LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-CONF-464200

PyMercury: Interactive Python for the Mercury Monte Carlo Particle Transport Code

F. N. Iandola, M. J. O'Brien, R. J. Procassini

December 17, 2010

International Conference on Mathematics and Computational Methods Applied to Nuclear Science and Engineering
Rio de Janeiro, Brazil
May 8, 2011 through May 12, 2011

International Conference on Mathematics and Computational Methods Applied to Nuclear Science and Engineering (M&C 2011)
Rio de Janeiro, RJ, Brazil, May 8-12, 2011, on CD-ROM, Latin American Section (LAS) / American Nuclear Society (ANS)
ISBN 978-85-63688-00-2

PYMERCURY: INTERACTIVE PYTHON FOR THE *MERCURY* MONTE CARLO PARTICLE TRANSPORT CODE

Forrest N. Iandola
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801, USA
iandola1@illinois.edu

Matthew O'Brien and Richard Procassini
Lawrence Livermore National Laboratory
Livermore, CA 94551, USA
mobrien@llnl.gov, spike@llnl.gov

ABSTRACT

Monte Carlo particle transport applications are often written in low-level languages (C/C++) for optimal performance on clusters and supercomputers. However, this development approach often sacrifices straightforward usability and testing in the interest of fast application performance. To improve usability, some high-performance computing applications employ mixed-language programming with high-level and low-level languages. In this study, we consider the benefits of incorporating an interactive Python interface into a Monte Carlo application. With PyMercury, a new Python extension to the Mercury general-purpose Monte Carlo particle transport code, we improve application usability without diminishing performance. In two case studies, we illustrate how PyMercury improves usability and simplifies testing and validation in a Monte Carlo application. In short, PyMercury demonstrates the value of interactive Python for Monte Carlo particle transport applications. In the future, we expect interactive Python to play an increasingly significant role in Monte Carlo usage and testing.

Key Words: Monte Carlo, Python, Neutron transport, Parallel programming, Scientific computing, Computer modeling

1. INTRODUCTION AND MOTIVATION

Monte Carlo particle transport applications play a crucial role in studying health physics, nuclear reactor shielding, and United States homeland security [1]. For optimal performance on clusters and supercomputers, Monte Carlo applications have traditionally been implemented in low-level languages such as C, C++, and Fortran [2]. In comparison to these low-level languages, high-level scripting languages such as Python [3] offer simplified syntax for improved usability. However, scripting languages are less efficient than low-level languages such as C, C++, and Fortran for scientific computation. Thus, the decision to implement a Monte Carlo application in a low-level language typically requires sacrificing straightforward usability for fast application performance.

To avoid compromising between usability and performance, mixed-language programming models can combine the simplicity of a scripting language with the fast performance of a low-level, compiled language. A study of mixed-language programming shows that extending C++ code with Python does not add to overhead, so long as the C++ code is retained for performance-critical computation [4]. We illustrate the usability benefits of C++/Python mixed-language programming for Monte Carlo applications with PyMercury, a Python extension to the Mercury [5] [6] general-purpose Monte Carlo particle transport code.

Mercury is written primarily in C++, and it uses Message Passing Interface (MPI) [7] to scale to thousands of processors on scientific computing clusters. PyMercury is an easy-to-use, interactive Python interface for use during Mercury runtime. We demonstrate the usability benefits of PyMercury with a case study in Section 4. In addition, PyMercury demonstrates how Python interfaces can simplify the process of developing and testing Monte Carlo applications. We demonstrate these benefits with a case study in Section 5. While providing these usability and development benefits, Mercury retains its fast C++ based numerical computation code during PyMercury usage. Thus, the interactive PyMercury runtime environment improves runtime usability without contributing additional performance overhead.

2. RELATED WORK

Several high-performance computing applications have utilized mixed-language programming to combine Python with C, C++, or Fortran. These applications have succeeded in improving usability without necessarily degrading fast, scalable application performance. Python interfaces for runtime control have enhanced the usability of inertial confinement fusion [8], neuroscience [9], cosmology [10], and computer vision [11] parallel applications. High-level Python scripts also provide simple syntax for launching parallel applications for bioinformatics [12] and astrophysics [13].

Despite the success of mixed-language programming in some scientific application areas, the Monte Carlo particle transport community has been slow to adopt mixed-language programming. Two notable exceptions are Monte Carlo applications FLUKA and Geant4, both of which use some Python code. The FLUKA [14] Monte Carlo particle transport code is written in C++, Fortran, and Python. FLUKA offers Python-based runtime graphical debugging and data analysis tools [15]. In addition, the Geant4 particle transport code offers a Python extension with interactive runtime support called Geant4Py [16]. Together with Geant4Py, PyMercury leads the movement toward text-based, interactive Python interfaces for general-purpose Monte Carlo particle transport simulations.

3. PYMERCURY DEVELOPMENT APPROACH

In developing PyMercury, we considered a variety of strategies for connecting Python and C++ code. All of these strategies involve the Python/C API [17], which is a platform for extending C or C++ code with Python. With the Python/C API, developers can map any C or C++ data structure or function to any Python object or function.

Tools such as *Boost.Python* [18] and Simple Wrapper Interface Generator (SWIG) [19] facilitate rapid development of Python wrapper code for C++ data structures and functions.

Boost.Python and SWIG parse C++ header files and subsequently create Python wrapper functions for C++ data and functions. Both of these tools create code underpinned by the Python/C API. Nomenclature from C++ header files is reflected in the naming of the Python wrapper functions [20].

The Mercury C++ code offers a broad variety of features in order to support general-purpose Monte Carlo particle transport modeling. We expect data analysis, debugging, and validation to be three of the most common uses for PyMercury. Therefore, to make PyMercury intuitive for users, we have customized the Python nomenclature to ensure that the features relevant to data analysis, debugging, and validation are easy to use. This customization requires flexibility between the function and object nomenclature used in C++ and in Python.

To support an increasingly broad variety of particle transport problems, Mercury is under active development. In the development process, the Mercury C++ data structures and functions are sometimes changed significantly. Thus, if Boost.Python or SWIG were used for creating C/Python API code, the user-facing Python objects and function names would change whenever C++ data structures or function names are modified. To provide a consistent Python user interface, minor changes to C++ nomenclature between Mercury versions should not be reflected in Python function and object naming.

In contrast to Boost.Python and SWIG, directly programming the Python/C API allows for increased flexibility between the C++ code and Python function and data structure nomenclature. To maintain a consistent Python interface as the Mercury C++ code evolves and to provide a customized, intuitive interface, we directly programmed the Python/C API without using SWIG or Boost.Python to develop PyMercury.

4. CASE STUDY I: ENERGY DEPOSITION

In the radiologic sciences, Monte Carlo particle transport software is often used to analyze applications such as radiation therapy. A 2001 study by Yoriyaz et al. used *Monte Carlo N-Particle* (MCNP), a well-known Monte Carlo code, to model energy deposition in a human from a photon beam [21]. With *tallies*, or running totals of particle interactions, the study tracked energy deposition due to photons in the computationally modeled human kidney. For radiation treatment applications, Yoriyaz et al. sought to use MCNP to provide "timely patient-specific dose information to the physician and medical physicist."

While the study succeeded in modeling energy deposition, the study mentioned a difficult and hard-to-learn user interface as a major downside to MCNP usage. Yoriyaz et al. suggest that the interface must be operated "by a person familiar with programming and knowledgeable about the program structure." The study concluded that further MCNP interface development efforts were necessary for physicians and medical physicists to effectively use MCNP. It should be noted that, although the photon energy deposition study was conducted in 2001, the current version

of MCNP also lacks runtime tally access [22].

If the energy deposition study were conducted with Mercury, medical professionals could exploit runtime access to tallies through the PyMercury interactive interface. For example, with the PyMercury code in Figure 1, a medical professional could run Mercury until a user-specified threshold of energy deposition has been reached. To accomplish this, the code in Figure 1 accesses the energy deposition tally object with the PyMercury call `mc.tally.tal["EnergyDeposition"]`. (Note that all PyMercury functions begin with "`mc`" to denote Monte Carlo.) Then, the code uses the tally object's `getValue()` function to request the total energy deposited by neutrons in the kidneys of the computer-simulated human figure. With the interactive Python interface, this code can be injected between iterations of the Mercury particle transport computation. For medical professionals, code like this could simplify the process of analyzing energy deposition in a computationally-modeled human figure.

```
energyTally = mc.tally.tal["EnergyDeposition"]

#Call at each cycle of Mercury execution
if energyTally.getValue(Particle="Neutron", Cell="Kidneys") > 1e-6:
   print "Neutron energy deposition to the kidneys reached threshold."
```

Figure 1: PyMercury code for tracking neutron energy deposition.

In short, this case study demonstrates that Python interactive runtime interfaces can improve the usability of the Monte Carlo code. In addition, since runtime tally access is not computationally intensive, the PyMercury Python interface offers improved usability without contributing to performance overhead.

5. CASE STUDY II: GEOMETRY VALIDATION

Many Monte Carlo particle transport applications use highly-tuned code for geometric calculations [23] [2]. In the interest of fast application performance, optimizing the geometry code is often of high priority in Monte Carlo code development. In addition to being critical to performance, geometry code can also be difficult to debug and to validate. A study of geometric setup tools describes the geometric setup as "typically the most challenging and error prone portion of a Monte Carlo model" [24]. With PyMercury, we demonstrate the applicability of Python interfaces to validating geometry calculations in a Monte Carlo code.

In Mercury, a three-dimensional problem space is represented with combinatorial geometry (CG) spatial volumes, or *cells* [24]. Geometric setups for nuclear reactors, particle accelerators, and human figures can be represented in Mercury as collections of CG cells. For fast parallel performance, CG cell volume calculations are performed with highly optimized analytical calculations or with adaptive mesh refinement (AMR) [25].

When new CG surfaces are implemented in Mercury, a "sanity check" volume calculation can aid in validating the analytic or AMR volume calculations for the new surfaces. To be meaningful, the "sanity check" calculation must be performed independently of the AMR code. Some tools exist for testing geometry setups, such as VisIt [25] and *Form Z to Monte Carlo* (FZ2MC) [24].

These tools can provide visual representations of geometry setups in Monte Carlo simulations. However, testing and debugging with visual tools can be difficult to automate and often requires time-consuming human input. While visual tools can be useful for fixing incessant bugs in Monte Carlo geometry setups, many geometry bugs can also be identified with automated tests implemented with PyMercury. PyMercury offers all the tools necessary for creating simple, automated calculations with which to validate the analytical or AMR volume calculation code.

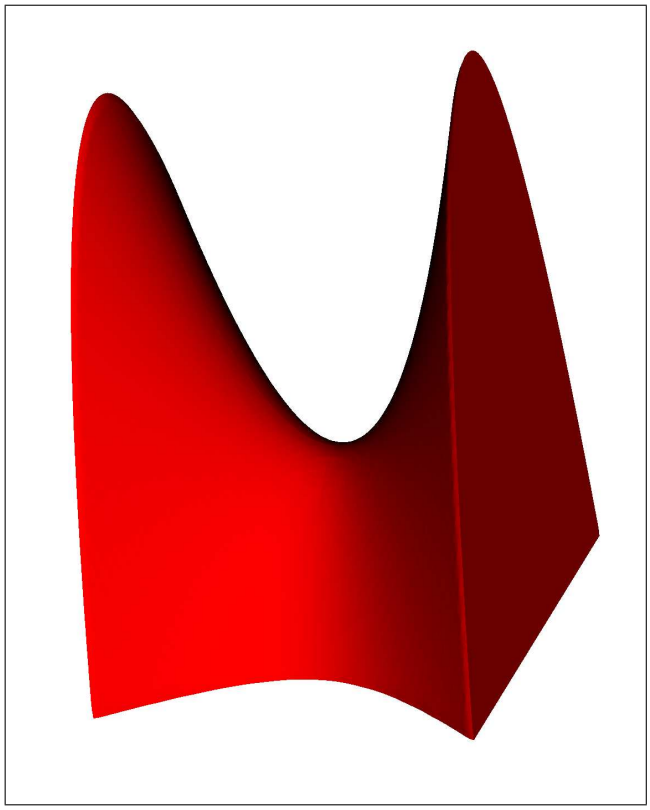

Figure 2: Hyberbolic Paraboloid calculated in Mercury; visualized in VisIt [25].

The Mercury development team has recently expanded Mercury's library of CG cell shapes. Mercury now supports quadric surfaces such as the hyperbolic paraboloid shown in Figure 2. PyMercury is used to validate Mercury's new, highly-optimized volume calculations for quadric surfaces.

Mercury uses an analytical formula to calculate the volume of a hyperbolic paraboloid. The exact volume of the hyperbolic paraboloid that we will study, can be computed from the integral:

$$\begin{aligned} ExactVolume &= \int_{-2}^{2}\int_{-\sqrt{x^2+2}}^{\sqrt{x^2+2}}(x^2 - z^2 + 2)dzdx \\ &= 4 * (3 * \sqrt{6} + ArcSinh(\sqrt{2})) \\ &= 33.97874025 \end{aligned}$$

The analytical calculation finds the exact volume of the cell to be 33.97874025. We use PyMercury to implement two "sanity check" volume calculation methods, with which we verify the correctness of the analytical volume calculation. The "sanity check" calculations each use Monte Carlo volume calculation methods, one based on point sampling and another based on ray tracing.

5.1 Monte Carlo Volume Calculation Method: Point-Sampling

With PyMercury, a simple Monte Carlo volume calculation with point sampling can be implemented in just a few lines of code. To perform this calculation, we begin with a 3D space in Mercury containing a shape, in this case the hyperbolic paraboloid in Figure 2. We then draw a bounding box around the hyperbolic paraboloid and sample random points in the box. An example bounding box for the hyperbolic paraboloid is: $(-2, 0, -\sqrt{6})$ to $(2, 6, \sqrt{6})$, which has a volume of

$$\begin{aligned} BoundingBoxVolume &= 48\sqrt{6} \\ ExactVolume &= 33.97874025 \\ VolumeFraction &= ExactVolume/BoundingBoxVolume = 0.288995 \end{aligned}$$

To determine whether a point is inside or outside the paraboloid, we use two PyMercury functions. First, we use the PyMercury function `mc.geometry.cell["paraboloid"]` to access the Python object corresponding to the CG cell "paraboloid," which is shown in Figure 2. Then, given a random point (x, y, z), we can determine which CG cell contains the point. If (x, y, z) is inside "paraboloid," then the function `mc.geometry.locateCoordinate(x,y,z)` returns the same object as was returned by `mc.geometry.cell["paraboloid"]`. We test whether `mc.geometry.locateCoordinate` and `mc.geometry.cell` returned the same object by comparing the names of the two objects. If the names are the same, then the given sample point is inside the hyperbolic paraboloid. Otherwise, the sample point is not inside the paraboloid. We then divide the number of points found to be inside the shape by the total number of points sampled.

Figure 3 shows an example implementation of this volume calculation with PyMercury. Since this Python-based calculation is independent of the Mercury volume calculation code, the calculation can always be used for validation of new Mercury volume calculations.

```
testCell = mc.geometry.cell["paraboloid"] #get geometric cell by name
BoxVolume = (xmax-xmin)*(ymax-ymin)*(zmax-zmin) #draw a bounding box
pointsFoundInCell = 0 #number of points found inside testCell

for i in xrange(numOfPoints):
   x = xmin + (xmax - xmin)*random.random()
   y = ymin + (ymax - ymin)*random.random()
   z = zmin + (zmax - zmin)*random.random()

   #determine which cell the randomly generated coordinates are inside
   cellFoundByCoordinates = mc.geometry.locateCoordinate(x,y,z)
   if cellFoundByCoordinates.name == testCell.name:
      pointsFoundInCell += 1

#volume of cell as determined by Monte Carlo trial.
volume = (BoxVolume * pointsFoundInCell)/(numOfPoints)
```

Figure 3: Point Sampling Monte Carlo Volume Calculation with PyMercury.

To statistically evaluate the convergence of the point-sampling calculation, we vary the number of points sampled or rays traced, n, within the cell's bounding box. We run 1000 trials for five values of n: 10^1, 10^2, 10^3, 10^4, and 10^5. For this test, we fix the processor count at 64 processors.

The mean of the point-sampling calculations converges to the analytically-calculated volume of 33.97874025. As shown in Figure 4, the standard deviation, σ, is proportional to $1/\sqrt{n}$. The standard deviation is approximately $\sigma = 53/\sqrt{n}$.

n samples	mean	σ	$\sigma\sqrt{n}$	$\lvert exact - mean \rvert$
1e+01	33.920534	16.6517	52.6573	5.8206e-02
1e+02	34.153333	5.4193	54.1932	1.7459e-01
1e+03	34.022119	1.6622	52.5621	4.3379e-02
1e+04	33.965413	0.5318	53.1824	1.3328e-02
1e+05	33.976479	0.1653	52.2728	2.2615e-03

Figure 4: Convergence for Point-Sampling Volume Calculation.

5.2 Monte Carlo Volume Calculation Method: Ray Tracing

To perform a volume calculation with ray tracing, we begin with the same setup as was used in the point-sampling volume calculation. However, instead of sampling random points, we launch rays from random points along one face of the bounding box of a CG cell towards the opposite face. We then calculate the fraction of the path length that was *within* the CG cell, compared to the *total* path length traveled. This fraction approximates the volume fraction of the CG cell within its bounding box.

The ray-tracing volume calculation is easily enabled with PyMercury, which provides access to the function `nearestFacet(cell, x,y,z,` α`,` β`,` γ`)`. This function returns the distance to the

nearest surface, given a ray with point coordinate (x, y, z) heading in the direction (α, β, γ). We sample n rays from the $zmin$ face of the CG cell's bounding box, which is: $(xmin, ymin, zmin)$ to $(xmax, ymax, zmax)$.

The Python code in Figure 5 is an example implementation of a ray-tracing volume calculation. This code demonstrates how concisely a ray-tracing based volume calculation can be implemented with PyMercury.

```
testCell = mc.geometry.cell["paraboloid"] #get geometric cell by name
BoxVolume = (xmax-xmin)*(ymax-ymin)*(zmax-zmin)
for i in xrange(numOfRays):
   #generate a random coordinate (x,y,z) on the zmin face of the bounding box.
   x = xmin + (xmax - xmin)*random.random()
   y = ymin + (ymax - ymin)*random.random()
   z = zmin
   CurrentCell = mc.geometry.locateCoordinate(x, y, z)

   while z < zmax:
      (distance, surf, AdjacentCell) = mc.geometry.nearestFacet(CurrentCell, x,y,z, alpha, beta, gamma)
      if CurrentCell == testCell:
         PathLength += min(distance, zmax - z)
      z += distance
      CurrentCell = AdjacentCell

TotalPathLength = n * (zmax - zmin)
Volume = BoxVolume * PathLength / TotalPathLength
```

Figure 5: Ray-Tracing Monte Carlo Volume Calculation with PyMercury.

To evaluate the convergence of the ray-tracing volume calculation, we use essentially the same method that was used in Section 5.1 to evaluate the convergence of the point-sampling calculation. However, instead of varying the number of points, we vary the number of rays, n.

Figure 6 shows that the mean converges to the analytically-calculated volume of 33.97874025. As in the point-sampling calculation, the standard deviation for the ray-tracing calculation, σ, is proportional to $1/\sqrt{n}$.

In the ray-tracing calculation, the standard deviation is approximately $\sigma = 35/\sqrt{n}$, for n rays. For the point-sampling calculation, the standard deviation is roughly $\sigma = 53/\sqrt{n}$, for n points. Therefore, for a number of rays n, the result is has a smaller standard deviation than for the same number of points n. However, given a sufficient number of rays or points, both volume calculation methods consistently converge to the analytically-calculated volume, 33.97874025. Thus, the point-sampling and ray-tracing calculations confirm that the Mercury analytical volume calculation works properly for the new hyperbolic paraboloid CG cell. These "sanity check" volume calculations will be used to validate analytical and AMR volume calculations for future additions to the Mercury CG cell library.

n samples	mean	σ	$\sigma\sqrt{n}$	$\lvert exact - mean \rvert$
1e+01	33.951046	11.5092	36.3953	2.7694e-02
1e+02	33.755955	3.4363	34.3635	2.2278e-01
1e+03	33.954917	1.1544	36.5049	2.3823e-02
1e+04	33.974577	0.3678	36.7823	4.1634e-03
1e+05	33.981827	0.1099	34.7668	3.0864e-03

Figure 6: Convergence for Ray-Tracing Volume Calculation.

To summarize, PyMercury demonstrates that Python interfaces can simplify the testing of geometry-related components of Monte Carlo codes. Geometry code is critical to the correctness of Monte Carlo particle transport applications, and we expect Python interfaces to play an increasingly significant role in the geometry validation process of numerous Monte Carlo codes. In addition to being used for debugging and validating geometry code, interactive Python code and Python test scripts can potentially replace the majority of low-level code currently used for testing Monte Carlo particle transport software.

6. CONCLUSIONS

Interactive Python interfaces are increasingly common in parallel scientific applications that are primarily written in C, C++, and Fortran. Mixed-language codes using Python and C++ can combine the performance advantages of C++ with the simplicity of the Python syntax. However, the Monte Carlo particle transport community has been slow to adopt mixed-language programming models that combine low-level and high-level languages.

With PyMercury, we showed that connecting a Python interface with a Monte Carlo application can improve usability and simplify the process of validating the Monte Carlo code. As we demonstrated in Case Study I, PyMercury improves application usability during Mercury runtime. As shown in Case Study II, PyMercury also simplifies the process of debugging and validating the Mercury code. PyMercury provides these benefits without contributing to performance overhead.

In summary, PyMercury illustrates the benefits of interactive Python and mixed-language programming for Monte Carlo particle transport applications. In the future, we expect mixed-language programming with interactive Python and a low-level language to be increasingly common among Monte Carlo applications.

ACKNOWLEDGEMENTS

The authors thank Laxmikant V. Kale, M. Scott McKinley, Brian Ryujin, Lila Chase, T.J. Alumbaugh, and Jonathan Walsh for providing expertise in parallel computing, mixed-language

programming, and particle transport. This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

REFERENCES

1. F.B. Brown, A.C. Kahler, G.W. McKinney, R.D. Mosteller, and M.G. White. MCNP5 + Data + MCNPX Workshop. *Proc. of Joint International Topical Meeting on Mathematics & Computation and Supercomputing in Nuclear Applications (M&C + SNA 2007).*, Monterey, CA, April 15-19, (2007).

2. B.L. Kirk. Overview of Monte Carlo radiation transport codes. *Radiation Measurements.*, **45**, pp. 1318-1322 (2010).

3. M.F. Sanner. Python: A Programming Language for Software Integration and Development. *Journal of Molecular Graphics and Modeling.*, **17**, pp. 57-61 (1999).

4. D.M. Beazley. Interfacing C/C++ and Python with SWIG. *Proc. of 7th International Python Conference.*, Houston, TX, November 10-13, (1998).

5. R. Procassini, et al. Verification and Validation of Mercury: A Modern, Monte Carlo Particle Transport Code. *Proc. of The Monte Carlo Method: Versatility Unbounded in a Dynamic Computing World.*, Chattanooga, TN, April 17-21, (2005).

6. R. Procassini, et al. New Features of the Mercury Monte Carlo Particle Transport Code. *Proc. of Joint International Conference on Supercomputing in Nuclear Applications and Monte Carlo (SNA + MC 2010).*, Tokyo, Japan, October 17-21, (2010).

7. W. Gropp, E. Lusk and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface.* MIT Press, Cambridge, MA (1999).

8. T.J. Alumbaugh. Dynamic Languages for HPC at LLNL. *Proc. of Virtual Executation Environments for Scientific Computing (VEESC 2010).*, Arlington, VA, September 3-4, (2010).

9. R.A.A. Ince, R.S. Peterson, D.C. Swan, and S. Panzeri. Python for information theoretic analysis of neural data. *Frontiers in Neuroinformatics.*, **3** (2009).

10. F. Gioachin and L.V. Kale. Dynamic High-Level Parallel Scripting. *Proc. of the 23rd IEEE International Parallel and Distributed Processing Symposium (IPDPS 2009).*, Rome, Italy, May 23-29, (2009).

11. B. Thorne and R. Grasset. Python for Prototyping Computer Vision Applications. *Proc. of New Zealand Computer Science Research Student Conference (NZCSRSC 2010).*, Wellington, New Zealand, April 12-15, (2010).

12. P.J.A. Cock, et al. Biopython: freely available Python tools for computational and molecular biology and bioinformatics. *Bioinformatics.*, **25**, pp. 1422-1423 (2009).

13. A.C. Calleja. Scripting a Large Fortran Code with Python. *Proc. of First International Workshop on Software Engineering for High Performance Computing System Applications.*, Edinburgh, Scotland, United Kingdom, May 24, (2004).

14. A. Ferrari, et al. Update on the Status of the FLUKA Monte Carlo Transport Code. *15th International Conference on Computing in High Energy and Nuclear Physics (CHEP 2006).*, Mumbai, India, February 13-16, pp. 294-302 (2006).

15. V. Vlachoudis, et al. FLAIR: A Powerful But User Friendly Graphical Interface For FLUKA. *Proc. of International Conference on Mathematics, Computational Methods, and Reactor Physics (M&C 2009).*, Saratoga Springs, NY, May 3-7, (2009).

16. B. Walker, J. Figgins, and J.R. Comfort. A Framework for Monte Carlo simulation calculations in GEANT4. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, and Associated Equipment.*, **768**, pp. 889-895 (2006).

17. G. van Rossum and F.L. Drake. *Python/C API Manual - Python 2.6.* CreateSpace, Paramount, CA, USA (2009).

18. D. Abrahams and R.W. Grosse-Kunstleve. Building Hybrid Systems with Boost.Python. *C/C++ Users Journal.* (2003).

19. D.M. Beazley. SWIG : An Easy to Use Tool for Integrating Scripting Languages with C and C++. *Proc. of the 4th USENIX Tcl/Tk Workshop.*, Monterey, CA, July 10-13, (1996).

20. J. Generowicz, P. Mato, W.T.L.P Lavrijsen, and M. Marino. Reflection-Based Python-C++ Bindings. *Lawrence Berkeley National Laboratory.*, October 14, (2004).

21. H. Yoriyaz, M.G. Stabin, and A. das Santos. Monte Carlo MCNP-4B-Based Absorbed Dose Distribution Estimates for Patient-Specific Dosimetry. *Journal of Nuclear Medicine.*, **42**, pp. 662-669 (2001).

22. F.B. Brown, R.D. Mosteller, and A. Sood. Verification of MCNP5. *Proc. of Nuclear Mathematical and Computational Sciences: A Century in Review, A Century Anew (M&C 2003).*, Gatlinburg, TN, April 6-11, (2003).

23. W.S. Kiger III, A.G. Hochberg, J.R. Albritton, and T. Goorley. Performance Enhancements of MCNP4B, MCNP5, and MCNPX for Monte Carlo Radiotherapy Planning Calculations in Lattice Geometries. *Proc. of Eleventh World Congress on Neutron Capture Therapy.*, Waltham, MA, October 11-15, p. NEP-14 (2004).

24. B. Hackel, D. Nielsen, and R. Procassini. FZ2MC: A Tool for Monte Carlo Transport Geometry Manipulation. *Proc. of International Conference on Mathematics, Computational Methods, and Reactor Physics (M&C 2009).*, Saratoga Springs, NY, May 3-7, (2009).

25. M. O'Brien, R. Procassini, and K. Joy. Mercury + VisIt: Integration of a Real-Time Graphical Analysis Capability Into a Monte Carlo Transport Code. *Proc. of International Conference on Mathematics, Computational Methods, and Reactor Physics (M&C 2009).*, Saratoga Springs, NY, May 3-7, (2009).

26. R. Procassini, M. O'Brien, and J. Taylor. Load Balancing of Parallel Monte Carlo Transport Calculations. *Proc. of Mathematics and Computation, Supercomputing, Reactor Physics and Nuclear and Biological Applications (M&C 2005).*, Palais des Papes, Avignon, France, September 12-15, (2005).