THE PYTHON SHELL FOR THE ORBIT CODE *

A. Shishlo, J. Holmes, T. Gorlov, ORNL, Oak Ridge, TN 37831, U.S.A.

Abstract

A development of a Python driver shell for the ORBIT simulation code is presented. The original ORBIT code uses the SuperCode shell to organize accelerator-related simulations. It is outdated, unsupported, and it is an obstacle to future code development. The necessity and consequences of replacing the old shell language are discussed. A set of core modules and extensions that are currently in PyORBIT are presented. They include particle containers, parsers for MAD and SAD lattice files, a Python wrapper for MPI libraries, space charge calculators, TEAPOT trackers, and a laser stripping extension module.

INTRODUCTION

The original ORBIT code has been very useful in the SNS ring design and in simulations of collective effects [1]. Thanks to a flexible structure, ORBIT can be extended very easily. After years of development by many scientists, ORBIT includes collimation, different types of space charge, impedances, electron-cloud effects, and numerous other features. These features are combined together by using a driving shell – the SuperCode (SC). SC is an interpreter programming language resembling C. At the time when ORBIT development started (1997), there were not many choices of driving shell language. SC was attractive because it is C-like, it is simple to learn, to understand, and to extend, and it has a set of effective auxiliary classes for arrays, vectors, strings, etc. As a result of deep integration, the ORBIT code has become inseparable from SuperCode, and SC has now become an obstacle to further ORBIT development.

There are several problems related to SC. First, SC is not an object-oriented language. This significantly slows down ORBIT development and limits the functionality of the code. All contemporary interpreters are object-oriented. Second, SC is not supported by anyone. Usually languages are surrounded by a community of users and developers, which facilitates an immediate response to problems and bugs. So, for SC the user is on his own. Finally, all auxiliary classes provided by SC have been implemented in the C++ Standard Template Library, and this implementation is probably more efficient. In SC none of these classes is protected by namespaces, and they could crush the ORBIT code compilation if there is a name conflict.

In an attempt to preserve the legacy of the ORBIT code, the PyORBIT project has been started. The motivation of PyORBIT is to replace the SuperCode driver shell by a modern interpreter language, Python [2]. Unfortunately, it is not possible to directly import the code of core ORBIT modules into the new project because of ubiquitous SC dependencies. On the other hand, this gives us an opportunity to start from scratch in the architecture and the source code development and to keep all original ORBIT physical algorithms.

DRIVER SHELL PARADIGM

PyORBIT, like the original ORBIT code, uses a driver shell language approach instead of an input file analysis, as in traditional accelerator codes like MAD, MAD-X, PTC, PARMILA, Trace3D etc. These traditional codes construct an accelerator lattice and perform calculations according to information inside specialized input files. They each use their own language created for the particular code, and the list of possible tasks is predefined and limited. PyORBIT uses another approach. We use an existing programming language and extend it with specific accelerator-related functionalities. The user can create a unique simulation code in the form of a main program or script by using a predefined set of classes and methods.

There are several requisites for a programming language that can be used for this scheme:

- The program language should be popular among physicists. There are many languages that fall under this category: FORTRAN, C, C++, Ruby, Python, and Java.
- It should be an object-oriented language with an automated garbage collection. This condition eliminates FORTRAN, C, and C++.
- It should be fast. That will eliminate Ruby and Python, which are interpreted languages.
- It should be capable of an effective usage of the Message Passing Interface (MPI) library for parallel calculations. That will remove our last candidate – Java. There are several available Java wrappers for MPI, but the overhead for array exchange makes these packages unacceptable for us.

These constraints necessitate a two-language scheme. To provide the necessary speed we must use FORTRAN, C, or C++ at the low level, and Ruby or Python to organize the calculation at the upper level. For the PyORBIT project we chose C++ for its object-oriented nature, better standardization, and better free compiler availability than FORTRAN. For the upper level we preferred Python, because its pseudo-code compilation feature makes it significantly faster then Ruby. This combination of a scripting language for orchestrating simulations and a fast compilation language to perform calculations is very popular in scientific computing [3]. Generally, code development in a scripting language is considered 3-5-10 times faster than it is in languages like C++ or Java. The downside of the two-level approach is the necessity of a “glue” code to connect the codes in the two languages.

* ORNL/SNS is managed by UT-Battelle, LLC, for the U.S. Department of Energy under contract DE-AC05-00OR22725

PYORBIT CODE STRUCTURE

The directory structure of the PyORBIT code is shown in Fig. 1. PyORBIT consists of three main parts: a core, extensions, and pure Python classes.

The core includes C++ classes and wrappers for them. The wrappers define the Python user interfaces for underlying C++ classes. After compilation of the core source code and linking with the Python language static library and MPI libraries, the PyORBIT executable is placed in the "bin" directory (see Fig. 1). This executable is an extended Python language interpreter that has all the functionality of Python and that can dynamically operate with classes and methods from the core and extensions.

The extensions are independent packages that have no common classes. If two or more extensions use the same class, it should be moved to the core of PyORBIT. Each extension package is dedicated to some particular physical phenomena. At this moment PyORBIT has only one extension, a package that simulates different aspects of the laser-assisted stripping of H^- ions [4]. The extensions are compiled into shared libraries and are placed into the "lib" directory. The libraries are dynamically loaded when invoked in the user's Python script.

The pure Python classes' directory in Fig. 1 has two subdirectories: one for core and one for extensions.

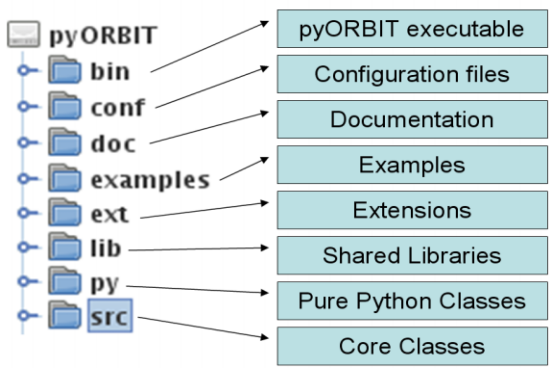

Figure 1: The directory tree of the PyORBIT code.

Today the C++ core of PyORBIT includes four components important for the future development: the MPI wrapper, the PyORBIT Bunch class, the TEAPOT elements library, and the 2D space charge package. The pure Python core components contain the accelerator lattice model, the TEAPOT-like implementation of the accelerator lattice, and the parsers for input files of MAD and SAD accelerator codes. Below we discuss these components.

PYTHON AND C++ CLASSES

To connect the Python and C++ levels, we want flexibility and full control and of our logic flow. Therefore, PyORBIT does not use an automated approach that extends the Python language with C++ classes. To create a wrapper class for a C++ class we follow the standard method described in the "Defining New Types" part of the Python documentation. Each wrapper class inherits from a PyORBIT_Object class that extends the standard PyObject with one void pointer to the wrapped C++ class instance. In turn each C++ class inherits from the CppPyWrapper class that keeps a reference to the Python wrapping object. This cross-reference scheme allows access to Python and C++ objects from any level, and it is used everywhere in PyORBIT except for the MPI library wrapper, because MPI is a collection of functions, not classes.

PYORBIT MPI WRAPPER

From the beginning, PyORBIT was developed as a parallel code based on MPI. At the same time all parallel features can be switched off if the user wants to use only one CPU. To provide this functionality, PyORBIT has the MPI wrapper, which isolates the standard MPI functions from the rest of PyORBIT. It accomplishes this by wrapping the MPI functions into functions with different names, but the same signature, and exposes these wrappers to the Python level. At this moment 45 MPI functions are available from the Python script level. In addition to the MPI functions, the MPI wrapper also transforms the MPI communicators, groups, operations, and the MPI status to PyObjects that can be accessed from the Python and C++ levels. The ability to move MPI objects between Python and C++ was the main reason to create our own MPI wrapper instead of using one of the available open sources. It is expected that MPI on the Python level will be used only to perform small data exchange and to create necessary MPI communicators which later will be used on the C++ level for fast and massive data exchanges.

The PyORBIT MPI wrapper package is completely independent from the rest of the PyORBIT code and can be extracted and used anywhere.

BUNCH CLASS

PyORBIT is a particle tracking code, so a class representing a container for particles is the most important class of the code. The Bunch class of the PyORBIT core is this container class. By default it keeps 6D coordinates and one flag specifying an "alive/dead" status for each particle, and it has the flowing features:

- It is dynamic. The user can add or remove particles from this container. Its size will adjust to the number of particles.
- It is efficient. It provides fast access to the coordinates and it maintains spare room to accommodate additional particles without frequent memory resizing.
- It is extendable. The user can dynamically assign additional properties to each particle in the Bunch. This allows the Bunch class to be used for different kinds of physical problems. For instance, this additional information could be a macro-size of the particle, its spin, or amplitudes of different quantum states, as for the hydrogen atom in the Laser Stripping PyORBIT extension [4]. The possibilities

are numerous. The absence of this kind of extendibility is a big drawback in the original ORBIT code.
- It can be dumped and restored from a file.
- It has parallel capabilities. It automatically distributes particles among CPUs in its local communicator when it restores a bunch from a file.
- All methods of the C++ implementation are exposed to the Python level.

The additional information that can be attached to each particle in the bunch should be stored as a double array of the predefined length. This condition limits the user freedom, but it provides a fast way to exchange particle information between CPUs in parallel calculations. Still, this approach is general enough to be acceptable for all physical phenomena that we have in mind right now.

As said before, the Bunch class has 6 phase-space coordinates for each macro-particle. The Bunch class does not define the meaning of these coordinates, and it is up to the user to define them. In the TEAPOT-like tracking we follow the original ORBIT. We consider them as transverse displacements and angles for the transverse plane, and as position and energy deviation from the design energy for the longitudinal direction. However, in PyORBIT we change transverse units to meters and radians from millimetres and milliradians in ORBIT. The longitudinal position is also given in meters instead of radians.

ACCELERATOR LATTICE PACKAGE

The accelerator lattice package is a lightweight pure Python implementation of a structure shown in Fig. 2. The package includes three classes: the Accelerator Lattice class, the Accelerator Node class, and the Action Container class.

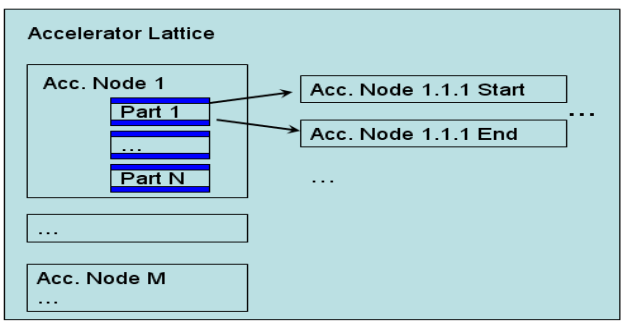

Figure 2: The PyORBIT accelerator lattice structure.

The Accelerator Lattice class is a container of the instances of the Accelerator Node class (nodes). The lattice class has methods to get the length of the lattice, to add a new accelerator node at any place in the lattice, to create a sub-lattice from the existing one, and to call the "trackAction" method for each accelerator node. This method accepts two objects: the instance of Action Containers and a dictionary with user parameters. The lattice puts into the parameters dictionary two references, one to itself and one to the accelerator node.

The Accelerator Node represents a single node in the lattice and is built according to E. Forest's concept of "fibre bundle" [5]. Each part of the node is a container for references to child Accelerator Nodes. When the lattice calls for the "trackAction" method of the node, this method is performed recursively for each child node and executes actions that are in the Action Container. The user should consider the Accelerator Node class as an abstract class for subclasses that will perform meaningful actions.

The Action Container class is a keeper of user-defined functions (actions) that will be called upon entering the node, at each part of the node, and at the exit of the node. By default this container is empty, and it is up to the user to supply the calculations and their order inside the container. The Python mechanism of lexical closures allows one to define such actions in the source code of the class methods.

The accelerator lattice package is a very flexible construction that can accommodate almost any kind of functionality, but there is no restriction in PyORBIT to prevent other approaches to defining a model for the accelerator lattice.

TEAPOT-LIKE ACCELERATOR LATTICE

There are two PyORBIT components that enable TEAPOT-like tracking of the macro-particles. The first component is the collection of C++ functions that were developed for the original ORBIT to track 6D coordinates of charged macro-particles through simple accelerator elements including dipoles, drifts, quads, multipoles, solenoids, kickers, etc. These functions were thoroughly benchmarked against analytical models, and their source code was directly imported into PyORBIT. The second part is a collection of pure Python classes that are subclasses of the Accelerator Node class. These classes keep parameters of the nodes and call the C++ TEAPOT tracking functions.

The TEAPOT lattice can be built right in the script by adding accelerator nodes one by one or by analyzing a MAD input file. PyORBIT includes a MAD parser that can read a MAD input file specifying the structure of the accelerator. The parser is discussed below.

At present, the TEAPOT-like lattice does not have such nodes as a foil injection node, collimators, space charge nodes, diagnostics nodes, etc. that are in the original ORBIT code. We plan to import them into PyORBIT in the near future.

MAD-FILE PARSER

The original ORBIT does not have any internal tools to use MAD input files directly. The user has to run MAD to get MAD output files with transfer matrices and Twiss parameters that will be used by ORBIT. This dependency is one of the weak points of the ORBIT code. For PyORBIT the pure Python MAD parser is developed. It is completely independent from the rest of the code, and it is used as a generator for the TEAPOT-like input file for PyORBIT. The parser can perform mathematical calculations defined in the MAD file and can handle

insertions of external files. PyORBIT also has a modification of this parser for the SAD code input files. The SAD code is the accelerator design and simulation code developed by KEK accelerator theoretical group [8].

RUNGE-KUTTA TRACKER

In addition to the TEAPOT-like tracking, PyORBIT has a package to solve the equation of motion with arbitrary electric and magnetic fields

$$d\vec{p}/dt = q\cdot(\vec{E}+\vec{v}\times\vec{B}) \quad (1)$$

The package uses the Runge Kutta 4-th order (RK4) solver. The user must specify both electric and magnetic fields as functions of position and time. For prototyping, this can be done on the Python level, but the speed of calculations will be very slow. The user also can attach a custom implementation of the External Effects class. The user has to define the "applyEffects" method of this class which will be called at each time step of the RK4 solver. This allows the user to specify other things that can happen to the macro-particles during their motion through the electromagnetic field region. For instance, there may be decay, ionization, excitation, or interaction with a collimator material. In the laser-stripping package [4] this tracker is used to simulate the dynamics of the internal states of hydrogen atoms in the laser field.

2D SPACE CHARGE SOLVER

As a base for future space charge modules PyORBIT has a FFT-based Poisson solver package. The package includes three classes: Grid2D, PoissonSolver, and Boundary classes. The Grid2D class represents the two dimensional rectangular grid with a space charge density or the electrostatic potential. The PoissonSolver class calculates the potential on the grid

$$\phi_0(\vec{r}) = \int \frac{\rho(\vec{r}-\vec{r}\,')\cdot d\vec{r}\,'}{\left|\vec{r}-\vec{r}\,'\right|^2} \quad (2)$$

by using the Fourier convolution theorem and discrete transformation (FFT) [9].

The Boundary class is a container of arbitrary boundary points inside the defined grid. It modifies the potential on the grid by adding the potential in empty space [10]

$$\phi_{empty}(r,\theta) = \sum_{n=0}^{N} r^n\left[a_n\cos(n\theta)+b_n\sin(n\theta)\right] \quad (3)$$

where r and θ are the usual polar coordinates, N is a user defined maximum number of harmonics. The coefficients a_n and b_n are found by minimizing the sum of potentials (2) and (3) at the boundary points in a least squares sense. The sum of two potentials is the solution of Poisson's equation with zero potential on boundary points. The number of boundary points and the number of harmonics determine the accuracy of the solution.

Again, this package is relatively independent from the rest of the PyORBIT code and can be used separately.

CONCLUSIONS

At present, PyORBIT does not have the full collection of physics modules of the original ORBIT code, but it has all the basic components to accommodate these modules. The new capabilities of PyORBIT include the customizable Bunch container, which provides the means to simulate a broader spectrum of physical problems. A fine example of these extended capabilities is the Laser Assisted Stripping module developed inside PyORBIT [4].

REFERENCES

[1] A. Shishlo, S. Cousineau, V. Danilov, J. Galambos, S. Henderson, J. Holmes, M. Plum, "The ORBIT Simulation Code: Benchmarking and Applications", ICAP 2006, Chamonix Mont-Blanc, France, 2-6 Oct 2006, pp. 53-58
[2] http://python.org
[3] http://www.scipy.org
[4] T. Gorlov, A. Shishlo, "Laser stripping computing with the Python ORBIT code." These proceedings.
[5] http://docs.python.org/extending
[6] E. Forest et al., "Polymorphic Tracking Code PTC," KEK Report 2002-3
[7] http://en.wikipedia.org/wiki/Closure_(computer_science)
[8] H. Hirata, in Proceedings of the Second Advanced ICFA Beam Dynamics Workshop, Lugano, Switzerland, 1988 (CERN, Geneva, 1988), p. 62.
[9] R. W. Hockney and J. W. Eastwood, Computer Simulation Using Particles, Institute of Physics Publishing (Bristol: 1988).
[10] F. W. Jones, A Method for incorporating image forces in multiparticle tracking with space chargein, Proceedings of EPAC2000, (Vienna, 2000) 1381.