BNL-99472-2013-TECH
C-A/AP/323;BNL-99472-2013-IR

User Guide for UAL (C++ Interface)

N. Malitsky

September 2008

Collider Accelerator Department
Brookhaven National Laboratory

U.S. Department of Energy
USDOE Office of Science (SC)

DISCLAIMER

C-A/AP/#323
September 2008

User Guide for UAL (C++ Interface)

N. Malitsky and R. Talman

Collider-Accelerator Department
Brookhaven National Laboratory
Upton, NY 11973

USER GUIDE FOR UAL (C++ INTERFACE)

N. Malitsky and R. Talman

Abstract. Unified Accelerator Libraries(UAL) provide a modularized environment for applying diverse accelerator simulation codes. Development of UAL is strongly prejudiced toward importing existing codes rather than developing new ones. This guide provides instructions for using this environment. This includes instructions for acquiring and building the codes, then for launching simulations and interpreting results, for example for the examples included with the distribution. In some cases the examples are general enough to be applied to different accelerators by mimicking input files and input parameters. The intention is to provide just enough computer language discussion (C++ and XML) to support the use and understanding of the examples and to help the reader gain an adequate understanding of the overall architecture. This document, and the underlying UAL code can be found at `http://code.google.com/p/ual` where they will be updated occasionally.

This user guide supercedes the UAL-PERL Interface User Guide[**7**], which, though now largely outdated, describes much of the motivation, organization and evolution of UAL. The main way in which that guide is outdated is that the user interface has now been migrated from PERL to C++. Most of the PERL interface examples continue to be valid but, since this interface is not being maintained, its use is deprecated. Browsing these examples gives a sampling of the sorts of capabilities, such as lattice tuning methods, that are available. If an example seems close to what is needed, it can be run from the PERL-interface to gain experience. But the code should then be ported to the C++interface before serious specialization is begun.

Portions of the present user guide consist of porting PERL script explanations, to line-by-line annotations of equivalent C++ code. The basic physics underlying the code is unchanged. New physical application examples (such as the space charge code including radiative effects, `stringsc`), use the C++ interface. Older (and typically more elementary) examples, using the (deprecated) PERL interface are in the process of migration to the new interface.

Contents

Chapter 1. Introduction 1
1.1. The Essential Complexity of Accelerators 1
1.2. The Intended Purpose for UAL 1

Chapter 2. A Superficial Introduction to C++ 5
2.1. Introduction 5
2.2. Modularity and Code Organization 6
2.3. Object Orientation 8
2.4. Encapsulation, Classes, and Methods 8
2.5. Polymorphism 9
2.6. Other Programming Terms 10

Chapter 3. Annotated Example 13
3.1. Evalustion of Lattice Functions, etc. 13
3.1.1. Explanations 13
3.1.2. Listing of Main Program `shell_sns.cc` 18
3.1.3. Listing of the SNS ADXF File `sns.adxf` 19
3.1.4. Sample APDF Element-Algorithm Linkage File `tracker.adxf` 22
3.1.5. Partial Listing of `setup-ual-env` 22
3.1.6. Makefile for `shell_sns.cc` 24
3.1.7. Output Listing `ring.twiss` 26
3.1.8. Graphical Output 27

Appendices 29

Chapter D. Annotated Examples (PERL-interface) 31
4.1. Basic Example (PERL) 32
4.2. Selective Lattice Function Output 39
4.3. A Personalized Shell for Code Development 40
4.4. Fringe Field Map 44
4.4.1. Map Generation 45
4.4.2. Map Application 48
4.5. SXF Input to UAL Simulations 49
4.5.1. SXF Rationale 49
4.5.2. FastTeapot 50

Chapter E. ADXF: A Markup Language for Accelerators 53
5.1. Data Interchange Among Accelerator Simulation Programs 53
5.2. Vocabulary 53
5.3. “Design” Elements, “Real” Elements, “Installed” Elements. 57

5.4. The Need For Local Reference Frames 58
5.5. Definition of Local Frames 59
5.5.1. A Minimal Discrete Abstract Skeleton 60
5.5.2. Reference Frames and Picture Frames 61
5.5.3. A SIF (Continuous) Skeleton 61
5.5.4. A Continuous Abstract Skeleton 61
5.6. Precision Positioning of Elements 63
5.7. Treatment of Lattices With Vertical Bends 63

Chapter F. Acronyms and Suffixes 65

Chapter G. Lattice Parameter Definitions 67
7.1. Local particle coordinates 70

Chapter H. Truncated Power Series and Lie Maps 71
8.1. Function evolution 71
8.2. Taylor series in more than one dimension and Lie maps 72
8.3. Symplecticity of Lie map 74
8.4. Hamiltonian maps 75
8.5. Discrete maps 75

Bibliography 79

Index 81

CHAPTER 1

Introduction

1.1. The Essential Complexity of Accelerators

If "pure physics" is the analysis of physical systems made possible by their idealization, then the pure physics of accelerators reduces to a rather small package. Circular accelerators *basically work*, with properties matching the expectations of Lawrence, Kerst-Serber, McMillen-Veksler, Courant-Snyder, and so-on. The rest of accelerator physics amounts to understanding why accelerators *don't work quite the way they are supposed to,* or rather, to efforts to make their performance correspond more nearly with their idealizations. The problem is that the actual performance of an accelerator depends on features that violate the idealization. At the risk of exaggerating the point, one might therefore say that the bulk of accelerator physics is an antithesis of pure physics, in that it amounts to analyzing nonideal systems. For example, Hamiltonian requirements are easily met only in a linearized approximation that becomes progressively less valid as accelerators become larger. This has required modest extensions of the theoretical foundations.

But far more essential difficulty comes from the vastly increased number of influential degrees of freedom that enter as *deviation from the ideal* needs to be described. This is nowhere more true than in accelerator physics, with its thousands of pieces of uncorrelated data, all needed for a complete description. Since the handling of complicated data is more nearly the subject of computer science than of physics, it seems sensible to take advantage of advances that have been made in computer science. Issues such as object modeling, exchange formats and database management become almost as challenging as the underlying physics.

1.2. The Intended Purpose for UAL

The Unified Accelerator Libraries (UAL)[**1**] [**2**] are intended to help to "manage the complexity" of accelerators by providing an environment for simulating a variety of properties of a variety of accelerators using a variety of simulation codes and methods. The intended value of the environment is to provide homogeneous access to these resources while preserving their functionality, yet masking their diversity and assuring their consistency. This allows different methods to be consistently applied to the same accelerator and the same methods to be applied to different accelerators. Another potential benefit is the feasability and economy of supporting "infrastructure" (shared resources) such as postprocessors, plotting/histogramming/fitting, input and output translation, and parallel processing.

Some of the object-oriented programs (all or part of which are) included in UAL are: PAC (Platform for Accelerator Codes), ZLIB (Numerical Library for Differential Algebra), TEAPOT (Thin Element Program for Optics and Tracking), ACCSIM (Accumulator Simulation Code), SIMBAD (particle-in-cell 3-D ppace charge calculations in the Unified Acceleratory Library, upgrade of ORBIT)[**48**] and TIBETAN (longitudinal phase space tracking). Modules that are partially supported or are under development include ICE (Incoherent and Coherent Effects), AIM (Accelerator Instumentation Module), and SPINK (tracks polarized particles in circular accelerator).

Accelerator simulation programs usually have a dedicated, proprietary, input language that describes the lattice that is to be investigated, and the quantities to be calculated. For simple systems and calculations, the semantics (meaning) of the input language is not very complicated, and the syntax (crossing t's and dotting i's) is (though very fussy) quickly learned. These codes are heavily *procedural*, with standard sequence of operations, and little variability of method.

UAL does not adhere to this pattern; it has no proprietary input language. Rather, the description of the calculation to be performed is a C++, *main* program. For reasons to be gradually explained this is a powerful approach from a computaional point of view (because it exploits the object-orientation of the environment). But, even for a simple system, this presents the beginning user with the steep learning curve caused by the intimidating level of abstraction.

The intention of this User Guide is to give a not-at-all detailed discussion of the UAL environment, as it has been updated to have a C++interface, and to provide annotated examples that can be used as templates for getting started on using the UAL code for simulating the performance of any accelerator of interest. The Application Programming Interface (API), (written or copied knowledgably by the user) in C++ provides a shell for integrating and managing all project extensions.

Two UAL manuals resembling this one have been produced previously:

- *UAL User Guide,* BNL-71010-2003, 2003. A manual containing the bare minimum of information needed to acquire and build the code and to run the examples. The example codes were intended to be close enough to a problem of interest to enable the user to perform other simulations by mimicking external inputs and PERL scripts. This manual remains applicable as a guide for using the PERL interface to UAL. However that interface is now deprecated as it has been replaced by a C++ interface. Since the physics underlying the examples has not been changed, browsing this manual to see the sorts of simulation supported may still be worthwhile. These examples are in the process of being upgraded to the C++ interface. The present manual largely supercedes this guide. The annotated descriptions of a "basic example" is contained in the present manual as an appendix.
- *Text for Accelerator Simulation Course.* Course guide for USPAS Accelerator School, 2005, Cornell University. This tutorial manual. describes and explains the physical methods used in some of the UAL modules. It supports "looking under the hood" at some of the underlying physics.

Technical documentation of the programming aspects of the UAL environment is also available `http://code.google.com/p/ual`. The programming language of UAL is (object-oriented) C++.

In this manual `typewriter font` is used to exhibit text that would be visible on a computer monitor, either because the text is being viewed by a text editor or because a program has generated it as output. *Italic font* is used when a term having a narrow technical sense is being used (often without immediate definition) or *for emphasis.* "Quotation marks" indicate either that a term is actually being defined or that it is being used loosely.

Of the many contributors, some to TEAPOT, some to UAL, the following deserve special mention: Chris Iselin (for establishing a style to be emulated), Lindsay Schachinger (for establishing the pattern and getting the project off the ground), Alexander Reshatov (for contribution to the UAL conceptual design), and George Bourianoff and Jie Wei (for enthusiastic support).

CHAPTER 2

A Superficial Introduction to C++

2.1. Introduction

Most accelerator simulation codes use a proprietary (i.e. specific to the particular code) input language to specify the calculation to be performed. UAL does not use this mechanism. Rather, the calculation to be performed is specified by a user-generated C++ main program. After specifying input parameters, this program calls upon those functions from the various UAL libraries that are appropriate for calculating the desired output quantities. One such example program, `shell_sns.cc`, is listed in full in Chapter 3, along with the files it reads in and the peripheral information needed to generate the executable code.

To generate such a C++ program from scratch would represent an almost insurmountable task for a novice user of UAL. Fortunately, the program `shell_sns.cc`, or a similar program, can serve as a *template* which can be adapted, one step at a time if desired, to the intended simulation task. As listed, `shell_sns.cc` assumes that the SNS accelerator is being analysed. To model some other accelerator one need only change the line in the code (line 21 in the example) that reads in the SNS lattice description file (`ff_sext_latnat.adxf`) and replace that file name by the file name of the desired accelerator.

After even such a minor change the code has to be recompiled. Barring a typo or an absent file, such a small change cannot cause the recompilation to fail. In the (distant) past such a recompilation step might have taken an unacceptably long time. But, nowadays, it takes only a few seconds. So, as a mechanism for specifying a simulation, the need for recompilation is not an impediment.

However, most changes, starting say from the `shell_sns.cc` template will be more substantial. Even for the change to another accelerator just mentioned, other changes will likely be needed to preserve consistency. For example, in lines 70 through 76 one sees that initial Twiss function values are being entered. These values will surely be inappropriate for any new simulation that is about to be performed.

The changes mentioned so far can be performed by a person completely ignorant of C++ (or of any other computer language). But most changes will be more substantial. In some cases, such as looping through parameter values in the calculation, a knowledge of the C language would be adequate—at such a "low" level, C and C++ are essentially identical. But, for more substantial changes, such as changing from adjusting the accelerator tune to adjusting the accelerator chromaticity, at least a modest understanding of C++ is needed. As well as needing to find the required code it is necessary to understand its *interface*; e.g. its input-argument and output-result passing syntax. Changes like this require only a modest understanding of C++.

Any serious simulation is likely to require changes more substantial than those mentioned so far. For example, the tune-changing algorithm might need to be altered, which might require *overriding* the tune-adjustment *method*.

And so on. One could continue this list to successively more and more challenging tasks. In practice, however, the level of C++ expertise required will stop well short of the expertise needed to establish the overall architecture (especially establishing the class structure) of an effective calculational environment.

Ideally the potential UAL user will already have had sufficient experience with C++, or a similar language such as Java, to proceed directly to hacking `shell_sns.cc` into a more useful form without stopping to improve their C++ skills. Individuals without this experience will have to acquire at least some C++ expertise.

Most books explaining C++ are far more sophisticated than what is really needed. One worthwhile source (especially because ROOT is important for UAL) is available at `ftp://root.cern.ch/root/doc/6ALittleC++.pdf`. It is Chaper 6, *A Little C++*, in the ROOT[1] documentation. Language features such as *Classes*, *Methods*, and *Constructors* are explained. Then higher level concepts such as *Inheritance* and *Data Encapsulation* are discussed. Then *Method Overriding* (which is especially appropriate for UAL) is mentioned. The chapter ends with discussion of the degree of permanence of objects created on the *stack* or on the *heap* and comparisons with FORTRAN.

Another book, *Teach Yourself C++ in 10 Minutes*, author Jesse Liberty, publisher Sams, seems appropriately elementary, clear, and adequately comprehensive. (The title is hyperbolic, even recognizing that 10 minutes means 10 minutes *per lesson* and there are 27 lessons.)

Little attempt will be made here to provide explanations parallel to those just mentioned. But, in the following sections, some of the important ideas will be discussed in the context of accelerator simulation, in particular within UAL.

2.2. Modularity and Code Organization

Modularity is a key attribute of any program whatsoever. If asked, any self-respecting code developer will boast that his or her code is nicely modular. This will be true no matter the computer language, irrespective of whether it is *procedural* or *object-oriented*. With more than one developer, different modules can be generated by different programmers. The essential requirement for successful modularity is the existence of well-defined *interfaces*. The most essential property of an interface is that it be unambiguous, and the next most is that it be clearly documented. Even a solitary programmer, with the interface committed only to memory, is, with time, likely to regret the absence of documentation.

The organization of an accelerator simulation program is simpler in some ways than the code for simulating a physical system such as an airplanes or a bridge. There is a natural starting point (the injection point), natural initial conditions (the parameters of an injected beam, or, for simplicity, let us say the coordinates of

[1]ROOT is a calculational environment developed at CERN and much used in the handling and processing of data from high energy physics experiments.

one injected particle), a natural problem, *pushing* the particle through the lattice, and a natural order, namely the order of elements in the accelerator.[2]

Because of this relative simplicity, and the limited number of calculationsl options, accelerator simulation codes have, until recently, been *procedural*. In simple form there can be a one-to-one correspondance between elements in the ring and mathematical algorithms for propagating a particle from input to output of each type of element. In the simplest (paraxial) case the algorithm can amount to matrix multiplication. Nothing could be more procedural than to multiply the *transfer matrices* of individual elements sequentially to obtain the transfer matrix of a complete lattice. In spite of this seemingly natural sequencing, the UAL code is object-oriented rather than procedural.

To discuss and justify this, it is useful to first introduce the UAL abstraction of general accelerator simulations. We call it the *element-algorithm-probe* analysis pattern. *Elements* are things like bending magnets, RF cavities, collimators, and so on. *Algorithms* are mathematical formulas capable of evolving quantities known at element inputs to their corresponding values at element outputs. So both *element* and *algorithm* are terms of common usage, likely to be understood unambiguously by all workers in the field. The term *probe*, because it is less standard, requires more explanation. A similar term is *observable*. As used in UAL a particle or beam bunch *probes* the lattice, element-by-element. Other things also probe the lattice. For example, the transfer map (identity at the origin) evolves into the transfer map from origin to element output as it evolves through an element. So a *probe* is anything whatsoever for which continuous evolution is meaningful and the evolution is unambiguously caused by the elements making up the lattice. It therefore makes sense to evaluate the evolution of a probe caused by the lattice. Examples of probes are 6D particle coordinates of all particles in a bunch, spin components, moments of a bunch of particles, lattice functions such as Twiss functions and dispersion functions, transfer matrices (i.e. linear maps), nonlinear maps (which are represented by truncated power series), wake fields, and so on.

As accelerators have become more and more complicated, and their particle dynamics more and more delicate, simulation codes have acquired more and more options. Propagation can be handled by linear matrices or by nonlinear maps or by numerical integration of the equations of motion. Beam wall and space charge forces may or may not be included. Execution speed may be at a premium (as for example in a control system) or if assurance of long term stability is needed. Different practioners may disagree on how best to model the situation. Large sectors may be represented by concatenated maps which handle propagation between specialized elements that need to be handled by, say, Runge-Kutta numerical integration.

In short, the simulation scene has acquired substantial complication. Lots of mixing and matching of simulation methods is called for. This places even greater demands, than those emphasized earlier, on the interfaces of the modules needed for effective simulations. It is this expanded complexity that makes it appropriate to switch from procedural to object-oriented computation.

[2]Though it is not particularly germane to the present discussion we will refer to accelerators as "closed" if they return to the starting point, as in a storage ring, or "open", if they do not, as in a linear accelerator or transfer line.

2.3. Object Orientation

Object orientation is usually said to have four essential design principles: *inheritance*, *encapsulation*, *polymorphism*, and *modularity*. Modularity has been discussed already and has been declared to be *a good thing*, not specific to object-orientation. The main purpose of the other three design principles is to make modularization more powerful and less error-prone.

An example of inheritance in an accelerator lattice is that one can introduce an abstract "lattice element" from which actual elements such as bending magnets, quadrupoles, RF cavities, collimators, and so on are "descended". They "inherit" from the abstract element the property of supporting propagation of probes from their inputs to their outputs. Once the interface of the abstract element has been established it can be inherited by all the descendants. This saves the effort required to produce a separate interface for each individual element type. But this time-saving is not the real benefit. The real benefit comes when it becomes necessary, as it always will, to modify or expand the interface requirements. With inheritance, changes in the code in one place automatically propagate to all descendants. This avoids the tedious and bug-prone effort in applying "the same" changes to the multiple, one-per-element, blocks spread through the code.

Inheritance reminds one of Darwinian evolution. A dog and a flea have common properties they have *inherited* from an abstract *animal* type. But they also have distinguishably different properties. (e.g. A dog can have fleas but a flea cannot have dogs.) If dogs and fleas had no distinguishing characteristics there would have been no point in having introduced the *animal* abstraction. So a computer language supporting inheritance must necessarily provide mechanisms for incorporating new characteristics or *overriding* inherited characteristic.

The real pay-off from inheritance is not the re-use of existing code (which was originally anticipated) but the improved logical organization of code, leading to briefer, more extensible, more modifiable and more easily maintainable code.

In the context of UAL the remaining two object-oriented features, *encapsulation* and *polymorphism*, are especially important, though for different reasons.

2.4. Encapsulation, Classes, and Methods

Most data comes in groupings of more than one component—complex numbers, vectors, enumerations, and so on are examples. Support for such groupings is a feature of all programming languages. In C they are called `structs` and they can even be promoted to be user-defined *types*, thereby extending the basic, built-in types such as characters, integers, and floats. The programmer can assign a single name to such a multi-component type and then, for most purposes, deal with it as a single entity, rather than having to treat its components individually.

This is a form of encapsulation but, in object-oriented terminology, the term *encapsulation* includes other features. After inclusion of these features the *types* are referred to as *classes*. As well as encapsulating data, the class encapsulates also *methods* that can access the encapsulated data and combine it with data from other classes to perform the calculations needed for the simulation. A member of a class is referred to as an *object*.

Even ancient forms of FORTRAN, without using the term *class*, contained an example, namely the complex variable *type*. For most purposes, such as complex

conjugation, a complex number can be treated as a single entity. This is accomplished by an encapsulated *method*. There also need to be *methods* providing the programmer with access to the real or imaginary parts. This class-like, complex number functionality is "built into" FORTRAN. (Even programmable calculators have this feature. Modern forms of FORTRAN are said to have full support for object-oriented code organization.)

As it happens, C++ has no such built-in complex number type. But the language provides a mechanism for user definition of classes. As a simple example, the user could introduce the class of complex numbers. This would be less convenient than in FORTRAN; the user would have to provide methods, for example to extract real and imaginary parts. Nevertheless this would be an entirely routine activity in C++.

An equally important aspect of encapsulation, in fact almost a synonym, is the concept of *data hiding*. As well as being *freed* from handling individual data components of an object, the user can be *forbidden* from accessing individual elements except by using methods provided by the class. Once a method of access has been produced (and debugged) users can be required to use only that method for that purpose. If some other processing is required a new method may need to be introduced into the class. This is a mechanism for "protecting one from oneself". Experience shows that constraining the user in this way vastly reduces the number of ways that "bugs" can be introduced into a program.

In UAL the most sophisticated classes are the *truncated power series*, `TPS`, provided by the DA (differential algebra) library. For example, the x_1 coordinate at the output of an element can be regarded as a function of the six input coordinates, $x_1(x_0, x_0', y_0, y_0', z_0, z_0')$. This function can be represented by a power series in the six arguments. For small amplitudes it is only the leading terms that are important. (For large amplitudes the representation breaks down because of chaos.) Actually, because it is natural to group all six of the output coordinates, UAL supports also vectors of truncated power series, `VTPS`. Theoretical discussion of the use of TPS's is contained in Appendix H.

The argument list could even be expanded from six to seven (or more) elements, to include other parameters, such as magnet strength of an element through which the particle is evolving. Though this functionality is supported by UAL it has rarely been used in practice.

In order to make encapsulation practical it is necessary for C++ to also incorporate *polymorphism*.

2.5. Polymorphism

A class of complex numbers (consisting of two numbers of type `real`) has already been mentioned. For a complex number A methods are available for calculating $\Re(A)$, $\Im(A)$, and complex conjugate A^*. But, for this to be at all useful one would have to be able to produce a product $A \times B$ of complex numbers A and B. In ordinary mathematics, when one sees $A \times B$ (or even AB) one accepts, without a second thought, that A and B can be either real or complex numbers. The same comments apply even to $A + B$. In mathematical terms, what is needed is an *algebra* of complex numbers.

A computer needs to be trained to support algebra like this. One way of supporting such algebra would be to introduce different names for the algebraic

operations, depending on whether reals or complexes were being operated on. (Matrix multiplication in MATLAB, MAPLE, and MATHEMATICA) is handled this way.)

Another feature is required in languages supporting *polymorphism*. The same syntax $A * B$ is used to specify the multiplication of A and B, irrespective of their types. For this to work there must necessarily be just one definition of the product. One says that operators like $*$ and $+$ are "overloaded". In ordinary mathematics using complex variables one is using overloaded operators without giving it a second thought. But, in a program, it is necessary for the programmer generating a class to also generate the code that implements the overloading. This code takes the form of operators encapsulated in the class.

Yet another feature of C++ is necessary to support this overloading. It is *strong typing*. In order for $A * B$ to give the intended result it is necessary for the program to know the types of both operands A and B. This allows the computer to pick out the code that implements the $*$ operation for those particular types. In C++, whenever a variable name of any type is declared, its type has to be specified explicitly. This is *strong typing*.

As mentioned previously UAL supports a very complicated form of polymorphism in which a dynamical variable can be represented either by a `real` (or, more typically, a `double`) or by a `TPS`. Only in this way is it possible to model particle propagation through nonlinear elements by truncated power series of arbitrary order. Naturally the code to support the overloading of even the multiplication operator $*$ is a *tour de force* of computer programming. A function (analytic at the origin so that a Taylor series exists) of a TPS can then itself be evalusted mechanically in the form of a TPS.

The VTPS containing the six TPS's that represent the six particle phase space coordinates is referred to as a "transfer map" (of pre-specified order). Such a nonlinear map can be used to model particle propagation through sectors, large or small, of an accelerator. Issues of symplecticity that arise are discussed in Appendix H.

2.6. Other Programming Terms

A random sampling of issues to be aware of follows. Any line numbers given refer to the listing of file `shell_sns.cc` in Section 3.1.2.

scope: The scope of variables is much as in C. But, for example at line 4, `using namespace UAL` implies that variables introduced in that file are global.

shell: The UAL shell (see line 8) is not to be confused with an operating shell such as *bash* or *csh*. But it plays a similar role, of making available whatever commands are available and defining a user world protected from the outside world, (and vice versa).

variable declaration: Every variable has to be declared before use. The declaration can be direct, as in line 69, or by using *pointer* * syntax or *address-of* & syntax.

variable type: "Strong typing" requires the *type* of every variable to be included with its declaration.

templates: This is a mechanism that permits the same coding to serve for multiple types.

function arguments: Argument passing rules are much as in C. The basic mechanism is "pass by argument value" and the result is given by the *return value*, which every function must have. Both argument type and return type have to be specified in the function declaration. Knowing only a value, a called function cannot change the value contained in the memory location from which the argument value was obtained; i.e. the variable whose value it is called with. But, results can be (and, except for single component types, usually are) returned by using *pointer **, or *address-of &* syntax. The called function can receive this address information in the argument list.

pointers * and address-of & operators: are not very intuitive. Here is an example from the *C++ Pocket Reference,* K. Louden, O'Reilly, 2003.

```
int i=20;
int *iptr = &i;
int j;
int k = 50;

j = *iptr; // This sets j to i.
*iptr = k; // This sets i to k.
```

One can always confirm behavior by trial and error.

position of *: does not matter.

```
int *i;
int* i;
```

are equivalent

address-of & and reference &: When declared as a type, &r is known as a "reference", as in

```
int i = 20;
int &r = i;
```

direct argument passing: as in lines 69-76 is the normal argument passing syntax in C++.

indirect argument passing: Most argument passing in the annotated example is *indirect*. The first example of this (UAL-specific) mechanism for argument passing, is in line 14. Introduced for UAL backward compatibility, this mechanism is explained further below.

declaration and definition: A declaration announces intended use. A definition allots memory. Usually declaration implies definition, for example because the variable being declared is initialized.

CHAPTER 3

Annotated Example

3.1. Evalustion of Lattice Functions, etc.

3.1.1. Explanations. As explained above, there is no proprietary command language for UAL simulations. Rather the command language is simply C++, and the directives are contained in a valid C++ main program. One such program, `shell_sns.cc` is given in Section 3.1.2. The present section provides line-by-line explanations for that code. These explanations assume the existence of the *environment variables* that are established by the `setup-ual-env` script; a partial (and likely to be out-dated) listing is given in Section 3.1.5. There are more environment variables listed there than are actually used for the example given in this chapter.

To get started one first obtains the code from `http://code.google.com/p/ual` where it is made available anonymously using `subversion`. The following directions are copied from the README file there:

```
Linux is the primary UAL development environment of the UAL
software at present. The UAL installation can be done in a few steps:

1. place the source code into the ual1 directory
2. cd ual1
3. setenv UAL ‘pwd‘
4. setenv UAL_ARCH linux
5. source setup-linux-ual
6. make >& make.log
```

The second last step takes linux-specific actions from `setup-linux-ual` and *sources* `setup-ual-env` file. The last step invokes the `Makefile` that builds the entire UAL library structure. Any errors in compilation would show up in `make.log`. Errors tagged as "ignored" can, indeed, be ignored.

The example documented in this chapter is found in `$UAL/examples/UI_Analysis`. Actually this directory contains both the original PERL version of the example (documented in Chapter D) and the up-to-date C++ version which performs (essentially) the same calculations. The C++ Makefile for this code is listed in Section 3.1.6 (but not explained in any detail here). The steps

```
cd $UAL/examples/UI_Analysis
make clean
make
```

builds the executable program `$UAL/linux/shell_sns`. In Section 3.1.7, is (part of) a table of output results in the form of lattice function values obtained when this code is executed. In this case the results correspond to SNS, the Spallation Neutron Source. The lattice functions themselves are plotted in Fig. 3.1.

At least a modest understanding of object-oriented programming and, in particular, C++, is assumed. In most cases, however, similar calculations for other lattices, or other calculations for the same lattice, can be performed by changing only a few lines. This can be done satisfactorily even with only a fuzzy understanding of what the code is actually doing. In this respect preparing the simulation directives is not very different, or very much more complicated, than preparing the "input deck" of the various available accelerator simulation programs. Of course, in order for the C++ program to run properly (or even compile), it is imperative that every line contain valid C++ code.

In general the description of any simulation to be performed by UAL is contained in three separate files. The first of these is the main program, in this case `shell_sns.cc`. Another file, in this case `ff_sext_latnat.adxf`, describes the lattice. This is purely a description of the physical elements making up the lattice; it contains no content concerning whatever accelerator algorithms are intended to be used in a simulation. The third file, is the so-called APDF (accelerator propagator description format) file, which associates an evolution algorithm with each element in the lattice. There are default assignments and, in the present simple example, because all element/algorithm links are assigned by default, there is no need for an `.apdf` file. However, to give a sense of the typical brevity of APDF files, an example is given in Section 3.1.4.

Digression concerning lattice description file formats. The two lattice description files in `$UAL/examples/UI_Analysis`, namely `ff_sext_latnat.adxf` and `ff_sext_latnat.mad` obviously describe the same lattice. In principle the `.adxf` file could have special purpose elements, or extended parameter sets, such as aperture sizes, but, for this example, all the `.adxf` elements used, with all their parameters, are identical to equivalent `.mad` elements.

There is a sense in which these representations are essentially different however. Because of the discipline imposed on `.adxf`, which necessarily *validates* against an XML schema, it is quite simple to use modern tools (XSLT) to transform an `.adxf` file into a `.mad` file, (using a transformer file called `adxf2mad.xsl`). Going the other way, from `.mad` to `.adxf` is much harder—it requires, for example, writing a special purpose `.mad` parser. It is similarly straightforward to transform an `.adxf` file into any other proprietary simulation input language though, of course, incompatabilities would require require special treatment.

The discussion in the previous paragraphs emphasized *design* or *idealized* lattices, that preserve some or all of the symmetries and repetitions of the idealized lattice design. For representing a functioning accelerator, with its imperfections, one needs a *fully-instantiated* lattice description, in which every parameter of every element has its own numerical value. These values are presumably close to, but typically not quite equal to the design values for that element. The ADXF lattice description is designed to fill this fully-instantiated role as well.

Historically, within UAL, there has been another fully-instantiated lattice description called SXF. Some of the UAL example codes use this form of lattice description. Though they are ASCII files, because they are fully-instantiated they are not at all user-friendly or user-editable. Furthermore, because they do not satisfy the discipline of an XML schema, it is not practical to transform from SXF to any other format. However it is easy to transform from ADXF to SXF using

`adxf2sxf.xsl`. It is also possible to transform a MAD file into an SXF file using the MAD-X accelerator simulation code.

Returning to the `UI_Analysis` example after this digression, the following annotations apply to the file `shell_sns.cc` in Section3.1.2. Line numbers from that file are followed by brief explanations of the purpose for those lines. There is considerable redundancy in that embedded comments also describe the functions of blocks of code.

3.1.1.1. *Annotations, line numbers are from* `shell_sns.cc` *in Section 3.1.2.*

2 The file "`UAL/UI/Shell.hh`" serves numerous purposes. It is not exhibited in this guide, but it would be a sensible file to browse after reading through this chapter. Its full address is `$UAL/ext/UI/src/UAL/UI/Shell.hh`. First this file includes other needed header files, for example by lines such as

```
#include "UAL/Common/Def.hh"
#include "UAL/APF/AcceleratorPropagator.hh"
#include "PAC/Beam/BeamAttributes.hh"

#include "Optics/PacVTps.h"
#include "Optics/PacTwissData.h"

#include "UAL/UI/Arguments.hh"
#include "UAL/UI/OpticsCalculator.hh"
#include "UAL/UI/BunchGenerator.hh"
```

The names of these files suggest the purposes for the classes they specify. The rest of file `UAL/UI/Shell.hh` supplies the interface for the `Shell` class, which provides the user interface of the code that follows. The actual implementation is in `$UAL/ext/UI/src/UAL/UI/Shell.cc`. While getting started there is no reason for the user to be aware of these files. Furthermore this code would not ordinarably be subject to change.

4 The namespace is specified as UAL. Identifiers from the "standard" library then have to be prefixed with `std::`, for example as `std::string` in line 11.

6 every C++ program has precisely one `main` function from which program execution commences. This program specifies the simulation to be performed. To invoke the program, from the command line, one simply enters `shell_sns` followed by carriage return.

8 declares the `shell` object whose methods are to be employed.

11 The `//` at line beginning converts the line to be a comment.

14 The method `shell.setMapAttributes(Args() << Arg(‘‘order’’, 5))` of the shell sets the maximum order of TPS's to 5. The curious argument structure `(Args() << Arg(‘‘order’’, 5))` supplies the actually needed arguments `‘‘order’’` and 5, indirectly, (as mentioned above). This indirection effectively replaces the PERL interface. The polynomial "order" is arbitrary but, because the amount of memory storage needing to be allotted increases strongly with increasing polynomial order, the

"order" argument will rarely exceed a fairly small integer value such as 10.

21 specifies the accelerator lattice description file. Normally this will be an ADXF (Accelerator Description Exchange Format) file (with suffix `.adxf`) that describes the lattice. See Section 3.1.3. The lattice description can also be in the form of an earlier, fully-instantiated, but now-deprecated SXF format, with `.sxf` suffix.

21+ would specify the APDF (Accelerator Propagator Description File), with suffix `.apdf` if one were needed. This file would link a particular evolution algorithm with each lattice element. In principle there is one entry for every element but, with entire categories of elements (specified either by element name or type) being processed by the same algorithm, this file is typically quite brief. See Section 3.1.4. For the present example the file is so brief as to be absent altogether.

27,30 Of the beamlines in the file `ff_sext_latnat.adxf`, the one that is actually to be analysed is called "ring". If there is no line by that name in the lattice description file the program will stop with an error message. Writing to memory the lattice elements with all their parameters is the province of `ual1/PAC/src/SMF/PacLattices.cc` in the PAC (Platform for Accelerator Codes) library. Line 15 of the Makefile makes this code available.

27-31 Most UAL propagation algoriths are based on numerical integration. (One exception is transfer matrix, linearized evolution, as for example in the TIBETAN model.) For the numerical integration in the present case all quadrupoles and sbends are assigned a "complexity index" of `ir=2.0`. (The symbol `ir`, (abbreviation for "intersection region") is a somewhat archaic usage deriving from intersection regions typically containing the elements requiring fine splitting.) The element slicing is not quite uniform but, the number of slices is `4 ir`. [1] Choice of elements to be sliced can be based on element type, as here. To link an elements of this type to a different algorithm one would link it by name, superceding the linkage by type.

37 selects `ring` as the lattice to be used, where "use" has much the same meaning as in MAD.

43 The lattice description is echoed as a check.

50 The beam information is made available to the `PAC/Beam` class. The same indirect argument passing as elsewhere is used.

60 generates six dimensional once-around transfer map for `ring`. With `order` argument being 1, only the linear transfer matrix is calculated. The results are written to `./out/cpp/map1`.

65 calculates the Twiss functions at all positions in `ring`. By default `shell.twiss` determines the initial Twiss function values (if the lattice is stable.) The results are written to `./out/cpp/ring.twiss`. Portions of this file are

[1] It is a fundamental UAL design principle that algorithmic information, such as the the number of slices, *not* be contained in the lattice description file. That is why the complexity index is provided in this control program. In practice this index would be adjusted empirically. At the modest cost of frequent recompilations this avoids the horror of artificially slicing elements in the lattice file and the proliferation of files having different slicing.

listed in Section 3.1.7. If the lattice were unstable this step would fail with an error message.

69 Object `tw` from class `PacTwissData` is declared. Its data is entered in the following lines.

70-76 In typical cases the purpose for this code will have already been achieved. But what if `rimg` were unstable? In lines 70-76 tentative starting values for the lattice functions are assigned. Since these are the exactly correct values, the subsequent calculation will just repeat the calculation performed previously.

79 propagates the lattice functions and writes them to `./out/cpp/tline.twiss`. This file can be compared to `./out/cpp/ring.twiss`.

3.1.2. Listing of Main Program `shell_sns.cc` .

```
2: #include "UAL/UI/Shell.hh"
3:
4: using namespace UAL;
5:
6: int main(){
7:
8:   UAL::Shell shell;
9:
10:   // ****************************************************************************
11:   std::cout << "\nDefine the space of Taylor maps." << std::endl;
12:   // ****************************************************************************
13:
14:   shell.setMapAttributes(Args() << Arg("order", 5));
15:
17:   // ****************************************************************************
18:   std::cout << "\nBuild lattice." << std::endl;
19:   // ****************************************************************************
20:
21:   shell.readADXF(Args() << Arg("file",  "./data/ff_sext_latnat.adxf"));
22:
23:   // ****************************************************************************
24:   std::cout << "\nAdd split ." << std::endl;
25:   // ****************************************************************************
26:
27:   shell.addSplit(Args() << Arg("lattice", "ring") << Arg("types", "Sbend")
28:   << Arg("ir", 2));
29:
30:   shell.addSplit(Args() << Arg("lattice", "ring") << Arg("types", "Quadrupole")
31:   << Arg("ir", 2));
32:
33:   // ****************************************************************************
34:   std::cout << "Select lattice." << std::endl;
35:   // ****************************************************************************
36:
37:   shell.use(Args() << Arg("lattice", "ring"));
38:
39:   // ****************************************************************************
40:   std::cout << "\nWrite ADXF file ." << std::endl;
41:   // ****************************************************************************
42:
43:   shell.writeADXF(Args() << Arg("file",  "./out/cpp/ff_sext_latnat.adxf"));
44:
46:   // ****************************************************************************
47:   std::cout << "\nDefine beam parameters." << std::endl;
48:   // ****************************************************************************
49:
50:   shell.setBeamAttributes(Args() << Arg("energy", 1.93827231)
51:    << Arg("mass", 0.93827231));
52:
53:   // ****************************************************************************
54:   std::cout << "\nLinear analysis." << std::endl;
```

```
55:   // ************************************************************************
56:
57:   // Make linear matrix
58:   std::cout << " matrix" << std::endl;
59:
60:   shell.map(Args() << Arg("order", 1) << Arg("print", "./out/cpp/map1"));
61:
62:   // Calculate twiss
63:   std::cout << " twiss (ring )" << std::endl;
64:
65:   shell.twiss(Args() << Arg("print", "./out/cpp/ring.twiss"));
66:
67:   std::cout << " twiss (transfer line)" << std::endl;
68:
69:   PacTwissData tw;
70:   tw.beta(0, 2.626);
71:   tw.alpha(0, 0.5705);
72:   tw.d(0, 1.406e-05);
73:   tw.dp(0, 0);
74:
75:   tw.beta(1, 12.32);
76:   tw.alpha(1, -2.26);
77:
79:   shell.twiss(Args() << Arg("print", "./out/cpp/tline.twiss"), tw);
82: }
```

3.1.3. Listing of the SNS ADXF File `sns.adxf`. The full SNS lattice description has many elements not shown here. But this excerpt is complete for a simplified version of the lattice. This file is in ADXF 2.0 format. Minor changes, especially to <`sector`>, < `frame`>, and < `mfield`> syntax, will be required for ADXF 3.0 format.

```
1: <?xml version="1.0" encoding="utf-8"?>
2: <adxf xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3:   xsi:noNamespaceSchemaLocation="file:/home/xslt/ADXF/adxf.xsd">
4:   <constants>
5:     <constant name="brho" value="5.6573735" />
6:     <constant name="qh" value="6.3" />
7:     <constant name="qv" value="5.8" />
8:     <constant name="muh" value="qh/4.0" />
9:     <constant name="muv" value="qv/4.0" />
10:     <constant name="bexd" value="2.428" />
11:     <constant name="beyd" value="13.047" />
12:     <constant name="o1l" value="6.85" />
13:     <constant name="o3l"  value="6.25" />
14:     <constant name="twopi" value="2*3.1415926536" />
15:     <constant name="ang"  value="twopi/32" />
16:     <constant name="ee" value="ang/2" />
17:     <constant name="lbnd" value="1.5" />
18:     <constant name="kf" value="4.66011/1.1981566" />
19:     <constant name="kd" value="-4.94176/1.1981566" />
```

```
20:     <constant name="kmat" value="-3.41118/1.1981566" />
21:     <constant name="ks2" value="4.29830/1.1981566" />
22:     <constant name="ks3" value="-4.58404/1.1981566" />
23:     <constant name="lq" value="0.5" />
24:     <constant name="lq1" value="0.25/2" />
25:     <constant name="lq2" value="0.7" />
26:     <constant name="lq3" value="0.55" />
27:     <constant name="lref" value="1.7" />
28: </constants>
29:   <elements>
30:     <sbend name="bl" l="lbnd/2" angle="ee" />
31:     <sbend name="bnd" l="lbnd" angle="ang" />
32:     <sbend name="br" l="lbnd/2" angle="ee" />
33:     <drift name="ldas" l="0.55" />
34:     <marker name="mcol1" />
35:     <marker name="mcol2" />
36:     <drift name="o1" l="o1l" />
37:     <drift name="o11" l="o1l/4" />
38:     <drift name="o2" l="0.4" />
39:     <drift name="o3" l="o3l" />
40:     <drift name="o31" l="o3l/5" />
41:     <drift name="oarc" l="1" />
42:     <drift name="oq1" />
43:     <drift name="oq2" />
44:     <drift name="oq3" />
45:     <drift name="orf" l="0.57" />
46:     <drift name="oz" />
47:     <quadrupole name="qdch" l="lq3/2" k1="ks3/brho" />
48:     <marker name="qdch1" />
49:     <marker name="qdch2" />
50:     <quadrupole name="qdh" l="lq/2" k1="kd/brho" />
51:     <marker name="qdh1" />
52:     <marker name="qdh2" />
53:     <quadrupole name="qdmh" l="lq/2" k1="kmat/brho" />
54:     <marker name="qdmh1" />
55:     <marker name="qdmh2" />
56:     <quadrupole name="qfbh" l="lq/2" k1="kf/brho" />
57:     <marker name="qfbh1" />
58:     <marker name="qfbh2" />
59:     <quadrupole name="qfh" l="lq/2" k1="kf/brho" />
60:     <marker name="qfh1" />
61:     <marker name="qfh2" />
62:     <quadrupole name="qflh" l="lq2/2" k1="ks2/brho" />
63:     <marker name="qflh1" />
64:     <marker name="qflh2" />
65:     <drift name="rf1" l="1.7" />
66:     <drift name="rf2" l="1.7" />
67:     <sextupole name="s1d" l="0.15" />
68:     <sextupole name="s2f" l="0.15" />
69:     <drift name="s30" l="0.3" />
70:     <sextupole name="s3d" l="0.15" />
71:     <sextupole name="s4f" l="0.15" />
72:     <marker name="mend" />
```

```
  </elements>
  <sectors>
    <sector name="qd" line="qdh1 qdh qdh qdh2" />
    <sector name="qf" line="qfh1 qfh qfh qfh2" />
    <sector name="qfb" line="qfbh1 qfbh qfbh qfbh2" />
    <sector name="qdm" line="qdmh1 qdmh qdmh qdmh2" />
    <sector name="qfl" line="qflh1 qflh qflh qflh2" />
    <sector name="qdc" line="qdch1 qdch qdch qdch2" />
    <sector name="ssx1d" line="s30 s1d ldas" />
    <sector name="ssx2f" line="s30 s2f ldas" />
    <sector name="ssx3d" line="ldas s3d s30" />
    <sector name="ssx4f" line="ldas s4f s30" />
    <sector name="ssx5d" line="ldas s1d s30" />
    <sector name="rfl1" line="orf orf rf1 orf orf rf1 orf" />
    <sector name="rfl2" line="orf rf1 orf orf rf2 orf orf" />
    <sector name="sc" line="qdm o11 o11 o11 o11 qfl o2 qdc o31 o31 o31 o31 o31" />
    <sector name="scm" line="o31 o31 o31 o31 o31 qdc o2 qfl o11 o11 o11 o11 qdm" />
    <sector name="scol" line="qdm o11 o11 mcol1 o11 o11 qfl o2 qdc o31 o31 mcol2 o31 o31 o31" />
    <sector name="scolm" line="o31 o31 o31 o31 o31 qdc o2 qfl o11 o11 mcol1 o11 o11 qdm" />
    <sector name="scrf" line="qdm o11 o11 o11 o11 qfl o2 qdc rfl1" />
    <sector name="scrfm" line="rfl2 qdc o2 qfl o11 o11 o11 o11 qdm" />
    <sector name="acf" line="oarc bnd oarc qf" />
    <sector name="acfm" line="oarc bnd oarc qd" />
    <sector name="acs1" line="ssx1d bnd oarc qfb" />
    <sector name="acs2" line="ssx2f bnd ssx3d qd" />
    <sector name="acs3" line="oarc bnd ssx4f qfb" />
    <sector name="acs4" line="oarc bnd ssx5d qd" />
     <sector name="acfl" line="oarc bnd oarc" />
     <sector name="arc" line="acf acfm acs1 acs2 acs3 acs4 acf acfl" />
     <sector name="insert" line="sc oz scm" />
     <sector name="insertc" line="scol oz scolm" />
     <sector name="insertr" line="scrf oz scrfm" />
     <sector name="sp" line="insert arc" />
     <sector name="spc" line="insertc arc" />
     <sector name="spr" line="insertr arc" />
     <sector name="ring" line="sp sp sp sp mend" />
   </sectors>
</adxf>
```

3.1.4. Sample APDF Element-Algorithm Linkage File `tracker.adxf`. This file is not actually used by the example discussed in this chapter.

```
1: <apdf>
2:   <propagator id="svd" accelerator="ring">
3:      <create>
4:  <link algorithm="TEAPOT::DriftTracker"       types="Default" />
5:         <link algorithm="TEAPOT::DriftTracker"    types="Marker|Drift" />
6:         <link algorithm="TEAPOT::DipoleTracker"   types="Sbend" />
7:         <link algorithm="TEAPOT::MltTracker"      types="Quadrupole|Sextupole|[VH]kicker"/>
8:         <link algorithm="TEAPOT::RFCavityTracker" types="RfCavity"/>
9:         <link algorithm="AIM::Monitor"            types="[VH]monitor" />
10:        <link algorithm="AIM::PoincareMonitor"    types="Monitor" />
11:      </create>
12:   </propagator>
13: </apdf>
```

3.1.5. Partial Listing of `setup-ual-env`. The file `setup-ual-env` mainly establishes environment variables and the executable and library look-up paths. Only a partial listing is given here, to show the general character of the file. Most of the environment variables are superfluous to the example discussed in this chapter. These environment variable are only useful to the user who is "looking under the hood" or tracing the execution path. In that case one should look at the file itself. One can also browse the initializing `setup-linux-ual` file.

```
#################################################################
# Core Libraries
#################################################################

setenv UAL_CORE $UAL/codes/UAL
setenv LD_LIBRARY_PATH $UAL_CORE/lib/$UAL_ARCH/:$LD_LIBRARY_PATH

# OPTICALC - Fast tracking program
setenv UAL_OPTICALC    $UAL/codes/OPTICALC
setenv LD_LIBRARY_PATH $UAL_OPTICALC/lib/$UAL_ARCH/:$LD_LIBRARY_PATH
...
# ZLIB - Numerical Library for Differential Algebra
setenv UAL_ZLIB        $UAL/codes/ZLIB
setenv LD_LIBRARY_PATH $UAL_ZLIB/lib/$UAL_ARCH/:$LD_LIBRARY_PATH

# PAC - Set of containers with the element's parameters
setenv UAL_PAC         $UAL/codes/PAC
setenv LD_LIBRARY_PATH $UAL_PAC/lib/$UAL_ARCH/:$LD_LIBRARY_PATH

# TEAPOT - element by element tracker (with errors)
setenv UAL_TEAPOT          $UAL/codes/TEAPOT
setenv LD_LIBRARY_PATH $UAL_TEAPOT/lib/$UAL_ARCH/:$LD_LIBRARY_PATH

# ORBIT - objective ring injection and tracking code
```

```
setenv UAL_ORBIT           $UAL/codes/ORBIT
setenv LD_LIBRARY_PATH $UAL_ORBIT/lib/$UAL_ARCH/:$LD_LIBRARY_PATH

setenv UAL_ACCSIM          $UAL/codes/ACCSIM
setenv LD_LIBRARY_PATH $UAL_ACCSIM/lib/$UAL_ARCH/:$LD_LIBRARY_PATH
...
# TIBETAN - longitudinal phase space tracking program
setenv UAL_TIBETAN           $UAL/codes/TIBETAN
setenv LD_LIBRARY_PATH  $UAL_TIBETAN/lib/$UAL_ARCH/:$LD_LIBRARY_PATH
...
# SIMBAD - simulation of beam advanced dynamics
setenv UAL_SIMBAD       $UAL/codes/SIMBAD
setenv LD_LIBRARY_PATH  $UAL_SIMBAD/lib/$UAL_ARCH/:$LD_LIBRARY_PATH

# DA Runge-Kutta and Lie Integrators
setenv UAL_DA              $UAL/codes/DA

#############################################################
# Extensions
#############################################################

setenv UAL_EXTRA $UAL/ext

setenv UAL_ROOT $UAL_EXTRA/ROOT
setenv UAL_SXF  $UAL_EXTRA/SXF
...
#############################################################
# Tools
#############################################################

setenv LD_LIBRARY_PATH $UAL/tools/lib/$UAL_ARCH/:$LD_LIBRARY_PATH
set path=($UAL/tools/bin/$UAL_ARCH/ $path)

# XERCESCROOT
setenv XERCESCROOT $UAL/tools/tars/xerces-c-src_2_6_0

# LIBXML2

setenv UAL_LIBXM2 $UAL/tools/include/libxml2
# setenv UAL_LIBXM2 /usr/include/libxml2
# setenv LD_LIBRARY_PATH $LD_LIBRARY_PATH/:/usr/lib

# SXF

setenv SXF $UAL/tools/sxf/
setenv SXF_ARCH $UAL_ARCH
setenv LD_LIBRARY_PATH $SXF/lib/$SXF_ARCH/:$LD_LIBRARY_PATH
```

```
# MPI

setenv UAL_MPI_PERL $UAL/tools/shortmpi

# CORAANT - COmprehensive Root-based Accelerator ANalysis Toolkit
setenv CORAANT $UAL/tools/coraant
setenv LD_LIBRARY_PATH $CORAANT/lib/$UAL_ARCH/:$LD_LIBRARY_PATH
...
```

3.1.6. Makefile for `shell_sns.cc`.

```
1: CC       = g++                       # Compiler
2: CCFLAGS  = -O -fpic -ansi -Wall      # Compilation flags
3:
4: DLD      = g++                       # Dynamic linker
5: DLDFLAGS = -shared                   # Dynamic linker flags
6: LD       = g++                       # Linker
7: LDFLAGS  =                           # Linker flags
8:
9: INC += -I. -I./src
10: INC += -I$(UAL)/ext/UI/include
11: INC += -I$(UAL_SXF)/src
12: INC += -I$(UAL_SIMBAD)/src
13: INC += -I$(UAL_ACCSIM)/src
14: INC += -I$(UAL_TEAPOT)/src
15: INC += -I$(UAL_PAC)/src
16: INC += -I$(UAL)/codes/UAL/include
17: INC += -I$(UAL_ZLIB)/src
18: INC += -I$(UAL)/tools/include
19: INC += -I$(SXF)/src
20: INC += -I/home/ual/tools/gsl/include
21:
22: LIBS += -L$(UAL)/ext/UI/lib/$(UAL_ARCH)   -lUalUI
23: LIBS += -L$(UAL_SIMBAD)/lib/$(UAL_ARCH)   -lSimbad
24: LIBS += -L$(UAL_TIBETAN)/lib/$(UAL_ARCH)   -lTibetan
25: LIBS += -L$(UAL_SXF)/lib/$(UAL_ARCH)  -lUalSXF2
26: LIBS += -L$(UAL_ACCSIM)/lib/$(UAL_ARCH)  -lAccsim
27: LIBS += -L$(UAL_TEAPOT)/lib/$(UAL_ARCH)  -lTeapot
28: LIBS += -L$(UAL_PAC)/lib/$(UAL_ARCH)   -lPacSMF -lPacOptics -lPac
29: LIBS += -L$(UAL_ZLIB)/lib/$(UAL_ARCH)  -lZTps
30: LIBS += -L$(UAL)/codes/UAL/lib/$(UAL_ARCH)  -lUal
31: LIBS += -L$(SXF)/lib/$(UAL_ARCH)  -lSXF
32: LIBS += -L$(UAL)/tools/lib/$(UAL_ARCH)  -lpcre -lxml2 -lrfftw -lfftw
33: LIBS +=  -lm
34:
35: SRC_DIR  = .
36: OBJ_DIR  = ./lib/$(UAL_ARCH)/obj
```

```
37:
38: OBJS =
39:
40:
41: compile : $(OBJS) ./shell_sns
42:
43: clean :
44:  rm -f $(OBJS) ./shell_sns
45:
46: ./shell_sns  : $(OBJ_DIR)/shell_sns.o $(OBJS)
47:  $(LD) -o $@  $(LDFLAGS) $(INC) $(OBJ_DIR)/shell_sns.o $(OBJS) $(LIBS)
48:
49:
50: $(OBJ_DIR)/%.o : $(SRC_DIR)/%.cc
51:  $(CC) $(CCFLAGS) $(INC) -c $< -o $@
```

3.1.7. Output Listing `ring.twiss`.

```
#     name         suml            betax      alfax      qx         dx         betay      alfay      qy         dy
-------------------------------------------------------------------------------------------------------------------
    0 mbegin       0.0000000e+00  2.626e+00  5.705e-01  0.000e+00  1.406e-05  1.232e+01 -2.260e+00  0.000e+00  0.000e+00
    1 qdmh1        0.0000000e+00  2.626e+00  5.705e-01  0.000e+00  1.406e-05  1.232e+01 -2.260e+00  0.000e+00  0.000e+00
    2 qdmh         0.0000000e+00  2.626e+00  5.705e-01  0.000e+00  1.406e-05  1.232e+01 -2.260e+00  0.000e+00  0.000e+00
    3 qdmh         2.5000000e-01  2.450e+00  1.406e-01  1.580e-02  9.637e-06  1.307e+01 -7.233e-01  3.120e-03  0.000e+00
    4 qdmh2        5.0000000e-01  2.482e+00 -2.714e-01  3.204e-02  5.522e-06  1.303e+01  9.033e-01  6.154e-03  0.000e+00
    5 o11          5.0000000e-01  2.482e+00 -2.714e-01  3.204e-02  5.522e-06  1.303e+01  9.033e-01  6.154e-03  0.000e+00
    6 o11          2.2125000e+00  4.680e+00 -1.012e+00  1.158e-01 -2.193e-05  1.034e+01  6.646e-01  2.972e-02  0.000e+00
    7 o11          3.9250000e+00  9.416e+00 -1.753e+00  1.574e-01 -4.938e-05  8.473e+00  4.258e-01  5.901e-02  0.000e+00
    8 o11          5.6375000e+00  1.669e+01 -2.494e+00  1.792e-01 -7.683e-05  7.423e+00  1.871e-01  9.365e-02  0.000e+00
    9 qflh1        7.3500000e+00  2.650e+01 -3.235e+00  1.921e-01 -1.043e-04  7.192e+00 -5.171e-02  1.313e-01  0.000e+00
   10 qflh         7.3500000e+00  2.650e+01 -3.235e+00  1.921e-01 -1.043e-04  7.192e+00 -5.171e-02  1.313e-01  0.000e+00
   11 qflh         7.7000000e+00  2.669e+01  2.693e+00  1.942e-01 -1.058e-04  7.820e+00 -1.791e+00  1.388e-01  0.000e+00
   12 qflh2        8.0500000e+00  2.292e+01  7.804e+00  1.964e-01 -9.915e-05  9.831e+00 -4.102e+00  1.453e-01  0.000e+00
   13 o2           8.0500000e+00  2.292e+01  7.804e+00  1.964e-01 -9.915e-05  9.831e+00 -4.102e+00  1.453e-01  0.000e+00
   14 qdch1        8.4500000e+00  1.711e+01  6.724e+00  1.996e-01 -8.702e-05  1.340e+01 -4.827e+00  1.508e-01  0.000e+00
   15 qdch         8.4500000e+00  1.711e+01  6.724e+00  1.996e-01 -8.702e-05  1.340e+01 -4.827e+00  1.508e-01  0.000e+00
   16 qdch         8.7250000e+00  1.438e+01  3.364e+00  2.025e-01 -8.084e-05  1.543e+01 -2.415e+00  1.538e-01  0.000e+00
...
  420 bnd          2.4150000e+02  4.137e+00 -1.075e+00  6.132e+00  8.067e-01  8.490e+00  1.812e+00  5.572e+00  0.000e+00
  421 oarc         2.4300000e+02  8.281e+00 -1.652e+00  6.173e+00  5.396e-01  4.190e+00  1.055e+00  5.612e+00  0.000e+00
  422 qfh1         2.4400000e+02  1.203e+01 -2.102e+00  6.189e+00  4.661e-01  2.584e+00  5.507e-01  5.661e+00  0.000e+00
  423 qfh          2.4400000e+02  1.203e+01 -2.102e+00  6.189e+00  4.661e-01  2.584e+00  5.507e-01  5.661e+00  0.000e+00
  424 qfh          2.4425000e+02  1.257e+01 -2.341e-02  6.193e+00  4.378e-01  2.445e+00  1.208e-02  5.677e+00  0.000e+00
  425 qfh2         2.4450000e+02  1.206e+01  2.059e+00  6.196e+00  3.908e-01  2.572e+00 -5.245e-01  5.693e+00  0.000e+00
  426 oarc         2.4450000e+02  1.206e+01  2.059e+00  6.196e+00  3.908e-01  2.572e+00 -5.245e-01  5.693e+00  0.000e+00
  427 bnd          2.4550000e+02  8.374e+00  1.624e+00  6.212e+00  1.678e-01  4.116e+00 -1.020e+00  5.743e+00  0.000e+00
  428 oarc         2.4700000e+02  4.271e+00  1.075e+00  6.252e+00  3.253e-05  8.293e+00 -1.764e+00  5.784e+00  0.000e+00
  429 mend         2.4800000e+02  2.626e+00  5.705e-01  6.300e+00  1.406e-05  1.232e+01 -2.260e+00  5.800e+00  0.000e+00
```

3.1.8. Graphical Output. Tables of lattice function like the one just shown in Section 3.1.7 are needed for many purposes. *Plots* of the main functions are typically needed as well. There are any number of plotting programs that can produce graphs from such tables. But one needs a plotting procedure that is quite tightly coupled to the simulation code so that normal choices are handled by default, yielding automatic graphical output, with standard formatting, of output results. Such coupling can free the occasional user from the distraction (and likely error) associated with figuring out how to plot the results of the simulation. Of course in detailed practice it is also essential to be able to produce non-standard graphs.

The text for the UAL course at the USPAS school at Cornell in 2005[**8**] *Text for UAL Accelerator Simulation Course,* by Malitsky and Talman describes a GUI geared to the tutorial requirements of the course. (Much of the *theory* underlying the UAL code is also documented in that text.) This GUI gives immediate pictorial feedback demonstrating the effect of changing one or more parameters. While this GUI was a very useful pedagogical tool, it was specialized to a few tasks and is not easily generalized to supporting arbitrary simulation projects.

To preserve flexibilty and to reduce code maintenance overhead, a different mechanism is used to produce graphical output. The USPAS GUI used the CERN ROOT environment (also based on C++) for intermediate calculations and graph plotting. ROOT was tightly integrated, with calls to ROOT compiled into the UAL code. As was true of the USPAS GUI, the UAL graph plotting is still handled within the ROOT environment. But the communication between UAL and ROOT will be via shell scripts. Though this mechanism is not as flashy as one sometimes sees with GUI's, it will provide the needed functionality.

The decoupling of ROOT from UAL also makes it possible to use tools other than ROOT for any required post-processing. or plotting, for example to generate moving pictures.

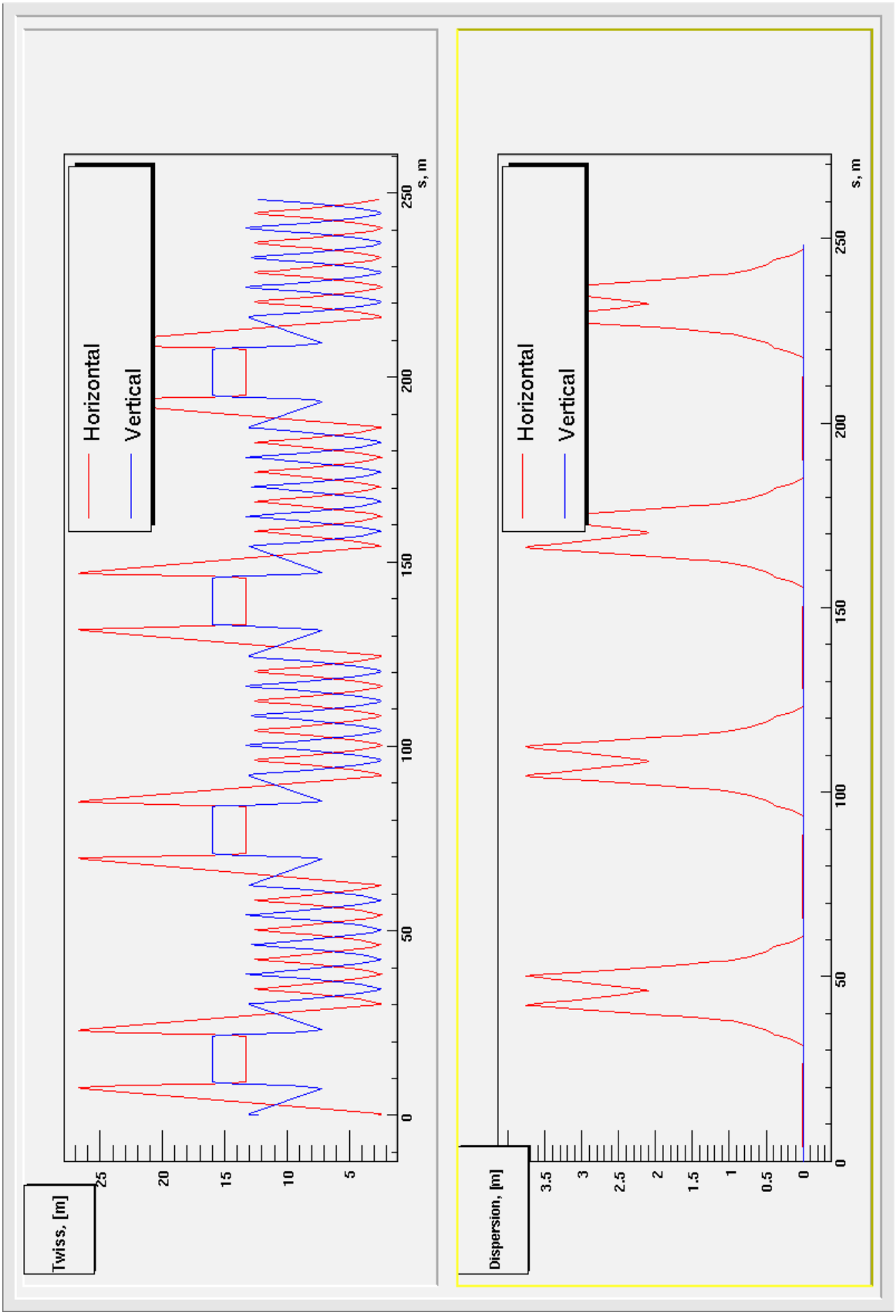

FIGURE 3.1. Beta functions for SNS, the Spallation Neutron Source.

Appendices

CHAPTER D

Annotated Examples (PERL-interface)

This appendix is a copy (unaltered except for minor updating) of a chapter in the earlier PERL-interface documentation. PERL-interface examples are now deprecated and are in the process of being replaced by C++interface examples. At this time the examples documented in this appendix run as described. They document features not documented in the C++ examples given earlier. This documentation will be updated and moved to the body of this report as the examples are upgraded to the C++interface.

The simulation scripts for this manual are mainly contained in sub-directories of the directory `$UAL/examples`.[1] Of these, the first, to be referred to as the "basic example", is the most important as it includes most of the commands that would be present in any simulation. Explanations of routine steps in this example will not be repeated in subsequent examples. The first few examples are closely related to the basic example; they represent an evolution from using the code as delivered, through making minor changes, to establishing a personalized, independent environment that can be used to reliably perform more extreme surgery on the delivered code, and finally to incorporating maps for special elements. These examples are all based on the same SNS lattice. Later examples illustrate some of the special features of UAL, and introduce other accelerators. Since only code fragments are exhibited in this manual there will inevitably be "loose ends" (apparently undefined quantities for example) that the reader can only sort out up by bringing up the complete code in a text editor.

Starting lattice files and other input information are in directories named `data`. For SNS the starting file is `$UAL/examples/UI/data/ff_sext_latnat.mad`. There are also RHIC and LHC examples based on SXF input.[2]

In using a computer code having any flexibility whatsoever, a certain amount of detective work is required of the user. Apart from getting help from an expert or trial-and-error running of the code—which one will always resort to eventually for confirmation—there are two sources of information: the (often ambiguous or obscure, like the present guide) documentation; or the source code (assuming it has been found). Even though the source code is definitive, many users will be unwilling to follow the latter route.

[1] There are example scripts in directories other than `$UAL/examples`. They are available for browsing and testing, and they function correctly when run, but they may rely on specialized shell features that are considered to be obsolete.

[2] Some scripts involving RHIC have complicated input descriptions that cannot be traced because they are derived using RHIC databases and dedicated and complicated "filters" not included in the UAL distribution. These descriptions, which are also Perl scripts, have to be regarded as yet another (though archaic) input format.

Because UAL supports the flexibility of a multiplicity of codes the problem of documentation is all the more acute. But UAL is structured in such a way that there is a source of information intermediate between text documentation (such as this manual) and C++ source code. It is the Perl code of the numerous examples included in the distribution. Like source code, this source of information is, by definition, accurate. The serious user of UAL has no real alternative to looking at this Perl code to figure out how the environment is to be used. The annotation supplied with the following examples is intended to provide help along this route.

In an ideal world the UAL code would be architecturally and stylistically homogeneous, to improve its intelligibility. In fact, having evolved over quite a few years, the organization of recently developed code may seem to be, and is, stylistically inconsistent with modules developed earlier. For example most of the examples in this guide do not need to rely on the Element-Algorithm-Probe framework mentioned in a lengthy footnote in section 4.5.2. The prime purpose of this framework is to support *extensions,* which are typically quite specialized. It is expected that mastery of the material in the present guide will help to make using these more advanced applications straightforward. The recently completed `FastTeapot` code, explained in section 4.5.2, is an example of such an extension. Installation and checkout of another extension, MPI (Message Passing Interface,) is described in the PERL interface User's Guide. Working accelerator simulation examples that use multiple processors will be included in the distribution shortly, and a simulation of injection "painting" will also be included.

The online C++ documentation is currently being mechanically upgraded using *doxygen.*

4.1. Basic Example (PERL)

To run the first example, a simulation of SNS with realistic errors, enter

```
$ cd \$UAL/examples/UI; perl shell_sns.pl
```

expecting to see the following output [3] [4]:

[3] The numbers in parenthesis are line numbers in the Perl script. These numbers were arranged to be present for this particular script just for purposes of reference in this manual. (See `__LINE__ in code fragment` ① .) Normal scripts would not issue these numbers. For reference elsewhere in the manual code fragments are identified by circled numbers like ① , along with the number of the section the fragment appears in.

[4] To follow the code more closely and examine values of relevant variables it is possible to step through the Perl script line-by-line using a Perl debugger, which is invoked by `$ perl -d shell_sns.pl` or `$ ddd --perl shell_sns.pl`.

```
mkdir ./out/test
Create the ALE::UI::Shell instance (15)
Define the space of Taylor maps (23)
Read MAD input file (33)
Define aperture parameters (46)
Select and initialize a lattice (78)
Define beam parameters (92)
Linear analysis (100)
Add systematic errors (119)
Add random errors (141)
Track bunch of particles (175)
End (202)
```

These lines are issued by the code to indicate its progress, as can be seen by correlating with line numbers in the script `shell_sns.pl`, fragments of which will be displayed in this section (with blank lines and comments suppressed), starting with the first several lines which are listed next: (1)

```
#!/usr/bin/perl #1
my $job_name   = "test"; #3
use File::Path; #5
mkpath(["./out/" . $job_name], 1, 0755); #6
use lib ("$ENV{UAL_EXTRA}/ALE/api"); #12
use ALE::UI::Shell; #13
print "Create the ALE::UI::Shell instance (", __LINE__, ")\\n"; #15
my $shell = new ALE::UI::Shell("print" => "./out/" . $job_name . "/log"); #17
```

Here line numbers in the script have been appended to each line as `#1, #3`, etc. These numbers are included for reference purposes in this manual; they would not normally be present but, as it happens, since they are expressed as comments (everything following `#` on the same line) their presence would have no affect on the script. The substantial accomplishment of (1) is contained in line `#17` which has defined the "user shell" to be an `ALE::UI::Shell`. To find this, line `#12` instructs Perl to look in `$UAL_EXTRA/ALE/api`, which is to say, of `$UAL/ext/ALE/api`, relative to which the script pathname is `ALE/UI/Shell.pm`.[5] This script initializes many UAL objects: *shell, code, lattice, beam, orbit, bunch, space, map, twiss,log,* and *constants,* all objects that the shell needs to access as Perl commands are interpreted and executed. Most of this will be spelled out more fully below.

The next statement in `shell_sns.pl` declares the maximum order of truncated power series to be 5, (also known as "dodecupole order"); (2)

```
$shell->setMapAttributes("order" => 5); #25
```

and statement (3) specifies the lattice input file;

```
$shell->readMAD("file" => "./data/ff_sext_latnat.mad"); #35
```

[5]The `ALE::UI::Shell.pm` script itself depends on other libraries that it accesses with the command use lib { ..., "ENV{$UAL_ZLIB}/api", "ENV{$UAL_TEAPOT}/api", "ENV{$UAL_EXTRA}/PAC/api" };. These libraries, for example, make UAL tools such as power series capability available. These capabilities will not be discussed at this point. It is more appropriate to defer this discussion to section 4.4.1 where the map generation features of UAL are exercised.

In this case the lattice is described by the mad file `ff_sext_latnat.mad`, a few lines of which are excerpted next, to give a general idea of its content: (4)

```
!  ff_sext_latnat.mad
   ...
   Brho := 5.6573735 ! 1.0 GeV (for 1.3 GeV: factor 1.1981566)
   ...
   lbnd := 1.5
   lq := 0.5
   LSEX:=0.15
   LREF:=1.7
   ...
   O3 : DRIFT, L = 6.25
   O31 : DRIFT, L = O3[L]/5
   ...
   ANG:= 2*PI/32
   EE := ANG/2
   BL: Sbend, L=lbnd/2, Angle=EE, E1=0., E2=0.
   BR:  Sbend, L=lbnd/2, Angle=EE, E1=0., E2=0.
   BND: Sbend, L=lbnd, Angle=ANG, E1=0.0, E2=0.0
   ...
   KD:=-4.94176/1.1981566
   QDH : QUADRUPOLE, L = lq/2, K1 = KD/Brho ! focusing arc quad (21Q40)
   QFH : QUADRUPOLE, L = lq/2, K1 = KF/Brho ! defocusing arc quad (21Q40)
   QFBH : QUADRUPOLE, L = lq/2, K1 = KF/Brho ! large focusing arc quad
   QDMH : QUADRUPOLE, L = lq/2, K1 = KMAT/Brho ! "matching" quad (21Q40)
   ...
   ...
   QDH1 : MARKER
   QD : LINE = (QDH1,QDH,QDH,QDH2)
   QF : LINE = (QFH1,QFH,QFH,QFH2)
   ...
   VS1D:=-2.891275
   S1D:   SEXTUPOLE, L=LSEX, K2=VS1D
   ...
   RF1 : RFCAVITY, L = LREF, HARMON = 1, VOLT =  0.0
   ...
   ARC    : line = (ACF,ACFM,ACS1,ACS2,ACS3,ACS4,ACF,ACFL)
   INSERT : line = (SC,OZ,SCM)
   SP : line = (INSERT,ARC)
   RING : line = (SP,SP,SP,SP)
```

Because this UAL example reflects the realistic complexity of an actual (SNS) lattice, the script is fairly long so, for brevity in this manual, lines that are pedagogically repetitive will not be shown. The line numbers shown refer to the actual file. The reader is expected to read along in the actual file using a text editor, tolerant of minor line numbering disagreements that may have occurred due to reformatting or line-wrapping.

Since particle tracking is to be done by numerical integration, it is necessary to specify integration intervals. The intervals can be full elements, or quarter

elements (¨ir¨ => 1) or eighth elements (¨ir¨ => 2), and so on. [6] It is often adequate to treat entire quadrupoles as single kicks, especially when the quads are, in fact, represented as paired half-quads in the lattice description, as is the case in the lattice being studied. For some purposes overly fine splitting is "apple polishing", unjustified by the accuracy of the element descriptions. But for the exact comparison of results from different codes a fine subdivision is appropriate when computation time is not an issue. Deciding the extent to which any particular element is to be subdivided belongs to the domain of the tracking engine to be used rather than to the domain of the lattice description. This is why the subdivision has to be taken care of in the present Perl script. To specify which elements are to be subdivided the regular expression mechanism is used: (5)

```
$shell->addSplit("elements" => "^(q[df]h|q[fd][lmc]h|qfbh)\$", "ir" => 2);  #40
$shell->addSplit("elements" => "^(bnd)\$", "ir" => 2);  #42
...
```

The cryptic expression ¨^(q[fd][lc]h|qfbh)$¨ is the regular expression specifying the quadrupoles that are to be subdivided into eighths (because ¨ir¨ => 2).

Only enough comments will be made about this process to give the general idea of how regular expressions work. (i) For compatability with most simulation codes, UAL element names are case-insensitive. Uppercase names are converted to lowercase before regular expression matching. Hence, for example, "QFBF" becomes "qfbh". (ii) The symbol ¨^¨ forces name matching to start at the beginning of the name. So the ¨^(q...¨ part of the regular expression allows names beginning with "q" to be candidates for inclusion (though they must also pass later tests). (iii) The symbol ¨...$¨ forces name matching to end at the end of the name. So the ¨^(...)$¨ part of the regular expression limits both the beginning and ending match. (iv) The part of the regular expression (... |qfbh) gives qfbh as one of the strings whose presence makes the name a candidate for inclusion. This (along with the requirements already mentioned) shows that the element "QFBH" from the lattice file (4) will be one of the magnets to be subdivided; its name (after conversion to lower case) includes the string "qfbh" and matches both at beginning and end. This is an example of a "last resort" explicit name inclusion in that element name "QFBH" is *really* specified explicitly (not counting upper/lower case.) (v) The regular expression also allows other matches, though all must start with q and end with h. The portion [fd] allows the second character of a three character name to be either f or d, so quadrupole names QDH and QFH also match. (vi) The portion [fd][lmc] admits certain 4 character names, such as QDMH. Putting it all together, it turns out that all quads exhibited in (4) will be subdivided. If there were a non-quadrupole (say an RF cavity, for which subdivision makes no sense) with name QFCH, the name would match the regular expression but the processing algorithm would just ignore the selection. Nevertheless, to reduce the likelihood of

[6] The name "ir" is archaic; it derives from the common circumstance that intersection region quadrupoles are most sensitive and most in need of fine subdivision. In future this usage will be specific to TEAPOT and a corresponding parameter will be known as a "complexity index" or as a "divisibility index". Different simulation modules may interpret this parameter differently. The fourfold subdivision is used by TEAPOT because it permits a near-optimal, unequal-interval kick algorithm, somewhat more accurate than uniform interval splitting. The same unequal interval splitting is available in MAD. Unlike any thick element truncated map, these algorithms preserve exact symplecticity.

error, it makes sense to maintain naming conventions such as "all quadrupoles, and only quadrupoles, have names beginning with Q".

If vacuum chamber dimensions are important, they should, because they are installed hardware, be included in the lattice description file. But, since the SIF description language does not support this, it is necessary to introduce aperture information (for bends, quads, and sextupoles) via the present script: (6)

```
$shell->addAperture("elements" => "^(bnd)\$",
                    "aperture" => [1, 0.116, 0.079]); #51
$shell->addAperture("elements" => "^(q[fd]h|qdmh)\$",
                    "aperture" => [1, 0.105, 0.105]); #56
$shell->addAperture("elements" => "^(s[24][fd])\$",
                    "aperture" => [1, 0.13, 0.13]);
 ...
```

The aperture attributes are `1` (ellipse) with half-widths and half-heights as shown.

Such apertures have no effect on analytical lattice calculations, but in particle tracking they correctly model the loss of particles that would strike a vacuum chamber wall.

At this point all lattice description is complete, though not yet fully specific in the sense that options remain as to what constitutes the actual beamline to be investigated. The next command fixes this: (7)

```
$shell->use("lattice" => "ring"); #82
$shell->writeFTPOT("file" => "./out/" . $job_name . "/tpot"); #86
```

Here line `#86` has also output a file `$UAL/examples/UI/out/test/tpot` that can serve as input to the Fortran version of TEAPOT. To run this file with FTPOT it is only necessary to append the lines

```
use, ring
makethin
twiss
analysis, energy=1.93827231, xtyp=1.e-6, pxtyp=1.e-6, ytyp=1.e-6, &
          pytyp=1.e-6, dptyp=1.e-6, particle=proton, print
```

This facilitates code comparison and result checking and makes available certain TEAPOT algorithms that are not supported in UAL as well as certain post-processing tools.[7] performed by the ZLIB module (an approach first taken by Forest.) This is probably the most significant analytical improvement of the present version of TEAPOT compared to earlier versions. The earlier approach, though exact in principle, was in fact limited to low order (roughly through sextupole order) by considerations of machine precision. Differential algebra gives results exact to all orders, limited only when computation resources are exhausted. These thin element results also converge exactly to continuum results in the limit of vanishing interval lengths. (This was confirmed by Forest.) The next step in the simulation is to define beam parameters: (8)

```
$shell->setBeamAttributes("energy" => 1.93827231, "mass" => 0.93827231); #94
```

[7] Incidentally, the original version of TEAPOT uses the arguments xtyp, pxtyp, ..., to the `analysis` command as "infinitesimals" in the evaluation of lattice functions by numerical differentiation. This is *really* crude! As TEAPOT is implemented within UAL all differentiation is performed by differential algebra

Here only particle mass and total energy have been specified, but other properties such as charge, or revolution frequency could be entered. At this point the *ideal* physical system to be simulated has been fully described. Before incorporating "blemishes" it is appropriate to investigate its linearized behavior and its momentum dependence: (9)

```
my $dp; #104
for($dp = -0.02; $dp <= 0.02; $dp += 0.005){
   $shell->analysis("print" => "./out/" . $job_name . "/analysis" . "." . $dp,
   "dp/p" => $dp);
} #108
$shell->map("order" => 1, "print" => "./out/" . $job_name . "/map1");  #112
```

The `for` loop ending at line `#108` performs Twiss analyses for each of nine fractional momenta, from -0.02 to +0.02. The construct `¨./out/¨.$job_name.¨/analysis¨.¨.¨.$dp` opens a sequence of files, `./out/test/analysis.-0.02`, ..., one for each of the momenta in the `for` loop. Line `#112` outputs the 6×6 once-around transfer matrix of the ring for the most recent analysis. More precisely the output consists of a ZLIB-formatted listing of all orders up to 1 (i.e. 0 and 1) of the "VTps" (vector of truncated power series) that make up the nonlinear once-around transfer map. The (reduced-precision) output is: (10)

```
ZLIB::VTps : size = 6 (dimension = 6  order = 1 )
 0 -3.512e-17 -1.387e-17  0.000e+00  0.000e+00 -1.022e-15  0.000e+00   0 0 0 0 0 0
 1  2.338e-01 -4.801e-01  0.000e+00  0.000e+00 -2.090e-05  0.000e+00   1 0 0 0 0 0
 2  2.497e+00 -8.513e-01  0.000e+00  0.000e+00  2.000e-05  0.000e+00   0 1 0 0 0 0
 3  0.000e+00  0.000e+00  2.458e+00  4.715e-01  0.000e+00  0.000e+00   0 0 1 0 0 0
 4  0.000e+00  0.000e+00 -1.171e+01 -1.840e+00  0.000e+00  0.000e+00   0 0 0 1 0 0
 5  0.000e+00  0.000e+00  0.000e+00  0.000e+00  1.000e+00  0.000e+00   0 0 0 0 1 0
 6  5.689e-05 -2.744e-05  0.000e+00  0.000e+00  6.414e+01  1.000e+00   0 0 0 0 0 1
```

As well as the closed orbit in the first line (which vanishes on-momentum), the 2×2 horizontal and vertical transfer matrices are included in lines 1 through 4, and longitudinal matrix elements are also included. The vanishing of "off-diagonal" transverse elements shows that the lattice is uncoupled at this stage. The meaning of the indexing columns on the far right should be obvious. A cubic map is exhibited in section 4.4.1 (9) .

Having completed the analysis of the ideal machine, to make the simulation more realistic, one adds imperfections using commands like: (11)

```
$shell->addFieldError("elements" => "^(bnd)\$",   "R" => 0.13,
       "b" => [0.0, 0.1, 51.0, 0.5, -26.0, 0.2, 0.0, 0.0, 0.0, 0.0]); #124
...
my $iseed = 973431; #143
my $rgenerator = new ALE::UI::RandomGenerator($iseed); #144
my $qSigB  = [0.0, 0.0, -2.46, -0.76, -0.63, 0.00, 0.02, -0.63,  0.17, 0.00]; #148
my $qSiqA = [0.0, 0.0, -2.50, -2.00,  1.29, 1.45, 0.25,  0.31, -0.11, 1.04]; #149
$shell->addFieldError("elements" => "^qdh",  "R" => 0.1,
                      "b" => $qSigB, "a" => $qSiqA, "engine" => $rgenerator); #157
...
```

As before, the element selection is performed by regular expression matching. Field errors can be systematic, and added to BND elements, as in line `#124`, or they can be random, and added to elements whose names begin with QDH, as in lines `#157`. For the random assignments a starting seed is assigned in line `#143` and the random number generator is specified in `#144`. "Erect" field multipoles are introduced via

the ¨b¨ list and "skew" multipoles are introduced via the ¨a¨ list. Not shown are the many error assignment commands for the other elements in the ring.

This example ends with multiturn particle tracking; (12)

```
my ($i, $size) = (0, 10); #177
my $bunch = new ALE::UI::Bunch($size); #179
$bunch->setBeamAttributes(1.93827231, 0.93827231); #181
for($i =0; $i < $size; $i++){
    $bunch->setPosition($i, 1.e-2*$i, 0.0, 1.e-2*$i, 0.0, 0.0, 1.e-3*$i);
} #185

$shell->run("turns" => 100, "step" => 10,
   "print" => "./out/" . $job_name . "/fort.8", "bunch" => $bunch); #188

open(BUNCH_OUT, ">./out/" . $job_name . "/bunch_out_new")
                || die "can't create file(bunch_out_new)"; #191
my @p; #193
for($i =0; $i < $size; $i++){ #194
    @p = $bunch->getPosition($i); #195
    $output= sprintf
    ("i=%5d x=%14.8e px=%14.8e y=%14.8e py=%14.8e ct=%14.8e dp/p=%14.8e \\n",
     $i,$p[0],      $p[1],    $p[2],    $p[3],    $p[4],     $p[5]);  #198
     print BUNCH_OUT $output; #199
} #200
print "End (", __LINE__, ")\\n"; #202
1; #204
```

In line #177 the number of particles is set to 10. In line #181 the energy and particle mass (both in GeV, as always) are set.[8] The first argument of `$bunch->setPosition` is a particle index; the remaining arguments are starting coordinates for the particles to be tracked, expressed as functions of `$i`. In this case they all lie on a x, y, dp diagonal. Line #188 starts the tracking process, requesting tracking of all particles for 100 turns. (The "step" argument is ignored and should be deleted from example.) The remaining lines give the output of the tracking. The file handle BUNCH_OUT is set to `./out/test/bunch_out_new` in line #191, the output is generated by line #198, and the printing is performed by line #199. The output file `out` therefore contains the six phase space coordinates of all ten particles after they have been tracked for one hundred turns.

(The purpose of the lonely final statement (`1; #204`) is to produce a non-zero return value (indicating successful completion) when control "falls out" the bottom of the routine. In general the most recently calculated value is what is returned from a subroutine. Exiting from the middle of a subroutine, with return value `$value`, can be accomplished by the statement `return $value;` .)

[8] By using the DDD debugger to trace through the code it is possible to step down into the code where inputs like these are actually used in order to confirm what physical quantities the inputs stand for. This is easier and more reliable than hunting for the information in external documentation (such as this manual).

4.2. Selective Lattice Function Output

One frequently wishes to output lattice functions evaluated at particular locations in the lattice. Locations in the lattice can be specified by their longitudinal coordinates or by the names of the elements at those locations. The latter approach is easier because it can use the regular expression mechanism described earlier.

Consider the same SNS lattice as was studied in the basic example (section 4.1) and suppose that we want global position (i.e. survey) data at every element and Twiss output at every bend element named BND. The script `$UAL/examples/UI_Analysis/shell_sns.pl` has been tailored to this task. When this script is run the immediate output is much the same as in section 4.1, except for the lines (1)

```
Linear analysis:
 ...
 survey
 twiss
```

The new lines in the script that generate this output are (2)

```
$shell->survey("elements" => "", "print" => "./out/" . $job_name . "/survey");
$shell->twiss("elements" => "bnd", "print" => "./out/" . $job_name . "/twiss");
```

The `survey` output appears in `$UAL/examples/UI_Analysis/out/test/survey`. Output occurs at every element since the empty string `""` matches all element names. The first several lines of output (slightly reformatted for this guide) are: (3)

```
-------------------------------------------------------------------------------------------
# name   suml(thick)  suml(thin)      x        y       z         theta     phi       psi
-------------------------------------------------------------------------------------------
0 qdmh1  0.0000e+00   0.0000e+00  0.0e+00  0.0e+00  0.0e+00     0.0e+00  0.0e+00  0.0e+00
1 qdmh   0.0000e+00   0.0000e+00  0.0e+00  0.0e+00  0.0e+00     0.0e+00  0.0e+00  0.0e+00
2 qdmh   2.5000e-01   2.5000e-01  0.0e+00  0.0e+00  2.5e-01     0.0e+00  0.0e+00  0.0e+00
3 qdmh2  5.0000e-01   5.0000e-01  0.0e+00  0.0e+00  5.0e-01     0.0e+00  0.0e+00  0.0e+00
4 o11    5.0000e-01   5.0000e-01  0.0e+00  0.0e+00  5.0e-01     0.0e+00  0.0e+00  0.0e+00
5 o11    2.2125e+00   2.2125e+00  0.0e+00  0.0e+00  2.2125e+00  0.0e+00  0.0e+00  0.0e+00
6 o11    3.9250e+00   3.9250e+00  0.0e+00  0.0e+00  3.9250e+00  0.0e+00  0.0e+00  0.0e+00
7 o11    5.6375e+00   5.6375e+00  0.0e+00  0.0e+00  5.6375e+00  0.0e+00  0.0e+00  0.0e+00
8 qflh1  7.3500e+00   7.3500e+00  0.0e+00  0.0e+00  7.3500e+00  0.0e+00  0.0e+00  0.0e+00
9 qflh   7.3500e+00   7.3500e+00  0.0e+00  0.0e+00  7.3500e+00  0.0e+00  0.0e+00  0.0e+00
 ...
```

The next request is for `twiss` output at all BND elements. The output appears in `$UAL/examples/UI_Analysis/out/test/twiss`, the first several lines of which (also reformatted) are: (4)

```
-------------------------------------------------------------------------------------------------------------
#  name  suml     betax      alfax      qx         dx         betay      alfay      qy         dy
-------------------------------------------------------------------------------------------------------------
 0     0.00e+00  2.626e+00  5.705e-01  0.000e+00  1.406e-05  1.232e+01 -2.260e+00  0.000e+00  0.000e+00
46 bnd 3.15e+01  4.271e+00 -1.075e+00  6.547e-01  3.253e-05  8.293e+00  1.764e+00  4.720e-01  0.000e+00
53 bnd 3.55e+01  8.281e+00  1.652e+00  7.330e-01  5.396e-01  4.190e+00 -1.055e+00  6.442e-01  0.000e+00
62 bnd 3.95e+01  4.137e+00 -1.075e+00  9.074e-01  2.050e+00  8.490e+00  1.812e+00  7.216e-01  0.000e+00
71 bnd 4.35e+01  8.374e+00  1.624e+00  9.867e-01  3.617e+00  4.116e+00 -1.020e+00  8.929e-01  0.000e+00
80 bnd 4.75e+01  4.271e+00 -1.075e+00  1.155e+00  2.857e+00  8.293e+00  1.764e+00  9.720e-01  0.000e+00
89 bnd 5.15e+01  8.281e+00  1.652e+00  1.233e+00  3.246e+00  4.190e+00 -1.055e+00  1.144e+00  0.000e+00
98 bnd 5.55e+01  4.137e+00 -1.075e+00  1.407e+00  8.067e-01  8.490e+00  1.812e+00  1.222e+00  0.000e+00
 ...
```

The `survey` and `twiss` subroutines called in code fragment (2) are contained in scripts `Shell.pm`, `SimpleSurvey.pm`, and `SimpleTwiss.pm` in directory `$UAL/ext/ALE/api/ALE/UI`. From these files one sees that the separate calls each trigger multiple calculations. Most of the function calls are self-explanatory; in any case fuller explanations will not be given here. But from this code it should be obvious how one can tailor the output, as regards quantities to be printed out, lattice locations where the quantities are to be exhibited, and general formatting. Perhaps one prefers $\sqrt{\beta}$ rather than β? One need only replace `$columns[3]` by `sqrt($columns[3])` (and similarly for β_y).

4.3. A Personalized Shell for Code Development

The examples presented so far showed how to run canned UAL scripts. If these are classified as "elementary UAL" we now advance to "intermediate level UAL". A key purpose of UAL is to be an environment in which an accelerator physicist's attention can be concentrated on a specialized problem without being overwhelmed by the complication of "the rest of the system". The reason this is important is that the detailed and correct understanding of any subsystem requires that the rest of the accelerator perform more-or-less as it is supposed to. A perfect simulation code would correctly model all systems and be prepared to answer any question concerning the functioning of the accelerator, but it is unrealistic even to strive for such a utopian situation. Rather, individual physicists strive to define and then answer sufficiently narrow, well-posed questions concerning the performance of accelerator subsystems. The purpose of UAL is to support such activity.

The physicist zeroing in on some area of interest, and wishing to use the tools of UAL, will often have to alter the existing code in the area of concentration as well as generating new code. In this sense "hacking into the code" is strongly encouraged. This is fairly straightforward, especially when only Perl code is involved. What may be less straightforward is keeping track of the changes in case it is necessary to "back them out" or to cause the improved code to be integrated eventually into the distributed version of UAL. Failure to do this leads, at best, to multiple versions or, at worst, to dis-use and eventual loss of the improvement.

The present example suggests a discipline to be followed, when revising the code; it is intended to facilitate the coordination of revised UAL code with the CVS-installed version of the code, with the goal of eventually facilitating the merging of the codes. The idea is to use the object-oriented feature called *inheritance* to establish a specialized user shell `UALUSR::Shell` that inherits all methods from the generic user shell `ALE::UI::Shell`, overriding some and generating others as required.

The physics of this example concerns itself with a family of quadrupoles, each equipped with a steering elements (kicker), and a BPM. The task is to center the beam horizontally at each of the (imperfectly aligned) quadrupoles. This example is based on an actual study of beam-based alignment of quadrupoles in the SNS and is discussed in detail in a technical note by Talman and Malitsky.[**10**]

This example is located in directory `$UAL/examples/BmBasedBPMAlign`. The Perl script is `BmBasedBPMAlign.pl`; it reads the lattice from `data/BmBasedBPMAlign.mad`. These files differ only slightly from the corresponding files in the basic example, section 4.1. The reader looking for a comprehension-testing exercise could generate

these files by hand rather than accepting the distributed versions. The main purpose is to establish an environment from which subsequent code development can proceed.

In the SIF (i.e. MAD) lattice description, the quadrupoles, eight in all, are treated as paired half-quads called QFH and, for simplicity the kickers and BPM's are treated as if centered on the quadrupoles. In the lattice file these packages are described by (1)

```
QF_a : LINE = (QFH, kickha, bpmha, QFH)
QF_b : LINE = (QFH, kickhb, bpmhb, QFH)
...
QF_h : LINE = (QFH, kickhh, bpmhh, QFH)
```

As mentioned already, this example continues to use the same SNS lattice as the previous example. The present changes are that the line `QF : LINE = (QFH1,QFH,QFH,QFH2)` in lattice file (4) of section 4.1 has been replaced by the eight lines just shown, in order to be able to adjust their elements individually. Where QFH pairs appears at eight places further down in the lattice file they are replaced respectively by `QF_a, QF_b,...,QF_h`.

Proceeding as in the first example, we start by applying random misalignments to the QFH elements, using the instruction (2)

```
my ($rMisalignIndices,$rdelx,$rdely,$rdeltheta) = $shell->addMisalignment
        ("elements" => "^qfh\$", "dx" => 0.01, "dy" => 0.01);
```

This `$shell->addMisalignment` simulation method did not appear in the previous examples. It differs from `$shell->addFieldError` only in that it perturbs magnets in position rather than in magnetic field. For present purposes we wish not only to misalign the magnets but also to know what misalignments have been applied. In real life these displacements would be unknown but, since we are testing a method, we need to "cheat" by peeking at the assigned values. The return arguments `($rMisalignIndices,$rdelx,$rdely,$rdeltheta)` in (2) point to this information. The first argument is a reference to the array of element indices of the displaced elements and the others arguments are references to the arrays of misalignments (of which we will discuss only the first in this example.) Unfortunately, by viewing the file `$UAL_EXTRA/ALE/api/ALE/UI/Shell.pm`, one sees that subroutine `$shell->addMisalignment` does not, in fact, return any values. We therefore have to modify that code by inserting the lines marked `# USR extension` in the following: (3)

```
sub addMisalignment
{
  my $this = shift;
  my %params = @_;
  my $pattern = " ";
  if(defined $params{"elements"}) {$pattern = $params{"elements"}; }
  my $arg_counter = 0;
  my $sigx = 0;
  if(defined $params{"dx"}) {$sigx = $params{"dx"}; $arg_counter++;}
  ...
  my @elemIndices =  $lattice->indexes($pattern);
```

```
  ...
  # - USR extension -------------------------------
  my @delx;
  my @dely;
  my @deltheta;
  # -----------------------------------------------
  ...
  for($i=0; $i < $#elemIndices + 1; $i++){
    $element = $lattice->element($elemIndices[$i]);
    ...
    if($sigx){
        $element->add(($sigx*$rvalue)*$dxKey);
        ${\\@delx}[$i] = $sigx*$rvalue; # USR extension
    }
    ...
  }
  return (\\@elemIndices,\\@delx,\\@dely,\\@deltheta); # USR extension
}
```

Too little detail has been given here to understand this code in every detail, but the following things have been accomplished. The first seven lines decipher the input arguments as was explained earlier. The next line shown saves the indices of displaced elements in the array `@elemIndices`. So far there has been no change but the following lines squirrel away information that is now known will be needed later. Then (because `dx` was included in the argument list) `$sigx` is *true*, so the misalignment values `$sigx*$rvalue` are saved in the array `@delx`. The last line returns the array reference.[9] The other return values will be ignored.) We have therefore modified code block (3) to be consistent with the call in code block (2) .

A major virtue of CVS is that changes like these can be hacked into the code without worrying about polluting the original version, since the original can always be retrieved. But, once your revisions have been frozen, you will want to save them; so you may as well plan for this from the start.

A new personalized "application interface" `UALUSR::Shell` is needed. This is something that you would have to generate and save as the file `api/Shell.pm`. Here, since this is an example, the script is supplied for you:[10] (4)

```
package UALUSR::Shell;
use Carp;
use strict;
use vars qw(@ISA);
```

[9] A quicker and dirtier approach to accessing the parameter changes would have been to make the saved arrays global variables so they would not have to be returned by reference. But this is the sort of slipshod practice that evolves inexorably into *spaghetti* code.

[10] The line `use Carp` modifies error reporting and is therefore inconsequential. The curious Perl construct `qw(x y z)` is equivalent to placing quotation marks around the individual arguments; i.e. equivalent to "x" "y" "z". The statement `use strict` prevents access to global variables and the statement `use vars qw(@ISA)` restores access to just the `@ISA` array. The `@ISA` array array is used three lines further down to declare that the newly-defined shell will start with all methods defined in `ALE::UI::Shell`.

```
use lib ("$ENV{UAL_EXTRA}/ALE/api");
use ALE::UI::Shell;
@ISA = qw(ALE::UI::Shell);
sub new
{
  my $type = shift;
  my %params = @_;
  my $this = new ALE::UI::Shell(%params);
  ...
  return bless $this, $type;
}
sub addMisalignment
{
   ...
   return (\\@elemIndices,\\@klvalue);
}
sub getErectMagnetStrengths
{
   ...
   return (\\@elemIndices,\\@delx,\\@dely,\\@deltheta);
}
```

The subroutine `addMisalignment` listed here was spelled out above in (3) . In this new shell the inherited method `addMisalignment` has been over-ridden; also a new method `getErectMagnetStrengths` has been defined (of which only one line is shown in (4) .)

To make use of the new shell the UAL command script (call it `BmBasedBPMAlign.pl`) now begins with (5)

```
 ...
use lib ("./api");
use UALUSR::Shell;
my $shell = new UALUSR::Shell("print" => "./out/" . $job_name . "/log");
 ...
```

which creates the new UALUSR shell instance. This replaces code fragment (1) of section 4.1. Following this will be the lattice definition commands, for example fragments from section 4.1: (2) , (3) , (6) , (7) , (8) , and (11) . Finally lines specific to the simulation being developed are included, such as: (6)

```
 ...
my ($rMisalignIndices,$rdelx,$rdely,$rdeltheta) = $shell->addMisalignment
    ("elements" => "^qfh\$", "dx" => 0.01); #190

my $numMisaligns = $#{$rMisalignIndices}; #192
my $numkicks = ($numMisaligns + 1)/2; #193
 ...
$shell->hsteer("adjusters" => "^kickh", "detectors" => "^bpmh"); #204
 ...
```

```
my ($rErectIndices,$rkickhs) = $shell->getErectMagnetStrengths
    ("elements" => "^kickh", "multindex" => 0); #208
 ...
```

Line `#190` is the revised command discussed above as code fragment (2) . Line `#192` illustrates the Perl syntax for obtaining the number of elements in an array and line `#193` accounts for the pairing of the quadrupoles (artificially present in the SIF lattice description.) Line `#204` uses the `hsteer` algorithm to resteer the beam through quad centers and line `#208` uses the newly-defined method `getErectMagnetStrengths` to obtain the strengths that have been determined by `hsteer`. From these strengths the quad misalignments can be inferred and then compared to the (known by cheating) actual quad misalignments. The calculations (not shown) required for this comparison are the sort of postprocessing activity for which Perl is ideal.

4.4. Fringe Field Map

Trajectory evolution through lattice sectors can be represented by maps. In this example the end fields of quadrupoles in the SNS will be modeled. Just as the end fields of dipole magnets are (predominantly) of quadrupole order, the end fields of quadrupoles are predominantly of octupole order—deflections are cubic functions of the transverse coordinates. A theoretical discussion of maps is given in appendix H.

Especially for hadron accelerators, it is essential to preserve symplecticity to high accuracy. No special treatment is required for dipole magnets with end fields modeled by quadrupoles, since the end fields are linear and symplectic. Often the end fields of quadrupoles can simply be ignored but, for a large aperture accelerator like the SNS, the octupole end fields have significant effect—their leading effect is dependence of tune on amplitude. To a good approximation an end field can be treated as if it acts impulsively at a single plane. The relation between input-to and output-from coordinates for this plane can be represented by a vector of truncated power series.

For extremely large amplitudes the convergence of such power series may simply be too poor for the series to be applicable. But when the fields are weak, as here, this is not expected to be an issue. The quad end deflections are approximated well by the cubic terms in the power series of the map. Yet one cannot simply include cubic terms without further investigation. Truncation causes nonsymplecticity[11] that can invalidate long term tracking. A cubic map, even if symplectic "to cubic order", is necessarily nonsymplectic "to quartic order" or, in general, a truncated map can be symplectic to its own order, but not to higher order.

To be able to investigate these issues it is valuable to have the capability of introducing maps that are truncated to arbitrary order and that are symplectic at least to their order of truncation. The so-called "Lie transform" formalism makes this possible. In this approach the map element corresponding to each phase space coordinate is obtained by appropriate differentiation of a polynomial "Hamiltonian" (sometimes known as a pseudo-Hamiltonian). The actual differentation process

[11] There are techniques involving (nonlinear) transformation to new variables such that the exact map elements are actually polynomials rather than infinite series. No such procedure is being considered here.

is in fact a Poisson bracket evaluation, an operation that is provided by ZLIB. What results, for each coordinate, is a power series representing its output value as a power series of all input coordinates. In first approximation the polynomial order of each of these series is one less than the order of the Hamiltonian. But it is possible, by iterating, by keeping more terms in the exponential series entering the Lie transform, and by truncating to higher order, for the resulting map to be symplectic to higher order than the initially-truncated Hamiltonian. This does not make the map "correct" to higher order, but it does make it symplectic to higher order. Starting from an approximate map this process can therefore produce a map that is symplectic to whatever order one is willing to evaluate the power series.

These mapping capabilities of UAL are the subjects of this section. First the map or maps have to be generated and then they have to be introduced into the simulation script. These are the tasks of the next two sections.

4.4.1. Map Generation. The code illustrating map generation is located in directory `$UAL/examples/HardEdge`. The script `ff.pl` begins (1)

```
use lib ("$ENV{UAL_ZLIB}/api", "$ENV{UAL_DA}/api");
use Zlib::Tps;
use HardEdge;
my $dimension = 6;
my $maxOrder  = 5;
my $space = new Zlib::Space($dimension, $maxOrder);
```

This example uses the truncated power series tools made available by the command `use Zlib::Tps;`. This capability was already exhibited, with minimal explanation, in section 4.1; a linear order, once-around, transfer map for the whole SNS ring was displayed as (10) . Here, as there, the full 6×6 phase space dimensionality is specified and the maximum truncation order is set to 5.[12] Continuing with script `ff.pl` (2)

```
my $I = new Zlib::VTps($dimension);
$I += 1.0;
# N - number of terms in the Lie transformation
# K - MAD quad coefficient multiplied by +1 (entrance) or -1 (exit)
my $ff_integrator = new HardEdge("N" => 1, "K" => -4.353051/5.6575, );
```

an identity map `$I` is defined, number of terms parameter `N` and strength parameter `K` assigned and the routine `HardEdge.pm` for propagation through fringe field (from the `HardEdge` package) declared. The header material of script `HardEdge.pm` is[13]: (3)

```
package HardEdge;
use vars qw(@ISA);
use Da::Const qw($X_ $PX_ $Y_ $PY_ $CT_ $DE_);
use Da::Lie::Integrator;
@ISA = qw(Da::Lie::Integrator);
```

[12] The order of a polynomial in UAL is dynamically determined as the power series is being evaluated. The order of any particular power series can therefore be less than the maximum order, but it cannot be greater.

[13] Header material like this was explained in a footnote to section 4.3.

As well as associating map variables with physical coordinates this causes the script to inherit all the methods of the parent class `Da::Lie::Integrator`. The `Da::Const` package relies on the Perl *typeglob* data type which is too technical and idiosynchratic to be explained here. The package defines and exports global references to read-only constants such as particle rest energies or (in this case) the indices 0,1,2,3,4,5, as they correspond to the variables x, p_x, y, p_y, ct, de. This has the seemingly cosmetic, self-documenting purpose of permitting a variable value such as `$p0->value(0)` to be expressed as `$p0->value($X_)`—the actual purposes are: to support the overloading of coordinate representations by both scalar value and power series; and to allow the truncated power series code to be mathematically general, absent of any particular identification of its variables with physical quantities.

The instantiation code in the new integrator is (4)

```
sub new
{
  my $type = shift;
  my %params = @_;
  my $self = new Da::Lie::Integrator(@_);
  $self->{K} = 0.0 unless defined ( $self->{K} = $params{K} );
  return bless $self, $type;
}
```

which receives K and N (not visible in the code fragment shown because it is received by `Da::Lie::Integrator`) as input arguments. The Lie integrator assumes propagation through drift as default; this behavior has to be overridden (in `HardEdge.pm`) by introducing a Hamiltonian appropriate for propagation through the hard edge of a quadrupole;[**9**] (5)

```
sub hamiltonian
{
  my ($this, $p) = @_;

  my $h = 1.;
  if($p->size < $PX_) { return $h;}

  # Beam

  my $v0byc  = $this->{v0byc};
  my $charge = $this->{charge};

  my $p0 = new Zlib::VTps($p->size);
  $p0 += 1;
  # Hamiltonian

  my $x2 = $p0->value($X_)*$p0->value($X_);
  my $y2 = $p0->value($Y_)*$p0->value($Y_);

  $h  = 3.*$x2*$p0->value($Y_)*$p0->value($PY_);
  $h -= 3.*$y2*$p0->value($X_)*$p0->value($PX_);
  $h +=    $y2*$p0->value($Y_)*$p0->value($PY_);
  $h -=    $x2*$p0->value($X_)*$p0->value($PX_);

  $h *= $this->{K}/12.;

  return $h;
}
```

Note the line `my $p0 = new Zlib::VTps($p->size);` which has defined a new vector of (initially vanishing) power series. The subsequent operations in the subroutine are *overloaded* in the sense that all operations (such as $=$, $-=$, and $*$) are performed on complete (truncated) power series.

Continuing with script `ff.pl`; (6)

```
my $ff_map = $I + 0.0;
$ff_integrator->propagate($ff_map, $beam_att);
```

the map is instantiated and then propagated through the fringe region using a method inherited from `$UAL_DA/api/Da/Lie/Integrator.pm`. This subroutine (with four inconsequential lines deleted) reads: (7)

```
sub propagate
{
  my ($this, $object, $beam_att) = @_;

  my $morder =  $this->{N};
  ...
  my $h = $this->hamiltonian($object) + 0.0;

  my $i;
  my $tmp = $object + 0.0;
  my $sum = $object + 0.0;
  for($i = 1; $i <= $morder; $i++){
     $tmp = $h->vpoisson($tmp)/$i;
     $sum  += $tmp;
  }
  for($i =0; $i < $object->size; $i++) { $object->value($i, $sum->value($i)); }
  $object->order($object->order);
}
```

The final lines of `ff.pl` (with their accompanying explanatory comments describing their purpose) are (8)

```
# truncate the order of power series
$ff_map->order($maxOrder - 2);
# write power series coefficients into the specified file
$ff_map->write("./out/ff_map.new");
```

The hard edge map has now been calculated and saved to `./out/ff_map.new`. The non-zero cubic rows are: (9)

```
ZLIB::VTps : size = 6 (dimension = 6  order = 3 )
28 -6.411917e-02  0.000000e+00  0.000000e+00  0.000000e+00  0. 0.  3 0 0 0 0 0
29  0.000000e+00  1.923575e-01  0.000000e+00  0.000000e+00  0. 0.  2 1 0 0 0 0
30  0.000000e+00  0.000000e+00  1.923575e-01  0.000000e+00  0. 0.  2 0 1 0 0 0
31  0.000000e+00  0.000000e+00  0.000000e+00 -1.923575e-01  0. 0.  2 0 0 1 0 0
35  0.000000e+00  0.000000e+00  0.000000e+00  3.847150e-01  0. 0.  1 1 1 0 0 0
39 -1.923575e-01  0.000000e+00  0.000000e+00  0.000000e+00  0. 0.  1 0 2 0 0 0
40  0.000000e+00 -3.847150e-01  0.000000e+00  0.000000e+00  0. 0.  1 0 1 1 0 0
54  0.000000e+00  1.923575e-01  0.000000e+00  0.000000e+00  0. 0.  0 1 2 0 0 0
64  0.000000e+00  0.000000e+00  6.411917e-02  0.000000e+00  0. 0.  0 0 3 0 0 0
65  0.000000e+00  0.000000e+00  0.000000e+00 -1.923575e-01  0. 0.  0 0 2 1 0 0
```

The format of this file was explained along with output (10) in section 4.1. This time (not shown) there are identity matrix elements in the linear part and zero elements in the constant and quadratic part. After calculating maps like these for all quadrupoles in the lattice one proceeds to incorporate the maps into the lattice description.

4.4.2. Map Application. The code illustrating the inclusion of maps in lattice descriptions is located in directory `$UAL/examples/UI_FF`. The starting lattice file is

`$UAL/examples/UI_FF/data/ff_sext_latnat.mad`

It is an SNS lattice much like the lattices in previous examples. The end field maps for its quads have been pre-calculated and reside in directory `quadff`. The first seventy or so lines of the `shell_sns_ff.pl` script are much like the corresponding lines of the script in section 4.1. Deviation begins with the lines (1)

```
print "Define 3D fringe fields", "\\n";

# We include ff only for quads because it was shown
# that contribution from bends was negligible

$shell->addMap("elements" => "^(qdh1)\$",
               "map" => "./quadff/fr1qd.zmap");
$shell->addMap("elements" => "^(qdh2)\$",
               "map" => "./quadff/fr2qd.zmap");
 ...
$shell->addMap("elements" => "^(qfbh2)\$",
               "map" => "./quadff/fr2qf.zmap");
```

Because the deflections (and even displacements that make the orbit discontinuous) caused by fringe fields occur at fixed longitudinal positions, their treatment by TEAPOT is just like the treatment of deflections by thin multipole elements. Therefore, nothing more needs to be done. The remaining lines of script `shell_sns_ff.pl` are identical to the corresponding lines of `shell_sns_.pl`.

4.5. SXF Input to UAL Simulations

4.5.1. SXF Rationale. There has always been a need for a portable, fully-instantiated lattice description. When parameter deviations are entered by a particular code from a measurement database or, even more so, when errors are generated by Monte Carlo programs, it becomes difficult to perform accurate result comparisons. Commonly one is forced to repeat calculations already performed, using possibly-suspect code, just for the purpose of regenerating identical data. This limitation was felt strongly when the US-LHC collaboration was starting to perform LHC simulations. The result was SXF (Standard eXchange Format).[**42**]

By now an even more important use for SXF has been realized at RHIC. It is for capturing "snapshots" of actual lattice conditions, encountered during operations, to be used for offline simulations and "post mortem" analysis. Though it is straightforward, starting from an SIF input, to collect and apply all the field imperfections, misalignments, apertures, operational procedures, etc. needed to instantiate all lattice elements, this is very time consuming and hard to maintain. Furthermore (for better of for worse from the point of view of database management) it is easier to output the current, fully-instantiated lattice parameters, than to update the original data sources.

To understand some of the issues involved in reconstructing a lattice from an SXF file some understanding of SMF (the UAL accelerator model) is useful; a brief description is contained in Appendix D of the PERL-interface guide. For now the only point to be made is that each element in the lattice has two names, a design/generic *GenName* and a fully-instantiated *LatName*; sometimes known as a site-wide name. Within SXF the lattice is represented by a sequence of *LatName*'s, along with their attributes. Since one of these attributes (the *tag* attribute) is

the *GenName*, it is easy to cross-reference one name to the other, even when only SXF information is available. But, within the original SIF lattice design, there are hierarchical features like sub-lines, symmetric sections, repetitions, and so on. None of this information finds its way into the SXF file. Hence a simulation that depends on hierarchical information (other than *GenName* association) must either start from a SIF lattice description or re-insert the required relationships *post facto.* So far this has turned out to be either unnecessary or straightforward, so the absence of hierarchical information from SXF files has proved not to be a serious impediment.

4.5.2. FastTeapot. As mentioned in the introduction, `FastTeapot` consists of recently developed C++ code whose purpose is implied by its name. Since the present user guide primarily documents the Perl interface, as contrasted with the C++ code, it cannot properly document `FastTeapot`. However, a few words can be said about the motivation behind examples

```
$UAL/examples/FastTeapot/linux/evolver,
$UAL/examples/FastTeapot/linux/tracker,
```

both of which are C++ executables. (Compilation instructions are given in the README file.) Their purpose is to exercise the Element-Algorithm-Probe framework and to compare computation times and results obtained using traditional element-by-element TEAPOT results with (matrix multiplication through sectors) `FastTeapot` results. Matrix multiplication is the most obvious speed-up mechanism for mapping through sectors. An application that came up recently requires tracking that needs to be fast while retaining a faithful representation of chromatic effects; in particular the second order coefficients T_{116}, T_{226}, T_{336}, and T_{446} need to be accurate.

With canonical (x, p_x) variables, even transport through drifts brings in chromatic effects which, being second order, are not accurately modeled by pure linear matrix multiplication. On the other hand drifts sections *are* linear when $(x, x' \equiv dx/ds)$, (position, slope) variables, are used instead of x, p_x (and likewise for y, p_y). In these coordinates, even with no quadratic terms included, the desired chromatic effects are retained. So matrix evolution through drifts retains the correct contribution of drifts to chromaticity. Of course quadrupoles, sextupoles, and octupoles continue to need symplectic, TEAPOT kick treatment. It is not claimed that this particular procedure is universally applicable, but it does exercise the Element-Algorithm-Probe framework.

Program `evolver` calculates second order, once-around maps two ways: one uses only traditional element-by-element TEAPOT kick-tracking; the other uses kick-tracking through quadrupoles and nonlinear elements, but uses the non-canonical (x, x') coordinates of the previous paragraph through other elements. Of the second order coefficients determined, it is only T_{116}, T_{226}, T_{336}, and T_{446}, that are expected to agree in this comparison. Program `tracker` is similarly motivated. It uses the same two evolution mechanisms to perform multiparticle tracking of a bunch of particles. The idea is that tunes obtained by FFT processing of the outputs of program `tracker` can be used to extract, and hence compare, off-momentum tune dependence.

Most theoretical formulas in lattice physics involve integration (usually approximated by summation) over all elements in the ring, of element parameters such as quadrupole gradients, length-strength products, inverse focal lengths, sextupole

strengths etc., weighted by fractional powers or other functions of lattice functions (β_x, β_y, ϕ_x,, dispersion, etc.) All of the required lattice functions are available from `$UAL/ext/ALE/api/ALE/UI/Shell.pm`. An later example will show how to gain access to the magnet strengths required. Once this information is in hand one can use Perl to evaluate the accelerator physics formulas involving such summations (or even integrations if necessary). Though Perl may not be as fast as a compiled language, it is plenty fast for typical postprocessing tasks.

CHAPTER E

ADXF: A Markup Language for Accelerators

The purpose of this chapter is to explore, at as elementary level as possible, issues associated with communicating lattice descriptions between diverse simulation environments. Some of the features emphasized in this chapter were first introduced in Etienne Forest's simulation code PTC. The discussion needs to be updated to apply to ADXF 2.0.

5.1. Data Interchange Among Accelerator Simulation Programs

This chapter describes an accelerator *exchange format* (XF) for communicating fully instantiated (warts and all) lattice descriptions between heterogeneous simulation programs or environments. This ADXF (Accelerator Description Exchange Format) mechanism is, at present, employed by UAL, as ADXF 2.0. For describing idealized *design* lattices one is accustomed to using SIF (*Standard Input Format*)—more commonly known as MAD files. A partial response to the need for a fully instantiated lattice description is a fully-instantiated description called SXF (*Standard Exchange Format*). This format has had substantial usage, especially at RHIC and in the US/LHC collaboration. What is described here is much in the spirit of SXF, but with two major enhancements: it encorporates orientation and misalignment information, as used, for example, in PTC (*Polymorphic Tracking Code*) and it is sufficienly disciplined to be fully implemented using XML (*Extensible Markup Language*). The proposed XF will be referred to as ADXF (*Accelerator Description Exchange Format*). (Actually ADXF 2.0 as a preliminary version exists.)

The XML Schema Standard, released more recently than the original ADXF, provides useful features such as standard containers and inheritance. Even if XML is less than ideal, its very existence can be helpful in facing up to a "tower of Babel" problem that accelerator language has. Like XML, support for extensibility is one of ADXF's fundamental design specifications.

5.2. Vocabulary

This section consists of a mini-glossary of terms applicable to a language capable of communicating *all* the (relevant to beam simulation) parameters of an accelerator. Most of the terms, though familiar, are too abstract to have unambiguous meanings. The purpose is to re-introduce them in the present context, if only to consider their appropriateness to the task at hand.

The "Grammar" of Accelerator Simulation. A sentence describing accelerator performance has (as subject noun) a magnet, which deflects (verb) a beam (object noun). When discussing accelerator simulation it is appropriate to use matching language. We will refer to the hardware making up the lattice (the subjects) as *elements*, the formulas describing propagation (the verbs) as *algorithms*,

and the objects being propagated as *probes*. *Probes* might also be referred to as *observables*. Examples of probes are particle coordinates, beam moments, transfer maps and transfer matrices, spin vectors, Twiss functions, distortion functions, normal forms, etc. The full language is required to describe any particular simulation code. But only a description of the *elements* should be needed for communicating accelerator structure between two codes.

A self-contained simulation code does not need to be fussy about grammatical correctness in the sense just introduced. Since the "Physics" is contained in the algorithms it is not uncommon for data that is essentially algorithmic to get intermingled with true element data. Internal to a code based entirely on, say, transfer matrices, there is no need to distinguish between a magnet as an element and the magnet as a transfer matrix. For such a program a quadrupole *is* a 2×2 matrix with entries expressed in terms of sines, cosines, hyperbolic sines, etc. Here the subject noun is thought of as a verb. Alternatively a quadrupole might be visualized as a bundle of valid trajectories through it—i.e. a subject noun is thought of as an object noun. Furthermore, different physics leads to the different intermingling of algorithmic information in different simulation codes.

Implicit physics-containing algorithmic "built-ins" like this are impediments to the exchange of lattice information among programs. Since we are interested in more general communication we have to use more careful grammar. As already mentioned we need only be concerned with the *elements* in the lattice—the *subject nouns* of the grammar. For the exchange formalism to be useful there needs to be more or less unambiguous agreement as to what constitutes the data needed for a complete lattice description and the data in customer programs has to be organized in such a way that algorithmic directives and data data can be segregated from element-describing data.

Markup Language. *Markup language*, of which XML is the most relevant example, was introduced to permit the electronic transfer of textual material such as books and articles. A book and an accelerator lattice have surprisingly similar structure. Both consist of a one dimensional sequence of discrete (and rather heterogeneous) elements that are normally intended to be "read" sequentially from beginning to end. Much of a book is already sequential text, but other features like tables and figures have also to be communicated by the markup language. A markup language for books is also concerned with page layout and presentation. Frequently the eventual layout requires tables and figures to be almost, but not quite, in their original sequential order. At first glance this layout capability may seem to have no analog in lattice description. But, in fact, the need to convey global orientations and misalignments is the main lattice description feature newly being added in ADXF 2.0. (The need for textual representation of geometrical information in scientific applications has recently been addressed by GDML *Geometry Description Markup Language*)). The essential requirement for a markup language is that everything of importance for accelerator simulation—geometry, alignments, deviations, parameters, names, etc.—has to be convertible to text using the markup language.

Recursive Aggregation. This term is introduced at this point "out of the blue" since, combining the concepts of inheritance and association, as it does, it is the only conceptually challenging ingredient of this chapter. A *recursive aggregation* is said to be based on a *composite design pattern*. The fundamental element of

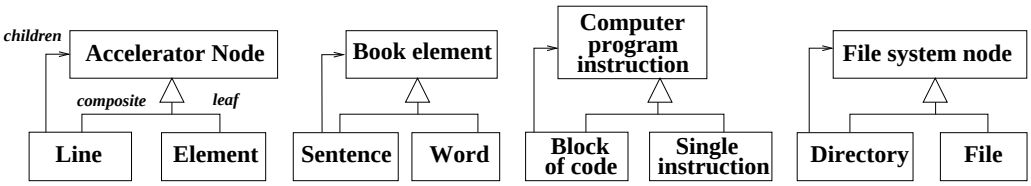

FIGURE E.1. Some familiar recursive aggregations. The open, upward pointing arrow represents the *is a* relationship, or inheritance. Read in the direction of the arrow this association is also known as "generalization". The simple arrow represents the *has a* aggregation relationship. The file system case differs in one important way—element order does not matter; this makes name assignment to every file and directory obligatory. In all cases the possibility of recursion has to be supplied mentally. XML is well matched to the task of describing structures like this by text.

a book is a *glyph*, the most fundamental and familiar of which are the letters *a*, *A*, *b*, ..., punctuation marks, and so on. A *word* is an (ordered) aggregation of glyphs and a sentence is an ordered *aggregation* of words—then on to *paragraphs*, *sections*, *chapters*, etc. (Here *aggregation* is referred to as a "has a" relation—as in a word "has" letters.) The aggregations are themselves something like glyphs. Rather than calling them "superglyphs" or something like that, we generalize the meaning of the word "glyph". So the book's glyphs can be considered to be its words, or its sentences, or its paragraphs, etc. That is, an ordered aggregation of glyphs can, itself, be treated as a glyph. Using these glyphs the entire structure can be abbreviated or expanded as desired. This aggregation makes it possible to view the book at various levels of granularity. Letters and words can serve as their own names. Larger glyphs, like chapters, may, but need not, be given abbreviated names.

Recursive aggregation is similarly useful for the description of an accelerator lattice. The finest granularity may be a single *element*, such as a bending magnet, quadrupole, or drift section. A few elements, perhaps all mounted on a single girder, may be though of as a coarser element. A longer sequence may form a *cell*; many cells may form an *arc*, and so on. In short, the whole lattice is a recursive aggregate of generalized elements. The meanings of the terms *parent*, *child*, and *sibling* are obvious in this context. One element is special: it is the *root* element, having children but no parent. For a book it is the whole book, for a lattice, the whole lattice. Another type of element is special: it is a *leaf* element, which has a parent but no children. To facilitate referencing, for example to recover hierarchy, all elements should be named, and child elements can keep track of their parents by name. (The inheritance capability of XML supports this mechanism.) These names also facilitate the external communication of lattice descriptions. In SIF such named ordered sequences are referred to as "lines".

These ideas are exhibited diagramatically in Figure E.2 which exhibits some familiar recursive aggregates as well as some symbols needed to understand later figures.

The fact that books and accelerator lattices have such similar recursive organization should make it not surprising that very similar markup languages can be

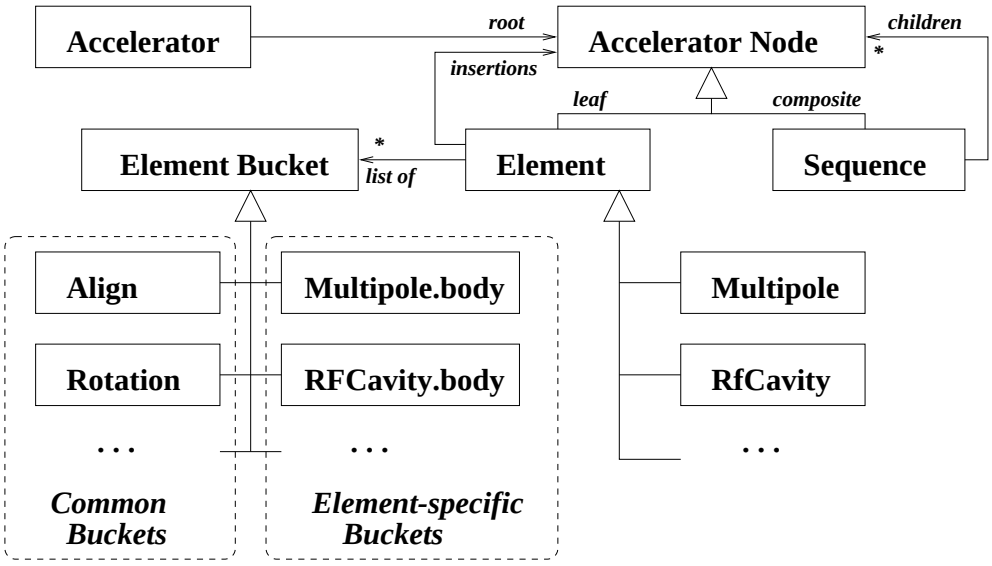

FIGURE E.2. The ADXF 1.0 object model. The symbols used in this diagram have been defined by example in Figure E.1. This model includes *real element* parameters but not the geometric information needed for *installed element* specification.

used for their description. As a matter of fact, Accelerator Description Exchange Format (ADXF) describes an accelerator lattice using only XML.

The symbols used in object diagrams have been defined by example in Figure E.1. An expanded version of a lattice object model is shown in Figure E.2. This (original ADXF) model does not support the geometric information whose inclusion will be discussed later in this chapter. It does, however, indicate the way parametric information is associated with physical elements.

Object Orientation. This term will not be defined here, but some of its ideas are needed to understand XML and the diagrams used in conveying an accelerator *object model.* The intention is to include just minimal explanations of a minimal set of needed ideas. The following several features are usually considered to be object-oriented properties. It is the *unimportance* of some of the terms that is the most important thing to be conveyed.

Polymorphism. Though not necessarily regarded as the most essential object-oriented feature, *polymorphism* has become a *sine qua non* of accelerator simulation. A particle coordinate has to be regarded both as a *real* number type (in both mathematical and computer science senses) and as a truncated power series type (TPS). In earlier days this statement was trivially accurate since matrix theories amount to the lowest order of Taylor series. It is the need to perform analytical differentiation that makes the TPS so necessary. (Actually, the original TEAPOT contradicted this statement, even in the presence of nonlinearity, by evaluating derivatives by numerical differencing. But this becomes progressively less accurate and less practical with increasing order of differentiation.)

Encapsulation. From a computer science point of view *encapsulation* (along with its natural concomitant *data hiding*) is of great importance. Both *data* and *methods*

can be encapsulated within *classes.* The task of computer science is to process complicated data in error-free fashion and encapsulation is thought to contribute greatly to success at this. The computer scientist believes in "executive privilege" in which the peons are better off not knowing the details. A physicist, on the other hand, believes in "open government" where nothing should be hidden. Data hiding is a self-imposed constraint, unlikely to please the physicist. More seriously it can be said that, though important within individual codes, encapsulation is not important for the task at hand (which is external communication of lattice parameters.)

There is, however, one encapsulation issue that will have to be faced. Within the PTC evolution paradigm (to be explained more fully later on) a single step, or unit, of beam evolution proceeds from a reference plane of one element, let's call it the entrance face, to the entrance face of the next active element. Exit faces are implicitly present in this evolution, but they are *hidden* externally.

In conventional accelerator language one would say that the active element is concatenated with its following drift. This hides the exit face. To communicate the evolving beam properties at the exit of an element between two programs it is clear that an exit face needs to be defined. ADXF, like most simulation environments, assumes the presence of exit faces as part of a dynamics-free geometric lattice description. To integrate PTC into ADXF therefore requires the un-hiding of exit face information in PTC. A straightforward way of doing this is explained later on.

Inheritance. As mentioned already in a figure caption, *inheritance* is an "is a" association. To support recursive aggregation, let us use the term *virtual element* (also to be called an *element node*, the term used in Figure E.1.) so that either a magnet or a sequence of magnets "is a" real actualization of a *virtual element.*

The attraction of inheritance within computer codes is said to be the economy provided by "code re-usage"; code or data applicable to a parent is available for a child. Some of the encapsulated methods, such as store, adjust, or read-out, amount to database management functions that can usefully be inherited. But inheritance can be vitiated if the child has traits incompatible with the parent's methods. Such methods have to be overridden.

In the present context the presence of "accelerator physics" within an encapsulated method is likely to foil the useful inheritance or sharing of a code library. This may present an impediment to the sharing of self-consistent class libraries like CLASSIC and LEGO—the very things that make two codes individually powerful may impede transfer of lattice description from one to the other. Here it is assumed that no accelerator physics algorithms whatsoever are to be conveyed by an XF.

The main problem being addressed in this chapter then, is how to define *static* accelerator structure in such a way that different (and certainly not inheritable) methods can to be applied to the same structure. Data description inheritance mechanisms within XML will be helpful even though the inheritance of physical algorithms is not.

5.3. "Design" Elements, "Real" Elements, "Installed" Elements.

In the idealized lattice described by SIF there are only idealized "design" elements. They are described by a relatively small number of parameters like type, length and strength. A "real" element is constructed to be as nearly identical as

possible to the *design* element it is intended to reproduce. But its parameters invariably deviate from the ideal, possibly in quite complicated ways (such as field nonuniformity or end effects) and it may have peripheral characteristics, such as aperture dimensions. In PTC documentation such elements are referred to as being "on the bench" in order to stress that the deviant properties can be, and have to be, determined with no reference to the element's eventual placement in an accelerator. The LEGO metaphorical language similarly, and appropriately, calls to mind a free-standing element that can be individually characterized without reference to where it may eventually be installed. As well as its deviation parameters, there needs to be a mechanism to associate a *real* element with its *design* element.

Finally there are "installed" elements. Each is a *real* element, augmented by the geometrical information need to fix its position and orientation within an actual lattice. In PTC, an element, along with its orientation and its geometric relation to its neighbours is known as a "fibre". So a *fibre* and an *installed element* are, except in technical detail, essentially equivalent.[1] The data of an *installed element* includes the *patch* and *chart* information defined in PTC documentation.

It is useful to regard *real* elements as descendants of *design* elements and *installed* elements as descendents of *real* elements. But this chain of inheritance does not need to be shown explicitly in an object diagram because the three objects are so similar. Descendants differ only by acquiring more parameters. Originally XML did not support inheritance, but the recently released XML Schema standard supports inheritance in the form of *extension* of the data associated with children. Each *installed* element has a unique "swn" (site-wide-name), and can refer by name to its *real* parent (which might, for example, be a serial number) and to its *design* grandparent (for example from the SIF design description).

5.4. The Need For Local Reference Frames

Much has been made of the requirement that "Physics" be kept out of any exchange format. But geometry *is not* physics, at least for present purposes. To have any hope of precision data interchange there needs to be unambiguous agreement about at least some geometric attributes of a lattice. Certainly it should be clear that the detailed description of an accelerator has to include the location and orientation of all of its elements. This is an area in which the SXF exchange format is unable to convey the geometric information needed, for example, by PTC.

It would be handy if all accelerator calculations could be done using a single global coordinate system (as might be appropriate for describing a room-sized particle detection apparatus.) But accelerators can have overall sizes of order 10^5 m, and displacements as small as 10^{-9} m may be of interest. It is impractical to waste the 14 decimal places that would be needed to use a single global reference frame.

The workaround for this problem, is to introduce local frames of reference, relative to which nanometer (or better) accuracy is feasible. Such local frames are typically introduced without comment in simulation codes. Here such frames are introduced accompanied by comments.

[1]It would correctly represent authorship priority for what we call an *installed element* to be given its chronologically earlier name FIBRE. In PTC the collection of all FIBRES is a FIBER BUNDLE or LAYOUT, which, along with all *real element* descriptions, completely describes all geometric relationships in the whole installedlattice. The term "layout" seems appropriate but some find the name "fibre" overly mathematical and "fibre bundle" even more so.

Accuracies in placement of the elements of an accelerator are typically at the sub-millimeter level. As a result, even when a displacement is quoted to nanometer accuracy in a local frame, it is implicit that the displacement would have to be quoted with ± millimeter error bar in global context. This error bar would characterize the likely amount by which the dead-reckoned beam would miss a bull's eye at that point. Any experimentalist realizes, if nanometer accuracy will be required, that an experimental determination of the local position will have to be made using a precision microscope-like measuring device. Viewing the beam position with the microscope, the upstream parameters of the beamline will be adjusted empirically to achieve the required accuracy.

The microscope needed for this tuning is normally not present in the sort of lattices being described here. Even if it were, absent its measured values, it could not be used to refine the precision. But a computer simulation pretends to have access to local positioning to arbitrarily high accuracy. The computer code might be said to possess a kind of "virtual microscope" to obtain this high accuracy.

The purpose of local reference frames is to "sweep these problems under the rug" in this way. When realistic bend errors are introduced into the simulation, millimeter scale uncertainties become evident even though individual coordinates are being calculated to nanometer accuracy. The above-mentioned empirical centering can be simulated in the computer by appropriately tweaking upstream steering elements. Operationally one must eventually replace the virtual microscope by a real device, but the simulation experience gained offline can be expected to be a valid aid in anticipating the behavior of a real device.

5.5. Definition of Local Frames

Different codes are likely to choose local frames differently. But a more abstract source of variability comes from what might be called the evolution paradigm employed. Traditional tracking codes have evolved beams from active element to active element with " entry→deflection→exit→drift→entry→deflection→exit→drift..." sequencing. In PTC "deflection→patch_to_entry→deflection→patch_to_entry→... " more accurately describes the evolution. One way of viewing this is, perhaps, that, whereas conventional codes run the beam past the elements, PTC runs the elements past the beam.[2] Another way is to say that PTC's *patch* replaces the combination of determination_of_exit_position plus propagation_through_drift. Except for one thing these distinctions are largely academic as far as actual computation is concerned. The exception is that, in PTC, in setting up the specification of coordinate systems to be used, none are centered on the exit planes of magnets.

The entry_plane/exit_plane paradigm seems to be inextricably woven into the fabric of UAL. This complicates our current task, which is to integrate PTC into UAL. This requires the presence of exit plane local frames in ADXF. A straightforward work-around that does not offend the PTC paradigm is is to augment the PTC layout by artificial null, but treated as active, elements (customarily called "markers") at the exit planes of active elements. Since exit planes are already present (though hidden) in PTC, this entails only minimal extra calculation. This will force, within PTC, and without offending PTC principles, the inclusion of the

[2]There are ambiguities of perspective associated with the active and passive interpretations of transformations and the distinction between transfer maps and compositional maps. This is discussed in Section 2.3.1 of Forest's book *Beam Dynamics, A New Attitude and Framework.*

local frames needed to integrate PTC into UAL. The following sections describe in greater detail how this can be done.

5.5.1. A Minimal Discrete Abstract Skeleton. In PTC the complete global geometry of a lattice is included in what is known as a LAYOUT. There is a local frame centered near every active element. Its origin will be referred to as a "local frame origin" (LFO). A frame is typically defined to align with its associated *design element* but, in principle, the frame is specified arbitrarily. One thing needed to locate an *installed element* is the displacement, expressed in local coordinates, of some fiducial point (FP) (such as element center) from the LFO. Then the orientation can be fixed by matching fiducial directions fixed in the element to orientation arrows specified in the local frame.

Continuing to mimic PTC, the geometric relation between two adjacent local frames can be described by a *patch.* Except for absolute location and orientation of the collection as a whole, this collection of frames and patches, to be referred to as a "minimal abstract skeleton", provides all geometric information needed for global coordinate definition.[3]

Being completely general (which "abstract" connotes) with one frame per active element (which "minimal" connotes) the *minimal abstract skeleton* permits even bizarrely-scattered positioning of the elements making up the lattice. (Here "minimal" refers to PTC. As explained already this skeleton is, as yet, less than minimal from ADXF perspective.)

It goes without saying that the full generality of the minimal abstract skeleton is not expected to be exploited in typical practice. Commonly some global design exists and all LFO's are picked to lie on a well-defined continuous curve (such as a design closed orbit) threading its way through the apparatus. This *does not* require any true orbit to follow this curve but it does imply the possibility of establishing a curve more or less following a beam line.

What is being proposed is the use of the *skeleton* (to be optionally augmented in ways to be introduced later) to convey a full geometric description from one simulation environment to another. For this to be successful it is not necessary for either program to actually use these particular local frames for internal calculations. But each program needs the capability of transforming its coordinates to and from these frames.

A single local frame can be quantitatively described by the (X_i, Y_i, Z_i) global coordinates of its LFO along with three other numbers, for example Euler angles, describing its orientation. There may be practical reasons for using other (or more) parameters, such as matrix elements or direction cosines, or SO(3) parameters, to express orientation but, in principle, just 3 orientational parameters are needed. Therefore a minimal abstract skeleton having N_F frames is quantitatively specified by a list of N_F pairs of triplets.

In the differential geometry jargon of PTC the local frames are known as "fibres" and the *skeleton,* along with its local frames is a "fibre bundle".

[3]In topological mathematics much is made of "atlases of overlapping charts", a là Rand-McNally or Michelin, which cover a whole space and, in regions of overlap, are required to "agree". The *skeleton* introduced here is quite similar to an atlas. In a mathematical sense a *skeleton* is trivial however, in that, except for the measurement precision complication already described, any local frame could cover the whole lattice. In PTC the mechanism for enforcing agreement between local frames (or charts) is known as "patching".

5.5.2. Reference Frames and Picture Frames. Something that provides a rich source of confusion is the dual meanings of the word "frame". So far, the only meaning has been the mathematical one of geometric reference frame. But another meaning of the word is picture frame. This latter meaning is valuable in accelerator lattice description when a sequence of elements is constrained to move rigidly, for example because the elements are mounted on the same girder.

There is yet another need for "frames" used in this sense. Multiple pass accelerators like CEBAF and intersecting rings like KEK and PEP II, in which one or more elements is shared between two rings, present a problem for lattice description.

In an art gallery each picture, or group of pictures is mounted in a frame. Then the layout of the gallery amounts to the placing of the frames. The same organization applies to an accelerator, which can therefore be thought of as being like an art gallery. The support for this feature is what distinguishes ADXF 2.0 from ADXF 1.0.

Most of the discussion in this chapter should actually refer to the placement of frames (in the sense of picture frames) rather than to the placement of individual elements. When the term `frame` appears in the ADXF 2.0 lattice description file it has the meaning of picture frame.

!!!!! This discussion needs to be tidied up.

5.5.3. A SIF (Continuous) Skeleton. When a lattice is described using the Standard Input Format, a collection of local reference frames is being defined implicitly, and there is no concept of "picture frames". The primary location coordinate is arc length s along a *reference* or *design* orbit. "Natural" reference frames are established wherever needed using the Frenet-Serret geometric prescriptions. The full geometry is established using a so-called SURVEY instruction. Using global (X, Y, Z) coordinates, for lattices lying purely in the (X, Z) plane, the natural local coordinates are tangential deviation (from local origin) Δs, along with "vertical" coordinate $y = Y$ and "horizontal" (centrifugal) coordinate x. For noncoplanar lattices the assignment of natural local frames is trickier but, with the inclusion of conventions that resolve orientation ambiguities in drift spaces, the specification is still unambiguous. Recapitulating, a SIF file implicitly contains all information needed to construct an unambiguous *skeleton.*

5.5.4. A Continuous Abstract Skeleton. Of the two skeletons introduced so far, PTC and SIF, the SIF version has the advantage of continuous definition. The continuously defined set of reference frames make possible the insertion of elements at unambiguous location and orientation. This is impossible within a discrete abstract skeleton since interpolation between frames is undefined. By introducing a few restrictions (easily satisfied by all, or at least most, practical accelerator configurations), and some more parameters, it is possible to convert a *discrete abstract skeleton* into a *continuous abstract skeleton.*

To support insertion based on a single longitudinal parameter s, there has to be a single continuous "skeletal curve", intimately associated with the skeleton, and threading its way through every LFO of the discrete skeleton being refined. The parameter s is (presumeably) arc length along this curve.

The analytical description of a kinky space curve can, in general, be quite complicated. To reduce complexity the following restrictions to the minimal abstract skeleton are suggested:

1 The reference face associated with an active element is required to pass through the corresponding SIF *entrance* point. This restriction refers to the local frame origin and places no restriction on the orientation of the local frame.
2 The minimal abstract skeleton must also contain an LFO at the *exit* face point defined in the SIF file. It has already been explained how this can be done without offending PTC philosophy.

In making use of the *skeletal curve* it is inevitable that points other than LFO's will be introduced on it. Such labeled points will be referred to as "smooth skeletal points" (SSP), where "smooth" connotes the absence of kinks (i.e. slope discontinuities) at an SSP. (Kinks are preferably absent also at LFO's, but they are allowed there.) LFO's and SSP's therefore form disjoint sets of points, all lying on the skeletal curve.

Preferably every point on the skeletal curve would have its own local frame, but one wishes to avoid this overhead. By imposing some more restrictions the continuous skeleton can provide this information implicitly, just as the SIF skeleton does. These restrictions, along with extra parameters and explanatory comments, can be covered in a few lines:

1 The skeleton curve is restricted to consist only of circular arc segments, attached end to end. Straight lines qualify as circles. Smooth joins (incoming tangent equal to outgoing tangent) are favored, but not required. A kink, if present, can occur only at an LFO. Stated differently, the skeleton consists of the circular arcs joining all the LFO's.
2 Each of the N_F local frames is, so far, defined by a location triplet and an orientation triplet. This data needs to be augmented by the entrance and exit directions of the skeletal curve, expressed in the global coordinates. For the sake of definiteness let us specify the entrance direction by polar angle $\theta_{i,<}$ and azimuthal angle $\phi_{i,<}$ and, in the same coordinates, the exit direction by $(\theta_{i,>}, \phi_{i,>})$. This data, along with the circles-only requirement and the coordinates of adjacent local frame origins, completely specify the entrance and exit skeletal curve segments.

The only redundancy of this description is that an individual sequence is determined by both the succeeding and preceding LFO. This redundancy is intentional. Its purpose is convenience and to avoid the need for distinguishing forward and backward evolution. No provision for checking self-consistency of this redundancy can be expected from ADXF. If the lattice purports to be closed then the global coordinates of both starting and ending LFO will be $(0, 0, 0)$. Self-consistency here will also not be checked. For multiple turn tracking, points in the final transverse plane would simply be identified with points having the same coordinates in the starting transverse plane. To the extent the global survey is inconsistent this will assume "beam me up, Scottie" propagation from final exit plane to starting entrance plane.

Even with these restrictions the *abstract skeleton* is still completely general since arbitrary local frames can always be joined by circles. But, to contribute to its usefulness, the skeleton will preferably correspond to some sort of reference orbit through the system. From an SIF lattice description the *design* global coordinates (X_i, Y_i, Z_i) of every element are inferred by sequentially laying elements tail to head and completing a survey. These are the favored LFO's. For the traditional

SIF elements, including *rbends* and *sbends,* the restriction to circular segments is automatically met. Special bending elements in which the design orbit is not circular, such as combined function *rbends,* will require special treatment.

5.6. Precision Positioning of Elements

Using the local frames it is possible to identify and label other points, close to, but not necessarily on, the skeletal curve, for the purpose of locating elements. Along with the SSP's and LFO's, these points will be referred to as "virtual survey monuments", abbreviated as VSM. To the extent the *installed* elements deviate from their *design* values the true closed orbit (and all other orbits) may miss these points by arbitrarily large amounts—in metaphorical terms, the flesh is not necessarily on the bones of the skeleton. But the VSM's provide globally positioned points from which local deviations can be measured. In practice, since accelerators are, in fact, carefully constructed, the deviations will usually be fairly small.

Many accelerator lattices, by design, lie entirely in the same (by definition "horizontal") plane because there are no intentional vertical bends. With the entire lattice smooth and coplanar, the *skeleton* of LFO's can be automatically augmented by unambiguous "transverse planes" through each LFO. Though they have the same origins, their orientations may differ from those of the *minimal abstract skeleton.*

Since most accelerator lattices automatically satisfy the restrictions imposed so far, it is useful to describe how the VSM's can be used, in those cases, to describe global geometry with the full freedom required by PTC. We must require that every *installed* element has a "fiducial point" that is nominally supposed to coincide with a particular VSM (which has to be present and identifiable in the *skeleton*). Furthermore each *installed* element has to have "fiducial orientation axes" (FOA) that nominally lie along corresponding axes in the design.

These requirements undoubtedly need to be spelled out more explicitly, but the idea should by now be clear. The location and orientation of every element can be made completely general by expressing the dispacement vector of the FP from its corresponding VSM and the FOA from its corresponding *design* orientation axes. Though these deviations can, in principle, be arbitrarily large, they will, in practice, be quite small. It may even be valid to make smallness assumptions that are, technically, unphysical, without introducing intolerable error. For example, it will probably always be legitimate to neglect the lack of commutativity of rotational misalignment; that is, the error caused by applying horizontal and vertical angular deviations in the wrong order is small compared to the uncertainties in the individual deviations.

5.7. Treatment of Lattices With Vertical Bends

The presence of *design* vertical bends and the noncoplanarity they cause complicate this description seriously enough to require the use of the fully general PTC patching mechanism (which might, within ADXF, more aptly be called a "stitching" mechanism). To the extent possible all simulations should try to use the planar case, with vertical bends treated as deviations from ideal. But we have to consider cases for which vertical bends are intentional and substantial. We must, at least, continue to demand a design orbit that is made up only of straight lines and circles. Even so, the coordinate triplets in the *skeleton* will no longer be coplanar.

For noncoplanar curves one's initial inclination is to to specify local axes *a là* Frenet-Serret, with tangential, centrifugal, and orthogonal-to-osculating-plane axes determined unambiguously, at least in curved regions. One problem with this is that the straight line—the most important lattice element—is degenerate from the Frenet-Serret perspective. That is, the straight line does not determine a unique centrifugal direction. One might try to overcome this impediment by requiring the "centrifigal direction" everywhere in a drift to be inherited from the centrifugal direction at the exit from the most recent drift. But this axis would then vary discontinuously at the next bend and, furthermore, the axes would depend on the forward/backward choice of sequencing. Worst of all, the axis would swing by an angle of order $\pi/2$ on entering even an arbitrarily weak vertical bend. Theoretically this could be tolerated but the presence of even a tiny bend would force the use of fully correct 3D trigonometry (rather than, say, small angle approximation). For these reasons the Frenet-Serret option has to be rejected.

A good choice of local frame orientation for communication among programs it to simply use the global (X, Y, Z) orientation. This choice has the unfortunate property, for a circular lattice that, unlike s, the closest thing to a "longitudinal" coordinate, namely Z, does not vary monotonically and passes through a point where it does not even have a longitudinal component. Nevertheless this choice of local frame orientation should be supported by ADXF.

Most accelerator simulation codes rely on a distinction between a longitudinal (tangential) coordinate and two transverse coordinates. It will be useful for ADXF to standardize the definition of such a frame unambiguously. This can start with the unambiguous unit tangent vector $\mathbf{t}$ (expressed in terms of $\hat{\mathbf{X}}$, $\hat{\mathbf{Y}}$, and $\hat{\mathbf{Z}}$) and arc length s at every point along every segment of the skeleton. (The discontinuous variation of $\mathbf{t}$ at kinks, if there are any, will have to be tolerated.) All that remains is to fix the zero of azimuth in the transverse plane. Since all accelerators remain close to a single horizontal plane, the global "vertical" $\mathbf{Y}$ vector tends to be everywhere roughly normal to $\mathbf{t}$. Then the vector $\mathbf{Y} \times \mathbf{t}$ (after normalization by factor close to 1) defines an unambiguous local "radial" coordinate x. The remaining, local, y direction can be defined by a further cross product, $\mathbf{y} = \mathbf{t} \times \mathbf{x}$. This definition of local axes avoids the erratic orientation assotiated with weak vertical bends as their bend angle approaches zero.

The suggestion is that an ADXF include parameterization of the relation between the two local frames described in the previous two paragraphs. For lattice without vertical bends this would be harmless redundancy. For roller coaster lattices the redundancy would help to avoid inevitable miscommunication.

Even for lattices containing vertical bends, many or most of the elements will, by design, be aligned relative to a single horizontal plane. Evolution in those regions can proceed as it would with planar lattices. For elements residing in non-horizontal regions the full PTC "patch" mechanism has to be employed.

CHAPTER F

Acronyms and Suffixes

While browsing the UAL directory tree to "get the big picture" it can be useful to see all files with a given file name extension. For example, one might wish to

TABLE F.1. GLOSSARY

ACCSIM	F. Jones Material/Bunch/Collimation code,[**32**]
ADXF	Accelerator Description Exchange Format[**43**]
AIM	Accelerator Instrumentation Module
ALE	Accelerator Libraries Extensions
APDF	Accelerator Propagator Description Format
API	Application Program Interface
CVS	Concurrent Version System
DA	Differential Algebra
DDD	Data Display Debugger
DOXYGEN	Documentation Generator
FTPOT	Fortran Teapot
MAD	Methodical Accelerator Design[**44**]
ICE	Incoherent and Coherent Effects (M. Blaskiewicz)
MPI	Message-Passing Interface[**45**] or if one prefers, MultiProcessor Interface
SIMBAD	Parallel 3-D Space Charge Calculations, Updating ORBIT [**48**]
PAC	Platform for Accelerator Codes[**?**]
PERL	Practical Extraction and Report Language
PERLXS	Perl eXternal Subroutine
SIF	Standard Input Format[**34**]
SMF	Standard Machine Format[**1**]
SNS	Spallation Neutron Source
SPINK	tracks polarized particles in circular accelerator
SVN	Subversion version control system
SXF	Standard eXchange Format[**42**]
TEAPOT	Thin Element Program for Optics and Tracking[**?**]
TIBETAN	Jie Wei's acceleration code
UAL	Unified Accelerator Libraries[**1**]
UAL2	Java accelerator-commissioning version of UAL[**47**]
UI	User Interface
XML	eXtensible Markup Language
XSLT	eXtensible Stylesheet Language Transformation
ZLIB	Y. Yan DA library,[**35**] [**36**]

TABLE F.2. FILENAME EXTENSIONS

.adxf	ADXF lattice description
.bs, .xs, .xsc, .PL	Perl/C++ interface
.c	C source code
.cc, .cpp	C++ source code
.css	Cascading Style Sheet
.hh, .h	C++/C header
.ll, .yy	lex, yacc parser support
.pl	Perl main program
.pm	Perl Module
.map, .zmap	DaVTps map
.cfg	configuration data
.so	shared library
.sxf	SXF lattice description
.xml	XML file
.xsl	XSLT transformation (of XML) file

see all available examples. All such examples are Perl main programs, and all such programs have the file name extension `.pl`. The rough area of applicability of a script can be inferred from the name of the directory in which it is contained. Here is an instruction to list all examples, followed by two lines of response:

```
   % cd $UAL; find . -name \"\,*.pl\" -print
...
./examples/ShortMPI/test_MPI.pl
./examples/UI/shell_sns.pl
...
```

As mentioned before, the scripts in directories other than `./examples,` though out-dated, can be browsed and, in most cases, run. Some may be useful for code test purposes.

CHAPTER G

Lattice Parameter Definitions

The global (X, Y, Z) and local (x, y, s) coordinate systems used by UAL are identical to those used by MAD[**44**]. The global survey code was ported from that source, as was much of the discussion in this section. There is a strong prejudice towards having the reference orbit lie close to a plane perpendicular to the Z-axis.[1]

The `survey` command reconstructs the global coordinates of all elements in the lattice. For this purpose all elements except RBEND's and SBEND's are treated as drifts. Elements like KICKER's, which could, in practice, influence the closed orbit, are assumed to have been set to zero. Similarly, QUAD's etc., which would steer the beam centroid if they were misaligned, are assumed not to be misaligned for purposes of first calculating the ideal closed orbit. For the lattice to make sense as a storage ring the lattice should "close", but exact closure (or any degree of closure whatsoever) is not required. In multiturn tracking every particle is displaced to take up any closure defect as its orbit passes the origin. Though unphysical, this discontinuous translation is at least symplectic and is unlikely to cause any harm to a simulation, provided it is not very large.

Unlike the `survey` command, the `analysis` command calculates the closed orbit in the presence of all steering perturbations just mentioned, and also includes any vertical steering due to RBEND and SBEND roll errors.

The global coordinates, both displacements and angles relative to the global (X, Y, Z) frame, are exhibited in Fig. G.1, which is a revised (but not intentionally changed) version of a figure in the MAD manual.[**44**] Fig. G.2 shows the beginning of the reference trajectory. It starts, by definition, at $(X, Y, Z) = (0, 0, 0)$ and is directed along the global Z-axis. A local coordinate triad for the reference orbit passing through global position (X, Y, Z) can be obtained by successively applying,

[1] Within the TEAPOT module the prejudice toward the reference orbit lying in a single plane goes so far as to exclude non-horizontal RBEND and SBEND components. Out of plane reference orbits have to be modeled by KICK elements. This restriction could be, but has not been, lifted using the Element-Algorithm-Probe framework. In any case this restriction is orthogonal to the definition of both local and global coordinates.

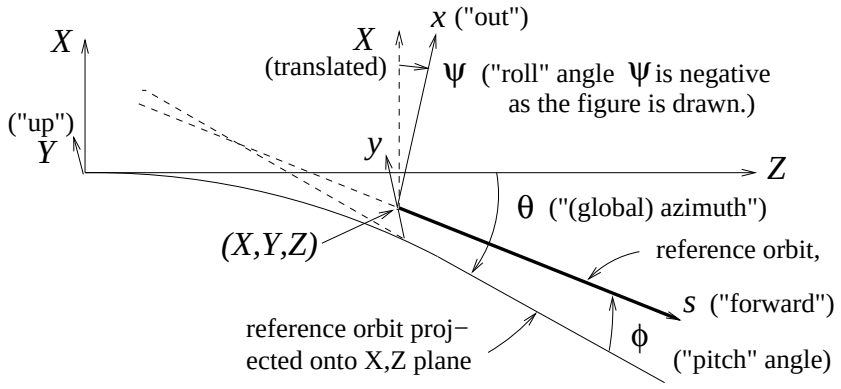

FIGURE G.1. Global coordinate definitions showing angles that define the direction of the reference orbit relative to the global (X, Y, Z) frame. Copied (with revision, but no intentional change) from MAD[**44**].

to the global triad, the rotation matrix $\mathbf{W} = \Theta\Phi\Psi$, where Θ, Φ, and Ψ are 3×3-matrices, expressed in terms of the respective angles θ, ϕ, and ψ shown in Fig. G.1:[2]

$$\Theta = \begin{pmatrix} \cos\theta & 0 & -\sin\theta \\ 0 & 1 & 0 \\ \sin\theta & 0 & \cos\theta \end{pmatrix}$$
$$\Phi = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\phi & \sin\phi \\ 0 & -\sin\phi & \cos\phi \end{pmatrix} \tag{G.1}$$
$$\Psi = \begin{pmatrix} \cos\psi & -\sin\psi & 0 \\ \sin\psi & \cos\psi & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

The local s-axis points along the reference orbit, the local y-axis is (ordinarily) parallel to the magnetic field axis, and x is chosen to be positive "outwards"; y-axis orientation is fixed by the requirement that the (x, y, s) triad be right-handed. The signs depend on the sign of the particle charge and the magnetic field. Fig. G.2 is drawn assuming positively charged particles in an accelerator with bending magnetic field directed along the positive Y axis. (For these choices the entire accelerator lies in the negative X half-space.) Because the x and y axes would vary erratically (i.e. angle ψ would be erratic) if referred to the local magnetic field direction, the best policy is probably to require the y axis to be always parallel to the Y axis, which defines a kind of globally-averaged magnetic field direction. In any case it is required that there be no net "twist" of coordinate ψ while advancing completely around the ring. With the global position of the reference orbit specified by vector $\mathbf{V} = (X, Y, Z)^T$ and its local orientation by matrix $\mathbf{W}$, these quantities are updated element-by-element using the equations

$$\mathbf{V}_i = \mathbf{W}_{i-1}\mathbf{R}_i + \mathbf{V}_{i-1}, \quad \mathbf{W}_i = \mathbf{W}_{i-1}\mathbf{S}_i, \tag{G.2}$$

[2]The orientation of matrix Θ in Eq. (G.1) has been reversed relative to the formula given in the MAD manual in order to conform to Fig. G.1 while leaving global azimuth angle positive.

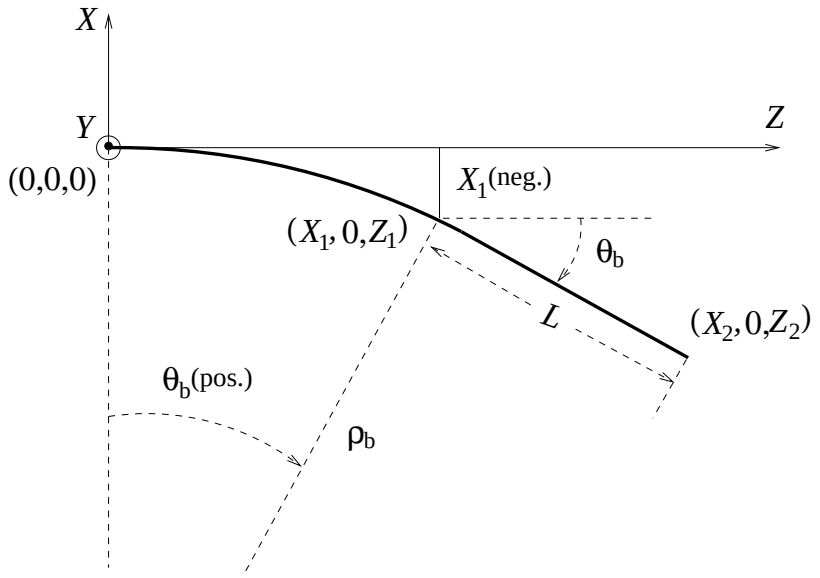

FIGURE G.2. The reference particle starts at the lattice origin and passes through a bend followed by a drift. The orientation of the orbit and the signs of the coordinates assume the particle charge is positive and that the magnetic field points (up) along the positive Y axis.

where, at the origin, $\mathbf{V}_0 = \mathbf{0}$, and $\mathbf{W}_0 = \mathbf{1}$ (the identity matrix) and where $\mathbf{R}_i$ and $\mathbf{S}_i$ are, respectively, the translation vector and the rotation matrix appropriate for the i-th element.

To illustrate this evolution, and especially the signs, consider the bend element at the beginning of the lattice in Fig. G.2. Its displacement vector $\mathbf{R}_1$ and rotation matrix $\mathbf{S}_1$ are

$$\mathbf{R}_1 = \begin{pmatrix} \rho_b(\cos\theta_b - 1) \\ 0 \\ \rho_b \sin\theta_b \end{pmatrix}, \quad \mathbf{S}_1 = \begin{pmatrix} \cos\theta_b & 0 & -\sin\theta_b \\ 0 & 1 & 0 \\ \sin\theta_b & 0 & \cos\theta_b \end{pmatrix}, \tag{G.3}$$

where radius of curvature ρ_b and bend angle θ_b are both positive (for the assumed charge and field direction). Then, by Eqs. (Survey.2), $\mathbf{V}_1 = \mathbf{R}_1$ and $\mathbf{W}_1 = \mathbf{S}_1$. For the drift that follows

$$\mathbf{R}_2 = \begin{pmatrix} 0 \\ 0 \\ L \end{pmatrix} \quad \mathbf{S}_2 = \mathbf{1}, \tag{G.4}$$

which yield

$$\mathbf{V}_2 = \mathbf{W}_1\mathbf{R}_2 + \mathbf{V}_1 = \begin{pmatrix} \cos\theta_b & 0 & -\sin\theta_b \\ 0 & 1 & 0 \\ \sin\theta_b & 0 & \cos\theta_b \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ L \end{pmatrix} + \begin{pmatrix} \rho_b(\cos\theta_b - 1) \\ 0 \\ \rho_b \sin\theta_b \end{pmatrix} \tag{G.5}$$

and

$$\mathbf{W}_2 = \mathbf{W}_1\mathbf{S}_2 = \begin{pmatrix} \cos\theta_b & 0 & -\sin\theta_b \\ 0 & 1 & 0 \\ \sin\theta_b & 0 & \cos\theta_b \end{pmatrix} \mathbf{1} \tag{G.6}$$

TABLE G.1. Comparison of various notations for phase-space variables.

general	x_1	x_2	x_3	x_4	x_5	x_6
Canonical	x	p_x	y	p_y	ct	$(E-E_0)/p_0c$
Courant-Snyder	x	$x' \equiv dx/ds$	y	$y' \equiv dy/ds$	Δt	$(p-p_0)/p_0$
TRANSPORT	X(1)	X(2)= $\sin^{-1} p_x/p_0$	X(3)	X(4)= $\sin^{-1} p_y/p_0$	X(5)= l	X(6)= $(p-p_0)/p_0c$
MAD	X	PX= p_x/p_0	Y	PY= p_y/p_0	DT= $-c\Delta t$	DELTAP= $(E-E_0)/p_0c$
TEAPOT	X	VXBYC= v_x/c	Y	VYBYC= v_y/c	n.a.	DELTA= $(p-p_0)/p_0$

7.1. Local particle coordinates

There has been an unfortunate lack of consistency in the definition of particle phase space coordinates in accelerator programs. In no case are the coordinates precisely equal to canonical coordinates (x, y, z, p_x, p_y, p_z). For the sake of generality and neutrality a generic set of coordinates will be referred to here as (x, y, l, f, g, h) and some of the choices are indicated in Table G.1. In all of the cases in the table (though not necessarily for other codes) longitudinal momenta offsets are "normalized" by a reference momentum p_0 or energy offsets are normalized by reference energy E_0.

The transverse (x, y) coordinates are common to all systems and, in all cases, the transverse "momentum" coordinates (f, g) are identical in the "paraxial" or "linearized" order of approximation. As a result it is relatively straightforward, because they are "first order" to compare transfer matrices produced by different programs. Quantities that can be extracted from the transfer matrices, such as tunes, are easily compared for the same reason. But to the next, and higher, orders there is no consistency. This makes it especially difficult to compare high order maps generated by different programs, or even elementary accelerator parameters such as chromaticity. Within UAL there needs to be provision for translating between pairs of these conventions; the most important combination being MAD/TEAPOT. UAL performs these translations transparently to the user.

Fortunately the coordinate choice option has no impact on the external description of the standard hardware elements that enter accelerator lattice descriptions.

CHAPTER H

Truncated Power Series and Lie Maps

8.1. Function evolution

Truncated power series play an important role in UAL. Their role is to approximate the "maps" that express "output" particle coordinates (at some place in the ring) in terms of "input" particle coordinates (at a different place in the ring). When truncated to linear order these power series reduce to the elements of the traditional, Courant-Snyder, transfer matrix description of the accelerator lattice. Historically, most of accelerator physics has been (very successfully) based on analysis performed in this limit. But effects appearing already at a "next order of approximation" such as chromaticity and amplitude-dependent detuning, have ways of intruding, even in elementary contexts, and nonlinearity becomes increasingly important as amplitudes are increased to achieve higher beam current. As soon as any nonlinearity whatsoever is allowed to enter the description the issue of symplecticity, or rather lack thereof, rears its head. Especially for hadron accelerators, for which there is essentially no true damping, any anti-damping artificially and erroneously introduced through non-symplecticity can ruin an accelerator simulation program's ability to predict the long term future.

Symplectic maps (typically nonlinear) are also known as Lie maps. One therefore seeks to describe particle trajectories in an accelerator by a Lie map. As with all physics, such a description can only be approximate. For one thing the idealized model of the accelerator, on which the "idealized map" is based, is undoubtedly inaccurate and incomplete. Accepting this as inevitable, possible further inaccuracy results from the computer program representation of the map. It is the latter source of inaccuracy that is the subject of this appendix. Maps based on truncated power series can only approximate idealized maps. For reasons explained in the previous paragraph, failure of symplecticity is expected to be more serious than other inaccuracy. An important goal of UAL is to preserve symplecticity, or rather to keep the inevitable failure of symplecticity controllably small.

There is no shortage of excellent reference material concerning Lie maps; for example Dragt[**39**] and Forest[**9**]. Because the subject is abstract, and is sometimes considered impenetrable, this appendix tries to give a self-contained, elementary discussion of the general ideas. To reduce complexity the discussion will be restricted to two dimensional, (x, p), phase space; (for simplicity p is used instead of p_x). All results generalize easily to higher dimensions.

If (x_0, p_0) represents input particle coordinates, the sort of map $\mathcal{M}'_{10}$ under discussion expresses output coordinates (x_1, p_1) as functions of input coordinates (x_0, p_0). For linear maps this map reduces to a 2×2 matrix, the traditional transfer matrix of standard accelerator theory. If nonlinearity is present it is natural to introduce a "generalized transfer matrix" $\mathcal{M}'_{10}$ in which the four matrix elements

are nonlinear functions of x_0 and p_0). Like it or not, this is the representation one is forced to use in a computer representation of the map.

Consider an arbitrary function $f(x,p)$—one may think of f as expressing the dependence on position in phase space of some physical quantity. A particle trajectory defines an evolution of the particle coordinates and it is natural to inquire about the corresponding evolution of f. One has to be aware of the ambiguity accompanying the distinction between function *form* and function *value*. For example, suppose transformation $\mathcal{M}'_{10}$ yields forward formula $x_1 = x_1(x_0,p_0) = ap_0 + bp_0$ and backward formula $x_0 = x_0(x_1,p_1) = cx_1 + dp_1$, and that the value of function f is defined to be "the first component squared"; at input this is x_0^2, at output it is x_1^2. An assignment one might have received in calculus class was to figure out the value of x_0^2 from knowledge only of x_1 and p_1. Expressed in terms of output coordinates the input value of f is $(x_0(x_1,p_1))^2 = (cx_1 + dp_1)^2$. From a physicist's point of view, this is tortured usage. By the "evolved value of f" one presumably means x_1^2, the square of the first component, evaluated at the evolved location. This is the way functions of coordinates are to be interpreted; for example

$$x_1^2 = f(x_1,p_1) = f(\mathcal{M}'_{10}(x_0), \mathcal{M}'_{10}(p_0)) = (ax_0 + bp_0)^2. \tag{H.1}$$

Since the *form* of the function does not change, to evaluate this evolution, as Eq. (H.1) shows, it is adequate to have formulas for the evolution of individual components. This is the functionality provided by the vectors of truncated power series provided, for example, by UAL. But, for theoretical purposes, a slightly more abstract generalization of transfer matrices is preferable. Let us define transfer map $\mathcal{M}_{10}$ as operating on *functions* (of location phase space) rather than acting individually on the components. That is

$$f_1 = \mathcal{M}_{10} f_0, \tag{H.2}$$

which is defined to mean the same thing as Eq. (H.1). Forest calls $\mathcal{M}$ a "compositional map". It is a one-component map acting in an infinite dimensional space (of functions defined on phase space.) Note that it is the *value* of the function that evolves; the *form* of the function does not change. Since x_0 and p_0 can, individually, be thought of as functions of the (x_0,p_0) pair, the specialization back to the representation by a vector-organized set of nonlinear functions is immediate. So there is no "physics" in Eq. (H.2) to distinguish it from Eq. (H.1).

Assuming, as we are, that the physical elements in the lattice are known perfectly, the equations of motion can, in principle, be used to determine $x(s), p(s)$, the dependence on longitudinal coordinate s of a particle trajectory. Commonly the equations of motion are written in Hamiltonian form and knowing the equation of motion is sometimes expressed as "knowing the Hamiltonian". Because of the complexity of accelerator lattices it is almost never practical to solve the equations of motion analytically and it is rarely practical to solve them numerically. Rather the map through a sector of the lattice is formed by concatenating the maps of the individual elements in the sector. This usually involves truncation of power series.

8.2. Taylor series in more than one dimension and Lie maps

The Taylor series representation of one dimensional functions is second nature to most scientists (perhaps because learned about in high school as the binomial

theorem?) The function of Lie maps is to generalize this description to more than one dimension.

The theory of function evolution, as invented by Lie, has been applied a century later, in the context of celestial mechanics, by Hori[**30**] and, in the context of accelerator mechanics, by Dragt.[**40**] The discussion here more nearly follows Hori than Dragt.

Let (x, p) be coordinates in 2D phase space, and $f(x, p)$ be a function that is arbitrary (except for possible requirements such as smoothness and absence of vanishing derivatives.) We wish to express the value of f at some phase space point in terms of the values of its derivatives at some other point.

We know how to do this in 1D—use a Taylor series. We therefore try to reduce the 2D problem to 1D. Toward this end we draw a family of smooth curves in phase space (to be referred to as a "congruence" of curves) that have properties: (a) there is a curve through every point, (b) no curve crosses any other in the region under discussion, and (c) there is a function $S(x, p)$, not necessarily unique, such that $x(\tau), p(\tau)$ (the coordinates of the curve as functions of a running parameter τ) are solutions of the equations

$$\frac{dx}{d\tau} = \frac{\partial S}{\partial p}, \quad \frac{dp}{d\tau} = -\frac{\partial S}{\partial x}. \tag{H.3}$$

The function $S(x, p)$ is such that its derivatives on the right hand side of this equation define, at every point (x, p), the direction of the tangent to the curve passing through that point. Note that S is an arbitrary function.

Along any one of the curves of the congruence, the value of arbitrary function f can be expressed, as a function of τ, by $f(x(\tau), p(\tau))$. One can define an along-the-curve derivative operator

$$\{\cdot, S\} \equiv \left.\frac{d}{d\tau}\right|_S = \frac{dx}{d\tau}\frac{\partial}{\partial x} + \frac{dp}{d\tau}\frac{\partial}{\partial p} = \frac{\partial S}{\partial p}\frac{\partial}{\partial x} - \frac{\partial S}{\partial x}\frac{\partial}{\partial p}. \tag{H.4}$$

In this notation the $\cdot$ is a "place holder" indicating the operator $\{\cdot, S\}$ is "waiting for" a function, such as f, for its argument. (Except for change in sign/order-of-arguments, $\{\cdot, S\}$ is the same as the function for which Dragt introduced the notation : S :.) When acting on function f, the result is $\{f, S\}$, which can be recognized as the "Poisson bracket" of f and S.

Now we can exploit our congruence of curves for its advertised purpose of relating values of f at separated points, at least if the points lie on the same curve because, on that curve, the function depends only on the single variable τ. Let the parameters of the points that are to be related be τ and $\tau + \epsilon$. It may be helpful conceptually to regard ϵ as being "small", and this may be appropriate when discussing the convergence of the series, but no such formal requirement is assumed. Expressing the Taylor series in somewhat unconventional form, we have

$$f(\tau+\epsilon) = \left(1 + \epsilon\{\cdot, S\} + \frac{1}{2!}\epsilon^2\{\{\cdot, S\}, S\} + \frac{1}{3!}\epsilon^3\{\{\{\cdot, S\}, S\}, S\} + \cdots\right) f(\tau+\epsilon)\Big|_{\epsilon=0}, \tag{H.5}$$

As usual the derivatives on the right hand side must be evaluated for general ϵ but then ϵ is set to zero. This is known as the Lie map corresponding to function S. Recognizing the terms in this series as corresponding to an exponential function,

this series is traditionally abbreviated to

$$f(\tau+\epsilon) = e^{\epsilon\,\{\cdot,S\}}\, f(\tau); \tag{H.6}$$

but, to evaluate the series, expansion Eq. (H.5) is what is required. Furthermore the evaluation has to be truncated at some point. Any differential algebra package, such as COSY[**41**] or the ZLIB module of UAL, can calculate derivatives of functions, and can therefore evaluate the Poisson bracket expressions appearing in Eq. (H.5).

This section has been about calculus, no more, no less. There has been no mechanics, Hamiltonian or otherwise. If the signs in Eq. (H.3) had been chosen differently, say both positive, the analysis would go through unchanged except for the switching the sign in the bracket expression, which would therefore no longer deserve be called a "Poisson bracket".

8.3. Symplecticity of Lie map

Hori[**30**] gave a different interpretation to Eq. (H.6), regarding it as a change of variable rather than as an evolution equation. To encourage this interpretation let us replace (x_0, p_0) by (ξ, η) and (x_1, p_1) by (x, p) and interpret the equation as a change of variables from (ξ, η) coordinates to (x, p) coordinates. The coordinates (ξ, η) are assumed to be "canonical"—this means that their Poisson brackets reckoned using some known-to-be canonical starting coordinates, call them (x', p'), have the appropriate, 0 or 1 values. Copying from Eq. (H.5) and restoring the 2D arguments of f;

$$f(x,p) = (1+\epsilon\,\{\cdot,S\}+\frac{1}{2!}\epsilon^2\,\{\{\cdot,S\},S\}+\frac{1}{3!}\epsilon^2\,\{\{\{\cdot,S\},S\},S\}+\cdots)f(\xi,\eta)\Big|_0 . \tag{H.7}$$

Here S is, as before, an arbitrary function, and evaluation of the derivatives on the right hand side depends upon the congruence of curves determined by Eqs. (H.3). (The cryptic subscript 0 is supposed to convey this.)

It was mentioned above that either one of the coordinates, say ξ, is a satisfactory version of the function f. Plugging this into Eq. (H.7) yields

$$x = (1+\epsilon\,\{\cdot,S\}+\frac{1}{2!}\epsilon^2\,\{\{\cdot,S\},S\}+\frac{1}{3!}\epsilon^2\,\{\{\{\cdot,S\},S\},S\}+\cdots)\xi\Big|_0 , \tag{H.8}$$

and a similar formula relates p to η. By restoring the single variable, along-curve parameterization (and for compactness introducing a vector display) these equations can be written in a more useful form;

$$\begin{pmatrix}\xi(\tau+\epsilon)\\ \eta(\tau+\epsilon)\end{pmatrix} = \begin{pmatrix}x\\ p\end{pmatrix} \tag{H.9}$$

$$= (1+\epsilon\,\{\cdot,S\}+1/2!\epsilon^2\,\{\{\cdot,S\},S\}+1/3!\epsilon^2\,\{\{\{\cdot,S\},S\},S\}+\cdots)\begin{pmatrix}\xi(\tau+\epsilon)\\ \eta(\tau+\epsilon)\end{pmatrix}\Big|_{\epsilon=0}$$

This shows that the pair (x, p) are, except for "translation" along a curve of the congruence, the same as the pair (ξ, η).

This has still been "just calculus", but let us now use the assumption that (ξ, η) are canonical variables of a Hamiltonian system. Then Eq. (H.9) provides a change of variables to new variables (x, p). Now the amazing part; since the (ξ, η) variables

are, by hypothesis canonical through the region under discussion and (x, p) are just "translations" of (ξ, η), transformation (eq:Tps.6q) is necessarily canonical.

Hori[**30**] goes on to develop a perturbation theory based on this formulism. He regards the function S as a *kind of* "generating function" (though it must not be confused with a "Goldstein" generating function) and goes on to develop an iterative procedure to determine S and new coordinates in ascending powers of a "small parameter" of the perturbation. None of this is relevant for UAL. What *is* relevant is that transformations generated by Lie maps are symplectic. By controlling the number of terms retained in the power series evaluation one can control (or even make negligible) the degree of nonsymplecticity.

8.4. Hamiltonian maps

Returning to the trajectory evolution interpretation of our equations, the Taylor series derived so far might seem to be useless for the following reason: it relates only phase space points lying on the same curve and no prescription has been given for choosing the function $S(x, p)$ such that two arbitrarily chosen points lie on the same curve. But, as it happens, we do not have to insist that the points be arbitrarily chosen. We are interested in points lying on a single particle trajectory. One visualizes this trajectory as a three dimensional curve in the (x, p, t) space, where t is time, or if one prefers, a longitudinal coordinate. Projected onto the (x, p) plane the curve passing through input point (x_0, p_0) necessarily passes through output point (x_1, p_1). The orbit is determined by solving Hamilton's equations;

$$\frac{dx}{dt} = \frac{\partial H}{\partial p}, \quad \frac{dp}{dt} = -\frac{\partial H}{\partial x}. \tag{H.10}$$

where $H(x, p)$ is the Hamiltonian function. Notice that these equations are identical to Eqs. (H.3) if the function S in those equations is replaced by H (and τ by t.) This magically eliminates both limitations of the formalism of the previous section. The map has become

$$f(t_0 + t) = e^{t\,\{\cdot, H\}}\, f(t_0). \tag{H.11}$$

(As explained above, when written in this form, this notation is too compressed for the required operations to be exhibited explicitly, as they are in Eq. (H.5).) Replacing f by the individual coordinates, as before, yields

$$\begin{pmatrix} x(t_0 + t) \\ p(t_0 + t) \end{pmatrix} = e^{t\,\{\cdot, H\}} \begin{pmatrix} x(t_0) \\ p(t_0) \end{pmatrix}. \tag{H.12}$$

Generalized to six dimensions and truncated to arbitrary order, Eq. (H.12) is a form in which the evolution of a particle trajectory can be simulated in a computer. If Hamiltonian H is only approximate the evolution it produces can be only approximate, but any failure of symplecticity can be reduced by keeping more terms in the expansion.

8.5. Discrete maps

Eq. (H.12) represents a continuous mapping—the explicit appearance of t invites taking the limit $t \to 0$. Similarly the occurence of factor ϵ in Eqs. (H.9) invites the limit $\epsilon \to 0$ and a continuous interpretation. But, if the ϵ factor is subsumed into the S function, Eqs. (H.6) represents a discrete map, potentially propagating the particle coordinates through a sector of arbitrary length.

For example consider the function

$$S = S_0^3 x^3 + S_1^3 x^2 p + S_2^3 x p^2 + S_3^3 p^3. \tag{H.13}$$

Substitution into Eq. (H.9) yields propagation $(x, p) \to (x', p')$

$$\begin{aligned} x' &= x + \{x, S\} + \cdots = x + S_1^3 x^2 + 2S_2^3 xp + 3S_3^3 p^2 + \ldots, \\ p' &= p + \{p, S\} + \cdots = p - 3S_0^3 x^2 - 2S_1^3 xp - S_2^3 p^2 + \ldots. \end{aligned} \tag{H.14}$$

This map is special in that it is an identity map to linear order. It could therefore not represent arbitrary propagation through a general sector. But, after "factoring out" the linear part of a general map the remaining part could be reduced to Eq. (H.13) by truncation to quadratic order.

Perhaps the procedure just mentioned can be reversed? Suppose that propagation formulas (eq:Discrete.2) have been determined by applying some integrator to an arbitrary lattice sector. If the sector has more than a few nonlinear elements such a determination would have required truncation, for example to quadratic order, as in Eq. (H.14). The integrator will therefore have determined the coefficients in expansions

$$\begin{aligned} x' &= x + X_0^2 x^2 + X_1^2 xp + X_2^2 p^2 + \ldots, \\ p' &= p + P_0^2 x^2 + P_1^2 xp + P_2^2 p^2 + \ldots. \end{aligned} \tag{H.15}$$

For these equations to be consistent with Eqs. (H.14) the six equations obtained by equating coefficients must be satisfied. Regarding the four S_i^3 coordinates as the unknowns, they can be determined from just four of the equations. The remaining two equations will not, in general, be satisfied. But, if the integrator determining series (eq:Discrete.3) were symplectic (to the order of terms retained), then these equations would be redundant and the redundant equations would necessarily be satisfied. These equations can therefore be applied as a check on the symplecticity of the integrator.

Assuming the integrator is symplectic so that the redundant equations (to quadratic order) are satisfied, the function S will have been determined to cubic order. A function S determined in this way can be called a "pseudo-Hamiltonian".

By using this function in Eq. (H.9), and retaining more terms in the series, propagation formulas for the coordinates can be obtained to higher than quadratic order. Such formulas would be useless for studying large amplitude features such as resonant islands, onset of chaos, or dynamic aperture. But for "intermediate" amplitude trajectories the formulas can represent propagation that is both "correct to quadratic order" (for example modeling chromaticity) while being symplectic to higher than quadratic order.

This procedure can be illustrated by explicit example. Consider a map

$$\mathbf{x}_2 = \mathbf{M}\,\mathbf{x}_1 \approx \mathbf{M}^{(1)}\,\mathbf{x}_1, \tag{H.16}$$

where $\mathbf{M}^{(1)}$ is the necessarily symplectic, linearized matrix approximation of the map. (Since $\mathbf{x}$ represents the components as a vector, we may as well take it to represent the coordinates in 6D phase space.) Define $\widetilde{\mathbf{M}}$ such that

$$\mathbf{x}_2 = \widetilde{\mathbf{M}}\,\mathbf{M}^{(1)}\,\mathbf{x}_1, \quad \text{or} \quad \widetilde{\mathbf{M}} = \mathbf{M}\,{\mathbf{M}^{(1)}}^{-1}. \tag{H.17}$$

Suppose that $\mathbf{M}$ has been obtained to some order of accuracy, say $\mathbf{M}^{(2)}$. Then $\widetilde{\mathbf{M}}$ is known to corresponding order. Let S be determined such that

$$\widetilde{\mathbf{M}}^{(2)} = \mathbf{M}^{(2)}\,\mathbf{M}^{(1)\,-1} = 1 + \{\cdot, S\}. \tag{H.18}$$

Defining

$$\widetilde{\mathbf{M}}^{(3)} = 1 + \{\cdot, S\} + \frac{1}{2}\,\{\{\cdot, S\}, S, \}, \tag{H.19}$$

then

$$\mathbf{M} \approx \widetilde{\mathbf{M}}^{(3)}\,\mathbf{M}^{(1)}, \tag{H.20}$$

is symplectic to higher order than was $\widetilde{\mathbf{M}}^{(2)}$. The quadrupole end field correction described in section 4.4.2 is an example of this procedure. Since the longitudinal interval for this correction was taken to have zero length, terms beyond the first vanish because they are proportional to higher powers of ϵ.

Bibliography

[1] N. Malitsky and R. Talman, *Unified Accelerator Libraries,* AIP 391, Williamsburg, 1996

[2] N. Malitsky and R. Talman, *Status of Unified Accelerator Libraries,* IEEE Particle Accelerator Conference, p. 2434, 1997

[3] In high energy physics there is a facility called ROOT R. Brun et al., *An Object-Oriented Data Analysis Framework,* `http://root.cern.ch/root` whose purpose is to unify data analysis of elementary particle physics experiments. Except for working on accelerators instead of experiments the UAL motivation and design principles are the same. We use the term "environment" instead of "framework" only to reserve the latter term for a slightly different sense below

[4] QT website, `http://www.trolltech.com`

[5] R. Brun et al., *The ROOT Framework,* AIHEPNH-96, Lausanne, Switzerland, August 1996

[6] V. Fine, *Cross-Platform Qt-Based Implementation of the Low Level GUI Layer of ROOT,* ACAT'2002 Workshop, NIM **502**, Issues 2-3, April 2003

[7] N. Malitsky and R. Talman, *UAL User Guide,* BNL-71010-2003, available at `http://www.ual.bnl.gov`, 2003

[8] N. Malitsky and R. Talman, *UAL-USPAS: Text for UAL Accelerator Simulation Course,* Cornell, 2005

[9] E. Forest, *Beam Dynamics, A New Attitude and Framework,* Harwood Academic Publishers, Amsterdam, 1998, p. 390. Forest explains why the seemingly-unphysical discontinuity in particle trajectory occurring at a quadrupole edge is, in fact, a best approximation, at a point, of an effect that actually accumulates over an extended longitudinal interval.

[10] R. Talman and N. Malitsky, *Beam-Based BPM Alignment,* BNL/SNS Technical Note No. 116, September 16, `http://server.ags.bnl.gov/bnlags/bnlsns/116.pdf`, 2002

[11] L. Schachinger and R. Talman, *Teapot: A Thin-Element Accelerator Program for Optics and Tracking,* Part. Accel. **22**, 35, 1987

[12] C. Wang, V. Sajaev, and C. Yao, *Phase advance and β function measurements using model-independent analysis,* PRST-AB **6**, 104001 (2003)

[13] J. Irwin, C. Wang et al., *Model-Independent Beam Dynamics Analysis,* Phys. Rev. Letters, **82**, 1684 (1999)

[14] I. Jolliffe, *Principle Component Analysis*, Second Edition, Springer, 2002

[15] F. Press et al., *Numerical Recipes*, ISBN 0-521-43108-5, Cambridge University Press, 1992

[16] Koutchouk, J-P, *Trajectory and Closed Orbit Correction*, in Frontiers of Particle Beams; Observation, Diagnosis and Correction, Joint US-CERN School on Particle Accelerators, M. Month and S. Turner, (Eds.), Anacapri, Italy, 1988

[17] A. Verdier and T. Risselada, CERN/ISR-BOM-OP/82-19(1982)

[18] G. Bourianoff, S. Hunt, D. Mathieson, F. Pilat, R. Talman, G. Morpurgo, *Determination of Coupled-Lattice Properties Using Turn-By-Turn Data*, SSCL-Preprint-181, 1992

[19] F.R. Gantmacher, *The Theory of Matrices*, Chelsea, New York, 1977, Section II§5

[20] W. Press, B. Flannery, S. Teukolsky, and W. Vetterling, *Numerical Recipes in C,* Cambridge University Press, Cambridge, 1988, p156

[21] Jie Wie, *Longitudinal Dynamics of the Non-Adiabatic Regime on Alternating Gradient Synchrotrons,* SUNY Stony Brook PhD thesis, 1990, revised 1994

[22] C. Montag and J. Kewisch, *Commissioning of a first-order matched transition jump at the Brookhaven Relativistic Heavy Ion Collider*, PRST-AB, **7**, 011001 (2004)

[23] R. Talman, *Theory of Head-Tail Chromaticity Sharing,* Cornell Report, CBN 97-16 (1997)

[24] L. Mandel and E. Wolf, *Optical Coherence and Quantum Optics*, Cambridge, p. 128 (1995)

[25] R. Meller, A. Chao, J. Peterson, S. Peggs, and M. Furman, *Decoherence of Kicked Beams*, SSC-N-360 (1987)

[26] Y. Okamoto and R. Talman, *Rational Approximation of the Complex Error Function and the Electric field of a Two-Dimensional Gaussian Charge Distribution*, Cornell Report, CBN 80-13 (1980)

[27] M. Bassetti and G. Erskine, *Closed expression for the electrical field of a two-dimensional Gaussian charge*, CERN-ISR-TH/80-06

[28] M. Abramowitz and I.A. Stegun, *Handbook of Mathematical Functions,* National Bureau of Standards, 1966

[29] A. Dragt, *Lectures on Nonlinear Orbit Dynamics,* American Institute of Physics, New York, 1981

[30] G. Hori, *Theory of General Perturbations with Unspecified Canonical Variables,* Publ. Astro. Soc. Japan, Vol. 18, No. 4, 1966

[31] A. Dragt, in *Proceedings of the 1984 Study on the Design and Utilization of the SSC*, ed. by R. Donaldson and J. Morphin, Am. Phys. Soc., Snowmass, 1985

[32] F. Jones, http://www.triumf.ca/compserv/accsim.doc/refguide.ps, *User's Guide to Accsim,* 1990

[33] J. Galambos et al. *ORBIT User Manual,* Version 1.10, SNS/ORNL/AP Technical Note Number 011, Rev. 1

[34] D. Carey and F. Iselin, *Standard Input Language for Particle Beams,* Snowmass, Colorado, 1984, MAD etc. input language

[35] Y. Yan and C-Y Yan, *A Numerical Library for Differential Algebra,* SSCL-300, 1990

[36] N. Malitsky, A. Reshatov, and Y. Yan, *ZLIB++: Object-Oriented Library for Differential Algebra,* SSCL-659, 1994

[37] S. Peggs et al. *LAMBDA Manual,* RHIC/AP/13, 1993

[38] N. Malitsky, *A Prototype of the SNS Optics Database,* BNL/SNS Technical Note No. 085, November, 2000

[39] A. Dragt, *Lectures on Nonlinear Orbit Dynamics,* American Institute of Physics, New York, 1981

[40] A. Dragt, in *Proceedings of the 1984 Study on the Design and Utilization of the SSC*, ed. by R. Donaldson and J. Morphin, Am. Phys. Soc., Snowmass, 1985

[41] M. Berz, *Differential Algebraic Description of Beam Dynamics to Very High Orders,* Particle Accelerators, **24**, 109, 1989

[42] H. Grote, J. Holt, N. Malitsky, F. Pilat, R. Talman, C. Trahern, *SXF: Definition, Syntax, Examples,* RHIC/AP/155, 1998

[43] N. Malitsky and R. Talman, *Accelerator Description Exchange Format,* ICAP98, Monterey, 1998

[44] H. Grote and F. Iselin, *The MAD Program (Methodical Accelerator Design), User's Reference Manual,* CERN/SL/90-13(AP), 1990

[45] N. Malitsky and A. Shishlo, *A Parallel Extension of the UAL Environment,* PAC, 2001

[46] G. Bourianoff, A. Reshatov, and N. Malitsky, *Object-Oriented Approach for the Design of the Simulation Facility of the SSC,* SSCL-667, 1994

[47] N. Malitsky, J. Smith, and J. Wei, *A Prototype of the UAL 2.0 Application Toolkit,* 8th Int. Conf. on Accelerator and Large Experimental Phys. Control Systems, San Jose, CA, 2001

[48] N. D'Imperio, A¿ Luccio, N. Malitsky, and O. Boine-Frankenheim, *Parallel 3-D Space Charge Calculations in the Unified Acceleratory Library,* BNL-77 07 6-2006-CP, 2006

Index

ACCSIM, 2
address-of, 11
adjuster, 40, 41
adjusters, 43
ADXF, 65
adxf
  file, 5
AIM, 2, 65
ALE, 32, 33, 37, 38, 40–42
algorithm, 35, 36, 44
analysis, 39
aperture, 36
  dynamic, 76
  shape, 36
API, 2
argument
  of function, 11
  direct, 11
  indirect, 11

BPM, 40, 41

C++
  introductions to), 6
  references, 6
case (in)sensitivity, 35
checkout, 32
class, 8
code checkout, 32
command
  basic, 31
  UAL, 31
complexity index, 35
computation time, 35
CVS, 40, 42, 65

DA, 9, 45–47, 65, 66
DDD, 32, 38
debug, 32, 38
declaration, 10, 11
definition, 11
detector, 40, 41
detectors, 43
differential algebra, 9, 36, 74
divisibility index, 35
doxygen, 32

element-algorithm-probe, 7
Element-Algorithm-Probe framework, 50
encapsulation, 8
environment, 31, 32, 40, 41, 79
erect multipole, 38
error, 36
  magnetic field, 32, 37, 49
  message, 42
example
  annotated, 5

family
  explicit enumeration, 35
  reg. exp. definition, 35
FastTeapot, 50
file handle, 38
fonts in guide, 2
FORTRAN, 8, 36, 65
FTPOT, 36, 65
full-instantiation, 14

global
  geometry, 67
  Perl scope, 42, 46
  survey, 39

hamiltonian, 46, 72, 75
hard edge, 46
hsteer, 43, 44
http, 65

ICE, 2
inheritance, 8, 40
integrator, 45–47
  simplectic, 76
interface, 50, 65
  personalized, 42
ir
  complexity index, 35
ir, complexity index, 35

kicker, 41

lattice description, 35, 36, 41, 44, 48–50, 70
lattice function weighted sums, 51
LHC, 31, 49
Lie
  integrator, 45–47
  map, 45, 71–73
  transform, 44
Linux, 50

MAD, 32, 33, 35, 41, 45, 65, 70
makethin, 36
manual
  C++interface, 1
  PERL-interface, 2
  physics, 2
  USPAS school, 2
map
  -kick track comparison, 50
  application, 48
  compositional, 72
  cubic, 37, 44, 48
  discrete, 75
  fringe field, 44, 47, 49
  generation, 33, 45
  Hamiltonian, 75
  identity, 45
  Lie, 45, 71–75
  linear, 37, 71
  once-around, 37
  order, 37
  polynomial, 44
  pseudo-Hamiltonian, 75
  sector, 50
  symplectic, 35, 44, 71, 74, 75
  Taylor, 32
  transfer, 72
  truncated, 35, 44, 71
  UAL, 37
method, 6
mkpath, Perl command, 33
modularity, 6
module, 2
  UAL, 35
multipole
  coefficients, 38
  erect, 38
  skew, 38

object, 8
object orientation
  inheritance, 40
  overloading, 46
object-oriented, 6
  design principles, 8
ORBIT, 2, 65
order, of polynomial, 33
overloading, polymorphism, 47
overriding, 6, 8

PAC, 1, 2, 33, 65
package, Perl, 42
Perl
  debug, 32
  mkpath, 33
  package, 42
  reference, 41
  typeglob, 46
pointer, 11
Poisson bracket, 45, 73, 74
polymorphism, 9
post-processing, 51
probe, 7
procedural, 2, 6
pseudo-hamiltonian, 44, 46, 75, 76

reference
  Perl, 41
regular expression, 35, 37, 39
RHIC, 31, 49, 66

scope, 10
  global, 42, 46
shell, 10
  user, 2, 32
shift, Perl statement, 41, 42, 46
SIF, 36, 41, 44, 49, 50, 65
SIMBAD, 2
skew multipole, 38
SMF, 31, 49, 65
SNS, 31, 32, 34, 39–41, 44
SPINK, 2
steer, 43, 44
strong typing, 10
subdivision, element, 35
survey, 67
SXF, 31, 49, 65

Taylor series, 2D, 72
TEAPOT, 1, 2, 33, 35, 36, 49, 65, 70
template, 10
TIBETAN, 2, 65
TPS, 9
trucated power series, 9
Twiss, 33, 36
twiss output, 39
type, 10
typeglob, 46

UAL
  code source, ii
  environment, 1
  infrastructure, 1
  module, 1
  URL, 2
URL, 65
US-LHC, 49
use
  Carp, Perl command, 42

lib, Perl command, 43, 45
Perl command, 33
shell command, 36
strict, Perl command, 42
TEAPOT command, 36
vars, Perl command, 45
USPAS, 2

VTPS, 9

XML, 65

ZLIB, 1, 2, 33, 36, 37, 44, 45, 48, 65, 74
maximum order, 33, 45