The Workshop on Stability of Particle Motion in Storage Rings,
BNL, Upton, NY, 10/19-24/92.

BNL-48790

The Design of the MAD Design Program *

J. Niederer

AGS Department
Brookhaven National Laboratory

1. Introduction

The study of long term stability in particle accelerators has long been served by a group of widely circulated computer programs. The progress in these programs has mirrored the growth and versatility in accelerator size, complexity, and purpose, as well as evolving technologies in computing software and hardware. A number of large accelerator projects during the last decade were designed with the aid of physics programs either written for, or tailored for the project at hand, each invariably benefiting from contributions of previous workers. This paper outlines the recent history of one example of an accelerator lattice model tool kit, the Methodical Accelerator Design (MAD) Program, which has tried to knit together this collective wisdom of the accelerator community, The ideas behind the software design of the program itself are traced here; the accelerator physics contents and origins are thoroughly documented elsewhere.[1] These informal notes have a Brookhaven flavor, in part because of early BNL efforts to generalize the ways that technical problems are organized and presented to computers. Some recent BNL applications not covered in the extensive CERN documentation are also included.

2. Roots

2.1. At CERN

Physicists author programs primarily to design accelerators, and are seldom able to give the time to refine their own programs for others. CERN has frequently provided an extra dimension of preparing, refining, and documenting a battery of programs for broad distribution as well as for its own needs. MAD, a carefully crafted mixture of accelerator physics and a supporting software environment, is a part of this CERN tradition that has blended scientific, technical, and computing skills over the years.

The MAD Program is one of the results of a long and continuing collaboration among many members of the international accelerator community. Earlier collaborations among D. Carey, K. Brown, and F. C. Iselin and colleagues had produced the Turtle and Transport Programs for beam transport optics. [2,3] This series of programs were upgraded in incremental steps as more was expected of them and newer computers came into use. E. Keil and others had written basic programs to aid in their designs of circular machines at CERN, A. Wrulich at DESY, and H. Wiedemann, and A. Garren and E.Courant in the US were among those who wrote and shared their own programs for circular machines. [4,5,6,7] As accelerators became more complicated, and computer techniques more powerful, CERN accelerator physicists appreciated the need for a well maintained and documented set of comprehensive design programs, based upon data expressed in a clearly stated input language. This accelerator physics software was to be a continuing project, reflecting advances in both accelerator and computing technologies.

Early in their quest for better ways to design accelerators with computers, the CERN authors of MAD introduced a workable accelerator description language. The spectrum of relatively personal input formats among programs had become a quite

*Work performed under the auspices of the U.S. Department of Energy.

noticeable time waster, as programs coming into service were invariably checked against existing ones. The MAD language denoted kinds of lattice elements and program commands by easily recognized keyword names, and numbers and character based information could be entered in free format ways. The early form of MAD directed statements towards intended routines according to keywords, and the individual routines decoded the remaining "attribute = value" phrases of each statement using a set of primitive functions. The menus of keywords and their attribute names were hardwired into the routines, so the flexibility of the language was not yet realized in the programs themselves. Earlier collaborations had already relied upon character formats in inputs, because among other reasons Fortran decoders killed a program at the first mistake, in days when turnaround was hours or even days. Of course even the fixed format programs usually devoted at least one column of an input line to identify contents to the readers, perhaps an early form of class distinction.

The MAD input language design also simplified the organization of beam line and other information. For example, a MAD beam line is simply a list of the names of elements of an accelerator lattice in the order they appear in the machine. For convenience, lines can be subdivided into smaller sublines, and language aids can note repeated elements, reflected sections, and symmetry. The spacing between elements is given by null elements called drifts, and sizes or coordinates of the elements themselves. The earlier CERN MAD releases also included the powerful feature of writing numerical inputs in symbolic form: expressions composed of numbers, references to numerical data in an element data table, and arithmetic operators. A common parameter, such as the current of a power supply shared among magnets, could be linked to each magnet with this facet of the problem description language. Expressions proved particularly helpful in adjusting parameters to specific criteria in matching processes. In retrospect, similar features were appearing in computer operating systems then, and are now taken for granted, but at the time these language innovations were a big improvement in describing increasingly large machine lattices.

2.2. At Brookhaven

During this same period in the early 1980's, both machine and controls designers for the ISA Collider recognized the opportunities for common approaches to accelerator models and to controls systems. Given that the same information was involved in both endeavors, it was certainly reasonable to hope that similar representations of that accelerator information would support both kinds of problems. Budgets of that period for the first time included large sums for major modeling computing capacity in accelerator controls systems. Physics trends were towards detailed tracking analyses for long term stability studies, and towards exploring a much wider variety of possible lattice choices. Concerns about the rich multipole content of the early superconducting magnets often dominated these studies, and also indicated that controls strategies should be tested on viable orbit models. The combination of larger, more demanding studies of more complicated accelerators with far more lattice elements than in the past, and quite limited computing and staff resources to do this work all pointed in favor of improving the way that people pursued these studies. Often the major part of people's time was spent on handicaps that very simple changes to programs could eliminate. By investing modestly in better programming techniques and data organization, it appeared possible to support accelerator design far more effectively. Three kinds of problems were apparent: cumbersome input formats and styles, nitpicking input readers and interpreters, and dreadfully slow computing. The likely order of magnitude improvements in computing speed gained through better organization could support far more realistic applications. As orbit simulations were already filling the fastest computers available, visions of on line controls models needed strong, practical demonstrations of their worth, and potential speed.

BNL designs were also influenced by the major technical advances occurring in the accelerator controls field. In particular, the design of the CERN SPS controls system had achieved an admirable flexibility that also depended on an easy to use accelerator control language.[8] The SPS ideas offered an elegant solution to the problems of building a system that proved essentially open ended as more and more demands were placed upon it, and dramatically overcame serious limitations of the computer technology then available. A series of short controls statements were treated as messages and distributed to the various controls computers over the SPS machine. These messages to read or set apparatus were interpreted locally, and executed by attached hardware. The concepts of classes of information and common information services were present implicitly in these pioneering designs, specifically in the emphasis and treatment of modules that carried out the work. BNL workers had the privilege of reviewing the strengths and limitations of these SPS accomplishments, and improving upon them in light of advancing technology, in particular the advent of modern UNIX based, networked computer workstations.

By the time of the 1982 BNL Workshop on Non-linear Problems, the important principles were understood for creating designs of software that could handle the continual change and evolution of accelerator environments. A popular term applied to these goals was "living code". At BNL a demonstration package was built around the Patricia Tracking Program then in use.[6] The elementary information handling service ideas were further explored for an upgrade of the AGS controls system as well. With a small effort a primitive data base server was built from first principles. The BNL prototypes indicated that a common data base manager, quite close to the technology now called object oriented relational data bases, could serve most accelerator program needs.[9] Accelerators could be easily described within computers as sets of objects identified by name, along with phrases of parameters and their values. All information about each object was stored in a compact module, one per accelerator object. Collections of objects, such the lattice elements of beam lines, were treated as simple lists of names of objects. Moreover, this kind of data management scheme, based upon ordinary and list objects in particular, could be applied directly to the distributed computing servers being planned for BNL control systems. A set of defining input statements described the vocabulary of the kinds of problems which could be treated by the physics programs. An interpreter could understand data entry statements by applying simple syntax rules to distinguish among numbers, names, or punctuation, together with a few rules about line control. Ideally each kind of input statement would be directed to one specific part of the program, usually a distinct routine, and to no other part. There was also a considerable body of physics event processing program experience in dealing with continually changing information: two events seldom had the same number of vertices, the same number of tracks at a vertex, or the same number of data points on a track. A more careful history would undoubtedly find similar ideas among many of the particle physics theses and technical notes of the time, perhaps unheralded because physics was the real goal rather than computing practices. Of course the computer industry was moving in similar directions, especially as the merits of workstations became more apparent.

2.3. The Merger

The separate CERN and BNL developments came together during late 1984, following a set of visits and exchanges among the authors at workshops during the preceding year. This union came about remarkably easily, producing a usable program version in about three months. By now there was reasonable progress towards a uniform vocabulary for describing accelerator lattice elements among the community, and the case for a particular standard language for accelerator programs was further advanced at the 1984 Snowmass Conference.[10] A design for a corresponding interpeter and data base was also presented at Snowmass.[11] Consequently, the decoders of the MAD Program of the time were substantially reworked to become an interpreter of the proposed standard accelerator language, and language extensions which were expected to follow. This interpreter read and checked information presented in input statements, and delivered the extracted information into the modular object structures of a data base which improved upon the Brookhaven design. The BNL experiments with smart, forgiving interpreters to ease input preparation were put aside in favor of the simpler new interpreter based on well tested MAD primitives. BNL and CERN network links were established, and heavily used during this stage of the collaboration. The first releases preserved the form of the original CERN computations, mainly adapting them to compute from information now organized in the modules of the data base. The various matching, optics, survey, and tracking computations all worked off the beam lines, lists, and individual element information stored and managed as individual modules, or objects as they are now called.

This merger was probably much more of a reorganization of the more useful ingredients of the two software projects than a major set of inventions, and indeed some people still wonder what all the fuss about objects is about. Yet a number of previously scattered practices were generalized into a few simple ones. First, it was recognized that information could be grouped into three main categories: basic objects such as magnets, lists of objects, and data tables. A set of services based on the forms rather than contents of data could be designed for each category. The form of the data was specified in keyword or class definition statements collected on a dictionary input file. The language to describe accelerators was thus treated as merely a set of formal parameters, information that could be supplied as data to tell the data base managers how to manage their information. The dictionary file statements were converted to class definition objects, which told the program what information was included in each class, its form, and optionally its default values, conversion factors, and values for validity checking. A class object was in effect a list of the names of the attributes or data items that described each member of the class of object: lengths, strengths, apertures, etc. Thus the definitions of the language were now contained as a set of simple objects in the data base, essentially unlimited in number, and unlimited in the quantity of data or number of attributes described by each. With this organization, it was easy to add helpful language features such as "B is like A, but has the value of 'Color' added". The input decoding process was thus generalized to one interpreter that read all input statements, matched each to a class object in the data base, and created an image of the statement in the data base according to the information on the class object. The resulting objects produced by the interpreter were just compact lists of names and numbers, together with some administrative information to aid with storage. Statement decoding became a self contained service, completely removed from any of the rest of the program. The collection of objects, open ended and flexible, organized as a data base, eliminated all of the troubles of fixed length tables in earlier programs.

Each class of object had a corresponding section of code, which was reached via branches depending upon the class keyword involved. This code was performed only if a statement in the input data stream called for it. As a corollary, if the program crashed, the crash would be in the section called by the last statement accepted. If it was helpful to add more attributes to a given class of object, only the code attached to the class was involved. Changes were transparent to the interpreter and the data base manager, which cared only about the form of the data given by the class definition statements. Along the way it was realized that in addition to number and character forms, combinations of data such as constraints and beam lines could also be conveniently considered as data types. Each data type was really a cell holding a number or name quantity, a data format flag that showed whether the quantity had been filled on the input, and lastly a pointer to tell how to compute the quantity if it depended upon an expression or other reference. In the early work, data structures had to be written manually to agree with the information in the class definition statements. In time, nearly all hard wired information was shifted into the object structures of the class definitions so programs seldom had to be recompiled to change parameters.

This flexibility caused several complications in the way code now had to be written in the processing sections. The most bothersome was having to deal separately with integers, reals, and character data types all nominally equivalenced in the data base.

What had been simple named quantities in previously fragmented small common blocks were now usually dimensioned variables in data structures accessed by an index to the structure. Separate indices were involved for reals and integers in the same structure, a distinctly Fortran affliction, but there were no alternatives then. Travels along linked lists replaced simple DO loops, and there were a few lesser intrusions of style. Nevertheless several thousand lines of code per month could be revised into the new styles, and the inconveniences proved minor compared to the unexpected gains of fewer errors and highly localized trouble shooting. The program was shorter because the duplicated individual decoders and formatters were now combined as general servers keyed to the data base. Entire new sections could indeed be attached to the program without changes to any existing sections.

2.4. Expansion

Over the next several years a number of other major sections were added to the MAD Program ensemble. M. Donald contributed the HARMON Program during a visit to CERN.[12] E. Nordmark contributed to an alignment and field error section. J. M. Veuillin drafted detailed documentation while a visiting intern at CERN. The Lie algebra tracking treatments of A. Dragt and colleagues were revised and added at CERN By F. C. Iselin, and later enhanced by L. Healy. Direct access table managers were added at BNL for handling the large tables generated by tracking of LEP and other large machine orbits. Services to archive and retrieve modules, tables, and sections of the entire data base were added to make the internal data base interchangeable with the file system, features influenced by the limited memory of computers of that era. Several helpful features of the Synch Program were added: archiving of optics results, and the subroutine system, whereby a group of commands could be logically combined for easy reference.[7] Various experiments were tried to make a convenient file system interface that would be portable among the zoology of computers then in use. Internal debugging aids were an integral part of each development, ideally hidden to the users after the codes had been developed and released to others.

The rapidly drafted original sections matured rather well, especially considering their haste of preparation. The orderly mapping between data base class definition statements and program routines indeed proved a powerful method for supporting software change and growth, and has survived almost intact. Various weaknesses in the early memory manager in particular have been addressed at CERN by replacing it with the CERNLIB Zebra manager. While not without its own compromises, Zebra code has the advantage that others are responsible for its compatibility across the spectrum of computers. Various earlier MAD sections judged of little use in CERN applications have been dropped in the most recent CERN Version 8 releases. During LEP commissioning, CERN development naturally focussed on specific LEP needs, and electron machine calculations missing in the earlier versions. More recently H. Grote has been developing spin tracking sections, which in time are expected to be available in a publicly distributed version. A MAD section incorporating differential algebraic methods is under consideration.

3. Object Oriented Design

The general, but fairly straightforward, programming styles which were employed to foster the development of MAD and other accelerator software needs fall into a rapidly evolving computer sub discipline known as object oriented software design. Given their simplicity, the methods have blossomed in many applications. There is already a vast literature and many formal studies of these techniques, which at the time seemed to be a rather natural way to deal with accelerator complexity in computers. In the object oriented paradigm outlined in the previous sections, the representation of information closely follows that of the way that people visualize the information. In the computer, an object is a self contained unit with a name, a place, and descriptive information. The properties of a magnet reside compactly in a magnet object, not scattered widely and unrecognizably over poorly annotated arrays. Other features include a problem description language based on the definition of objects, and an interpreter that moves information from the definitions into usable data structures organized as objects. Storage of information as objects is dynamic, and is handled transparently by a memory manager internal to the program, after the program is loaded. Information from one class of object may be extended to other classes of object, a feature usually called inheritance. Similarly members of a class may inherit information from other members of the class. A more modern compiler can be told that "B inherits the properties of A, and additionally has 'Color'". Data hiding is strongly emphasized: data of an object is shielded from all parts of the program except those which are attached to the object. In pure object oriented software constructions, the actions (processes or methods) of a class are also inherited by other classes and objects. In practice, coding so that actions are inherited among classes may tend to distort the more important computational flow in physics programs when written in Fortran, so this factor is often compromised in MAD code. General inheritance of data among objects is supported in the MAD language, but usually not in the code itself in the modern sense of inheritance.

An object primer written from a physics point of view by P. Kunz, and early texts which helped to draw attention to the advantages of these styles are noted in the references.[13,14,15] A recent issue of IEEE Software give a more formal overview.[16] While the MAD authors struggled with a rather ill suited Fortran as the only computer language readily portable in the accelerator physics world, the object methods have spawned a number of compilers and commercial software tools in the last few years. The Fortran situation will undoubtedly improve with a newer version because the massively parallel processing industry has found that Fortran can be made essentially platform independent and, perhaps not too surprisingly, that object oriented data representations are very well suited to parallel applications.

3.1. Applications

In applying the object design methods, the nature of a problem is first blocked out, taking the form of shopping lists of what actions are wanted, and what information is needed to carry out the actions in the lists. The problem is then partitioned further, preferably into relatively independent units, with a strong emphasis on identifying the kinds of data involved. Routine data services are separated from parts unique to the problem. Related sets of information and the operations to be performed upon them are grouped together to define objects. Implementation considerations are avoided or at least minimized before the shopping and information lists have been assembled.

The level of partitioning or granularity is important. In Smalltalk [17], a pioneering development of the object methods, granularity is very fine, at the operator and operand level. The corresponding flexibility is great, but the complication and overhead is prohibitive in computing intensive applications. In accelerator problems, the granularity is more naturally at the level of accelerator objects such as members of a lattice, and the conventional tracking operations. This level of object is a better balance between the desired accelerator physics and the potential distortions of more elegant formal programming techniques.

For our purposes, it may be helpful to discuss two typical accelerator calculations, beam line organization and the matching of focusing magnet strengths to a required tune, as examples of the use of object classes in MAD. In each example, we begin with a list of what jobs are to be done, and lists of the information needed to do these jobs.

3.2. Example 1. The MAD Beam Line Service

A MAD Line object has a corresponding set of duties or functions:

1. Interpret a LINE command statement, and rearrange its information into a data base form more convenient for later expansion.

   The information of the command is a list of names of lattice elements in the order they appear in a beam line, together with language aids to simplify repetitions and substitutions. This information is rewritten into a LINE image object in the data base, a form of list with multipliers, recursions, and other directives included. The contents of the command are essentially parked until all of its elements have been described to the program.

2. Expand the symbolic information of the line image list into an actual lattice map to link all of the elements required.

   This expansion creates a map kind of list object which contains a cell for each occurrence of an element in the lattice, together with markers for beginning and ending of sublines. Each cell holds a pointer to an element or list object, a code to distinguish lines and elements and mark printing. The cell later may get pointers to related error and matrix structures.

3. Carry out changes to the elements of the map or its stored form.

4. Display map information.

   (View, Dump, Save Services.)

5. Delete a map when finished.

3.3. Example 2. Matching to Constraints

In constraint problems, various characteristics of a particle beam are to be satisfied as well as possible by varying a collection of accelerator parameters. Essentially the application of a constraint can be considered as a calculation of how well one or more quantities approach a specified value or limit at one or more points in the machine lattice. The approach is typically measured by a weighted least squares penalty function involving the deviations resulting from iterating values of the quantities. Thus the primary constraint information includes the point of application, and lists of the constrained variables involved, of corresponding weights, and of the type and magnitude of each constraint. The actual values of the constrained variables will be provided by some outside operation such as tracking, or be computed within the constraint from information provided by the calling operation. This computation and list of information required is said to define a class of constraint object.

The interpreter prepares a constraint object in the data base from a constraint command statment in the input stream. Later a matching command will start a process supervised by the matching driver. This driver will request the routine affiliated with each constraint to perform the following work:

1. The constraint operation must be initialized: references to external information must be satisfied, actual values and weights collected and entered into internal tables, and similar duties performed. Any compromises and overlaps among objects are resolved at this stage. In principle ambiguities should not happen, but a few may nevertheless survive to preserve a language compatibility with an earlier program version, or may be a temporary way around a more serious problem elsewhere.

2. After making a copy of the present lattice map, the matching driver then directs the constraints involved to link themselves into the map at those places specified on the individual constraint commands.

3. The matching driver then constructs a derivative matrix that correlates the effect of each variable on each constraint at a series of grid points. This step amounts to

making a number of tracking passes over the modified lattice map. At each encounter with a constraint, the constraint routine is directed to compute contributions of the constraint to an overall penalty function. The constraint routine also gathers various numbers into the current constraint data object, holding them for possible reference by tabulation services.

4. At the conclusion of matching, a summary pass is made through the lattice. Each constraint is told to write a summary of its final approach to the specified values, and resulting penalty contributions.

MAD supports several classes of constraint: ordinary ones as outlined above, coupled constraints in which values at one location are to be matched at another, elements of first and second order transfer matrices computed between two locations, and specialized calculations such as beam phase envelopes gathered into a skeleton constraint routine designed to accept changes easily. A form of difference constraint that compares changes between undisturbed and undisturbed orbits measured on a machine with values computed by model has been added at BNL.

In this partitioning of the work, all computations for each form of constraint are performed in a corresponding constraint section. Only one kind of data structure has to be involved for each class of constraint. Calling programs in effect send messages to the constraint routines, identifying which constraint object is involved and what is to be done. A summary may be returned to note any errors involved. For efficiency, the constraint data object is referred to by the index to its location in the data base, once the name of the constraint has been matched to its location by the calling program. The kind of action can be noted by a simple index, or by a name more characteristic of the choices involved.

4. MAD as a Programming Environment

This section outlines features which can help with the use of MAD, and aid development of additional computational modules. These features for the most part operate off the data base, and depend upon the form of the data rather than its content, so they may be applied to any sections, old or new, that are compatible with the data base organization. Much of this material is very helpful to people actively involved in maintaining and building the program, but conversely may also be criticized as rather useless bulk by others who just use the program as it is given.

4.1. Data Logging and Reporting.

The major computations provide tabular output in both Ascii formats and direct access table and file formats. These table creating services have evolved from one of a kind codes embedded in individual physics sections to the more recent common servers that respond to an input list of program variables to be tabulated. Opinions have varied widely over the amount of effort to give to data analysis features, a number of which for example are already provided in CERNLIB facilities at CERN. These issues are further confused because the methods and results should be portable across many kinds of computing system. Some have argued that only the simplest of Ascii tables should be written, so more personal methods can be applied to analyze the table files. Part of this viewpoint is clearly influenced by a history of problems in dealing with files in large computer center environments. For the most part, the CERN efforts have tried to provide graphics features within current CERN standards, which unfortunately change every few years as technology advances. The Graphics Kernel System, (GKS) adopted by CERN has not been particularly popular in the US, however, and as GKS now appears obsolete, this kind of graphics tool selection problem again appears. BNL workers have introduced a fairly elementary graphics display package based on the Silicon Graphics, Inc Graphics Library(GL)[18], following an object oriented style that isolates the vendor specific calls. Both the careful GKS and the draft GL efforts involved 6 - 8 k lines of code, plus vast vendor library packages, perhaps a consistent measure of the busy work involved in embedding graphics. Yet with modern software methods, including the MAD facilities, graphics upgrades now take only a few weeks, not months or years, so the advantages of the graphics compete more favorably with the drudgery of providing them.

4.2. Debugging and Maintenance

To be portable among the diverse computers in use among a widely scattered user community, the design of a major program should help to locate both its own faults and those problems which are unique to a particular host computer system. Languages and interpreters invariably introduce a number of unintended amusements as they struggle with abbreviations, misspellings, and other mistakes in the inputs. Debugging traces and snapshots to isolate problems have been embedded in all parts of the MAD Program, and may be enabled selectively when needed. These traces are particularly effective in trouble shooting on distant computers, where quite often compilers have been found to be either at fault, or misapplied in some way. It may seem to be almost asking for trouble by putting character, integer, and real data forms in the same pool common of the data base under a spectrum of different Fortrans, where equivalencing is the only way to describe structures of differing objects residing in the same region. As an example of obscure problems, one vendor thoughtfully rounds off data base integers (pointers and indices in the MAD data base) in a simple data transfer operation (once) presumed equivalent between integers and regular double words. The object oriented nature of the program and data base is particularly helpful in isolating and recognizing problems, as any new section usually involves one major kind of object, described by a data structure. In the test phase, the DUMP service can be directed to print the contents of an example object, which may contain both input information and the results produced by the section being tested. Almost at a glance offending results can be spotted from the trace and dumped snapshots of objects. Specialized commands can provide detailed listing of lattice element matrices to help in trouble shooting lattice designs.

4.3. Data Descriptions.

Successful designs avoid data entry schemes that have artificial obstacles to proper physics, particularly as large machines have thousands of elements. For example, there is very little in accelerator physics that depends on the order in which information is presented for calculation. The task of ordering information is easily shifted to the program itself, so statements detailing the parts of the machine can be loaded at the pleasure of the designer. Only at the time of computation does the program have to be told via a USE command to construct beam lines, organize expressions, and handle initializations and defaults from the acquired ensemble of information.

4.4. Memory Management

Memory management services insert and find the objects mentioned in the beam line and other lists, and link them to the calculations. To a tracking calculation, the list of names of elements in a beam line becomes a list of pointers to defined objects represented by data structures. The memory manager locates objects mentioned in the beam line and other lists, and links them to the calculations. A tracking operation is simply a series of messages to routines affiliated with individual kinds of elements - magnets, drifts, cavities, to advance an incoming track through each element. Transfer matrices or other algorithms may be built into the routines, but for example, anything that is to happen to a quad is treated only in a quad section. In program jargon, all data objects are members of linked lists maintianed by the memory manager. The term pointer here is used in the elderly Fortran sense of an index to the first member of a structure in the common data pool. A single pointer to a structure reaches all of information of an object. A few simple style rules to insure the correctness of the pointer suffice to protect the data base as a whole.

The MAD Program executes directly upon data organized in the object structures of the data base. Pointers to structures are usually supplied by initialization steps that locate objects from their names. A name form of variable in the data base also implies a pointer to the object or expression that bears that name. As both computations and data in structures are compact, significant gains in computer cache and paging efficiencies are achieved. This use of record like structures of information, referred to by a simple pointer in carefully isolated routines, has proved both stable and safe. In the newer BNL sections, variable names in the structures have the same names as those of attributes of object classes defined on statements used to describe the data base in the input dictionary. At BNL, a tool based upon the input dictionary builds the necessary structures, declarations and offsets correctly as the program is generated without further intervention.

Object oriented design texts advise instead that data should be protected by a layer of indirect services that provide copies of safely stored information, a general way to enforce what is now called data hiding. In theory, accessing programs can then only mangle that part of their own data which they are able to get through servers. In practice, this style of access is both slow, and not necessarily as safe as advertized. Another possibility, that of using external directory files for objects, is much too slow to be practical on serious computations, although in principle it is safe.

MAD has one large storage region that holds all objects - lists, elements, tables. The memory manager allocates storage for new objects, locates objects, and may delete objects on request. Temporary tables may come and go in this shared region. This use of a single common information area solved an otherwise annoying problem in predecessor programs of fixed length tables overflowing because storage had been fragmented into numerous smaller dimensioned units of common. In the older fragmented array style, users either have to know a program in enough detail to find and adjust the offending dimensions, or dimensions are set cautiously large, often forcing use of virtual memory paging, and consequently slowing performance. There is a small overhead, in the range of 10 - 20%, in storage for linkages and MAD memory manager administration carried by each object, but more serious wastage from fragmentation is avoided. Of course efficiency considerations become less important as workstation resident memories grow larger and cheaper with time.

4.5. File Systems

MAD makes generous use of files for inputs and outputs. As active use of MAD can easily strain file directories, naturally the MAD authors have wrestled through various thickets seeking practical ways of dealing with files. Computer vendors differ markedly in their approaches and convenience with files, so the MAD authors tried to minimize these differences to users. These attempts to reconcile divergent views about file systems sorely taxed their spirits of altruism. CERN approaches have been influenced somewhat by the realities of working within the IBM file system on which much of the MAD development has been based. Particularly in the early years, there were severe batch machine memory and file directory limitations. In contrast, as very early users of modern workstations, BNL developers have favored the more relaxed approaches offered by UNIX based styles. On some host machines, scripts are helpful for setting up file complements and dealing with file names and versions, so in principle, file assignments can be avoided within the program itself. In UNIX, naming input files and program mode options directly as parameters on the MAD command line is usually more helpful than scripts. Much effort was put into direct access table drivers that in effect provided open ended tables of data to be passed on from one computation to another. It appears that these features were seldom used at CERN, in part because one of their primary uses, the plotting of these tabulated results, was never fully appreciated. On the other hand, these table file features together with a file service option that sequences output file names have proved quite valuable elsewhere on UNIX hosts. There has always been give and take over adding and shedding features which seem useful to one author, but not necessarily to others, and much of this debate has reflected differences in the way computing is supplied at CERN and elsewhere. Furthermore, user patience with more professional computer features decreases as the manuals get longer and more littered with features other than the simplest of accelerator ones.

5. Recent Development at BNL.

Lately several experimental model sections have been introduced at BNL using MAD as a programming environment. These specialized sections do not conform fully to the current CERN maintained versions, and so are not properly considered a part of the publicly distributed MAD at this time. Yet they do illustrate the power and flexibility of the MAD approach to increasingly complex simulations.

The language, interpreter, data base and memory managers, display, trace and snapshot facilities which support MAD calculations can also be viewed as parts of a powerful set of development tools. Several major BNL enhancements are intended to make the program resemble an operating accelerator with a continuously changing environment. Both optical tracking and individual particle tracking were rewritten for greatly increased speed, achieving approximately a factor of 15 improvement in AGS and its booster applications. The data base services were substantially upgraded so a much wider variety of program quantities could be reached by the general change and logging-tabulation services. New drivers manage various stepped procedures, cycling a sequence list of events which might include: set or step selected parameters, compute a resulting orbit, gather, display selected information, perhaps interpret selected data for controlling some feature of the orbit, and repeat the cycle. The object oriented paradigm has proved a natural basis for exploring neural network based models for controllers of accelerator orbits subject to time varying conditions.

5.1. A Feedback Controller Model.

Experience at the SLC has shown the merits of various kinds of feedback systems for directing particle beams through paths subject to deflections varying in time.[19] The development of the SLC beam steering fast feedback system illustrated principles that can be attractively generalized in the form of a neural like network that receives monitor signals, combines them through various filters, computes an error correction signal, and then applies the correction.[20] This error signal is the difference between a measured and a desired value. Each step can be easily represented as a layer of simple neurons, with rules for combining signals, and forming outputs. A neural style of configuring controller signal paths is shown on Figure 1. This kind of feedback control model has been easily connected to the MAD particle tracking sections. Such "real time tracking" models have also shown the usefulness of a noise generator service module to vary lattice conditions. A general data gathering, tabulating module records the evolution of orbit information in time as conditions are varied.

The design of a neural controller simulation involves an analysis of the kinds of information to be sampled, the paths of the information as it is processed among the layers, the range of the computations to be involved, and the paths of the error correction signals back to the correction mechanisms. In the object oriented style, these signals are just the names of items in monitor, filter, or other computational data objects. These basic objects can be thought of as neurons, both for structures and for code. The essential information needed by each neuron is a list of the names of its sources, weights to be applied to the incoming signals, rules for combining the signals, and rules for producing an output signal. An output signal is usually non-linear with respect to the combined signal. The weights may have to be computed or adjusted with the aid of each neuron while the network is functioning, or may be calibrated conventionally by varying correctors and measuring the response at monitors. The hardest part of the software for this kind of a problem tends to lie in the initialization steps, where the names of signals have to be linked to their locations in the programs. This particular model was further complicated by permitting various message passing steps among configured controls computers and networks, to be able to study the contributions of signal passing delays to instabilities.

A typical controller simulation cycle might include:

Set correctors

Apply noise to selected items.

Circulate or pass beam

Read Monitors, subject to noise

Process signals from monitors, specify correction

Distribute correction over network

Log readings, corrector settings / plot online.

Repeat

The beam steering model shows responses and average improvement of the beam displacements similar to those achieved on the SLC when applied to the lattice section just before the arcs. It also shows quite dramatically the ranges of acceptable filter and other parameters that must otherwise be obtained from lengthy measurements. By adjusting several of the non-linear features, characteristics of the neural approach, the model achieves impressive improvement in stability with better gain than in the early SLC controllers. Figures 2 and 3 show the downstream beam response of the controller model to an entering beam stepped off axis, for several different filter parameters in the model.

The model computes rapidly enough to be considered as a real time servo for injection beam lines, accepting measurements and delivering corrections as various quantities drift, achieving millisecond response times. In circular machines, it is more useful to arrange the inputs so they appear to sample the harmonic content of a beam. At one of the layers, the weights for the input signals then tend to resemble the Fourier coefficients of the harmonics. The choice of basis functions can affect how well the strengths of the various correctors are minimized, and the model can be helpful in avoiding geometries where corrector sets oppose each other unrealistically in trying to reduce a small harmonic.

5.2. Operations Models

Early visions at BNL, as well as elsewhere, appreciated the practical advantages of closely linking accelerator models and machine controls applications.[21] In varying degrees, the accelerator is considered a data collecting apparatus, providing information to be analyzed at various levels for possible improvements in orbits and stability. One recently developed constraint feature compares difference data between perturbed and unperturbed orbits with similar difference data computed from the working model. This comparison appears to yield a sensitive measurement of beam momentum relative to the presumed value. A joint display of the separate contributions to a least squares penalty function by each difference constraint, typically applied at each orbit position monitor, dramatically illustrates the sensitivity of the calculation to each measurement and the parameters being fitted. [Figure 4] The set of histograms show the individual approaches to minima at each of the monitors. Any deviations suggest problems with the monitors or the understanding of the lattice in the adjacent region. The sharpness of the combined minimum, the usual criterion for a fit, naturally depends upon each of the individual minima being tuned to occur at the same values for the quantities being fitted.

Most matching processes can also be readily inverted to become models of orbit stabilizing feedback systems, using the neural network features noted above. For example, the collection of difference constraint values computed at monitors can also be considered as a vector of error signals, suitable as inputs for the neural controller model to analyze in on line stabilizing schemes. The information in the correlation matrix is in effect a set of sensitivities among variables such as power supply currents, which translate to weights for a set of neurons. By adding noise, filters, and the like from the menu of objects in the BNL neural network sections, it is easy to build a neural feedback controller that regulates primary machine quantities based upon these error signals, all within the quite general form of the MAD Program.

5.3. Displays

Market surveys have indicated a strong interest in display oriented calculations which show the sensitivity of particle beams to a wide range of variables. Accordingly the BNL data base managers have been extended somewhat to take advantage of new structure capabilities. Several simple data types have been added so table services can be generalized in more consistent, compact forms. To make more information available to the handlers, structures can now be stacked so that succeeding steps of a calculation can access matrices, optics results, or other forms common to a number of different objects. These relatively minor changes have opened up the program for major improvements that have also yielded an order of magnitude increase in speed. With faster tools, matching can be done in non-linear situations with bundles of particles. Orbits and matching behavior may be followed in detail as almost any combination of quantities in a model can now be varied and observed.

It is becoming increasingly attractive to consider gathering together within the MAD programming environment some of the older programs in use for designing, debugging, and operating BNL beam lines. These programs usually have at most a few thousand lines of code, most of it doing computing and data organization and display which is already in MAD in a much more general form. The necessary conversions to the MAD style are straightforward and very rapid with the MAD programming aids. Whole programs can now be converted and tied to MAD workstation based color graphics with much less effort than building graphics into these programs. The advantages lie in closer coupling of the results of dissimilar injection line and early turn capture calculations, and ejection orbits continuing ino external beam lines, together with a common language for describing the problems.

Graphics servers have been organized as a series of objects so that various sections of the program can contribute curves, icons, and other material onto the same screen picture. In this form, a picture is a list of names of objects that includes members of Borders, Axes, Labels, Texts, Icons, ... , Curvedefs, Pendata, etc. classes. A *Curvedef(inition)* draws points, or bars, or a line, worrying about colors, line styles, scaling, log and other conversions, and similar details prescribed on input statements. The data for plotting resides in flexible, open ended table objects.

5.4. Style Evolution

The newer BNL material discussed here has involved some major style changes from those pursued at CERN, the most important of which concern the way structures are designed, constructed, and used. As such, it is best described as being in the spirit of the MAD Programs, but has not been exposed to the careful review and meticulous pruning that distinguishes the final CERN products. The BNL work carried out in a UNIX based workstation environment is in the form of plug in source files which can be readily included with (or excised from) the conventional parts of the older MAD Version 7. Several simple software tools have aided these developments, which involve tens of thousands of lines of code. Probably most helpful is a tool to create Fortran INCLUDE common files that describe the structures of data base objects which is done by reading the class definition statements of the MAD dictionary.[22] Ideally, any change in any of the class definitions is now propagated automatically into both the data base organization and the structure definitions. This tool prepares the necessary compiler declarations of data type and dimension, and corresponding offset statements for the intended structure, and writes each structure as a file in some dedicated directory. In addition to the usual forms of input information supplied on earlier styles of object class prescription, additional attribute cells may be assigned to receive computed quantities, such as matrices and snapshots of various related calculations. Names of command or element attributes defined in the dictionary are combined with a class header prefix to become names of structure variables, using the record - structure formats introduced in the SLC controls software styles.[23] Thus the information about a QUAD, with attributes length, k2, and tilt, is available to the targeted Quad subprogram as the dimensioned variables *quad_length*, *quad_k2*, and *quad_tilt*, each offset properly to match the offsets in the data base, and each addressable by the same pointer. This *Include* tool, actually begun as a high school student programming exercise, essentially removes the pain of composing.the long, ugly, and complicated structures of objects in Fortran77, and in particular gives syntactically correct structures. Probably about half of the composing and debugging work that would otherwise be involved in this style is avoided. Structure definitions are now composed in the simple MAD dictionary formats in one workstation window, and the program itself written in a second program source window.

As an aid to achieving code portable among numerous computers, MAD has been written in a program library format that allows sections to be selected according to kind of computer, operating system, or other criteria. The program library utility also manages insertion of common groups and other aids to consistency. BNL has developed a simple variant of the original CERN Patchy Utility [24] to operate in the UNIX environment.[25] This *Compile* tool converts source library format files into Fortran files for compilation. It rewrites CALL directives (references to common groups - Comdecks) in each routine as INCLUDE statements requesting files from some appropriate sub directory. Group file type suffixes, and prefixes of paths to appropriate directories are attached to bare INCLUDE file names automatically in the conversion step. Source files are organized as groups of related routines for easier editing, following the older CERN Patchy style. A UNIX Makefile governs the assembly of the MAD program from source, dictionary, and INCLUDE files. The Makefile causes Include and Fortran files to be generated from changed dictionary and other source files, and Fortran files dependent upon other changes to be compiled. It links object files to create an executable program. The Makefile technique effectively absorbs all of the complication of managing the file ensemble. The library and structure tools and the advantages of the UNIX workstation environment make it possible to take on some rather large model projects, effectively neutralizing the complications of the object oriented style aggravated by the not so well suited Fortran language.

6. Programming Languages

MAD was designed and largely programmed before the advent of programming languages better tailored to object oriented practices. These newer languages handle more of the technical details of mixing numerous types of data and operations in a data base oriented computation, whereas with Fortran all of this detail has to be managed by the programmer. Thus a frequently heard reply to the question of how one writes object oriented programs in Fortran is "With difficulty". Moreover the detail can overwhelm the code and hide what it is really trying to do, causing a style that readily discourages others. While a major rewrite of MAD still appears marginal, progressively more attractive alternatives may be expected to cause some nervousness. Up to now object oriented languages have implied a layer of highly specialized programmers between accelerator designers and the physics programs, which simply isn't the way accelerator physics is done. It will take a lot of work to find properly portable techniques for adapting newer languages, which is not really the province of accelerator physics. In some trials at

CERN, one new compiler failed to handle temporary objects properly, a major problem for MAD. The benefits of other languages for smaller programs are more evident, and differential algebra examples in particular have been shown to be good match to C++ language use, mostly because of operator definition features.[26]

Fortran 90 appears to have data typing and structure capabilities built into the language that the MAD authors introduced long ago as details of the data base coding.[27] Although much of the Fortran 90 notation is cumbersome and intrusive, especially with regard to declarations, the pointer and structure features do simplify the often complicated indexing now used to accommodate multiple data types within the code body. Moreover, the flavor of the resulting code is very similar to that in the newer BNL structure oriented sections. It appears to offer the same features that make C++ attractive for differential algebra applications. Nevertheless major rewriting at all levels and probably substantial redsign of the data base is involved to take advantage of these features. The newer Fortran, like the object oriented languages, carries a huge amount of baggage in dealing with data types when applied to a large program such as MAD. What is not yet known is whether structure storage schemes lead to predictable patterns of storage among various computer platforms, a critical factor in using common data base services across a broad variety of structures. However painful they might be, the Equivalence features of the older compilers lead to reproducible storage offsets. The Fortran 90 declaration matters probably can be handled by the same kind of tool now used to create declarations from the simpler MAD dictionary definitions. Some of the newer Fortran features are very complicated and can be expected to need an appreciable time before they are widely available, broken in, and portable.

7. Reflections

MAD has surely supported the strong CERN commitment to serve the entire accelerator design community with strong and solid software. The ideas about data organization and management have proved their worth, and the eight year old designs remain timely with respect to current software practices, give or take the Fortran basis. Maintenance and growth aspirations have been realized: the number of MAD commands and features have tripled without affecting the original interpreter or data base design. At Brookhaven, and at CERN, dedicated sections have been attached to aid in increasingly close coupling between accelerator operations and corresponding models of the machines and their injection and external beam lines. The new code sections still join to the old by a single new branch statement in MAD, one new selection switch for the source library utility, and at BNL a few new Unix command lines in the Makefile.

Although the ideas of an integrated software environment have served the authors well, few others have become involved with the detailed development of the MAD Program. While others have also noted that the physics parts are seldom more than 20% of most programs, MAD has not succeeded as a vehicle of choice to provide the remaining supporting functions for other contributors. Other authors have not composed MAD style modules for broad distribution. MAD is hardly singular in this respect, as it serves a small field in which programs are usually highly personal. Indeed, various sections of MAD over the years have instead been adapted into the more personal works of others, perhaps completing or reversing a cycle.

The admirable restraints that provide well tested, portable, and robust code to others also impose major compromises, and professional sacrifices. The CERN style enforces professional programming standards, but from a physicists point of view, these standards unfortunately are often slow to evolve, may be out of date at inception, and may delay new techniques. It takes a lot of extra effort to make programs that are compatible among different kind of computers, and to be widely compatible invariably limits the variety of compiler features to those that work on all of the computers considered.

Passing new code through different compilers may help a little with getting better code, as the various nitpicking differences may also chance upon a real problem from time to time. CERN authors are concerned personally to provide code that is orderly, consistent among its parts, and professionally documented. Releases are broadly checked by physicists among a number of client machine physics groups. But development of newer sections can be very slow, and often others create one of a kind, more narrowly targeted programs rather than wait, a reality that works against the idea of careful standards. But in practice, the results of other codes are invariably compared against those of MAD, the familiar standard.

Aspiriations toward offering state of the art interactive interfaces have been clear victims of emphasizing standards and portability. The MAD object oriented data base is ideally suited to screen displays, but the details of implementing screen displays simply have not been portable across differing computers. Hopefully this situation improves with the next generation of display graphics technology. Also, workstations are now so cheap that it may be preferable to develop a screen display skeleton package on one brand, and expect others to obtain the same brand, or modify the skeleton section to taste, or just generate the program without the graphics.

References

1. Hans Grote and F. Christoph Iselin, *The MAD Program. User's Reference Manual. Version 8.4.* CERN/SL/90-13(AP), (1990), and revisions.

   F. Christoph Iselin and James Niederer, *The MAD Program. User's Reference Manual, Version 7.2.* CERN/LEP-TH/88-38, CERN, (1988).

2. K. L. Brown and Ch. Iselin, *TURTLE.* CERN 74-2 (1974)

3. Karl L. Brown, D. C. Carey, Ch. Iselin , and F. Rothacker, *TRANSPORT - A Computer Program for Designing Charged Particle Beam Transport Systems.* CERN 73-16, (1973), and revisions.

4. E. Keil, Y. Murti, B. W. Montague, A. Sudboe, *AGS.* CERN 75-13 (1975)

5. A. Wrulich. *Tracking Studies for HERA.* Proceedings of Workshop on Accelerator Orbit and Particle Tracking Programs, BNL 31761 (1982)

6. H. Wiedemann *The Program PATRICIA.* PEP Note 220, SLAC (1979)

7. A. Garren, A. Kenney, E. Courant, M. Syphers, FNAL Report FN 420 (1985)

8. M. C. Crowley-Milling, *The Design of the Control System for the SPS,* CERN 75-20 (1975)

9. J. Niederer and B. Morris, *LILA. The Long Island Lattice Analogue.* Proceedings of Workshop on Accelerator Orbit and Particle Tracking Programs, BNL 31761 (1982)

10. D. C. Carey and F. C. Iselin, *A Standard Input Language for Particle Beam and Accelerator Computer Programs.* Proceedings of the 1984 Summer Study on the Design and Utilization of the SSC, AFS, (1984)

11. F. C. Iselin, E. Keil, and James Niederer, *Common Data Base Programs for Accelerator Physics.* Proceedings of the 1984 Summer Study on the Design and Utilization of the SSC, APS, (1984)

12. M. Donald, *The Program Harmon.* PEP Note 311, SLAC (1979)

13. P. F. Kunz, *Object Oriented Programming.* Particle World, Vol2, No 4, Gordon and Breach (1991)

    R. Blankenbecler and L. Lonnblad, *Particle Prodution and Decay in an Object-Oriented Fromulation.* ibid.

14. Bertrand Meyer, *Object Oriented Software Construction.* Prentice Hall, New York, N.Y., (1988)

15. Brad J. Cox, *Object Oriented Programming, An Evolutionary Approach.* Addison Wesley, Reading, Mass., (1987)

16. IEEE Software, Vol 10, No1, (January 1993)

17. A. Goldberg and D. Robson, *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley, Reading Mass, (1983)

18. *Graphics Library.* Licensed Software Product and Documention of Silicon Graphics, Inc.

19. J. T. Seeman, *Observation and Analysis of Oscillations in Linear Accelerators.* The Physics of Particle Accelerators, AIP Conference Proceedings 249, AIP, New York (1992)

20. J. Niederer, *Design Notes for Neural Models.* Informal Documentation, BNL, (1993)

    The BNL MAD Controls Feedback Section: Input Language Guide. Informal Documentation, BNL, (1993)

21. M. J. Lee, et al, *Models and Simulations.* SLAC Pub 3217, (1983)

    M. J. Lee, et al., *GLAD, A Generic Debugger.* SLAC Pub. 5700 (1991)

22. J. Niederer, *The Include Program for Generating Structure Files.* Informal Report, CCD, BNL, (1991)

23. J. Bogart, et. al., *The SLC Control System Basic Users Guide.* SLAC, Internal Documentaton, 1988

24. H. Grote and M. Metcalf, *Patchy.* CERN DD/EE/79-4 (1979), and revisions.

25. J. Niederer, *The BNL Source Library Program.* Informal Report, CCD, BNL, (1991)

26. L. Michelotti, *MXYZPPLK, A C++ Version of Differential Algebra.* FNAL Report FN 535 (1990)

27. Walter S. Brainerd, Charles H. Goldberg, Jeanne C. Adams, *Programmer's Guide to Fortran 90.* McGraw-Hill, New York, N.Y. (1990)

Figure Captions

1. Signal Connections Among Neurons in a Linac Steering Model.

Monitors, Filters, Mixers, and Correctors can be viewed as Planes of connected neurons, feeding signals forward to succeeding layers.

2. Linac Feedback Model. Typical Response Curves. Gain.

An offset entering beam with noise (LX1 Curve) is corrected by the simulated controller. The four regions of the corrector response curves show the effect of different gains in the mixer neuron layer.

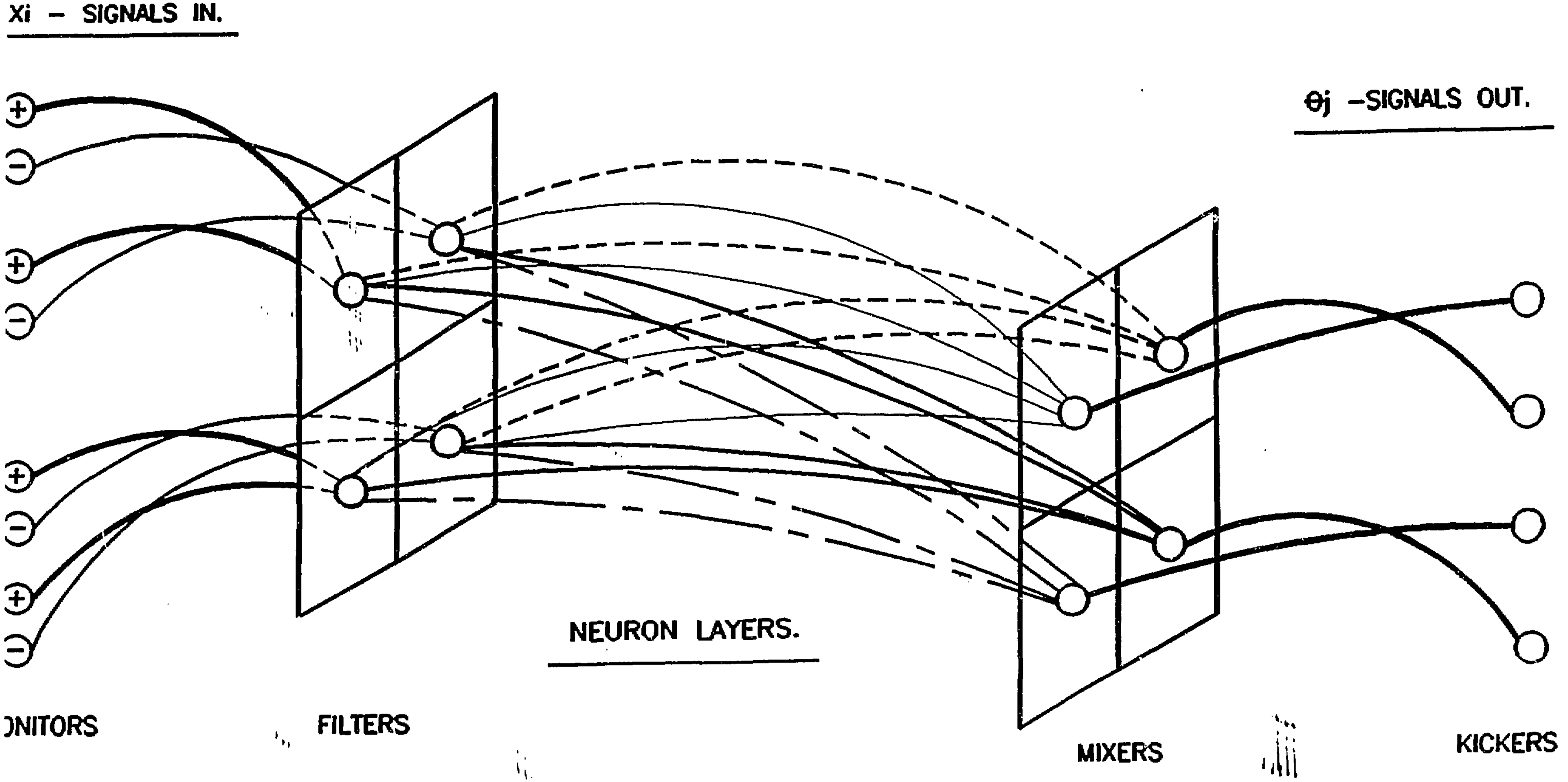
3. Linac Feedback Model. Typical Resonse Curves. Filter Options.

A rather noisy entering beam (LX1 Curve) is corrected. The four regions of the X Position Monitor Curves show the response of the correction system with various combinations of filters (median, exponential, predictive) activated in the filter layer.

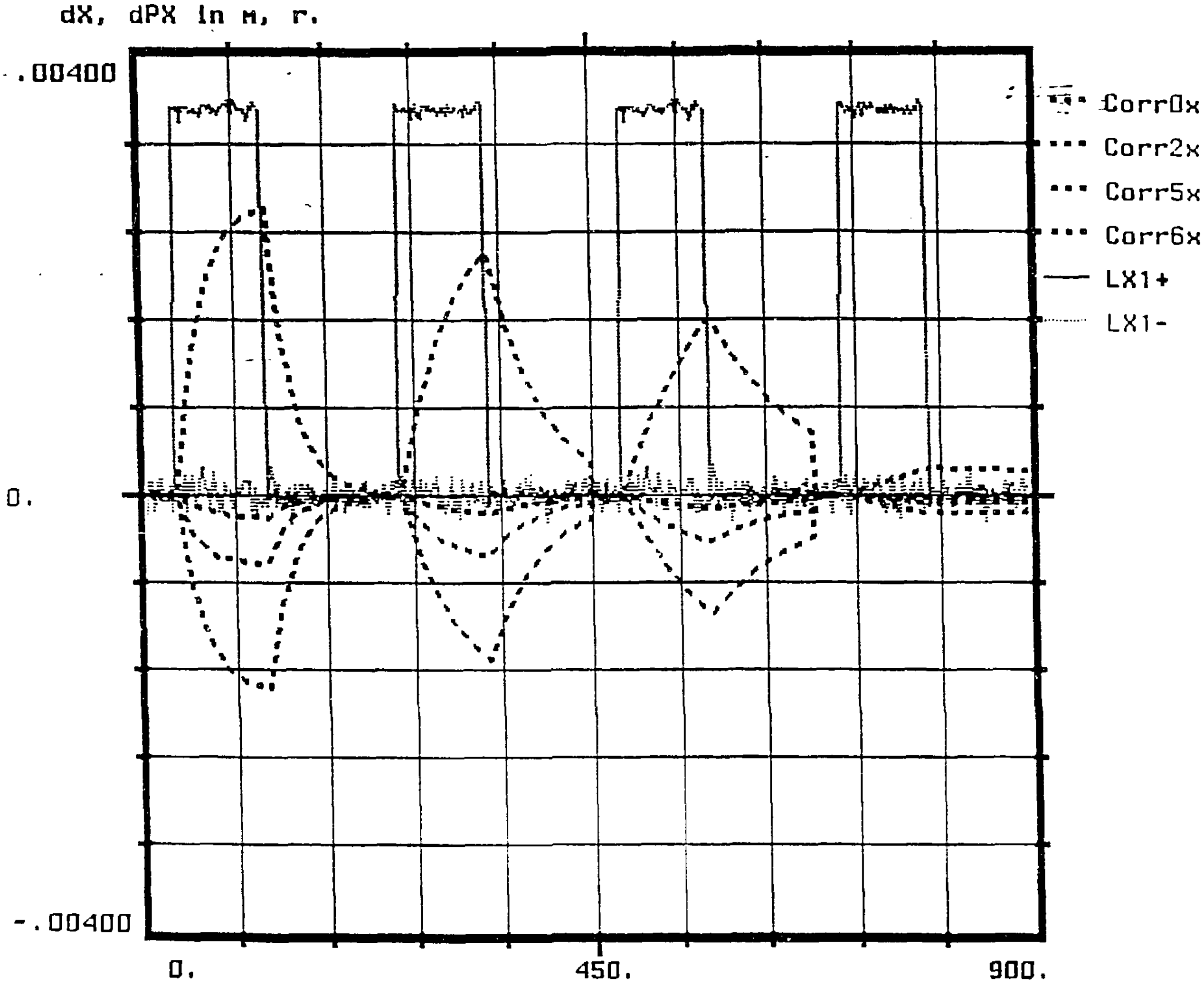
4. Display of Least Squares Comparison of Circular Orbits with Model.

Measurements of perturbed and unperturbed orbits are compared with results computed with models of the AGS Booster at orbit monitor positions. This kind of tool helps to locate regions of disagreement, and to show the sensitivity of matching calculations to features of the model.

NEURAL MODEL OF SLC BEAM STEERING.



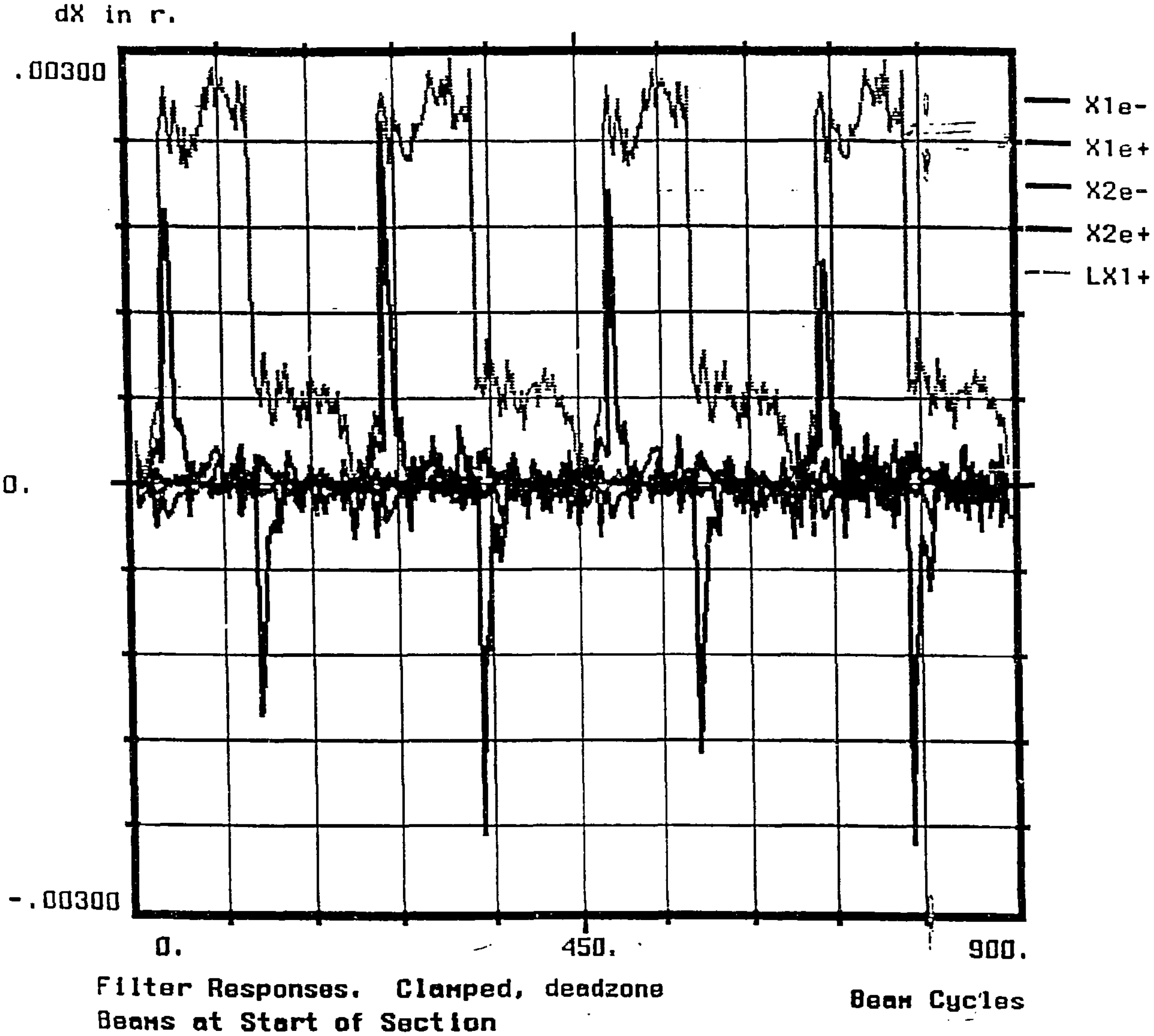
Feedback. Response to Offset in X.
Compare Gain Options.
4 * 4 SLC Co..igurations.



Corrector Values
Beams at Start of Section.
Gain = .5, .2, .1., .01

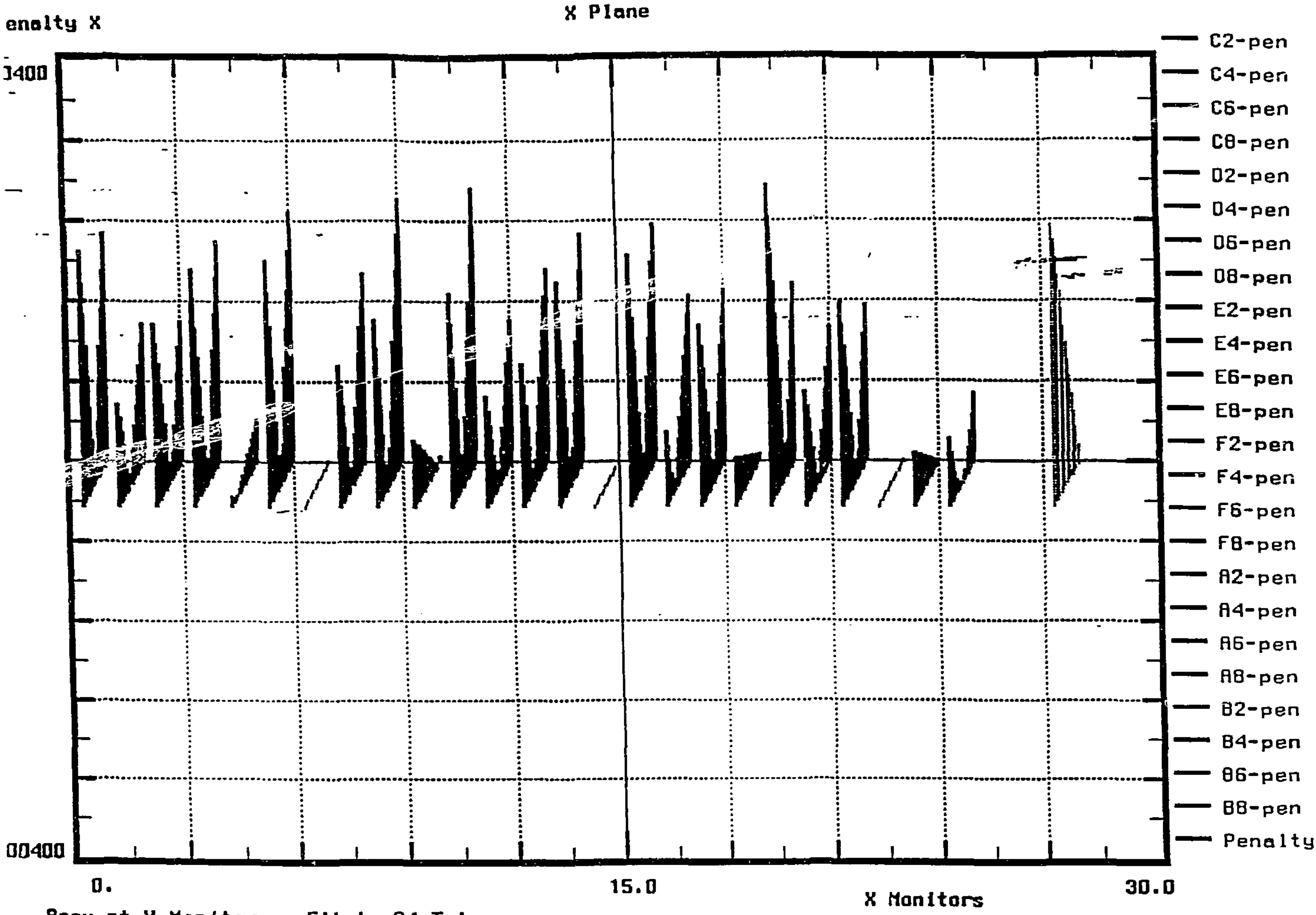
Beam Cycles

Feedback. Response to Offset in X.
Compare Filter Options.
4 * 4 SLC Configurations.



Filter Responses. Clamped, deadzone
Beams at Start of Section
Gain = 1., 1., 1., 1., Filter = -m, -xp, -p, -mxp

Figure 4
Booster Orbit Fitting
X Plane



Beam at X Monitors - Fit to A4 Trim.
Constraint Least Squares at Monitors.
FMatching - Eddy Runs

AKICK

Result = 0.00013933

03/26/93
10:53:13