

Scripting NISTMonte with Jython

Nicholas W. M. Ritchie
28-Feb-2006

The Jython scripting language is an effective way to use the NISTMonte library to perform Monte Carlo simulations. Jython is a Java* implementation of the Python scripting language. The syntax of Jython is similar to the syntax of Python but, unlike Python, Jython has complete access to libraries compiled as Java byte-code. Jython is freely available at <http://www.jython.org>. A simple GUI based scripting environment (JythonGUI) is included with the NISTMonte package.

Jython isn't the only way to make use of NISTMonte. NISTMonte can be called from within a Java application, library or applet like any Java library however writing and compiling a new Java application for each new simulation quickly becomes tedious. Jython provides a more responsive environment in which scripts can be readily written, modified and executed.

This document is an introduction to Jython with a focus on scripting NISTMonte. It takes a pragmatic approach. Rather than attempt to teach the language and then teach how to apply the language to NISTMonte, this document dives right in. It doesn't spend much time discussing the syntax of Jython – this information is available in the resources listed below. Instead, it demonstrates through examples how Jython can access NISTMonte and how you can configure NISTMonte to perform sophisticated Monte Carlo simulations.

Both the Windows Installer version of NISTMonte and the ZIP file version come with a set of example scripts. These scripts are identified by an extension “.py”. You will probably want to acquaint yourself with them. Not only are they good examples of NISTMonte but they can also serve as models on which to base your own scripts.

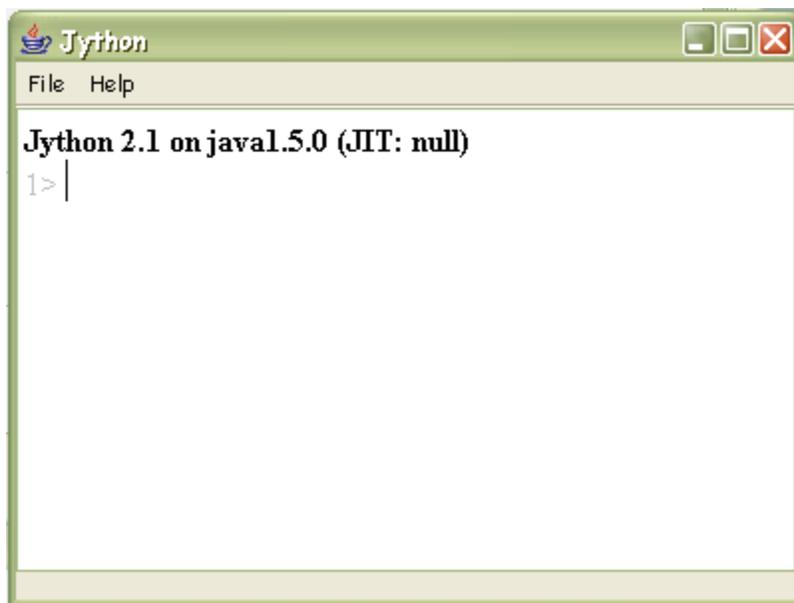


Figure 1: The Jython command line.

* Disclaimer: Certain commercial equipment, instruments, or materials are identified in this paper to foster understanding. Such identification does not imply recommendation or endorsement by the National Institute of Standards and Technology, nor does it imply that the materials or equipment identified are necessarily the best available for the purpose.

Jython is a scripting language. This means you can enter command directly from the command line. The command will be executed and the result displayed as soon as you type <enter>. Alternatively, you can load scripts from a text file. Generally, the second mode of operation is a better match for the NISTMonte library. In either case, the syntax is identical. Load scripts using the File – Open menu item.

Resources:

- 1) A Jython language tutorial is freely available (upon registration) at http://www-106.ibm.com/developerworks/java/edu/j-dw-java-jython1-i.html?S_TACT=104AHW02
- 2) Book - *Jython Essentials* (Samuele Pedroni and Noel Rappin, O'Reilly, March 2002) - Recommended!

What is Jython?

Jython is an implementation of the high-level, dynamic, object-oriented language [Python](#) seamlessly integrated with the [Java](#) platform. The predecessor to Jython, JPython, is certified as [100% Pure Java](#). Jython is freely available for both commercial and non-commercial use and is distributed with source code. Jython is complementary to Java and is especially suited for the following tasks:

- **Embedded scripting** - Java programmers can add the Jython libraries to their system to allow end users to write simple or complicated scripts that add functionality to the application.
- **Interactive experimentation** - Jython provides an interactive interpreter that can be used to interact with Java packages or with running Java applications. This allows programmers to experiment and debug any Java system using Jython.
- **Rapid application development** - Python programs are typically 2-10X shorter than the equivalent Java program. This translates directly to increased programmer productivity. The seamless interaction between Python and Java allows developers to freely mix the two languages both during development and in shipping products.

There are numerous [alternative languages](#) implemented for the Java VM. The following features help to separate Jython from the rest:

- **Dynamic compilation to Java bytecodes** - leads to highest possible performance without sacrificing interactivity.
- **Ability to extend existing Java classes in Jython** - allows effective use of abstract classes.
- **Optional static compilation** - allows creation of applets, servlets, beans, ...
- **Bean Properties** - make use of Java packages much easier.
- **[Python Language](#)** - combines remarkable power with very clear syntax. It also supports a full object-oriented programming model which makes it a natural fit for Java's OO design.

Figure 2: The description of the Jython language from <http://www.jython.org/docs/whatis.html>

Writing a script

All NISTMonte scripts have some common elements. These elements are introduced in the following sections.

Importing external libraries

The scripting environment has a small amount of basic functionality built right in. To extend these capabilities you load external libraries in the form of JAR files. The NISTMonte library (part of epq.jar) is loaded using the statement:

```
import gov.nist.microanalysis.NISTMonte as nm
```

The “as nm” syntax provides an alias that will be used extensively to refer to items within the gov.nist.microanalysis.NISTMonte library. Notice the line does not need to be terminated by any special character (except a carriage return.)

The import statement will usually be the first line in your Jython script files. It may also show up elsewhere if other libraries are needed. There are other common libraries that you might want to load. These are imported by adding the lines:

```
import gov.nist.microanalysis.EPQLibrary as epq
import gov.nist.microanalysis.EPQTools as ept
import java.io as jio
import java.util as jutil
```

Monte Carlo specific code is located in NISTMonte. General microanalysis type code is located in EPQLibrary. EPQTools contains non-algorithmic helper functions. The other libraries are standard Java libraries.

Building the sample

A sample is defined in terms of one or more Region objects. A Region object is defined by a Material object and a Shape object. The Material object describes the composition and density of the Region and the Shape object describes the volume inside the Region. Multiple Region objects may be combined to form complex samples.

Constructing a simple material

```
m1=epq.Material()
m1.defineByMoleFraction([epq.Element.Si, epq.Element.O], [1.0, 2.0])
m1.setDensity(epq.ToSI.gPerCC(2.65))
m1.setName("Silicon dioxide")
```

The first line creates a new variable called ‘m1’ that is a Material. The name ‘m1’ will be used in subsequent lines to refer to this particular Material variable. The syntax is often ‘m1.someMethod’ where the ‘.’ indicates that the method operates on the specified variable. So ‘m1.setDensity(...)’ sets the density of the Material referred to as ‘m1’.

The second line defines the constituents of the material as Silicon and Oxygen in ratio one atom of Si for each two atoms of O. The elements are identified by their

abbreviation so lead is `epq.Element.Pb` and other elements are likewise identified. You can add elements to the list but the number of elements in the first list (bracketed by '[' and ']') and the number of elements in the second list must match. There is no upper limit to the number of elements in a material.

The third line defines the density of the Material. The NISTMonte library uses exclusively SI units. However, SI units are not always the most friendly. The class `ToSI` (and its mirror `FromSI`) provides a handful of methods to facilitate converting from natural units into SI. In this case, `epq.ToSI.gPerCC` converts 2.65 g/cc into 2650 kg/m³ before passing the quantity to the `setDensity` function.

The fourth line simply gives the Material a name.

The following lines define a second Material – calcium carbonate. The second material is given a distinct name 'm2' to differentiate it from 'm1' and to allow both definitions to remain in memory simultaneously.

```
m2= epq.Material()
m2.defineByMoleFraction([epq.Element.Ca, epq.Element.C,↓
epq.Element.O],[1.0, 1.0, 3.0])
m2.setDensity(epq.ToSI.gPerCC(2.7));
m2.setName("Calcium carbonate");
```

The ↓ character indicates that the line should continue without a line break in the script file. (Don't enter the ↓ character.) The limits of the written page have forced a line break on this page.

Defining Shapes

Various different types of Shape objects are provided with the library including spheres, blocks, shapes bounded by planes. These basic shapes may be combined and differenced to create more complex shapes.

```
s1=nm.Sphere([0,0,-1.0e-6],1.0e-6)
subs = nm.MultiPlaneShape.createSubstrate([0.0, 0.0, -1.0],[0.0,↓
0.0, 0.0])
```

The first line creates a Sphere object assigned to the variable 's1'. The center of the sphere is at the point $x=0.0$, $y=0.0$, $z=-1.0e-6$ and the radius of the sphere is $1.0e-6$. Again, the units are SI so lengths are in meters. The default initial beam trajectory is along the z-axis from negative z to positive z. The natural place to center the sample is the origin $[0.0,0.0,0.0]$. In this case, the sphere is located slightly above the origin.

The second line creates a substrate by defining a plane. The plane is defined by a surface normal and a point on the plane. Planes can be used to define a volume. Everything on one side of the plane is inside and everything on the other side is outside. The surface normal defines the inside and outside. The surface normal always points outside. So in this case, the substrate is defined as everything with z greater than 0.0.

All locations and directions are defined using three-dimensional lists of floating point numbers. As above, the list is bracketed by '[' and ']' and list items are separated by commas.

Here are some other examples of basic shapes.

```
film = nm.MultiPlaneShape.createFilm([0.0, 0.0, -1.0],[0.0, 0.0, ↓ 0.0],
1.0e-6)
sqrt2= Math.sqrt(2.0)
tilted = nm.MultiPlaneShape.createFilm([sqrt2, 0.0, -sqrt2], ↓ [0.0,
0.0, 0.0], 1.0e-6)
dims = [0.8e-6, 0.6e-6, 0.5e-6]
point = [0.0, 0.0, 0.0]
block = nm.MultiPlaneShape.createBlock(dims, point, 0.0, 0.0, ↓ 0.0)
tiltedBl = nm.MultiPlaneShape.createBlock(dims, point, Math.PI / ↓ 4.0,
Math.PI / 4.0, Math.PI / 4.0)
```

Documentation

This is a good time to mention the documentation. The NISTMonte library has user oriented HTML documentation. The documentation was written using Java-style syntax but also applies to Jython. As an example, the HTML documentation for the MultiPlaneShape class looks like...

gov.nist.microanalysis.NISTMonte

Class MultiPlaneShape

java.lang.Object

gov.nist.microanalysis.NISTMonte.MultiPlaneShape

All Implemented Interfaces:

[MonteCarloSS.Shape](#)

public class **MultiPlaneShape**

extends java.lang.Object

implements [MonteCarloSS.Shape](#)

MultiPlaneShape implements simple or more complex shapes as the region bounded by a series of planes. The planes are defined by a normal pointing to the outside of the body (imagine a porcupine) and a point on the plane. A point is determined to be inside the MultiPlane object if the point is on the inside (the side away from the direction of the surface normal) of each plane.

Copyright: Not subject to copyright - 2004

Company: National Institute of Standards and Technology

And the documentation for the MultiPlaneShape.createBlock method looks like...

createBlock

```
public static MultiPlaneShape createBlock(double[] dims,  
                                           double[] point,  
                                           double phi,  
                                           double theta,  
                                           double psi)
```

Create a block of dimensions specified in dims, centered at point then rotated by the Euler angles phi, theta, psi. The rotation is a rotation phi around the z-axis, followed by a rotation theta around the y-axis and finally a rotation psi around the z-axis.

Parameters:

dims - double[] - The unrotated dimensions (x,y,z axis)
point - double[] - The location of the center of the block
phi - double - rotation about the z-axis (radians)
theta - double - rotation about the y-axis (radians)
psi - double - rotation about the x-axis (radians)

Returns:

MultiPlaneShape

The documentation is provided in the NISTMonte package in the directory 'doc'. The documentation is navigable through the 'index.html' file. The documentation is the second best resource for learning the details of the library – the ultimate reference is to ask the author.

Before leaving Shape classes, it is worth mentioning two special types of compound shapes.

ShapeDifference(shape1, shape2) – The Shape that results from removing the volume that shape1 and shape2 have in common from shape1.

SumShape([shape1, shape2, ..., shapen]) – The Shape resulting from the sum of the volumes in shape1, shape2, ..., shapen. SumShape is most useful for combining Shapes that overlap into a single Shape.

Combining Materials and Shapes into Regions

A Region is defined by a Shape and a Material. Regions are implemented as follows...

```
r1=monte.addSubRegion(monte.getChamber(), m1, s1)  
r2=monte.addSubRegion(monte.getChamber(), m2, s2)
```

All Region objects (except the Region returned by `monte.getChamber()`) are contained within other Region objects. The `monte.getChamber()` Region is special. It is predefined and consists of a 10 cm sphere filled with nothing (perfect vacuum).

The method `addSubRegion` is a member of the `MonteCarloSS` class and takes as arguments a Region, a Material and a Shape. The Material and Shape are combined into a Region and the Region is placed inside the argument Region (`monte.getChamber()` is this case). The new region is returned and in this case assigned to `r1` and `r2`.

The new Region (`r1` or `r2`) may then be used as the parent for a subsequent sub-Region.

The relationship between Regions is subtle and worthy of a little extra explanation. Regions are organized in an ancestral relationship. Parent Region objects may contain child Region objects. Child Region objects may act as parent Region objects for their own child Region objects. The key work here is *contain*. Parent Region objects must *fully contain* all child Region objects. In addition, child Region objects *must not overlap*.

These rules are sufficiently important that it is worth restating them.

1. Parent Region objects must fully contain all child Region objects.
2. Child Region objects must not overlap other child Region objects.

Adherence to these rules is not verified by `NISTMonte`. If they are not obeyed, the resulting simulations will not perform as you intend.

The ancestor of all user defined Regions is the Region returned by `monte.getChamber()`. As mentioned earlier, this Region is a 10 cm sphere filled with vacuum. You should always add at least one Region using `monte.getChamber()` as the parent. Think of this as being equivalent to placing your sample in the chamber.

What does it mean for one Region to contain another? When an electron is making a single step within a Region, there are three possible alternative end points – the electron may remain entirely within the current Region; the electron may reach the edge of the current region and exit to the parent Region; or the electron may intersect a child Region and thereby exit the current Region. When an electron exits the current Region, the trajectory of the electron is stopped at the border of the Region and restarted in the new Region. An electron is considered to be within only one Region at a time – the Region furthest down the ancestral tree (most child-like) that contains the current electron position. The Region in which the electron is located fully defines the Material with which the electron is interacting. It is as though the parent Region is hollowed out in the volume in which a child Region is located and filled with the child's Material.

```
r3=monte.addSubRegion(r1,m2, nm. Sphere([0.0,0.0,-1.0e-6],-0.5e-6)
```

This statement adds a sub-Region to the recently defined `r1` that is a 0.5 μm sphere of material `m2`. The net result is the ancestral relationship in which `r3` is a child of `r1` which is a child of `monte.getChamber()`.

You must use care if two sub-Regions overlap. If they are of the same material, you ought to create a `SumShape` to represent the two Shapes as one. If they are of

different Materials then there is an ambiguity – a volume defined as two different Materials. In this case, it is best to use DifferenceShape to subtract the volume in one shape from the other to clarify which Material is desired.

There is no limit to the number of generations of children.

Configuring the simulation

So, now that you have defined the sample, you may define the instrument conditions.

```
monte.setElectronGun(epq.GaussianBeam(1.0e-9))
monte.setBeamEnergy(epq.ToSI.keV(25.0))
```

By default, the electron source is defined to be a Gaussian beam with width of 10 nm. The first line changes this to a Gaussian beam of 1 nm width. The second line sets the beam energy. Again, the native units for energy are Joules so use epq.ToSI.keV(...) to convert keV to Joules.

Adding detectors

To accumulate statistics it is necessary to add various different types of observers to the model.

This can be a little confusing because by default all the model does it track electrons through a user defined set of Regions. X-ray generation is not considered unless an instance of the XRayEventListener is attached to the model.

```
xrel=nm.XRayEventListener(monte)
monte.addActionListener(xrel)
```

Now x-rays are generated but no statistics are accumulated. To accumulate statistics you need to determine what type of statistics to accumulate.

```
# add phi-rho-z stats
przs=nm.PhiRhoStats.watchDefaultTransitions(xrel,-1.0e-6,4.0e-6)
# add generation images
imgs=nm.EmissionImage.watchDefaultTransitions(xrel,512,5.0e-6)
```

These two expressions add two different mechanisms to visualize the x-ray generation. The first creates a set of accumulators to collect phi-rho-z curves for the dominant line in each family for each element in the sample. The second creates images that represent the spatial dependence of emission from the sample for the dominant line in each family for each element in the sample.

In more detail:

```
przs=nm.PhiRhoStats.watchDefaultTransitions(xrel,-1.0e-6,4.0e-6)
```

`xrel` – is the instance of `XRayEventListener` created above
-1.0e-6 and 4.0e-6 represent the z-range that is divided into bin, accumulated and tabulated.

`przs` is a variable that following the call will contain a list of `PhiRhoStats` objects one for each observed transition.

```
imgs=nm.EmissionImage.watchDefaultTransitions(xrel,512,5.0e-6)
```

`xrel` – is the instance of `XrayEventListener` created above
512 is the image dimension (512 x 512)
5.0e-6 sets the scale of the viewport.

`imgs` is a variable that following the call will contain a list of `EmissionImage` objects one for each observed transition.

The view port is of height and width equal to the scale. The image is centered horizontally on the origin but vertically the view port is 10% above the origin and 90% below. Thus, it is best to create samples horizontally centered on the origin but located below the origin in the positive Z direction.

If this default view port does not work for you, it is possible to change the view port for each image independently.

```
# rescale the images
for i in imgs:
    i.setXRange(-1e-6,1e-6);
    i.setYRange(-1e-6,1e-6);
```

Here the y range refers to the vertical axis in the image that represents the z-axis in the model.

Note the syntax for loops. As previously stated `imgs` is a list of `EmissionImage` objects. The for loop selects each one in turn and calls it 'i'. The extent of the for loop is defined by indentation - in this case two lines. These lines are repeated for each `EmissionImage` object in `imgs`.

Adding non-x-ray related statistical accumulators

Other non-x-ray related accumulators are available. They attach directly to the `MonteCarloSS` instance.

To collect statistics on how many collisions occur before an electrons energy drops below the excitation energy for a specific shell add the following code...

```
ss = nm.ScatterStats(epq.AtomicShell(epq.Element.Fe,↓
epq.AtomicShell.K))
monte.addEventListener(ss)
```

To collect backscatter statistics add the following code...

```
bs= nm.BackscatterStats(monte)
monte.addEventListener(bs)
```

To collect images of the first 100 electron trajectories... (The number 100 can be modified.)

```
ti=nm.TrajectoryImage(512,512,5e-6)
monte.addEventListenera(ti)
```

5e-6 sets the scale much like for EmissionImage

Run the simulation

Now that the simulation is configured and the quantities of interest observed we can run the simulation.

```
monte.runMutipleTrajectories(1000)
```

This line runs 1000 electrons through a full trajectory down to an energy of 50 eV.

Saving the results

After a little while the simulation will complete the desired number of trajectories. All that remains is to save the results.

```
dest="C:\\Documents and ↓
Settings\\nritchie\\Desktop\\inclusion\\PbBaSb\\"+nTraj.toString()+"\\"
# output the phi-rho-z stats
nm.PhiRhoStats.dumpToFiles(przs,dest)
# output the trajectory image
img.dumpToFile(dest)
# output the transition image
nm.EmissionImage.dumpToFiles(imgs,dest)
# output the scattering statistics
import java.io as jio
ss.dump(jio.FileOutputStream(dest+"scattering.prn")
# output the backscatter statistics
bs.dump(jio.FileOutputStream(dest+"backscatter.prn")
# output the trajectory image as a PNG file
ti.dumpToFile(dest+"traj.png")
```

Conclusion

That is all there it to it. Use the example scripts. With these, there is rarely a reason to create a new script from scratch. Instead, find the script that is closest to what you want to do, rename it, and modify it.

Using the Jython scripting language, it is (relatively) easy to set up scripts to run multiple simulations while varying any of the simulation parameters. A couple of the scripts show examples of this. You can raster the beam; you can change composition; you can change sample scale; you can change beam energy.

Example 1

```
# load the necessary libraries
import gov.nist.microanalysis.EPQLibrary as epq
print "Starting...";
# create an instance of the model
monte=epq.MonteCarloSS()
monte.setBeamEnergy(epq.ToSI.keV(25.0))
# set the scale
r=0.25e-6
imgSize=512
nTraj=10000
print "Radius = "+r.toString()
c=epq.MaterialFactory.createPureElement(epq.Element.C)
partC=monte.addSubRegion(monte.getChamber(),c, nm.Sphere([0.0,0.0,-r],r))
# create a lead particle
partS = nm.Sphere([0.0,0.0,-r],0.95*r)
pb=epq.MaterialFactory.createPureElement(epq.Element.Pb)
part=monte.addSubRegion(partC,pb, partS)
# create Ba-Sb Material
basb=epq.Material()
basb.defineByMoleFraction([epq.Element.Ba,epq.Element.Sb],[1.0,1.0])
basb.setDensity(epq.ToSI.gPerCC(16.0))
basb.setName("Ba-Sb")
# combine the shape and material into a region and place it in the
chamber
inc0S= nm.Sphere([r/4,0,-r-r/4],r/4)
inc1S= nm.Sphere([-r/4,0,-r-r/4],r/4)
inc2S= nm.Sphere([0,0,-r+r/4],r/4)
# combine the shape and material into a region and place it in the
chamber
monte.addSubRegion(part,basb, inc0S)
monte.addSubRegion(part,basb, inc1S)
monte.addSubRegion(part,basb, inc2S)
# create a C substrate
subs = nm.MultiPlaneShape.createSubstrate([0.0, 0.0, -1.0],[0.0, 0.0,
0.0])
# combine the shape and material into a region and place it in the
chamber
monte.addSubRegion(monte.getChamber(), c, subs)
# add an x-ray event listener
xrel= nm.XRayEventListener(monte)
monte.addActionListener(xrel)
przs= nm.PhiRhoStats.watchDefaultTransitions(xrel,-1.0e-6,4.0e-6)
# add a trajectory image
img= nm.TrajectoryImage(imgSize,imgSize,5.0e-6)
img.setXRange(-1e-6,1e-6);
img.setYRange(-1e-6,1e-6);
monte.addActionListener(img)
# add generation images
imgs= nm.EmissionImage.watchDefaultTransitions(xrel,imgSize,5.0e-6)
# rescale the images
for i in imgs:
    i.setXRange(-1e-6,1e-6);
    i.setYRange(-1e-6,1e-6);
```

```
# run the simulation
monte.runMutipleTrajectories(nTraj)
# determine where to save the results
dest="C:\\Documents and
Settings\\nritchie\\Desktop\\inclusion\\PbBaSb\\"+nTraj.toString()+"\\"
# output the phi-rho-z stats
nm.PhiRhoStats.dumpToFiles(przs,dest)
# output the trajectory image
img.dumpToFile(dest)
# output the transition image
nm.Image.dumpToFiles(imgs,dest)
print "Done!"
```

Example 2 – An example of a loop modifying the sample scale

```
import gov.nist.microanalysis.EPQLibrary as epq
# create the material, shape and region
mat=epq.MaterialFactory.createMaterial(epq.MaterialFactory.K3189)
radii=[1.0e-7,2.0e-7,4.0e-7,8.0e-7,1.6e-6]
for r in radii:
    monte=nm.MonteCarloSS()
    shape = nm.Sphere([0.0,0.0,-1.01*r],r)
    epq.MonteCarloSS.Region(monte, monte.getChamber(), mat, shape)

    mat=epq.MaterialFactory.createPureElement(epq.Element.C)
    shape = nm.MultiPlaneShape.createSubstrate([0.0, 0.0, -1.0],[0.0,
0.0, 0.0])
    epq.MonteCarloSS.Region(monte, monte.getChamber(), mat, shape)

    # add event listeners
    xrel= nm.XRayEventListener(monte)
    monte.addActionListener(xrel)
    przs= nm.PhiRhoStats.watchDefaultTransitions(xrel,-2.0e-6,8.0e-6)
    # add a trajectory image
    img= nm.TrajectoryImage(512,512,10.0e-6)
    monte.addActionListener(img)
    # run the simulation
    monte.runMultipleTrajectories(1000)
    # determine where to save the results
    dest="c:\\temp\\crap\\K3189\\particles\\radius=1 micron\\"
    # output the phi-rho-z stats
    nm.PhiRhoStats.dumpToFiles(przs,dest)
    # output the trajectory image
    img.dumpToFile(dest)
print "Done"
```

Example 3 – Another example of a loop modifying the sample scale

```
import gov.nist.microanalysis.EPQLibrary as epq
# create an instance of the model
print "Start...";
for i in range(1,11):
    monte=epq.MonteCarloSS()
    monte.setBeamEnergy(epq.ToSI.keV(25.0))
    # create the material, shape and region
    mat=epq.Material()
    mat.defineByMoleFraction([epq.Element.Mn,epq.Element.S],[1.0,1.0]
)
    mat.setDensity(epq.ToSI.gPerCC(4.0))
    mat.setName("MnS")
    r=1.0e-6+i*0.1e-6
    print "Radius = "+r.toString()
    # create a hemispherical region
    sphere = epq.MCSS_Sphere([0.0,0.0,0.0],r)
    plane = epq.MCSS_MultiPlaneShape.createSubstrate([0.0, 0.0,
1.0],[0.0, 0.0, 0.0])
    epq.MonteCarloSS.Region(monte, monte.getChamber(), mat,
epq.MCSS_ShapeDifference(sphere, plane))
    # create a hole in the substrate into which to place the
hemisphere
    mat=epq.MaterialFactory.createPureElement(epq.Element.Fe)
    sphere = epq.MCSS_Sphere([0.0,0.0,0.0],1.001*r)
    subs = epq.MCSS_MultiPlaneShape.createSubstrate([0.0, 0.0, -
1.0],[0.0, 0.0, 0.0])
    epq.MonteCarloSS.Region(monte, monte.getChamber(), mat,
epq.MCSS_ShapeDifference(subs, sphere))
    # add event listeners
    xrel=epq.MCSS_XRayEventListener(monte)
    monte.addActionListener(xrel)
    przs=epq.MCSS_PhiRhoStats.watchDefaultTransitions(xrel,-1.0e-
6,4.0e-6)
    # add a trajectory image
    img=epq.MCSS_TrajectoryImage(2048,2048,5.0e-6)
    monte.addActionListener(img)
    # add generation images
    imgs=epq.MCSS_Image.watchDefaultTransitions(xrel,256,5.0e-6)
    # run the simulation
    monte.runMutipleTrajectories(1000)
    # determine where to save the results
    dest="e:\\K3189\\inclusions\\MnS\\radius="+r.toString()+"\\"
    # output the phi-rho-z stats
    epq.MCSS_PhiRhoStats.dumpToFiles(przs,dest)
    # output the trajectory image
    img.dumpToFile(dest)
    # output the transition image
    epq.MCSS_Image.dumpToFiles(imgs,dest)
print "Done!"
```