

ORBIT Code Development Recent Progress

Jean-Francois Ostiguy
Beam Physics Department
FNAL
ostiguy@fnal.gov

Collaborators ...

Jeff Holmes SNS/ORNL

Leo Michelotti BD/BP FNAL

Weiren Chou BD/BP FNAL

Why ORBIT ?

- We considered a variety of existing codes
- ORBIT was selected as our primary tool because of:
 - Source code and documentation publicly available
 - Well-developed diagnostics (tune footprints, moment evolution etc ..)
 - Some validation at existing machines (PSR)
 - Support for both decoupled transverse and longitudinal ($2^{1/2}D$) and 3D space charge
 - Support for parallel execution (MPI)
- Note: (Synergia) - essentially a derivative of IMPACT (developed to study halo in high intensity linacs) is also under development at FNAL. The focus is "full 3D" simulation in synchrotrons. Cross checks are useful and important for validation of both ORBIT and Synergia.

Beam Physics Dept Parallel Linux Cluster



32 2-CPU Nodes (1.4 Ghz AMD Athlon)
1 Gbyte RAM / Node
Gigabit Ethernet
Total Cost: 65 K\$

Adequate for $2^{1/2}$ D simulations.
100-1000 turns with $O(10^5)$ macro particles.

Code Development

Recent development efforts at Fermilab have been focused on:

- Support for high order maps
- MAD parser
- A high level Python shell
- Correct tune Footprint Computation
- Better support for Acceleration (e.g. multipoles, transition etc...)

Machine Description

- Until recently, ORBIT had been relying on MAD or DIMAD to produce maps and lattice functions. The lattice information is read from MAD (ascii) output.
- As a consequence, ORBIT could not internally recompute maps.
- The process of importing a machine description into ORBIT is cumbersome and potentially error prone (e.g. changes in MAD output file format between different versions/platforms)

A Lex/Yacc based MAD parser

- A Lex/Yacc-based MAD parser was developed in the BP dept a few years ago for internal needs.
- Designed as a generic system usable either for off-line translation to another human-readable description language or for dynamic definition of objects.
- Few restrictions (no abbreviations, no action commands, no use of undefined variables)
- Successfully validated on very large lattice files (e.g. complete Tevatron lattice)
- BTW: The parser is also used by Synergia

Proceedings of the 2001 Particle Accelerator Conference, Chicago

A LEX-BASED MAD PARSER AND ITS APPLICATIONS*

O. Krivosheev[†], E. McCrory, L. Michelotti, D. Mokhov[‡], N. Mokhov, J.-F. Ostiguy
FNAL, Batavia, IL 60510, USA

[‡] University of Illinois at Urbana-Champaign, USA

Abstract

An embeddable and portable Lex-based MAD language parser has been developed. The parser consists of a front-end which reads a MAD file and keeps beam elements, beam line data and algebraic expressions in tree-like structures, and a back-end, which processes the front-end data to generate an input file or data structures compatible with user applications. Three working programs are described, namely, a MAD to C++ converter, a dynamic C++ object factory and a MAD-MARS beam line builder. Design and implementation issues are discussed.

1 INTRODUCTION

The MAD[1] lattice description language has become the *lingua franca* of computational accelerator physics. In order to achieve acceptance, new codes and libraries need to recognize lattice descriptions expressed in MAD format. Our objective was [2] to produce an embeddable parser able to read, parse and store lattice descriptions in memory. The parser had to be flexible enough to support various formats;

MAD variables, and thus beam element definitions, can be altered at any point, the only sensible way to build a parser is to make it a two-stage program. The first stage, or front-end, reads the MAD input file and parses it in memory. The second stage or back-end, generates output in a suitable format, e.g. C++. This design is very flexible since the back-end can be modified to support other formats or to dynamically instantiate data structures (C++ factory).

2.2 Front-End

The front-end uses a lexical analyzer built with Lex (in its Flex[5] incarnation). It recognizes MAD keywords, identifiers, numbers, strings, and comments from regular-expression-based rules and returns corresponding tokens and semantic values. The parser, written in YACC (we are using the Bison[6] flavor of YACC), contains the grammar for MAD definitions. It recognizes those definitions

Mxyzptlk and Beamline

(L. Michelotti)

- mxyzptlk is a C++ class library to perform automatic differentiation to a user-specified order n . It provides overloaded operators, trig functions etc ...
- Beamline is a C++ class library build on top of mxyzptlk. It provides facilities to create beamlines hierarchies, compute lattice functions, chromaticities, maps (to order n), map concatenations etc ...

Mxyzptlk ?



Superman's foe from the 5th dimension.
He will return to his own dimension
if he spells his name backwards ...

Why Use the BEAMLINe Class ?

- Written in C++, just like ORBIT
- The same code automatically supports 1st , 2nd and or order n maps if desired
- Support for arbitrary misalignments (tilt, yaw, offset etc ..)
- Very few assumptions This may be relevant for smaller rings.
- propagation physics and computation completely under user control if desired/necessary (e.g. thin kicks a la Tpot)
- BTW: Synergia uses BEAMLINe to compute map coefficients.

Propagation is handled by IMPACT

Beamline Class Library

Computational Performance

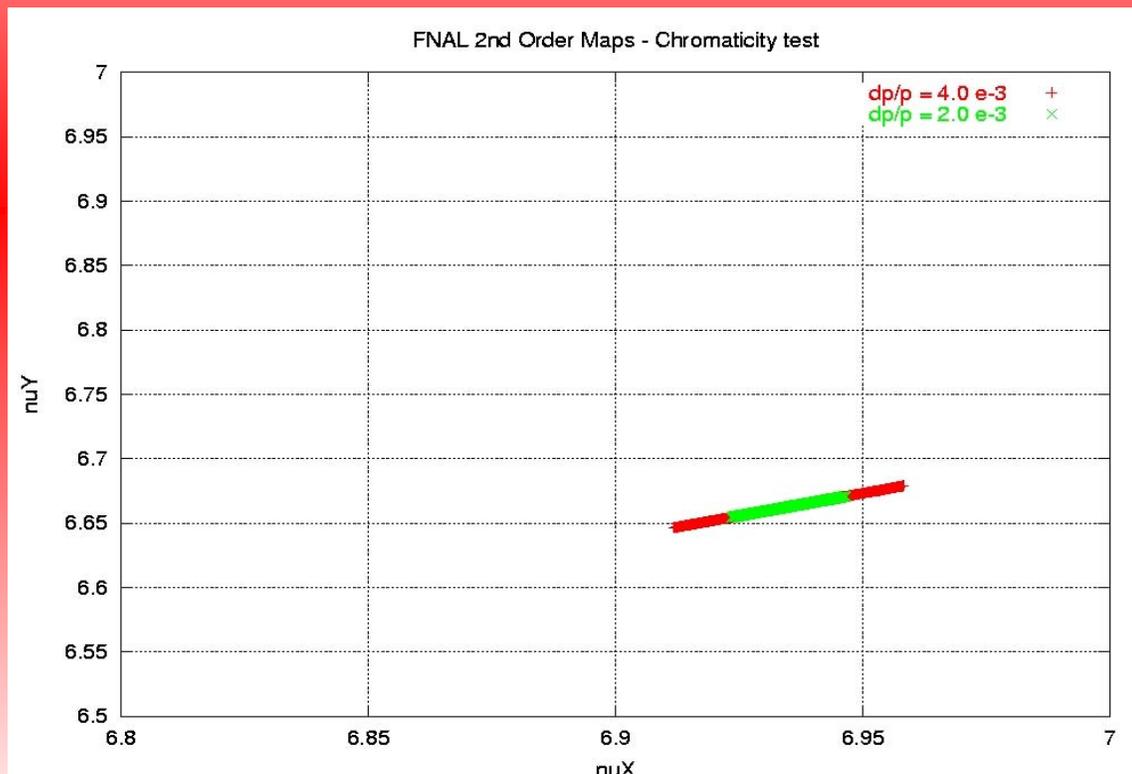
- Initial attempts at using the BEAMLIN library resulted in discouraging performance degradation (1-2 orders of magnitude !)
- In MXYZPTLK, polynomials are implemented as doubly linked lists. Each list node contains a non-zero monomial coefficient as well as an integer which can be uniquely mapped to actual monomial exponents.
- evaluating a polynomial (map) implies traversing a linked list (indirections) and recovery of the actual monomial exponents
- Especially at low orders, it was found necessary to store map coefficients and individual monomial exponents explicitly in linear arrays in order to get satisfactory map evaluation performance.

Propagator

- In the beamline library, the Propagator is a **functor** (i.e. a function object) that determines how a particle is propagated through a beamline element.
- Each element is assigned a default propagator, which can be overridden by a user-supplied alternate
- When the element is a **Map**, the propagator operator() trivially evaluates a polynomial in $6N$ variables for each phase space dimension .
- Because the Propagator is an object, an alternate (private) polynomial representation can be instantiated by the propagator constructor.
- Using this technique, tracking using the facilities provided by the beamline library has been verified to be as efficient as with existing code.

2nd Order Map Code Validation

Simple test: observe the tune spread associated with 2 different momentum distributions.



Independently calculated
tunes:

6.935 H, 6.662 V

Independently calculated
chromaticities:

-9.86H, -6.89V

ORBIT Code Structure

- ORBIT is structured as “modules” controlled by a high level interpreter, SuperCode
- SuperCode was designed to have a C++ -like syntax
- interface code generated from special interface definition files by a program : MGen
- Modules could be written in f77, C or C++
- Exported interface from modules is the common denominator between all these languages: static functions and variables
- BTW: In ORBIT, the shell is an integral part of the code. Input syntax checks and runtime diagnostics are generated by the shell.

Why Preserve the Interpreter/Modules Structure ?

- A high level interpreter allows for rapid implementation of custom features. While the efficiency of a low-level language is required to propagate large no of particles, diagnostics and a posteriori analysis can benefit from high level language implementations because they are often problem-specific.
- If efficiency becomes an issue, functionality implemented at the interpreter level can be reimplemented into a compiled module without affecting existing scripts.
- The interpreter/module structure promotes well-defined interfaces. This makes it easy to contribute new functionality without deep knowledge of the entire code.

Python Shell

Problem: SuperCode is orphaned and poorly documented

Solution: use Python !

- Python is a mature scripting language
- Its object model is highly compatible with that of C++
- It supports operator overloading
- It supports the concept of exceptions
- Good tools are available for interface code generation
- A wealth of publicly available high quality python code is available for reuse

Python/C++ Interface Code Generation

Currently are three systems available:

- SWIG (www.swig.org) by David Beazley, U of Chicago. Comprehensive system, support for most scripting languages. While support for C is excellent, support for C++ constructs has serious limitations.
- SIP (www.riverbankcomputing.co.uk) by Phil Thompson. Similar in philosophy to SWIG, but python/C++ specific. Not well documented, requires special interface files
- Boost.python (www.boost.org) by David Abrahams. Python/C++ specific. Implemented as a C++ library (mostly header files). Uses template metaprogramming techniques to generate interface code; no special program needed beyond a C++ compiler.

Porting Strategy

- Emulate existing SuperCode data types (e.g. Vector, matrix, 3D array)
- As much as possible, emulate existing syntax.

Boost.python Example 1

```
#include <iostream>
#include <string>
#include "sc-types.h"
#include "sc-string.h"
#include "Python.h"
#include <boost/python/operators.hpp>
#include <boost/python/class.hpp>
#include <boost/python/module.hpp>
#include <boost/python/handle.hpp>
#include <boost/python/extract.hpp>
using std::complex;
using namespace boost::python;

void wrap_supercode() {
//  ** ComplexMatrix **

    python::class_<Matrix<complex<Real> > >("ComplexMatrix", init<int,int>())
        .def(init<const Matrix<complex<Real> > >())
        .def("get",      &Matrix<complex<Real> >::get)
        .def("set",      &Matrix<complex<Real> >::set)
        .def("__repr__", &Matrix<complex<Real> >::print)
        .def("clear",    &Matrix<complex<Real> >::clear)
        .def("resize",   &Matrix<complex<Real> >::resize)
        .def(python::self + python::self)
        .def(python::self - python::self)
        .def(python::self * python::self)
        .def(python::self ^ python::self)
        ;
}
```

Boost.python Example 2

```
void wrap_bump()
{

    class_<Bump>("Bump", no_init)
        .def("addIdealBump",      &addIdealBump_wrap, "Add a Bump Node")
        .staticmethod("addIdealBump")
        .def("eFoldBump",        &Bump::eFoldBump,   "Routine used to apply an e-fold bump scheme
with specified start and end points")
        .staticmethod("eFoldBump")
        .def("eFoldBump2",       &Bump::eFoldBump2,  "Routine used to apply an e-fold bump scheme
with specified fixed and ramped magnitudes")
        .staticmethod("eFoldBump2")
        .def("interpolateBumps", &Bump::interpolateBumps, "Routine to do linear interpolation of the
bumps from input vectors")
        .staticmethod("interpolateBumps")
        .def("sizeBumpPoints",   &Bump::sizeBumpPoints, "Sizes vectors used in iterpolating bump
points, see routine interpolateBumps")
        .staticmethod("sizeBumpPoints")
        .def_readwrite("xIdealBump", &Bump::xIdealBump )
// - "The x value of the ideal bump at a point in time (mm)",
        .def_readwrite("xPIdealBump", &Bump::xPIdealBump )
// - "The x prime of the ideal bump at a point in time (mrad)",
        .def_readwrite("yIdealBump", &Bump::yIdealBump )
// - "The y value of the ideal bump at a point in time (mm)",
        .def_readwrite("yPIdealBump", &Bump::yPIdealBump )
// - "The y prime of the ideal bump at a point in time (mrad)";
        .def_readwrite("bumpOn",      &Bump::bumpOn )
// - "Switch indicating whether ideal bump is on (==1)";

    ...
}
```

ORBIT vs PyORBIT

myfile.s

C

```
runName = "Booster";
of1      = runName + ".out";

RealArray hist(1000);
RealArray hist2(1000);

for i=1,1000 {
  hist(i) = 0;
}

cout << "Make a synchronous particle" <<
"\n";

TSync = 0.400; // Kinetic Energy (GeV)
mSync = 1;     // Mass (AMU)
charge = 1;    //charge number

addSyncPart(mSync, charge, TSync);
mainHerd = addMacroHerd(100);

FNALMapLine(file1, file2);

hist2 = hist2 + hist;
```

myfile.py

```
/bin/env python
import orbit

runName = orbit.String("Booster")
of1      = orbit.String(runName) +
orbit.String(".out")

hist = RealArray(1000)

for i in range(1000):
  hist.set(i+1,0.0)

print 'Make a synchronous particle'

TSync = 0.400; # Kinetic Energy (GeV)
mSync = 1;    # Mass (AMU)
charge = 1;   # Charge number

orbit.Particles.addSyncPart(mSync, charge, TSync)
orbit.Particles.mainHerd =
orbit.Particles.addMacroHerd(100)

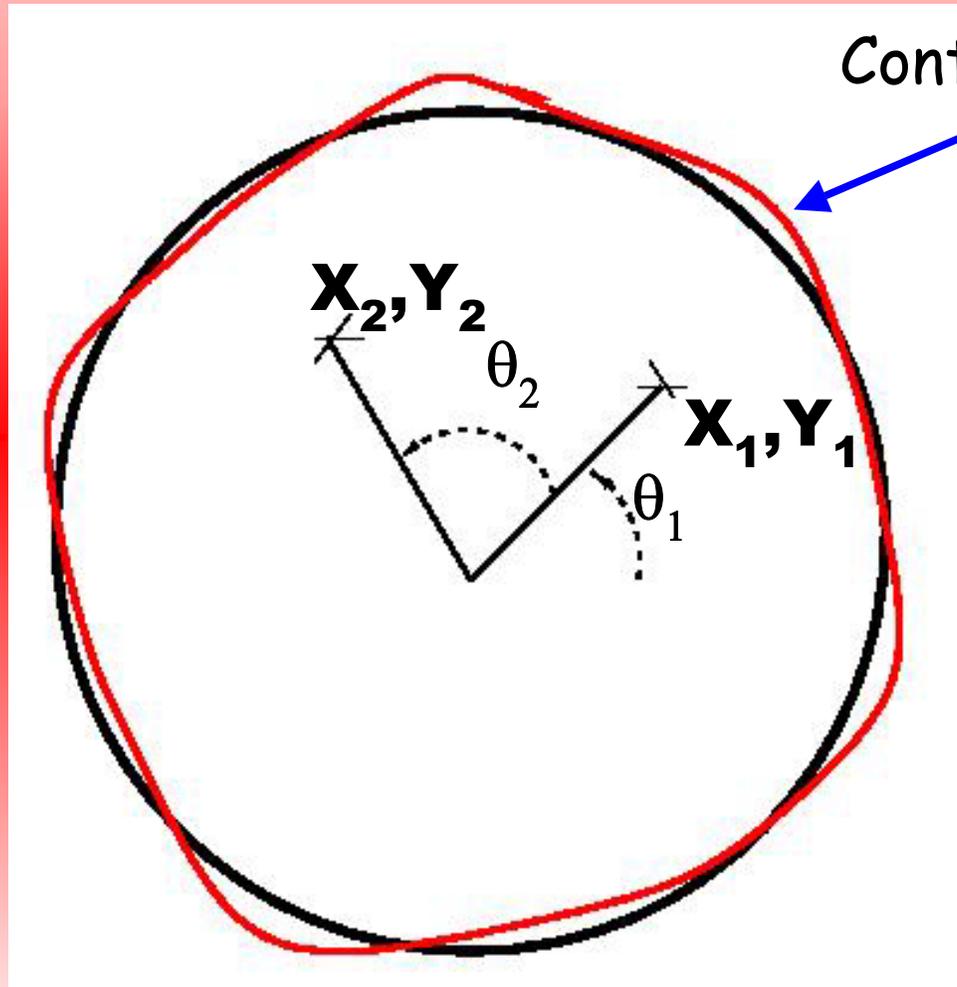
orbit.TransMap.FNALMapLine(file1, file2);

hist2 = hist2 + hist;
```

Tune Computation

- To efficiently produce a (transverse) tune footprint, ORBIT computes tunes by accumulating phase advance in normalized (Floquet) coordinates.
- This is an **approximation** since constant action contours are circular only in the linear approximation.
- The normalized coordinates do not include dispersion. To compute the tune of an off-momentum particle, the dispersive contribution to the trajectory must be subtracted off.
- All collective fields are “frozen” during a tune computation i.e. they influence the macro particles, but they are **not** updated.
- All RF cavities must effectively be turned OFF during a tune computation, otherwise the momentum upstream and downstream of a cavity is different, leading to erroneous dispersive corrections and erroneous tunes. **This was not automatically enforced in ORBIT.** The effect is obviously more noticeable the larger dE/E per turn is.

Phase Accumulation



Contour distorted by non-linearities

Weakly nonlinear map

$$(X_2, Y_2) = M (X_1, Y_1)$$

In presence of nonlinearities, averaging over multiple turns gives better results.

Acceleration

ORBIT currently provides some support for acceleration. However, because SNS is an accumulation ring, investing time and efforts to further develop the existing functionality has not been the highest priority of the main developers (justifiably so!).

The code has relied on externally computed maps, which by definition, do not change during acceleration. While this may be an acceptable approximation, a more realistic simulation should include:

Saturation effects (energy dependent field defects)

Remanent field effects (constant field defects)

Transition crossing (phase jump, pulsed quadrupoles etc ...)

Conclusions and Status

- FNAL MAD parser integrated into ORBIT
- Beamline maps tested
- 1st and 2nd order propagation optimized. Execution speed is on par with the previous implementation.
- Python shell work complete and ready for testing.
- Fixed: tune footprint computation in presence of RF voltage.
- Fixed: incorrect scaling of longitudinal coordinates.
- Fixed: obscure memory management problems in mxyzpltk
- Work on improved support for acceleration has just begun