4.9 in order to show the approximate behaviour of the function.

4.10.2 ScanMinimizer

Although the SCAN method is not intended for minimization it can be used as a minimizer in its most primitive form.

4.11 MnSimplex and SimplexMinimizer

SIMPLEX is a function minimization method using the simplex method of Nelder and Mead. MnSimplex provides minimization of the function by the method of SIMPLEX and the functionality for parameters interaction. It also retains the result from the last minimization in case the user may want to do subsequent minimization steps with parameter interactions in between the minimization requests. The minimization is done by the SimplexMinimizer. Minimization of the function can be done by directly using the SimplexMinimizer if no parameters interaction is required. As SIMPLEX is a stepping method it does not produce a covariance matrix.

4.11.1 MnSimplex(const FCNBase&, const std::vector<double>&, const std::vector<double>&, unsigned int)

Constructor for the minimal required interface: FCN and starting values for parameters and uncertainties. Optional the strategy level in MnStrategy can be specified.

4.11.2 MnSimplex(const FCNBase&, const MnUserParameters&, unsigned int)

Constructor for high level parameters interface. Optional the strategy level in MnStrategy can be specified.

4.11.3 MnSimplex(const FCNBase&, const MnUserParameterState&, const MnStrategy&)

Constructor from a full state (parameters + covariance) as starting input plus the desired strategy.

4.11.4 operator()

MnSimplex::operator()(unsigned int maxfcn, double tolerance) causes minimization of the `FCN` and returns the result in form of a FunctionMinimum. Minimization terminates either when the function has been called (approximately) `maxfcn` times, or when the estimated vertical distance to minimum (`EDM`) is less than `tolerance*up`. The default value of `tolerance` is `0.1`. (see 3.1.2).

4.11.5 Parameter interaction

MnSimplex retains the result after each minimization (MnUserParameterStae, see 4.15) and forwards the interface for parameter interaction.

4.11.6 SimplexMinimizer()

The SimplexMinimizer() is instantiated using default constructor.

4.11.7 minimize(const FCNBase&, ...)

The SimplexMinimizer provides several overloaded methods minimize with return value FunctionMinimum. Together with the user `FCN` (either an implementation of FCNBase or FCNGradientBase) the user has to give as input the parameters with starting values in one of the defined formats (std::vector<double>, MnUserParameters or MnUserParameterState).

4.12 MnStrategy

Sets the strategy to be used in calculating first and second derivatives and in certain minimization methods. In general, low values of `level` mean fewer function calls and high values mean more reliable minimization. Currently allowed values are 0 (low), 1 (default), and 2 (high).

4.12.1 MnStrategy()

Default constructor, sets all settings according to `level`= 1.

4.12.2 MnStrategy(unsigned int level)

Explicit constructor for predefined settings of desired `level` 0 (low), 1 (default), or 2 (high).

4.12.3 setLowStrategy(), setMediumStrategy(), setHighStrategy()

Methods to set specific strategy level.

4.12.4 Other methods

In addition, methods for individual settings such as setGradientNCycles() are provided.

4.13 MnUserCovariance

MnUserCovariance is the external covariance matrix designed for the interaction of the user. The result of the minimization (internal covariance matrix) is converted into the user representable format. It can also be used as input prior to the minimization. The size of the covariance matrix is according to the number of variable parameters (free and limited).

4.13.1 MnUserCovariance(const std::vector<double>&, unsigned int nrow)

Construct from data, positions of the elements in the array are arranged according to the packed storage format. The size of the array must be $nrow * (nrow + 1)/2$. The array must contain the upper triangular part of the symmetric matrix packed sequentially, column by column, so that arr(0) contains covar(0,0), arr(1) and arr(2) contain covar(0,1) and covar(1,1) respectively, and so on. The number of rows (colums) has to be specified.

4.13.2 MnUserCovariance(unsigned int nrow)

Specify the number of rows (colums) at instantiation. It will allocate an array of the length $nrow * (nrow + 1)/2$ and inititalize it to 0. Elements can then be set using the method operator()(unsigned int, unsigned int).

4.13.3 MnUserCovariance::operator()(unsigned int, unsigned int)

Individual elements can be accessed via the operator(), both for reading and writing.

4.14 MnUserParameters

MnUserParameters is the main class for user interaction with the parameters. It serves both as input to the minimization as well as output as the result of the minimization is

converted into the user representable format in order to allow for further interaction. Parameters for MINUIT can be added (defined) specifying a name, value and initial uncertainty.

4.14.1 add(...)

The method MnUserParameters::add(...) is overloaded for three kind of parameters:

- add(const char*, double, double) for adding a free variable parameter
- add(const char*, double, double, double, double) for adding a variable parameter with limits (lower and upper)
- add(const char*, double) for adding a constant parameter

When adding parameters, MINUIT assigns indices to each parameter which will be the same as in the std::vector<double> in the FCNBase::operator(). That means the first parameter the user adds gets index 0, the second index 1, and so on. When calculating the function value inside FCN, MINUIT will call FCNBase::operator() with the elements at their positions.

4.14.2 setValue(...)

setValue(unsigned int parno, double value) or setValue(const char* name, double value) set the value of parameter `parno` or with name `name` to `value`. The parameter in question may be variable, fixed, or constant, but must be defined.

4.14.3 setError(...)

setError(unsigned int parno, double error) or setError(const char* name, double error) set the error (sigma) of parameter `parno` or with name `name` to `value`.

4.14.4 fix(...)

fix(unsigned int parno) or fix(const char* name) fixes parameter `parno` or with name `name`.

4.14.5 release(...)

release(unsigned int parno) or release(const char* name) releases a previously fixed parameter `parno` or with name `name`.

4.14.6 setLimits(...)

setLimits(unsigned int n, double low, double up) or setLimits(const char* name, double low, double up) sets the lower and upper bound of parameter `n` or with name `name`.

However, if `low` is equal to `up`, an error condition results.

4.14.7 setUpperLimit(...)

setUpperLimit(unsigned int n, double up) or setUpperLimit(const char* name, double up) sets the upper bound of parameter `n` or with name `name`. The parameters does not have a lower limit.

4.14.8 setLowerLimit(...)

setLowerLimit(unsigned int n, double low) or setLowerLimit(const char* name, double low) sets the lower bound of parameter `n` or with name `name`. The parameters does not have an upper limit.

4.14.9 removeLimits(...)

removeLimits(unsigned int n) or removeLimits(const char* name) removes all possible limits on parameter `n` or with name `name`. The parameter can then vary in both directions without any bounds.

4.14.10 value(...)

value(unsigned int n) or value(const char* name) return the current value of parameter `n` or with name `name`.

4.14.11 error(...)

error(unsigned int n) or error(const char* name) return the current uncertainty (error) of parameter `n` or with name `name`.

4.14.12 index(...)

index(const char* name) returns the index (current position) of the parameter with name `name` in the list of defined parameters. The index is the same as for the calculation of the function value in the user's FCN (FCNBase::operator()).

4.14.13 name(...)

name(unsigned int n) returns the name of the parameter with index n.

4.15 MnUserParameterState

The class MnUserParameterState contains the MnUserParameters and the MnUserCovariance. It can be created on input by the user, or by MINUIT itself as user representable format of the result of the minimization.

4.15.1 MnUserParameterState(const std::vector<double>&, const std::vector<double>&)

Construct a state from starting values specified via std::vector<double>. No covariance is available.

4.15.2 MnUserParameterState(const MnUserParameters&)

Construct a state from starting values specified via MnUserParameters. No covariance is available.

4.15.3 MnUserParameterState(const MnUserParameters&, const MnUserCovariance&)

Construct a state from starting values specified via MnUserParameters and MnUserCovariance.

4.15.4 parameters()

The method parameters() returns a const reference to the MnUserParameters data member.

4.15.5 covariance()

The method covariance() returns a const reference to the MnUserCovariance data member.

4.15.6 globalCC()

The method globalCC() returns a const reference to the MnGlobalCorrelationCoeff data member.

4.15.7 MnUserParameterState::isValid() and MnUserParameterState::hasCovariance()

isValid() returns true if the the state is valid, false if not. hasCovariance returns true if the the state has a valid covariance, false otherwise.

4.15.8 MnUserParameterState::fval(), MnUserParameterState::edm(), MnUserParameterState::nfcn()

After minimization:

- fval() returns the function value at the minimum
- edm() returns the expected vertival distance to the minimum EDM
- nfcn() returns the number of function calls during the minimization

4.16 MnPrint.h

The following std::ostream operator<< output operators are defined in the file MnPrint.h.

4.16.1 operator<<(std::ostream&, const FunctionMinimum&)

Prints out the the values of the FunctionMinimum, internal parameters and external parameters (MnUserParameterState), the function value, the expected distance to the minimum and the number of fucntion calls.

4.16.2 operator<<(std::ostream&, const MnUserParameters&)

Prints out the MnUserParameters.

4.16.3 operator<<(std::ostream&, const MnUserCovariance&)

Prints out the MnUserCovariance.

4.16.4 operator<<(std::ostream&, const MnGlobalCorrelationCoeff&)

Prints out the MnGlobalCorrelationCoeff.

4.16.5 operator<<(std::ostream&, const MnUserParameterState&)

Prints out the whole MnUserParameterState: MnUserParameters, MnUserCovariance and MnGlobalCorrelationCoeff.

4.16.6 operator<<(std::ostream&, const MinosError&)

Prints out the MinosError of a given parameter.

4.16.7 operator<<(std::ostream&, const ContoursErros&)

Prints out the MinosError of the two parameters and plots a line printer graphic of the contours on the output terminal.

5 How to get the right answer from MINUIT

The goal of MINUIT — to be able to minimize and analyze parameter errors for all possible user functions with any number of variable parameters — is of course impossible to realise, even in principle, in a finite amount of time. In practice, some assumptions must be made about the behaviour of the function in order to avoid evaluating it at all possible points. In this chapter we give some hints on how the user can help MINUIT to make the right assumptions.

5.1 Which minimizer to use

One of the historically interesting advantages of MINUIT is that it was probably the first minimization program to offer the user a choice of several minimization algorithms. This could be taken as a reflection of the fact that none of the algorithms known at that time were good enough to be universal, so users were encouraged to find the one that worked best for them. Since then, algorithms have improved considerably, but MINUIT still offers several, mostly so that old users will not feel cheated, but also to help the occasional user who does manage to defeat the best algorithms. MINUIT currently offers four applications which can be used to find a smaller function value, in addition to `MINOS`, which will retain a smaller function value if it stumbles on one unexpectedly. The objects which can be used to minimize are:

5.1.1 `MIGRAD`

This is the best minimizer for nearly all functions. It is a variable-metric method with inexact line search, a stable metric updating scheme, and checks for positive-definiteness. It will run faster if you instantiate it with a low–level MnStrategy and will be more reliable if you instantiate it with a high–level MnStrategy (although the latter option may not help much). Its main weakness is that it depends heavily on knowledge of the first derivatives, and fails miserably if they are very inaccurate. If first derivatives are a problem, they can be calculated analytically inside `FCN` (see 3.1) or if this is not feasible, the user can try to improve the accuracy of MINUIT 's numerical approximation by adjusting values of MnMachinePrecision and/or MnStrategy (see 4.5 and 4.12).

5.1.2 `MINIMIZE`

This is equivalent to `MIGRAD`, except that if `MIGRAD` fails, it reverts to `SIMPLEX` and then calls `MIGRAD` again.

5.1.3 SCAN

This is not intended to minimize, and just scans the function, one parameter at a time. It does however retain the best value after each scan, so it does some sort of highly primitive minimization.

5.1.4 SIMPLEX

This genuine multidimensional minimization routine is usually much slower than MIGRAD, but it does not use first derivatives, so it should not be so sensitive to the precision of the FCN calculations, and is even rather robust with respect to gross fluctuations in the function value. However, it gives no reliable information about parameter errors, no information whatsoever about parameter correlations, and worst of all cannot be expected to converge accurately to the minimum in a finite time. Its estimate of the *expected distance to the minimum* EDM is largely fantasy, so it would not even know if it did converge.

5.2 Floating point precision

MINUIT figures out at execution time the machine precision 4.5, and assumes that FCN provides about the same precision. That means not just the length of the numbers used and returned by FCN, but the actual mathematical accuracy of the calculations. Section 2.7 describes what to do if this is not the case.

5.3 Parameter limits

Putting limits (absolute bounds) on the allowed values for a given parameter, causes MINUIT to make a non-linear transformation of its own internal parameter values to obtain the (external) parameter values passed to FCN. To understand the adverse effect of limits, see 1.3.1.

Basically, the use of limits should be avoided unless needed to keep the parameter inside a desired range.

If parameter limits are needed, in spite of the effects described in Chapter One, then the user should be aware of the following techniques to alleviate problems caused by limits.

5.3.1 Getting the Right Minimum with Limits

If MIGRAD converges normally to a point where no parameter is near one of its limits, then the existence of limits has probably not prevented MINUIT from finding the

right minimum. On the other hand, if one or more parameters is near its limit at the minimum, this may be because the true minimum is indeed at a limit, or it may be because the minimizer has become "blocked" at a limit. This may normally happen only if the parameter is so close to a limit (internal value at an odd multiple of $\pm\frac{\pi}{2}$ that MINUIT prints a warning to this effect when it prints the parameter values.

The minimizer can become blocked at a limit, because at a limit the derivative seen by the minimizer $\partial F/\partial P_{\rm int}$ is zero no matter what the real derivative $\partial F/\partial P_{\rm ext}$ is.

$$\frac{\partial F}{\partial P_{\rm int}} = \frac{\partial F}{\partial P_{\rm ext}}\frac{\partial P_{\rm ext}}{\partial P_{\rm int}} = \frac{\partial F}{\partial P_{\rm ext}} = 0$$

For a stepping method (like `SIMPLEX`) this seldom poses any problem, but a method based on derivatives (`MIGRAD`) may become blocked at such a value. If this happens, it may be necessary to move the value of the parameter in question a significant distance from the limit (e.g. with MnMigrad::setValue(...)) and restart the minimization, perhaps with that parameter fixed temporarily.

5.3.2 Getting the right parameter errors with limits

In the best case, where the minimum is far from any limits, MINUIT will correctly transform the error matrix, and the parameter errors it reports should be accurate and very close to those you would have got without limits. In other cases (which should be more common, since otherwise you wouldn't need limits), the very meaning of parameter errors becomes problematic. Mathematically, since the limit is an absolute constraint on the parameter, a parameter at its limit has no error, at least in one direction. The error matrix, which can assign only symmetric errors, then becomes essentially meaningless. On the other hand, the `MINOS` analysis is still meaningful, at least in principle, as long as `MIGRAD` (which is called internally by `MINOS`) does not get blocked at a limit. Unfortunately, the user has no control over this aspect of the `MINOS` calculation, although it is possible to get enough printout from the `MINOS` result to be able to determine whether the results are reliable or not.

5.4 Fixing and releasing parameters

When MINUIT needs to be guided to the "right" minimum, often the best way to do this is with the methods e.g. MnMigrad::fix(...) and MnMigrad::release(...). That is, suppose you have a problem with ten free parameters, and when you minimize with respect to all at once, MINUIT goes to an unphysical solution characterized by an unphysical or unwanted value of parameter number four. One way to avoid this is to fix parameter four at a "good" value (not necessarily the best, since you presumably don't know that yet), and minimize with respect to the others. Then release parameter four and minimize again. If the problem admits a "good" physical

solution, you will normally find it this way. If it doesn't work, you may see what is wrong by the following sequence (where `xxx` is the expected physical value for parameter four):

```
MnMigrad migrad(...);
migrad.setValue(4, xxx);
migrad.fix(4);
FunctionMinimum min = migrad();
migrad.release(4);
MnScan scan(...);
std::vector<std::pair<double, double> > points = scan(4);
```

where `SCAN` gives you a picture of `FCN` as a function of parameter four alone, the others being fixed at their current best values. If you suspect the difficulty is due to parameter five, then add

```
MnContours contour(...);
std::vector<std::pair<double, double> > points = contour(4, 5);
```

to see a two-dimensional picture.

5.5 Interpretation of parameter errors

There are two kinds of problems that can arise: The **reliability** of MINUIT 's error estimates, and their **statistical interpretation**, assuming they are accurate.

5.5.1 Statistical interpretation

For discussuion of basic concepts, such as the meaning of the elements of the error matrix, parabolic versus `MINOS` errors, the appropriate value for `up` (see 3.1.2), and setting of exact confidence levels, see (in order of increasing complexity and completeness):

- *"Interpretation of the Errors on Parameters",* see Part 3 of this write-up.
- *"Determining the Statistical Significance of Experimental Results"*[7].
- *"Statistical Methods in Experimental Physics"*[8].

5.5.2 The reliability of MINUIT error estimates

MINUIT always carries around its own current estimates of the parameter errors, which it will print out on request, no matter how accurate they are at any given

point in the execution. For example, at initialization, these estimates are just the starting step sizes as specified by the user. After a MIGRAD or HESSE step, the errors are usually quite accurate, unless there has been a problem. If no mitigating adjective is given in the printout of the errors, then at least MINUIT believes the errors are accurate, although there is always a small chance that MINUIT has been fooled. Some visible signs that MINUIT may have been fooled are:

- Warning messages produced during the minimization or error analysis.
- Failure to find new minimum.
- Value of EDM too big. For a "normal" minimization, after MIGRAD, the value of EDM is usually more than three orders of magnitude smaller than up, unless a looser tolerance has been specified.
- Correlation coefficients exactly equal to zero, unless some parameters are known to be uncorrelated with the others.
- Correlation coefficients very close to one (greater than 0.99).
  This indicates both an exceptionally difficult problem, and one which has been badly parametrized so that individual errors are not very meaningful because they are so highly correlated.
- Parameter at limit. This condition, signalled by a MINUIT warning message, may make both the function minimum and parameter errors unreliable. See section 5.3.2, *Getting the right parameter errors with limits*

The best way to be absolutely sure of the errors, is to use "independent" calculations and compare them, or compare the calculated errors with a picture of the function if possible. For example, if there is only one free parameter, SCAN allows the user to verify approximately the function curvature. Similarly, if there are only two free parameters, use CONTOURS. To verify a full error matrix, compare the results of MIGRAD with those (calculated afterward) by HESSE, which uses a different method. And of course the most reliable and most expensive technique, which must be used if asymmetric errors are required, is MINOS.

5.6 Convergence in MIGRAD, and positive–definiteness

MIGRAD uses its current estimate of the covariance matrix of the function to determine the current search direction, since this is the optimal strategy for quadratic functions and "physical" functions should be quadratic in the neighbourhood of the minimum at least. The search directions determined by MIGRAD are guaranteed to be downhill only if the covariance matrix is positive–definite, so in case this is not true, it makes a positive–definite approximation by adding an appropriate constant along the diagonal as determined by the eigenvalues of the matrix. Theoretically,

the covariance matrix for a "physical" function must be positive–definite at the minimum, although it may not be so for all points far away from the minimum, even for a well–determined physical problem. Therefore, if MIGRAD reports that it has found a non-positive–definite covariance matrix, this may be a sign of one or more of the following:

- **A non–physical region.** On its way to the minimum, MIGRAD may have traversed a region which has unphysical behaviour, which is of course not a serious problem as long as it recovers and leaves such a region.
- **An underdetermined problem.** If the matrix is not positive–definite even at the minimum, this may mean that the solution is not well–defined, for example that there are more unknowns than there are data points, or that the parametrization of the fit contains a linear dependence. If this is the case, then MINUIT (or any other program) cannot solve your problem uniquely, and the error matrix will necessarily be largely meaningless, so the user must remove the underdeterminedness by reformulating the parametrization. MINUIT cannot do this itself, but it can provide some hints (contours, global correlation coefficients, eigenvalues) which can help the clever user to find out what is wrong.
- **Numerical inaccuracies.** It is possible that the apparent lack of positive–definiteness is in fact only due to excessive roundoff errors in numerical calculations, either in FCN or in MINUIT . This is unlikely in general, but becomes more likely if the number of free parameters is very large, or if the parameters are badly scaled (not all of the same order of magnitude), and correlations are also large. In any case, whether the non–positive–definiteness is real or only numerical is largely irrelevant, since in both cases the error matrix will be unreliable and the minimum suspicious.

5.7 Additional trouble–shooting

When MINUIT just doesn't work, some of the more common causes are:

- **Precision mismatch.** Make sure your FCN uses internally the same precision as MINUIT .

  If the problem is only one of precision, and not of word length mismatch, an appropriate MnMachinePrecision::setPrecision() may fix it.
- **Trivial bugs in FCN.** The possibilities for C++ bugs are numerous. Probably the most common among physicists inexperienced in is the confusion between double and int types, which you can sometimes get away with, but not always. [3]

[3] For example, if a and b are double precision variables, the C++ statement a = 2*b is not good programming, but happens to do what the user probably intended, whereas the statement a = b + 2/3 almost certainly will not do what the user intended.

MINUIT can spot some trivial bugs itself, and issues a warning when it detects an unusual `FCN` behaviour. Such a warning should be taken seriously.

MINUIT also offers some tools (especially `SCAN`) which can help the user to find trivial bugs.

- **An ill–posed problem.** For questions of parameter dependence, see the discussion above on postive–definiteness. Other mathematical problems which can arise are: **excessive numerical roundoff** — be especially careful of exponential and factorial functions which get big very quickly and lose accuracy; **starting too far from the solution** — the function may have unphysical local minima, especially at infinity in some variables; **incorrect normalization** — in likelihood functions, the probability distributions must be normalized or at least have an integral which is independent of the values of the variable parameters.

- **A bug in MINUIT .** This is unlikely, but it happens. If a bug is suspected, and all other possible causes can be eliminated, please try to save a copy of the input and output files, listing of `FCN`, and other information that may be relevant, and send them to `fred.james@cern.ch`.

6 A complete example

Here a full example of a fit is presented, following the example DemoGaussSim.cpp.

6.1 The user's FCN

The implementation of FCNBase by the user's GaussFcn is shown here.

6.1.1 GaussFunction.h

The user's model function is a Gaussian.

```
#ifndef MN_GaussFunction_H_
#define MN_GaussFunction_H_

#include <math.h>

class GaussFunction {

public:

  GaussFunction(double mean, double sig, double constant) :
    theMean(mean), theSigma(sig), theConstant(constant) {}

  ~GaussFunction() {}

  double m() const {return theMean;}
  double s() const {return theSigma;}
  double c() const {return theConstant;}

  double operator()(double x) const {

    return
       c()*exp(-0.5*(x-m())*(x-m())/(s()*s()))/(sqrt(2.*M_PI)*s());
  }

private:

  double theMean;
  double theSigma;
  double theConstant;
};
#endif // MN_GaussFunction_H_
```

6.1.2 GaussFcn.h

The user's FCN (GaussFcn) to calculate the χ^2 (combining the user's data with the user's model).

```
#ifndef MN_GaussFcn_H_
#define MN_GaussFcn_H_

#include "Minuit/FCNBase.h"

#include <vector>

class GaussFcn : public FCNBase {

public:

  GaussFcn(const std::vector<double>& meas,
           const std::vector<double>& pos,
   const std::vector<double>& mvar) : theMeasurements(meas),
               thePositions(pos),
      theMVariances(mvar),
      theErrorDef(1.) {}

  ~GaussFcn() {}

  virtual double up() const {return theErrorDef;}
  virtual double operator()(const std::vector<double>&) const;

  std::vector<double> measurements() const {return theMeasurements;}
  std::vector<double> positions() const {return thePositions;}
  std::vector<double> variances() const {return theMVariances;}

  void setErrorDef(double def) {theErrorDef = def;}

private:


  std::vector<double> theMeasurements;
  std::vector<double> thePositions;
  std::vector<double> theMVariances;
  double theErrorDef;
};

#endif //MN_GaussFcn_H_
```

6.1.3 GaussFcn.cpp

The actual implementation of the FCNBase::operator() (called by Minuit):

```
#include "GaussFcn.h"
#include "GaussFunction.h"

#include <cassert>

double GaussFcn::operator()(const std::vector<double>& par) const {

  assert(par.size() == 3);
  GaussFunction gauss(par[0], par[1], par[2]);

  double chi2 = 0.;
  for(unsigned int n = 0; n < theMeasurements.size(); n++) {
    chi2 += ((gauss(thePositions[n]) - theMeasurements[n]) *
             (gauss(thePositions[n]) - theMeasurements[n]) /
             theMVariances[n]);
  }

  return chi2;
}
```

6.2 The user's main program

This is the main program DemoGaussSim.cpp:

```
#include "GaussFcn.h"
#include "GaussDataGen.h"
#include "Minuit/FunctionMinimum.h"
#include "Minuit/MnUserParameterState.h"
#include "Minuit/MinimumPrint.h"
#include "Minuit/MnMigrad.h"
#include "Minuit/MnMinos.h"
#include "Minuit/MnContours.h"
#include "Minuit/MnPlot.h"
```

```
#include <iostream>

int main() {

  // generate the data (100 data points)
  GaussDataGen gdg(100);

  std::vector<double> pos = gdg.positions();
  std::vector<double> meas = gdg.measurements();
  std::vector<double> var = gdg.variances();

  // create FCN function
  GaussFcn theFCN(meas, pos, var);

  // create initial starting values for parameters
  double x = 0.;
  double x2 = 0.;
  double norm = 0.;
  double dx = pos[1]-pos[0];
  double area = 0.;
  for(unsigned int i = 0; i < meas.size(); i++) {
    norm += meas[i];
    x += (meas[i]*pos[i]);
    x2 += (meas[i]*pos[i]*pos[i]);
    area += dx*meas[i];
  }
  double mean = x/norm;
  double rms2 = x2/norm - mean*mean;
  double rms = rms2 > 0. ? sqrt(rms2) : 1.;

  {
    // demonstrate minimal required interface for minimization
    // create Minuit parameters without names

    // starting values for parameters
    std::vector<double> init_par;
    init_par.push_back(mean);
    init_par.push_back(rms);
    init_par.push_back(area);

    // starting values for initial uncertainties
    std::vector<double> init_err;
    init_err.push_back(0.1);
```

```
  init_err.push_back(0.1);
  init_err.push_back(0.1);

  // create minimizer (default constructor)
  VariableMetricMinimizer theMinimizer;

  // minimize
  FunctionMinimum min =
       theMinimizer.minimize(theFCN, init_par, init_err);

  // output
  std::cout<<"minimum: "<<min<<std::endl;
}

{
  // demonstrate standard minimization using MIGRAD
  // create Minuit parameters with names
  MnUserParameters upar;
  upar.add("mean", mean, 0.1);
  upar.add("sigma", rms, 0.1);
  upar.add("area", area, 0.1);

  // create MIGRAD minimizer
  MnMigrad migrad(theFCN, upar);

  // minimize
  FunctionMinimum min = migrad();

  // output
  std::cout<<"minimum: "<<min<<std::endl;
}

{
  // demonstrate full interaction with parameters over subsequent
  // minimizations

  // create Minuit parameters with names
  MnUserParameters upar;
  upar.add("mean", mean, 0.1);
  upar.add("sigma", rms, 0.1);
  upar.add("area", area, 0.1);

  // access parameter by name to set limits...
  upar.setLimits("mean", mean-0.01, mean+0.01);
```

```
// ... or access parameter by index
upar.setLimits(1, rms-0.1, rms+0.1);

// create Migrad minimizer
MnMigrad migrad(theFCN, upar);

// fix a parameter...
migrad.fix("mean");

// ... and minimize
FunctionMinimum min = migrad();

// output
std::cout<<"minimum: "<<min<<std::endl;

// release a parameter...
migrad.release("mean");

// ... and fix another one
migrad.fix(1);

// and minimize again
FunctionMinimum min1 = migrad();

// output
std::cout<<"minimum1: "<<min1<<std::endl;

// release the parameter...
migrad.release(1);

// ... and minimize with all three parameters
// (still with limits!)
FunctionMinimum min2 = migrad();

// output
std::cout<<"minimum2: "<<min2<<std::endl;

// remove all limits on parameters...
migrad.removeLimits("mean");
migrad.removeLimits("sigma");

// ... and minimize again with all three parameters
// (now without limits!)
```

```
  FunctionMinimum min3 = migrad();

  // output
  std::cout<<"minimum3: "<<min3<<std::endl;
}

{
  // demonstrate MINOS error analysis

  // create Minuit parameters with names
  MnUserParameters upar;
  upar.add("mean", mean, 0.1);
  upar.add("sigma", rms, 0.1);
  upar.add("area", area, 0.1);

  // create Migrad minimizer
  MnMigrad migrad(theFCN, upar);

  // minimize
  FunctionMinimum min = migrad();

  // create MINOS error factory
  MnMinos minos(theFCN, min);

  {
    // 1-sigma MINOS errors
    std::pair<double,double> e0 = minos(0);
    std::pair<double,double> e1 = minos(1);
    std::pair<double,double> e2 = minos(2);

    // output
    std::cout<<"1-sigma minos errors: "<<std::endl;
    std::cout<<"par0: "
             <<min.userState().value("mean")<<" "
             <<e0.first<<" "<<e0.second<<std::endl;
    std::cout<<"par1: "
             <<min.userState().value(1)<<" "
             <<e1.first<<" "<<e1.second<<std::endl;
    std::cout<<"par2: "<<min.userState().value("area")
             <<" "<<e2.first<<" "
             <<e2.second<<std::endl;
  }

  {
```

```
    // 2-sigma MINOS errors
    theFCN.setErrorDef(4.);
    std::pair<double,double> e0 = minos(0);
    std::pair<double,double> e1 = minos(1);
    std::pair<double,double> e2 = minos(2);

    // output
    std::cout<<"2-sigma minos errors: "<<std::endl;
    std::cout<<"par0: "
             <<min.userState().value("mean")
             <<" "<<e0.first<<" "<<e0.second<<std::endl;
    std::cout<<"par1: "
             <<min.userState().value(1)
             <<" "<<e1.first<<" "<<e1.second<<std::endl;
    std::cout<<"par2: "
             <<min.userState().value("area")
             <<" "<<e2.first<<" "<<e2.second<<std::endl;
  }
}

{
  // demonstrate how to use the CONTOURs

  // create Minuit parameters with names
  MnUserParameters upar;
  upar.add("mean", mean, 0.1);
  upar.add("sigma", rms, 0.1);
  upar.add("area", area, 0.1);

  // create Migrad minimizer
  MnMigrad migrad(theFCN, upar);

  // minimize
  FunctionMinimum min = migrad();

  // create contours factory with FCN and minimum
  MnContours contours(theFCN, min);

  // 70% confidence level for 2 parameters contour
  // around the minimum
  theFCN.setErrorDef(2.41);
  std::vector<std::pair<double,double> > cont =
       contours(0, 1, 20);
```

```
    // 95% confidence level for 2 parameters contour
    theFCN.setErrorDef(5.99);
    std::vector<std::pair<double,double> > cont4 =
         contours(0, 1, 20);

    // plot the contours
    MnPlot plot;
    cont4.insert(cont4.end(), cont.begin(), cont.end());
    plot(min.userState().value("mean"),
         min.userState().value("sigma"),
 cont4);
  }

  return 0;
};
```

References

[1] CN/ASD Group. *MINUIT – Users Guide, nProgram Library D506*. CERN, 1993.

[2] F. James and M. Winkler. *http://www.cern.ch/minuit*. CERN, May 2004.

[3] Paul F. Kunz. *http://www.slac.stanford.edu/grp/ek/hippodraw*. SLAC, May 2004.

[4] R. Brun and F. Rademakers. *http://root.cern.ch*. CERN, May 2004.

[5] CERN. *http://www.cern.ch/seal*, May 2004.

[6] CERN. *http://www.cern.ch*, May 2004.

[7] F. James. Determining the statistical Significance of experimental Results. Technical Report DD/81/02 and CERN Report 81–03, CERN, 1981.

[8] W. T. Eadie, D. Drijard, F. James, M. Roos, and B. Sadoulet. *Statistical Methods in Experimental Physics*. North-Holland, 1971.