Accelerator Physics Analysis with an Integrated Toolkit

J.A. Holt, Leo Michelotti, T. Satogata

Fermi National Accelerator Laboratory
P.O. Box 500, Batavia, Illinois 60510

August 1992

Presented at the XVth International Conference on High Energy Accelerators, Hamburg, Germany, July 20-24, 1992
Disclaimer

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.
Accelerator Physics Analysis with an Integrated Toolkit

J. A. Holt, Leo Michelotti, T. Satogata
Fermi National Accelerator Laboratory†
P. O. Box 500, Batavia, IL 60510, USA

Abstract

Work is in progress on an integrated software toolkit for linear and nonlinear accelerator design, analysis, and simulation. As a first application, "beamline" and "MXYZPTLK" (differential algebra) class libraries, were used with an X Windows graphics library to build an user friendly, interactive phase space tracker which, additionally, finds periodic orbits. This program was used to analyse a theoretical lattice which contains octupoles and decapoles to find the 20th order, stable and unstable periodic orbits and to explore the local phase space structure.

1 Introduction

There are a number of good general-purpose accelerator tracking codes in wide use such as SYNCH [1], MAD [2], or TEAPOT [3]. Each of these programs has an input format or language in which the user formulates the problem to be solved. This language is different from the language the code is written in. Frequently the accelerator physicist wants to solve a problem which does not fit within the constraints of the input language. The physicist must then duplicate the basic lattice handling functions which all codes must have as well as the special functions which solve the problem at hand. Also some problems can be solved more quickly when the input can be changed interactively and both the intermediate and final results be seen graphically. Most codes at present are a batch type operation; an input file is edited, the program is run and the final results are output. Exploring phase space to search for a separatrix for example, in this manner is a very time-consuming process.

We are developing a set of tools in which the input language is the same as the programming language. We have implemented our code in C++ because of the ease in creating new types that behave in every respect like fully functional variables of the language. We have integrated several types of class libraries to form a custom application to "solve" a particular problem. Different authors worked on different parts of the libraries. Because everything was encapsulated in objects, one author did not have to know any details of the objects created by another; only the interface to that object had to be known.

The following sections briefly describe the class libraries used to build the application. Some have been explained in greater detail elsewhere. We have used MXYZPTLK which is a set of DA classes which implement differential algebra, beamline which is a class used to simulate beamlines and rings, and a collection of X Window/Motif classes which are used for the user graphical interface.

2 Differential Algebra Class Library

The class library MXYZPTLK [4] is an implementation of differential algebra in C++. There are two classes DA and DAVector which are derived from doubly linked lists. Each link in a list contains the "index array" for a particular non-zero derivative and its weighted value. When a DA variable is first declared, it is a list with no links. The links are created dynamically as calculations proceed. Full use is made of the function overload capability of C++. All arithmetic operations including mixed-mode operations are handled automatically. No special function calls or precompiling is necessary. In addition, each DA variable keeps track of its own attributes, such as accuracy and reference point. For example, consider the simple function $x(m)$ defined by the equation

$$x(m) = \cos(m \cdot x(m))$$

(1)
Simple recursion can be used to construct \( x(m) \) for \( m \) in the approximate range \( m \in (-1.2, 1.2) \), determined by the condition \( |m \sin(m \cdot x(m))| < 1 \). The same recursion, applied to \( \text{DA} \) variables, provides the derivatives as well. The C++ code fragment [5]

\[
\begin{align*}
\text{DA} & \quad m, x; \\
m & \quad \text{setVariable}(0.5, 0); \\
x & \quad \cos(m \cdot 0.9); \\
\text{for} \( i = 0; i < 15; i++ \) & \quad x = \cos(m \cdot x); \\
x & \quad \text{peekAt}();
\end{align*}
\]

will evaluate derivatives of \( x(m) \) at \( m = 0.5 \) \((0.5) = 0.900376\ldots\). Note that C++ automatically handles the mixed mode arithmetic in the third line. The \text{peekAt} \ member function prints the desired derivatives.

3 Beamline Class Library

An accelerator is a collection of objects connected together to form beamlines, a structure modelled very naturally by C++. The class \text{beamline}[6] \ is derived from two base classes; a doubly linked list and \text{bmlnElmnt}, which holds information common to all beamline elements, such as geometry. Because \text{beamline} \ is derived from \text{bmlnElmnt} it is easy to insert one \text{beamline} into another, thereby building complicated models hierarchically. As a trivial example, the following C++ code fragment is one way of constructing a five-cell FODO lattice.

\[
\begin{align*}
\text{double} & \quad \text{length} = 1.0, \text{focalLength} = 1.0; \\
\text{drift} & \quad \text{O}(\text{length}); \\
\text{thinQuadrupole} & \quad \text{F}(\text{focalLength}); \\
\text{thinQuadrupole} & \quad \text{D}(\text{-focalLength}); \\
\text{beamline} \quad A & \quad (\text{F}); \\
A & \quad \text{append}(\text{O}); \\
A & \quad \text{append}(\text{D}); \\
A & \quad \text{append}(\text{O}); \\
\text{beamline} \quad B; \\
\text{for} \( \text{int} i = 0; i < 5; i++ \) & \quad B.\text{append}(\text{A});
\end{align*}
\]

The first lines declare variables \( \text{O}, \text{F}, \text{D} \) as the basic beamline elements. This is followed by a series of \text{append} statements in which the elements are inserted into a cell, called \( \text{A} \). The loop at the end constructs a beamline \( \text{B} \) consisting of five cells of \( \text{A} \). In a beamline constructed in this manner, adjustment of \( \text{F} \) will adjust \( \text{F} \) everywhere.

Once declared, \text{beamline} objects can be used to do tracking, evaluate lattice functions, construct polynomial maps, and so forth. For example, the following code fragment belongs to a simple program which compares element by element tracking of a nonlinear beamline, called \text{E778}, with polynomial map evaluation.

\[
\begin{align*}
\text{beamline} & \quad \text{E778}; \\
\text{proton} & \quad \text{p}; \quad \text{DAproton} \quad \text{pd}; \\
\text{double} & \quad \text{w}[6], \text{y}[6], \text{z}[6], \text{zero}[6]; \\
\text{DA} & \quad \text{zd}[6]; \\
\ldots & \quad \text{// Construct polynomial map} \\
\text{pd}.\text{setState}(\text{zero}); \\
\text{E778}.\text{propagate}(\text{pd}); \\
\text{pd}.\text{getState}(\text{zd}); \\
\ldots & \quad \text{// Do tracking} \\
\text{for} \( \text{int} i = 0; i < 50000; i++ \) & \quad \{ \\
& \quad \text{// -- with the map} \\
& \quad \text{for} \( \text{int} j = 0; j < 6; j++ \) \\
& \quad \text{y}[j] = \text{zd}[j].\text{multiEval}(\text{z}); \\
& \quad \text{for} \( \text{j} = 0; \text{j} < 6; \text{j++} \) \text{z}[j] = \text{y}[j]; \\
& \quad \text{// -- element by element} \\
& \quad \text{E778}.\text{propagate}(\text{p}); \\
& \quad \text{p}.\text{getState}(\text{w}); \\
& \quad \ldots
\}
\end{align*}
\]

First, a polynomial map is constructed by propagating a \text{DAproton} object around the beamline and storing its resultant state in an array of \text{DA} variables. Within the loop, the \text{multiEval} \ method is used to evaluate this map for comparison with element by element tracking, done by propagating a proton object around \text{E778}.

4 Graphic Class Library

Using Young's widgets [7] as foundation, we have implemented an X Window/Motif graphic interface in C++. Classes such as fileselection, menubars, command buttons, popup windows and text input have been implemented making it easy for the user to customize the interface. The most important class is the \text{phaseSpace} class in which 4D phase space variables amplitude and phase (this can be changed to \( x-x', y-y' \) easily) are plotted. The user can click anywhere in the phase space region to set the initial conditions for tracking. This can be done while propagating through a beamline or map. Searches for a separatrix for example can be done very quickly.

The graphics interface communicates with the beamline class through a class called \text{genericMap}. This class has methods such as start and stop propagating, setting or retrieving the tunes, and finding periodic orbits. The user can change maps or beamlines or vari-
ables to be plotted without touching the graphics interface. It is also easy to add a new method to this class and connect this method to a command button or menu item in the interface.

5 A Simple Application

Figure 1 shows a simple application which was used to look for the periodic orbits of a theoretical lattice [8], the main elements of which are three octupoles separated by sixty degree phase advances and a decapole separated by a forty-five degree phase advance. A stable and an unstable period twenty orbit are displayed, along with the corresponding separatrix. One of the tori surrounding the stable orbit is also drawn.

Previous work by one of the authors [9] used coupled three-dimensional projections to explore four-dimensional phase space. Figure 2 shows the same lattice and periodic orbits using AESOP, a graphics shell originally written for the Evans and Sutherland PS330 terminal. AESOP, and the “GrafXPad” classes on which it is based, are being rewritten using X11R5 Phigs so as to make them available on any X workstation.

References


