FERMILAB-TM-2276-CD

About 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 in stand-alone applications. The object-oriented nature of the package makes it easy to exploit in applications under other frameworks, such as Root or the statistical package R. And the classes can be used as components in other utilities; the obvious application is as the central part of a fitter. (Some fitter classes, including binned χ^2 and unbinned maximum likelihood, will in fact be included in the Minimization package, both for convenience of users and as examples of how the minimization problem can be set up.)

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 needed to evaluate the function at various points.

The second assumption dictates when to trade speed of execution for better interface clarity, design cleanliness, or flexibility. Other than choices which will increase the number of function evaluations, the designers were free to ignore the question of whether each design choice might slow processing. Minuit, of course, makes the same assumption; one consequence is that these packages are not optimal for applications involving millions of minimizations of rapidly-evaluated functions.

The functionality ground rule gave us specific requirements to aim for. As originally stated, we wanted to retain ways to identically mimic all behaviors of Minuit. We were aware at the start of one necessary deviation from this principle: There can be no guarantee about the order of execution of statements, and there is always the possibility that code generated by different compilers (in this case, compilers for different languages) will, when executed in finite precision, yield non-identical results.

In the course of developing the package, however, we have modified the principle a bit further. In cases where

- the intent of the algorithm in Minuit is clear;
- a mis-coding in Minuit has caused straying from that attempt;
- and the actual Minuit code is found not to be an "accidental improvement" over the apparent intended code

we provide the corrected algorithm, and do not feel compelled to provide an option to perfectly replicate the Minuit behavior.

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`, are required. The intention is to require no other packages; at the time of writing this paper, Minimization depends on CLHEP, but that dependency is being cut.

The sequence of steps to install the package is:

1. Download and unpack the tar file obtained from the above URL.
2. Read the README file to see examples of installations.
3. Create a build directory.
4. Run the `configure` script which comes with the package. This can accept arguments specifying non-g++ installation, non-standard places for libraries and headers, and so forth.
5. Issue the `make` (or `gmake`) command to build the libraries.

* Expanded version of CHEP 04 paper

† mf@fnal.gov

6. `make check` to run validation tests.
7. `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, and other Zoom packages are also being put into this standard form.

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. A functor is an instance of a class which has an `operator()` method so it can be invoked as if the instance were itself a function. In our case, `operator()` takes as its argument a `Point`, which is works like a fixed-length vector of doubles. Using a functor 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 in filename
  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

Benefits of O-O Re-Write

The work involved in re-writing a major package using an object-oriented design approach can be considerable. We had to decide what the natural concepts and interface for minimization are, to implement this interface, and to read, understand, document, and re-implement the algorithms. But this forced re-inspection of methods has advantages. Sometimes, there are features which mesh well with the existing capabilities, and which a user might justly expect to find, but which were overlooked by the original authors. An example of this in Minuit is half-open ranges for parameters.

Sometimes, coding flaws are uncovered. In the MNSIMP simplex method, we have stumbled on three minor coding errors which can slightly increase the number of function calls required, one place where the original code is susceptible to division by zero, and an algorithmic vulnerability which can cause the method to converge to a fallacious minimum. While most these flaws are innocuous for most applications, repairing them does represent a slight improvement.

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, or one minimization within another. 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 provide orthogonality of features in the package, 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 and object-oriented design are highlighted when the capabilities of the package are not quite adequate "out of the box." Sometimes it is necessary to develop new functionality, or to inject the user's ideas into the capabilities of the package. The Zoom Minimization 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, unlike the case for a purely algorithmic package such as Minuit, 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 is built to accommodate extensions by the user community and by algorithm experts, in three directions. Extensions are added by creating subclasses of `Terminator`, `Domain`, and `Algorithm`.

The `Terminator` classes can represent criteria like "how small is my estimated distance to minimum?", "how small is the largest remaining uncertainty in any of the parameters?", and "how many times has the function been evaluated?". 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–in this package, the range can be open at both, neither, or either end. Some plausible custom `Domain`s would be:

- An orthogonal domain, but with constraints enforced by a different map than is used in Minuit, for example, by sigmoid mappings. This mapping is likely to be superior when and algorithm wants to take large steps which might shoot past a boundary, but could have slower convergence if the minimum is right on or very near a boundary.
- 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, the best way to write the new class would be to start from `RectilinearDomain`, substitute appropriate routines for the methods that let the user establish the limits, and replace the maps, inverse maps, gradients and inverse gradients with versions that work for the new restrictions. Adding a custom domain thus takes a bit more attention to detail (to change the mappings) than adding a new type of `Terminator`.

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 still more attention to detail and would typically be done by a central development effort rather than an individual user. 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, 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.

Here is a fairly complete sketch of the "boilerplate" code needed:

```
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) { }
ProgressRate* clone() const {
    return new ProgressRate(*this);}
void startMinimization(ProblemState & p) {
    restartMinimization(); }
void restartMinimization(ProblemState & p) {
    lastNf=p.functionCallCount;
    lastEDM=BIG; }
    // ... }
```

Of these methods, the only unfamiliar one might be the `clone()` method, and that takes the identical simple form for every `Terminator` subclass.

The "business end" of the class is equally straightforward in form:

```
class ProgressRate: public Terminator {
// ...
 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 THE SIMPLEX ALGORITHM

The second most important minimization algorithm in Minuit (behind MIGRAD) is MNSIMP, implementing an N-dimensional simplex-based scheme. (The simplex minimization algorithm we refer to here is not related to and

should not be confused with the simplex method for linear optimization problems.) In the course of rewriting MNSIMP as the Simplex algorithm, we have come across (and repaired in the Zoom Minimization package) five errors in Minuit's MNSIMP code.

1. When choosing points for an initial simplex, the algorithm tries to find decent points along each direction in turn, from the best point thus far. However, if none of the points tried in a direction prove superior, an arbitrary one is added to the simplex so as to move on the the next direction. In this case, it is slightly better–since the intent is to start at a collection of "good" points–to start the probe of the next direction at the first (best) of the points tried; MNSIMP instead uses the last inferior point.
2. Again at startup, if MNSIMP chooses a point in one direction by virtue of improving the function value for each of 6 increasing steps, and then in the next direction the first small step taken makes the value worse, MNSIMP erroneously selects that point immediately. This gives a very narrow simplex, and the algorithm must spend a few steps recovering from that before it begins to show serious progress.
3. MNSIMP had an incorrect formula for the minimum of the parabola passing through the old, reflected, and test points.
4. MNSIMP is vulnerable to a division by zero in the case where the function happens to be dead linear through the three trial points defining to a parabola.
5. The simplex algorithm as implemented in Minuit suffers from a fundamental difficulty: It 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.

Why Improve Simplex?

The lore about MNSIMP in Minuit is that it is a bit flaky – it is generally slower than MIGRAD, its Estimate Distance to Minimum is "largely fantasy" [5], and it sometimes fails to converge to a proper (even local) minimum. Although we were able to find several places where the code did not reflect the developers' intentions, and thus the MNSIMP implementation could easily be improved, it was unclear that fixing these minor flaws would turn the Simplex algorithm into a useful tool. At least one flaw (false convergence) proved to be inherent in the mathematical method, rather than a coding artifact, and would be difficult to rectify.

One can ask whether it is worthwhile, in a re-written package, to spend much time repairing these tough flaws. The alternative would be to replicate the basic MNSIMP algorithm (to meet our promise of all Minuit functionality), perhaps fixing some of the easy flaws, and to take the attitude that we need not waste effort polishing a tool which will be of little value anyway.

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, and in particular all functions representing χ^2 error accumulations for an N-parameter fit, do 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, and not let them unduly distort the fit parameters. In such a fit, the function to be minimized will be continuous and differentiable, but highly non-analytic; it would appear "faceted" with a facet scale dictated by the fact that 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 a familiar slightly 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. In each box on the grid, there is a unique function which is of no more than third degree in each of x and y, and whose values and gradient match at all four corners of the box the values of w and ∇w. The join of these functions is the (2-D) cubic spline approximation to w, and is continuous and differentiable everywhere. Finally, form our "faceted wine-glass" function $f(x,y)$ by matching w at the grid points but forcing the gradient at the grid points to be $\alpha\nabla w$ 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 in fact ∇f is also continuous). The function has only 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. Eventually the guess comes close enough to the actual minimum that the steps are small compared to $1/k$, at which point Migrad no longer is probing seriously non-analytic behavior, and will converge rapidly.

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

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

This function has its minimum at $(2.5, 5.0, 2.5, -7.0, 5.0)$ and in fact has no other local minima. Let's choose a starting guess where each variable is 1000, and terminate when the Estimated Distance to Minimum (EDM) is 10^{-3}.

After a brief initial phase of simplex expansion and rapid improvement in the function value, 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.

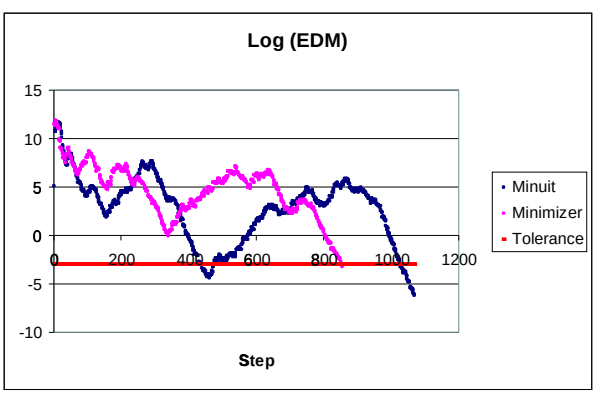

Figure 1: The EDM shrinks and expands, tripping the termination condition in MNSIMP.

During the contraction periods, there is little movement of the location of the simplex and little improvement in the function value (particularly toward the end, when the simplex is tiny). But as the points draw very close together, the EDM (taken to be the variation in function values over the simplex) becomes small. Almost all of the movement of the simplex occurs during expansion phases and, until the last contraction to the actual minimum, almost all of the improvement in function value occurs in steps when the simplex is large. This is illustrated in figures 1 and 2.

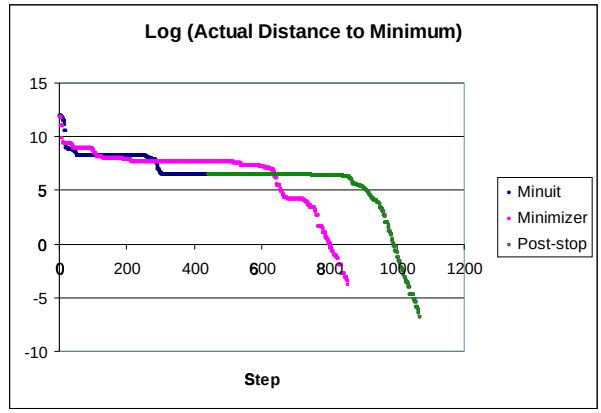

Figure 2: The function value does not plunge toward zero until the last contraction phase shown.

For the above function, the second episode of contraction occurs near the point $(-.27, -2.56, -812.5, -1769.3, -1643.0)$. 170 steps bring the volume down by a factor of 10^{19}, at which 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 (at least local) 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. By step 900, the volume is large, the simplex has moved to straddle the actual minimum, and a (final) period of contraction begins. The algorithm quickly homes in on the correct minimum.

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 a sharp positive parabola, while in other directions $\{\lambda\}$ it looks like a gently sloped line. Assume that the points are typically a distance of at least x away from the minimizing surface in the $\{\mu\}$ directions.

Suppose the $\{\mu\}$-parabolas look like $f_0 + ax^2$ and the steepest slope in the λ directions is s, and take $s < ax$. In those circumstances, the question of whether a given proposed step improves the function value is dominated by whether the step improves the distance from minimum in the μ directions.

The issue of whether a proposed step improves or worsens the position in the λ directions becomes moot, and the simplex wanders aimlessly with respect to those directions. 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.

Of course, as the simplex shrinks, the slopes of the parabolas in the $\{\mu\}$ directions will decrease, until eventually they become comparable to the gentle λ slopes. When $s > ax$ the cone of improvement becomes wide. Once that happens, the simplex begins to exploit the potential for improvement in the λ directions, which causes many steps to be of the expansion variety. The volume increases, 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.

Fixes For False Simplex Convergence

Any fix for this false convergence situation must, either explicitly or implicitly, somehow detect the presence of the phenomenon. The difficulty can be surmounted only to the extent that false convergence can reliably be detected, and that the steps taken in reaction are effective.

One indication that there is a problem is a long sequence of steps which leave the same simplex point as the best point. The end stage of genuine convergence to a minimum is characterized by frequent improvement in the function values, thus frequent changes in the best point. A possible 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 we have seen.

However, it is not guaranteed that false convergence cannot occur, even with this best-point-expansion strategy. Another characteristic of false convergence is collapse of one or more dimensions of the simplex. This can be diagnosed by small values for the ratio of simplex volume to the (hyper) surface areas of one or more of the faces of the simplex. We have observed these ratios becoming small as we approach false convergence.

Another way to detect a problem will be to exploit a new, more meaningful notion of EDM in the Simplex algorithm. If the EDM based on quadratic fitting of the last $\sim N^2$ points radically disagrees with the EDM based on the difference between the best and worst simplex point, this indicates overly rapid shrinking. The frequency of applying this sort of test would have to be tuned: Even though we are free to assume each function evaluation is expensive, and the quadratic fitting involves no extra evaluations, it is distasteful to add an order N^3 or worse quadratic fitting step to each step taken.

Having diagnosed over-damping, it is still difficult to see a good way to remedy the situation. We currently choose one tempting guess at a good direction for re-expansion; this cures most but not all cases of false convergence. There are always good directions available, but if the phase space of good directions for re-expansion were large, the original simplex algorithm would stumble onto them and the over-damping would not persist.

One obvious technique is restarting the algorithm at the purported minimum after apparent convergence. There is, of course, the issue of how to decide on the initial directions (the original coordinates might or might not be the best choice) and the scale of the new starting simplex. If you are in a gently sloping steep valley, there is a chance that barring a lucky choice of directions, the over-damping will still be present and you will again converge to a nearby false minimum.

It does look promising to combine the idea of a restart when false convergence is suspected, with the idea of fitting to a quadratic form based on the last $\sim N^2$ points. Eigenvector analysis easily decomposes the local function into the sum of N parabolas in linear combinations of the parameters. Some such combinations will, among the last $\sim N^2$ steps, have remained positive or negative – these represent the gentle slope directions. If the starting simplex is initialized based on these directions first, it should move rapidly away from the false convergence point. Clearly, there are opportunities for improvement in the Simplex family of algorithms.

DIRECTIONS AND INTENTIONS

Although the Minimizer package is available for use, we are still completing it in several respects. Aside from cleanups in internal code and documentation, these additions are in store:

- In terms of full Minuit capability, the one remaining gap is MINOS.
- We intend to flesh out the collection of Terminator classes, and to add a way of combining two Terminators to form a logical AND or OR of the conditions.
- We intend to add a non-"fantasy" EDM calculation to the Simplex method, and to continue to explore ways to avoid false convergence.
- Although GUI's for minimization are not part of the package, we are designing a set of methods to act as an interface, supplying information and conveying control that will allow GUI developers to easily interact with the Minimization classes.
- We have preliminary versions of a couple of fitter classes (unbinned binomial and binned χ^2) which will be included with the package as sample applications.
- We will be working with people to see how this Minimization package can be integrated into or coordinated with Root.
- We hope to extend the set of Algorithms, to include one or more of FUMILI[3], bi-conjugate gradient[4], or other modern approaches.

Other Minimizers

The situation with regards to commercially available minimization tools is, to our knowledge, about the same as

it was 18 months ago at the previous CHEP conference – no commercial minimizer exploits the capabilities of C++ to make it useful as a component in an application; most are algorithmically inferior to Minuit for HEP purposes, and certainly none are so superior as to overcome the HEP community's distaste for paying for a minor software component.

On the front of C++ Minuit clones, however, the situation has changed quite a bit in the past year and a half. Root has a fitter, based on a translation of Minuit. There are other fitters around, mostly based on Minuit or very similar code. And developers at CERN are well along in a project (initiated by Fred James and Matthias Winkler) to provide a C++ version of Minuit that sticks very close to the original interface and concepts.

Root developers have expressed interest in incorporating this Minimization package, and people developing the CERN C++ Minuit clone have expressed interest in coordinating work on these very similar projects.

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

The authors would like to thank Marc Paterno for giving the conference presentation based on this paper, and for key design input and reviews. We should also acknowledge that the basis for every algorithm presently in the package stems from the Minuit program, developed by Fred James at CERN.

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.