

# Optimizing the Performance and Structure of the D0 Collie Confidence Limit Evaluator

Mark Fischler\*

July 27, 2010

## Abstract

D0 Collie is a program used to perform limit calculations based on ensembles of pseudo-experiments (“PEs”). Since the application of this program to the crucial Higgs mass limit is quite CPU intensive, it has been deemed important to carefully review this program, with an eye toward identifying and implementing potential performance improvements. At the same time, we identify any coding errors or opportunities for potential structural (or algorithm) improvement discovered in the course of gaining sufficient understanding of the workings of Collie to sensibly explore for optimizations.

Based on a careful analysis of the program, a series of code changes with potential for improving performance has been identified. The implementation and evaluation of the most important parts of this series has been done, with gratifying speedup results.

The bottom line: We have identified and implemented changes leading to a factor of 2.19 speedup in the example program provided, and expected to translate to a factor of roughly 4 speedup in typical realistic usage.

---

\*Fermi National Accelerator Laboratory

## Contents

<b>1</b>	<b>Executive Summary</b>	<b>4</b>
1.1	This Technical Memo . . . . .	4
<b>2</b>	<b>Introduction and Overall Structure</b>	<b>5</b>
2.1	Review Ground Rules . . . . .	6
2.2	Test and Evaluation Programs . . . . .	7
<b>3</b>	<b>Time Profiling: Targets For Improvement</b>	<b>9</b>
3.1	What Does Collie Do? . . . . .	9
3.2	Calculating CL For a Given Signal Strength . . . . .	9
3.3	Where Does Collie Spend Its Time? . . . . .	11
3.4	Time Profile for the Initial Program . . . . .	12
<b>4</b>	<b>Most Promising Performance Improvement Suggestions</b>	<b>13</b>
4.1	Caching of Partial Likelihoods . . . . .	13
4.2	Factoring Out fval Logic For Each s Value . . . . .	17
4.3	Pre-computing s-dependent Part of Interpolating Function . . . . .	18
4.4	Low-efficiency Cut Folding . . . . .	18
4.5	Rectifying Inefficiency in Calculating Chi2 LLR . . . . .	19
4.6	Non-limit Minuit Variables . . . . .	19
4.7	Improved Fit Starting Points . . . . .	20
4.8	Partial Caching Extended to the 2-Change Case . . . . .	20
<b>5</b>	<b>Principal Other Performance Improvement Suggestions</b>	<b>21</b>
5.1	Data Structured For Rapid Expectation Calculation . . . . .	21
5.2	Recopying Parent Distributions . . . . .	22
5.3	Enabling the Analytic Derivatives . . . . .	22
5.4	Options for Tuning Fit Starting Points . . . . .	23
5.5	Combining Expectation Calculation With LLR Accumulation . . . . .	24
5.6	Ideas that Won't Help . . . . .	24
5.7	Controversial Changes . . . . .	25
<b>6</b>	<b>Effects of Implemented Suggestions</b>	<b>28</b>
6.1	Summary of Speedups . . . . .	28
6.2	Profiling this Code – General Points . . . . .	29
6.3	Profiling Results – Initial Program . . . . .	29
6.4	Profiling Results After First Optimizations . . . . .	30
6.5	Profiling Results After “Most Promising” Optimizations . . . . .	32
6.6	Non-performance Perturbations . . . . .	33
<b>7</b>	<b>Other Suggested Changes</b>	<b>34</b>
7.1	Erroneous Code . . . . .	34
7.2	Erroneous Interface with Root . . . . .	36
7.3	Erroneous Accumulation of Parent Expectations . . . . .	36

7.4	Memory Leaks . . . . .	37
7.5	Possibly Unintended Physics Behavior . . . . .	37
7.6	Poor Style Impacting Performance . . . . .	39
7.7	Bad Style That Could Be Corrected . . . . .	40
7.8	Clean-up Opportunities . . . . .	41
<b>8</b>	<b>Summary/Results of the Review</b>	<b>41</b>
8.1	Modified Code Provided . . . . .	41
8.2	Performance Differential Measurements . . . . .	42
8.3	Expectations in More Realistic Running . . . . .	43
8.4	Tricks That Did Not Help . . . . .	43
8.5	Further Opportunities For Tuning . . . . .	43
8.6	Issues the Developer Should Consider . . . . .	45
<b>A</b>	<b>Coding Steps</b>	<b>46</b>
A.1	Files Affected . . . . .	46
A.2	Setup for meaningful exploration of performance . . . . .	46
A.3	Partial Likelihood Caching . . . . .	48
A.4	Use of vectors in optimization . . . . .	52
A.5	Optimization of calculateChi2LLR . . . . .	52
A.6	Cleanup of Errors and Serious C++ Faux-Pas . . . . .	52
A.7	Moving inline Code Where Appropriate . . . . .	53
A.8	Minor cleanups . . . . .	54
<b>B</b>	<b>Math of Collie Calculations</b>	<b>54</b>
B.1	Computing Expectations Based on Systematics . . . . .	54
<b>C</b>	<b>Using the SimpleProfiler</b>	<b>55</b>
<b>D</b>	<b>Finding Signal Strength for a Given CL</b>	<b>57</b>
<b>E</b>	<b>Status of Investigations and Implemented Changes</b>	<b>58</b>

# 1 Executive Summary

The goal of this project was to analyze the D0 Collie program, specifically trying to make it faster. This project is delivering a tarball containing modifications to the code which result in a factor of 2.2 speed increase for the example program, with expectation that that factor will be larger when the problem has more bins and s-parameters. The project is also delivering this detailed document, explaining the reasoning and measurements leading to these optimizations, so that a future maintainer will not be in the dark about what the new code is doing.

Most of the speed increase comes about as a result of four changes:

- A rearrangement of loop ordering over bins and “systematic” parameters (s-parameters) allowed the most critical loop in the creation of model expectations to use a much smaller stride through memory being accessed (thus much less frequent cache misses).
- Analysis of the pattern of Minuit function evaluation requests allowed us to cache much of the work needed to evaluate the function, from one request to the next, greatly reducing the total work needed in all but a small fraction of cases.
- The final step of calculation of the chi-squared likelihood ratio was restructured such that a cut which had required a division and the computation of a logarithm for every bin now pre-computes an exponential and amortizes that cost over all the bins.
- The use of bounded variables in Minuit turns out to be unnecessary for this application, and was costing a noticeable amount for internal transformations; unbounded variables were substituted.

The tarball, `collieOptimization-v1.tar` was emailed to D0, for use in production running.

## 1.1 This Technical Memo

The concepts underlying Collie are much the same as the statistical techniques being proposed for confidence limits at CMS. The motivation for issuing this publication as a Fermilab Technical Memo is that other experiments (CMS in particular) are or will be relying on similar programs, and perhaps by reading about key optimizations, developers can write superior code.

Another motivation for making this document available is as an illustration of the kind of optimization which can be done due to the availability of the SimpleProfiler, developed by the CET group at Fermilab. Although an early version of the profiler was used in this work, the current version, while being less awkward to use, operates along the same lines.

This document’s primary purpose was in assisting D0 software people in understanding what was done to achieve the speedup (and attain confidence that the program remains correct). As such, it was confined to a D0 note. Since the Collie

program and details of its performance are the “property” of the D0 experiment, publication as a Fermilab Technical Memo was delayed until permission was given by a D0 software leader; thus the date of the TM is substantially later than the February 2009 date of the original document. Changes consist of this section, along with correction of minor typos.

## 2 Introduction and Overall Structure

D0 Collie is a program used to perform limit calculations based on ensembles of pseudo-experiments (“PEs”). The theory behind the way D0 performs these limit calculations is described in slides from the 2008 Prague D0 Workshop [1]. The ideas and operation of Collie are explained in some detail in the Collie Tutorial [2]; this note will not try to replicate that explanation. But for the purposes of others who may be reading the review without necessarily being expert in Collie usage and theory, and to provide a discussion framework about the steps the program performs, we explain the general idea as follows:

A SignalBkgdDist (sbd) consists of three distributions: One concrete, holding actual (or manufactured) data, and two conceptual, representing expected distributions if there is signal+background or background only. The object of the program is to take these distributions, and decide with what confidence limit we could reject the signal+background hypothesis. But the situation is made complicated by the presence of a number of systematic (s) parameters, whose values are not known *a priori*. The actual value of each systematic parameter can affect the value expected in each bin, in a roughly linear way. (The effect of each s-parameter may be different for the background-only and the signal+background case.)

In the Systematic Uncertainties presentation [1] it is explained that these parameters are not the quantities which can be obtained based on data outside the critical region and applied to the models; those sorts of systematics parameters are in fact incorporated into the models and do not add further complexity to the limit calculations. The s-parameters of concern to Collie come from poorly understood features of data measurement leading to “shape uncertainties” and from uncertain theoretical modeling aspects. That is, they are the parameters whose estimates (according to the best physics conclusions D0 can reach) need to be refined by fitting the shape of the data being analyzed.

The naive data analysis mantra would be to fit the data to the background-only hypothesis (using a best likelihood fit), and then to the signal+background hypothesis (possibly arriving at different best values of the s-parameters for the two models). By comparing these likelihoods, one could assign a degree of improbability to the signal+background hypothesis.

But this naive approach ignores the fact that tuning for the best s-parameters could very easily affect the likelihood for the signal+background case much more (or less) heavily than for the background-only case. Thus a log-likelihood ratio LLR of some value tells you nothing honest about whether to accept or reject the hypothesis.

To get a feel for what a meaningful LLR difference would be, one does a large ensemble of pseudo-experiments (PEs). For each PE, you pick a hypothetical set of  $s$ -parameter values, and generate a pair of fake data distributions based on those values (with Poisson variation in each bin): One distributions based on the signal+background hypothesis, the other on background-only. Then for each of these two “data” distributions, you do what the naive analysis would have done with that experiment data: Fit the  $s$ -parameters, and find a LLR.

If you gather the LLR’s calculated based on the signal+background distributions you end up with some roughly Gaussian histogram of LLR values. Similarly the LLR’s calculated based on background-only distributions gives some different histogram. Together these two histograms provide an honest basis for evaluating whether some LLR value is more “signal+background-like” or more “background-like”, and for quantifying that evaluation. And finally, one applies that honest evaluation on the LLR found for the actual observed data.

So this is what Collie does – forming an ensemble of thousands of PE results takes up almost all the time. The primary goal of this review is to provide code changes that will reduce the total time taken. Since the code was created by a sophisticated physicist but not a coding expert, it was hoped that there would be some easily achieved major performance improvements.

## 2.1 Review Ground Rules

This is existing, working code, incorporating physics and statistics technique consensus among the body of D0 scientists. The primary objective is to create a version of the code with the identical result behavior, which runs faster.

Accordingly, the desirable changes will be those which do not alter the algorithm at all, but by eliminating wasteful techniques or rearranging structures, gain in speed. (Memory size usage is, to first order, not a serious issue for this application.)

Equally desirable would be assembler-level coding to exploit special hardware for large gains in performance in critical loops. We have not pursued this potential, because it looks to be fairly difficult. But since the number of “hot-spots” can be made to be small, it is not out of the question that after restructuring the code for optimal memory access patterns, work on hand-coding the one or two critical loops, making use of the SSE vector floating point registers, might pay.

Another equally desirable improvement would be changes that allow better exploitation of multicore architectures or multiple processors. We do not pursue this at all, since the problem is “embarrassingly parallel”: Full parallel resource usage can be achieved by farming different  $m$ -value combinations to different cores, and that is assumedly already being done.

### Less Straightforward Changes

While any change that modifies order of operations may have the potential for altering results at the last-bits level, and that is fine, there are also some types of changes which are attractive from the performance standpoint but which have the

potential of causing larger differences in results. Suggestions for such changes are less desirable as an outcome of this work, and we are presenting only a very few. These are chosen on the basis of being fairly safe in the physics sense, or having potential for great improvement with little loss of physics precision.

In each case, I can easily supply the coding for the modified algorithm. However, it will take physics judgment (or in one case statistical reasoning) to decide whether a given change is worth pursuing, and I don't have the standing to make that decision for D0.

Two sorts of changes, in particular, are deemed not to be in the scope of this review. The first is the use of Minuit as the minimization driver. It is alleged that there are superior minimization programs, but the physics community is by far more comfortable with Minuit, and it is not my business to question that comfort.

The second is the algorithm used for automatic finding of the cross section limit for a given confidence level. This involves using Ridder's algorithm to find the value of signal strength which yields the specified CL; but each step in this algorithm uses a different ensemble of PE's. The issues of whether it is better to use the same ensemble of PE's for some sub sequences of steps, of perhaps using fewer PE's in earlier steps, and in general, of creating a superior algorithm is out of scope for this review. This is particularly so since the critical use of Collie does not use the automated cross section limit search capability anyway.

### **Expected Deliverables**

What is wanted is a set of modified source codes, along with performance measurements that quantify how much performance is gained when each modification is implemented.

Along the way, it is necessary to take a careful look at the code, and this will uncover various "unimportant" problems. A secondary deliverable is a collection of erroneous code fragments, which assumedly are innocuous in the current mode of usage but which clearly do not reflect the developer's intentions.

Poor coding techniques from the viewpoint of easily understanding, maintaining, optimizing and enhancing the program will also be mentioned if they are particularly serious, but this is not the purpose of the review.

## **2.2 Test and Evaluation Programs**

For initial exploration, I have started with `exampleLimitCalculation.cc`. This is a (fairly) clean example program, but has the long-range drawback of using atypical values for the numbers of bins (20) and s-parameters (6, of which 3-4 actually vary in the fits). I was told that the typical application uses more like 50 bins and 10 s-parameters. The heavy time consumers (which assumedly are critical to optimize) use up to 2500 bins and 100 s-parameters (of which assumedly many more than 4 will vary.)

The initial program is fine for learning what we can from profiling, and for trying out the effect of various optimizations. However, quantitative statements

about performance gains need to be made in the context of more realistic uses of Collie. This is particularly so because one key optimization will involve improving memory striding behavior; for small cases such as the example, it is possible that the lowest level cache will hold enough that this optimization will not help, even though on the realistic cases it may be the most important improvement.

### Modifications for easier exploration

In the context of the example program, I have made several modifications to try to make examination of performance easier without changing the overall behavior (except in trivial ways).

- I change the example program to use `CLfit2` in place of `CLfast`.
- I freely insert a line at the beginning of `calculateCLs` to set the PE iteration limit (effort) to some reasonably small number. Depending on whether the code is being profiled, timed, or debugged, this can vary.
- For profiling purposes, I moved some code which had been inlined (but was large enough that the compiler might not be inlining it anyway) into non-inlined code, which the profiler would see.
- I temporarily split the huge `fillArrays` method into 5 parts, so that I could validate the assumptions about where time was being used.

Probably the most important temporary modification is the removal of irreproducible behavior. It was observed that identical runs of the example would yield wildly different (by about a factor of 1.5) numbers of fits, even though the time taken per function evaluation in those fits remained steady. I surmised that this was because of some randomization at initialization time; this is desirable behavior in a Monte-Carlo application (to avoid accidental replication of statistics). However, it makes it difficult to compare times, and nearly impossible to check the invariance of results with coding changes.

For the purposes of these investigations, temporary changes in how `rp_` is constructed in `CLfit2.cc` and in how `m_randgaus` is constructed in `SigBkgdDist.cc` appear to be the only ones needed to eliminate this variation.

- In `CLfit2.cc` and `SigBkgdDist.cc`, instead of basing initial engine states on `gettimeofday`, I substitute a constant.

(`rp_` in `CLfit2.cc` and `m_rp` in `FitTest.cc` also construct random engines using `gettimeofday`, but these do not occur in the timing tests concerning this project.)

Finally, to get a handle on how the speed improvements vary with numbers of bins and/or s-parameters, I made changes in `collieIOexample.cc`, to produce a different root output file with more bins and so forth.

- `collieIOexample.cc` is, during the measurement phase, adjusted to use larger numbers of bins and/or more s-parameters. Performance behavior can be plotted against problem “size.”

## 3 Time Profiling: Targets For Improvement

### 3.1 What Does Collie Do?

Collie has an overall structure that loops over mass combinations, but for the purposes of this note, we care about what happens for one particular combination. The physics information (about histogram shapes) originates, for this purpose, in a file read in; the action begins when `CollieLoader::get(...)` is called, to form a `SigBkgdDist`.

This `SigBkgdDist` is used as a basis for computing confidence limits corresponding to various signal strengths (that is, rescalings of the signal in the model implied in the `SigBkgdDist`). A key number delivered by Collie, at least in the example program, is the signal strength at which we can reject the signal+background model at a confidence limit of 95%. This is calculated by the `CrossSectionLimit` class, in the crucial line

```
csLim.calculate(*sbd,clresults);
```

which is where almost all the time is spent.

`CrossSectionLimit::calculate()`, in turn, explores in the space of signal strength, using a root-finding method (Ridder's algorithm) to look for that strength at which the confidence level is 95% (or whatever desired level has been set). That means that it needs to repeatedly call a function which delivers the CL for some given trial value of signal strength. Each of these is done by calling `CLfit2::calculate()` (or some other corresponding `calculate()` method depending on the chosen treatment of systematics).

The bulk of the work is then done within `CLfit2::calculate()` calls (and their descendants in the call tree). This is the code being optimized in this review.

One can ask how optimal is the pattern of trials done by `CrossSectionLimit::calculate()`, and whether significant performance improvement could be obtained by tweaking this. That question arose late in the review process, and potential things to investigate appear in appendix D. At any rate, this has been deemed to be out of scope for this review.

### 3.2 Calculating CL For a Given Signal Strength

The `calculate()` method in a `clcompute` class such as `CLfit2` is based on a `SigBkgdDist` instance defining the data and models, a signal strength which rescales the signal model, and a choice of how many PE's to to to evaluate this one confidence limit.

The `SigBkgdDist` class holds 3 collections of `CollieDistributions`, corresponding to data, signal, and background. It also holds 5 vectors of doubles, logically containing expectation values (or data values): `data`, `signal-parent`, `signal`, `background-parent`, and `background`. These are derived from the `CollieDistributions` when `fillArrays()` is called; the constructor calls `fillArrays()`. (This call to `fillArrays()` uses `varySyst` as `false`. In consequence, `signal` and `signal-parent`

end up identical, formed for each bin by adding results of `getEfficiency` for several distributions; similarly for background and background-parent.)

The `SigBkgdDist` object is passed to `doCLs` as the `sbd` argument. The first thing done by `doCLs` is to create two reference `SigBkgdDists`, by (deep) copying `sbd` and replacing its data with either its background alone (`blike`) or its background plus signal (`sblike`). Two other copies are also created: `sbBase` is hardwired as the `SigBkgdDist` that `fitProfile()` works with, and `rand` is used in the PE loop while pseudo-data is being formed. The fates of these four `SigBkgdDists` are:

**sbBase** has signal and background expectations which which always remain the expectations from `sbd`. But it takes its data from either `sblike` or `blike` for purposes of various fits.

**sblike** is used to hold the expectation data for the signal plus background model. When that model is being used in a fit, data is copied from `sblike` to `sbBase`. This is an inefficiency which could easily be eliminated by building the data in `sblike` itself and setting the `SigBkgdDist` used in `pLH` to `sblike`; internal code evidence indicates that may have been the original intent. However, profiling indicates that this copying cost is negligible, so that change is not worth making.

**blike** is used to hold the expectation data for the background-only model. When that model is being used in a fit, data is copied from `blike` to `sbBase`.

**rand** is used as a place to compute the s+b and b models based on “fluctuated” s-parameters. After `rand.fluctuate()` is called, the signal and `bkgd` arrays in `rand` are used to provide mean values for Poisson variates which go into the `data` arrays for `sblike` and `blike`, respectively.

At this point, we calculate three log-likelihood ratios (LLRs), all based on expectations which do not include any fluctuation of systematics: that for the actual data; that for data which matches the expected signal+background, and that for data which is purely the expected background.

Now we do a bunch of pseudo-experiments, all of which are based on the `SigBkgdDist` `rand`: Each PE has the following steps:

1. Fluctuate the systematic parameters used by `rand` to compute its expected distributions. (This is the `rand.fluctuate()`, which does `varySystematics()` and then does `fillArrays()`. In this call to `fillArrays`, the effect of systematics changing is turned on.)
2. Based on the signal and background expected distributions for `rand`, the data for `sblike` and `blike` are set, using Poisson variates for each bin.
3. The s+b pseudo-data in `sblike` is used to do two fits, one with signal included, then one without signal. This gives a log likelihood ratio for this sample s+b, based on this sample underlying systematics.
4. The b pseudo-data in `blike` is used to do two fits, one with signal included, then one without signal. This gives a log likelihood ratio for this sample b-only.

5. Some bookkeeping, mostly irrelevant for performance considerations, is done on these LLR's.

One can notice that most of the arrays in the four `SigBkgdDist` objects are moot. For example, the parent distributions in `rand` are never used. While this is inefficient and somewhat misleading style, it has negligible effect on the overall performance, and it is not worth changing.

Finally, Collie uses the collection of LLR's produced in the course of these pseudo-experiments (and the LLR's calculated using the actual data) to produce the confidence limits and other result quantities.

### What determines how many doCLs are called?

Although it may not make a difference for optimization, it is notable that while `calculateCLs` (in `CLfit2` or another systematics-treatment class) calls `doCLs` with some fixed number of iterations, and each `doCLs` call creates a semi-fixed number of pseudo-experiments (between its iterations argument and twice that), the limit calculation for a given mass point may do many `calculateCLs` calls. While exploring the nature of program time consumption, it would be useful to have a handle on this. The relevant code is in `CrossSectionLimit.cc`.

When the number of PE's in each outer iteration is changed, the effective random sequences driving each PE after the first become completely different. The response of both the the bracket mechanism and the ensuing Ridder's algorithm to such changes (which fluctuate the value of confidence level obtained for a given fixed signal strength) are difficult to anticipate. For example, using the reproducible seeding put in place for study purposes, a sample change of 2% in number of `doCLs` per PE yielded a 35% change in number of sets of PE's used overall.

Therefore, when studying performance, one must simply use as the metric time per PE, and studying small deviations from linearity in that metric is not feasible within the context of the example program.

## 3.3 Where Does Collie Spend Its Time?

Of course, virtually all the time in the program is spent analyzing the many pseudo-experiments. A general analysis of the flow of calculation for each experiment shows that almost all the time will be spent evaluating the likelihood function during the fitting of the s-parameters. That is, `Minuit` calls `chiFun()` many times for each fit. Any large loops occurring in the course of `chiFun()` would be expected to dominate the time used.

The original guess was that most of the time would be spent in `calculateChi2LLR()`, in particular, on a line reading

```
if(m_bkgdParent[i]>0)
  sigLog = log10(1.0+m_signalParent[i]/m_bkgdParent[i]);
```

Actually, the entire `calculateChi2LLR()` function occupies only about 20% of the total time spent, and that is expected to decrease as the number of s-parameters grows.

By far the largest fraction of time is spent in `fillArrays()`. This is where the effects of changing s-parameters are translated to changes in the expected b and s+b distributions. For every bin, and for each s-parameter, some contribution to the overall efficiency into that bin is computed based on asymmetric response parameters  $\sigma_{s,i}^+$  and  $\sigma_{s,i}^-$  (where  $s$  is an s-parameter index and  $i$  a bin index). Minuit samples by changing the s-parameters; for each trial point, several calls to `calculateChi2LLR()` will result in that same number of calls to `fillArrays()`.

Within `fillArrays`, almost the entire time is spent in the two loop structures that loop over signal or background channels, the distributions in those channels, `binX`, and `binY`. For each bin in each distribution, a call is made to `getNormalizedBinValueVaried()` (which is inlined in `io/CollieDistribution.hh` in the original code). That, in turn, loops over the s-parameters, computing the efficiency contribution for each “active” s-parameter for this bin.

### 3.4 Time Profile for the Initial Program

To get a baseline, we profiled time spent in the version of Collie we started with.

Profiling was done using an early but good version of the Paterno/Kowalkowski “SimpleProfiler,” which was already present on `clued0`. Since it might be useful in the future for D0 users to do some profiling, instructions are given in appendix C.

Thus far, only the standard Collie example program (but with `CLfit2` used in place of `CLfast`) has been carefully profiled. Although at first the headache of doing different problems each time (due to use of `gettimeofday` to seed the randoms) was present, that only caused variations in the profiler results for each function by about a fifth of a percent. (Of course, subsequently a more stable seeding approach was temporarily put in place, so that overall timings could easily be compared—see section A.2.)

The principle consumers of CPU time were:

- `getNormalizedBinValueVaried()` consumed 46.4% of the time. Of this, other profiling runs indicated that about 2/3 was in the key `fillArrays()` loop done for background distributions, and 1/3 in the loop for signal distributions; this is easy to understand as there were 2 of the former and one of the latter.
- `calculateChi2LLR()` consumed 18.9% of the time. (No further breakdown of this routine was done.)
- The background and signal loops in `fillArrays` consumed 8.9% and 4.0% respectively. At the time of initial analysis, I had not looked carefully at whether this can be streamlined. (It turns out that some routine cleanups up unnecessary computation have cut this time by a factor of four.)
- The next heaviest consumer is the `sin()` function, which assumedly is being called internally to Minuit to implement parameter limits. This consumes

3.6% of the time in the example profile looked at for these numbers; it was more like 4% in other profiles.

- The sum of all the various Minuit routines (other than the fact that it calls `sin()` and of course the user function) amounts to about 4.7%. No single one of these is itself more than 0.8%, so they would not make this list. Of course, nothing can be done about this cost, except if the use of improved starting points reduces the number of migrad steps taken.
- `dist->getEfficiency(binX,binY)`, called from `fillArrays()`, consumes 2.7%. About half of that probably consists of recopying parent distributions which can be done much more efficiently.
- The `chiFun()` function in ProfileLH consumes 1.0%. Other than permuting the parameters used in Minuit to those known but Collie, this is pure scaffolding for calls to `fluctuate()` (which calls `fillArrays()`), `calculateChi2LLR()`, and others.

This adds up to just over 90%; the remaining time is spread over quite a few functions, and would be too much work to optimize, given the small possible payoff.

## 4 Most Promising Performance Improvement Suggestions

This section describes a unified set of performance improvement suggestions, with the following motivation: Within the bounds of the scope of the review, I consider a primary idea: the single change with the greatest potential performance gain. (This is “Caching of Partial Likelihoods”, as described in section 4.1.)

It is, of course, possible that the choice of primary idea is a failure: Either the technique is too difficult to implement, or after implementing it the speedup obtained is completely disappointing. But assuming the speedup is good, then this can be used as a base for the improved Collie.

I then look at the other principle performance improvement ideas, and include any which meet two criteria:

1. The idea must be compatible with the primary idea, in the sense of not complicating its coding and not clashing with what it is trying to do.
2. The idea must have potential for significant speedup.

After implementing these most promising suggestions, a delivered faster Collie should be made available. If the speedup is adequate, it may be right to call it quits rather than doing other suggested improvements described in section 5.

### 4.1 Caching of Partial Likelihoods

At a late date in the review thought process, the following notion arose:

Migrad (the algorithm used in Minuit) computes the gradient and (implicitly via prior trial points) the Hessian of the function at the trial point, by taking small steps in the various directions and subtracting values from the value at the trial point itself. This means that relative to some reference trial point, the `fval` values for all but one of the `s`-parameters will be the same as for the trial point!

There is certainly the potential here for saving these values and only computing (in a loop over bins) the one `fval` for the changed `s`. This is just a bit awkward as currently structured, since the savings comes from not having to do the bin-loop, but you do have to do the bin-loop for the single changed value of `s`. However, by creating a single-`s`-value version of the critical `getNormalizedBinValueVaried()` function, one could probably make this work without too much grief.

Of course, one would have to verify that the anticipated pattern of sampling points is what actually happens; one would be vulnerable to changes in the Minuit technique that change this pattern; and one has to cope with the fact that we don't know the number of `s`-parameters at compile time. Probably, one would need to investigate the pattern of same and different `s`-values, and verify in code that the expected pattern is valid before taking the shortcut of using partial values. (This study has been done; see section 4.1.)

Also, the example program, which uses only 3-4 active `s`-parameters, might not gain a lot from this technique. There may be better potential in cases where the number of fitted parameters is large.

The main worry in this technique is that the natural ordering of computation becomes `s`-parameter major rather than bin-major (since the overhead of checking which `s`-parameters need not be recomputed would be large if done for each bin separately). Thus the idea cannot easily be combined with the promising data restructuring technique discussed in section 5.1. Also, it probably does not combine well with gains obtained from supplying explicit gradients (section 5.3) if those are put in place. Finally, while possibly very beneficial, this technique contains some subtle possibilities for losing back all the speedup (particularly if there are relatively few bins).

Caching of partial likelihoods is not easy to implement in a promising way. The simplest thing to try would be to check, in the inner loop in `getNormalizedBinValueVaried()`, whether a given `s` value is changed relative to some stored base value (or relative to its prior value), and if it has not, simply look up the corresponding result on a by-bin basis. That approach would almost certainly be comparable to or more costly than the current code. Instead, one has to re-order the loops in the `fillArrays-getNormalizedBinValueVaried()` combination, such that the `s` loop is the outer loop. This is a substantial change.

Caching of partial likelihoods should be be tried, because the large potential payoff. The experiment will be more meaningful if realistic test cases are available.

### Observations relevant for partial likelihood caching

In the example program, Minuit is moving in 5-parameter space, and the distributions have 3 to 4 active `s`-parameters. The pattern of function requests for each

obvious trial point looks like:

- The trial point itself. I'll call this the base point.
- One of the parameters increased by a little bit.
- That same parameter decreased by the same little bit.
- (Usually; in 4 of 5 cases) That same parameter increased by about twice as much.
- That same parameter increased by the larger amount.
- Move on to another parameter, resetting the one that had moved to its base value. (The parameters are not done in any particular order.)
- Often, but not always, a way-off sample point not used as a base.
- The computed next trial point - a new base.
- The pattern repeats.

Thus much of the time exactly one parameter has changed relative to some base point. To quantify this for the example, in which there are 3 distributions with 3, 3, and 4 active s-parameters:

- One pattern consists of 20 evaluation points.
- That adds to 20 times  $(3+3+4) = 200$  f-value computations for each bin.
- Two of the points are base (actually, one is the base of nothing in particular, but costs as much as a true base point since we can't anticipate that storing its information is a waste of time.) 20 f-value computations per bin.
- Of the 18 remaining cases (which in principle require one f-value per bin for each of three distributions) only 2/3 of the s-parameters that change are active for the distribution being looked at. That gives 36 f-value computations per bin.

After about 5 or 6 such patterns, minuit does a block of one base point and 15 points which each differ in 2 parameters from that base. While in principle caching can speed these up as well, this was considered a potential future optimization step (see section 4.8). Effectively, this amortizes to roughly 3 base points per pattern – 30 f-value computations per bin.

Thus the example program could take only  $86/230 = 37\%$  as long as before. Of course, storing and retrieving the cached numbers will give back some time, and once the `fillArrays()` work is reduced by this much, the remaining pieces of code will become more important. Still, the net reduction could be significant.

### Techniques for caching of partial likelihoods

In coding the caching of partial likelihoods, we have to be careful not to add significant conditional logic in the innermost loop, since that would immediately give back all the potential gains. We also have to watch out for trading some

easy computations of `fval` for equally costly (or worse) memory accesses to look up cached values.

The code section which is to be overhauled is the pair of huge loops in `SigBkgdDist::fillArrays`, that is, the loops over channel (`iterSB`), Collie Distribution (`iter` or `dist`), `binY`, `binX`, and (internal to the invoked (`getEfficiencyVaried()`) `s`). In order to avoid alterations scattered throughout the package, we will still place the results in `m_signal` and `m_bkgd` by calling `signal()` and `bkgd()`. The strategy for restructuring these loops is:

1. Distinguish the case where `varySyst` is true from the case where it is not. The latter takes much less time and, for the purposes of this optimization, can be left as is. The routine that is called when `varySyst` is true will be named `adjustExpectations()` and will take as arguments:
  - `channelStart`
  - `channelEnd`
  - perhaps other control handles
2. `adjustExpectations()` will see if this is a totally new trial point, or if only one `s` has changed relative to the last “base point.” Note that his logic is done just once per trial point.
3. Assume this is a tweak on an existing base point, with parameter `s` changing. (The new-base-point alternative will be considered separately.) Then:
4. Loop over Channels (other than the excluded channel, if any) and distributions within the channel:
5. Translate `s` to the “local” `s'` for this distribution. If this `s` is moot to the distribution, simply move on to the next distribution.
6. Decide whether for this particular `s` you will use the large-negative, central, or large-positive formula. Three separate but similar routines are potentially invoked. While we are at it,, this is the point to accommodate any sort of bridge functions other than the  $(-1, 1)$  quadratic bridge, and to detect the need for exponentiation for form a log-normal efficiency—this avoids the need for conditionals inside the loop over bins. Each combination of choices leads to a separate small routine.
7. Pre-compute the coefficients (functions of `s`) to be used with  $\sigma_P$  and  $\sigma_N$  to form the interpolating function for each bin. In the case of the quadratic bridge, this saves just one multiply and perhaps two adds per bin. But if the interpolating function used is more heavyweight, such as a true sigmoid, but is not different for each bin (other than in the values of  $\sigma_P$  and  $\sigma_N$ ), then this implements the suggestion in section 4.3, at perhaps a significant gain.
8. Decide whether the `linLog` model is used for this `s` on this distribution. If so, all three branches will revert to the common `linLog` formula involving an exponential.
9. Having done all possible preparatory work so as to minimize the work needed for each bin, now loop over bins:

10. Apply the appropriate formula to compute the factor. In the large negative and positive cases, only one sigma needs to be looked up.
11. Truncate to  $\epsilon$  if too small or negative.
12. Multiply by `dist->partialProduct [s'][z]`.
13. Although we may want to eliminate this conditional in the near future, because a similar truncation will happen after scaling, again truncate to  $\epsilon$  if too small or negative.
14. Finally, multiply by the `sigScale`, re-truncate, and increment the appropriate signal or background expectation.

### Memory usage for partial likelihoods

For each active s-parameter in each distribution, the caching of partial likelihoods method requires storing one value per bin: The product over all the other s-parameters of the smeared efficiencies. This can be compared to the two numbers  $\sigma_N$  and  $\sigma_P$  already stored. Thus the total memory usage in this category is increased by about 50%.

To get some feel for the sizes we are talking about, let us consider an extremely large problem, in which 2500 bins are used, and there are 8 channels with an average of 3 distributions in each. Say we have 100 s-parameters, and for a typical distribution 50 of those are active. Then the current storage need is about 48 Mbytes, and this would be increased to 72 Mbytes. This does not have the feel of a prohibitive increase.

## 4.2 Factoring Out fval Logic For Each s Value

The logic in the critical step in of computing fval in `getNormalizedBinValueVaried()` does different things depending on whether  $s$  (the named variable in the routine is `rand`) is between -1 and 1, greater than 1, or less than 1. These `if` statements may be interrupting instruction flow, particularly if one  $s$  is in one range and the next is in a different range.

This can be cured completely by looping over  $s$  more slowly than the loop over bins. The point is that  $s$  does not change, so by doing one overall decision of which looping routine to call, you can eliminate all the decision logic within the routine itself. It is plausible that this was not tried because it implies storing some result for each bin, and at the end going back over the bins to take a product (or storing the “product-thus-far” for each bin and pulling it back in for each  $s$ ). So it is unclear whether such a restructuring, on its own, would represent a gain or a loss.

However, this restructuring meshes perfectly with restructuring already needed for caching partial likelihoods. The matter of storing things for each bin is already settled in that case; factoring out the fval logic would not add extra memory traffic. So when that optimization is done, this change should be made as well.

### 4.3 Pre-computing s-dependent Part of Interpolating Function

Let's say it is decided that for physics reasons, the interpolation between a sensitivity of  $\sigma_N$  for large negative  $s$  and  $\sigma_P$  for large positive  $s$  ought to be a true sigmoid. Taking the exponential once per bin is overly costly, but that exponential depends only on  $s$  and not on the specific bin number.

When this optimization is done, it will be allowable for different distributions to use different choices of interpolating functions for a given  $s$ . Of course, to exploit that, somebody would have to apply physics thought to the issue of which choices are best. For now, we will just hardwire in either some fixed sigmoid or the original quadratic bridge.

### 4.4 Low-efficiency Cut Folding

When taking the product of partial efficiencies, where each is a monotonic function of the relevant s-parameter, it is always possible that some partial efficiency will come out negative. This, of course, would give ridiculous results, particularly if two negative likelihoods were multiplied to give a non-zero positive likelihood.

To deal with this, Collie truncates values in three (!) places:

1. Each individual fval is truncated to be above  $10^{-5}$ .
2. Then the overall fluctuation for one distribution (in one bin) is truncated, again at  $10^{-5}$ . (It is easy for this to happen if two or more individual fval's were small. But in many such cases, the third cutoff would also take effect.)
3. Finally, the overall contribution to the bin from this distribution, which is the "efficiency" times the signal scale, is truncated to be above  $10^{-6}$ . (This can happen if `fLinBins[i]` times the `fEfficiency` times the signal scale is below 0.1, since `getEfficiencyVaried()` will return the fluctuation times `fLinBins[i]` times the distribution `fEfficiency`.)

(See section 7.5 for a discussion of the nature of these truncations.)

The first of these truncations is an ugly necessity based on non-physical properties of the simplified interpolation scheme (for sigmoid interpolation it would not be needed).

The third is a practical desire, to avoid bins with expectations so small that they distort all fitting if there is any data there at all.

But the second overall fluctuation cutoff is not serving any vital purpose; it looks to be there so as to avoid zero bins, but that is already done by the third cutoff. The impact of this second cutoff, on performance, is that it requires an extra conditional and prevents combining the effects of `sigScale` and `fEfficiency`. Since the physics impact must be extremely low (given tolerance of the peculiar existing cutoff mechanism as discussed in section 7.5) it must be sensible to do this combining, and use only the two cutoff stages.

After talking with Wade Fisher, it was determined that as long as the purpose of the cuts is upheld, any approach is OK. In the end, we have a single cut when the partial efficiency is computed, and this does the trick.

## 4.5 Rectifying Inefficiency in Calculating Chi2 LLR

There are minor inefficiencies in `calculateChi2LLR()`. Since this method consumes close to a fifth of the total time, any easy fixes here are worth trying.

The opportunities for improvement that we see are, in order of code appearance (not, that is, in order of potential impact):

- The `if(!fitSig)` test can be pulled out of the bin loop, breaking that loop into two separate loops. This saves one conditional in each iteration; I would expect that savings to be negligible since the chip almost surely will do successful code look-ahead on this.
- The two lines starting with `sigLog = 10` can profitably be combined into a single `sigLog = (m_bkgdParent[i] <= 0) ? 10 : whatever`. Again, expected savings are small.
- The whole business of checking whether to skip because of the sigLLR cutoff is done in a very inefficient way. Rather than comparing sigLLR to the  $\log_{10}$  of  $1 +$  the parent signal-background ratio, it is equivalent to compare the parent background to a number (dependent on sigLLR) times the parent signal. The point is that this number can be computed at the start; it is the same for all bins. (This is so even ignoring the probable situation that sigLLR is passed as the same value many consecutive times). The expected savings are about 25% of the total time taken for `:calculateChi2LLR()` since one transcendental calculation out of four, plus one division out of three, are eliminated.
- The loop over `m_delbins[jdel]` looks to be disastrously inefficient; much better is to create a single look-up array of bools and break if this specific bin is among the deleted. (Note that his loop is also the subject of a possible error in coding, discussed in section 7.5.)
- In the test for data and model both being positive, I believe the model is inherently positive since small minimum is applied if it is less than or equal to zero. Thus one comparison can be avoided.

These changes are fairly safe and do not require substantial or global code reorganization, so even though the potential payoff is not huge, they are worth a try.

## 4.6 Non-limit Minuit Variables

It is observed by profiling that about 4% of the original program time is being spent in the `sin` routine. I infer that this is called by Minuit, in translating from

the internal value of a fitting parameter, to the external value of one of our s-parameters. (Minuit uses an arcsin transformation and needs to invert it to deliver the point values to our `chiFun`.) It should be possible to allow unlimited parameter variation; our function is sufficiently well behaved far from the origin that the various pathologies people worry about when forcing limits are not going to occur.

To eliminate this cost, I would need to (re-)learn how to tell Minuit to not limit the parameter ranges. This may be an easy thing to try.

The potential gain is 4%; this is of course increased in proportion to whatever other gains are achieved by other optimizations. However, as the number of bins increases, I'm fairly certain the relative amount of time spent doing these transformations will fall reciprocally.

## 4.7 Improved Fit Starting Points

The current code starts each fit at s-parameters all zero. While we know the values are centered somewhere near there, we may be able to do better. The possibilities are discussed in more detail in section 5.4; here we describe only the most promising choice.

### Cross-informed starting points

The most interesting thing to try is something which can also be done in the actual data analysis: Having fit the s-parameters for the background-only case, you have what might be an excellent starting point for the s-parameters for the signal+background case. This does not help at all in half the fits, but might help a lot in the other half.

## 4.8 Partial Caching Extended to the 2-Change Case

During the fit, Minuit generally either changes all the non-pinned s-parameters, or changes just one relative to a recent “base” point (see section 4.1).

However, by looking at the pattern of trial points, we observe that the last group of FCN calls starts from a base point and varies *two* parameters for each request. Clearly, this is being done to formulate off-diagonal terms in an error matrix.

It is conceivable that the error matrix is not needed for the pseudo-experiment fits, and in that case it may be possible to change the way ProfileLH controls Minuit such that the unneeded function calls are eliminated. But if not, since the number of these calls grows quadratically with the number of s-parameters, it is worth trying to speed these up.

A technique very similar to single-change partial caching can apply to this case. The idea is that for each bin and distribution you store, for each s-parameter, the fval produced for that parameter along with the product of fvals at the base point over all the other s-parameters. This allows rapid computation (though not quite as rapid as for the single-change case) of the 2-change efficiency.

(A slight refinement would be to store the reciprocal of the fval for the parameter; this changes a per-bin divide into a multiply. This is not necessarily a win, however, since it requires a divide for each point whenever a new base point is detected.)

Although this is closely related to the improvement discussed in section 4.1, it is right to wait on implementing it, to see if the extra calls can be eliminated altogether by appropriate Minuit control.

## 5 Principal Other Performance Improvement Suggestions

### 5.1 Data Structured For Rapid Expectation Calculation

The current loop structure in the critical `chiFun` path (executed every time one new-s-parameters step in the LLR fit is done) is a loop over bins  $i$  (or  $(i, j)$ ). For each bin, there is a loop over values of “internal” parameter  $s$  (that is, the parameters being varied by Minuit), in which:

- The internal parameter  $s$  is translated to an “external” parameter  $s'$  via a look-up array.
- $\sigma_P$  and  $\sigma_N$  are looked up in an array whose fast-varying index is the bin index, and whose slow-varying index is  $s'$ .
- The deviation of the parameter is combined with  $\sigma_P$  and  $\sigma_N$  so as to form an efficiency number  $f_s$ .
- The expectation for that bin is proportional to the product of those  $f_s$ .

Later, the `calculateChi2LLR` function retrieves this overall bin value, which is an expectation value, and compares it to some (pseudo-)data value to form a contribution to the likelihood ratio.

The problem (or more cheerily, the optimization opportunity) is that the innermost loop over  $s$  deals with data for one particular bin, that is, data which is maximally spaced in the expectation fluctuation arrays. In the example program, the number of bins is 20, so we are walking two arrays (for  $\sigma_P$  and  $\sigma_N$ ) with a stride of 160 bytes. In most chip architectures, a single read brings in, from the lowest level of cache, about 16-32 bytes; the miss latency to the L1 cache is not disastrous but probably burns several cycles.

For the realistic applications, the number of bins is larger (at least 50) and the number of s-parameters is more like 10. This introduces the likelihood of frequent misses in L1 cache due to the stride size of 400 bytes. And for the largest and most time-consuming applications, there are 500 bins (and 100 s-parameters), virtually guaranteeing frequent cache misses in this loop. While it is conceivable that the chip architecture is so sophisticated that this latency is completely hidden, the

presence of conditionals in the remainder of the loop makes it more likely that some memory access time hit is being taken.

The change to reduce this inefficiency is to prepare data in the best possible format for walking through to compute the efficiency value for each bin. (While we are at it, the data should be arranged in order of internal  $s$ -parameters, so that the translation for each  $s$  is not needed.)

This auxiliary data, which we shall call `ResponseCache` consists of an array (indexed by bin) of a structure of the form:

- $n_s$  pairs of  $(\sigma_P, \sigma_N)$  corresponding to the values for each (inner)  $s$ , for one specific bin. Only active  $s$  values should be present, and the ordering should be such that a simple linear traversal in  $s$  gives the needed values.
- $n_{\text{bins}}$  such sets, one for each bin.

Who “owns” each `ResponseCache`? And when is it prepared?

Since `CollieDistribution` holds the `fLinSystPos` and `fLinSystNeg` arrays, a natural possibility is to also have each distribution hold its own segment of the `ResponseCache`. The loop over distributions in `fillArrays()` is outside the loop over bins, so the time lost in the “gaps” between one distribution and the next should not be large compared to the time that is currently lost in going from one  $s$ -parameter to the next.

`CollieDistribution::linearize()` prepares the `fLinSystPos` and `fLinSystNeg` arrays, based on data from the root file. It also prepares the maps from “inner” to “outer”  $s$ -parameters. Thus it is very natural for `CollieDistribution` to own the `ResponseCache`, and to fill it as part of the `linearize()` method. (Yes, it would be superior C++ style to construct it in the initializer list, but that would require code reorganization not aimed at performance, which this review should not do.)

Eventually, I believe the current `fLinSystPos` and `fLinSystNeg` arrays will become superfluous, so the net memory usage will actually decrease when `ResponseCache` is fully deployed. At first, a version of the code using both methods of computation should be prepared, to check that results are identical.

## 5.2 Recopying Parent Distributions

Every time `fillArrays()` is called, the parent distributions are recomputed. Since this is just a matter of summing looked-up efficiencies times a scale factor, this is not a disastrous cost, but it is unnecessary since they certainly don’t change within a single fit. We can come up with some way to skip that work when it is unneeded.

## 5.3 Enabling the Analytic Derivatives

The way the `migrad` method of `Minuit` works is that a trial point is chosen, and then a quadratic model of the function near that point is created—the minimum point of that model is found analytically, and used as the next trial point. To form this quadratic model, `migrad` needs the gradient and Hessian (second derivative matrix) at the trial point. In usual usage, it obtains this by evaluating the function

at the trial point, at two straddling points in each dimension (which allows an estimate of diagonal terms in the Hessian), and at one “corner” point for each pair of distinct directions (to give the off-diagonal Hessian terms).

Minuit allows the user to help it by providing a function to use directly as the gradient.

Enabling the use of analytic derivatives will entail three steps of work:

1. Learning how to tell Minuit to use explicit gradients. This should be relatively easy.
2. Figuring out and coding up what those derivatives are. There is already code in Collie that attempts just that; of course, any optimizations applying to the function evaluation are relevant to the derivative evaluation as well.
3. Evaluating whether the use of gradients was a net gain or loss.

Perhaps the toughest part of this would be the evaluation of the gain. The example program uses a fixed number of active s-parameters in each distribution, and a fixed number of overall s-parameters. The gain from use of derivatives is highly sensitive to the dimensionality of the problem (there is a much bigger gain from large numbers of parameters). To explore the performance for realistic running, I would need to see how to increase the dimensionality of the s-parameter space. But that involves the generation of the root files; this may not be easy.

It is quite possible that for the example case, explicit derivatives hurts performance, but that for larger cases it helps significantly. When I do this optimization, I will need to ask for help in evaluating whether it is worth keeping.

## 5.4 Options for Tuning Fit Starting Points

The current code starts each fit at s-parameters all zero. While we know the values are centered somewhere near there, we may be able to do better. A better starting point would likely reduce the number of fit iterations needed by Minuit; even a 10% improvement there would match the effect of some difficult code optimizations.

However, we must be careful not to use a method which “cheats” in utilizing information known for the pseudo-experiment but not available as part of the analysis of real data. For example, we could use “super-improved” starting points by starting at the actual values of s-parameters used to form the distributions in each PE. But unless we somehow know that the minimization is going to always converge to the same point regardless of starting point, this sort of “peeking” will ruin our confidence in the physics results because the ensemble of PE’s will have been analyzed slightly differently than the actual data—“don’t go there.”

### Central-value starting points

The example code has no non-zero central values for s-parameters, so this issue is probably moot. But if there are non-zero central values, it is clearly better to use those rather than zero for the starting points.

### **Cross-informed starting points**

The most interesting thing to try is something which can also be done in the actual data analysis: Having fit the s-parameters for the background-only case, you have what might be an excellent starting point for the s-parameters for the signal+background case. This does not help at all in half the fits, but might help a lot in the other half.

This is the technique described in section 4.7. I may try to see if it improves speed.

### **Least-squares-optimal starting points**

In principle, with some math and a lot of coding, you can find a best-fit to the s-parameters based on a linear least-squares fit. This does not match the best fit for maximizing likelihood, but is probably a really good starting point. The CPU work done to perform such a fit is small – a bunch of summing and a small matrix inversion.

However, the coding work and problem analysis that would be needed is significant. I will not be trying to do this as part of this review.

## **5.5 Combining Expectation Calculation With LLR Accumulation**

The current code loops over bins, finding and storing expectations, and then uses those expectations to compute the likelihoods. It is plausible that by immediately computing the contribution to the likelihood function one could avoid some memory accesses.

However, this would require some code restructuring in the direction of less clarity. Since the expected speedup is rather small, and goes down with increasing number of fitted s-parameters, I do not propose to try this speed improvement.

## **5.6 Ideas that Won't Help**

The following ideas were considered and looked possibly promising, but turn out to be dead ends:

### **Caching fvals for inactive parameters**

One idea was that the loop computing the likelihood was perhaps doing all the s-parameters, even the non-active ones. In that case, one could gain by carefully caching the efficiency effect of the non-active parameters and using those values for each bin.

It turns out this inefficiency is not happening. the code is looping only over “active” s-parameters; the effect of the values of the inactive parameters is assumedly taken into account. (I did not verify that this is being done properly, but given that the program has been validated, changing the way the non-fitted systemics

are treated would be like changing the physics being done—out of scope for this review.)

## 5.7 Controversial Changes

The following changes may make a lot of sense, but they do change the physics of the program and thus should not be implemented without the necessary physicist’s input.

### Improved interpolating function between $\sigma_N$ and $\sigma_P$

Given some value of an s-parameter, what is the effect of that value on the content that will be added to some bin  $b$  in the expectation for this model? Collie allows for asymmetric effects, where if the s-parameter is negative, some different response factor ( $\sigma_N$  instead of  $\sigma_P$ ) is used. (By the way, the term “sigmaP” as a variable name for that quantity is a bit misleading because it is in no way a measure of deviation; instead it is a slope of a response, or a measure of the response *when s* is one sigma from zero. But I wouldn’t suggest changing nomenclature at this point.)

It was observed that the simple approach of letting the `fval` multiplier corresponding to the value  $r$  of each s-parameter be given by

$$f = \begin{cases} \sigma_N r & r < 0 \\ \sigma_P r & r \geq 0 \end{cases}$$

is flawed, in that it presents a discontinuous gradient wherever one of the s-parameters is near zero. In one dimension,  $f$  looks like a pair of straight lines meeting at the origin. Migrad exhibits poor convergence near such a situation, and the origin is the most common region for the underlying s-parameters.

As long as the same asymmetry function is used when generating the  $\sigma_N$  and  $\sigma_P$  based on the model, as when using  $\sigma_N$  and  $\sigma_P$  for the fit, some other form of asymmetry could be used without improperly distorting the physics. In fact, one can validly study the effect of different functions as long as the same form is used for generating the expectations as for fitting the s-parameters; and this is automatic in the Collie structure.

Ideally, what one would want is a function which is continuous and twice-differentiable. For example, a sigmoid

$$f = r \left( \sigma_N + \frac{\sigma_N + \sigma_P}{1 + 9^r} \right)$$

smoothly interpolates between slopes of  $\sigma_N$  and  $\sigma_P$ , and is nine-tenths of the way to either asymptotic slope at  $r = \pm 1$ . However, the cost of evaluating  $e^{\alpha r}$  is non-trivial.

The current code uses a compromise: A “quadratic bridge” of the form

$$f = \begin{cases} r \frac{\sigma_P + \sigma_N}{2} + r^2 \frac{\sigma_P - \sigma_N}{2} & |r| < 1 \\ \frac{\sigma_P + \sigma_N}{2} r & |r| \geq 1 \end{cases}$$

This function looks like a straight line  $\sigma_N r$  to the left of  $r = -1$ , a straight line  $\sigma_P r$  to the right of  $r = +1$ , and these are joined by a parabola passing through  $(-1, \sigma_N)$ ,  $(0, 0)$ , and  $(+1, \sigma_P)$ . The first derivative of this curve is composed of three line segments: Horizontal lines at  $\sigma_N$  and  $\sigma_P$  at the far left and right, and a line overshooting these, going from  $(-1, \sigma_N - \frac{\sigma_P - \sigma_N}{2})$  to  $(+1, \sigma_P + \frac{\sigma_P - \sigma_N}{2})$ . We see that there remains a discontinuity in the first derivative (actually two of them now, and their magnitudes sum to the same amount as in the simple case). However, these are displaced to  $|r| = 1$  which is a rarer region.

Exploration using the simple formula, the sigmoid, and the quadratic bridge in the context of the example program reveals that:

- The simple formula averages about 161 function evaluations per fit.
- The quadratic bridge averages about 126 function evaluations per fit. This is quantitatively understandable on the assumption that the dominant cause of extra evaluations comes in the regions near the gradient discontinuities. A single quadratic bridge computation is a bit more time-consuming than the simple formula (especially since the latter need not look up both  $\sigma_N$  and  $\sigma_P$  for a given  $r$ ), but the savings in number of evaluations more than makes up for that.
- The sigmoid formula also averages about 125 function evaluations per fit. However, the work involved in taken an  $\exp()$  each time more than doubles the time spent in this code, and swamps any conceivable gain.

The number of evaluations needed with the sigmoid function was a bit unexpected; I would have guessed about 95 based on the gain in going from the simple formula to the quadratic bridge. What this tells me is that the “penalty” in the simple formula case comes more from the fact that we are always starting right at the origin, where the discontinuities are all concentrated, than from encounters with the discontinuities near the convergence point of the fit.

However, an “accidental” experiment with a pure quadratic function (which is way steeper than  $\sigma_N$  and  $\sigma_P$  would imply when an s-parameter is greater than 1 in absolute value) showed an improvement in number of function calls per fit over the quadratic bridge of only about 0.4%, so this is consistent with the notion that the latter is not doing much thrashing do to discontinuous derivatives. By the way, the overall time for the pure quadratic function is about 16% faster than that for the quadratic bridge. (But the physics makes a lot less sense.)

This, in turn, tells us that the currently used quadratic bridge is pretty much optimal, at least if we assume we can’t pre-compute things based on the s-parameter for all the bins in a distribution (and thus use a sigmoid almost without extra cost). Although it is tempting to try increasing the range of the quadratic interpolation region (for example,

$$f = \begin{cases} r \frac{\sigma_P + \sigma_N}{2} + r^2 \frac{\sigma_P - \sigma_N}{6} & |r| < 3 \\ \frac{\sigma_P + \sigma_N}{2} r & |r| \geq 3 \end{cases}$$

matches the simple formula outside  $|r| = 3$  instead of one), we would not gain much by such a change. Thus it is not worth the extra work (which includes

physics thought about the consequences of the form of the interpolation for this asymmetry) if all we are trying to do is improve speed—it won't help much at all.

### **Super-informed starting points**

If we were confident that the fitting always converges to a unique best point we could probably improve performance a lot by the following “cheat”: In each pseudo-experiment, take as the starting point for the s-parameters the values that were actually used to generate the data used in that PE. The data generated will not perfectly reproduce these values for two reasons: The data is formed using Poisson variates, which change the shape a bit, and we are including in our likelihood function some prior assumption about the likelihood of a given deviation of some s-parameter from its central value. Still, this “almost right answer” must be a much better starting point than the origin, and it can be used for both the s+b and b-only fits.

Of course, the analysis of the actual data would have to start at the origin (or at the optimal point found for the other fit if we so choose). So unless we know that the answer is independent of the starting point, this “cheat” will have introduced worries that we are not comparing “apples to apples.” Since the whole purpose of Collie is to avoid such uncontrollable uncertainties about technique, I don't think this approach is worth trying.

### **Analysis of optimal number of pseudo-experiments**

Probably the most important opportunity for physics or statistics thought to significantly impact time taken by this program is in deciding what the best number of pseudo-experiments should be.

For a given actual data set (containing some number of entries into the histogram bins) there is some level of precision in any confidence-level statement you could make, such that claims to better precision than that would be meaningless. This level corresponds (but not in a trivial way) to the purely Poisson-statistical fluctuations that could have occurred in the bin data (ignoring entirely fluctuations in the values of systematics parameters).

Let's say, for example, that the Poisson data uncertainty leads (at average) to a fluctuation in LLR which is 1/10 as large as the spread due mainly to systematics uncertainties. In that case, your confidence level is inherently imprecise to some degree; you could just as easily have gotten data which is a tiny bit different, and it makes no sense at all to refine your ensemble to the point where your ensemble error is much smaller than this fluctuation. So perhaps 400 ensemble members are more than enough. (Collie itself can tell you something about the spread of the LLR including systematics, of course, and it can do so with much fewer than 10000 PEs.)

Absent a fairly careful analysis of the statistics involved, the finest sensible granularity for the PE curves (hence the diminishing-returns point for the number of PEs) is just a guess. D0 is guessing 15000 because (I suppose) they want quotes

of CL to be meaningful at the less-than-1% level. If the data quantity does not support that degree of precision, then they ought to cut back on the number of PE's. This, of course, would linearly reduce the CPU time consumed.

My point is that the choice of how many PEs to use, currently embedded by a fixed choice of fine, veryfine, veryveryfine, fast, and so forth, requires some careful physics thought for optimal performance without loss of true precision. It is entirely possible that this thought has already been done, and that the data does support precision corresponding to ensembles of thousands of PE's. In that case, the current fixed levels for fine, veryfine, etc. may be quite sensible. Of course at some point the experiment has to decide whether being able to quote CL's at the 0.1% (or 0.01%) level is worth taking the time for 50,000 PE's (or whatever that would require).

But if the data inherent uncertainty makes it statistically meaningless to go beyond some number of PEs, then this fact should be embedded into Collie. Perhaps the matter should be automate by estimating the spread of the LLR distributions in samples of 100 or so and then basing the number of PEs on those spreads.

Such an improvement, if indeed it is available, is not in the scope of this review, but the review is hereby raising the issue as a possible (and perhaps huge) way to save total CPU time.

## 6 Effects of Implemented Suggestions

### 6.1 Summary of Speedups

Looking exclusively at the example program (but using CLfit2) we have achieved the following speedups in the time per evaluation of chiFun:

- The original code took about 10.32  $\mu$ sec per chiFun.
- Re-arranging the innermost s-parameter and bin loops when computing expected profiles cut the time taken by 29%, to 7.38  $\mu$ sec.
- Eliminating parent distribution computation in places where it will not be used shaved off a further 0.35  $\mu$ sec.
- Storing quantities which would later be used by partial likelihood caching gave back that 0.35  $\mu$ sec.
- Implementing partial likelihood caching cut the time to 5.74  $\mu$ sec.
- A small tuning of the method of discovering opportunities for using the cache shaved off another 2%.
- An improvement in the calculation of chi2LLR after the bin expectations have been computed reduced the time to 5.09  $\mu$ sec.
- Eliminating the use of bounded parameters for Minuit reduced the time to 4.72  $\mu$ sec per chiFun.

This represents a net speedup by a factor of almost 2.2.

## 6.2 Profiling this Code – General Points

For various versions of the code, it is advantageous to compare times for various steps; then when optimizations are in place, one can see where the gains lie, and where there remain fertile areas of potential speedup.

All the measured times are, of course, profoundly affected by the number of calls to the `chiFun` function. Although some optimizations (e.g. section 4.7) focus on reducing the number of calls needed per fit, and there is potential for reduction in the number of fits needed by improving the algorithm used in `CrossSectionLimit::calculate()`, the fundamental comparison is how much time is needed per `chiFun`.

Even dividing out the number of `chiFun` calls, there are three categories of time measurements, based on analysis of how work grows with number of s-parameters and with number of bins:

1. Work that is proportional to the number of s-parameters times the number of distributions times the number of bins. (In analyzing this, we will assume that the fraction of “active” s-parameters per distribution is  $f$ , which tends to be about 1/2 to 2/3 in the example and in the typical cases described by D0.)
2. Work that is proportional to the number of bins alone.
3. Work that is proportional to the number of s-parameters.

Note that the effect of “we have more s-parameters so that requires more trial points per fit” is *not* taken into consideration. The reason is that we are comparing performance of the improved versus the original code on specific problems; the fact that both methods scale as some identical non-unity power of the number of s-parameters would be moot.

In this and analogous subsections, we will state times taken by time per the appropriate unit, and present the multiplied-out breakdown per `chiFun` in terms of:

**d** number of distributions

**s** number of s-parameters

**f** average fraction of s-parameters active for a distribution

**b** number of bins

We also illustrate for the example program ( $d=3$ ,  $s=6$ ,  $f=5/9$ ,  $b=20$ ) and for a hypothetical “typical” computation ( $d=6$ ,  $s=10$ ,  $f=0.6$ ,  $b=50$ ). All timings are quoted for running on `driel-clued0.fnal.gov`, which is not the fastest machine available in the `clued0` cluster, but which was found to be otherwise idle almost all the time (crucial for wall-clock overall timing).

## 6.3 Profiling Results – Initial Program

The example (using `CLfit2` and forcing a reproducible sequence of random seeds) program took  $10.32 \mu\text{sec}$  per `chiFun`. The time occupied by each function is expressed in  $\mu\text{sec}$  in the last column of this chart:

gNBVV	46.4%	per $d*s*f*b$	23.9 nsec	= 4.788
fAloops	12.9%	per $d*b$	22.2 nsec	= 1.331
gEff	2.7%	per $d*b$	4.6 nsec	= 0.279
chi2LLR	18.9%	per $b$	97.5 nsec	= 1.950
chiFun	1.0%			= 0.103
sin	3.6%	per $s$	61.9 nsec	= 0.372
minuit	4.7%	per $s$	80.5 nsec	= 0.483
other	9.8%			= 1.013

Here, gNBVV is the time taken to compute and accumulate the distributions efficiency responses to the fluctuated s-parameters (done in getNormalizedBinValueVaried()). fAloops is the loop and bookkeeping and accumulation overhead in fillArrays; this is done per bin per distribution. gEff is the non-fluctuated getEfficiency() called from fillArrays(). The total of those three – 6.398  $\mu$ sec per chiFun – represents the time spent preparing the expectations for the various bins.

chi2LLR is the time taken to compute the likelihood function once the expectations have been filled in. chiFun is the chiFun code scaffolding itself.

sin is the sin function internal to Minuit. minuit represents the rest of the time taken by the internals of minuit. This time would go up roughly linearly in the number of s-parameters.

The formula, in nsec per chiFun, is

$$T = 23.9dsfb + 26.8db + 142.4s + 97.5b + 1126 \quad (1)$$

For the “typical” case, this would become 58.43  $\mu$ sec per chiFun; and of course the number of chiFun calls needed will be expected to grow with the number of s-parameters as well.

## 6.4 Profiling Results After First Optimizations

The initial optimizations were related to partial likelihood caching (section 4.1). These focus on reducing the time taken by the gNBVV and fAloops portions of fillArrays, in particular, when minuit trial points vary from some base point only in one parameter.

The net speedup is by a factor of 1.80 – 5.737  $\mu$ sec per chiFun instead of the initial 10.32  $\mu$ sec per chiFun.

The pre-optimization time (per chiFun) taken for this part of the code was 6.019  $\mu$ sec, which includes by far the largest chunk of time consumed but is only 60% of the total. (Thus, Amdahl’s rule will get us if we speed this part greatly; our theoretical best speedup would be by an overall factor of 2.5.)

The first step in this optimization was to rearrange the two inner loops over s and bin, placing s on the outside so that we could later exploit the caching for a given sole perturbed s-parameter. This rearrangement had a side effect of mostly correcting the “large stride” issue mentioned in section 5.1. When this was put into place, the time per chiFun dropped to 6.93  $\mu$ sec. This represented an immediate speedup by a factor of 1.4, due to the reduction in time for these two combined activities by a factor of 1.9. (This was a bit of a pleasant surprise; I would have

expected a change by 20% or so.) After a few other preliminary cleanups were put into place, the time per chiFun dropped to 6.93  $\mu\text{sec}$ . I did not profile the code at this point to get a more detailed background, since I was anxious to put in the actual caching. Since this would reduce the work done in filling the expectation arrays by a further factor of .37 (in principle), I naively expected another factor of 2 speedup, and wanted to verify this as soon as I could.

Placing code for storing information needed for the partial product caching gave back those further cleanup improvements (but left in place the speedup factor of 1.4). Then when the partial caching was implemented, the time per chiFun dropped to 5.73  $\mu\text{sec}$ . This represents a further reduction of time in the two combined filling activities by almost precisely a factor of 2. (Not the factor of 2.5 I had expected, because the simple and quick bookkeeping for the non-changing s-parameters now becomes noticeable compared to the lesser amount of computational work for the changed partial efficiencies.) We have entered the realm of Amdahl's rule – we can't gain much by speeding this component up further because it is no longer the dominant component.

Profiling illustrates this:

bEQbase	7.1%	per d*s*f*b	2.0 nsec	= 0.407
abeBase	2.6%	per d*s*f	14.9 nsec	= 0.149
bEQpert	4.8%	per d*b	4.6 nsec	= 0.275
abePert	3.7%	per d*s*b	0.6 nsec	= 0.212
pExProd	2.9%	per d*s*f*b	0.6 nsec	= 0.166
fArrays	2.4%	per d*b	0.2 nsec	= 0.138
areDiff	2.4%	per s	23.0 nsec	= 0.138
fsoleD	1.1%	per s	10.5 nsec	= 0.063
chi2LLR	35.1%	per b	100.6 nsec	= 2.013
chiFun	2.2%			= 0.126
sin	6.8%	per s	65.0 nsec	= 0.390
minuit	10.5%	per s	100.3 nsec	= 0.603
other	18.4%			= 1.106

Here, gNBVV disappears, or rather it and most of fAloops together are broken into several other identified pieces. bEQbase is the time taken by binEfficienciesQbridge to compute and multiply into products the efficiencies for each bin for a specific s-parameter, under the request of addBinEfficiencies done for a base point. bEQpert is that same time, taken by the same routine, only requested while doing perturbed points. abePert and abeBase are the rest of the work done by addBinEfficiencies. pExProd is prepareExclusionProducts, where the overall products are divided by the baseEfficiency for each s-parameter.

areDiff is the function areDifferent(). This is a new and annoying place to lose time: In order to decide whether a point is a perturbation off the prior base point, we are comparing various s-parameter values. It is observed that sometimes these differ by a negligible amount (less than a part in  $10^{13}$ ), purely because Minuit has or has not used a value in a register. This would significantly increase the apparent number of base points, for no good reason. So when testing for equality, it seems we need to test for a relative difference being less, in absolute value, than

this epsilon. Surprisingly, (because this is not proportional to the number of bins or distributions) this occupies non-negligible time. `fsoleD` is `findSoleDifference()`, which calls `areDifferent()`. (Shortly after preparing this table, and after being annoyed by the time cost discovered, I made a change to these two functions, causing the total time to drop from 3.5% to 1.6%. This will be reflected in future charts analyzing further optimization steps.)

`gEff` disappears from the list of leading time hogs because one of the cleanups took it out of the critical loops. `fArrays` is the entire rest of the `fillArrays()` function. The total of the above eight functions – 1.548  $\mu\text{sec}$  per `chiFun` – represents the time spent preparing the expectations for the various bins. This is the work this optimization step was concerned about, and it is done about 4.1 times faster than before.

Finally, the apparent minuit contribution has grown (from 416 to 511 nsec per `chiFun`) because several more of the smaller routines were counted this time.

The formula, in nsec per `chiFun`, is

$$T = 2.6dsfb + 0.6dsb + 14.9dsf + 4.8db + 198.8s + 100.6b + 1232 \quad (2)$$

Compare this to the earlier formula:

$$T = 23.9dsfb + 26.8db + 142.4s + 97.5b + 1126 \quad (3)$$

For the “typical” case ( $d=6$ ,  $s=10$ ,  $f=0.6$ ,  $b=50$ ), the optimized time would become about 16  $\mu\text{sec}$  per `chiFun`, a speedup of a factor of 3.6. Why do we expect a speedup factor that is so much better than the factor of 1.8 we see in the test example? Partly, it is because as the number of bins increases the relative importance of the unimproved overheads (such as minuit’s computations) falls. But mostly it is because when we double the number of  $s$ -parameters, the gain from partial caching goes up for two reasons: The number of perturbed points per base point increases, and the gain in using the `exclusionProduct` rather than recomputing for every  $s$ -parameter increases.

## 6.5 Profiling Results After “Most Promising” Optimizations

After most of the remaining “promising” optimizations were done, code was delivered. In addition to the optimizations profiled above, which dealt with preparation of expectation values for the bins, we now include optimizations of `findSoleDifference` (discussed above), of the `chi2LLR` calculation (section 4.5), and elimination of the overhead associated with use of limits on Minuit variables (the costs of the `sin` function – see section 4.6). The following profile still does not include the effects of improved fit starting points (section 4.7) which would not, in any case, be reflected in a “per `chiFun`” number. And extending partial caching to the two-change case (section 4.8) was not done.

The net speedup is by a factor of 2.19 – 4.722  $\mu\text{sec}$  per `chiFun` instead of the initial 10.322  $\mu\text{sec}$  per `chiFun`.

The pre-optimization time (per chiFun) taken for calculating chi2LLR and for the sin function is about 42% of the total left after the first round of optimizations. (Thus, our theoretical best speedup would be by an overall factor of 1.7 relative to that first round result. The actual relative speedup factor is a bit more than 1.2.)

The most glaring optimization opportunity is that of pre-computing the sigLLR cut information when sigFit is false, so that instead of taking a log for each bin, the code need only compare the parent value to the background times some pre-computed number. As mentioned earlier, this was expected to slash about 25% of the time of this function, and in fact it did. The remaining efficiency optimizations caused no measurable change in performance, and since they significantly obfuscated the code, I rolled them back out.

One thing that did not impact efficiency in our case, but would tremendously affect efficiency if the array of `m_delBins` were fairly long, is correcting the inefficient and incorrect loop described in section 7.1. This has no impact in the example program because evidently no bins are ever deleted.

The sin optimization turned out to be almost trivial to implement and had the expected effect of wiping out that contribution to the time per chiFun.

bEQbase	8.6%	per d*s*f*b	2.0 nsec	= 0.406
abeBase	2.2%	per d*s*f	10.3 nsec	= 0.103
bEQpert	5.7%	per d*b	4.5 nsec	= 0.269
abePert	3.7%	per d*s*b	0.5 nsec	= 0.175
pExProd	3.5%	per d*s*f*b	0.6 nsec	= 0.165
fArrays	3.1%	per d*b	0.2 nsec	= 0.146
fsoleD	1.6%	per s	13.0 nsec	= 0.078
chi2LLR	34.4%	per b	80.1 nsec	= 1.622
chiFun	2.5%			= 0.117
minuit	11.2%	per s	88.0 nsec	= 0.528
other	23.2%			= 1.096

Here, sin disappears, and fsoleD absorbs all of what used to be itself and areDiff.

The formula, in nsec per chiFun, is

$$T = 2.6dsfb + 0.5dsb + 10.3dsf + 4.7db + 101.0s + 80.1b + 1313 \quad (4)$$

Compare this to the initial formula:

$$T = 23.9dsfb + 26.8db + 142.4s + 97.5b + 1126 \quad (5)$$

For the “typical” case (d=6, s=10, f=0.6, b=50), the optimized time would become about 14  $\mu$ sec per chiFun, a speedup of a factor of 4.1. Notice that while the impact of the first set of optimizations is expected to be more pronounced as the numbers of s-parameters and bins increase, the impact of this second round of optimizations on larger problems is more modest - roughly a fixed factor.

## 6.6 Non-performance Perturbations

Some changes done to speed up performance yielded, or potentially could yield, changes in results. Although we were careful to stay away from changes that could

affect the physics, it is worth noting all such differences, so that the D0 physicists can make the call on whether to say that some difference might be a problem.

The first change comes from repairing the lack of variety in random seed values (see section 7.5). Obviously, this changes the results for each PE, and thus can change the overall results. Equally obviously, the extent to which this changes any overall measurement or result is bounded by the existing uncertainty on that result.

An interesting observation was made with the number of PE's per Ridder iteration pinned at 1000 (which is, of course, laughably low). We accidentally saw the effect of using a different set of random seeds (stemming from a cleanup eliminating the Mtwist engine that had been created to randomize the fortran lun; this was no longer being done, but the engine was still being created and seeded). The effect changed the number of PE's generated from 38000 to 63000 (note that our timing figures are per chiFun, thus still valid), demonstrating how unstable Ridder's algorithm can be when faced with randomized sample points. But the final numbers output changed only from 2.334 (med) and 2.622 (obs) to 2.348 and 2.637 respectively.

Another change comes from the rearrangement of the s-parameter and bin loops. This causes different round-off to occur, and because Minuit magnifies this round-off effect when numerically estimating derivatives, it makes a small difference in the values of points tried when finding the minimum. Again, if the round-off particulars were to matter in the end physics, something would be fundamentally wrong in the first place. At any rate, the net effect of this change is undetectable in the program results.

Another potential change comes from the way we are handling positivity cutoffs on bin efficiencies. We don't even have to worry about saying that if the particulars of this non-physics cutoff matter, something is wrong in the first place – because this change has no detectable effect on our results or any intermediates.

Finally, the switch to non-limit Minuit parameters (see section 4.6) can perturb details of the minimization process. As expected, the details of points requested by Minuit change slightly. Also as expected, the ultimate results are unaffected (to at least the 4 digits of precision output).

## 7 Other Suggested Changes

The following observations are not part of the primary deliverable of this review, but were thought to warrant some interest for the person responsible for Collie.

### 7.1 Erroneous Code

Although the purpose of this review was not to uncover incorrect coding (since the program does work properly in practice), the following palpable mistake was discovered in the course of learning what the code does. Although the uses thus far by actual D0 applications do not encounter adverse consequences of these mistake (or if they do, the incorrect results have not been noticed) it should be corrected.

One deliverable of this review is modified version of the relevant files, with these flaws corrected.

### **Improper ProfileLH initialization**

Improper use of default constructor to initialize data members.

```
///Constructor with SBD initializer
ProfileLH::ProfileLH(SigBkgdDist* asbd){
    ProfileLH::ProfileLH();
    // ...
```

Calling the constructor the way it is done in this conversion constructor from `SigBkgdDist*` merely creates a temporary unnamed additional `ProfileLH` object on the stack; that object goes away upon completion of the statement in which it was constructed. None of the initialization applies to the `ProfileLH` actually being constructed!

The fix is to perform the necessary initialization explicitly. There are tricks to avoid duplication of code for this (and the coming standard will provide a clean way to do it), but for now I will just fix it the simple and mechanical way.

By the way, as a matter of good programming practice, a single-argument constructor which can inadvertently effect an automatic conversion (thus my calling this a “conversion constructor”) should be declared `explicit` unless the intent was to supply a conversion constructor (which it clearly was not in this case). I will make this change as well.

### **Invalid behavior when skipping deleted bin**

In `calculateChi2LLR`, when a bin is skipped because `m_delbins[jdel] == i`, the coder took a shortcut of simply incrementing `i` to skip that bin. The code looks as follows:

```
for(jdel=0; jdel<db; jdel++) if(i==m_delBins[jdel]){ i++; break; }
```

Two incorrect consequences accrue:

The first, and most disastrous, is that the test for loop termination does not occur. So if the last bin is deleted, the code will in fact use the number past the end of `m_bkgd` and similarly for the signal, giving meaningless contributions to the total LLR.

The second adverse consequence is that the tests for deleted bins and for skipping contributions to bins with very low parent background are not applied to the next bin. This might give unintended results.

Since `calculateChi2LLR` is an area for serious optimization and will be altered anyway, this error will be corrected in the delivered code.

### Incorrect build behavior

I have observed that when a header file changes, the need to compile the corresponding .cc file is often missed by the gmake. For example, I changed CrossSectionLimit.hh to get some verbosity, and gmake saw no need to recompile anything; I had to touch CrossSectionLimit.cc.

I don't propose to correct the Makefile myself unless requested, but this is a bit of an annoyance.

## 7.2 Erroneous Interface with Root

The nature of Root I/O is such that you need to declare, in the header, the size of arrays, even if they are expressed as simply `double*`. The way to do that is to add a comment like `//[fNsyst]` where you specify the class variable that tells the array size. Or, if you have variables for which you don't care about Root I/O handling, as in the comment in `CollieDistribution.hh`

```
///non-ROOT tools for fast linearized systematic calculations...
Int_t fNsyst;
int* fSystIndexOuter;  //!
```

you can add a comment of the form `//!` as shown.

I have modified `CollieDistribution.hh` in this way, and now the messages like

```
Error: *** Datamember CollieDistribution::fLinBins: no size indication!
Error: *** Datamember CollieDistribution::fLinBins: pointer to fundamental type
```

no longer are issued when gmake is done.

It is suggested by a strong Root expert that it is best to migrate to use the newer Root IO interface which has better support for STL containers and is much more efficient. This should be done by creating a `linkdef.h` file containing lines like

```
#pragma link C++ class CollieDistribution+;
```

However, I don't think this issue is material in terms of performance of the actual big computations.

## 7.3 Erroneous Accumulation of Parent Expectations

In `fillArrays()`, `m_signalParent` and `m_bkgdParent` are accumulated in the main loop, but are never cleared. So as Minuit tries more and more points to do the fit, the expectations in these arrays get larger and larger.

In the key activity of fitting for the s-parameters, these arrays are unused (thus the program as actually used was not haywire). However, I will fix this flaw in case other paths rely on proper values here.

## 7.4 Memory Leaks

Although we were not looking for memory leaks *per se*, Walter Brown and I noticed a few cases of new-ing variables which are not later deleted. These likely are slow enough leaks that they don't make a performance difference (nor do they crash the program by exhausting memory) but they are worth repairing when we see them.

Use of `std::vector` where appropriate would have avoided most or all of these.

## 7.5 Possibly Unintended Physics Behavior

### Strange low-end cutoff behavior

The way the code is currently written, bin expectations that come out at or below zero are reset to  $10^{-6}$ . This makes sense, but introduces the following glitch: If the expectation comes out to some positive number less than  $10^{-6}$ , it is left intact. Thus s-values that should produce a less severe likelihood penalty (if there is any content in the bin) produce a more severe penalty. This probably has little impact on the physics results, but if Minuit ever probes that region the discontinuity and quirky behavior could well cause more iterations than are necessary. The fix for this issue is trivial: When truncating at  $10^{-6}$  (or  $10^{-5}$  in the case of certain partial likelihoods, for which a similar issue arise), instead of testing for positivity, test for being greater than the truncation value.

However, when timing code where this change was done in the critical loop, I have found that the technique originally used is about 5% faster than doing `std::max(fval, 1e-5)`. The reason appears to be that comparison against zero is a bit faster than comparison against a constant double. I will be doing the "strange" truncation, therefore.

I would suggest that physics thought be applied here; it is easy to change to the less quirky approach, but at the cost of a few percent in performance.

### Strange sigLLR cutoff for zero bins

If the background in the parent distribution is zero for a bin, then this bin is skipped if and only if the sigLLR cutoff happens to be less than 10. given that the default value appears to be a million, and that only numbers less than 10 make much sense, this behavior is a bit unsettling. It is almost certainly not what was intended.

### Possible incorrect behavior regarding sigLLR

If a bin has high enough ratio of signal to background in the parent distributions, then it will be skipped (driven by sigLLR). However, it will only be skipped when just the background is being fit. When the signal is included in the fit, this bin will not be skipped. This means that the set of bins used to evaluate the fit to signal+background is different from that used to evaluate the fit to background only.

Since the computation of likelihood ratios does the same thing for the data fitted to background and the PE pseudo-data fitted to background, the code is still comparing “apples to apples,” and is not biased. It is also possible that it was intended that when fitting the background, you don’t want to distort things by having a huge penalty for data in a bin that the model says should be nearly empty, while you do want to keep the measure of how well the data fits when the signal models says it should not be empty. So I will work on the assumption that this is in fact the intended behavior.

### Usestat might not have sensible behavior

The `m_usestat` member of `SigBkgdDist` appears not to ever be enabled, at least in the `CLfit2` program I looked at. In the way it is currently used, I believe enabling it would be dreadfully costly and might not make physics sense anyway.

`m_usestat` enables the use of `BinStatError`. The idea appears to be that for each bin, along with the Poisson fluctuations from the expected bin content used when forming the data, there is to be a Gaussian fluctuation in the expectation value itself. This is in addition to the induced fluctuations in the expected value stemming from the Gaussian random fluctuations in the values of s-parameters used to produce the expectations.

In my not-so-humble opinion, the way to introduce such a fluctuation is to adjust the central expected distribution when it is first created in the PE. What is currently coded for the case where `useStat` is true applies the Gaussian random fluctuation to each expectation every time the `chi2LLR` function is evaluated.

There are two significant problems with doing what is currently coded: As an in-principle matter, what does it even *mean* to change the bin expectations for a given set of s-parameters during the process of fitting the systematics? Remember, the Minuit fitting is not itself a model of some physics, only the results of the fit are meaningful.

The second problem is practical, and if the magnitude of the Gaussian fluctuation is of any appreciable size, is extremely costly. The issue is that most of the Minuit calls to the function being minimized are done for the purpose of computing derivatives. Some trial point is evaluated, and then Minuit asks for the function at some carefully chosen small distance from the trial point. The first and second derivatives are built up by differences in these results. The currently coded fluctuation would defeat this utterly, by introducing random differences which, as the algorithm begins to converge and step sizes get small, will inevitably swamp the tiny differences in function value.

The minimization will end when Minuit “gives up,” which if you are very lucky will be when the change in function due to the step used in computing derivatives becomes comparable to the Gaussian fluctuations. In all likelihood, turning this feature on would greatly increase the number of fitting iterations, and thus the total time consumed.

### Lack of variety in seeds for random engine

Each time a random engine is instantiated, the engine is based on a random integer obtained by taking `getTimeOfDay` and taking the result mod 456. This may (I am speculating) be an artifact of some mis-understanding about how the CLHEP `MTwistEngine` is seeded. Thus there are a total of about 456 different sequences available.

This would be a complete disaster if each PE instantiated a different engine. It would mean that the LLR curves would have a maximum granularity of 456 levels, and imply that any ensemble using more than 456 PE's would be an exercise in futility. Fortunately, this is not the case.

Each mass combination does instantiate four engines. Since the seed is based on the microseconds field supplied by `getTimeOfDay`, one can hope that these four are seeded differently (though in today's hardware, a microsecond can be a long time). The engines are then used for all the thousands of PE's, so in that sense the random generation is fine, and each LLR curve has the expected granularity.

However, the restriction to 456 possible seed values means that of the many mass combinations done, some will share identical random sequences. (The example program does about 49 combinations; real applications will do many more.) This may not be a disaster but is assumedly not the intended behavior.

The fix for this, assuming that a couple of billion distinct sequences will be enough, is easy. I propose to substitute for the code sequence (which occurs several times in `SigBkgdDist` and once in `CLfit2` and each of the other CL routines such as `CLfit`)

```
timeval a;
gettimeofday(&a,NULL);
m_randgaus = new RandGauss(new MTwistEngine(a.tv_usec%456+7),0,1);
```

the similar sequence

```
m_randgaus = new RandGauss(new MTwistEngine(timeBasedSeed()),0,1);
```

where `timeBasedSeed()` utilizes both the microseconds and seconds delivered by `gettimeofday`, to form a seed integer with a greater range of possible values and which inherently avoids sequential repetition for two requested seeds.

The fix in `CLfit2` is very similar, except there it seems `gettimeofday` is called once and the result used to form two related seeds; each case is changed to use `timeBasedSeed()`.

## 7.6 Poor Style Impacting Performance

The coder does not use initializer lists in his constructors, preferring instead to set the data members inside the constructor body. This introduces double-initialization of each data member, which is some performance hit. The compiler can sometimes catch this and optimize it away, but particularly in constructors with large bodies,

it usually won't. This issue is easy to correct in any given case, and when correcting the actually erroneous constructor for `SigBkgdDist` I will also move the initialization to where it belongs. Frankly, however, the cost associated with this problem is tiny (since the coder was careful not to construct objects inside the key loops), so it is not worth serious pursuit.

There are the usual stylistic faux-pas occurring whenever a talented Fortran-comfortable physicist programs in less-familiar C++. These include the use of post-increments where pre-increments would be faster, awkward constructs in some `for` loops, and use of allocated data arrays where `std::vector` might actually be more performant. In *Collie*, none of these appear to be of any performance importance.

## 7.7 Bad Style That Could Be Corrected

The following areas were discovered in which really bad violations of the usual and expected object semantics appear. Since in the cases described here the flaws do not impact performance, and do not cause incorrect behavior in the current program, there is no urgency about correcting these issues. However, these deviations from usual expectations would make it more difficult for future maintainers to enhance, correct, or further optimize the code.

The copy constructor for `SigBkgdDist` creates a not-exactly-equivalent object. It varies in two important ways: Firstly, the copy ctor does a whole bunch of setup (including adding all the distributions) that at least one other ctor never does, leading to the fact that you can use a copied `sbd` for some purpose but if you substituted the `sbd` you took a copy of, that would fail mysteriously. Secondly, you assign a new random engine using a new seed based on time of day. Giving copy ctor's non-copy semantics is a misleading coding practice. (The latter actually burned me when I was trying to get reproducible results, but not badly because I "knew" the only source of irreproducibility must be calls to `gettimeofday`.) The fix for this particular case of copy-plus-more semantics is probably to factor out the remaining setup work; this review is not going to do that.

It is considered poor practice to introduce a `using` declaration in a header file. For example, `ProfileLH.hh` does `using namespace std`; while this is perhaps the most innocuous introduction of a namespace, it is still a bad habit to be in. (`using` declarations in the corresponding `.cc` file are fine.)

The `SigBkndDist` class tries to do an inappropriately large portfolio of duties. This probably led to a few actual inefficiencies, where the entire class is prepared just to do some small subset of what it can do; these are indirectly discussed above. However, it does not look to be productive to do the massive overhaul it would take to make this and other classes more comfortable from a "what is my role" viewpoint.

I'm not certain how wise or unwise it is to have a class template fully enclosed in the `.cc` file for another class, as `Seeker1d` is relative of `CrossSectionLimit`. At any rate, it is more usual to place it into a header; if there were problems caused by this peculiar structure, they would have something to do with dynamic loading. I don't know if this is a trap for future maintainers or not.

In ProfileLH.cc, the code fragment

```
if(pf_params!=NULL) delete [] pf_params; pf_params = NULL;
pf_params = new double[m_systNames.size()];
```

has a minor inefficiency (the if is superfluous and the set to 0 is not needed) and a genuine C++ mistake: You don't want to use the C macro NULL, you would rather use 0, which gets correct typing.

## 7.8 Clean-up Opportunities

This review did not by intent go through the code looking for places to clean up obvious things such as unused code—except where it might have affected performance. But I list here those opportunities that were noticed while looking at the code for optimization. These have not been cleaned up, but somebody could do so.

In CLfit2, the MTwistEngine `twist` is apparently constructed but never used. It was in the past evidently used to randomize the logical output unit `lun_`.

In SigBkgdDist, the distinction between `n_bins` and `n_truebins` is too subtle for me to follow. I believe there is no place where `n_truebins` is set to something other than `n_bins`, but there are cases where `fillArrays` is reached with `varySyst` true, where `n_truebins` is zero. Knowledge of what `n_truebins` was meant to represent would go a long way toward rationalizing this situation.

In CollieDistribution.cc, the copy ctor and copy assignment operator do not (at least manifestly) obey the expected C++ semantics of yielding an identical object. Other than adding code to properly copy any new data members added for the optimization, we are not touching these, because of the risk that the program subtly depends on the non-standard behavior. However, it is strongly suggested that if the copy is an “almost copy”, somebody should do what it takes to make it a true copy and ensure that the code is not relying on hinky side-effects or modifications magically appearing in the copy.

## 8 Summary/Results of the Review

### 8.1 Modified Code Provided

A list of modified and new files is presented in section A.1 of the Coding Steps Appendix. The key changes are:

The CollieDistribution class, which previously contained a function `getNormalizedBinValueVaried()` which provided the value for a single bin (based on the changes in each of the s-parameters, the efficiency for this distribution, and the basic unvaried value for the bin) now has in addition a couple of key variants on this theme. The first is `addBinEfficiencies`, which takes the fluctuation map and a pointer to the destination signal or background profile, and accumulates into that profile the changes for *all* the bins. The second is a different signature for `addBinEfficiencies`, taking – instead of the `fluctMap` – the identity and new value of only one

s-parameter. This routine uses previous stored “exclusion products” to compute how much to accumulate for a bin, based on only this s-parameter and the stored result for all the other s-parameters, which are assumed to be unchanged. This is, when valid, a much faster substitute for the other form of `addBinEfficiencies`.

We have factored out the calculation of the partial efficiency for a single bin, (into routines like `binEfficienciesQbridge`), and re-structured the code such that all manipulations which depend on the value of the s-parameter but are not bin-dependent are pulled out of the loop over bins. Because of this re-structuring, the cost of a more careful treatment of the asymmetric sensitivities is amortized over all the bins. For example, our timing tests use a quadratic bridge with cutoff to the linear sensitivity at  $|s| > 1$ , and this does not cost noticeable time compared to the simpler (but physically problematical) pure quadratic bridge. Further modifications to use a sigmoid transition from the positive to the negative realm are now easy to incorporate, at very little performance cost.

The `SigBkgdDist` class is the other end of the major optimization path for computing the expected “model” profiles. Here, the major change is in `fillArrays()`, particularly in the paths taken when `varysyst && m_sigFluct` is true (which is the code executed when filling the models for each `chiFun` call in all the pseudo-experiments. Here, we detect (using a new function in `findSoleDifference.hh`) the case of only one s-parameter changing (relative to a saved “base” point), and call one of two new methods, which set either new or perturbed expectations, relying on the new varieties of `addBinEfficiencies` for each distribution.

A second type of change is also present in `SigBkgdDist`, namely, an optimization of the `sigLLR` cut computation in `calculateChi2LLR`.

`ProfileLH` has a few minor changes to align with the way `SigBkgdDist` expects its parameters passed, and a performance tweak affecting Minuit’s internal treatment of variables.

Various classes are modified to utilize the new function `timeBasedSeed`, repairing a couple of potential headaches in the way random engines were seeded from `getTimeOfDay`.

Finally, cleanups were done in a couple of header files and in `ProfileLH.cc`, to eliminate all the compiler fictitious error messages and warnings about code not used.

The developer of the code can enable certain changes which help to better performance-measurements (for example, by tracking and outputting the number of `chiFuns` and the number of fits). This is most easily done by adding `-DSTUDY_TIMING` to the `DEPENDFLAGS` line in the `Makefile`.

## 8.2 Performance Differential Measurements

As discussed in detail in section 6.2, looking exclusively at the example program (but using `CLfit2`) we have achieved a net speedup by a factor of 2.19 in the time per evaluation of `chiFun`.

This was the result of a series of optimization steps:

- The original code took about  $10.32 \mu\text{sec}$  per `chiFun`.

- Re-arranging the innermost s-parameter and bin loops when computing expected profiles cut the time taken by 29%, to 7.38  $\mu\text{sec}$ .
- Eliminating parent distribution computation in places where it will not be used shaved off a further 0.35  $\mu\text{sec}$ .
- Storing quantities which would later be used by partial likelihood caching gave back that 0.35  $\mu\text{sec}$ .
- Implementing partial likelihood caching cut the time to 5.74  $\mu\text{sec}$ .
- A small tuning of the method of discovering opportunities for using the cache shaved off another 2%.
- An improvement in the calculation of chi2LLR after the bin expectations have been computed reduced the time to 5.09  $\mu\text{sec}$ .
- Eliminating the use of bounded parameters for Minuit reduced the time to 4.72  $\mu\text{sec}$  per chiFun.

### 8.3 Expectations in More Realistic Running

In section 6.2, the times for individual parts of the program are presented. Sensible assumptions about the scaling behavior (with number of bins, number of s-parameters, number of distributions, etc.) of the time needed for each activity help translate these measured numbers into a formula for time taken, which yields the wall-clock measured time for the test example. We thus have before (optimization) and after formulas.

We can then plug the approximate characteristics of more typical physics runs into this formula, getting a good estimate of the time the original code would take, and the time the new code will take. The ratio does not remain constant; because a major part of the optimization saves chunks of time which grow faster than linearly in the number of s-parameters, we expect the optimized code to show a larger improvement factor for the larger realistic problems.

Extrapolation is not an exact process, but we confidently expect that the optimized code will run at least 3.5 times faster (our best estimate is 4.1) than the original on typical 10-parameter, 50-bin problems.

### 8.4 Tricks That Did Not Help

Aside from a pre-computation of a quantity allowing for applying the sigLLR cut to be applied without needing to take a logarithm for each bin, all the other efficiency cleanups of the calculateChi2LLR method yielded immeasurably slight improvements, and were rolled back out in favor of cleaner code.

Other than that, pretty much everything that looked promising panned out.

### 8.5 Further Opportunities For Tuning

With one exception, I believe we have reached the point of diminishing returns on optimizations which reduce the time taken per chiFun.

The exception is implementation of the 2-change caching described in section 4.8, or elimination of the routine per-PE requests for error matrices if those are unused and are causing the sequence of such trial points. This optimization would gain little in the test example (where a typical distribution only has 3 active s-parameters, and needs to deal with 2 of them even in the “fast” case), but might be worthwhile for larger problems.

Further optimization would need to focus on reducing the number of chiFun calls needed. Several avenues of work could do this, if deemed necessary; none have any certainty of payoff:

1. The effect of analytic derivatives could be tried. This will not be easy, particularly because of the complicated set of cutoffs and interpolations currently in use, but with some physics thought that can be simplified and handled. The potential for gain is greater the more systematics parameters are being varied. However, the way Minuit works with analytic derivatives will be to obviate the need for as many function evaluations per trial point. This might negate most of the gains we have achieved by partial likelihood caching.

There is probably the opportunity for some net gain, but it might be small. It would take a fair amount of work (including some physics thought) to implement this optimization, study the new trial point pattern, and optimize the equivalent of partial likelihood caching for that pattern.

2. Some sort of improved starting points might be worth our while. For example, the cross-informed starting points (described in section 4.7 use the best-fit points for one model in a PE as the starting guess for the other. This is a sound possible analysis technique and does not introduce bias in the context of Collie. If this eliminates, at average, one of the roughly five Minuit trial points per fit, it will show a ten percent improvement (since it is applicable half the time). I don’t think the likely improvement will be that great, however.

A more important change, and one which is very easy to do, would be to use the center of the *a priori* s-parameter distributions, rather than zero, as the starting point for Minuit fitting. Although the current example program always uses s-parameter distributions centered on zero, so that this is moot here, if that is not always the case, the code should be adjusted to use the actual center.

3. I believe it is possible to automate the search for the signal strength yielding a given confidence limit, in a way which is much more efficient (in number of PEs needed) than the current Ridder’s algorithm (see section D). Although a thorough analysis of the situation to produce a near-optimal scheme for deciding what scalings to try would require significant statistics thought, a reasonable scheme that significantly improves on the current method should not be hard to develop.

It has been stated that for actual production, this search is done by hand rather than in an automated way; I think this is a consequence of the fact that the current algorithm is not very efficient (nor very stable), and one can reach the obviously intended goal of automation without too much effort.

## 8.6 Issues the Developer Should Consider

The suite of validation and test programs was very limited, so there is some chance that changes which did not adversely effect the example program will nonetheless prove problematic when Collie is used in other contexts. The person of principle responsibility for the use of Collie ought to pay attention to checking the following issues, in particular:

1. There are two approaches to walking through all the bins. In several places, `SigBkgdDist` explicitly considers  $N_y$  and  $N_x$  separately; yet by the time `calculateChi2LLR` is reached, it is all folded into one array of bins. The optimizations rely on this being valid under all cases (at least during fits). It certainly is valid for all work in the example program. Somebody should check that this assumption is not incorrect under some other circumstances.
2. I have done only one mathematical transformation; it appears in the new version of `calculateChi2LLR()` routine, where the `sigLLR` cut is applied when `fitSig` is false. I would appreciate if somebody glance at this and verify that it is mathematically equivalent to the original code.

I have introduced some replication of data, in that the sensitivity factors `sigmaP` and `sigmaN` are stored both as raw-pointer-based arrays as originally, and as members of a newly-introduced structure containing info for partial likelihood caching. The old arrays are still being used elsewhere in the code. This could be unified, though I doubt that one would see any performance improvement.

Also, I did not do code cleanups except where they directly intersected code relevant for optimizations. If there is a “long run” in this case, it may in the long run pay for somebody to do some of these cleanups. However, the code is not in terrible shape as it stands.

Finally, I will probably be asked by management to make this document available in the CD DocDB. However, it contains information which logically belongs to D0, not to myself; I have not tried to hide any information about how D0 is doing their analysis, nor have I filtered any discussion or criticism of physics or statistics methodology or coding technique. (To the extent that little such criticism appears in this document, that is a reflection of the fact that Collie was IMHO a pretty well-conceived and implemented product to begin with.) The experiment has a right to decide how much of their detailed procedures they wish to disclose publicly. I ask that one of my D0 contact points for this project let me know whether to put this in DocDB (and if so, whether to make its access permissions D0 only, D0 and CD, or public), or perhaps they can submit it as a D0 note, or make it available in some other way.

## A Coding Steps

### A.1 Files Affected

The following files were either modified or added to do the optimizations and cleanups:

In io:

- CollieDistribution.cc and CollieDistribution.hh (*extensive changes*)

In limit:

- SigBkgdDist.cc and SigBkgdDist.hh (*extensive changes*)
- findSoleDifference.hh (*new file - do not forget to cvs add this*)
- ProfileLH.cc and ProfileLH.hh (*Use of vector for pf\_params, fix of ctor, other changes*)
- CLfit2.cc (*Utilize timeBasedseed, and tracking mods left available*)
- CrossSectionCalc.cc (*Utilize timeBasedseed, and other minor changes*)
- CLfit.cc, CLsyst.cc, CLfast.cc, FitTest.cc (*Utilize timeBasedseed*)
- CrossSectionLimit.cc (*Minor performance tracking mods left available*)
- timeBasedSeed.cc and .hh (*new files - do not forget to cvs add these*)
- PerformanceStats.hh (*new file; not needed unless performance tracking enabled*)

In examples:

- ExampleLimitCalculation.cc (*Activated CLfit2 instead of CLfast; this is undone in delivered code*)

In top (collie) directory:

- Makefile

### A.2 Setup for meaningful exploration of performance

#### Use of CLfit2 instead of CLfast

*Temporary:* In exampleLimitCalculation, commented out

```
CLfast clcompute;
```

and dis-commented (to make active)

```
CLfit2 clcompute;
```

to explore using CLfit2, which is more representative of the work done in the time-consuming jobs to be optimized.

### Control of number of PE's

*Temporary:* In CLfit2, added a line at start of CLfit2::calculateCLs

```
return doCLs(sbd,CLs,100);
```

where the number 100 was, in the course of studies, varied to give reasonable (not too long) numbers of PE's and to get a handle on the effect of all the non-PE work.

Also,, replaced the controlling loop in doCLs with a simpler one:

```
for (nmc=0; nmc<2*its && (denom<its/2.0 || nmc<its); nmc++) {
```

becomes

```
for (nmc=0; nmc<2*its; nmc++) {
```

### Obtaining reproducible runs

*Permanent problem-fix:*

Everywhere `gettimeofday` is used to seed a random engine, substituted the fix described in section 7.5 to avoid the severely restricted set of random seeds. The fix involved changes in 3 places in SigBkgdDist and one each in CLfit2, CLfit, CLfast, and CLsyst. Also, I changed CrossSectionCalc and FitTest (which uses `gettimeofday` in one place to seed an engine), although their usage allows for a million (instead of 456) possible seeds.

This fix is included here because I will enabled a special define in `timeBasedSeed.cc` described below.

Added the `timeBasedSeed` method to the limit directory. This will be checked in.

In the Makefile, added

```
LIMITSRCS += ${LIM_SRC}/timeBasedSeed.cc
```

and

```
LIMITHEADERS += ${LIM_INC}/timeBasedSeed.hh
```

Now to allow for reproducibility:

In `timeBasedSeed.cc` there is a special `ifdef` on the symbol `FORCE_REPRODUCIBLE_SEQUENCES`. When this is defined, the program will deliver seeds which are reproducible numbers, by simply returning a linear progression. (I know enough about MTWistEngine to know this sort of seeding will still yield excellent inter-engine randomness.)

*Temporary:* I enabled the reproducible sequences. But I wanted to do this in a manner which would preclude any possibility of accidentally propagating this behavior into production code. To do this, I placed at the start of `timeBasedSeed.cc` an include of `force_reproducible.hh`, which in turn defines `FORCE_REPRODUCIBLE_SEQUENCES`. But `force_reproducible.hh` will not be mentioned in the Makefile, and will never be checked in. So if I were to forget to remove the include of that header, Collie would fail to build rather than build with the reproducible seeding.

## Collection of execution data

To explore performance, it is useful to be able to collect cumulative numbers such as the count of calls to `chiFun` and the number of fits done. Since we may want information generated by many different classes, and since we don't want the collecting to unduly influence performance, what was done is to create a `PerformanceStats` struct, which any `.cc` file can include.

The `PerformanceStats` struct will evolve if different information is to be gathered. All members of `PerformanceStats` are static.

That class has a (static) `print` method. The issue of when collection is done and when results are printed out is up to the individual investigation being pursued.

As a matter of not wanting to affect the normal execution, I surround all use of this `PerformanceStats` with an `ifdef` of `PERFORMANCE_STATS_TRACKED`.

## A.3 Partial Likelihood Caching

This step implements the partial likelihood caching described in section 4.1. While I was doing that restructuring, I also included the improved efficiency for interpolating functions described in section 4.3.

### Figuring out when caching can be exploited

The first issue is how to decide whether the trial point supplied by Minuit is a single perturbation off some "base" point, or a new base point. If it is the former, the expectations can be computed more efficiently by exploiting saved partial results for the unchanged s-parameters.

The s-parameters are passed into `fillArrays()` as `fluctMap`. This is the array known in ProfileLH as `pf_params`. It was immediately realized that the new code would need, at the `SigBkgdDist` level, to work with the s-parameters embodied in the array `pf_params` in a clean manner; in particular, the size of the array (which is known by ProfileLH but not so easy to get in `SigBkgdDist`) is important. So I changed the type of `pf_params` to `vector`. This required adjusting the initialization in the ctor and in `fillSyst`. The most common usage of `pf_params` is places where the code references an element; this retains the identical syntax so no changes were needed in those places. Other uses of `pf_params` had to be changed.

The change of type for `pf_params`, while cleaning up a fair bit of code, had implications for other classes and functions. I wanted to contain the set of changes to as small a set as possible. This implied changing `getFitParams()` to return a reference to `pf_params`, and changing `gl_systRand` to a `vector`, to match `pf_params` because they are both passed to the same functions. Finally, I needed to modify some of the "non-common" uses of `gl_systRand` to reflect this new type.

Now that we have `fluctMap` in a tamer form, determining whether it is closely related to some cached value is easy. We wrote a class `TrialPoint` to deal with remembering the last Minuit base point and to discern whether only one s-parameter has changed relative to that. A function `findSoleDifference` assists. One subtlety in `findSoleDifference`: We noted that in some of the explorations about a

trial point, Minuit supplies new points which agree for all but one of the parameters, but for which the matching parameters are “off” in the last bit or two. We surmise this is a consequence of conversion to the internal variables (via the arcsin function), with different conversion sequences leading to presentation of different Minuit “external” variables. `findSoleDifference` deals with this by accepting differences smaller than one part in  $10^{-14}$  as being irrelevant. `TrialPoint.hh` and `findSoleDifference.hh` will be checked in to cvs.

To avoid misusing a last trial point from a previous fit, `fitProfile()` resets its new `TrialPoint` data member `m_lastMinuitTrialPoint` just before calling the `MINI` method of Minuit.

Finally, `chiFun()` is adjusted to pass `m_lastMinuitTrialPoint` to `sbd's fluctuate()`. In turn, adjust `fluctuate()` to pass that along to `fillArrays()`. Since others call `fillArrays` as well, the default is a value which indicates no valid remembered point.

### Structure for Cached Likelihoods

In `CollieDistribution` we create `InfoForEfficiencyCalculation` to hold both the asymmetric sensitivities and the cached partial likelihoods. So we have a vector (on `s-values`) of vectors (on bins) of these structures. This vector, `fLinSyst` replaces the two `double**`'s `fLinSystPos` and `fLinSystNegs` (and does a bit more).

Note that with naive (but correct C++) usage, one will have trouble with a `Root` streamer trying to do something with this vector of vectors, leading to a crash. The fix is via the use of a `//!` comment, as indicated in section 7.2.

The use of `fLinSystPos` and `fLinSystNegs` is modified to use `InfoForEfficiencyCalculation` in several places: In `getNormalizedBinValueVaried()` (three places); in `getBinSystValue()`; in `linearize()` (both setting up the memory and establishing the values in 1D and 2D cases).

### Repairing erroneous parent accumulation

In the case where `m_signalParent` and `m_bkgdParent` are used, the code as originally provided sets them incorrectly (see section 7.3). This is repaired by zero-ing these arrays before the respective loops that accumulate them.

However, in the end, the critical loops (that is, where `varySyst && m_sigFluct` is true) don't need to deal with the parent distributions at all. (This is verified by noting identical results and intermediate results for every value returned by `chiFun()` when the code dealing with these parents was excised from the loop done when `varySyst && m_sigFluct` is true.)

### Branching to the Cache-Enabled Code

The main work of `fillArrays()` during fitting is dominated by a pair of deep loops,, filling `m_signal` and `m_bkgd`. However, the significant work part of these loops does not occur unless `varySyst && m_sigFluct` is true. To avoid unnecessary

conditionals in the work loops, the overall control structure is provided via going to new code if `varySyst && m_sigFluct` is true, and using the old code otherwise.

The first step is to detect the opportunity to use the shortcut. A struct `TrialPoint` is created (`TrialPoint.hh`). With the aid of the function in `findSoleDifference.hh`, the `diff()` method of `TrialPoint` gives us an iterator which we call `onlyNonBaseS`. If this matches `fluctMap.end()` that indicates more than one change.

Next, we substitute two possible routines for the single big loop in `fillArrays` that fills the `m_signal` array. (But we do this only if `varySyst && m_sigFluct` is true; we keep the original code for other cases.)

```
if ( onlyNonBaseS == fluctMap.end() ) {
    setNewPointSignalExpectations (&fluctMap[0], m_signal);
} else {
    setPerturbedSignalExpectations (fluctMap, onlyNonBaseS, m_signal);
}
```

Although these are methods of `SigBkgdDist` and thus know about `m_signal`, we pass it as an argument because these same two member functions will be called to handle the second big loop, this time filling `m_bkgd`.

`setNewPointSignalExpectations()` does the same as the combination of existing code in the `fillArrays` big loop, and the code in `getEfficiencyVaried()` calling `getNormalizedBinValueVaried()` in `CollieDistribution`. However, the methods called reverse the order of the `s` and `bin` loops (doing the `bin` loop innermost so that less simple interpolation methods could be used and so that memory strides are minimized), and also save crucial partial likelihood information. The main work, of course, is done in the `CollieDistribution` class, where a new method `addBinEfficiencies` handles the `bin` and `s`-parameter loops.

`setPerturbedSignalExpectations()` will utilize that partial likelihood information to compute the `bin` expectations by dealing with only one of the `s`-values, and is thus almost  $N$  times faster, where  $N$  is the number of active `s`-parameters for a typical distribution. Since most Minuit trial points are of this single-change nature, this is a huge potential speedup.

## Computing products for the base point

To avoid repeatedly multiplying by various scale factors, `addBinEfficiencies` pre-computes those, and in fact keeps those products in a new vector data member `fEfficiencyProducts`, because they will be needed if there are other points which are perturbations off this base point.

It then loops over `s`, and since choosing between `linLogNormal` and ordinary for each individual `bin` is costly, it makes the choice once and for all by calling either `binEfficienciesQbridge` or `binEfficienciesQbridgeLinLogN`.

(This is the place where, if one wished to enhance the `CollieDistribution` by specifying which of several interpolation methods one should use, we could place a switch statement without costing too much time. And it has become practical,

now, to implement interpolations which might involve some calculation on  $s$  (such as a true sigmoid), since now this will be amortized over the many bins to be handled.)

`binEfficienciesQbridge` is coded as tightly as possible. In particular, since there are three cases depending on whether  $s$  is below, inside, or above the range  $(-1,1)$ , we write three distinct loops over bins to avoid the cost of conditionals inside the innermost loop. By the time this routine completes, `fEfficiencyProducts` has the full contribution of each bin to the signal or background. Along the way, we take the time to save the partial likelihood for each bin for this  $s$  in `info->baseEfficiency`.

Finally `addBinEfficiencies` accumulates those full products into the signal (or background) array.

One might well ask where the exclusion products, needed for the partial likelihood shortcut, have been stored. The answer is, they have not (yet)! It is real work to compute these, and just in case the next point will be a new base point, we defer that work till it is known to be needed.

### **Perturbed Signal Expectations - the Shortcut**

`setPerturbedSignalExpectations()` (and similarly for `bkgd` instead of `signal`) is called when it is recognized that this point differs from the base point by only one  $s$ -value. At this point, we check whether the exclusion products have yet been calculated, and if not, invoke `prepareSignalExclusionProducts()`. This sets up the same distribution loop structure as is used for the signals in `fillArrays()`, and invokes each distribution's `prepareExclusionProducts()` method.

That routine, in `CollieDistribution.cc`, looks at the stored vector of efficiency products for this distribution (`*fep`), and at the vector (indexed by bin) of `InfoForEfficiencyCalculation` for each active  $s$ -parameter and sets

```
info->exclusionProduct = (*fep) / info->baseEfficiency;
```

That is, we have the overall product; we have the contribution from this  $s$ , and we divide to find the product of contributions from all the others. This is costly, but of course it is done only the first time a perturbation off a given base is detected.

With the exclusion products calculated, we now loop over signal distributions, calling a version of `addBinEfficiencies` with signature including the identity of the perturbed  $s$ -parameter. This method looks up `fSystIndexInner[perturbed_s]`, and sets up to loop over the bins by loading the exclusion products into an array which will be used by the *same* `binEfficienciesQbridge()` routine employed in the base point case. Note that this is now done for just the one perturbed  $s$  – that is the primary time savings.

Finally, we add the products formed by that routine, to the signal we wish to accumulate. One subtlety: Sometimes the  $s$ -value that has changed is not active for this distribution. In that case, the work is even easier: The efficiency products for this distribution are added instead.

## A.4 Use of vectors in optimization

In the course of doing partial likelihood cache optimization, I have introduced certain `std::vector`'s in place of raw pointers used as user-managed arrays. The most important such change is that `fluctMap` became a vector – code which was passed `fluctMap` needed its size (for the optimizations) and previously had no way of getting it. This had spillover consequences; for example, `gl_systRand` is used in place of `fluctMap` at times, and also needed to become a vector.

Most usage of vectors matches, in syntax, the corresponding pointer code; for example, lines like

```
diff = gl_systRand[s] - syst[s];
```

don't care whether the objects are raw arrays or vectors. But I have not done a change from raw pointers to vectors across the board; it is not the purview of this review to make that sort of stylistic improvements. Thus, several routines which expect raw pointers are left intact.

In consequence, sometimes a routine expecting a pointer needs to be passed a pointer to a vector's data, leading to changes looking like

```
&gl_systRand[0];
```

These changes occur in `SigBkgdDist.cc` and `.hh`,

## A.5 Optimization of calculateChi2LLR

The routine `SigBkgdDist::calculateChi2LLR` was modified to implement the optimizations discussed in section 4.5. Of the five plausible optimizations, two were done: The major boon of pre-computing an exponential instead of repeatedly needing to do a `log10`, and the repair of the `delBins` potentially disastrous inefficiency.

The other optimizations (for example, pulling the conditional on `sigFit` out of the most critical loop) were coded and run, and found not to significantly affect performance (within a 1% uncertainty). Since they made the code somewhat convoluted, it was decided to restrict the changes to the ones mentioned above, and to leave most of the method intact.

This was the area originally suggested for optimization, and although the gain is much smaller than those from re-ordering loops and partial likelihood caching, it is about a 20% effect in the final version.

## A.6 Cleanup of Errors and Serious C++ Faux-Pas

In `SigBkgdDist.hh`, a `using namespace std;` declaration is removed, and corresponding changes to types used in the header itself would have been made, but there were none needed. For convenience and readability reasons, `SigBkgdDist.cc` does have such a declaration; this does not pollute the user's namespace and is accepted practice.

Also in `SigBkgdDist` is a “fix” that has *not* been made: The copy constructor gets a fresh `randgaus`, seeded by a new call to `gettimeofday` in the original code, or to `timeBasedSeed` in the modified code. Also, `setBaselineModel` is called, which in turn calls `fillArrays`. A copy ctor should have no effect other than creating an identical copy of the argument object. But the existing code depends on the extraneous side actions. Rather than unthreading that dependence (and possibly having to restructure a fair bit of code and risking breaking a working program), I have left this faux-pas untouched.

In `ProfileLH.hh`, a `using namespace std;` declaration is removed, and corresponding changes to types used in the header itself have been made.

In `ProfileLH.cc`, the improper initialization in the constructor taking a `SigBkgdDist*` has been repaired. To do this, I have replicated the behavior of the default ctor (which is clearly what the code meant to do); I have placed as much of this as possible into an initializer list. As long as this is being done, I have also eliminated the improper (but working) use of `NULL` in favor of the more proper (in this context in C++) `0`.

### Cleanup of `delBins` Error

I set out to avoid the erroneous behavior which would have occurred if any deleted bins were established at the end of the set of bins or immediately after another deleted bin (see section 7.1).

Although this can be done in the context of `m_delBins` alone, I noted that if there are several deleted bins (the example program never has any) this will cause massive inefficiency in the calculation of `chi2LLR`. To also address this matter, I have created and a vector data member of `SigBkgdDist` called `m_delBinsLookup`. This is set up in all places where `m_delBins` is changed, and is used in `calculateChi2LLR()`.

`excludeBins()` and `clearExcludedBins()` have been removed from inline because they are now less trivial.

## A.7 Moving inline Code Where Appropriate

Large chunks of complex code should not be inlined. Not only is the potential gain minuscule, but the added bulk of the code produced may actually make this a “pessimization.” The compiler often will note this and place the inline code into a subroutine anyway, but the inline directive costs in a couple of ways: It becomes easier to have conflicting versions of a routine in one executable, and it become more difficult to debug and profile the code.

I have not addressed this issue across the board, but have done so with routines which were major players in the optimization.

The following inline routines were moved to the corresponding `.cc` files:

- `SigBkgdDist::calculateChi2LL()`

## A.8 Minor cleanups

Elimination of `twist` in `CLfit2.cc`.

Declaration of `ProfileLH(SigBkgdDist* asbd)` to be `explicit`.

Cleanup of all the Root-originating fake error messages during the build, by declaring (via `/// comments) various raw pointers as being transient.`

Cleanup of all remaining compilation warnings, by creating fake usage of the `cfortran` and other externals that the build was griping about in `ProfileLH.cc`.

*Other minor cleanups not listed here.*

# B Math of Collie Calculations

## B.1 Computing Expectations Based on Systematics

The key part of calculating a likelihood is fitting for the best s-parameters, and the key part of that is computing the expectation curve for a given (supplied by `Minuit`) set of s-parameter values. The expectation curve is one value per bin (here, I use  $b$  for bin index). The issue of whether the bins are arranged in one or two dimensions is moot.

The mathematical definition of the expectation at bin  $b$  is a sum over channels in that model and distributions within each channel, of the product of efficiency factors for each of some subset of the s-parameters.

Let  $M$  be the model for which we want the expectations. The model is a set of channels indexed by  $c \in M$ . A channel  $c$  is defined by a set of distributions  $d|d \in c$ .

Not every s-parameter is relevant in the efficiency of every distribution. Let  $S(d)$  be the set of s-factors (indexed by  $s$ ) relevant to  $d$ . Then the basic efficiency for the bin is the sum over participating distributions of

$$E_b = \sum_{c \in M} \sum_{d \in c} W L_b^d \prod_{s \in S(d)} f_b^{sd} \quad (6)$$

where:  $f_b^{sd}$  is the response sensitivity function of that distribution  $d$ , in bin  $b$ , to the change in value of parameter  $s$ ;  $L_b^d$  is the linearized central bin value for distribution  $d$  in bin  $b$ , and  $W$  is `m_sigScale`, some scaling factor which is a property of the `SigBkgdDist`.

This is actually modified in several ways, and these tweaks might be very relevant when trying to restructure for optimal performance.

1. For each  $s$  in a distribution, if a preliminary `fval` used in forming  $f_b^{sd} \leq 0$  it is replaced by  $10^{-5}$  (note a critique of this in section 7.5, but the modified suggested method would have the same issue discussed here). The best way to look at this is to say that this is part of the computation of the overall  $f_b^{sd}$ .
2. For most distributions, the last step is  $f_b^{sd} = 1 + f$ , but for some,  $f_b^{sd} = e^f$ . Again, this is best viewed as part of the computation of the overall  $f_b^{sd}$ .
3. After multiplying all the  $f_b^{sd} = 1 + f$  for one distribution, if the result is non-positive, it is replaced by  $10^{-5}$ . This can't be expressed as a re-definition

of  $f_b^{sd}$ . Below, I introduce the notation  $\vartheta(f_b^{sd}, \epsilon)$  to mean this potentially changed value.

4. If  $W$  and/or  $L_b^d$  are small,  $WL_b^d \prod_{s \in S(d)} f_b^{sd}$  might for some distributions might be non-positive; in that case, it is replaced by  $10^{-6}$ . Again, the notation  $\vartheta(f_b^{sd}, \epsilon)$  is handy for this situation.
5. The set of channels in play may be diminished by one explicitly excluded channel  $M - \tilde{c}$ . The identity of the excluded channel is not changed throughout the fit, so this variation is easily handled by replacing  $M$  by  $M - \tilde{c}$ .
6. Some bins may be excluded from the fit, but the current program nonetheless computes the expectation value for those excluded bins; they just don't get processed in the chi2LLR step.
7. If `m_useStat` in the `SigBkgdDist` is set, then each term for a distribution in a bin gets an additional Gaussian noise contribution  $\eta_b^d \zeta_b^d$  where  $\eta_b^d$  is a unit Gaussian variate and  $\zeta_b^d$  is a bin statistical error scale. In that case, the function being fit is no longer a true function; see section 7.5 for a critique of this situation.

Combining all these tweaks, the equation becomes:

$$E_b = \sum_{c \in M - \tilde{c}} \sum_{d \in c} \vartheta \left[ WL_b^d \prod_{s \in S(d)} \vartheta(f_b^{sd}, \epsilon) + \eta_b^d \zeta_b^d, \epsilon \right] \quad (7)$$

## C Using the SimpleProfiler

This section was written when the optimizatoin was being done. Since then, the SimpleProfiler has evolved and the usage is somewhat more convenient. So while generalities can be followed, the particulars of this section may by now be obsolete.

After setting up for running Collie, the profiler can be used from the `collie/examples` directory as follows:

First, the Makefile needs a `-g` in the compilation options. This costs nothing in terms of speed, and provides the symbol table. For Collie, the size of the executable is not so large as to make one wish to strip out symbols. The Makefile supplied with the new version changes the necessary line:

```
DEPENDFLAGS := -g -O3 -ffast-math -Wall ${SEARCHDIRS} -fPIC -Df2cFortran
```

Note that `-O3` is fine; you are measuring the actual optimized code. Then:

```
~jbk/perf_tools/bin/RunProfiler.sh ./collieLimitCalc.exe test.root
```

This runs the job, with profiling turned on. Note that no rebuild or special insertion of profiling preparation is needed; the profiler looks at the actual execution.

The profiler will report some number of “frame pointer errors.” These are OK; they occur when the profiler ticks during a system call in which the stack frame is not of the usual type. Later versions of the profiler drastically reduce these, but as it is, they introduce only about a one part in  $10^4$  distortion of timing results.

Next, *ls -lrt* to see the name of the file produced by the profiler (which has a job id number embedded in it. It should be the latest file created, and the name will look like `prof.out.8201`).

Now choose some identifying name to give this particular study; here I will call it “chiOpt1.” Extract the key information from the profiler output by:

```
~jbk/perf_tools/bin/ProfParse prof_libs.out.8201 prof.out.8201 chiOpt1
```

This will report a bunch of addresses that it could not understand, but again, that is OK and represents a tiny loss of statistics. (Most available profilers have a much larger loss of statistics, both above and here. The difference is that SimpleProfiler was intended to keep this loss as small as possible, so it reports all the glitches so that we can deal with them. Newer versions of the profiler have yet better coverage.) This command will produce a file `chiOpt1names` and other files all starting with `chiOpt1`. Next, we demangle the names of functions appearing in `chiOpt1names`:

```
c++filt < chiOpt1names > chiOpt1names.dem
```

(This is in C-shell; Bourne shell users know how to direct input and output similarly.)

Now we have a file `chiOpt1names.dem` with numbers pertaining to fractional time spent in many functions. View this in an editor in a wide window. A couple of typical lines look like:

```
84 0x916060 SigBkgdDist::calculateChi2LLR(bool, double) const
    6122 6122 6122 0.369441 0.369441 11 4 107 80 1807 60 67 3775 211 0
83 0x936270 chiFun(int*, double*, double*, double*, int*, void (*)())
    330 11234 11234 0.0199143 0.677931 19 3 102 22 12 1 4690 6091 244 50
```

where I have broken each line into 2 so that it fits on this page.

The relevant information in that is:

Function 84 is `SigBkgdDist::calculateChi2LLR` with the calling signature `(bool, double)`. It occupied 0.369441 of the total time in the function itself, and it and its all descendants occupied 0.369441 of the total time. (Not a surprise that these are equal; this routine calls nothing else.)

Function 83 is `chiFun` with the calling signature shown. It occupied only 0.0199143 of the total time in the function itself, but `chiFun` and all its descendants occupied 0.677931 of the total time.

Finally, you can obtain a fairly nice call graph based on this information. Go over to `dipole-clued0` (the others are all missing a key shared library `libpng.so.2`) and get to `collie/examples`. Then to look at the call tree involving, say, function 83 and its antecedents and descendants up to 5 levels, do:

```

setup graphviz
~/jkb/perf_tools/bin/ProduceGraph.sh 83 5 chiOpt1
ghostview 83.ps

```

## D Finding Signal Strength for a Given CL

(Here, it was hoped to place some explanation of the existing algorithm, plus suggestions as to how the number of iterations might be dramatically improved.)

This careful thought needed for this appendix section cannot be completed on time for changes in analysis for the Spring 2009 conferences. Therefore, it will not be in this document; a future physics note may be forthcoming.

The idea is that the current algorithm effectively discards all but the last four sets of PE's. By using all the data available (with appropriate treatment and weighting), a better estimate can be reached more quickly. The gist of the analysis on this matter is the following:

Each set of pseudo-experiments based on some common signal strength scaling factor yields information equivalent to saying that at that signal strength, M PE's analyze as more signal-like than the data, and N as more background-like. At any given point in the calculation, you have in hand a collection of such sets, taken at different values of signal strength. This can be fit to an error-function sigmoid (the computation to do that is non-trivial, but minuscule compared to doing the thousands of pseudo-experiments that go into each set.) In turn, using the fitted mu and sigma, you can transform the scale values, such that if the match to that sigmoid were perfect, the data of fraction more signal-like would lie on a straight line. In reality, it will be some curve which is a small perturbation off the straight line.

The issue becomes deciding how to select a next signal scaling value, to get the best improvement in error on the point at which that curve intersects .95 (or whatever confidence limit is desired.)

If that fit were assumed to be perfect, then it would be possible to determine precisely where one would best add, say, an extra 10,000 PEs. Again, this calculation is non-trivial but affordable compared to the PE cost. But the fit won't be perfect; small non-linear terms about the point of intersection will mean that far-off (in signal scale) measurements are of less value (despite the fact that they have a larger "lever arm" in determining the intersection).

I suspect that a fairly optimal algorithm is to choose the next set of points as follows: Let  $\rho_0$  be the current best fit for the intersection point of the the curve with the desired CL. Let  $\hat{\rho}$  be the "center of mass" of all the current measurements, possibly weighted by some decreasing function of the distance from  $\rho_0$  (with the weight function possibly depending on how far off linearity the current points are). Then probably the optimal next set of measurements should be done at  $\rho_0 + k(\rho_0 - \hat{\rho})$  where  $k$  is some constant of order 1 (perhaps 1/2). This will in the long run cause the collection of many points near to the intersection, where the non-linearity of the transformed data is unimportant. The fact that most of these points are at nearby

values of signal scale means that the slope of the fit line is less-well determined, but is much less important in dictating the position of the intersection.

(This effect is implicitly acknowledged in the current algorithm, which tries as rapidly as possible to get as close as possible to the intersection point. Unfortunately, the statistical nature of the individual set measurements implies that you don't converge to the intersection point if you keep discarding old data.)

## E Status of Investigations and Implemented Changes

This section describes what has been learned, what steps have been tried, and what steps are about to be tried.

- The structure of the calculation is now understood, and profiling results which point to routines which make heavy use of CPU time make sense in terms of the calculations. This has led to several potentially useful speedup ideas (described in sections 4 and 5). *Done.*
- The structure of the code preparing the test case is understood, and it is now known how to prepare variants which make for easier comparisons, as well as variations on bin counts and parameter counts. this enables the coding of some of the ideas in sections 4 and 5 and the evaluation of which, if any, are helpful. *Done.*
- Create temporary accommodations that allow reproducible behavior rather than seeding with time of day. (In the course of this, a slight flaw in that seeding was uncovered; create changes to fix that flaw.) *Done.*
- Verify that the changes in `collieIOexample.cc` (section 2.2) have the desired effects, and explore the time behavior when bin count or parameter count is varied. *Done.*
- Carefully profile the initial version of Collie, both for a baseline and to decide where to work to improve performance. *Done.*
- Rectify the one or two clear mistakes spotted in the course of the review; this won't speed the code up but will correct erroneous behaviors which are not probed by the current use example. *Done.*
- Examine the s-parameter pattern delivered by Minuit, to see if there is potential for gain by using caching of partial likelihoods (section 4.1). *Done. Large potential gain found.*
- Implement caching of partial likelihoods (section 4.1). Measure speedup and deliver code. *Implementation Done. Code delivery done.*
- Implement one of the physics-conservative approaches to improving fit starting points (section 5.4). I suspect this will lead to a noticeable improvement, in which case deliver the code. *Postponed - likelihood of big improvement is slight.*

- Implement the skipping of unneeded recomputing of parent distributions during a fit, in `fillArrays()` (section 5.2). I consider this change closely related to the next one, so unless the speedup achieved is surprisingly large, I will delay delivering the code till the restructuring of data is in place. *Done.*
- Implement the “Data Structured For Rapid Expectation Calculation” change described in section 5.1. Assuming there is a speedup of 10% or better, deliver this code. *An effective equivalent was done, as part of partial likelihood caching. A speedup of 40% was noted.*
- Clean up several inefficiencies in `calculateChi2LLR` (see section 4.5 for a list of these). *Done.*
- Tighten up memory usage by eliminating (if possible) the arrays which have logically been replaced by the data structures described in section 5.1. *Later. Probably no speedup.*
- Also make certain that D0 users know that turning on the `m_ustat` option in `SigBkgdDist` may be very costly and probably would not do what they intend it to do (section 7.5).
- Use non-limit internal variables for Minuit (section 4.6). This is not expected to give a large speedup, but if it does, deliver the improved code. *Done.*

Following these main intended steps, there are a couple of steps which might be considered if we are still desperate for improved performance. At this time, I feel these will be unlikely (partly because the time limit for starting production calculations will be approaching).

- Combine expectation calculation with LLR accumulation (section 5.5). *Unlikely to be helpful.*
- Enable analytic derivatives (section 5.3). Since this potential improvement will be highly dependent on the number of fitted s-parameters, get agreement as to what constitutes the improvement measurement. If there is a significant speedup, deliver this code. Whether or not these should be enabled, inform D0 that the first round of prime opportunities is now in place; expectations for further large improvements are diminished. *Postponed: Effect clashes with that of Partial Likelihood Caching.*
- Try to find out who did the computation of the maximal appropriate number of pseudo-experiments for a given data level, and where it is written up. Raise the issues discussed in section 5.7 to some relevant D0 physicist (probably Wade Fisher, possibly some statistics committee). If it turns out that there is use for some automated calculation as part of Collie, code that up. (This is not really in the scope of this review, but the potential payoff is so large that it is worth raising the issue.) *Not in scope.*

## References

- [1] Systematic Uncertainties in Higgs Searches, Prague DZero Workshop, Aug. 12, 2008  
<http://www-d0.fnal.gov/d0dist/dist/releases/development/collie/examples/SystematicsDiscussion.pdf>
- [2] Collie: COncidence Level LImit Evaluator,  
Wade Fisher, talk presented Sept 21, 2008. In the examples directory of the collie product in the D0 cvs repository. *Note - I cannot find a better way to access this tutorial.*