Minimizer User Examples Explained

Mark Fischler

Version 1.0, February 12, 2001

1 Skeleton of How to Use Minimizer

A `Minimizer` is an object, containing state and of course methods. The state is information about the minimization problem—where we are in trial $\vec{x}$ space, gradient information, error estimations for the current trial point, and so forth. The methods are various strategies for going from one trial point to another (assumedly superior) point.

The user program can be doing more than one minimization, that is, multiple `Minimizer` instances can co-exist and will not interfere with each other. This is in contrast to Fortran-based programs like Minuit, where the state information tendds to be kept in a COMMON area, which prevents two interleaved minimization efforts from proceeding.

The key steps in setting up a problem using a `Minimizer` are:

Create the Function Set up an a class which is derived from an interface class called `Fcn`. This class needs to have an `operator()` method which returns the value of the function to be minimized, and a method to tell how many parameters are involved. It can optionally contain methods to provide gradients and so forth.

Supply Constraints and Boundaries Some problems either look for a solution that must lie on some surface, or are of the nature that solutions outside the boundary of some are worthless. In such a case you can supply one or more functions of $\vec{x}$ which either have to be zero (constraints) or positive (boundaries) at the point of minimization.

Supply Parameter Information Although defaults are available, it is generally a good idea to supply starting values for $\vec{x}$. Also, for a parameter which must lie in a specific domain $[a, b]$, you may wish to provide a a function which internally distorts the variable such that it always lies in that domain.

Pick a Strategy A variety of minimization strategies are provided, including all those from Minuit, slightly improved versions of similar algorithms, and methods involving conjugate gradient and simulated annealing. There are also methods which combine these strategies, for robustness and rapid convergence.

Do Some Minimization Set up termination criteria (for example, a budget of some number of calls to the function) and invodke the selected strategy to move to some improved point in $\vec{x}$.

Inspect the Solution By looking at estimated errors, sensitivity to each parameter, and so forth, decide whether to take more steps with the same algortihm, coose a new strategy, or stop.

Do More Minimization You may decide to continue from the same state information, restart an algorithm keeping only the point in $\vec{x}$ space, or perhaps even change something about the function being minimized. (For example, when minimizing a chi-squared fit to data, it might make sense to use some small subset of points for the early steps, and then expand to the full set as the solution becomes more refined.) You may also decide to change strategies, if that seems appropriate.

Refine the Answer Finally, without moving off the last and best minimum point, get a description of the solution. This involves producing the best values possible for parameter errors and uncertainties, error correlation matrix, and estimated potential for further improvement.

1.1 A Bare-bones Example

The Minimizer is designed to allow the user considerable control over minimization strategies. This, of necessity, implies that there are quite a few options available. However, most of these choices are given well-chosen defaults, so generally, using the Minimizer should be quite simple. To demonstrate this fundamental simplicity, and ally fears that you need to grasp this entire document, here is an explanation of SimplestMinimizer.cc, a bare-bones example.

Add explanation here!

2 Creating the Fcn Object

The `Fcn` object is the way the user tells the minimizer what function to minimize. The user is providing a "functor", that is, an object which has an `operator()` method so it can be called as if it were a function. The advantage is that a functor is a full-fledge instance of a class: it can have its own state data, and it can be initialized in a constructor and cleaned up in a destructor.

The Minimizer defines an interface for this functor: the base class `Fcn`. The user-defined class must inherit from `Fcn`, and must provide two mandatory methods to override pure fvirtual methods in the base class. Other methods may optionally be provided, for faster or more accurate computation of such things as function gradients.

2.1 Required Methods

The minimizer absolutely needs to know two things abpout the function to be minimized: the value of the function at any given point $\vec{x}$, and the number of parameters taken by the function (thus, the dimension of the space it will be minizing over).

The value function is that `operator()` method:

```
double operator()(std::vector<double> &x) const;
```

The other mandatory method is:

```
int nParameters() const;
```

2.2 Optional Methods

gradient

Several minimization strategies are based on information about the gradient of the objective function, and/or the Hessian matrix of second derivatives. The minimizer has algorithms to compute these as needed (the algorithms used may differ from one strategy to another). However, the Fcn class may supply its own methods. The `gradient()` method returns a vector:

```
std::vector<double} gradient (std::vector<double> &x) const;
```

Typically, this would be an analytic calculation of the gradient; for a fitting minimization it would be sensible to sum the analytic form of the gradient of the chi-squared contribution over all the data points. It is not recommended that the user write his own purely numerical method to compute the gradient, since the Minimizer has such methods and they are taylored based on steps being taken and on what level of accuracy would be useful at this stage of computation.

If the work necessary to compute the gradient is very different from $2n$ function evaluations, then there can also be a method `gradientCost()` which returns a double specifying how many function evaluations a gradient evaluation should count for. This is relevant if the user sets a budget of no more than M function evaluations.

hessian

The hessian() method returns a `SymMatrix` which is a struct representing an N by N symmetric matrix. The header file for that struct shows how to work with this struct.

```
SymMatrix hessian (std::vector<double> &x) const;
```

Again, it probably doesn't pay to override this method to insert purely numerical difference methods.

If the work necessary to compute the hessian is very different from n^2 function evaluations, then there can also be a method `hessianCost()` which returns a double specifying how many function evaluations a hessian evaluation should count for.

fScale

The fscale() method returns a numerical scale associated with the uncertainty in the objective function. This is equivalent to the number in the "SET ERRordef" command of Minuit—it defines what is meant by one unit of uncertainty in the parameters. One unit of uncertainty in parameter x_i is that change in x_i away from its optimal value which will result in the objective function increasing by its fScale.

In the case where the user is minimizing a chi-squared sum which is normalized to the measurement errors, fScale should be 1, and the interpretaion of the uncertainty of a parameter is the variation which would lead to a 1 sigma change in chi-squared—this is the default. For a negative log likelihood function, the appropriate fScale wouldbe 0.5 instead.

(The Minimizer uses the term parameter "uncertainty" in place of the Minuit term parameter "error." The uncertainty is meaningful even if the Minimizer has done a perfect job of finding $\vec{x}$ such that the objective function is minimized; it takes note of the fact that the objective function is itself not sacrosanct. The term "error" is used to mean the likely distance between the purported optimal parameter value and the true optimal value, and that has nothing to do with an uncertainty scale in the objective function.)

3 Supplying Parameter Information

The Minimizer needs some information about the parameters over which it is to minimize.

(Actually, the only *essential* information is the number of parameters, and that is supplied by the Fcn function. All information described below defaults to values which might or might not make sense in any given application. Most uses of the Minimizer will, however, want to supply at least starting guesses.)

The way to tell the minimizer about a parameter is to instantiate and modify a Parameter struct. This contains information as described below. Having modified the struct as desired, the user then calls the Minimizer's `parameter()` member function to connect that Parameter information to the n-th parameter. This can be done for each parameter, before starting the first minimization strategy.

For example, a 3-parameter minization problem might do:

```
Minimizer mz;
Parameter a, b, c;
a.startingGuess = .5;
a.characteristicScale = .001;
// b is OK with the default starting guess of 0
c.startingGuess = -4.5;
mz.parameter(0,a);
mz.parameter(1,b);
mz.parameter(2,c);
```

Parameter Type

The enum MZR_ParameterType provides several choices; these are based on the parmater types avialable in Minuit:

MZR_FreelyVariable allowed to have any value. This is the default.

MZR_VariableWIthLimits allowed to vary, but the parameter fed to the Fcn is different from the internally-used parameter, with a mapping specified by a DomainEnforcer (see below).

MZR_Fixed Was once or may in the future be switched to variable, but now taking on only its current value, which will not be changed by a strategy step.

MZR_Constant

If a parameter is `MZR_Constant`, it is by convention meaningless, at the end of the minimzation, to ask about its error, uncertainty, or correlation with other parameters. However, one can ask such questions concerning an `MZR_Fixed` parameter. In particular, `errorMatrix` computation will include fixed parameters, varying them so as to find derivatives to estimate uncertainties.

Starting Guess

Initial value to set this parameter at. Obviously, a starting guess close to the actual global minimum will be very advantageous in terms of speed of convergence. Default starting guess is zero.

Error Estimate

Initial estimate of the distance to the true minimum. Unless this is set at a very low value, strategies tend not to be sensitive to this. However, if the initial error estimate supplied is very small, and the starting guess is close to some local minimum, then some strategies are likely to center on that minimum rather than the global minimum. If this is desirable, set the initial error estimates to something much less than your estimated size of the local well. Default error estimate is 1.

Characteristic Scale

The scale in this parameter on which "interesting" behavior occurs. This is only an advisory number; it is used to decide how small a variation to take when finding derivatives. (For example, if the Fcn looks like $\sin(x/1000000)$ you might supply a scale of 10^{-6}; if the best variation to use for a "typical" function of scale 1 were 10^{-5}, then for this function the appropriate variation would be 10^{-11}.

Since most strategies wich care about scale can re-estimate it at each step, based on second derivative computations and on how well the value at the next solution point was predicted, results tend not to be very sensitive to the supplied characteristic scale. Default scale value is 1.

DomainEnforcer

If a parameter is of type `MZR_VariableWIthLimits` then the actual ("internal") variable used by the minimizer will differ from the value of ("external") parameter passed to Fcn. The `DomainEnforcer` is an object which provides maps from internal value to external parameter and vice-versa, and the derivative of that mapping function. Typically, a `DomainEnforcer` would be given an upper and lower limit when it is constructed.

Unless the parameter type is `MZR_VariableWIthLimits`, the `DomainEnforcer` methods won't be invoked in the first place. Details of the `DomainEnforcer` object are discussed in the next section. The default `DomainEnforcer` is the trivial one, which maps internal values to the same external value and imposes no actual limitations on the domain.

Altering Parameters After Starting Minimization

It is permissible to alter the nature of a parameter after applying a strategy and then to continue with anther round of the same (or a different) strategy. This is discussed below in "Tinkering With Parameters." Suffice it to say that this will rarely be a good idea.

4 Constraints, Boundaries, and Restrictions

The naive minimization problem is free to vary all its parameters x_i over the entire number line, to find the point $\vec{x}$ at which $f(\vec{x})$ is minimal. Sometimes, this is to permissive—some regions of the x-space are not eligible to be considered when seraching for the minimum.

There are four ways in which regions of x-space can be forbidden.

Restriction is a multivariate extension of the concept of the range of a variable. It takes the form $R(\vec{x}) > 0$ where $R(\vec{x})$ is an arbitrary function. A problem can have any number of restrictions; the minimum found is required to satisfy all of them. The promise made by all strategies, with repect to restrictions, is that they will never attempt to call $f(\vec{x})$ at a point $\vec{x}$ where $R(\vec{x}) \leq 0$.

Boundary is a slightly freer form of restriction. It still takes the form $R(\vec{x}) \geq 0$, and the minimum found is required to satisfy all of the boundaries. However, a strategy is allowed to invoke $f(\vec{x})$ at a point outside the boundaries. In choosing between a restriction and a boundary, prefer to use a boundary unless either

- $f(\vec{x})$ is meaningless or will cause an exception if invoked in the forbidden region, or
- $f(\vec{x})$ outside the boudary is not an analytic continuation from the function inside.

Constraint is a way to express a problem of reduced effective dimension, by insisting that $R(\vec{x}) = 0$ for some arbitrary function $R(\vec{x})$. A problem can have any number of restrictions; the minimum found is required to satisfy all of them to the best precision achievable. In a sense, a constraint behaves more like a boundary which turns out to be relevant, than like a restriction: the strategy is permitted to try out points $\vec{x}$ where $R(\vec{x}) \neq 0$, but is not permitted to return such a point as an answer.

DomainEnforcer is a slight generalization of the Minuit capability of setting upper and lower limits for the external variable. The variable is given a range of $[a, b]$ and a pair of functions which map internal variables to external variables (which are inherently in $[a, b]$) and vice-versa. In Minuit these are functions involving arcsin and sin; here they can be specified as any monotonic functions with the proper range and domain properties.

When working problems with constraints, the minimization strategies will either add a "penalty function" involving distance from the constraint, or will use Lagrange multiplier techniques based on locally linear approximaitions to the constraints. In any event, each step of the path of solutions will lie on the constraint surface (or as nearly on it as machine accuracy permits), but probes to decide on the next step may evaluate the function at points where the constraint is not satisfied.

When working with a boundary, most strategies will ignore the boundary unless a step would take it outside the allowed region. In that case, that step will be re-done treating the boundary as a constraint, giving a next solution which saturaties the boundary condition.

A Restriction defines an open interval, and is intended for use where the expected minimum will not be at the edge of that restriction—it should be used to guard against trying function evaluations at "silly" points. A boundary, on the other had, will often be used in situations where the desird minimum may well saturate that bound. It is generally more efficient to use a boundary because it permits centered difference calculations from points near or on the boundary, rather than having to distort calculations to avoid ever straying into the forbidden zone.

4.1 DomainEnforcer

Sometimes the nature of a variable is such that it is not permitted to leave some domain. (We call it a domain, rather than a range, because the parameters are the independent variables on which the Fcn depends. The set of allowable values for independent variables defines the domain of a function.) In such cases, and especially if some portions of that domain should be looked at in finer detail than others because the Fcn is expected to be more sensitive to the parameter value, the parameter can be declared to be variable with limits, and a `DomainEnforcer` can be assigned.

The `DomainEnforcer` can implement a restriction to any connected closed or open portion of the real line. This can be a finite interval a to b or right or left the portion delimited by some point p, or even the entire real line. The way it implements the restriction is by providing three methods, mapping from internal to exteral parameters and vice-versa, and supplying the derivative of that map. (A fourth method, adjusting an internal parameter if it attains an inappropriate value, may also be supplied.) The user can create a `DomainEnforcer` by inheriting from that base class and overriding those four virtual functions. For example:

```
class MinuitDomainEnforcer : public DomainEnforcer {
public:
  MinuitDomainEnforcer (double lower, double upper)
    : a(lower), b(upper) { if (a>b) {b=lower; a=upper}  }
  double internalToExternal (double Pint)
    { return a + .5*(b-a)*(1+sin(Pint)); }
  double externalTointternal (double Pext)
    { return asin ( 2*(Pext-a)/(b-a) - 1); }
  double dExtdInt (double Pint)
    { return .5*(b-a)*cos(Pint); }
  double adjustPint (double Pint)
    { if ( Pint > piOver2 )
        return piOver2 - DISTNN;
      else if ( Pint <  piOver2 )
        return DISTNN - piOver2;
      else return Pint;
    }
private:
  double a,b;
}
```

This enforces the domain: It is impossible to choose a value of the internal parameter such that the external parameter is outside of $[a, b]$. And it obeys one further requirement: The mapping from external to internal parameter must be monotonic increasing.

The Minimizer package supplies the following pre-made `DomainEnforcer`s. The interfaces to these differ only in their constructors.

RightHalfDomainEnforcer

LeftHalfDomainEnforcer

ArctanDomainEnforcer

MinuitDomainEnforcer The class described above.

There are three limitations on the usefulness of the `DomainEnforcer` concept. Firstly, as discussed in the Minuit Reference Manual, the effects of the necessarily maps and of numerical roundoff implies can have adverse impact on the accuracy of the calculation. Secondly, depending on the details of the enforcer, the minimization may become distorted near either endpoint; if it is thought likely that the minimum saturates one end or the other for a variable, a Boundary should probably be used instead. Thirdly, the mechanism permits only conditions on one parameter at a time. For example, to impose the condition that $x + y < 10$ is trivial using a Boundary but impossible using a `DomainEnforcer`.

The best occasions to use a domain enforcer would be

- To mimic the algorithm used in a Minuit problem done with limit variables;
- When there exists a very natural map which is expected to mitigate some behavior or a parameter, so that the mapping done is desirable rather than a potential distortion;
- If some minimization strategy which you absolutely must use is incapable of respecting restrictions, or will be devastated in performance because of poor handling of the restrictions, then a domain enforcer may be the only alternative.

5 Controlling the Minimization Process

The preceding parts of this document have told (in excruciating detail) about how to set up and prepare do do a minimization. Finally, we get to the fun (and useful) part: How to apply MinStrategy's to the Minimizer, to doing the actual minimization.

A MinStrategy is itself an object, which can have internal state and so forth, and which has an operator apply method taking a Minimizer and a TerminationCriterion.

In general, there is a series of steps consisting of:

- Establishing (or changing) termination criteria for this step.
- Invoking a minimizaton strategy, which will run until either those termination criteria are met or the strategy decides it can't improve the solution further.
- Inspecting the solution, and either choosing a minimization strategy for the next step (which may well be a continuation of the prior one) or deciding you are finished.

First, one wet blanket on the "fun": For most fucntions, it is probably best to just use the strategy `MinimizeStrategy`, which combines various other strategies such as quardratic minimization and simplex to do an accurate and efficient job for most sorts of Fcn functions. And the termination criteria should be omitted or kept as simple as possible, probably just a budget on the number of Fcn calls; the strategy itself will decide to terminate when it senses diminishing returns.

So a typical use of the minimizer will look like:

```
Minimizer mz;
// ...
// Code as above to set up parameters and provided th Fcn to mz
// ...
FcnBudget myTerminator (100000);  // A TerminationCriterion
MinimizeStrategy m;
m(mz,myTerminator);
// Now look at the solution.
std::vector x = mz.values();
// ..
```

6 Setting Termination Criteria

Each time a strategy for minimization is applied, it needs some way of deciding when to terminate. In the Minimizer package this can be determined in one of two ways:

- Every strategy has a set of inate termination criteria. These are conservative: they express the point at which the strategy itself estimates that meaningful improvement is no longer possible, or that extreme diminishing returns have set in. A user can invoke a strategy with no termination argument, getting the best solution that strategy can deliver.
- In many or most cases, the user will wish to constrain the work spent on minimization, and/or request that the strategy return when some degree of accuracy has been attained. This can be accomplished by supplying a termination argument to the strategy.

The termination argument is to be of a type derived from TerminationCriterion. If it is supplied, it will augment (rather than supplant) the inate termination criteria. That is, a strategy will not continue to operate after it has decided it is no longer improving the solution, even if the user has provided it with a more generous termination criteria.

A TerminationCriterion may be built three ways:

1. The package comes with several built-in TerminationCriterion classes:

   NalgSteps (n) directs the strategy to stop if it has taken n of its algorithm's steps. stop if the Fcn function has been evaluated roughly n times.

   FcnBudget (n) directs the strategy to stop if the Fcn function has been evaluated roughly n times.

   Tolerance (x) directs the strategy to stop if the estimated differnce between Fcn and its minimum possible value drops below x times `fScale`. Minuit uses a combination of Tolerance and FcnBudget to terminate its strategies.

   RelativeAccuracy (α) directs the strategy to stop if the estimated errors on every parameter is less than α times the corresponding parameter uncertainty.

   ParameterErrors (ϵ) directs the strategy to stop if the estimated error on every parameter is less than ϵ times the corresponding characteristic scale.

   BangForBuck (x) directs the strategy to stop if the improvement in relative accuracy, per Fcn call, drops below some value depending on x.

2. The user can create a TerminationCriterion from boolean combinations of other criteria, e.g.,

```
TerminationCriterion term =
      FcnBudget (100000) || Tolerance (.00001) ||
     (RelativeAccuracy (.005) && BangForBuck (.05));
```

3. The user can create a custom termination condition. This would be a class, derived from TerminationCriterion, which has: /beginitemize /item A constructor that lets the user program set up its behavior. /item A boolean method terminate() returning true if the termination condition is satisfied. /item A boilerplate `clone()` method, which protects against code using a TerminationCondition which is no longer in scope. /enditemize The data available for deciding wheter to terminate is discussed below.

6.1 Built-in Termination Criteria

Each built-in termination criteria also provide a method to let a strategy know about their values in advance. (This implies, by the way, that the base class TerminationCriterion knows about its built-in descendants, becuase it has to provide the various virtual methods.)

So for example, the method `TerminationCriterion::fcnBudget()` returns either the value supplied to an FcnBudget, or -1 if the criterion does not involve a function call budget. (Sophisticated strategies can use this information to do some planning. For example, a methdod that can track down multiple local minima can sensibly decide how much effort to put into one before looking at another.)

If a TerminationCriterion is built up by combining multiple simpler criteria, it will when queried about these values provide the least aggressive of each type (the highest FcnBudget or number of steps; the lowest tolerance, releative accuracy, or BangForBuck) for an *and* combination, and the most agressive for an *or* combination.

Some details about each of the built-in termination criteria:

NalgSteps

This forces exactly that number of algorithm steps. The definition of a step is highly strategy dependant, but each step at least implies going to a potential new "best" solution. If the inate termination criteria are met, then the minimization will terminate wheter or not this number of steps were taken.

FcnBudget

This works precisely like the *maxcalls* argument found in Minuit: The strategies do not promise to exceed this number of calls, but will stop after a step if the call budget is exceeded.

Most strategies will try to foresee some minimum number of Fcn calls necessary for the next step, and will also stop if they know that they will overshoot and that they are closer now to the budget than they will be after the next step. (But at least one step will always be taken.)

If Fcn supplies its own gradient and/or hessian, along with cost functions for those, then those costs will be used in computing the function budget.

Tolerance

Gradient and Hessian information, along with predicted and actual function values at the current best solution, can be used to estimate how close from the actual minimum the corrent solution comes. It is sensible, especially in cases where the objective Fcn represents some sort of probability or log probability with a meaningful scale, to say that when you are within some tolerance of the actual minimum, the solution is "good enough."

The Tolerance crterion works precisely like .001 times the *tolerance* argument found in Minuit (because in Minuit, terminationj occurs when the estimated vertical distance to minimum is `.001*tolerance`). The value .0001 corresponds the the Minuit default tolerance.

RelativeAccuracy

This criterion is based on comparing the estimated error on each parameter to α times the corresponding parameter uncertainty. As mentioned earlier, the uncertainty in parameter x_i is defined as that change in x_i which will result in Fcn increasing by its fScale. Thus for data-fitting problems, α is a number of standard deviations.

Relative accuracy is somewhat related to Tolerance, and if you accept the default Minuit phlosphy of terminating when the Tolerance falls below a .0001 times `fScale`, then for typical minizations to fit a data set, the relative accuracy α should probably be set no smaller than $.0001/N$ where N is the the total number of parameters. This is because refinements of parameters past the level of improving chi-squared by $.0001\sigma$ is somewhat meaningless.

Actually, at any given step the parameter uncertainty is itself only estimated. To avoid premature termination, instead of comparing the error to the uncertainty, this criterion compares the error to the smallest plausible estimate of uncertainty for the parameter. For example, if $\alpha = .01$ and the uncertainty is estimated as $.5 \pm .2$, then the startegy would stop if the parameter error were less that $.01 \cdot (.5 - .2) = .003$.

BangForBuck

This criterion looks at the log of the relative accuracy, $\mathcal{L}$. The previous step of the stratgy will have used k calls to Fcn, to (hopefully) improve $\mathcal{L}$ by some amount δ. The criterion says to stop if $\delta/k < x/N^2$. (The N^2 factor is used because it is more costly to minimize a function across a larger number of parameters.) To avoid being triggered by a one-shot fluctuation, the criterion requires that this condition be met for three consecutive steps.

This is a fairly sophisticated criterion, allowing you to quit if further improvement of the solution becomes too costly. But the catch is deciding on a suitable value for x. A good way to decide this is to use BangForBuck as a way to give up early in an situation where you are setting an FcnBudget at B. If you set up $x = \frac{N^2}{2B}$, this would say that if the improvement becomes so small that (at this rate) B more calls would not improve accuracy by a factor of $\sqrt{e}$, then stop early.

6.2 Custom Termination Criteria

The terminate() method, which returns a bool, is where the decision logic goes. It is invoked after each "step" of a strategy; the meaning of a "step" is up to each minimization strategy. The available information for this decision logic includes:

- Number of Fcn calls (counting for gradients and hessains as well) since this strategy was started.
- Number of Fcn calls since the last step in the strategy.
- Estimated number of Fcn calls for the next step.
- Estimated distance to the veritcal minimum.
- Estimated errors in each parameter.
- Best estimate of parameter uncertainties.
- Number of parameters, and number of variable parameters.
- Values each parameters.

Also, some strategies make available other information. The calls to get this information will return values in any case, using some smeantically sensible approximation to the requested number. (This information can generally be computed more accurately by methods described in Information About the Minimum, but those make calls to Fcn and can therefore be more time-consuming than would be appropriate for inter-step queries).

In any event, the following information

- Gradients and Hessian.
- Best estimate of parameter correlation matrix.
- Eigenvalues (and eigenvectors) of some estimate to the correlation matrix.

The classes derived from TerminationCriterion can, of course, have state (which is why the terminate() method is non-const) and thus can base decisions on historical information.

7 Minimization Strategies

The Minimizer provides a variety of strategies, each of which inherits from the interface class MinStragegy. A strategy is defined as an algorithm for starting with some estimated solution to the minimization (which may include parameter values, and information such as previous step sizes, gradients, etc.) and taking one or more strategy steps to improve on that solution. An instance of a strategy may have internal state (beyond that state which is associated with the Minimizer) and may have other tuning parameters established by the user, either when the MinStrategy is constructed or at some later time.

Each MinStrategy has a key method, operator() taking two arguments: the instance of Minimizer (which holds the information about the Fcn to be minimized and the current solution state), and an optional TerminationCriterion. The method then affects the state of that Minimizer instance by improving the solution until either the user-specified termination the criteria or some strategy-specific inate criteria for stopping are met.

The user can choose which strategy to apply, and invoking the operator() of the appropriate MinStrategy object. With this choice comes a burden: The user must know something about the various strategies to be able to make a sensible selection. However, the non-expert user need not wade through the whole list of possible strategies. The `Minimize` strategy does as good a job in the general case as the package designers were able to implement, by trying various strategies, inspecting the solution, and selecting strategies for further refinement. The `Minimize` class requires no arguments in its constructor. Thus a very typical application that is OK with starting from a default guess of 0 for all parameters would simply do:

```
  Minimizer mz(f); //
  MinimizeStrategy m; // m is an instance of the MinimizeStrategy class
  m(mz); // apply m to mz, improving mz.
// Since no TerminationCriterion was supplied,
// the minimization will go till m is satistisfied
// that it can't get further meaningful improvement.
```

The `minimize()` method is an example of a strategy which calls other strategies, tying them together by some logic. A user program will do the same sort of strategy sequencing and decision logic; this may pay if some special knowledge of the Fcn function allows the user to guide the process better than the general strategies can do. If not, unless the user is quite expert in minimization technique (in which case, the package supporteres would love to get hold of and incorporate the new superior algorithm!) we recommend selecting among the available strategies, or perhaps combining one or two in a simple "find then refine" scheme.

8 Strategies

The Minimizer has a couple of strategies which could be recommended as a single-shot way to do the entire minimization. The most common one to use is `minimize()`, but for functions of specialized nature, one of the alternatives might make sense.

minimize()

minuitMINImize()

minimize()

The following strategies are provided as building blocks for the more complex strategies to use. User programs can avail themselves of these as well.

Finally, there are several strategies provided for historical reasons (e.g., they were present in Minuit) or for completeness. It is felt that the following strategies are "dominated" by other strategies listed above.

8.1 Stategy Tuning

THere will be methods to affect the details of strategy behavior (for example, speed of cooling in simulatedAnnealing.

The precise mechanism has not yoet been fleshed out.

8.2 Tinkering with Parameters

Between strategy calls, the user has the opportunity to fix or unfix parameters, or to modify parameter values. For example, a parameter can be flipped from fixed to freely variable. This might be appropriate if calling the Fcn with a changed value of one parameter would require a massive re-calculation, so that you want to do as few such changes as possible.

Other changes are also permitted. In particular, one can change the "Starting Guess" to force the strategy to move to a new point. However, most strategies will tend to lose confidence in prior computations (or to have unjustified confidence in them), and thus may work slowly at the start, if the values of parameters are changed out from under them. Unless you have an expert-level understanding of the strategy being employed, ad-hoc tinkering with parameters is likely to slow down the optimization process, and might lead to less reliable estimates of errors.

If the reason for directly altering parameter values and/or error estimates is to avoid the current local minimum altogether, the techniques discussed in the next subsection should be applied.

8.3 Tinkering with the Fcn Function

While tinkering with parameters in the solution is probably not a good idea, it is actually plausible that altering the Fcn function dynamically as the iminization porceeds could be useful. For example, in training a simulated neural net or some multiparameter cut conditions, you might want to apply only a small fraction of your entire training set for a while; and then when you might be close to an optimal solution, refine the answer using the whole available training set. You can do this if your Fcn has a method to alter some of its behavior. (You could even use the Minimizer::setFcn() method to start using a new function entirely.)

Depending on how profoundly you affect the function to be minimized, you may well wish to cold start the minimization strategy at that point. To do this, merely instantiate a new MinStrategy of the same type (which will be in the "I was just born, I don't know much" state).

It is desirable in such a situation to avoid huge jolts by normalizing the new function in the same way as the old, so that the value will be roughly the same. If the function value or scale has changed much, or if you have reason to believe that the previous error estimates might now be garbage, also invoke the coldStart() method on the Minimizer.

```
class MyF : public Fcn { Fcn(int npoints); setN(int npoints); ... }
MyF f(1000);
Minimizer mz(f);
MinimizeStrategy  m1;
m1(mz);
f.setN(5000);
MinimizeStrategy m2;
mz.coldStart();
m2(mz);
```

8.4 A Strategy to Search For Another Minimum

Sometimes a local minimum will not represent the best global minimum. In general, the problem of finding the true global minimum is very difficult. The Minimizer provides some tools that might help; some involve trying to find another minimum by ignoring the one currently found.

There are two potentially sensible ways to influence the Minimizer so as to eschew the current local solution:

- If you have reason to suppose tha some area near a particular other point will be superior you can tinker with the parameter values to set the solution to that other point. If necessary, you can make the error estimates artificially small; then at least for a while the strategy will tend to stay near the new point. With luck, it will find a distinct minimum, which may be better or worse than the original.

  Of course, if you knew what a good starting point would be, you could have provided that point in the first place! Perhaps more common would be the idea of moving the solution by some amount in a random direction—particularly if there is a physical reason to suspect that minima are separated by sopme characteristic length scale.

- Invoke the squelch() method. This establishes that "from now on" steps in any strategy will work for a while with a modified version of Fcn. The modified fucntion is distorted such that if Fcn were perfectly bi-linear, the function being minimized would be almost perfectly flat, with a small positive bulge at the existing solution point. The idea is that this forces the minimizatin away from the original point, yet if Fcn has another minimum, the distorted function will likely also have a minimum somewhere near that point.

After working with the new pseudo-Fcn solution for a while, step sizes may be small enough and the solution may be close enough to aother minimum, that the strategy to go back to the original Fcn and yet not stray back to the eschewed local minimum. (If you do return to the original, then it is more likely that that was indeed the only or the deepest minimum.)

You can in principle go on to squelch both a first and a second located minima, and so forth. However, it seems likely that squelching a large number of local minima at once may distort the function enough that when you return to the origninal Fcn, you still gravitate to one of the original local minima. At that point, squelch() will no longer be helpful. There is no magic bullet for the hard global minimization problem.

A simulatedAnnealing strategy is also available. This is not a way to move from a minimum you have already refined; instead, it randomly moves around treating the Fcn as a thermodynamic potential, and slowly (and repeatedly) cools to local minima. The tendency is for a deep minimum to win out more often in this process.

(However, a deep minimum with very little phase space is likely to lose out to a slightly shallower, yet much broader local minimum. The implications of this may be subtle—are you really trying to find the global minimum, even if slight fluctuaions in the paramenters about that point would greatly increase Fcn? In some cases, yes you are; in others the physics or statistics behind the problem would prefer the same minimum found more frequently by the annealing process!)

Some tout simulated annealing as the magic bullet for tough minimization problems but beware: It by nature is rather expensive in terms of number of Fcn calls.

9 Information About the Minimum

After a strategy has terminated, it is appropriate to inspect the solution it has found. In some cases, this will cause the user code (or a supervising strategy) to call the same strategy for further minimization or to call a different strategy.

At any event, after an acceptable minimum is reached, the user can request various pieces of information which may have been computed during the course of minimizing, or which may have to be computed at that time.

9.1 Nature of the Solution

Questions can be asked about the general nature of the solution found.

The method terminationReason supplies an enum describing the circumstances of termination and the success or failure of the minimization:

completionTermination — the strategy's inate termination criteria for a successful minimization have been met.

suppliedTermination — the minimizatin was terminated because the user-supplied termination criterion was met.

frustratedTermination — the strategy detected a situation in which this algorithm is not progressing or is not resolving errors as it would expect to. Tthe solution is probably not a good minimum, but this strategy cannot productively improve it.

runawayTermination — the strategy did not find a minimum because it appears to be working with a function which can go to negative infinity. For example, if you tried to minimize $f(x) = -x^2$, strategies would report this.

asymptoticTermination — the strategy did not find a minimum because it appears to be working with a function which has a horizontal asymptote with upward curvature and thus the solution keeps moving further in some direction. For example, if you tried to minimize $f(x) = 1/x^2$, strategies would report this.

You can also find out other general information:

boundarySolution — the solution apparently saturates one or more Boundaries or Restrictions. If this does not return ture the (in the opinion of the latest strategy) the minimum is in the interior region.

9.2 The Solution

Values

Error Estimates

(Correlation in Error Estimates?)

f value

9.3 Gaussian Statistics

Uncertainty

Correlation

9.4 Advanced Statistics

MINOS

9.5 Fcn Properties Near The Solution

Gradient

Hessian

Eigenvalues/vectors

9.6 Quantification of Work

Fcn calls

10 Other Operations

10.1 Saving and Restoring State

The Minimizer class has an operator¡¡ to ostreams, and an operator¿¿ from istreams. These output date in an ascii text format (which is even fairly human-readable). So it is possible to direct a Minimizer state to an output file (via an ofstream) and later read it in as an ifstream in another job.