Gemini

A minimization and error analysis package in C++

Version 1.0

User's & Reference Guide

August 1999

Zbigniew Szkutnik

This document has been prepared with Release 5.5 of the Adobe FrameMaker® Technical Publishing System using the User's Guide template prepared by Mario Ruggier of the Information and Programming Techniques Group at CERN. Only widely available fonts have been used, with the principal ones being:

Running text:	Palatino 10.5 pt on 13.5 pt line spacing
Chapter numbers and titles:	AvantGarde DemiBold 36 and 24 pt
Section headings	AvantGarde DemiBold 20 pt
Subsection and subsubsection headings:	Helvetica Bold 12 and 10 pt
Captions:	Helvetica 9 pt
Listings:	Courier Bold 9 pt

Preface

Gemini is a GEneral MINImization and error analysis package implemented as a C++ class library. Minuit's functionality is provided in a 'Minuit-style' (even if, internally, another minimizer may actually do the work) and new functionality offered by NAG C minimizers is added. Gemini thus provides a unified C++ API both to standard Minuit and to NAG C family of minimizers. For the common subset of functionality, it is up to the user which minimization engine does the work: Minuit or NAG C. The user can easily switch between various minimizers without essential changes in the application code. The currently supported set of minimizers (Minuit and NAG C) can be extended without essential changes in the API.

The abstract base class GEmini defines an interface to the common functionality. The CMinuit class is derived from GEmini and provides a Minuit-based implementation of the GEmini functionality plus Minuit-specific extensions. Similarly, the NAGmin class is derived from GEmini as well and provides a NAG-based implementation of the GEmini functionality plus NAG-specific extensions.

If the user's minimization object is declared as being of type CMinuit, then Minuit will be used as the minimization engine. Correspondingly, if the user declares the object as being of type NAGmin, then NAG C minimizers will be used.

There is no single class which contains references both to Minuit and to NAG C. This is done on purpose so that orthodox Minuit or NAG C users are not forced to link the other library (only if the Gemini library is created as a static library).

Gemini finds a minimum of an objective function, possibly subject to general constraints, and performs an error analysis. The concept of errors is that of Minuit, so that it is the user's responsibility to properly scale the inversed Hessian or, equivalently, define the UP parameter, and to properly interpret the results. Both Hessian based errors and Minos errors are implemented. Correspondingly, two types of function contours (or confidence regions, in statistical problems) are available: elliptical and Minos ones. Minos error analysis is, however, possible only for unconstrained and bound constraint problems.

General constraints cover, in addition to the usual bound constraints, two additional types of constraints:

- linear constraints of the form *lb* <= *c'x* <= *ub*, where *c* is a column vector of coefficients, *x* is a column vector of objective function arguments, and *lb* and *ub* are lower and upper bounds

- general non-linear constraints of the form *lb* <= *h(x)* <= *up*, where *h()* stands for any function of the objective function arguments.

Problems with linear and/or non-linear constraints can only be solved be means of the NAGmin class. Internally, the NAG minimizer *nag_opt_nlp* is used for such problems. Although the objective function and the constraint functions are assumed to be smooth (at least twice continuously differentiable), the method of nag_opt_nlp will usually solve the problem, if there are only isolated discontinuities away from the solution.

The only currently used minimizers are: MIGRAD for CMinuit and *nag_opt_nlp* for NAGmin. Note, however, that the CMinuit-specific method *command()* allows for execution of any Minuit command, thus providing an indirect way of using other Minuit minimization methods, like SIMPLEX or MINIMIZE.

The objective function to be minimized is encapsulated in an OBJfun class. It can be used simply to capture the usual user's function (e.g. *OBJfun f(chi2f);*) or a user can derive his own function class from it, add all the functionality needed in his/her specific problem and override the virtual member function *objfun* with the actual objective function.

Contours are implemented as an abstract data type GEminiContour with overloaded assignment and addition operators. Its usage is described in detail in section 1.6.

Gemini internally maintains two set of parameters: the initial setup used as a starting point for the first minimization process and the result of the last minimization. If another minimization process is started after the previous one is completed and if no intermediate parameter redefinition has been done, then the result of the previous minimization is used as the starting point for the next one. Whenever a user (re)defines a parameter, both sets of parameters get updated. At any stage of the analysis, the original starting point can be set by calling the function *resetStartPoint()*. Thus, the same starting point can be re-used with another minimizer but the results of the current minimization overwrite the previous ones.

The covariance matrix is computed but not stored in its final form. Only the parameters' errors are stored. In NAGmin, the covariance matrix is re-computed from its Choleski factor and stored in a user's matrix, if explicitly requested by a call to *getCovMatrix()*, as described below. In CMinuit, the covariance matrix is retrieved from the Minuit's memory, if requested.

The application code must include *gemini.h*, the Gemini header file. NAG C header files are not needed when the application code is being compiled.

Some examples are provided with the installation kit. At CERN, they can be accessed at $LHCXXTOP/share/GEMINI/pro/examples.

Installation details are described in a HTML document accessible at CERN at $LHCXXTOP/share/GEMINI/pro/doc/geminilib.html.

The package has been tested on HP (CC and aCC), IBM (xlC), DEC (cxx), Sun (CC), Linux (egcs) and Windows NT (MS VC++5).

For details related to the minimization algorithms, the user is refered to the corresponding manuals for Minuit [1] and NAG C [2].

Contents

Chapter 1

Functions and objects

This chapter describes the functions and objects used in Gemini.

1.1 The objective function

The objective function is the function to be minimized. The user must supply a C/C++ function which computes the objective function value and, optionally, its gradient, for a given vector of function arguments. Computing as many gradient components as possible will generally improve both performance and reliability.

The objective function has the prototype

void (int n, double g[], double *objf, const double x[], int code)

where:

n : Input: the number of function arguments

g[n] : Output: the function must not change the elements of g if no gradient is computed. If the gradient is computed, its computed components should be filled in g, if on input code=2.

objf : Output: the value of the objective function at the current point given in x.

x[n] : Input: the point x at which the function value (and possibly gradient) is required.

code : Input: if code=0, only function value must be computed. If code=2, both the function value and available gradient components should be computed. (For Minuit, code=1 is allowed as well, in which case initialization tasks can be performed. Note, however, that the initialization tasks can better be encapsulated in the constructor of the user's function class derived from OBJfun.)

A simple example function which implements a second-degree polynomial of one variable may look as follows.

```
void fun(int n, double g[], double *f, const double x[], int code)
{ const double a = 1.0, b = 2.0, c = 1.0;

  // compute function value
  *f = a*x[0]*x[0] + b*x[0] + c;

  if( code == 2 ) {
     // compute gradient
     g[0] = 2*a*x[0] + b;
  }
}
```

The objective function may have any name, unless the user overrides the *objfun* member of the objective function object, as described in the next section.

Although the objective function is assumed to be smooth (at least twice continuously differentiable), the minimizers will usually solve the problem, if there are only isolated discontinuites away from the solution.

1.2 The objective function object

The objective function object encapsulates the objective function, a function calls counter and, possibly, some user-defined functionality, like initialization.

The objective function object OBJfun can simply capture a usual user function, say chi2f(...),

```
OBJfun f(chi2f);
```

or the user can derive his/her own class from OBJfun with overriden virtual member function *objfun* and, possibly, some problem-specific functionality added, e.g.

```
class alephRb: public OBJfun {
public:
    //  user added functionality (initialization e.t.c.)
    ...
    //  override virtual objective function from OBJfun
    void objfun(int n, double grad[], double *fval,
                const double x[], int code);
private:
...
};
```

A function object can either be assigned to a minimization object via its constructor or, alternatively, via a public method *setObjFun()*. At any time, only one OBJfun object can be assigned to the fitting object. Any new assignment with *setObjFun()* replaces the previous one. For CMinuit, any new function assignement forces Minuit's re-initialization, in order to avoid any interference with the previous analysis. As a side-effect, the minimization options (like, e.g. the output level) previously set get lost and have to be re-set.

Internally, a function calls counter is implemented. Two public methods give access to this counter, although there is normally no need to do anything with the counter.

long counter(void)

Return the current value of the counter.

void resetCounter(void)

Reset the counter to zero.

1.3 Minimization objects and their constructors

The minimization object is the main object which contains the complete problem definition. It also provides methods for assigning an objective function object, defining the objective function arguments and their admissible regions, setting minimization options, running a minimizer, obtaining the current status of the minimization process, obtaining results and error analysis.

If the minimization object is of type CMinuit, then Minuit is used as the minimization engine. Similarly, if it is of type NAGmin, then NAG C minimizers are used.

Current minimization object constructors are:

```
CMinuit()

CMinuit(int n, OBJfun *fptr)

CMinuit(const char *ptitle, int n, OBJfun *fptr)
```

and

```
NAGmin()

NAGmin(int n, OBJfun *fptr)

NAGmin(const char *ptitle, int n, OBJfun *fptr)
```

where:

n : number of arguments of the objective function

ptitle : optional problem title

fptr : pointer to the objective function object.

The length of the title is limited to 80 characters (as defined in *gemini.h*). If ptitle points to a longer string, only the first 80 characters are stored as the problem's title.

CMinuit() is equivalent to *CMinuit(0, NULL)* and *NAGmin()* is equivalent to *NAGmin(0, NULL)*. They create an empty minimization object. An objective function object can then be assigned to such an empty fitting object by executing *setObjFun()*, as described below.

At any time, only one OBJfun object can be assigned to the fitting object. Any new assignment with *setObjFun()* replaces the previous one. For CMinuit, any new function assignement forces Minuit's re-initialization, in order to avoid any interference with the previous analysis. As a side-effect, the minimization options (like, e.g. the output level) previously set get lost and have to be re-set.

1.4 Generic pointer to a minimization object

A generic pointer of the type (GEmini *) can point both to CMinuit and NAGmin type objects and can be used if minimization objects are dynamically allocated on the heap. This allows the user to select the minimizer at run time rather than at compilation time. For example:

```
...
// capture the objective function
OBJfun f(myObjectiveFunction);
GEmini *mcptr;
int nop = 4; // number of function argument
if( something )
    // use Minuit
    mcptr = new CMinuit(“Minuit test“,nop,&f);
else
    // use NAG C
    mcptr = new NAGmin(“NAG C test“,nop,&f);
...
mcptr->minimize();
mcptr->printResults();
delete mcptr;
...
```

1.5 General constraints functions

By default, the minimization problem is considered unconstrained, i.e. each argument can take any real value. The user can, however, restrict the objective function domain by imposing bound constraints (setting lower and/or upper bounds for any argument), linear constraints (setting lower and/or upper bounds for linear combinations of arguments) and general non-linear constraints (setting lower and/or upper bounds for the values of arbitrary functions of the arguments). In the latter case, for each non-linear constraint, the user must provide a C/C++ function which computes, for a given vector of arguments, the constraint function value and, optionally, its gradient. Computing as many gradient components as possible will generally improve both performance and reliability. The constraint can then actually be imposed with *setNlinConstraint()*.

A general constraint function has the prototype

void (int n, double g[], double* val, const double x[], int code)

where:

n : Input: the number of function arguments

g[n] : Output: the function must not change the elements of g if no gradient is computed. If the gradient is computed, its computed components should be filled in g, if on input code=2.

val : Output: the value of the constraint function at the current point given in x.

x[n] : Input: the point x at which the function value (and possibly gradient) is required.

code : Input: if code=0, only function value must be computed. If code=2, both the function value and available gradient components should be computed.

For example, if the objective function is a function of four arguments and if one wants to impose a constraint on the sum of squares of the last two arguments, the corresponding general constraint function might look as follows.

```
void cfun(int n, double g[], double *f, const double x[], int code)
{
  // compute constraint function value
  *f = x[2]*x[2] + x[3]*x[3];

  if( code == 2 ) {
     // compute gradient components
     g[0] = g[1] = 0;
     g[2] = 2*x[2];
     g[3] = 2*x[3];
  }
}
```

The constraint function can have any name. Although the constraint function is assumed to be smooth (at least twice continuously differentiable), the minimizers will usually solve the problem, if there are only isolated discontinuites away from the solution.

A pointer to a constraint function is typedef(ined) as NLCF and is passed as argument to the function *setNlinConstraint()*.

1.6 Contour objects

A contour is a set of points from the boundary of a (bounded) set in a two-dimensional subspace. In Gemini, it typically represents either an elliptical boundary of a Hessian-based confidence region for a selected pair of parameters, or a Minos contour, i.e. the curve on which the minimum of the objective function with respect to all the remaining parameters equals the current minimum plus the current value of the UP parameter, as set via the *setError()* function.

Contours are implemented in Gemini as a GEminiContour class. Along with the contour points, stored in counter-clock order, GEminiContour also stores the coordinates of the contour centre, the title assigned to the contour, and the point marks, which are used for contour plotting in the text mode. Two public members np and n contain, correspondingly, the number of data points to be filled and the number of data points actually stored. Thus, np will typically be set before the contour is filled, while n is set by the routine which actually fills the contour. The default value of np is 20 and the default point mark is '*'.

The following contour constructors are available:

```
GEminiContour()

GEminiContour(int nop)

GEminiContour(char c)

GEminiContour(int nop, char c)
```

where:

nop : the required number of points

c : the point mark

A suitable copy constructor is provided, so that contours can safely be passed by value.

The contour title is normally set by the contour filling routine. Once o contour is filled, its title can be redefined by the user through the *GEminiContour* public method

```
void setTitle(const char *str)
```

The title length is limited to 70. If str points to a longer string, it will be truncated.

Similarly, the contour centre is normally set by the contour filling routine to the current minimum point, but can also be set explicitly through the public method

```
void setOrigin(double xo, double yo)
```

The contour can be plotted in the text mode for preliminary inspection through the public method

```
void plot(void)
```

The *x*- and *y*-coordinates of the points are stored in public arrays x and y and the coordinates of the *i*-th contour points can be accessed as x[i] and y[i] with i=0,...,n-1. The pointers to the double arrays of *x*'s and *y*'s are returned by public member functions *xp()* and *yp()*.

Whenever a declared contour is being filled, a public member n should be set to the actual number of points filled. In Gemini it is done automatically and n is equal to nop if the requested number of points has successfully been found.

Contours are implemented in Gemini as an abstract data type with overloaded assignment and addition operators. Addition means, in this case, merging and can be used for overlaying the contours. If the marks used in the two contours to be added are non-conflicting, they will be preserved. Otherwise, new marks will be created in such a way, that the points from the two added contours can still be distinguished, when the sum is plotted. For example:

```
GEminiContour c1, c2;
...
// fill the contours
...
// plot overlayed contours
(c1+c2).plot()
```

The title and center are inherited by the sum from the left-most term.

Chapter 2

Common public methods

Gemini's functionality is implemented in a set of non-virtual, virtual and pure virtual methods in the abstract base class GEmini. Some virtual methods are overriden in the derived classes CMinuit and NAGmin. Suitable versions are then called according to the actual object type. All pure virtual functions are implemented in the derived classes.

2.1 Non-virtual methods

int initParms(const char *file)

Read parameters setup from a text file. Return 0/1 in case od success/failure. Lines starting with '/' are ignored as comment lines. A line defining a parameter consists of blank-separated tokens and must contain the parameter number and name. Optionally, the initial value, lower bound, upper bound and step can be given. Consequtive values are assigned to consequtive parameters in the given order. Parameters, which are not explicitly defined, are treated as unbounded with initial value zero.

void printSetup() const

Print the current problem setup to standard output, including title, minimization method, initial parameters values, parameters bounds and steps.

int setLinConstraint(int cid, const double c[], double lb, double ub)

Add linear constraint with identifier cid, *nop*-vector of coefficients c, lower bound lb and upper bound ub. Return 0/1 if success/failure. The function fails, if the identifier is already in use. *nop* stands for the number of objective function arguments. Pre-defined constants MINF (minus infinity) and INF (infinity) can be used in case a bound is not imposed.

int setNlinConstraint(int cid, NLCF funptr, double lb, double ub)

Add non-linear constraint with identifier cid, constraint function pointer funptr, lower bound lb and upper bound ub. Return 0/1 if success/failure. The function fails, if the identifier is already in use. Pre-defined constants MINF (minus infinity) and INF (infinity) can be used in case a bound is not imposed.

int removeConstraint(int cid)

Remove constraint number cid. Return 0/1 if success/failure. The function fails, if there is no constraint with the identifier cid.

int getResult(int no, double& estimate, double& error, double& fval) const

If called before minimization or after unsuccessful minimization, the function returns 1. Otherwise 0 is returned and, for parameter number no, estimate is set to the optimal parameter value, error to the error estimate and fval to the minimum function value. If error was not successfully computed, error is set to -1.

int getMinosErrors(int no, double& negative, double& positive) const

Retrieve Minos errors for parameter no. Zeros are returned in negative/positive if the respective error has not been computed. The function fails if no is out of range or minimization has not been performed yet. Return values: 0/1 = success/failure.

int ellipticalContour(int n, int m, GEminiContour& p) const

Find p.np points along a contour on the plane spanned by parameters n and m. The contour is defined as the set of points in which the local parabolic approximation to the objective function (based on the Hessian) takes the value *min+UP*, where *min* is the current minimum, and *UP* is the value defined in *setError().* Thus, it just constructs standard confidence regions based on the marginal distributions of the estimates. Note that elliptical contours can only be constructed after both a minimum and the Hessian at the minimum have been found. Return values: 0/-1 = success/failure.

void printResults() const

Print the results to standard output, including title, minimization method, number of function calls, parameters, errors and function value at the minimum.

2.2 Virtual methods

virtual int parmDef(int no, const char *name, double value, double step, double lBound, double uBound)

Define parameter number no by setting its name, value, step, lower bound lBound and upper bound uBound and return 0/1 in case of success/failure. Pre-defined constants MINF (minus infinity) and INF (infinity) can be used in case a bound is not imposed. Setting value=lBound=uBound results in fixing the parameter at that common value. Parameters, which are not explicitly defined, are treated as unbounded with initial value zero. name can be set to NULL, in which case a default name is assigned to the parameter.

virtual void setObjFun(int nofparms, OBJfun *fptr))

Assign a new objective function object pointed to by fptr and set the number of parameters to nofparms. For NAGmin, the current parametrs' definitions for parameters 1 to nofparms and the current options setup are preserved. For CMinuit, however, any new function assignement forces Minuit's re-initialization, in order to avoid any interference with the previous analysis. As a side-effect, the previous parameters' definitions and the minimization options previously set get lost and have to be re-set.

virtual int resetStartPoint(void)

Reset start point to the original one. If a second call to *minimize()* is made without any intermediate action, the current solution is taken as the starting point. However, if this routine is called before the second *minimize()*, the original starting point will be used rather than the current solution. Note that in any case the current Hessian approximation gets forgotten , which may be useful if the Hessian is extremely ill-conditioned (c.f. NAG C documentation [2]).

virtual void setTitle(const char *title)

Set problem title.

virtual void setError(double up)

Define the error parameter (same as Minuit's UP value). The scale factor for the inverted Hessian is then 2*up, thus leading to usual one-sigma-errors, if up=1 for chi^2 or up=0.5 for negative log-likelihood. The default value for up is 1. It is the user's responsibility to correctly set this parameter and correctly interprete the errors obtained. Negative values of up are ignored. Whenever *setError()* is called, the parameters errors (if any) are recalculated and Minos errors get forgotten so that the current stored solution is always consistent with the current value of the up parameter.

virtual int fixParm(int no)

Fix the parameter number no at the current value obtained in the minimization process and return 0. If minimization has not been done yet, the function returns 1 with no change in the parameter setup.

virtual void warningsON(void)

Switch the warnings printout on (default).

virtual void warningsOFF(void)

Switch the warnings printout off.

virtual void setPrecision(double prec)

Precision corresponds to the Minuit's parameter *EPS* or Nag *options.f_prec*. It defines the precision to which the objective function can be computed. See corresponding manuals for details.

virtual void setTolerance(double tol)

Tolerance is Nag *options.optim_tol* and corresponds to Minuit's *tolerance* parameter. The algorithm stops if the estimated vertical distance to the minimum is smaller than tol. (NOTE: The last statement is precisely true for MIGRAD, but only approximately true for *nag_opt_nlp* where the convergence criteria are slightly more complicated. See [2] for more details.)

2.3 Pure virtual methods

virtual void setPrintLevel(int level)

Set printout level (level= -1,0,1,2,3).

virtual int minimize(void)

Solve the current minimization problem by means of the currently selected method. Return 0/1 in case of success/failure.

virtual int getCovMatrix(int n, double cov[]) const

Let *nofp* stand for the number of currently free parameters. If n <= *nofp*, the function fills the n*n cov matrix with the n by n portion of the covariance matrix. If n > *nofp*, the upper left corner of the cov matrix is filled with *nofp*nofp* elements of the covariance matrix and the remaining elements of cov are not modified. Thus, following Minuit's tradition, we drop here the rows and columns of zeros, which correspond to parameters which are either fixed or on the boundary. The function returns 0/1 in case of success/failure. cov is a n*n array stored row-wise. Note that the returned matrix is the genuine covariance matrix only if the error parameter up has been set correctly, i.e. such that the usual one-sigma-errors are computed (c.f. the description of *setError()* above).

virtual int getCov(int n, int m, double& cov) const

Set cov to the covariance of the parameters n and m. Return 0/1 in case of success/failure. In the NAGmin implementation, the covariance matrix is re-calculated from its Choleski factor whenever this function is called. Thus, if many covariances are to be obtained, it will be much more efficient to call *getCovMatrix()* and pick up suitable elements rather than to call *getCov()* many times. As above, cov is the genuine covariance only if the error parameter up has been set correctly, i.e. such that the usual one-sigma-errors are computed.

virtual int minosErrors(int n, double& neg, double& pos)

Find Minos errors for parameter n, store internally and return them in neg and pos. Return values: 0=success, 1=errors out of limits, 2=new minimum found, 3=no of function calls exceded its limit, 4=other failure.

virtual int minosContour(int n, int m, GEminiContour& p)

Try to find p.np points along a contour on the plane spanned by parameters n and m. The contour is defined as the set of points in which the minimum of the objective function w.r.t. all the remaining parameters equals *min+UP*, where *min* is the

current minimum, and *UP* is the value defined in *setError()*. p.n is set to the number of points actually found. Return values: number of points found if successful, 0=less than 4 points found, -1=other failure

Chapter 3

Minimizer-specific extensions

Some useful minimizer-specific features are implemented as extensions, so that they are not part of the GEmini class.

3.1 Minuit-specific extensions

int command(const char *cmd, const double args[], int argsno) const

Execute Minuit command cmd with argsno arguments given in the array args. Return 0/1 in case of success/failure. This corresponds to the Minuit's subroutine MNEXCM.

void commandMode(void) const

Switch to the command mode with commands read from standard input. This is an analog of the Minuit's subroutine MNINTR.

3.2 NAG-specific extensions

Optional parameters can be set via a file *e04ucc.opt*. The file is read and options are initialized when an object of the type NAGmin is created. One very useful option which can be set via *e04ucc.opt* is *verify_grad*, which has the effect that the minimizer verifies user-provided gradient components. This is very useful while debugging a newly written application. See section E04 in the NAG C library manual [2] for more details.

If the user objective function computes ALL the gradient elements, then setting *options.obj_deriv* to TRUE will improve both performance and reliability.

NAGmin solves problems with general constraints and performs the related error analysis. Note that Minuit is not able to solve such problems, even if general constraints can be set in the CMinuit class (*minimize()* will fail in that case).

Appendix A

Error analysis in Gemini

3.3 General concept of errors

The general concept of 'errors' or 'uncertainties' in Gemini is the same as in Minuit. For a given objective function $F(\theta)$ to be minimized, with $\theta = (\theta_1, \ldots, \theta_p)$, and for a given error parameter UP, the 'uncertainty set' US of the solution $\tilde{\theta} = (\tilde{\theta}_1, \ldots, \tilde{\theta}_p)$ is defined as

$$\text{US} = \{\theta : F(\theta) - F(\tilde{\theta}) \leq \text{UP}\} \tag{1}$$

For any sub-vector of $\tilde{\theta}$, the uncertainty set is constructed as the orthogonal projection of US onto the corresponding plane spanned by the selected components.

This purely geometrical concept is meaningful, in *qualitative* sense, for arbitrary objective functions. 'Errors' or 'uncertainties' are related to the shape of the objective function in a neighbourhood of the minimum.

Well defined *quantitative* meaning, in probabilistic terms, can be assigned to such defined 'errors' or 'uncertainties' in statistical problems, when the objective function is a fit criterion, for example a chi-squared, log-likelihood or least squares loss function.

Error analysis based on the plain difference $F(\theta) - F(\tilde{\theta})$ is called Minos analysis, as in Minuit. In this context, we also use terms like Minos error and Minos confidence region. Minos analysis can be computationally very costly, however, as it requires multiple function minimization to find points on the boundary of US or of its projection. It will be seen below, how Minos analysis can formally be justified in statistical terms. For maximum likelihood estimators and standard minimum chi-squared estimators, for example, it can be done via the asymptotic chi-squared distribution of a suitably transformed likelihood ratio.

A standard way to overcome the computational difficulty of Minos analysis is to approximate $F(\theta) - F(\tilde{\theta})$ with $0.5 \cdot (\theta - \tilde{\theta})^T H (\theta - \tilde{\theta})$, with H being the Hessian of F at $\tilde{\theta}$. One obtains this approximation via the standard Taylor expansion of F around $\tilde{\theta}$ and using the fact that the gradient of F at the minimum is zero. With this approximation, approximate versions of both US and its projections can be found analytically, so that multiple function minimization can be avoided. This leads to the standard Hessian-based error analysis and is related to asymptotically normal distributions of estimators.

In the following sections, those two approaches are described in more detail and their links to standard statistics exposed. Unconstrained minimization problems are discussed first. Error analysis for problems with constraints is the subject of the last section. It is always assumed that the problem is regular enough for the underlying mathematical theory to be applicable. The aim of this description is to expose main ideas rather than to present technical details. Relevant mathematical results can be found, for example, in the books by Silvey [4], Bard [5] and Serfling [3]. The book by Eadie at al. [6] is a standard statistical reference for HEP-physicists. It contains, in particular, a discussion of the Minos idea in less formal terms of an 'implicit transformation to linearity and back', which provides further insight into the idea of Minos.

3.4 Minos error analysis

The Minos uncertainty set US for the whole vector θ is defined above in (1). In order to obtain an uncertainty set for two components only, say θ_1 and θ_2, we have to project US onto the plane spanned by those components. This projection is a set of points (θ_1,θ_2) such that, for some $\theta_3,\ldots,\theta_p$, $F(\theta) \le F(\tilde{\theta}) + \mathrm{UP}$. Equivalently, it is the set of points (θ_1,θ_2) such that the minimum of $F(\theta)$ with respect to $\theta_3,\ldots,\theta_p$ and with θ_1 and θ_2 fixed is not greater than $F(\tilde{\theta}) + \mathrm{UP}$. The boundary of this set is thus the contour of the function

$$\tilde{F}(\theta_1, \theta_2) = \min_{\theta_3, \ldots, \theta_p} F(\theta)$$

which corresponds to $\tilde{F}(\theta_1, \theta_2) = F(\tilde{\theta}) + \mathrm{UP}$.

For a single parameter, say θ_1, we define a function

$$\hat{F}(\theta_1) = \min_{\theta_2, \ldots, \theta_p} F(\theta)$$

and construct the uncertainty set, or the projection of US, as $\{\theta_1 : \hat{F}(\theta_1) \le F(\tilde{\theta}) + \mathrm{UP}\}$. For a regular function *F*, genuine local minimum $\tilde{\theta}$ and 'small' UP, this will be an interval $[\underline{\theta}_1,\bar{\theta}_1]$, say. The positive and negative Minos errors are then defined as, correspondingly, $\bar{\theta}_1 - \tilde{\theta}_1$ and $\tilde{\theta}_1 - \underline{\theta}_1$.

In order to give Minos errors a quantitative, statistical meaning, let us assume first that *F* is -2*log-likelihood for a regular statistical model. The unrestricted minimum $\tilde{\theta}$ of *F* is then a maximum likelihood estimator of θ. Let further $\hat{\theta}$ be the minimum of *F*, subject to *r* independent restrictions $h_1(\theta) = h_2(\theta) = \ldots = h_r(\theta) = 0$. It is well known that, for any true θ which satisfies the restrictions, $F(\hat{\theta}) - F(\tilde{\theta})$ is asymptotically chi-squared distributed with *r* degrees of freedom - a fact used for the construction of the so-called asymptotic likelihood ratio test (λ-test).

It follows immediately that, for any true θ with the given values of the first two components, $\tilde{F}(\theta_1, \theta_2) - F(\tilde{\theta})$ is asymptotically chi-squared distributed with two degrees of freedom (we impose two constraints by fixing the values of θ_1 and θ_2) and, for any true θ with the given value of the first component, $\hat{F}(\theta_1) - F(\tilde{\theta})$ is asymptotically chi-squared distributed with one degree of freedom (we fix the value of θ_1 only).

A standard Neyman asymptotic $(1-\alpha)$-confidence region for (θ_1,θ_2) can then be constructed as

$$\{(\theta_1,\theta_2) : \tilde{F}(\theta_1, \theta_2) - F(\tilde{\theta}) \le c_\alpha\}$$

with c_α being the $(1-\alpha)$-quantile of the chi-squared distribution with two degrees of freedom. This is exactly the projection of US, with UP = c_α, onto the plane spanned by the first two components.

Similarly, an asymptotic Neyman $(1-\alpha)$-confidence region for θ_1 is

$$\{\theta_1 : \hat{F}(\theta_1) - F(\tilde{\theta}) \le c_\alpha\}$$

with c_α being the $(1-\alpha)$-quantile of the chi-squared distribution with one degree of freedom. Again, this is the projection of US with UP = c_α onto the first axis.

With obvious modifications, the same argument applies, of course, to any subset of the components of θ.

To summarize:

> If *F* is -2*log-likelihood, then Minos confidence regions for *r* components of θ have the asymptotic coverage probability $1-\alpha$, if UP is the $(1-\alpha)$-quantile of the chi-squared distribution with *r* degrees of freedom.

The scale factor of *F* is essential. Additive terms which do not depend on θ can be dropped, however.

With *r* = 1 and UP = 1, the coverage probability corresponds to that of a '± one-sigma error bar' for a single parameter.

In Gaussian models, -2*log-likelihood equals, up to a constant, additive term, the chi-squared fit criterion and the whole analysis applies. In many other cases, the equality holds asymptotically, thus validating the Minos analysis with *F* being the chi-squared fit criterion. For example, using the Taylor expansion of log's around n_i, the -2*log-likelihood for the standard Poisson model for histogram cells counts can be written as

$$-2\cdot\log L = -2\sum_i [n_i \log f_i(\theta) - f_i(\theta) - \log n_i!] =$$

$$-2\sum_i \left[n_i \left(\log n_i + \frac{f_i(\theta)-n_i}{n_i} - \frac{(f_i(\theta)-n_i)^2}{2\zeta_i^{\,2}} \right) - f_i(\theta) - \log n_i! \right] =$$

$$\sum_i \frac{(f_i(\theta)-n_i)^2}{n_i} \cdot \left(\frac{n_i}{\xi_i}\right)^2 + C$$

where *C* does not depend on θ and each ζ_i lies between n_i and $f_i(\theta)$. Strong law of large numbers implies that $n_i / f_i(\theta)$ tends to 1 with probability one, as the sample size increases ($f_i(\theta)$ also increases, in this case!). We thus have

$$(-2)\cdot\log L = C + (1+o(1))\sum_i \frac{(f_i(\theta)-n_i)^2}{n_i}$$

which shows that, indeed, the chi-squared criterion is asymtotically equivalent to -2*log-likelihood and that Minos analysis is asymptotically valid for the Poisson histogram cells counts model.

3.5 Hessian based error analysis

In the Hessian-based error analysis, $F(\theta) - F(\tilde{\theta})$ is approximated with $0.5 \cdot (\theta - \tilde{\theta})^T H(\theta - \tilde{\theta})$, with H being the Hessian of F at $\tilde{\theta}$. The approximate version US' of the uncertainty set US, corresponding to a given value of the UP parameter takes then the form

$$\text{US}' = \left\{\theta : \frac{1}{2}(\theta - \tilde{\theta})^T H(\theta - \tilde{\theta}) \le \text{UP}\right\} \tag{2}$$

The orthogonal projection US'_r of US' onto the plane spanned by, say, the first r components of θ consists of all points $(\theta_1, \ldots, \theta_r)$ such that the minimum of $(\theta - \tilde{\theta})^T H(\theta - \tilde{\theta})$ with respect to $\theta_{r+1}, \ldots, \theta_p$ is not greater than 2*UP. Let us split $\theta - \tilde{\theta}$ into two sub-vectors: θ_I consisting of the first r components and θ_{II} consisting of the remaining *p-r* components. Correspondingly, we can write

$$H = \begin{bmatrix} H_1 & H_{12} \\ H_{12}^T & H_2 \end{bmatrix}$$

with H_1 of size *(r,r)* and H_2 of size *(p-r,p-r)*. Looking for a minimum with respect to θ_{II} and with θ_I fixed, we have then

$$(\theta - \tilde{\theta})^T H(\theta - \tilde{\theta}) = \theta_I^T H_1 \theta_I + 2\theta_I^T H_{12}\theta_{II} + \theta_{II}^T H_2 \theta_{II} = G(\theta_{II})$$

and

$$\text{grad } G(\theta_{II}) = 2H_{12}^T\theta_I + 2H_2\theta_{II}$$

The equation $\text{grad } G(\tilde{\theta}_{II}) = 0$ gives the minimum point $\tilde{\theta}_{II} = -H_2^{-1}H_{12}^T\theta_I$. The minimum value is

$$G(\tilde{\theta}_{II}) = \theta_I^T(H_1 - H_{12}H_2^{-1}H_{12}^T)\theta_I$$

which gives

$$\text{US}'_r = \left\{\theta_I : \frac{1}{2}\theta_I^T(H_1 - H_{12}H_2^{-1}H_{12}^T)\theta_I \le \text{UP}\right\}$$

On the other hand, using the symmetric, block matrix inversion formula, we have

$$\begin{bmatrix} H_1 & H_{12} \\ H_{12}^T & H_2 \end{bmatrix}^{-1} = \begin{bmatrix} (H_1 - H_{12}H_2^{-1}H_{12}^T)^{-1} & X \\ X^T & H_2^{-1}(I - H_{12}^T X) \end{bmatrix} \tag{3}$$

with $X^T = -H_2^{-1}H_{12}^T(H_1 - H_{12}H_2^{-1}H_{12}^T)^{-1}$. This means that, with $S = (0.5H)^{-1}$, we can write

$$\text{US}' = \{\theta : (\theta - \tilde{\theta})^T S^{-1}(\theta - \tilde{\theta}) \le \text{UP}\}$$

and, denoting by S_r the upper left *(r,r)* portion of S, the projection US'_r takes the form

$$\text{US}'_r = \{\theta_I : \theta_I^T S_r^{-1}\theta_I \le \text{UP}\} \tag{4}$$

In order to set this in relation with statistics, recall that if *F* is -2*log-likelihood, then the maximum likelihood estimator $\tilde{\theta}$ is, in regular cases, asymptotically normally distributed

$$\tilde{\theta} \sim \mathrm{AN}(\theta, I_\theta^{-1})$$

where $I_\theta = 1/2 \cdot \mathrm{E}_\theta H$ is the Fisher information matrix for the whole data set. Again, in regular cases, one can reasonably assume that $\mathrm{E}_\theta H \approx H(\tilde{\theta})$ and use the inverse of $1/2 \cdot H(\tilde{\theta})$ as an estimate *S* of the covariance matrix of $\tilde{\theta}$.

This is clearly related to (2) and means that, since $(\theta - \tilde{\theta})^T S^{-1} (\theta - \tilde{\theta})$ is asymptotically chi-squared distributed with *p* degrees of freedom, in order to have the asymptotic $1 - \alpha$ coverage probability for US', one should set UP to the $(1 - \alpha)$-quantile of the chi-squared distribution with *p* degrees of freedom.

Further, S_r in (4) can clearly be interpreted as the covariance matrix of the *marginal* distribution of $(\theta_1, \ldots, \theta_r)$ and, in view of its asymptotic normality, setting UP in (4) to the $(1 - \alpha)$-quantile of the chi-squared distribution with *r* degrees of freedom, we get the asymptotic coverage probability $1 - \alpha$ for US'_r.

More generally, the above argument can be extended to any M-estimator, in which case the asymptotic covariance matrix needs not to be the inverse of the Fisher information matrix, but continues to be the (properly normalized) inversed Hessian.

For the chi-squared fit criterion, the approximation argument from the previous section applies.

3.6 Practical summary

The covariance matrix in Gemini is computed for unrestricted minimization problems as $S = 2 \cdot \mathrm{UP} \cdot H^{-1}$ and the Hessian-based error of $\tilde{\theta}_i$ is calculated as $\sqrt{s_{ii}}$. The following table summarizes the rules for proper setting of UP for single parameter's errors

	one-sigma-error	two-sigma-error	three-sigma-error
-log L	UP = 0.5	UP = 2	UP = 4.5
-2 log L	UP = 1	UP = 4	UP = 9
chi-squared	UP = 1	UP = 4	UP = 9

For a standard least squares fit, the values of UP should be taken from the last row and multiplied by the standard estimate $\hat{\sigma}^2$ of the residual variance. Since the variance estimate can only be computed after the fit, the value of UP can be set before minimization as for a chi-squared fit and the computed errors multiplied by $\hat{\sigma}$ afterwards.

For simultaneous $1-\alpha$ confidence regions for *r* parameters, UP should be set to the $(1-\alpha)$-quantile of the chi-squared distribution with *r* degrees of freedom for -2log L or chi-squared fit criteria and to half of that value for -log L. For a least squares fit, the values of UP should be computed as for the chi-squared fit and multiplied by the standard estimate $\hat{\sigma}^2$ of the residual variance.

3.7 Error analysis in constrained minimization

A special case of constraints are bound constraints. Both Minos and Hessian-based error analysis apply to problems with bound constraints. If a bound constraint is active, i.e. a variable is on its boundary, it is considered fixed at that value and, correspondingly, its 'error' is set to zero. For non-active constraints, Minos confidence regions are forced to be contained in the admissible regions and are cropped, if necessary. Hessian-based error analysis is not affected by non-active bound constraints.

For general constraints, the situation is different. Even if Minos analysis is potentially applicable to problems with general constraints, it is likely that the uncertainty sets are rather irregular or even degenerate, due to the restrictions imposed. In effect, it is rather difficult to construct a general algorithm able to handle those, rather arbitrary, sets. The Minos algorithm has been imported to Gemini from Minuit with only minor modifications. It has been designed and works effectively, however, only if the uncertainty sets are rather regular, especially if they are convex and non-degenerate, as it is typically the case in fitting applications with no general constraints imposed. Although it seems feasible to use the Minos algorithm for single-parameter's errors in general constrained problems, it has not been sufficiently tested yet and remains to be done. In Gemini, only Hessian-based error analysis, as described below, has been implemented for problems with general constraints.

The standard approach to constrained minimization is via the Lagrange functional. Only active constraints influence the error analysis. Assume that *F*=-log-likelihood. Let $h(\theta) = [h_1(\theta), \ldots, h_k(\theta)]^T$ denote the vector of constraints active at the solution and $\lambda = [\lambda_1, \ldots, \lambda_k]^T$ be the vector of Lagrange coefficients. The solution $\tilde{\theta}$ is then obtained by setting to zero the gradient of the Lagrangian

$$G(\theta, \lambda) = F(\theta) - \lambda^T h(\theta)$$

and solving the equation with respect to θ and λ.

Denote by A_θ the *(p,k)* matrix of gradients of $h(\theta)$ and write

$$\begin{bmatrix} I_\theta & A_\theta \\ A_\theta^T & 0 \end{bmatrix}^{-1} = \begin{bmatrix} V_\theta & Q \\ Q^T & V_\lambda \end{bmatrix}$$

where I_θ is the Fisher information matrix for the whole data set. It can be shown that, in regular models,

$$\tilde{\theta} \sim \mathrm{AN}(\theta, V_\theta)$$

and

$$\tilde{\lambda} \sim \mathrm{AN}(0, -V_\lambda)$$

and $\tilde{\theta}$ and $\tilde{\lambda}$ are asymptotically independent. More explicitly, one can show that

(5)
$$\tilde{\theta} \sim \mathrm{AN}(\theta, I_\theta^{-1} - I_\theta^{-1} A_\theta (A_\theta^T I_\theta^{-1} A_\theta)^{-1} A_\theta^T I_\theta^{-1})$$

$$\tilde{\lambda} \sim \mathrm{AN}(0, (A_\theta^T I_\theta^{-1} A_\theta)^{-1})$$

Recall that I_θ equals the expectation of the Hessian of *F*=-log-likelihood and that it is usually approximated by the Hessian at the minimum. In the process of a constrained minimization, however, the Hessian of the Lagrangian at the minimum is easily obtainable and not that of *F*. Those two Hessians differ by terms of the form

$$\sum_i \lambda_i \cdot \frac{\partial^2}{\partial\theta_j \partial\theta_l} h_i(\theta)$$

which, according to (5), may increase at the rate of the square-root of the sample size. Since the Hessian of *F* increases at the rate of the sample size, the difference between inverses of those two Hessians becomes asymptotically insignificant. As a result, with *H* denoting the Hessian of the Lagrangian at $\tilde{\theta}$, one can approximate the covariance matrix of $\tilde{\theta}$ as follows

$$V_\theta \cong H^{-1} - H^{-1} A_\theta (A_\theta^T H^{-1} A_\theta)^{-1} A_\theta^T H^{-1} \tag{6}$$

As a by-product of the minimization process, the NAG C minimizer *nag_opt_nlp*, used in Gemini to solve constrained problems, provides the upper triangular Choleski factor *R* such that $H = R^T R$. A convenient way of computing V_θ is then to solve the equation $R^T B = A_\theta$ with respect to *B* and compute

$$V_\theta = R^{-1}(I - B(B^T B)^{-1} B^T)(R^{-1})^T$$

or, alternatively, to find the QR decomposition $B = QU$ of *B*, form a matrix Q_1 consisting of the last *p-k* columns of *Q*, and compute

$$V_\theta = (R^{-1} Q_1)(R^{-1} Q_1)^T$$

The latter method is used in Gemini.

It is instructive to see how this formalism works for simple bound constraints. In this case, we can permute the parameters so that the last *k* of them are effectively fixed and

$$A_\theta = \begin{bmatrix} 0 \\ I \end{bmatrix}$$

Let us write

$$H = \begin{bmatrix} P & Q^T \\ Q & D \end{bmatrix} \qquad \text{and} \qquad H^{-1} = \begin{bmatrix} P_* & Q_*^T \\ Q_* & D_* \end{bmatrix}$$

with D and D_* of size *(k,k)*. Obviously

$$(A_\theta^T H^{-1} A_\theta)^{-1} = D_*^{-1}$$

and, using (6), one obtains after some elementary algebra

$$V_\theta = \begin{bmatrix} P_* - Q_*^T D_*^{-1} Q_* & 0 \\ 0 & 0 \end{bmatrix}$$

The block matrix inversion formula (3) applied to $(H^{-1})^{-1}$ then immediately gives

$$V_\theta = \begin{bmatrix} P^{-1} & 0 \\ 0 & 0 \end{bmatrix}$$

which means that the covariance matrix of the free parameters can be obtained by removing those rows and columns of the Hessian which correspond to the fixed parameters and inverting the resulting sub-Hessian.

Bibliography

1 **Minuit. Function Minimization and Error Analysis. Reference Manual.**
F.James; CERN Program Library Long Writeup D506, 1994.

2 **NAG C Library Manual, Mark 5, Vol. 2.**
The Numerical Algorithms Group Limited, 1998.

3 **Approximation Theorems of Mathematical Statistics.**
R.J.Serfling; Wiley, 1980.

4 **Statistical Inference.**
S.D.Silvey; Chapman and Hall, 1975.

5 **Nonlinear Parameter Estimation.**
Y.Bard;Academic Press, 1974.

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

Index

L

M

N

O

P

R

S

U

W