A Dictionary Algorithm for CLEO III Silicon Tracking

Diane Chin

Department of Physics

Binghamton University, Binghamton, New York, 13902

Abstract

A new version of a fast dictionary-based pattern recognition algorithm will be implemented as part of the CLEO III tracking software package.designed to sort through hits and produce rough lists of track candidates. By sacrificing large chunks of memory for temporary data storage, the tracker is able to do most of its manipulations at the start of the job, allowing run-time tracking to be accomplished at a rate of on the order of milliseconds per event. Implementation of the algorithm has involved updating existing Fortran code to C++ to take advantage of object-oriented programming techniques and to improve organization. The new code is housed in the `DCTrackLayerSet` class, and uses a managing class called `Scungili` to perform memory allocation and enabling. The portions of the code which load and configure the dictionary data structures have been fully tested with a CLEO III interface package. While interfacing and code for the track-finding portion of the algorithm are complete, rigorous testing has yet to occur.

Introduction

The basic algorithm for the current dictionary-based track `FINDER` being implemented for CLEO III was first developed for use with the MARK III detector at SPEAR[1]. The approach was entirely novel, rejecting traditional iterative tracking techniques for a non-numerical pattern recognition scheme. The tracker is able to examine all of the hits from a detector event and extract a list of possible tracks. All true tracks are virtually guaranteed to be found; some spurious ones are also reported but can generally be eliminated through a more rigorous fitting program at the end of the job. As fitters are iterative by nature, the dictionary tracker provides a vast time savings by narrowing the list of candidates being sent to the fitter to only a few confirmed possibilities resembling true tracks. In the MARK III experiment it was shown to reduce the time spent in a tracking job by sixty percent.

The algorithm's innovative approach also allows the user extensive flexibility in tracking. The pattern recognition is done by comparing binned data from detector events with dictionaries of tracks, keeping tracking well isolated from changes in geometry, et cetera. Adaptation to new environments requires no changes to the code itself beyond specifying a different dictionary source file in the calling routine.[1] Further, the tracker can operate at any desired efficiency level as specified by a list of allowed legal patterns (see Figure 1). Thus by changing a maximum of two input files one may exercise extensive control over the nature of the tracker.

[1] More precisely, no changes are required to the *tracking* code. Changes and additions would almost certainly be necessary to assure that binning and data access was being correctly performed in the tracker's calling routine.

```
NLEGPAT 6
LEG_PAT 1 1 1 1 1
LEG_PAT 1 1 1 1 0
LEG_PAT 1 1 1 0 1
LEG_PAT 1 1 0 1 1
LEG_PAT 1 0 1 1 1
LEG_PAT 0 1 1 1 1
```

FIGURE 1. An example of a list of legal patterns. A legal patterns file need only include the number of legal patterns to be presented and the list itself. Legal patterns include 1s in those layers where a matching hit is required, and 0s in those layers where inefficiency is allowed for. This list is designed for a five layer dictionary. The legal patterns included would find only tracks that perfectly matched those in the dictionary (as denoted by all "1"s in the first pattern) or that deviated by no more than one miss in any layer.

The algorithm was implemented for CLEO II.V in Fortran but suffered significant problems due to higher than expected noise and beam backgrounds. This next update to the code seeks to prepare the code for adaptation to the new CLEO III silicon detector and drift chamber, as well as the developing CLEO III software environment and infrastructure while still allowing compatibility with CLEO II.V.

The Algorithm

The basic premise behind the dictionary tracking method is that extensive use of memory space can be converted to computational time savings. By completing the majority of necessary track finding calculations at begin job time and storing the results, the program places all the information it will need for processing at the event level at its virtual fingertips for quick access. The main point of this research has been to code, organize and test the methods whereby the program does its pre-job processing.

Because of this decision to sacrifice available memory space for reductions in processing time, the arrays used for temporary data storage can grow quite large. Steps have been taken to minimize the volume of memory being used so that the program remains manageable. To this end the tracker relies on tightly packed arrays to do most data storage. Because much of the data consists solely of binary entries that store only yes-or-no answers, the arrays are organized as bit masks where every bit in every variable has significance. The actual units in the arrays are referred to as *words*; an array of five integers (each of which contains four bytes of storage and thus could hold information on 32 different items) would comprise five words of data.

There are three major components to the algorithm. The first is the dictionary itself, which is a simple list of the possible tracks that the tracker will search for when perusing data. The layers of the detector are each divided into sections called bins which are numbered sequentially in R-phi and R-Z projections. Each dictionary entry consists of a list of the numbers of the bins referring to the specific spots in each layer of the detector where a hit must be registered. There is one bin number per tracking layer. The second component is the list of legal patterns that dictate efficiency. The list is binary, placing a "1" in each layer where a hit is required, and a "0" where the tracker will accept a miss. Finally, there is the data itself; for each event in the detector an array of structures is created, where each structure holds information for an individual hit including such items as the bin and layer number where the hit occurred and geometric information necessary for subsequent fitting passes.

The tracker operates in two modes: `MAKER` and `FINDER`. The `MAKER` portion of the routine is associated with writing a dictionary source file from single track Monte Carlo data. At each event, the track is evaluated to see if it contains a hit in each layer and that no more than one hit is registered in any layer. Assuming this is true, the track is added to the dictionary track list. At the end of the job data storage arrays are constructed just as they would be for a full track-finding job so that memory limits for each of the storage arrays can be determined ahead of time and included when the dictionary source file is written. The inclusion of these parameters greatly facilitates memory allocation during track finding.

During a `FINDER` job, a dictionary file created during a previous `MAKER` pass is read in and used in searching for viable tracks among detector hits. An efficient method is used for comparing the binned hit data from the detector with the bin lists in the dictionary; it is facilitated by bit manipulation of a variety of tightly packed arrays of the type described above, and made faster yet by processing information stored in three key groups of arrays: Action, Veto and WorkSpace. Given the location of a particular hit in the detector, the *Action* structures store information about where in the list of dictionary tracks one may find entries containing that hit. The *Veto* structures help to ensure the elimination of redundant tracks from the track candidate list. At certain miss levels, dictionary track entries may appear identical; under such circumstances, the same set of hits may be linked with two distinct dictionary entries and thus be erroneously reported as representing multiple track candidates. In order to prevent this, the Veto arrays hold extensive information on which tracks become mere subsets of each other at lower efficiency levels, as well as on which legal patterns represent these miss levels. When certain combinations of track entries and legal patterns exist that could result in redundant track candidates being reported, the Veto arrays indicate which tracks should be ignored if they are found[2]. The *WorkSpace* consists of a large chunk of memory used for carrying out the bit manipulations necessary for filling the Veto arrays at the beginning of the job and for finding tracks at each event.

[2] In the process of converting the original Fortran code to C++ an interesting bug was detected in the `makeVeto()` routine which fills these arrays. If any legal patterns were not included in the file that other legal patterns could be considered to be subsets of – that is, if a certain miss level were skipped over in the process of making the legal patterns file – those legal patterns which existed at higher miss levels might be ignored in certain cases. Fixing the problem required additional manipulations inside `makeVeto()` but did not extend further than the internal arrays that were deleted at the end of the `makeVeto()` routine; therefore the fix did not increase the amount of semi-permanent class memory being stored throughout a `DCTrackLayerSet` object's lifetime. This is relevant as the bug is unlikely to appear, for few users would be expected to create legal pattern files that allowed pattern recognition to occur at a low, but not a high level of efficiency, and it might have proved most efficient to let that assumption stand and choose not to fix the problem.

Code Structure

In the process of updating the tracker code to the CLEO III environment and to C++, it has undergone massive reorganization. It is now housed in the `DCTrackLayerSet` class, and has become the first functional member of what is expected to grow into a family of `LayerSet` classes inheriting from a generic base class. The `LayerSet` organization allows a user to pick and choose however many and whichever tracking layers of the detector he would like to concern himself with in an analysis. The new organization allows for multiple `LayerSet`s to be used simultaneously on the same data via a managing class called `Scungili`[3]. At the start of a job, an interfacing package calling routine will create an object of the class `Scungili` as well as any `LayerSet`s that might be of interest. The feature of being able to handle multiple `LayerSet`s simultaneously – be they multiple instances of the `DCTrackLayerSet` class or a heterogeneous mixture of a variety of `LayerSet`-derived classes – was one of the main purposes of the code upgrade, extending to the user a new level of flexibility and control over the variety of tracking options available to him. Those instantiated `LayerSet`s which will be used in the coming analysis are enabled through `Scungili`, at which time they are added to the `Scungili` object's list of `LayerSet`s that are to receive detector data whenever an active stream is passed. In addition, each `LayerSet` informs `Scungili` of how much memory it will require in terms of scratch space for its track finding computations. Enough memory is set aside for the largest of these scratch space areas, and a member function of each `LayerSet` is called, passing the `LayerSet` a pointer to the memory set aside.

Enabled `LayerSet` objects receive their detector event data in the form of `TrackHit` objects passed into the `LayerSet` (see Figure 2), each containing information about a particular hit registered in the detector. The hits are filtered by `Scungili` before they are passed in so that each `LayerSet` is only made aware of those hits occurring in layers with which it is explicitly concerned. Each `TrackHit` object contains a variety of detector information including the hit's logical address, three-dimensional world and local coordinates, error approximations and charge deposition quantities. At the `DCTrack` event level, the detector information about the hit is converted into a simple `Bin` and layer number pair through a binning function object of class `Bin`, which is held as member data in the `DCTrackLayerSet` class. The `Bin` class contains no substantial code and all of its methods are declared `virtual` in order to isolate the `DCTrackLayerSet` code from the geometry of the detector. The calling routine is expected to instantiate its own binning function object which inherits from `Bin` and overrides all of its methods. Arrays are stored in the binning function object which allow its methods to map quickly between `TrackHit` logical addresses and bin numbers.

When an object of type `DCTrackLayerSet` is instantiated and enabled in `MAKER` mode (see Figure 3 for a graphical representation of the calling sequence), its `setUp()` member function is called in order to read in and process a list of legal patterns and set aside internal storage for the legal pattern and track list buffers. Its `event()` method is later called for each of a series of thousands or even millions of single track Monte Carlo events that will become the basis of the dictionary. The `MAKER` checks each Monte Carlo track to see that it fits the requirements of an acceptable track entry; if this is true, the track is added to

[3]An Italian word regional to Southern Italy and Sicily meaning "octopus".

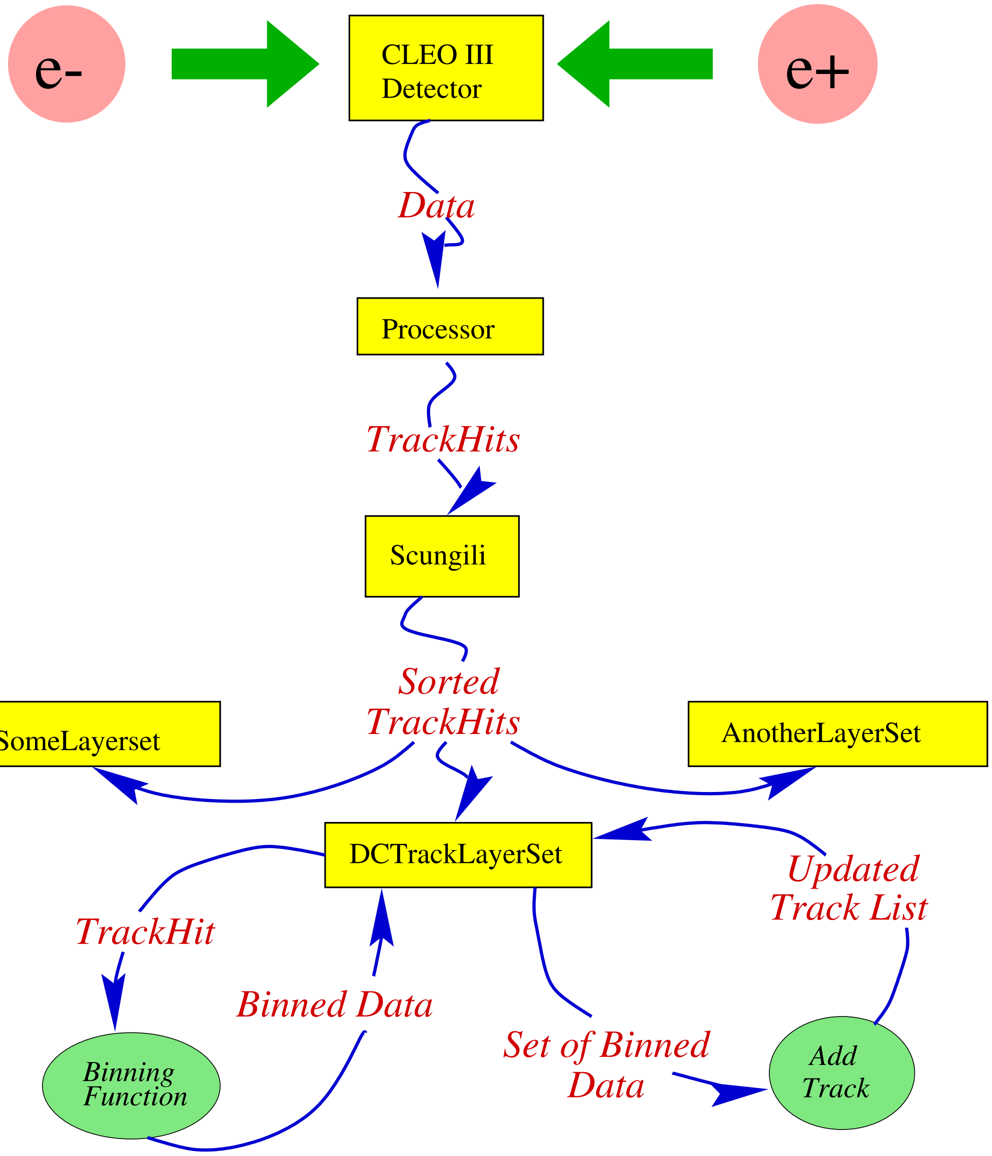

FIGURE 2. An illustration of the calling sequence when an event is registered in the detector. Data is accessed by an interfacing processor which stores hit location information in a vector of `TrackHit` objects. The objects are sorted by `Scungili` and the relevant hits are stored in each `LayerSet`'s `TrackHit` array. `Scungili` subsequently calls each enabled `LayerSet`'s `event()` function for proper processing to occur.

the dictionary. At the end of the job, the tracks are compressed to assure that there are no redundant entries, and all of the necessary storage and data access arrays are created as if in preparation for track finding. When it was originally enabled through `Scungili` at the start of the job, it would have reported that it needed no scratch space. Now that the `MAKER` knows how many tracks need to be processed, it is able to set aside memory for the Action and Veto structures and request its scratch space for WorkSpace arrays from `Scungili`, and therefore must be re-enabled. The `createDictionary()` method fills all of the Action,

FIGURE 3. An illustration of the flow in the life of a `DCTrackLayerSet` object. The yellow rectangles denote the major steps in the process, described in greater detail by the ovals to the side. Differences between the `MAKER` and `FINDER` routines are highlighted with `MAKER` items to the left and `FINDER` to the right. Both `MAKER` and `FINDER` follow the same basic flow, but differ slightly in the order in which they call `DCTrackLayerSet` member functions.

Veto and relevant WorkSpace areas by going through the motions of a full track finding job. The `makeDictionary()` routine which is responsible for creating Action structures is called, followed by `makeVeto()`. Once all of these pieces in memory have been constructed and filled, the `MAKER` uses the contents of the data structures to create the dictionary source file.

Finally everything is prepared and ready to find tracks using real data. A `FINDER` version of the class can be run to conduct pattern recognition and generate track candidates. The `setUp()` function now reads in the entirety of the dictionary source file created in a previous

MAKER pass. Flexibility in the program is maintained as the user has the option of overriding the legal patterns portion of the dictionary file with a separate legal patterns file. Once the legal pattern and track list arrays have been constructed, the FINDER sets aside internal memory for the Action and Veto arrays. It is able to immediately request and allocate the WorkSpace memory it expects from Scungili directly upon enabling; in this case it is extremely important that the setUp() function be called prior to enabling, as this is where the scratch space memory requirements are calculated. The DCTrackLayerSet object should not need to be re-enabled at any subsequent point. At the event level, TrackHits delivered into DCTrackLayerSet by Scungili are handled just as they are in a MAKER job, but this time there is no check requiring perfect single tracks. Rather, the FINDER takes a holistic approach and after receiving and binning all of the hits for the event, calls the initializeFillHitPattern() and fillHitPattern() methods to access the information stored in the Veto and Action structures to carry out the necessary bit calculations for comparing hits with dictionary entries. After all of this, the track candidates are finally extracted using findTracks().

Scungili is given the list of track candidates and subjects them to a quick circle fit to further screen the track candidates. Subsequently, a significantly narrowed set of data can be passed on as seeds to higher level tracking programs to undergo more rigorous fitting processes.

Performance Estimates

The trade-off is being made between memory and speed in this algorithm and while benefits reaped in terms of speed are great, the casualties in terms of memory will not be insignificant. A five-layer dictionary generated in the process of testing MAKER code used 1000 Monte Carlo events and a set of twelve legal patterns. The tracker found 462 good tracks losing over fifty percent of the events because they either failed to meet the appropriate criteria for a good dictionary track, or else were redundant with other tracks that had already been added to the dictionary. A steep fall-off was observed in the number of dictionary tracks being recovered per number of Monte Carlo events run after a certain point (see Figure 4); the option of completeness versus efficiency will have to be weighed when it comes time to create dictionaries on a more massive scale. It was feared initially that memory usage would expand exponentially with increases in the number of tracks being added to the dictionary, but the increase has actually proven quite linear in tests to date (see Figure 5). Dictionary track entries should be constructed with a binning scheme both coarse and fine enough to maintain the rule that a bin location in one layer of a track should never have to look farther than one bin over on either its right or left to find the hit in the next layer, yet still maintain a reasonable amount of resolution. As such, it is expected that 35 bins per layer will be ideal, necessitating that a comprehensive dictionary for an 8-layer LayerSet would include some 70,000 entries – 2000 words worth. Extrapolating from data compiled over the course of making a 7500-event, 5-layer dictionary, this would require on the order of 22 megabytes of storage space for the internal and external arrays. This would be subject to great variations with changes in the number of legal patterns or layers, or if a different, finer binning structure were implemented in its place, but for now looks quite manageable.

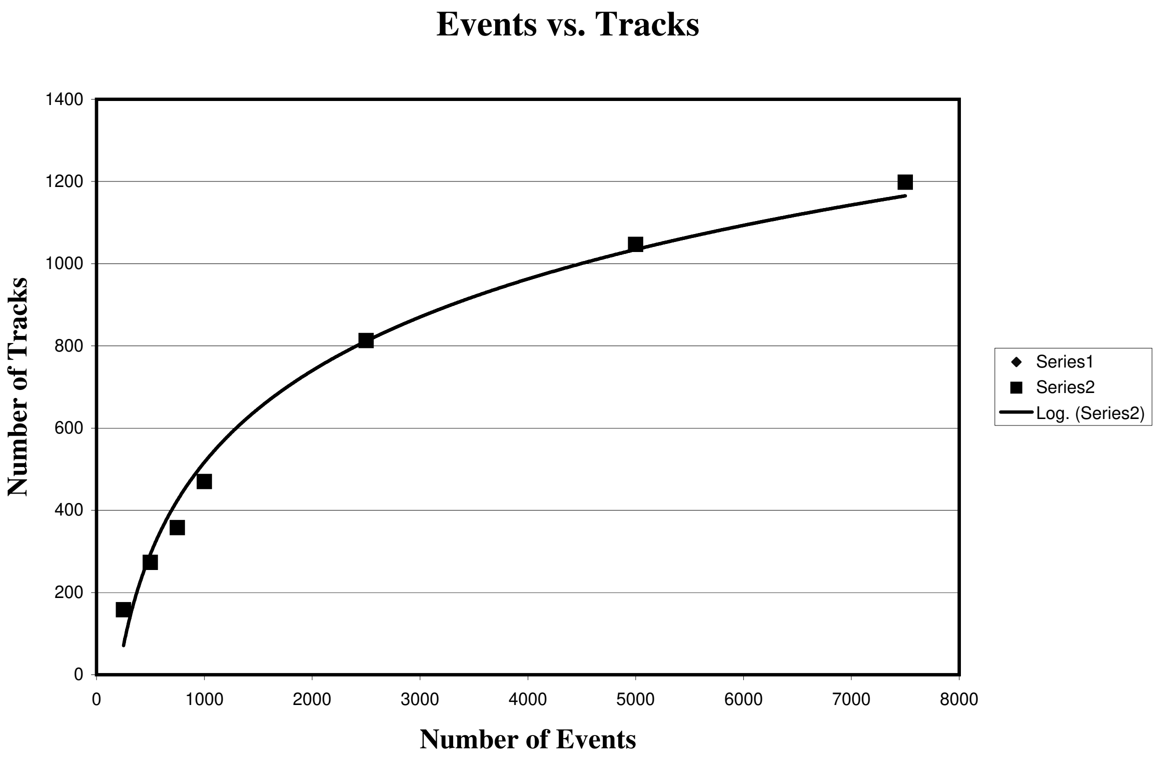

FIGURE 4. The figure above plots the number of unique, good tracks found, versus the number of events processed in the creation of a 5-layer dictionary. All of the events were taken from the same single-track Monte Carlo and processed using a list of 12 legal patterns.

The overall performance of the tracker will depend largely on detector performance, as one might imagine. It is expected that multiple scattering will not have a large effect on performance as it can be simulated with a good degree of faithfulness in a Monte Carlo simulation. Any track that can be produced by single track Monte Carlo – no matter how twisted and kinked – can make it into a dictionary. Any track present in a dictionary will be found in a real-time event. The greatest threat to the long-term effectiveness of the tracker is detector noise and occupancy levels. If these variables remain as low as current estimates predict, the tracker should function fairly well. However, significant rises in these levels could detract greatly from the efficiency and speed of the pattern recognition phase. A sparsely populated detector is an underlying assumption of the tracker, as several time-saving measures have been implemented that allow the `FINDER` to preview which areas of the detector have registered enough hits to represent a reasonable probability of containing real tracks. Regions of the detector which are not indicated as being strong track-finding candidates are ignored. Excessive noise or larger than anticipated occupancy levels would destroy this assumption, and the act of previewing which regions of the detector are worthy of full-scale examination would lose its worth as every region of the detector became highly populated.

In general, however, it is anticipated that the new detector will allow the tracker to function more efficiently than it did in CLEO II.V. This is due at least in part to the

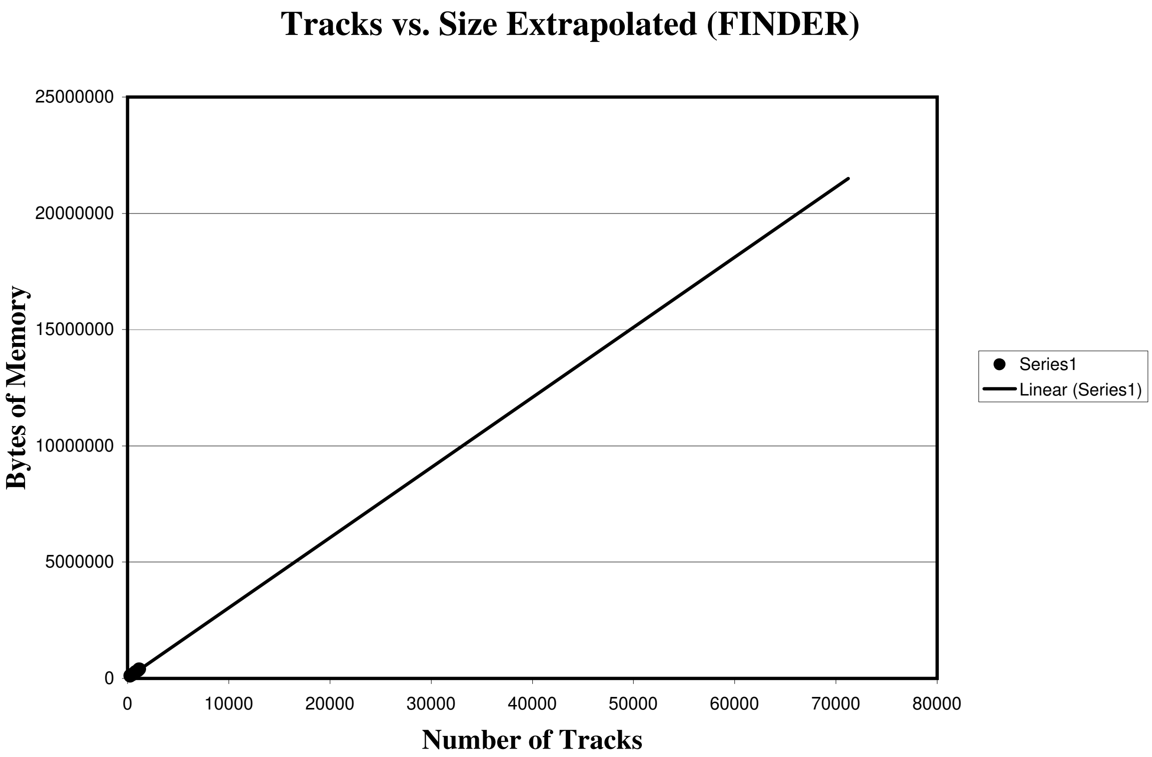

FIGURE 5. The amount of memory used for the various storage arrays in a `FINDER` job relative to the number of tracks present in the dictionary. The data was gathered using dictionaries ranging from 250 to 7500 tracks in length, but has been extrapolated to 70000 tracks to approximate the size of a realistic dictionary.

installation of a much improved silicon detector which expects to register signal to noise ratios ranging from 15 for the inner layers to a minimum of 20 in the outer layers. By itself, the simple addition of a fourth silicon tracking layer will allow for improved performance because of its implications on the way imperfect tracks will be handled. The restriction of having only three layers in CLEO II.V meant that in order to allow room for the possibility of even one miss during a stand-alone silicon tracking job, one was restricted to a mere two hits from the detector as the sole basis for pattern recognition. The presence of four layers in CLEO III will open the possibility of having information from a minimum of three layers at all times, greatly reducing the combinatoric load on the program.

Conclusions

All portions of the `MAKER` code have been completed and have undergone extensive testing in conjunction with a CLEO III interfacing package. The package consisting of a processor and binning function object which have been tested in the CLEO III data access program **Suez** where it is able to access single track Monte Carlo data as well as detector geometry through a variety of standard CLEO-III functions and classes. Construction of an updated CLEO II.V package is underway. Dictionaries ranging in size from 8 to 5000 events have been created, and it has been confirmed that the Action and Veto structures are being accurately constructed and filled. In addition, bugs in the original Fortran routines have been eliminated. With this testing complete, the task of creating dictionaries appropriate to CLEO III silicon tracking may now begin in preparation for when the detector goes on line.

Preliminary versions of the `FINDER` code have been completed, and a processor for use in CLEO III testing jobs has also been made. Testing is in the very earliest stages and is inconclusive, but it may be presumed that there will be a great deal of debugging ahead.

Acknowledgments

I would like to acknowledge Professor J. Alexander and Dr. F. Blanc of Cornell University for proposing this Research Experience for Undergraduates and for an introduction to the CLEO experiment. I would also like to thank Professor J. Thaler, Dr. M. Palmer and Mr. M. Marsh of the University of Illinois at Urbana-Champaign, and Mr. A. Magerkurth of Cornell University for guidance and assistance in completion of the project.

This work was supported by the National Science Foundation REU grant PHY-9731882 and research grant PHY-9809799.

Footnotes and References

1. J.J. Becker *et al.*, SLAC-PUB-3442 (1984).
2. J. Alexander *et al.*, *The CLEO III Silicon Vertex Detector* (1997).