LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-PROC-453671

New Features of the Mercury Monte Carlo Particle Transport Code

R. J. Procassini, P. S. Brantley, S. A. Dawson, G. M. Greenman, M. S. McKinley, M. J. O'Brien, S. M. Sepke, D. E. Stevens, B. R. Beck, C. A. Hagmann

September 8, 2010

Joint International Conference on Supercomputing in Nuclear Applications and Monte Carlo 2010 (SNA + MC2010)
Tokyo, Japan
October 17, 2010 through October 21, 2010

Disclaimer

This document was prepared as an account of work sponsored by an agency of the United States government. Neither the United States government nor Lawrence Livermore National Security, LLC, nor any of their employees makes any warranty, expressed or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States government or Lawrence Livermore National Security, LLC. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States government or Lawrence Livermore National Security, LLC, and shall not be used for advertising or product endorsement purposes.

New Features of the Mercury Monte Carlo Particle Transport Code

Richard PROCASSINI, Patrick BRANTLEY, Shawn DAWSON,
Gregory GREENMAN, Michael Scott McKINLEY, Matthew O'BRIEN,
Scott SEPKE, David STEVENS, Bret BECK and Christian HAGMANN

Lawrence Livermore National Laboratory, P.O. Box 808, Livermore, CA 94551, United States of America

Several new capabilities have been added to the ***Mercury*** Monte Carlo transport code over the past four years [1,2]. The most important algorithmic enhancement is a general, extensible infrastructure to support source, tally and variance reduction *actions*. For each action, the user defines a *phase space*, as well as any number of *responses* that are applied to a specified *event*. Tallies are accumulated into a correlated, multi-dimensional, Cartesian-product *result* phase space. Our approach employs a common user interface to specify the data sets and distributions that define the phase space, response and result for each action. Modifications to the particle trackers include the use of facet *halos* (instead of extrapolative *fuzz*) for robust tracking, and material interface reconstruction for use in shape overlaid meshes. Support for expected-value criticality eigenvalue calculations has also been implemented. Computer science enhancements include an in-line Python interface for user customization of problem setup and output[3].

KEYWORDS: Monte Carlo, particle transport, tallies, variance reduction, particle tracking, Python user interface

I. Introduction

Mercury is a Monte Carlo particle transport code that is being developed at the Lawrence Livermore National Laboratory (LLNL) in support of a variety of laboratory missions[1,2]. During the past four years, several new capabilities have been added to the code, and as a result, ***Mercury*** is now available for use in general-purpose particle transport calculations. These new capabilities include a general, extensible infrastructure to support source, tally and variance reduction methods, more robust particle tracking, accurate particle tracking through multi-material cells, expected-value criticality calculations and a Python interface for user customization of problem setup and output.

II. A General Infrastructure for Source, Tally and Variance Reduction Methods

The most important capability that has recently been added to ***Mercury*** is a general, extensible infrastructure to support source, tally and variance reduction *actions* (methods). The user defines the relevant *phase space* over which the action is to be applied in the form of *responses* and *results*. For source, tally and variance reduction actions, the user may define any number of responses (filters or intensity multipliers) that are applied to a specified Monte Carlo *event*. User-defined tallies are accumulated into a correlated, multi-dimensional, Cartesian-product result phase space. A common user interface to specify the data collections and distributions that define the response and result phase spaces for each action. This flexible infrastructure makes it easy for the developer to add new events, phase spaces, etc. for any supported action, while the holistic approach to code input minimizes the input syntax that the user is required to learn. This design feature of ***Mercury*** was inspired by the input syntax of several other Monte Carlo codes. A perusal of the user manuals for those codes revealed that the way the 'response' and 'result' phase spaces were defined varied from 'action' to 'action', depending upon who implemented the feature.

The flexibility inherent in this infrastructure permits the ***Mercury*** user to define an extremely complex set of actions to be executed for a variety of events, response and result spaces. In many cases, the flexibility that is available to users of ***Mercury*** far exceeds that available to users of other Monte Carlo particle transport codes. For example, weight windows are a popular variance reduction method provided by ***MCNP***[4]. The capability provided by ***MCNP*** allows the user to define weight windows for any cell that respond to either time *or* energy, but *not both*. As shown below, the capability provided by ***Mercury*** allows the user to define weight windows responses that are a function of time *and* energy, as well as which surface the particle is crossing, the material it is leaving and entering, the original coordinates of the particle, the angle relative to the surface normal, etc. (in either an uncorrelated or correlated fashion). In comparison to ***MCNP***, ***Cog*** provides additional flexibility and input-syntax uniformity for the definition of response and result phase spaces[5]. Indeed, the approach taken by the developers of ***Cog*** was the inspiration for the development of our common, flexible infrastructure. While ***Cog***'s flexibility is an improvement over ***MCNP***, the user cannot define correlated responses for tallies, greater than two-dimensional tally results, and all variance reduction responses have

[1] Corresponding Author E-Mail: spike@llnl.gov

limited functional dependence (cell/material and/or energy). A novel feature of both ***MCNP*** and ***Cog*** is the ability for the user to define their own tally or source functionality by writing Fortran routines that is executed by the code[4,5]. While this approach provides the user with additional flexibility, our philosophy is to provide users with an extremely flexible infrastructure that will meet theirs needs and not to burden them with code development and debugging responsibilities.

The actions, phase spaces, events and special features that are supported within the ***Mercury*** source/tally/variance-reduction infrastructure are detailed below.

1. Actions

Mercury supports various actions or methods which are defined by the category keyword in the relevant input data block. For sources, the supported actions include particle creation via sampling of an external source distribution (External_Source) and reading particle records from a disk file (File_Source). For user-defined tallies, the default action is to accumulate data into the result phase space. In this case, the category keyword refers to the quantity that is to be accumulated. Currently supported categories include:

- Count
- Weight
- Energy
- Path_Length
- Flux
- Fluence
- Point_Detector_Flux
- Point_Detector_Fluence
- Energy_Deposition
- Reaction_Energy
- Net_Current_Count
- Net_Current_Fluence
- Net_Current
- Net_Normal_Current_Count
- Net_Normal_Current_Fluence
- Net_Normal_Current
- Surface_Flux
- Surface_Fluence

In addition, it is possible to write a particle record to a disk file (Particle). For variance reduction, the supported actions fall into three broad classes: termination, population control and modified sampling:

- Cutoff (Termination)
- General (Population Control)
- Importance (Population Control)
- Weight_Window (Population Control)
- Forced_Collision (Modified Sampling)

2. Phase Spaces

While any number of response phase spaces may be defined for a particular source, tally or variance reduction action, tallies accommodate only a single result phase space. The response and result phase spaces are defined through the use of data collections and distributions, which are called sets and boundary sets, or bsets. Each response may include multiple sets and/or bsets, which defines a *correlated* phase space, such as $f(t, E, \Omega, cell)$. By definition, each result defines a correlated phase space. In contrast, if multiple responses each contain a single set or bset, those phase spaces are considered *uncorrelated* and multiplicative, such as $f_1(t) f_2(E) f_3(\Omega) f_4(cell)$. The overall magnitude of a group of responses is simply the product of each individual uncorrelated or correlated response.

A set is a collection of particle or problem attributes that form a discrete space with a histogram representation. The currently supported sets and some example settings for each are shown below:

- Particle (Neutron, Gamma, Deuteron, ...)
- Reaction (Fission, Elastic_Scattering, 2n, ...)
- Material (Uranium, Water, ...)
- Purpose (Transport, Diagnostic)
- Surface (Inner_Sphere, Lower_Plane, ConeA, …)
- Cell (Core, Reflector, Cell_23, …)
- Region (CombGeom, My_Mesh, …)
- From_Material (Air, Lead, …)
- To_Material (Steel, Uranium, …)
- ...

Boundary sets (bsets) are used to specify a distribution of a continuously varying quantity, using either a histogram (zeroth order interpolation) or piecewise linear (first order interpolation) representation:

- Time
- Energy
- X_Coord, Y_Coord, Z_Coord
- Origin_X_Coord, Origin_Y_Coord, Origin_Z_Coord
- Theta_Coord, Phi_Coord
- Alpha_Angle, Beta_Angle, Gamma_Angle
- Theta_Angle, Phi_Angle
- Normal_Angle
- Num_Collisions
- Net_Distance
- Creation_Time
- …

3. Events

While the source routines are obviously executed during source events prior to particle tracking, the tally and variance reduction actions in ***Mercury*** may be applied at a variety of Monte Carlo events. While not all events are applicable to both tally and variance reduction actions, the range of possible events includes:

- Collision_Pre
- Collision_Post
- Creation_Collision
- Creation_External_Source
- Creation_Splitting
- Facet_Crossing
- Energy_Boundary_Crossing
- Internal_Interface
- Thermalization
- Reflection
- Census
- …

4. Functionals

Functionals are tools that are available for use with ***Mercury*** tallies which are designed to 'modify' the analog physics of particle tracking with the intent of improving the quality of the tally result. In some respects, functionals are similar in nature to many variance reduction or biasing methods. Particles can be teleported to other regions of the problem geometry, undergo splitting or Russian roulette, or tagged with an identifier. The complete set of supported functionals is as follows:

- Move
- Move_To
- Project
- Rotational_Imaging
- Split
- Russian_Roulette
- Kill
- Add_Tag
- Remove_Tag

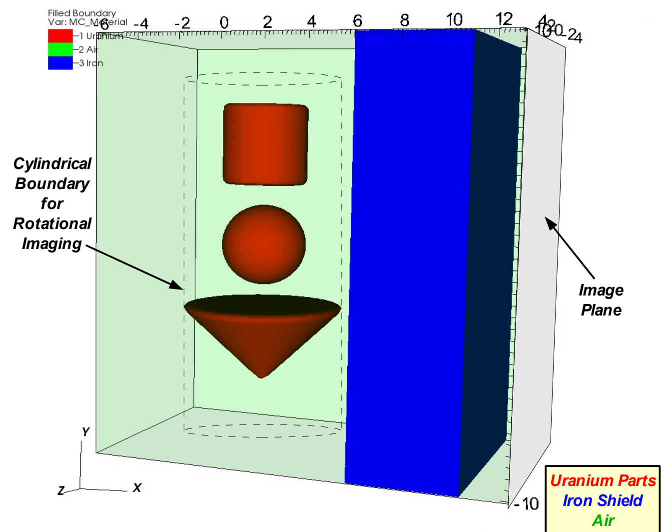


Fig. 1 The configuration of the three part imaging problem.

5. Illustrative Example

To illustrate these concepts, consider the following user-defined tallies. A tally was chosen over a source or variance reduction action in order to demonstrate the result phase space and a functional. Our intent is to create an intensity map of particles that strike an image plane after traveling through air (green) and an iron shield (blue), as shown in **Figure 1**. Neutrons are generated by fission events in the three (red) parts (cone, sphere and cylinder) which are comprised of ^{235}U. Three distinct tallies will be considered: standard transport (**Figures 2** and **3**), transport with a rotational imaging functional (**Figures 4** and **5**), and diagnostic ray tracing (**Figures 6** and 7).

This calculation was run with all three tallies active, and with two types of neutrons: Transport and Diagnostic particles. Transport particles are the 'standard' particles used in all Monte Carlo codes. As the Transport particles undergo collisions in any material, Diagnostic (ray-traced) particles are created and launched towards the image plane. The Diagnostic particles undergo straight line attenuation of their weight in accordance with the material mean-free path in which they are traveling. As either Transport or Diagnostic particles cross the image plane, their weight is accumulated into the result pixel corresponding to the y and z coordinates of the particle.

The input data block and resultant image for the standard transport (first) tally are shown in Figures 2 and 3, respectively. This tally contains three uncorrelated responses and the single result. The result (shown in green) is an image plane defined over $-10 < y < 10$ cm and $-5 < z < 5$ cm with 1 mm square pixels located at $x = 13$ cm. The first response (shown in red) only permits Transport particles to be tallied, while the second response (shown in blue) directs that particles will be tallied upon crossing the surface Imaging-Plane. The third response (shown in pink) is designed to ensure that only particles that are directed towards the image plane pixels (within 5 degrees of the image plane normal) will be tallied, and not those scattered from the extremities of the shield. This example shows that sets and bsets include a domain, which defines the extent of the variable over which the set/bset is active, and a range, which defines the values of the variable over that extent. For *n* domain entries, there are *n* range entries for sets and linearly-interpolated bsets, and *n-1* range entries for histogrammed bsets.

Since (a) fission events emit particles isotropically, (b) the shield is multiple mean-free path lengths thick, and (c) the shield and image plane are localized to one side of the problem geometry, one might expect that only a small fraction of the particles created by fission events would be tallied at the image plane. This is indeed the case, as shown in the tally image of Figure 3.

In an effort to improve the quality of the tallied image, the second tally also records Transport particles as they cross the image plane, however, this tally employs a Rotational_Imaging functional to increase the number of particles that reach the image plane. The input data block and resultant image for the transport-plus-rotational-imaging tally are shown in Figures 4 and 5, respectively. This technique employs (a) the axisymmetry of the uranium parts and (b) the isotropic nature of the fission process to increase the number of Transport particles that cross the image plane. As particles cross the cylindrical surface that bounds the three uranium parts (shown by the dashed cylinder in Figure 1), a Rotational_Imaging functional is used to rotate the velocity vector of the particle such that it has velocity components $v_z{}' = 0$ and $v_x{}' = \sqrt{v_x^2 + v_z^2}$, where the x-axis is normal to the image plane. While this ensures that all Transport particles are directed towards the image plane, the shield is optically thick, and only a fraction of the rotated particles that leave the cylinder will actually cross the image plane and be tallied.

As shown in Figure 4, this imaging techniques employs two tallies. The first is the standard-transport image tally

```
tal Standard-Transport-Image
  category Weight
  events
    Facet_Crossing_Transit_Exit
  end_events
  response
    set Particle-Purpose
      space Purpose
      domain
        Transport
      end_domain
    end_set
    range
      1
    end_range
  end_response
  response
    set Surfaces
      space Surface
      domain
        Imaging-Plane
      end_domain
    end_set
    range
      1
    end_range
  end_response
  response
    bset Acceptance-Angle
#   Accept particles headed within 5 deg of the x-axis.
      space Alpha_Angle
      interpolation None
      domain
        0.99619 1.0
      end_domain
    end_bset
    range
      1
    end_range
  end_response
  result
    bset Y-Values
      space Y_Axis
      domain
        [-10 :10 : 200]
      end_domain
    end_bset
    bset Z-Values
      space Z_Axis
      domain
        [-5 : 5 : 100]
      end_domain
    end_set
  end_result
end_tal
```

Fig. 2 The definition of the standard-transport image tally.

that was used to generate the previous image (shown in violet in Figure 4). The second tally is designed to implement the rotational imaging operation. This tally contains a single, correlated response (shown in red) which ensures that only Transport particles that crossing the surface

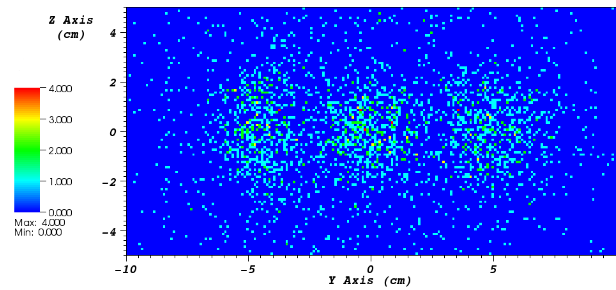


Fig. 3 The result of the standard-transport image tally exhibits exhibits extremely poor image quality.

Container-Cylinder into cell Air-Cell-1 will undergo the rotational operation. The Rotational_Imaging functional (shown in turquoise) is designed to rotate particles that are enabled by the response around the cylinder that is aligned along the Y-Axis and relaunch them in the Positive x direction. Note that no result block is included in this tally. In this case, a scalar tally result is being accumulated: the sum of the the number of particles that have been rotated. In comparison to the standard transport image (Figure 3), it is now possible to discern the general shape of the three uranium parts (see Figure 5). However, the periphery of each part exhibits “stochastic fuzz”, and the exact location of the interface between uranium and air is not obvious.

In order to further improve the quality of the tallied image, the third tally will record Diagnostic particles as they cross the image plane. The input data block and resultant image for the diagnostic-ray-trace image tally are shown in Figures 6 and 7, respectively. For this tally, any Transport particle that undergoes a collision within the system will launch a Diagnostic particle, oriented parallel to the x-axis, towards the image plane, such that all of the Diagnostic particles will cross the image plane. However, the tallied weight will be reduced from weight at “creation” by straight line attenuation and $1/r^2$ spreading as the particle travels through the intervening materials.

The tally definition shown in Figure 6 is similar to the definition of the standard imaging tally shown in Figure 2, with two differences. Instead of responding to Transport particles, this tally will only respond to Diagnostic particles. In addition, the response that accepts particles traveling within a small angular band of the x-axis has been removed, since this effect is now obtained through use of the Diagnostic particles. The resultant image, shown in Figure 7, demonstrates the superior image resolution capabilities of Diagnostic particles. The interfaces between the uranium parts and air environment can now be easily discerned. In addition, the image shows regions within the parts where fewer Diagnostic particles are being created. These peripheral regions either have fewer optical depths through the part than other regions (such as the large radius edges of the cone) or are farther from the location of the source (shown by the red dot at the center of the sphere) that initiated subsequent fissions (such as the corners on the right hand side of the cylinder).

```
tal Standard-Transport-Image
  # This is the same tally as shown in Figure 2.
end_tal

tal Cylinder-Rotate
  category Count
  events
    Facet_Crossing_Transit_Enter
  end_events
  response
    set Particle-Purpose
      space Purpose
      domain
        Transport
      end_domain
    end_set
    set Surfaces
      space Surface
      domain
        Container-Cylinder
      end_domain
    end_set
    set Entering-Cells
      space Cell
      domain
        Air-Cell-1
      end_domain
    end_set
    range
      1
    end_range
  end_response
  functional
    type Rotational_Imaging
    direction_axis X_axis
    about_axis Y_axis
    orientation Positive
  end_functional
end_tally
```

Fig. 4 The definition of the transport-plus-rotational-imaging image tally.

```
tal Diagnostic-Ray-Tracing-Image
  category Weight
  events
    Facet_Crossing_Transit_Exit
  end_events
  response
    set Particle-Purpose
      space Purpose
      domain
        Diagnostic
      end_domain
    end_set
    range
      1
    end_range
  end_response
  response
    set Surfaces
      space Surface
      domain
        Imaging-Plane
      end_domain
    end_set
    range
      1
    end_range
  end_response
  result
    bset Y-Values
      space Y_Axis
      domain
        [-10 :10 : 200]
      end_domain
    end_bset
    bset Z-Values
      space Z_Axis
      domain
        [-5 : 5 : 100]
       end_domain
    end_set
  end_result
end_tal
```

Fig. 6 The definition of the diagnostic-ray-trace image tally.

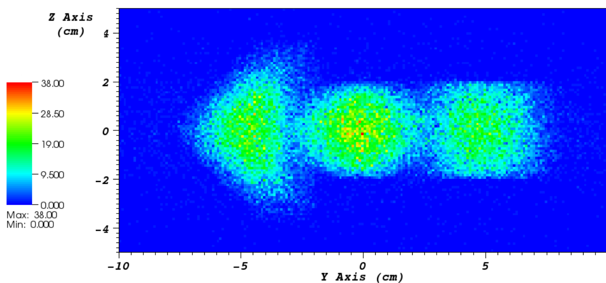


Fig. 5 The result of the transport-plus-rotational-imaging standard transport image tally exhibits improved image quality in comparison with the standard transport tally.

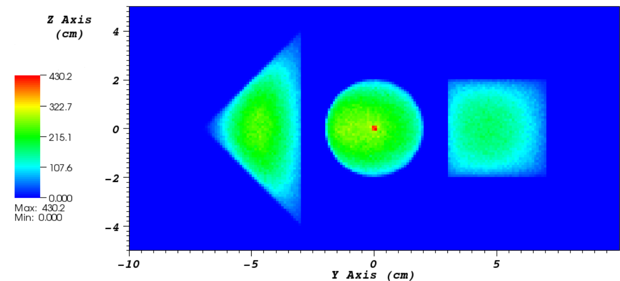


Fig. 7 The result of the diagnostic-ray-trace image tally demonstrates the superior image resolution capabilities of Diagnostic particles.

III. Robust Particle Tracking Via Halos

The limited precision arithmetic that is inherent with the use of digital computers often leads to inaccuracies in the tracking of particles through complex geometries. This is particularly true when particles are being tracked to second-order analytic surfaces, such as spheres, ellipsoids and cones. For these surfaces, solution of the quadratic equation to obtain the distance to intersection of the particle-velocity vector and the surface provides only $O(7)$ digits of accuracy for 64-bit double-precision arithmetic. This limited precision determination of the intersection coordinates can lead to tracking errors if the particle has not actually crossed the surface. This occurs when the particle is still located within the cell it is exiting, instead of the cell that it should have entered.

A common technique for overcoming the inaccuracies associated with limited-precision particle tracking is the use of extrapolative *fuzz*. This technique assumes that, since the particle coordinates are uncertain, due to limited-precision tracking, one is free to advance the particle forward in the direction of the travel by a small distance to ensure that the particle actually enters the intended cell, as shown by the red arrow in **Figure 8**. The distance that the particle is *fuzzed* can either be absolute or relative to a characteristic cell length.

While this technique is successful in many situations, there are two important limitations of the method. First, the addition of a fuzz may not be sufficient to ensure that the particle actually exits the current cell and enters the intended cell (see **Figure 9**a). If the method assumes that a single application of fuzz is sufficient to enter the intended cell, then the particle may be assigned to the wrong cell. This can lead to the particle becoming "lost" in the geometry. Second, it is possible that the addition of fuzz may be too large, and the particle may be moved outside of the intended cell (see Figure 9b). This situation is possible when the intended cell has a large aspect ration with a narrow length in the direction of particle travel, or when several cells converge at a corner of the intended cell. Our experience has shown that reliance on fuzz to correct particle tracking issues arising from limited-precision arithmetic is a "losing proposition". As one of these "end cases" would arise, a developer would "tweak" the magnitude of the fuzz to overcome the immediate problem, only to find that the new amount of fuzz would uncover tracking issues in other geometries. Clearly, there had to be better technique to overcome these issues.

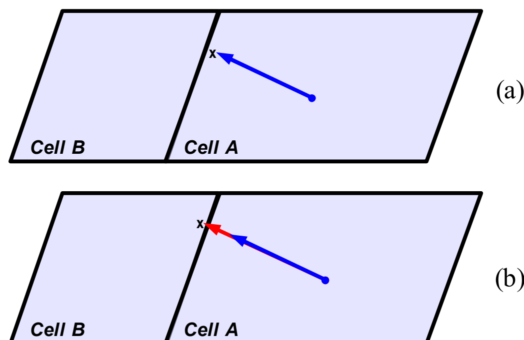


Fig. 8 The 'fuzz' method advances the particle along its trajectory to ensure that it actually enters the intended cell. (a) Limited-precision arithmetic places the particle to the left of the cell facet, so the particle remains in Cell A. (b) The particle is 'fuzzed' forward a small distance, so the particle crosses over the cell facet into Cell B.

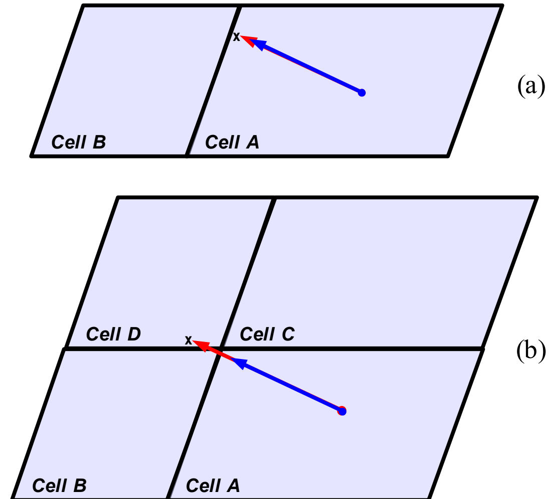


Fig. 9 Limitations of the 'fuzz' method. (a) The addition of fuzz is not sufficient to ensure that the particle exits Cell A and enters Cell B. (b) Too much fuzz is added such that the particle overshoots the intended cell (Cell B) and is located in a different cell (Cell D).

By changing our "frame of reference", a method has been developed that solves these issues. In contrast to the assumption inherent in the fuzz method that the particle coordinates are uncertain, the 'halo' method assumes (a) that the cell containing the particle is known and (b) the acceptance criteria for a particle being assigned to a cell includes the possibility of the particle being located within a small, exterior buffer zone (halo) of width ε. This method accepts a particle as within a cell if it lies either within the

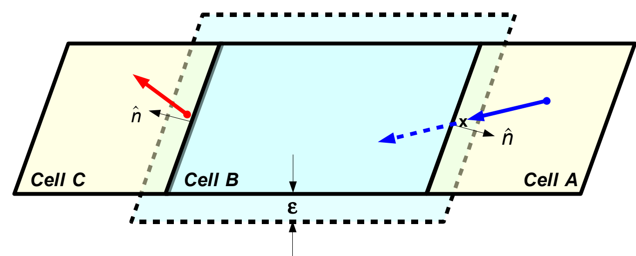


Fig. 10 The 'halo' method assumes that the cell containing the particle is known, and the location of the cell facets are extended outward by a distance ε. The blue particle is tracked from Cell A and is assigned to Cell B, even if its coordinates place it outside of, but within the halo of, Cell B. The red particle is assigned to Cell B because it was sourced onto the facet between Cells B and C, which is within the halo of Cell B.

cell or its associated halo (see **Figure 10**).

As a particle tracks to a cell bounding facet, the particle is unconditionally assumed to undergo a facet crossing event and enter the new cell. This is shown by the track of the blue particle in Figure 10, which is exiting Cell A and entering Cell B. The cell attribute of the particle is set to Cell B, even though the coordinates of the particle place it just outside the cell, but within its associated halo. During the next segment of the particle's track (the dashed blue particle vector), the standard tracking method would find the minimum facet distance to be from the point 'x' to the facet between Cells A and B. However, since (a) the particle has already been assigned to Cell B, and (b) the dot product of the particle velocity and facet outward normal is negative, distance to facet calculations will only be performed for facets with positive velocity-normal dot products (left and bottom facets of Cell B). This ensures that the particle exits the cell that it has been assigned to.

Unfortunately, the use of halos complicates the locate-coordinate methods. Consider the track of the red particle in Figure 10. This particle has been sourced into the system at a location that is within the halo of Cell B, as well as within Cell C proper. In order to prevent double counting of the particle, it must be uniquely associated with one of those cells. This task can be costly if the facet between Cells B and C is also a domain boundary, and the cells are assigned to different processors. In this case, a multi-step communication process is required, during the cycle initialization phase, in order to assign the particle to the correct cell / domain / processor so it is free to continue its track.

An important consideration is the choice of the halo width ε, and whether the width uses an absolute or relative scale length. After several tests, it was decided to set ε to a small multiple $O(10^{-7})$ of a characteristic length scale of the cell. Once the halo method was implemented in ***Mercury***, our team noticed a marked decrease in the number of occurrences in which a particle became "lost" during tracking. This method has produced more robust particle tracking through both mesh-based and combinatorial geometries. It has been proven effective in tracking particles through meshes with skewed or large aspect ratio cells, which have historically been problematic. Modifications of this method have also been developed to support nearly tangential tracking through cells with curvilinear surfaces.

IV. Material Interface Reconstruction for Accurate Mesh-Based Particle Tracking

A common technique for defining complex geometric regions on orthogonal meshes is known as shape (sphere, ellipsoid, cylinder, cone, etc.) overlaying. During this process, the interface between two regions or materials is mapped onto the cells of the mesh. The result is a set of multi-material cells along the interface, which effectively smears the interface over the width of a cell. The relative concentration of each material in a cell is defined by the material volume fraction f_V^m. The cell-based properties are then defined by the weighted sum of the material-based properties for each material *m* in the cell:

$$\rho = \sum_{m=1}^{M} \left(f_V^m \rho^m \right) \tag{1}$$

where ρ^m is the density of the m-th material in the cell and ρ cell density. A standard approach to tracking particles through multi-material cells is to "atomically mix" the isotopes in each material within the cell. The number density of any isotope *i* that occurs in multiple materials within the cell is given by:

$$n_i = \sum_{m=1}^{M} \left(\frac{f_V^m f_M^{i,m} \rho^m N_{Av}}{A^i} \right) \tag{2}$$

where $f_M^{i,m}$ is the mass fraction of the *i*-th isotope in material *m*, N_{Av} is Avogadro's number and A^i is the atomic mass of isotope *i*. This method is (a) easy to implement and (b) conservative with regard to the number of atoms of each isotope in the cell. However, it can lead to serious particle tracking inaccuracies when the optical depths of the materials within the interface cell are widely disparate. Relevant examples include the interface between the fissile material in a bare, spherical critical assembly and the surrounding air (neutron transport), or between a plasma and its confining metallic surface (charged particle transport).

To overcome the shortcomings associated with tracking particles through multi-material cells, a material interface reconstruction (MIR) algorithm has been implemented within the 1-D spherical and 2-D unstructured mesh-based particle trackers in ***Mercury***[6]. This method works by converting each multi-material cell into two single-material *sectors* within the cell. The material interface, which forms the facet between the two single-material sectors, is reconstructed from the volume fractions of the two materials in the cell f_V^1 and f_V^2 (see the thick black lines in **Figure 11**). Currently, only multi-material cells which contain two materials are supported. The atomically mixed prescription is used for cells with more than two materials.

To study the efficacy of the MIR mesh-based particle tracker, consider the 2-D, axisymmetric *r-z* representation of the Godiva critical assembly shown in Figure 11. Godiva is a bare spherical assembly of radius $r = 8.7407$ cm, which is comprised of highly-enriched uranium. Godiva has been given the International Criticality Safety Benchmark Evaluation Project (ICSBEP) name HEU-MET-FAST-001-001[7]. Our 2-D mesh-based model of Godiva shape overlays the sphere onto an orthogonal *r-z* mesh, in which the cells form cylindrical annuli. The region outside of the sphere is modeled as nominal density air. Figure 11 shows a coarse, 1.0 cm resolution representation of the 2-D mesh (the thin black lines), which extends out to 10.0 cm. The reconstructed material interfaces is shown by the thick black line segments, which together form a representation of the spherical interface. Also shown in Figure 11 is a point representation of the simulation particles for the MIR enabled calculation with a 1.0 cm cell size. The particles are color coded in accordance with the pure material within each cell or sector. Figure 11 clearly shows that the MIR tracking method is effective, since only red points are visible within

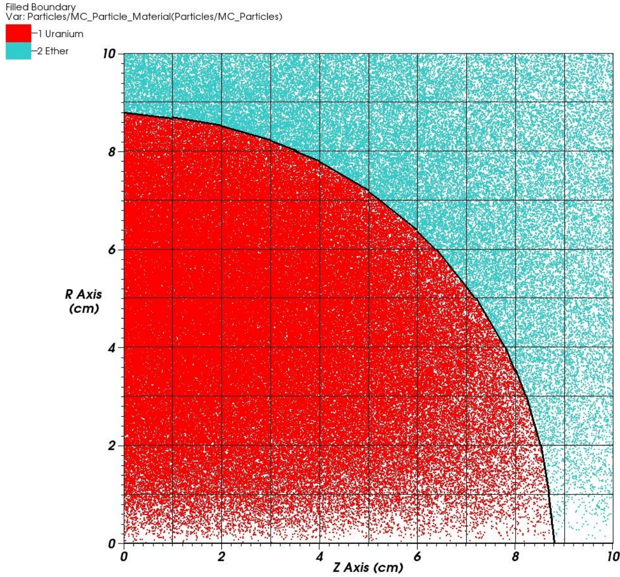


Fig. 11 Particles, mesh and material interfaces in a material interface reconstructed (MIR) mesh-based calculation of the *k*-eigenvalue for the Godiva critical assembly.

the piecewise discontinuous interface, and teal points are found outside of that interface. Note that the low particle density observed near the z-axis, commonly referred to as the "cone of inadequacy" arises due to the axisymmetric nature of the mesh, and hence, the low cell volumes close to the axis of rotation[8].

The *k*-eigenvalue of this system was calculated for several cell sizes in the range $0.1 \le \Delta_{r,z} \le 2.0$ cm. For each cell size, calculations were performed in which the MIR method was either disabled or enabled. The results of these mesh-based calculations are shown below in **Table 1** and **Figure 12**. These calculations used continuous-energy ENDL-2009.0 nuclear data, $N_p = 2.5 \times 10^4$ particles, 25 (500) "inactive" (maximum) generations, and a convergence tolerance of $\varepsilon = 1 \times 10^{-3}$. The MIR enabled (red curve) / disabled (blue curve) results are presented, along with a reference result (dark-gray curve) that was obtained from a combinational geometry version of the problem. The dotted lines in Figure 12 represent the standard deviation offset from the reference result. Table 1 indicates that the MIR

Table 1 The efficacy of mesh-based particle tracking using MIR

Cell Size *(cm)*	*k* Eigenvalue MIR Enabled	*k* Eigenvalue MIR Disabled	Difference *(%)*
0.10	0.9999893	0.9990349	0.10
0.25	0.9982933	0.9992048	-0.09
0.50	0.9979354	0.9982782	-0.09
1.00	0.9981178	0.9911490	0.70
2.00	0.9974251	0.9777487	1.97

Reference *k* Eigenvalue: $k = 0.9996063 \pm 0.001$
(Combinatorial Geometry)

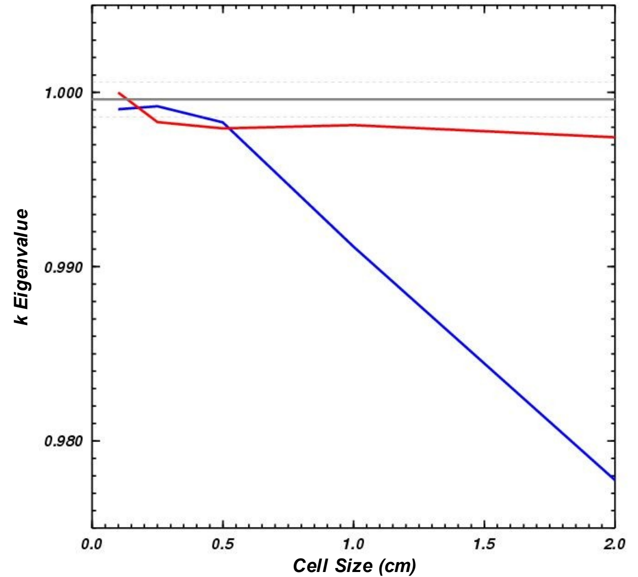
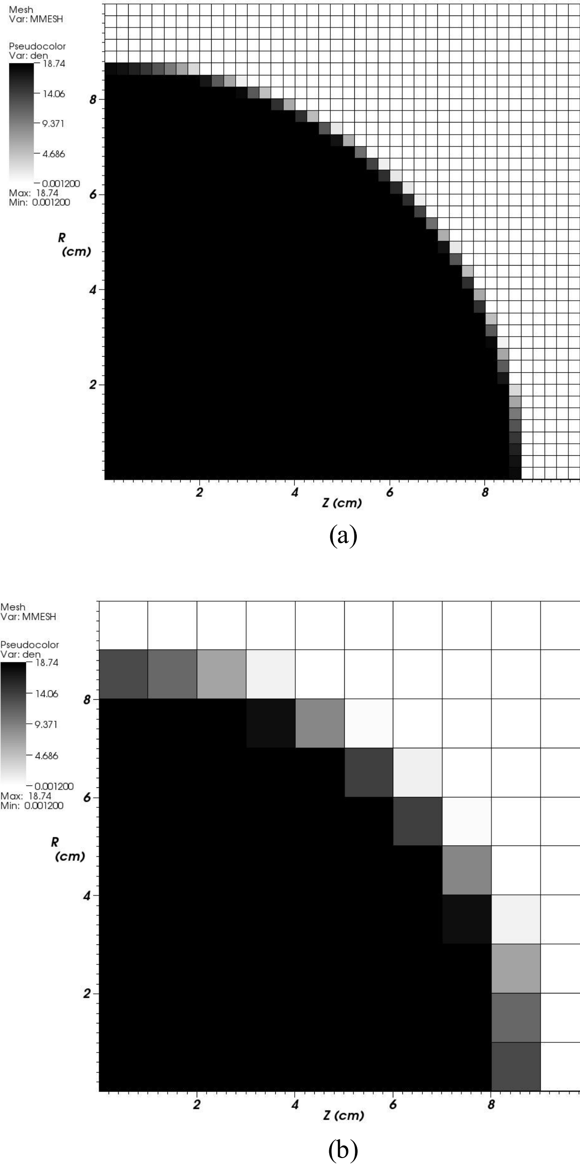


Fig. 12 The efficacy of mesh-based particle tracking using material interface reconstruction (MIR) is shown for a variety of cell sizes in the range $0.1 \le \Delta_{r,z} \le 2.0$ cm. The MIR disabled / enabled *k*-eigenvalue results are shown as blue / red curves, while the reference result and associated statistical uncertainty are shown as solid and dotted dark-gray curves.

enabled / disabled results for are in agreement to within ~1.5 times the statistical uncertainty (standard deviation ~0.001) for $\Delta_{r,z} \le 0.5$ cm, and are also in agreement with the reference results. For larger cell sizes ($\Delta_{r,z} \ge 1.0$ cm), the MIR disabled results fall off linearly from the reference results, while the MIR enabled results remain within about twice the standard deviation.

The reason for the sudden fall off in the computed *k* eigenvalues from the MIR disabled calculations with $\Delta_{r,z} \ge 1.0$ cm becomes clear when one examines the density profiles of the uranium / air interface in the mesh-based calculations. **Figure 13** presents "inverse radio-graphs" of the average cell density, as defined by Equation 1, for the $\Delta_{r,z} = 0.25$ cm (Figure 13a) and $\Delta_{r,z} = 1.0$ cm (Figure 13b) MIR-disabled calculations. The highest density (pure uranium) cells are black, while lower density cells are shown in a gray scale palette, and the lowest density (pure air) cells are white. As the resolution of the problem decreases (Figure 13a to 13b), the uranium-air interface is smeared out over a larger scale length. As the size of the cells increases and becomes comparable to, or larger than, the mean-free path length, one should expect that the total number of mean-free path lengths a neutron travels "within the sphere" to decrease on average.

Comparing the two images, it is clear that most of the cells in the higher resolution calculation (Figure 13a) are either pure uranium or pure air, with a small number of multi-material interface zones $\left(N_{cell}^{multi} / N_{cell}^{pure} \sim 1\%\right)$. As a result, the number of mean-free path lengths a neutron travels through a multi-material cell is small, and the *k* eigenvalue obtained from the MIR disabled calculation is



(a)

(b)

Fig. 13 Inverse radiographs of the cell density in the MIR-disabled, mesh-based calculations of the *k* eigenvalue for the Godiva critical assembly. The cell sizes for the calculations are (a) $\Delta_{r,z} = 0.25$ cm and (b) $\Delta_{r,z} = 1.0$ cm, respectively.

close to the reference value. In contrast, the fraction of multi-material cells in the lower resolution calculation (Figure 13b) is much larger $\left(N_{cell}^{multi} / N_{cell}^{pure} \sim 18\,\%\right)$, and the MIR-disabled *k* eigenvalue is about 0.8% less than the reference value.

Based upon the results presented in Table 1 and Figure 12, one can deduce that the neutron mean-free path length in Godiva is between 0.5 and 1.0 cm. The MIR enabled simulations produce accurate *k* eigenvalues, even when the mean-free path length is not resolved. The reduced cell counts in the lower resolution calculations mean fewer facet crossings, so the low-resolution MIR enabled calculations have shorter run times while continuing to produce accurate results. Although the Godiva simulations prove the efficacy and accuracy of the MIR method, the large ratio of the neutron mean-free path to the system scale length (~5–10%) limits the efficiency of the method. It is expected that the efficiency of the method will become larger as this ratio decreases (consider α particles transporting to a metal wall in a bounded ICF plasma).

V. Expected-Value Criticality Calculations

Consider the *analog* method employed in ***Mercury*** for calculating a *k* eigenvalue in a multiplying system. Neutrons are sourced into the system using an initial guess for the converged spatial and energy distributions. Particles are then tracked for a single 'generation' until experiencing a removal event, either leakage from the system or absorption by a nucleus. The secondary particles produced by non-terminal absorption (non-capture) events serve as the source for the next generation. The analog estimate of the *k* eigenvalue at any generation *i* is given by:

$$k^i = \left(\frac{N^i_{\mathrm{prod}}}{N^i_{rem}}\right) = \left(\frac{N^i_{\mathrm{prod}}}{N^i_{leak} + N^i_{\mathrm{abs}}}\right) \tag{3}$$

where N^i_{prod} is the number of simulation particles produced following an absorption event, N^i_{rem} is the number of particles removed from the system, N^i_{leak} is the number of particles that leak from the system and N^i_{abs} is the number of particles absorbed within the system. This procedure is repeated until a running-average of the eigenvalue converges within the user specified tolerance for five successive generations. The number of particles absorbed, and subsequently produced, are obtained directly from the collision routine on a per collision basis. Unfortunately, this method is prone to statistical fluctuations in the various terms of Equation 3 between successive generations, leading to fluctuations in successive estimates of *k*. Therefore, a large number of generations can be required to obtain a converged result.

To minimize the impact of these statistical fluctuations of the calculated value of *k*, an *expected value* eigenvalue method has recently been implemented. Instead of obtaining N^i_{prod} and N^i_{abs} by summing the results of each collision event, this new method begins by calculating the neutron fluence during the generation in each cell *c* and energy group *g*:

$$\phi_{c,g} = \sum_p \left(\frac{L^p_{c,g} W^p}{V_c}\right) \tag{4}$$

where $L^p_{c,g}$ is the segment path length of the *p*-th particle in cell *c* and group *g*, W^p is the weight of the *p*-th particle and V_c is the volume of cell *c*. The number of particles produced and absorbed in the *i*-th generation are given by:

$$N^i_{\mathrm{prod}} = \sum_c \sum_g \left(\sigma_{gain,g}\phi_{c,g}\right) \tag{5}$$

and

$$N^i_{\mathrm{abs}} = \sum_c \sum_g \left(\sigma_{\mathrm{abs},g}\phi_{c,g}\right) \tag{6}$$

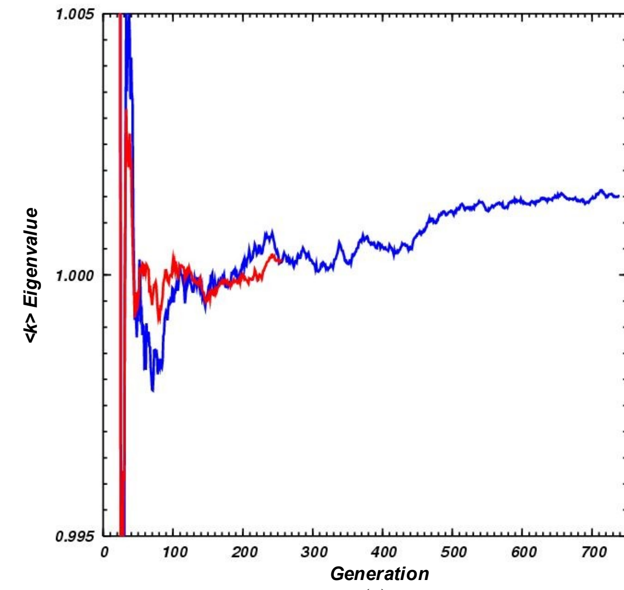
where

$$\sigma_{gain,g} = \sum_r \left((\nu_{r,g} - 1)\sigma_{r,g}\right) \tag{7}$$

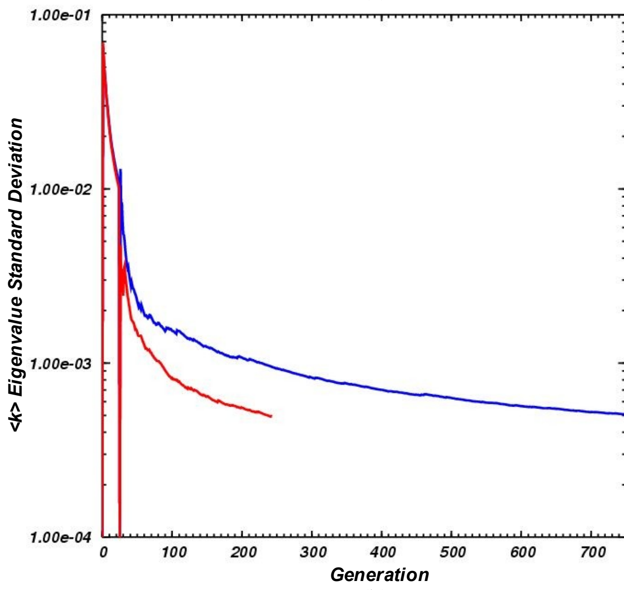
is the neutron 'gain' cross section $\sigma_{\text{abs},g}$ is the absorption cross sections for group *g*, $\nu_{r,g}$ is the secondary neutron multiplicity and $\sigma_{r,g}$ is the partial cross section for reaction *r* and group *g*. This method produces estimates of the *k* eigenvalue that are less noisy than the per collision method because the fluence averages neutron interactions over multiple track segments, and hence, multiple collisions.

To determine efficacy of this expected value eigenvalue method, it was compared to the analog method on two critical assemblies: the fast-spectrum, metallic-uranium Godiva system (HEU-MET-FAST-001-001) and a thermal-spectrum, plutonium-solution spherical system (PU-SOL-THERM-001-001)[7]. The generation histories of the (a) averaged *k* eigenvalue and (b) associated standard deviation are shown for these two critical assemblies in **Figures 14** and **15**, respectively. The converged eigenvalues, required number of iterations and resultant run times for these calculations are presented in **Table 2**. These calculations used 230 group ENDL-2009.0 nuclear data, $N_p = 1\times10^4$ particles, 25 “inactive” generations (after which the running averaged *k* eigenvalue is discarded), a maximum of 1000 generations and a convergence tolerance of $\varepsilon = 5\times10^{-4}$.

Figures 14a and 15a indicate that the analog (blue) and expected value (red) eigenvalue generation histories are similar, and converge within the tolerance of each other. However, the expected value calculations reach the convergence tolerance in only 29% to 72% of the number of active generations that are required for the analog

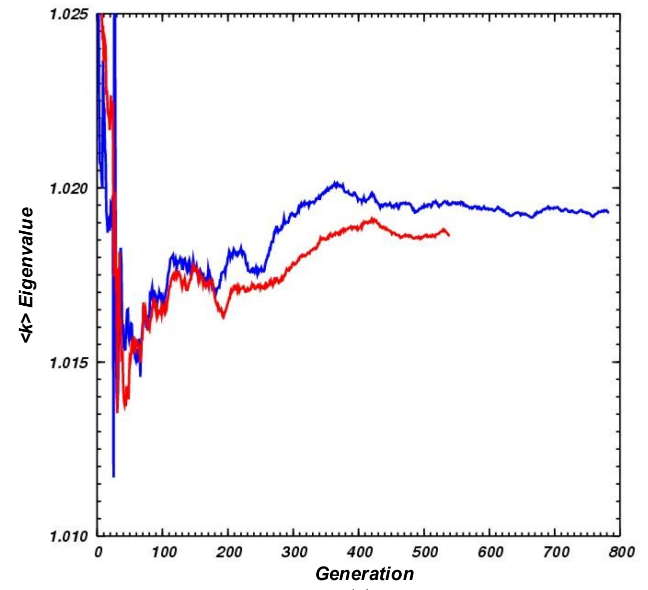


(a)

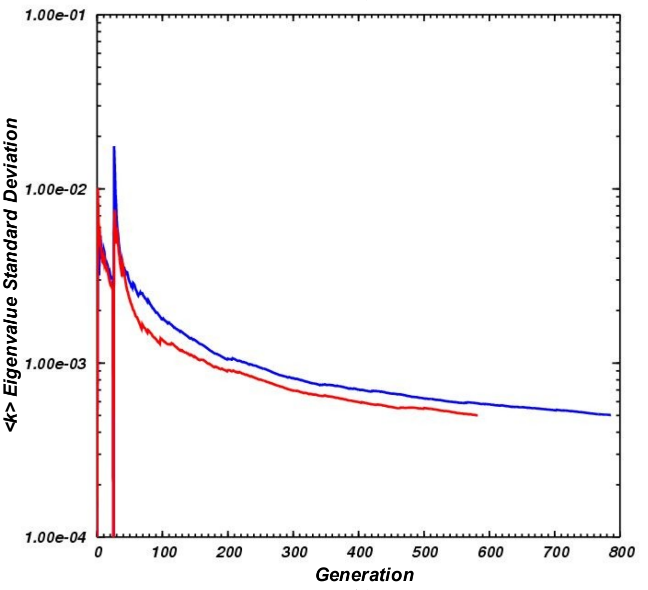


(b)

Fig. 14 Comparison of the analog (blue curves) and expected value (red curves) methods on the calculation of the *k* eigenvalue of the Godiva / HEU-MET-FAST-001-001 uranium critical assembly. (a) $\langle k \rangle$ eigenvalue, and (b) standard deviation of the $\langle k \rangle$ eigenvalue.



(a)



(b)

Fig. 15 Comparison of the analog (blue curves) and expected value (red curves) methods on the calculation of the *k* eigenvalue of the PU-SOL-THERM-001-001 plutonium critical assembly. (a) $\langle k \rangle$ eigenvalue, and (b) standard deviation of the $\langle k \rangle$ eigenvalue.

Table 2 Performance of analog and expected value criticality calculations

Method	*k* Eigenvalue	Generations *Total/Active*	Run Time *(sec)*	Time/Gen *(sec)*
Critical Assembly HEU-MET-FAST-001-001				
A	1.001503	776 / 751	4.98	6.42×10^{-3}
EV	1.000286	244 / 219	1.56	6.39×10^{-3}
Critical Assembly PU-SOL-THERM-001-001				
A	1.019285	787 / 762	111.68	1.42×10^{-1}
EV	1.018661	583 / 558	82.97	1.42×10^{-1}

calculations. This is borne out by the standard deviation histories shown in Figures 14b and 15b, in which the curves end when $\sigma/\langle k \rangle < \varepsilon = 5\times10^{-4}$. It is clear that the approach to convergence is more rapid for the fast uranium system than it is for th thermal plutonium system. Note that the variability in these figures near 25 generations is a consequence of the switch from inactive to active averaging of the *k* eigenvalue. Finally, consider the run times shown in Table 2. It is clear that the required expected value (EV) run times and generations to convergence are shorter than that require for the analog (A) calculations. However, the run time required per generation are the same for both methods, indicating that there is minimal overhead associated with calculation of the particle fluence and reaction rates (Equations 4 – 6).

VI. Python User Interface

Recently, our team has extended the capabilities of the ***Mercury*** code to permit user customization of problem setup and output via the use of Python[3]. Python is a remarkably-powerful, object-oriented, dynamic programming language that has been used extensively at LLNL to provide a flexible user interface in several multi-physics codes[9]. Since Python is designed for use as a high-level dynamic scripting language, it is not suitable for direct use in high-performance applications. However, Python has been designed to interface with high-performance code modules written in C and C++, which are the native languages of ***Mercury***. Use of Python within ***Mercury*** falls into two categories: (a) two separate Python pre-parsers have been developed to permit customization of problem input, and (b) several of the C/C++ data structures in ***Mercury*** have been wrapped to permit customization of the output and computational steering of the problem via an inline Python user interface. These applications of Python within ***Mercury*** are discussed below.

1. Python Input Pre-Parsing

Two Python input pre-parsing methods have been developed for use with ***Mercury***. The first method uses a mix of standard string-based input along with Python functions and scripts. In contrast, the second method solely uses a Python script to generate the string-based input file. In either case, the resultant string-based input file undergoes standard ***Mercury*** parsing via conversion of the string to XML and subsequent DOM tree parsing[10]. The details of these pre-parsing methods are discussed below.

(A) Mixed Python/String Input Pre-Parser

Customization of ***Mercury*** input files via Python pre-parsing minimizes the amount of time that users spend developing and maintaining input files. For example, users often perform sensitivity studies of a particular problem, varying the evaluated nuclear data (ENDF/B, ENDL, JEFF, etc.), cross section representation (continuous energy vs multigroup), geometry definition (combinatorial geometry vs mesh), criticality convergence criterion, etc. Previously, users were required to either (a) maintain separate input files for each variant or (b) edit the input file to change the parameters of interest for each specific simulation. Our experience has shown that these approaches are time consuming to maintain, difficult to keep in sync and prone to typo errors.

The first feature of this mixed method is the ability to embed a block of Python coding within the string input file. While this is placed within a set of python … end_python delimiters, the scope of Python variables defined within the block is global across he entire input file. This allows the user to perform a complex calculation in Python to define a variable, and use that variable later on in the input file. To aid the user in distinguishing these variables from keywords, the code requires hat all Python variables be placed with a set of { … } delimiters. Any number of Python blocks may be used within the input file, provided they are each placed within a set of python … end_python delimiters.

The second feature of this method are a set of Python functions which do not require the user to be fluent in Python in order to modify the values of keywords and control the execution of the problem. Currently, three types of functions have been implemented, each of which are pre-pended by the characters 'mc_'. The mc_ifundef command allows the user to set the default value for a Python variable:

mc_ifundef MyVariable {1.234}

where the *string* variable 'MyVariable' is initialized to the value '1.234'. Later in the parsing process, this string assigned to the proper type, based upon the keyword with which it is associated. The user is free to redefine the value of each variable defined with a mc_ifundef command by including a -def option on the ***Mercury*** execution line:

mercury <*input-file-name*> -def MyVariable=5.678

The mc_include command allows the user to insert a named file into the input file at the calling location:

mc_include {'OtherFile.txt'}

where the file name must be inserted within a set of single (') or double (") quotes. This feature permits modular construction of input files and re-use of common files across multiple input files. Finally, ***Mercury*** supports logical operations that permit the user to customize the input and problem execution based upon several possible options. This is accomplished via the mc_if {*expression*} …

```
mc_ifundef IncludeSphere    {True}
mc_ifundef IncludeTarget    {True}
mc_ifundef NuclearData      {'endl/2008.2'}
mc_ifundef NumParticles     {10.0}

python
PulseWidth = 2.0
FlightPath = 945.54
DetectorAngle = 30.0
DetectorBias = 'NE213_16'
end_python

mc_include {"Pulsed_Sphere_Control.txt"}
mc_include {"Pulsed_Sphere_Time.txt"}
mc_include {"Pulsed_Sphere_Material.txt"}

...

particle
# Particle Data
  part Neutrons
    particle_type Neutron
    target_num_particles {NumParticles*1.0e+6}
    thermalization Thermal_Scatter
    cross_section_data_file /usr/Mercury/data/
{NuclearData}/mcf1.pdb
    num_diagnostic_bins 36
    diagnostic Point_Detector_n
      direction Point
      point_location {PointDetectorX} 0.0 {PointDetectorZ}
    end_diagnostic
  end_part
end_particle

...

source
# External Source Data
  src ExternalSource
    source_type External_Source
    particle Neutrons
    geometry
      category Point
      center_coords 0.0 0.0 0.0
    end_geometry
      ...
    response
mc_if {PulseWidth == 0.3}
      bset Times_03
        space Time
        interpolation None
        domain
          ...
        end_domain
      end_bset
mc_elseif {PulseWidth == 2.0}
      bset Times_20
        space Time
        interpolation None
        domain
          ...
        end_domain
      end_bset
mc_endif
```

Fig. 16 An example of the mixed Python/string input pre-parser.

mc_elseif {*expression*} … mc_endif construct. The logical expressions make use of Python variables that have been defined either in a previous python … end_python block, or via the mc_ifundef / -def construct. Logical comparators and sub-expression grouping utilize the C/ C++ syntax: ==, &&, ||, !, (…):

```
mc_if {MyBoolean == True}
  ...
mc_elseif {(!(a == 2.0)) && (b == 3.0)}
  …
mc_endif
```

A typical usage of these features of the mixed Python/string input pre-parser is shown in **Figure 16**. In this input file for a pulsed sphere calculation, a Python code block is embedded within the pink python … end_python delimiters, while the various Python commands are shown in blue (mc_ifundef), red (mc_include) and green (mc_if, etc.). All Python variables are shown *italicized*, and braces {...} surround all variables and command arguments. This input file makes use of Python variables that are defined both within the embedded python … end_python block (*PulseWidth)*, as well as via the mc_ifundef / -def construct *(NuclearData, NumParticles)*. Note that those variables defined within the Python code block are considered static, while those defined via a mc_ifundef can be dynamically redefined each time the problem is run.

(B) Python Script Pre-Parser

A set of Python functions have recently been developed for those users who prefer to define the input parameters for their problems via a Python script. This capability permits a user to define Python objects which have the same hierarchical layout as the data blocks in the ***Mercury*** input file. Once the data elements have been initialized for each relevant object, the script executes a final function to write out a standard string version of the input file. This capability is demonstrated in **Figure 17**, which shows a portion of the script used to define the input for the Godiva critical assembly.

The function MCInstance (shown in red) creates a skeletal version of the Monte Carlo input object known as myMC. Once it is created, the object myMC is filled out via one of two techniques. One method is direct initialization of data elements, such as for the problem control data. This simple method is typically used to initialize single-level input data structures, such as problem control data. The other method is execution of the create function (shown in blue). This function is used to initialize data elements within a given level of the Monte Carlo object via "keyword = value' assignment. As shown in Figure 17, this is a powerful method for initializing multiple levels of hierarchical data structures. The display function (shown in green) is used to write out the status of the specified portion of the Monte Carlo object. This useful tool permits the user to determine which data elements of the specified level have been initialized, and the value it is set to. Once each of the data elements within the Monte Carlo object has been initialized to the appropriate value, the getString function (shown in

```
#!/usr/local/bin/python -i

import sys
sys.path.append('..')
from MCInstance import *

myMC = MCInstance()

# Problem Control Data
myMC.control.problem_type = "Dynamic_Alpha"
myMC.control.edit_verbosity = "Moderate"
myMC.control.total_nu_bar = True
myMC.control.energy_representation =
"Continuous_Energy"
myMC.control.verify_particle = True
myMC.control.population_control = "Weight_Windows"

# Material Data
myMC.material.mat.create("Uranium",
   iso = [create( "U234", za=92234, atom_fract=1.0250e-2,
              react_list="None" ),
          create( "U235", za=92235, atom_fract=9.3768e-1,
              react_list="None" ),
          create( "U238", za=92238, atom_fract=5.2070e-2,
              react_list="None" ), ] )
myMC.material.mat.create("Air",
   iso = [create( "N14", za= 7014, atom_fract=1.0,
              react_list="None" ) ] );

# Surface Data
myMC.geometry.surf.create( "Sphere_1",
   surf_type = create("Sphere", radius= 8.7407 ) )
myMC.geometry.surf.create( "Sphere_2",
   surf_type = create("Sphere", radius=12.0) )
# Cell Data
myMC.geometry.cell.create( "Godiva_Assembly",
   surf = "-Sphere_1",
   mat = [ create("Uranium",
temperature_electron=2.5e-5,
              density=18.74 ) ] );
myMC.geometry.cell.create( "Atmosphere",
   surf = "+Sphere_1 -Sphere_2",
   mat = [create( "Air", temperature_electron=2.5e-5,
              density=1.2e-3 ) ],
   bc  = [create("OuterSpherical",
          bc_type='Vacuum', surf='Sphere_3') ] )

# Particle Data
myMC.particle.display()
myMC.particle.part.create("Neutrons",
   particle_type = "Neutron",
   target_num_particles = 100000,
   allow_combing = True,
   allow_splitting = True,
   thermalization = "Thermal_Scatter",
   cross_section_data_file =
     "/usr/Mercury/data/endl99/mcf1.pdb",
   kinetic_energy_min = 1.0e-10 )

...

# Write out the standard string input file.
input_file = open( 'MyGodiva.inp', 'w' )
input_file.write(myMC.getString())
input_file.close()
```

Fig. 17 An example of the Python script input pre-parser.

pink) is executed to write out the input string to the specified file.

2. Python Wrapped Data Structures and Inline User Interface

While the previous section describes how Python is used to enter data *into* ***Mercury***, this section discusses how Python is used to get data *out of* ***Mercury***. Many of the ***Mercury*** data structures have been exposed to Python. This permits the user to easily query and retrieve the exact data they desire from ***Mercury***, using a programming language that is ripe for data analysis, visualization, and custom I/O.

From a user's perspective, the most useful data structures exposed to Python are associated with tally, source and variance reduction objects. A uniform user interface to these data structures has been implemented, enabling the user to access the data using human readable indices. This makes it clear what data the user is requesting, and hence, what data is being delivered. For example, consider an energy deposition tally, with a two-dimensional result space of particle categories and materials. Each tally object is accessed via its name, and a handle to the object is returned:

```
my_tal = mc.tally.tal['MyTally']
```

The user may then request the energy deposition by particles of type Neutron into the material Water as follows:

```
print my_tal[“Neutron”, “Water”]
```

This method employs a positional indexing scheme, where the indices are listed in the same order that the sets and bsets were listed in the input description of the tally. Alternatively, one can use named parameters to access the tally data. In this case, the order in which the parameters are listed does not matter:

```
print my_tal.getValue(Particle=”Neutron”,
                      Material=”Water”)
```

Source and variance reduction objects are accessed in a similar manner. The Python dictionaries that describe the tally, source and variance reduction objects are as follows:

```
mc.tally.tal
mc.source.src
mc.variance_reduction.vr
```

These are standard Python dictionaries, which can be accessed and iterated over as you would with any Python dictionary. To access objects by name, use the following syntax:

```
my_tal = mc.tally.tal[“MyTally”]
my_src = mc.source.src[“MySrc”]
my_vr  = mc.variance_reduction.vr[“MyVR”]
```

An extremely useful feature for ***Mercury*** developers, as well as advanced users, is the ability to register a Python function that will be executed at any specified particle event. ***Mercury*** has a list of standard tally events which may be linked to a Python function which is executed whenever the event occurs. The prototype of the Python function takes a

single argument, which is the particle object that is currently undergoing the event. A use case in which this feature has been extremely helpful is to examine, in detail, what happens to each particle that is sourced into the problem. The relevant Python function may be defined as follows:

```
def WatchSourceParticles(part):
    mc.control.debugging_identifier = part.identifier
...
mc.particle.register(WatchSourceParticles,
                     "Creation_External_Source")
```

This instructs ***Mercury*** to execute the Python function watchSourceParticle each time the particle undergoes the event Creation_External_Source. Whenever this function is executed, it sets a global *debugging identifier* to the unique identifier of the specified particle that was sourced into the problem. This triggers the code to print out to the terminal what is happening event-by-event as the particle tracks. Every facet crossing, collision, census, etc that the particle undergoes will be reported in detail. This permits a ***Mercury*** code developer to quickly diagnose what is happening to the source particles.

Another use case of Python-wrapped data structures is verification of combinatorial geometry (CG). A Python routine has been developed that returns the CG cell object which contains any physical Cartesian coordinate *(x,y,z)*. If one is certain that given coordinates, or set of coordinates, should be within a specific CG cell, it is possible to query those coordinates, ensuring that the code agrees that they are within the specified cell. Suppose one knew that a certain locus of points within a bounding box, defined by the limits $(x_{min}, y_{min}, z_{min})$ and $(x_{max}, y_{max}, z_{max})$, are entirely within a CG cell. One could use the following Python coding to verify that this box is indeed within the given CG cell:

```
my_cell = mc.geometry.cell[CellName]
for i in xrange(1000000):
    my_x = xmin + (xmax - xmin)*random()
    my_y = ymin + (ymax - ymin)*random()
    my_z = zmin + (zmax - zmin)*random()
    my_other_cell =
        mc.geometry.locateCoordinate(my_x,my_y,my_z)
    if (my_other_cell.cell != my_cell.cell) :
        print "Error: Specified point not within the cell!"
```

A similar bit of Python code could be used to calculate the volume of a complex CG cell. If one has knowledge of the bounding box of the CG cell, this may be achieved via use of a Monte Carlo rejection technique. In this case, the product of the fraction of points located within the desired cell and the bounding box volume converges to the volume of the CG cell.

A design goal of the inline Python user interface was that knowledge of the ***Mercury*** input syntax would be sufficient to permit access to data through the Python user interface. As a result, there is a one-to-one correspondence between the block hierarchical keywords used in the ***Mercury*** input file, and the Python variable names. Suppose one wants to access the energy bin boundaries of a tally via Python. The Python syntax is to obtain this data is as follows:

```
mc.tally.tal['MyTally'].bset['MyBset'].domain
```

whereas the corresponding input file syntax is:

```
tally
  tal MyTally
    …
    bset MyBset
      space Energy
      domain
        0 1 2 3 4 5
      end_domain
    end_bset
  end_tal
end_tally
```

The inline Python user interface has proven extremely useful for quickly and easily accessing the data out of the code, diagnosing problems, and verifying CG problem setups. The alternatives involve (a) writing custom C++ functionality to provide the user with the exact feature they want, (b) spending hours tracking particles in a debugger, or (c) spending hours checking a CG problem definition "by hand". Having the inline Python user interface eliminates all of this, while providing the flexibility to permit users and developers to solve future, unanticipated problems for which a solution was not already coded in C++.

Acknowledgment

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

References

1) Mercury Code Team, "Mercury Web Site", Lawrence Livermore National Laboratory, http://www.llnl.gov/mercury

2) R.J. Procassini, P. S. Brantley, S. A. Dawson, G. M. Greenman, M. S. McKinley and M. J. O'Brien, *Mercury User Guide: Version c.8*, UCRL-TM-204296, Revision 7, Lawrence Livermore National Laboratory (2010).

3) Python Code Team, "Python Programming Language - Official Website", Python Software Foundation, http://www.python.org

4) X-5 Monte Carlo Team, *MCNP — A General Monte Carlo N-Particle Transport Code, Version 5, Volume II: User's Guide*, LA-CP-03-0245, Los Alamos National Laboratory (2003).

5) Cog Code Team, *Cog User's Manual, Fifth Edition: A Multi-particle Monte CarloTransport Code*, M-221-1, Revision 4, Lawrence Livermore National Laboratory (2002).

6) R. Scardovelli and S. Zaleski, "Direct Numerical Simulation of Free-Surface and Interfacial Flow". *Annu. Rev. Fluid Mech.*, **31**, 567-603 (1999).

7) International Criticality Safety Benchmark Evaluation Project, *International Handbook of Evaluated Criticality Safety Benchmark Experiments*, NEA/NSC/DOC/(95)03, September 2009 Edition [CD-ROM], Nuclear Energy Agency (2009).

8) T. J. Urbatsch, Los Alamos National Laboratory, private communication (2008).

9) M.R. Collette, I. R. Corey and J. Johnson, *High Performance Tools & Technologies*, UCRL-TR-209289, Lawrence Livermore National Laboratory (2004).

10) R. J. Procassini, J. M. Taylor, M. S. McKinley, G. M. Greenman, D. E. Cullen, M. J. O'Brien, B. R. Beck and C. A,. Hagmann, "Update on the Development and Validation of Mercury: A Modern, Monte Carlo Particle Transport Code", *Proceedings, International Topical Meeting on Mathematics and Computations*, 12 - 15 September 2005, Avignon, France (2005). [CD-ROM]