Procedia Computer Science

Volume 51, 2015, Pages 1272–1281

ICCS 2015 International Conference On Computational Science

The Particle Accelerator Simulation Code PyORBIT

Andrei Shishlo, Sarah Cousineau, Jeffrey Holmes, and Timofey Gorlov
Oak Ridge National Laboratory, Oak Ridge, TN 37831, USA
shishlo@ornl.gov, scousine@ornl.gov, holmesja1@ornl.gov, gorlovtv@ornl.gov

Abstract
The particle accelerator simulation code PyORBIT is presented. The structure, implementation, history, parallel and simulation capabilities, and future development of the code are discussed. The PyORBIT code is a new implementation and extension of algorithms of the original ORBIT code that was developed for the Spallation Neutron Source accelerator at the Oak Ridge National Laboratory. The PyORBIT code has a two level structure. The upper level uses the Python programming language to control the flow of intensive calculations performed by the lower level code implemented in the C++ language. The parallel capabilities are based on MPI communications. The PyORBIT is an open source code accessible to the public through the Google Open Source Projects Hosting Service.

Keywords: Open Source, Python, C++, MPI, Accelerator Simulation, Particles-in-Cell, Space Charge, PyORBIT

1 Introduction

The PyORBIT code is a "Particle-In-Cell" (PIC) simulation tool for cyclical and linear particle accelerators. It simulates the motion of a beam of charged hadrons or ions through the drift spaces and electromagnetic fields of the elements of the particle accelerator lattice. The particles are contained in bunches and can interact with each other via electric self-fields (space charge) or through the electromagnetic fields induced in the accelerator elements (impedances). In accumulator rings and synchrotrons the positively charged beams can also interact with electron clouds created by ambient gas ionization or emission from the beam pipe walls. These collective effects can create instabilities in the beam and beam loss. The simulations are used to predict and to prevent such losses. The number of particles in the real bunch is usually too high to be simulate directly and thus the bunch is represented by a collection of macro-particles moving like a single particle in the electromagnetic fields but carrying a combined charge that contributes to the collective effects. The number of macro-particles required for simulation depends on the types of physical effects and the algorithms included in the simulation. This number can vary from several thousand to hundreds of millions of macro-particles. For big and computationally intensive simulations PyORBIT uses MPI-based parallelization.

 Selection and peer-review under responsibility of the Scientific Programme Committee of ICCS 2015

doi:10.1016/j.procs.2015.05.312

PyORBIT is a modern implementation of the original ORBIT code that was developed at Oak Ridge National Laboratory to improve the Spallation Neutron Source accelerator ring design and to study proton beam dynamics in the presence of collective effects [1]. The ORBIT code's flexible structure allowed the code to be easily extended. After years of development by many scientists, ORBIT included collimation, different types of space charge, impedances, electron-cloud effects, and numerous other physical modules. These modules were combined together in simulations by use of a driver shell – SuperCode (SC) [2]. SC is an interpreter programming language with a syntax resembling C. When ORBIT development started in 1997, there were not many choices of driving shell languages. SC was attractive for a number of reasons: It was C-like, simple, easy to understand and to extend, and had a set of effective auxiliary classes for arrays, vectors, strings, etc. As a result of the deep integration of SC and ORBIT, the ORBIT code became inseparable from SuperCode, and this eventually became an obstacle in the further development of ORBIT. There were several problems related to SC. First, SC was not an object-oriented language. This significantly slowed ORBIT development and limited the functionality of the code. All contemporary interpreters are object oriented. Second, SC is not supported by anyone. In most languages, there is a community of users and developers who respond immediately to problems and bugs. This is not the case for SC. Finally, all auxiliary classes provided by SC have since been implemented in the C++ Standard Template Library, and this implementation is probably more efficient. In SC none of these classes are 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 was started. The motivation for PyORBIT was to replace the SuperCode driver shell by a modern interpreter language, Python [3]. Unfortunately, it was not possible to directly import the core ORBIT modules into the new project because of ubiquitous SC dependencies. On the other hand, this gave us an opportunity to design the architecture and the source code development from the ground up while keeping all of the original ORBIT physics algorithms.

2 Two Language Structure

Like the original ORBIT code, PyORBIT uses a driver shell language approach instead of an input file interpreter, as in traditional accelerator codes like MAD, MAD-X, PTC, PARMILA, Trace3D etc. These traditional codes construct an accelerator lattice, activate specific physics modules, and perform calculations according to information inside specialized input files. They each use their own accelerator-physics-related language created for the particular code, and the list of possible tasks is predefined and limited. The extension of such codes is difficult, and must be performed by the code owners. These codes are mostly proprietary, and an ordinary user does not have access to the source code. PyORBIT uses another approach. We use an existing programming language and extend it with specific accelerator-related functionalities. In this case 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. The user always has access to the source code and the ability to modify it if he/she chooses.

There are several requirements for a programming language with 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 automated garbage collection. This condition eliminates FORTRAN, C, and C++;
- It should be fast. This eliminates 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. This 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 calculation speed we must use FORTRAN, C, or C++ at the low level, and Ruby or Python to organize the structure and the calculations at the upper level. For the PyORBIT project we chose C++ for its object-oriented nature, better standardization, and better free compiler availability compared to FORTRAN. For the upper level we preferred Python, because its pseudo-code compilation feature made it significantly faster than Ruby at the time when we made this choice (around 2006). This combination of a scripting language for orchestrating simulations and a fast compilation language to perform calculations is very popular in scientific computing [4]. Generally, code development in a scripting language is considered to be significantly faster than it is in languages like C++ or Java. Also, users can work primarily at the level of the scripting language without directly dealing with the computationally-intensive C++ code.

There are several downsides to this approach. First, the developer must be proficient in two languages instead of one. Second, the debugging process is difficult because debuggers are developed for single-language codes. Third, there are no fully functional Integrated Development Environments (IDEs) for scripting programming languages without strict typing of the variables; this is the downside of the flexible nature of the driving shell. The final disadvantage of the two-level approach is the necessity of a "glue" code to connect the codes in the two languages. Despite these problems the two language approach proves to be more efficient and popular than the traditional one.

3 The PyORBIT Components

Figure 1 shows the directory structure of the PyORBIT code. It includes three main parts: a core, pure Python classes, and extensions.

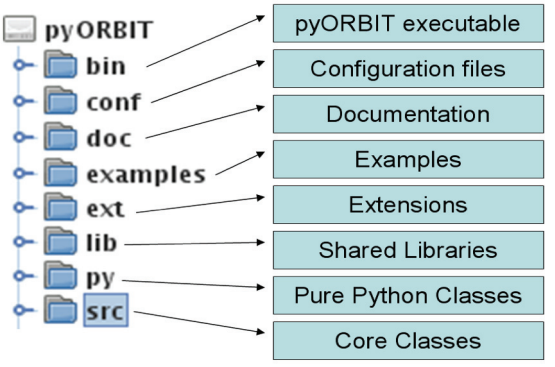

Figure 1: The PyORBIT directory structure.

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

The pure Python classes directory in Figure 1 has two subdirectories (not shown): one named "orbit" for the core, and another named "ext" for extensions. There is also a directory for C++ extensions – the upper level "ext" directory (see Figure 1). The extensions are independent packages that have no common classes. If two or more extensions use the same class, they should be moved to

the core of PyORBIT. Each extension package either contains a model of a particular physical phenomenon, or is a temporary place for a new package under heavy development. The extensions facilitate the development of new modules by independent coders. They do not interfere with other users. At this moment PyORBIT has two extensions: a package that simulates different aspects of the laser-assisted stripping of H- ions [5], and an interface to the PTC package [6]. 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.

At this moment, the most important components of the C++ and Python cores of PyORBIT are: the MPI wrapper; the PyORBIT Bunch class and its diagnostic classes; the abstract accelerator lattice model and its implementation for rings and linear accelerators; the TEAPOT elements library; and the 2D/3D space charge packages. Below we discuss these components.

4 Python Wrappers of C++ Classes

To use C++ classes from the core we have to create a wrapper code for each C++ class. There are several automated generators of wrappers for the Python language, such as Boost.Python, but PyORBIT does not use any of these generators. There are several reasons for not using them. First, we chose to minimize the dependency of PyORBIT on third party libraries and applications. Second, to use these generators you still have to write the glue code. Third, none of them provide full control over the logic flow between the two languages, and we want to preserve the ability to move references to the classes freely between the two levels. To create a wrapper class for a C++ class in PyORBIT we follow the standard method described in the "Defining New Types" section of the Python documentation. There are other native tools, such as ctypes and cython, that are in the standard distribution of Python, and the PyORBIT developer is welcome to use these as appropriate.

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.

This scheme is convenient, but it also creates the possibility of a memory leak or an attempt to destroy an already destroyed object. The developer should pay attention to the details at the level where an object's constructor and destructor are called. Usually the life cycle of the object is controlled on the Python language level where the garbage collector takes care of unused objects.

5 PyORBIT Parallel Environment

From the start, PyORBIT was developed as an MPI-based parallel code, so it can be used on large-scale parallel computers. 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 an 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 at 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 back and forth 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 elsewhere.

6 Macro-Particle Container Class

For a particle tracking code, one of the two most important components is the class that represents a container for the particles. In PyORBIT this is the Bunch class, and it resides in the PyORBIT core. The Bunch class structure as a container class is shown in Figure 2.

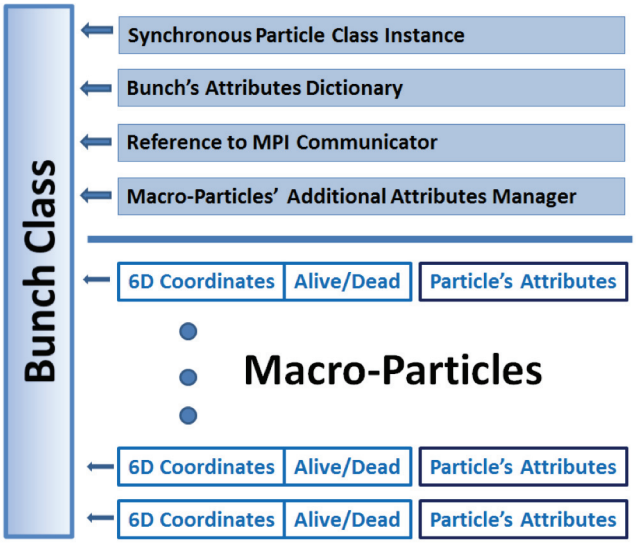

Figure 2: The Bunch class structure.

By default the Bunch instance 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 [5]. The absence of this kind of extendibility was a major drawback in the original ORBIT code;
- It can be dumped to 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 Bunch class has additional attributes that belong to the bunch as a whole: the mass of the elementary particle or ion represented by the bunch, its charge, the classical radius parameter, and the macro-size parameter that defines how many single particles are included into one macro-particle. The macro-size is used only in calculations related to collective effects such as space-charge. In simulations the bunch has an imaginary reference particle that defines the trajectory of the center of the bunch. This particle is called a synchronous particle, and each Bunch instance has an instance of the class representing this particle. Each Bunch instance also has a reference to a MPI communicator that defines the group of CPUs to which this bunch belongs. This communicator is used in the collective effects calculations.

Each macro-particle in the bunch also can have additional attributes. For instance attributes can be used for macro-particle indexing, because the position index inside the bunch can be changed during removal or addition of the particles to the bunch. These attributes are stored as a double array of the predefined length. This 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. Macro-particle particle attributes can be added to or removed from the bunch dynamically from the Python level of the PyORBIT scripts.

As previously stated, 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 left to the user to define them. In the TEAPOT-like tracking classes we follow the original ORBIT conventions. 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 use meters and radians for the transverse units, rather than millimeters and milliradians as used in ORBIT. The longitudinal position is also given in meters instead of radians. This latter choice accommodates transitions from one accelerator or RF length to another.

7 Accelerator Lattice Packages

The other most important component is the abstract accelerator lattice package, a lightweight pure Python implementation of a structure shown in Figure 3. The package includes three classes: the Accelerator Lattice class, the Accelerator Node class, and the Action Container class.

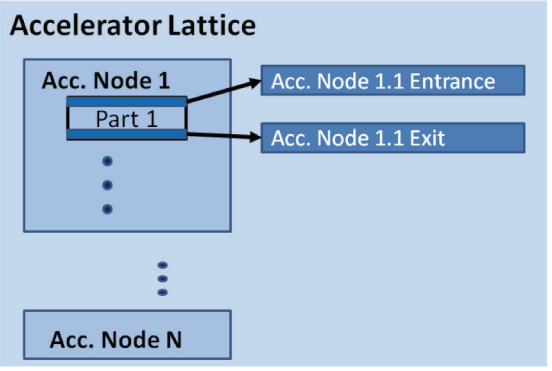

Figure 3: The Accelerator Lattice Structure.

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 Actions Container and a dictionary with user parameters. The lattice puts two references into the parameters dictionary, one to itself and one to each accelerator node.

The Accelerator Node represents a single node in the lattice and is built according to E. Forest's concept of a "fibre bundle" [6]. 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 the keeper of user-defined functions (actions) that will be called upon entering the node, at each part within the node, and at the exit of the node (see Figure 3). 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 users 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. At this moment, there are two implementations of the abstract accelerator lattice: one for particle transport lines and rings, and another for linear accelerators.

7.1 TEAPOT-Like Accelerator Lattice for Rings and Transport Lines

The first implementation of the abstract accelerator lattice package reproduces the original ORBIT code approach. The PyORBIT code includes a collection of C++ functions tracking 6D coordinates of charged macro-particles through simple accelerator elements including dipoles, drifts, quads, multipoles, solenoids, kickers, etc. These functions employ the symplectic integration scheme initially used in the accelerator code TEAPOT, so we call the lattice a "TEAPOT"-like lattice. These functions were thoroughly benchmarked against analytical models, and their source code was directly imported into PyORBIT. They are implemented in C++ language and provide fast tracking. They are wrapped and exposed to the Python level where they are used in a collection of pure Python classes that are subclasses of the Accelerator Node class. These Python classes keep the parameters of the nodes and call the C++ TEAPOT tracking functions.

The TEAPOT lattice for a particular accelerator 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 TEAPOT-like lattice is used for transport lines, synchrotrons, or accumulator rings. In these types of structures the energy of the particles is constant or changes very slowly, typically in only one or at most a few nodes. Therefore, changes of the magnet fields are relatively infrequent. In the early days of the original ORBIT code the accelerator ring was an accumulator ring and the energy of the particles was constant. There is another type of accelerator structure where the energy changes much faster – linear accelerators.

7.2 Accelerator Lattice for Linacs

The linear accelerator lattice package in PyORBIT is another concrete implementation of the abstract accelerator lattice classes described above. The energy of a synchronous particle changes along the linac lattice, and the parameters of the lattice elements must be changed accordingly. In addition to this feature of the linear accelerator lattice, we implemented a more complicated structure that includes subsequences and RF cavities that, in turn, consist of RF gaps. The RF cavities themselves are not lattice elements. They are used to synchronize the phases of all RF gaps that belong to a particular RF cavity. Before using this type of lattice, it must be initialized by tracking a design

particle to map its arrival time at each RF cavity. Only after that, will changes to the cavity amplitudes or phases in the model reflect the changes in the real machine.

At present, a linac lattice can be built in two ways. First, it can be constructed directly in a PyORBIT script by adding lattice elements one by one. Second, it can be built by using an input XML file and linac lattice parser. The parser assumes a certain structure of the input XML file. This structure will be standardized in the future when the list of necessary parameters is agreed upon among all users.

At this moment, all classes in the linac packages are considered experimental, and they are kept in a specific SNS linac directory. All these classes are pure Python classes. They are lightweight and can be easily modified to accommodate different requests. It is also possible to implement other types of abstract accelerator lattices if a more universal approach is required in the future.

There are several new C++ classes that have been created to simulate physics in linear accelerators. They include two types of space charge calculation and a simplified RF gap model. We plan to implement more sophisticated RF models in the near future.

8 Space Charge Calculations in PyORBIT

The charged particles that comprise the bunch and are tracked by PyORBIT interact with each other via the Coulomb force. This collective effect is called “Space Charge”. It can be significant and may cause unwanted beam loss. An accelerator code should include tools to add this effect to simulations. Usually, particle bunches in rings and linear accelerators have different ratios between longitudinal and transverse sizes, and therefore the methods of space charge calculation are different. At present, PyORBIT has two space charge models for linacs and one for rings. In linacs the transverse and longitudinal sizes are comparable, and for the space charge calculations we use two alternative methods: 3D uniformly charged ellipsoid field and a 3D FFT Poisson solver. For rings, where the longitudinal bunch size is much bigger than transverse size, we use a 2.5D space charge model. For all methods the calculations of the electromagnetic fields are performed in the co-moving bunch-centered coordinate system, where only the electric component of the field is important. This electric field is defined by the Poisson equation. Finally the calculated forces acting on each particle are transformed to the laboratory frame where the bunch is moving along the accelerator as a whole.

8.1 3D Electric Field of Uniformly Charged Ellipsoid

The simplest way to calculate space charge forces for the linac bunch is to approximate its charge distribution by a uniformly charged 3D ellipsoid. The electric field inside and outside of such an object can be easily calculated [7], and the space charge force momentum kicks are applied to each macro-particle in the bunch. The parameters of the ellipsoid are found from the condition of equality of rms sizes for the real bunch and the resulting approximation. This scheme is very simple and fast, and it will work even if the linac bunch consists of very few macro-particles (several hundred is an acceptable number). This approach can be considered a variant of the beam envelope calculations in Trace3D or in the XAL [8] online model. If the charge density distribution is more complicated, an arbitrary number of uniformly charged ellipsoids can be used.

8.2 3D FFT Poisson Solver

If the linac bunch has an asymmetric shape, the Poisson equation for the electric potential should be solved by an exact method. For this purpose PyORBIT has a 3D FFT Poisson solver that uses the FFTW library. This solver will work in the case of parallel calculations, but it has bad parallel scalability, because all calculations related to the FFT transformations are identical and are performed

on each CPU. The parallel efficiency of this module is determined only by the distribution of the macro-particles between CPUs. In the future we plan to add more efficient 3D Poisson solvers.

8.3 2.5D Space Charge Solver

For bunches with the length much bigger than the transverse size, PyORBIT uses a set of 2D grids uniformly distributed along the longitudinal axis of the bunch. Each grid has a total charge proportional to the longitudinal density of the bunch at this position. The sum charge of all 2D grids equals to the total charge of the bunch. Each slice represents a 2D Poisson problem whose solution gives us the potential and forces acting on the particles at this longitudinal position. The space charge package for the ring style bunches includes three classes: Grid2D, PoissonSolver, Direct Coulomb class, and Boundary classes. The Grid2D class represents the two dimensional rectangular grid. The Grid2D instances keep a space charge density, the electrostatic potential, or the transverse components of the Coulomb forces. The PoissonSolver class calculates the potential on the grid 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]. 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. The Direct Force class calculates the forces on the 2D grid directly from the space charge density using the FFT. In this case there are no boundary conditions.

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

9 Conclusions

The PyORBIT code is a two level parallel framework for simulation of beam dynamics in hadron accelerators including collective effects such as space charge. It uses the Python language as a driving shell for the computationally extensive calculations performed on the C++ level. The parallel capabilities are based on MPI. It is an open source project, so any user has access to the source code and the option to modify it. PyORBIT can be freely downloaded from the Google project hosting site [11].

10 Acknowledgment

The work was performed at the Spallation Neutron Source at Oak Ridge National Laboratory. ORNL/SNS is managed by UT-Battelle, LLC, for the U.S. Department of Energy under contract DE-AC05-00OR22725.

11 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] Haney, S.W., "Using and Programming the SUPERCODE", UCRL-ID-118982, Oct. 24, 1994.

[3] http://python.org

[4] http://www.scipy.org

[5] T. Gorlov, A. Shishlo, “Laser stripping computing with the Python ORBIT code". Proc. of ICAP ’09, TH3IOPK03 (2009); http://www.JACoW.org

[6] E. Forest et al., “Polymorphic Tracking Code PTC,” KEK Report 2002-3

[7] O. D. Kellogg, Foundations of Potential Theory, (Dover, New York, 1953), 192

[8] http://xaldev.sourceforge.net

[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 charge, in Proceedings of EPAC2000, (Vienna, 2000) 1381

[11] http://code.google.com/p/py-orbit/source/checkout