The Zoom Minimization Package

M. Fischler *, D. Sachs, FNAL, Batavia, IL 60510, USA

Abstract

A new object-oriented Minimization package is available for distribution in the same manner as CLHEP. This package, designed for use in HEP applications, has all the capabilities of Minuit, but is a re-write from scratch, adhering to modern C++ design principles.

A primary goal of this package is extensibility in several directions, so that its capabilities can be kept fresh with as little maintenance effort as possible. This package is distinguished by the priority that was assigned to C++ design issues, and the focus on producing an extensible system that will resist becoming obsolete.

THE MINIMIZATION PACKAGE

The ZOOM C++ Minimization package is a re-design, from the ground up, implementing the algorithms and capabilities of Minuit. The design exploits the object-oriented aspects of C++; the principle benefit is much easier extensibility of these capabilities.

The package can be obtained at

`http://cepa.fnal.gov/aps/minimization.shtml`

and depends only on the C++ Standard Library. It contains a library of classes, suitable for uses as components, and easy to exploit in applications under other frameworks, such as Root. An obvious application is as the central part of a fitter.

The ground rules and assumptions in designing the package were expounded in [1]:

- The Minimization package must contain all the functionality of Minuit.
- The processing time is assumed to be dominated by the time taken by function evaluations.

The second assumption dictates when to trade speed of execution for better design cleanliness or flexibility.

As originally stated, we wanted to retain ways to identically mimic all behaviors of Minuit. However, we have modified the principle a bit. In cases where the intent of the algorithm in Minuit is clear but a mis-coding in Minuit has caused straying from that attempt, if the actual Minuit code is found not to be an "accidental improvement," we provide the corrected behavior instead.

---

* mf@fnal.gov

Using the Package

Obtaining and installing the Zoom Minimization package follows the same model recently adopted for CLHEP [2]. No special build systems, nor applications beyond `make`, nor other packages will be required.

The sequence of steps to install the package is:

1. Download and unpack the tar file.
2. Create a build directory.
3. Run the `configure` script which comes with the package.
4. Issue the `make` command to build the libraries.
5. `make check` to run validation tests.
6. `make install` to finally place the headers and libraries in the selected places.

Applications will find the headers under the `Minimization` subdirectory of the specified include area, and the library `Minimization.a` in the specified libraries area – shared libraries are also built. This should be familiar to anybody who has used CLHEP 1.9 or newer.

A Sample Program

The user interface to the Minimization package is designed to be as natural as possible. In the simplest case, the user has some function of N variables to minimize:

```
double f(const vector<double> &x) {
  return x[0]*x[1]+ //... some function of
}                   // x[0] through x[4]
int main() {
  int Ndimensions = 5;
  Problem m(UserFunction(Ndimensions,f));
  m.minimize();
  cout << m.currentPoint();  }
```

An attractive alternative is to supply a functor (an instance of a class which has an operator() method, which behaves like a function). This conveys two advantages:

- The functor instance can keep local data. For example, a function used to fit data in a few columns of a big disk-resident N-tuple can read in and save the relevant columns once, and work from memory thereafter.
- The user can write other methods of that class to control the function object. This obviates the un-natural IFLAG and FUTIL mechanisms necessary in Minuit.

For example:

```
const int Ndimensions = 5;
class MyF : public Minimization::Function {
 vector<double> a, b, c;
public:
 MyF(string filename):Function(Ndimensions)
 {  } // extract a, b, c from Ntuple
 double operator()(const Point &x)     {
 return fittingError (a,b,c,x);        }
}
int main() {
 Problem m(MyF("myNtuple.dat"));
 m.minimize();
 cout << m.currentPoint();  }
```

EXPLOITING THE O-O DESIGN

An object-oriented re-write required understanding all the algorithms in Minuit, determining the natural concepts and interface for minimization, and designing a new implementation. This forced re-inspection was advantageous in several ways.

Some features which mesh well with the existing capabilities were easily added. For example, half-open ranges for parameters are supported. And a few coding flaws are uncovered; corrections to the MNSIMP simplex method are discussed below.

Once the re-design has been done, all the usual advantages touted for object-oriented code apply. A particular improvement is that since well-designed C++ code avoids global variables, it becomes possible to run several minimizations independently. And by supplying an interface to minimize a functor, we have eliminated the need for users to interact with their target functions via global variables.

More significant advantages stem from the separation of concerns inherent in a well-designed object model. This makes it easier to make the features "orthogonal", so that the user has an easier time anticipating capabilities of the package. And it clarifies the coding of applications. For example, when dealing with termination conditions in a minimization problem, you need not simultaneously consider issues about the domain or the algorithm.

Directions of Extensibility

The most significant advantages of separation of concerns are highlighted when it is necessary to develop new functionality, or to inject the user's ideas into the capabilities of the package. This package is designed to make it easy for a user, an experiment support group, or a centralized tool provider to incorporate extensions to the capabilities.

Such extensions invariably involve adding new code, and it is sensible to say that part of the "interface" to an object-oriented package includes the specification of which capabilities are designed to be extended, and recipes for incorporating new code to produce these enhancements. The

Zoom Minimization package can be extended by adding subclasses of `Terminator`, `Domain`, and `Algorithm`.

The `Terminator` classes can represent criteria like "how small is my estimated distance to minimum?" and "how small is the largest remaining uncertainty in any of the parameters?". The package will provide several such classes, and allow for combining them in a natural way:

```
TerminationCriterion myTerminator =
  FunctionCallLimit(500)||EDMtolerance(.01);
```

It is also easy to code a custom Terminator; an example is presented below.

The `Domain` expresses the notion that not every combination of parameter values is permitted. The `Domain` corresponding to that available in Minuit is `RectilinearDomain`, in which each parameter is mapped into a fixed range. The range can be open at both, neither, or either end. Some plausible custom `Domains` would be:

- An orthogonal domain, but with constraints enforced by a different map than is used in Minuit, for example, by sigmoid mappings.
- Multiple probability space, where each variable must be non-negative, and their sum must be 1.
- Points on a Dalitz plot with some maximum energy.
- The interior or surface of an N-sphere.

For each of these cases, one would replace the methods defining maps, inverse maps, gradients and inverse gradients with versions that work for the new restrictions.

The `Algorithm` class allows for adding other algorithms, for example, that used in FUMILI [3]. Some thought is needed concerning how the new algorithm relates to overall concepts such as Estimated Distance to Minimum, so adding an algorithm requires more attention to detail and would typically be done by a central development effort. Still, the C++ issues arising when incorporating an algorithm into the Minimization package will be small compared to the mathematical effort needed to create and polish the new minimization scheme.

A Sample Custom Terminator

To illustrate how easy (or hard) it is to extend this package, we show how a new `Terminator` might be coded. To be concrete, let's say the user wishes to terminate when the improvement per function call in Estimated Distance to Minimum (EDM), averaged over the last N calls, falls below some value.

The business end of any subclass of `Terminator` is the required `finished()` method, which is called by the package at times when the algorithm has decided this might be a valid stopping point. Adding a sketch of the "boilerplate" code needed, this class looks like:

```
class ProgressRate: public Terminator {
  double target, BIG, lastEDM;
  int n, lastNf;
```

```
public:
ProgressRate(int interval, double rate):
    target(rate), BIG(1E20)m lastEDM(BIG),
    n(interval), lastNf(0) { }
void startMinimization(ProblemState & p) {
 lastNf=p.functionCallCount; lastEDM=BIG;}
 TerminationType finished
              (const ProblemState &p)    {
   if (p.functionCallCount >= lastNf+n){
    if (lastEDM - p.edm <
       target*(p.functionCallCount-n)
                          return TTstop;
    lastNf = p.functionCallCount;
    lastEDM = p.edm; }
   return(TTcontinue);                    }
}
```

IMPROVING SIMPLEX

The second most important minimization algorithm in Minuit is MNSIMP, implementing an N-dimensional simplex-based scheme. In the course of rewriting MNSIMP as the Simplex algorithm, we have repaired five errors in Minuit's MNSIMP code. Two flaws involved mis-chosen points for the starting simplex, requiring a few extra function evaluations to get started. We also correct the formula used for the minimum of a parabola, and eliminate a possible division by zero.

The last flaw is serious, and difficult to deal with. The simplex algorithm as implemented in Minuit is capable of incorrectly converging to a point which is not even a local minimum, and reporting successful convergence. The Simplex method in the Minimization package takes measures to make this case much rarer. This issue is discussed in detail below, but first, we motivate why it is worthwhile to improve the Simplex algorithm, rather than just to replicate the basic MNSIMP algorithm.

A good reason for providing a strong Simplex algorithm is that conjugate gradient techniques such as Migrad depend heavily on the existence of stable first and second derivatives. While many functions, in particular functions representing χ^2 error accumulations for an N-parameter fit, have stable derivatives, this is not a universal trait.

Imagine a fit where the sum to be minimized is a sum of squares of errors, but cut off at 3σ for each point. This might represent a situation where you decide to explain outlier points as bad readings, assigned a fixed unlikelihood. In such a fit, the function to be minimized will be continuous but non-analytic; it would appear "faceted" because as the fit parameters change, different points enter and leave the constant-probability region.

One might expect that Simplex would do a better job than Migrad on such functions. To verify this, we have tested both algorithms on a "faceted wine-glass" function: Start with an asymmetric wine glass function such as

$$w(x,y) = (x^2 + y^2)^2 - 2(x^2 + y^2) - 24x \qquad (1)$$

This has a minimum at (2,0).

Now impose a grid on the (x, y) plane, with spacing $2/k$. At each grid point, note the value of w and ∇w. Form our "faceted wine-glass" function $f(x, y)$ as a 2-D cubic spline, matching w at the grid points but with gradient $\alpha \nabla w$ at the grid points (for some small positive α). Viewed on a really large scale, $f(x, y)$ looks just like w; viewed on a scale of order $1/k$ it looks as if it takes steps and sharp turns; and ultimately, $f(x, y)$ is everywhere continuous and differentiable (and ∇f is also continuous). The function has one minimum; for integer k, the minimum remains at $(2, 0)$.

This sort of faceted function gives the Migrad algorithm fits. As long as the step size used in computing first and second derivatives is large compared to $2/k$, Migrad successfully approximates f by a quadratic, and moves to a much better point. But as the algorithm finds success, the step sizes decrease. When steps taken are of order $2/k$, the derivative calculations are somewhat random; the conjugate direction property of the algorithm goes away, and Migrad dances about as ineffectively as a naive gradient descent method.

The Simplex algorithm, on the other hand, doesn't much care about the graininess of $f(x, y)$, and converges almost as rapidly as it would for $w(x, y)$.

Comparing the performance of the two algorithms in minimizing $f(x, y)$ with $\alpha = .1$, starting from $(1000, 1000)$, we find that for $k = 10$, MIGRAD requires 295 function evaluations, while MNSIMP requires only 120. (These numbers are pretty stable as k varies.)

The moral here is not that Simplex is superior to Migrad; typically it is the other way around. The point is that having a choice of these tools is better than being restricted to either one alone. It is right to fix and polish Simplex, rather than to relegate it to some "this is flaky" status.

False Convergence

Consider a quite well-behaved function such as

$$\begin{aligned} f &= 10(w - 2v)^2 + 250(x - 3v + w)^2 \\ &+ 123(z - 2x)^2 + 17(z + y - 2)^2 \qquad (2) \\ &+ 100(w - 5)^2 + (w - 5)^4 \end{aligned}$$

This function has its minimum at $(2.5, 5.0, 2.5, -7.0, 5.0)$ and has no other local minima. Start each variable at 1000, and terminate when the EDM is 10^{-3}.

The behavior of the simplex algorithm is characterized by long periods (hundreds of steps) of general contraction of the simplex volume, and long periods of expansion. During the contraction periods (particularly toward the end, when the simplex is tiny), there is little movement of the location of the simplex and little improvement in the function value . But as the points draw very close together, the EDM (taken to be the variation in function values over the simplex) becomes small.

For the above function, the second episode of contraction shrinks the volume by a factor of 10^{19}, with the sim-

plex far from the true minimum. At this point, the EDM is 10^{-4}. Since the convergence criterion is an EDM of less than 10^{-3}, "convergence is detected" and the simplex algorithm bails out, convinced that it has found a good minimum!

If termination were not triggered, the simplex would then begin to expand, reflecting a "realization" that the simplex does not appear to straddle a minimum. The simplex would move to straddle the actual minimum, and a (final) period of contraction would quickly home in.

How can we understand the phenomenon of long-term contraction toward a false minimum?

Say the simplex has wandered into a region which is a gently sloping steep valley. That is, in some combinations of variables $\{\mu\}$, the function looks like sharp parabolas $f_0 + ax^2$, while in other orthogonal directions $\{\lambda\}$ it looks like a gently sloped line, with slopes less than s. The question of whether a proposed step improves the function value is dominated by whether the step improves the distance from minimum in the μ directions.

Improvement in the λ directions becomes moot, and the simplex wanders aimlessly in that space. The gentle slope is felt only when the chosen step lies within a small cone. This is the only time that progress will be made in the λ directions, and the simplex will tend to contract for most other steps. In a high number of dimensions, the cone represents a tiny fraction of the available solid angle, and the simplex will shrink faster than it will progress; the system is (temporarily) over-damped.

As the simplex shrinks, the slopes of the parabolas in the $\{\mu\}$ directions will decrease, until eventually the cone of improvement becomes wide. Once that happens, the simplex begins to frequently expand in the λ directions, the steep-valley nature of the region becomes moot, and the algorithm escapes the false minimum.

However, if the tolerance $\epsilon > s^2/a$, the onset of expansion may occur too late. The over-damped behavior can persist long enough to trigger the convergence criterion, and the algorithm can stop at a false minimum.

One indication that this false convergence situation is present is a long sequence of steps which don't improve on the best simplex point. The end stage of genuine convergence to a minimum is characterized by frequent improvements. A remedy, which is implemented in the current package, is that when the best point has remained stagnant for N iterations, the algorithm attempts to expand the simplex in the direction of the best point. This enhancement makes the over-damping situation outlined above much less likely, and cures most cases of false convergence.

Another approach will be to exploit a new, more meaningful notion of EDM in the Simplex algorithm, based on quadratic fitting of the last $\sim N^2$ points. If this EDM radically disagrees with the difference between the best and worst simplex point, this indicates overly rapid shrinking.

DIRECTIONS AND INTENTIONS

Although the Minimizer package is available for use, we are still completing it in several respects, including:

- Implement MINOS to fill the one remaining gap in full Minuit capability.
- Flesh out the collection of Terminator classes, and logical operators combining forming a Terminator from two others.
- Add a meaningful EDM calculation to Simplex.
- Provide an interface that supplies information and conveys control to allow GUI developers to easily interact with the Minimization classes.
- Include as sample applications unbinned binomial and binned χ^2 fitter classes.
- See how this Minimization package can be integrated into or coordinated with Root.
- Extend the set of Algorithms, to include FUMILI[3], bi-conjugate gradient[4], or other modern approaches.

Conclusions

The effort of designing a true object-oriented minimization package, and rewriting the algorithms of Minuit from scratch in C++, has yielded a package which is more easily extensible and is cleaner in concepts than a straight translation of Minuit to C++ would have been. In the course of this re-write, several flaws have been corrected, but none were found in the heavily-used MIGRAD algorithm.

The Zoom minimization package is still in the stage of evolution driven by its developers, and has had little contact as yet with the world of HEP users. But at this point, it is being made available to these users, and we will be coordinating with other HEP developers working on minimization codes to make the most of the design effort already there.

The hope is to move to the stage where extensions and contributions initiated by individual users add to the package.

ACKNOWLEDGMENTS

REFERENCES

[1] M. Fischler and D. Sachs, "An Object-Oriented Minimization Package for HEP", Proceedings of 2003 Conference in High Energy and Nuclear Physics (2003) MOLT004.

[2] L. Garren, "CLHEP Infrastructure Improvements", these proceedings.

[3] I. N. Silin, "FUMILI", CERN Program Library D510.

[4] A. Borici and P. de Forcrand, "Fast Krylov Space Methods for Calculation of Quark Propagator", hep-lat/9405001.

[5] F. James, "Minuit Reference Manual V94.1", CERN Program Library Long Writeup D506 (1994) p. 27.