To my parents, who gave me everything I needed to become the person I am today
ACKNOWLEDGMENTS

I would like to thank Dr. Kelly Jordan for encouraging me to learn Python and for believing in me when I had trouble believing in myself. I thank Dr. Sanjiv Samant for always being there to push me and keep me moving forward when I needed it most. A sincere thank you to Dr. James Baciak for keeping the nuclear engineering program strong and for always looking out for your students. I’m grateful for having the chance to get to know all of you.

I would also like to thank my parents who have always supported me with unconditional love and instilled in me the drive and passion that led me to where I am now; I’m doing my best to carry it ever farther and I do it for both of you as much as for myself.

Last but certainly not least I want to thank Lauren and all my close friends, old and new, who bring so much joy into my life. You offered the greatest respite from school and I can’t imagine my life without the many amazing memories I’ve made with all of you.

This material is based upon work supported under an Integrated University Program Graduate Fellowship as well as the University of Florida
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACKNOWLEDGMENTS</td>
<td>4</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td>7</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>8</td>
</tr>
<tr>
<td>ABSTRACT</td>
<td>10</td>
</tr>
</tbody>
</table>

## CHAPTER

1. **INTRODUCTION AND BACKGROUND**                                      | 12   |
   1.1 Review of Photon Physics                                           | 13   |
   1.1.1 Compton Scattering                                              | 13   |
   1.1.2 Signal of a Compton Scatter Imaging System                     | 14   |
   1.2 Review of Compton Scatter Imaging Systems                        | 16   |
   1.2.1 System Design Principles                                        | 16   |
   1.2.2 Medical Applications                                            | 18   |
   1.2.3 Industrial Applications                                         | 20   |

2. **MCNP6 VALIDATION FOR BACKSCATTER IMAGING**                         | 26   |
   2.1 Motivation                                                        | 26   |
   2.2 Methods for Pencil-Beam XBI in MCNP6                              | 27   |
   2.3 Test Cases                                                        | 29   |
   2.4 Results                                                           | 30   |

3. **PARAMETRIC STUDY AND PROOF OF CONCEPT FOR XBI TOMOSYNTHESIS USING SPECTRAL DETECTION** | 37   |
   3.1 Overview                                                          | 37   |
   3.2 Methods                                                           | 38   |
   3.2.1 Imaging System                                                 | 38   |
   3.2.2 Test Phantoms                                                   | 39   |
   3.2.3 Determining Image Depth                                         | 40   |
   3.3 Results and Analysis                                              | 41   |
   3.3.1 Parametric Study                                               | 41   |
   3.3.2 Tomosynthesis Proof of Concept                                 | 43   |
   3.4 Conclusions                                                       | 44   |

4. **2D BACKSCATTER IMAGING: PYTHON TOOLKIT**                           | 54   |
   4.1 Motivation                                                        | 54   |
   4.2 Code Description                                                  | 55   |
   4.2.1 Backscatter System                                              | 55   |
   4.2.1.1 X-ray fan-beam                                                | 56   |
<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4-1</td>
<td>List of variables required from user for XRayFanBeam class.</td>
<td>64</td>
</tr>
<tr>
<td>4-2</td>
<td>List of variables required from user for Detector class.</td>
<td>64</td>
</tr>
<tr>
<td>4-3</td>
<td>List of variables required from user for Collimator class (optimized collimation grid).</td>
<td>65</td>
</tr>
<tr>
<td>4-4</td>
<td>List of variables required from user for UniformCollimator class (uniform collimation grid).</td>
<td>65</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-1</td>
<td>Mass attenuation coefficients in water for a range of photon energies.</td>
<td>22</td>
</tr>
<tr>
<td>1-2</td>
<td>Dependence of Klein-Nishina cross section on scatter angle for initial photon energies.</td>
<td>23</td>
</tr>
<tr>
<td>1-3</td>
<td>Schematic illustration of point-by-point arrangement for a Compton scatter imaging system.</td>
<td>24</td>
</tr>
<tr>
<td>1-4</td>
<td>Schematic illustration of plane-by-plane arrangement for a Compton scatter imaging system.</td>
<td>25</td>
</tr>
<tr>
<td>1-5</td>
<td>Schematic of a Radiography by Selective Detection setup containing a subsurface feature with a collimated detector.</td>
<td>25</td>
</tr>
<tr>
<td>2-1</td>
<td>Photos of the RSD scanning system at UF. Pencil-beam collimator surrounded by four detectors.</td>
<td>32</td>
</tr>
<tr>
<td>2-2</td>
<td>Experimental setup used to make X-ray scatter images by Towe and Jacobs.</td>
<td>32</td>
</tr>
<tr>
<td>2-3</td>
<td>Composite test object used to demonstrate three-dimensional imaging properties of the backscatter technique.</td>
<td>33</td>
</tr>
<tr>
<td>2-4</td>
<td>Experimental image of the aluminum calibration plate used in RSD imaging.</td>
<td>33</td>
</tr>
<tr>
<td>2-5</td>
<td>MCNP6 simulated image of the aluminum calibration plate used in RSD imaging.</td>
<td>34</td>
</tr>
<tr>
<td>2-6</td>
<td>Backscatter radiographs of the composite test object.</td>
<td>35</td>
</tr>
<tr>
<td>2-7</td>
<td>MCNP6 simulated backscatter radiographs of the composite test object.</td>
<td>36</td>
</tr>
<tr>
<td>3-1</td>
<td>Cross-sectional views of simulation geometry in MCNP6.</td>
<td>46</td>
</tr>
<tr>
<td>3-2</td>
<td>Cross-sectional views in the XY dimension of the tomosynthesis test phantom in MCNP6.</td>
<td>47</td>
</tr>
<tr>
<td>3-3</td>
<td>Depth distinguishability using image of tissue/bone phantom with 50 keV photons.</td>
<td>48</td>
</tr>
<tr>
<td>3-4</td>
<td>Raw images of soft tissue phantom with cylinder of air.</td>
<td>49</td>
</tr>
<tr>
<td>3-5</td>
<td>Image penetration depth versus X-ray interrogation energy.</td>
<td>50</td>
</tr>
<tr>
<td>3-6</td>
<td>Normalized pixel intensity vs pixel location along the middle row of tissue/bone phantom image.</td>
<td>51</td>
</tr>
<tr>
<td>3-7</td>
<td>Images acquired in MCNP6 using spectral detection in backscatter imaging system.</td>
<td>52</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
<td></td>
</tr>
<tr>
<td>---------</td>
<td>----------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>3-8</td>
<td>Comparison of normalized X-ray energy spectra.</td>
<td></td>
</tr>
<tr>
<td>4-1</td>
<td>Visualization of 2D backscatter imaging system using a uniform collimation grid.</td>
<td></td>
</tr>
<tr>
<td>4-2</td>
<td>Cross-sectional views of test simulation geometry inside MCNP6.</td>
<td></td>
</tr>
<tr>
<td>4-3</td>
<td>Simulated backscatter image of air ring inside aluminum block.</td>
<td></td>
</tr>
<tr>
<td>4-4</td>
<td>Line plot of pixel intensities along middle row of 2D detector array for the simulated image.</td>
<td></td>
</tr>
<tr>
<td>5-1</td>
<td>Images acquired in MCNP6 of a linear scanning method for 2D backscatter system.</td>
<td></td>
</tr>
<tr>
<td>6-1</td>
<td>Illustration of ray tracing performed for each row in the detector array.</td>
<td></td>
</tr>
<tr>
<td>6-2</td>
<td>Illustration of image sampling disparities with increasing projection angles for the top detector row measuring near object surface.</td>
<td></td>
</tr>
<tr>
<td>6-3</td>
<td>Illustration of image sampling disparities with increasing projection angles for the bottom detector row measuring the deepest within object.</td>
<td></td>
</tr>
<tr>
<td>6-4</td>
<td>Raw reconstructed image of UF industrial phantom from MCNP6 simulations.</td>
<td></td>
</tr>
<tr>
<td>6-5</td>
<td>Filtered reconstructed image of UF industrial phantom from MCNP6 simulations.</td>
<td></td>
</tr>
<tr>
<td>6-6</td>
<td>Comparison of linear and rotational scanning methods.</td>
<td></td>
</tr>
</tbody>
</table>
X-ray backscatter systems create images of the internal structure of objects while both X-ray source and detectors are placed on the same side. One sided imaging is useful when the object being imaged is either too voluminous or access to its opposite side is impossible/impractical. However, X-ray backscatter designs typically acquire only 2D images and previous attempts to go beyond that involved multiple scans.

Using Monte Carlo radiation transport simulations and a developed Python toolkit, proof of concepts for two new techniques are tested for beyond 2D (tomosynthesis) imaging using X-ray backscatter systems. The first technique uses spectral detection, whereby binning the collected backscatter signal into separate energy groups creates depth sensitive images, due to the energy dependence of a photon’s mean free path in materials. This technique requires only a single scan of an object with an X-ray spectra, unlike its predecessor which required a separate scan for each depth image.

The second technique considers the capabilities of a 2D backscatter system, consisting of a fan-beam X-ray source with a collimated, 2D detector array. With each row in the detector array viewing a separate depth inside the object, depth sensitive information is collected during a scan. Both a linear and rotational scanning method are considered. A Python toolkit is developed that enables the rapid design and simulation of 2D backscatter systems, as well as the use of novel image reconstruction algorithms for the rotational
scanning method. The rotational scanning method produced higher quality images than
the linear scan, for shallow depths.

Both proposed techniques were proven to have the capability for single-sided
tomosynthesis imaging, which can be used for a range of non-destructive testing and
imaging applications. Due to the complex nature of backscatter signal acquisition, depth
resolution and certain system characteristics are best understood on a case-by-case basis,
depending on the materials present in the object being imaged and the materials used in
the imaging system.
CHAPTER 1
INTRODUCTION AND BACKGROUND

Compton scatter imaging (CSI) is a technique similar to that of transmission radiography except that it utilizes the inelastic scattering of photons to acquire spatial information about the object under investigation. Since photons scatter in all directions, it’s possible to place photon detectors in a CSI system at locations that would otherwise be impossible for a transmission imaging system to use; their detectors must be in line with the primary beam path and on the opposite of the object.

X-ray backscatter imaging (XBI) can be considered a subset of CSI, where X-rays are the photon source and the detectors are placed on the same side as the X-ray source. This constitutes a design that collects X-rays scattering at angles greater than 90° and only needs access to one side of the object it’s imaging. XBI has useful applications in non-destructive testing and the potential for use in medical imaging, where the object being imaged is either too voluminous or access to its opposite side is impossible/impractical. Although it is possible to achieve tomographic imaging with CSI designs, they almost always place detectors on the far side of the object from the photon source and sometimes use the additional information from transmission images. This means very little work has been achieved in XBI in regards to tomography, primarily because of how limiting signal acquisition is from only a single side.

The work presented here seeks to expand upon the current capabilities of XBI, by laying the foundation for new techniques which can produce depth sensitive information, enabling possibilities beyond the norm of 2D imaging. This is achieved through Monte Carlo, radiation transport simulations and computer programs for processing and analysis; providing proof of concepts for ideas that can now be prototyped and tested by a physical system.
1.1 Review of Photon Physics

There are three primary events that a photon can undergo when interacting with electrons bound to an atom: photoelectric absorption, scattering, and pair production. Figure 1-1 shows how the likelihood of these interactions (mass attenuation coefficients) change with increasing photon energy in water. Photoelectric absorption is when a photon is fully absorbed by the electron, transferring all of its energy. This can lead to the electrons being removed from tightly bound inner orbitals of the atom. When this occurs, fluorescent X-rays are emitted as a way for electrons to release excess energy when filling the holes left behind in the inner orbitals.

Photon scattering can be broken into two different types, coherent and incoherent. Coherent scattering is actually the absorption and re-emission of a photon by an atom, thereby changing the direction of the photon but not its energy. It occurs at much lower energies (10’s of keV and less). Incoherent scattering, also called Compton scattering, is when a photon scatters off of an electron, causing the photon to change in both direction and energy; the loss of energy having been transferred to the electron.

Pair production is the conversion of a photon into an electron and a positron (positively charged electron), which typically results in the two particles later annihilating into two photons. Since photons must have enough energy to create the mass of two electrons (2 x 511 keV), this interaction has a threshold energy of 1.022 MeV in order to occur. Since the known range of diagnostic imaging techniques do not use photons in the MeV range, this interaction is of little concern and will not be further discussed.

1.1.1 Compton Scattering

Compton scattering refers to the interaction of photons with those atomic electrons whose binding energies are much less than the energy transferred on scattering. Assuming stationary, unbound electrons, the energy, $E_f$, of the Compton radiation scattered through an angle $\theta$ is related to the initial energy $E_o$ through Comptons equation:
\[ E_f(\theta, E_o) = \frac{E_o}{1 + \frac{E_o}{m_e c^2}(1 - \cos \theta)} \]  

(1-1)

where \( m_e c^2 \) is the rest mass energy of the electron (511 keV). When \( E_o \) is much greater than \( m_e c^2 \), the limit to the amount of energy remaining in a photon after a full 180° scatter is \( m_e c^2 / 2 \).

The likelihood of a photon scattering at certain angles \( \theta \) is not entirely random and is dependent on the initial energy of the photon. This angular distribution of Compton scatter from free, stationary electrons is defined by the Klein-Nishina model:

\[ \frac{d\sigma_{KN}}{d\Omega}(\theta, E_o, E_f) = \frac{r_o^2}{2} \left[ \frac{E_f^2}{E_o^2} \right] \left[ \frac{E_f}{E_o} + \frac{E_o}{E_f} - \sin^2 \theta \right] \]  

(1-2)

which determines the differential scattering cross section and has units of \( m^2 sr^{-1} electron^{-1} \); where \( r_o \) is the classical electron radius (2.82 \( 10^{-15} \) m). This cross section can be thought of as the probability per unit solid angle that a photon of initial energy \( E_o \), passing normally through a layer of one electron m\(^{-2}\), will be scattered into a solid angle \( d\Omega \) at angle \( \theta \), resulting in a photon with a final energy \( E_f \) [1, 2].

The Klein-Nishina model visualized in Figure 1-2 shows that as initial photon energies \( (E_o) \) become larger, the probability for angles of scatter shift towards the forward direction. Conversely, as initial photon energy decreases, the probability for scattering angles becomes more distributed over all possible directions; the limit to this being the probability function that describes Thompson scattering, a form of coherent scattering.

1.1.2 Signal of a Compton Scatter Imaging System

The methods by which an image is obtained from a Compton scatter geometry are far more complicated than those for transmission radiography. As photons are emitted, they are first attenuated along the primary beam path determined by the source geometry. At this point, for transmission radiography, an image is generated on the opposite side of the object as the attenuated primary beam and scattered photons are collected on the incident beams exit side; however for CSI, a photon must also Compton scatter with an electron
inside the object, thereby changing its direction and traversing a secondary path that will guide it to the detector face that is located somewhere outside of the primary beam path. In this way, scatter signals are affected by 3 main factors: attenuation of the initial photon source; Compton scatter (within the solid angle subtended by the detector); and attenuation of the secondary, scattered photons as they travel back towards the detector.

The general equation for the signal acquired from single scattered photons from a small volume is as follows:

$$S = \rho \frac{1}{\Delta A} \int_{\Delta A} \int_{\Delta Z} e^{-\int \mu(E_0, Z)dZ} \int_\theta \frac{d\sigma}{d\Omega}(E_0, \theta) e^{-\int \mu(E, l)dl} d\Omega dAdZ \quad (1-3)$$

where $\rho$ is the electron density of the sensitive volume, $\Delta A$ is the beam cross-sectional area, $\Delta Z$ is the entry (primary) path length, $l$ is the scattered (secondary) path length, $\mu$ is the linear attenuation coefficient, and $\frac{d\sigma}{d\Omega}$ is the differential Klein-Nishina cross section [3].

The consideration of multiply scattered photons in a signal complicates the equation and can be ignored only if the mean free path of the photons are significantly larger than the physical dimensions of the features that make up the object being imaged [4]. The use of detector collimation is a common method to reduce the collection of multiply scattered photons but studies have been conducted to try and quantitatively measure the effects [3, 5–7].

Like in transmission radiography, the linear attenuation coefficients are both energy and spatially dependent, although CSI has 2 separate paths and energies that affect it. CSI is also affected by the detector shape and location because it affects the solid angle from which a signal must originate and upon which the Klein-Nishina cross section depends.
1.2 Review of Compton Scatter Imaging Systems

1.2.1 System Design Principles

Ultimately, every CSI system seeks to provide an image of an object by mapping the detected signal and providing image contrast due to changes in materials present inside said object. These images are either in two spatial dimensions or three. Some systems go a step further by also calculating the electron density of the materials [3, 8–10].

When designing a CSI system there are a few main factors to consider: image resolution; signal to noise ratio (SNR); acquisition time; and mechanical complexity. The primary focus and most limiting factors are the first three and are intertwined; usually the improvement of one leads to a degradation in one or more aspects. Mechanical complexity refers to not only the complexity of actually building the parts for it but also can affect acquisition time. Since mechanical systems move slower than photons, you’ll likely have longer acquisition times if you have to physically move the system to scan a location rather than having it already illuminated by the photon source. There’s also the risk of more moving parts creating more ways a system can break and require maintenance or calibration.

Over the years a plethora of CSI systems have been designed to accomplish tasks from flaw detection to densitometry; ranging from medical applications to industrial non-destructive testing (NDT). These numerous CSI designs come in many shapes and sizes and therefore can become quite confusing to keep track of and understand. Before reviewing the history of CSI in the medical field and industrial field, it will be helpful to first organize the many designs into a simple hierarchy.

When creating an image, the image dimensions are split between the scanning dimensions and the system dimensions. Scanning dimension refers the the dimensions that the system must have a physical moving part in order to collect a signal. System dimensions are the dimensions that signals are collected for a single scanning point. The
number of dimensions a system achieves is whatever is smallest: the number of dimensions of the photon source or the detector.

For photon sources described in this work, pencil-beams refer to 1D lines penetrating into the object. Fan-beams are the next step up, where a fan shaped beam is penetrating the object and illuminating a 2D space within. Anything beyond a fan-beam, such as isotropic sources or cone beams, refers to a 3D source. All of these source geometries are achieved using collimation.

For radiation detectors, the number of sensitive volumes used typically refers to the number of dimensions it can see. However it could also be used to aid in the reduction of moving parts, where the multiple detection volumes are a replacement for having to physically move one detector. If the detector is also energy sensitive, then that adds another dimension to which it can see, which will be discussed later.

Most CBI systems can be categorized into point-by-point (0D), line-by-line (1D), and plane-by-plane (2D) imaging. This is referring to the system dimensions as described earlier. An example of point-by-point imaging would be a single radiation detector with no energy discrimination capabilities combined with a pencil-beam (1D), as seen in Figure 1-3. This system must physically scan in one, two, or three dimensions in order to develop a 1D, 2D, or 3D image, respectively. An example of plane-by-plane imaging would be a pixelated 2D detector array (2D) used with a fan-beam source (2D), where two dimensions are collected for every scan point; so one scanning dimension is needed to develop a 3D image, or none at all for a 2D image. Figure 1-4 shows such a scheme.

A less common technique is energy-coded imaging, where energy discrimination is used in the detector. Each energy bin in the detector has an associated scatter angle and therefore location within the object being imaged, for a mono-energetic photon source. This technique is heavily reliant upon the energy resolution of the detector however and is very sensitive to multiple scattering events.
In regards to imaging resolution and SNR, point-by-point is superior to line-by-line and plane-by-plane. However it comes at the cost of acquisition time since it must be scanned in every dimension. Line-by-line and plane-by-plane have increasingly fast image acquisition times but require collimators to be placed on the detectors, which is not always mandatory for point-by-point methods. This is because photon scattering is coming from a large area, so the individual detector pixels must be collimated so that they don’t pick up scatter signals meant for neighboring detector pixels. Either way, the collimation can only help so much and it can be expected that these designs will have a reduced SNR and poorer resolution. Eventually if too much collimation is used for such designs, in the attempts to remedy these factors, then the acquisition time may be equivalent or worse than that of point-by-point systems, since so much of the scattered signal will be removed by collimation.

1.2.2 Medical Applications

The first known attempt at CSI as a medical application was by P. G. Lale in 1959 [11]. In his set up, Lale used a pencil-beam of gamma radiation that was detected by a heavily collimated detector placed on the opposite side of the object. The heavy collimation made it possible so that the detector could only see scattered gamma rays coming from a small volume (commonly referred to as a “voxel”) inside the object. This is a clear example of a point-by-point design. By moving/scanning the pencil-beam and detector over an area on the object, a 3D mapping of voxels could be produced for the object. Therefore, CSI had its first medical application as a tomographic device, a technique currently achieved by CT and Magnetic Resonance Imaging (MRI). As expected, this CSI design suffered from long acquisition times.

Over the years new systems would be developed for Compton scatter tomographic imaging but many earlier designs would share a few core principles. One is that gamma emitting radionuclides were a common radiation source[11–14]. This was due to the fact that it was a mono-energetic source in the low MeV range. This made Compton scattering
the dominant interaction and also made analytical models much more reasonable to solve. It also allowed for energy discrimination to be used with scintillation detectors, since it could be easily determined what energies should be detected when the angle of scatter between a mono-energetic source and detector is known (energy-coded imaging). However the use of gamma radiation had its own drawbacks. Since gamma ray energies are typically in the MeV range, this means scattering is primarily in the forward direction. As a result, another common practice was to place the detectors on the opposite side of the object like in transmission radiography. In fact, forward CSI was usually coupled with transmission data in order to improve image quality [5, 11, 13]. This was because attenuation effects were a major source of image degradation and signal loss in the system.

In later years other systems decided to adopt an X-ray generator as the radiation source for tomographic imaging [7, 15–19]. A major benefit to using X-ray generators is that it is capable of higher intensities and adjustable energies, although it is at the cost of no longer being a mono-energetic source and therefore complicating the physics. Another practical benefit is that the radiation produced by a generator can be turned on and off, while using a highly active radionuclide requires careful storage and shielding. CSI systems still used collimated detectors that were placed at forward scattering angles or side scattering (90°) angles and would create 3D voxel maps by scanning an area of interest. However it was Bruce Towe and Alan Jacobs in 1981 [15, 16] that introduced the idea of using uncollimated detectors to detect backscattered X-rays to perform tomography. Both the idea of uncollimated detectors and utilizing backscattered radiation were novel concepts in this medical CSI field at the time. They still used a pencil-beam to scan over an object and acquire a signal for each voxel in two dimensions but they achieved image slices at different depths (3rd dimension) by changing the X-ray generator voltage, which in turn changes peak X-ray energies and subsequently the penetration depth of the beam. This was one of the first instances proving that it was possible to image objects with access to only a single side as well as imaging to specific depths in the object. It has been
repeatedly shown [1, 15, 20] that backscatter imaging can only see up to a certain depth in an object, based around the mean free path of the radiation being used. XBI has also been applied to the agriculture/food industry, such as finding bone fragments in chicken breast meat [6, 18].

CSI was also applied towards measuring electron densities, a benefit that Compton scatter systems have over transmission systems. This is because scattering probabilities are directly related to electron densities. One such application was for finding and measuring displacement and velocity patterns on the epicardial surface [21]. Another popular application was for tissue densitometry, such as bone [8, 9] and materials used as breast tissue equivalents [10]. This is not an exhaustive list of what has been tested with Compton scatter systems and more can be found in overview articles that have been written by Speller and Horrocks [22] and Harding [1].

Although less frequent, some studies have attempted to combine CSI with image reconstruction techniques for improved imaging capabilities [23, 24]. More recently, CSI has been used, in combination with CT image data, as a means for real-time tumor tracking during lung cancer treatment [25].

1.2.3 Industrial Applications

Overall, CSI and in particular XBI received greater success in industrial NDT. One factor being that there are many industrial imaging problems that cannot be solved by transmission imaging because the object of interest is either too voluminous or it is impossible to access the opposite side of the object to place a detector. Another cause was the invention of Computed Tomography (CT), which provided excellent resolution and 3D imaging of the human body in a reasonable time frame, thus dominating the medical field. XBI can achieve one or two of those capabilities but struggles to accomplish all three in the same package.

In two articles by Harding [1, 26], he overviews a range of applications from density measurements in soil, concrete, and steel plating, to the detection of buried land mines.
Backscatter imaging has proven quite useful in non-destructive testing (NDT), where importance is placed on the detection of voids, cracks, delamination, and other similar problems.

One specific NDT case that is of importance to this work was the flaw detection in space shuttle insulation foam, which had its origins in land mine detection \([27–29]\). The techniques used for this were, at the time, called Lateral Migration Radiography (LMR) \([30, 31]\) and Radiography by Selective Detection (RSD) \([32, 33]\). In particular, the theory behind RSD is a middle ground between heavily-collimated and under-collimated detectors used in CSI. The detectors are collimated so that they can better see features at a specific depth and deeper in an object, visualized in Figure 1-5. This “selectiveness” means that you will have a unique signal for that detector depending on the location of a feature, its properties with respect to its surroundings, and the depth of the plane at which a detector sees. For example: you will have a different signal if the void is beneath the viewing plane of your detector than if the void was above it. Also the difference in density and interaction coefficient of that feature with respect to its surroundings will effect your signal too. By looking at signals from multiple detectors (each seeing at their own specific depth), it is possible to tell where a void or change in density is in the object.
Figure 1-1. Mass attenuation coefficients in water for a range of photon energies. Data obtained from NIST.
Figure 1-2. Dependence of Klein-Nishina cross section on scatter angle for initial photon energies of 10 keV (outermost curve), 100 keV and 1000 keV (innermost curve). Cross-section is represented by radius vector, which is expressed in units of $r_o^2$. 
Figure 1-3. Schematic illustration of point-by-point arrangement for a Compton scatter imaging system [26].
Figure 1-4. Schematic illustration of plane-by-plane arrangement for a Compton scatter imaging system [26].

Figure 1-5. Schematic of a Radiography by Selective Detection setup containing a subsurface feature with a collimated detector [33].
CHAPTER 2
MCNP6 VALIDATION FOR BACKSCATTER IMAGING

2.1 Motivation

Computer simulations have become an effective way to conduct science that would otherwise be cost prohibitive or impractical if they were to be physically implemented. This especially holds true for nuclear science and engineering, where equipment and resources can be quite expensive, even before the cost of safety measures are taken into account. However, the accuracy of computer simulations are only as reliable as the data that it is put into it. In this case data are referring to: the algorithms used to model the physics; the physical, experimental data (temperature, nuclear cross-sections, etc.); and the information provided by the user. Therefore it is important that verification and validation studies are performed on computer codes in order to improve the trustworthiness of the results they provide.

For these reasons, the radiation transport code Monte Carlo N-Particle (MCNP) 6 was used to model XBI systems in order to assess their capabilities and test the new 3D imaging techniques. MCNP was chosen because it is a well established code, with extensive verification and validation performed on its physics modeling and up-to-date nuclear cross-section data. MCNP6 is the latest version in the long history of MCNP, offering new capabilities that were not available before. Examples of new capabilities include (but are not limited to): improved physics modeling, new source modeling capabilities, and mesh tally improvements. One improvement that is of particular interest is “enhanced photon form factors”, which provides the code with a more complete representation of photon scattering. For a complete list of improvements, see its release overview [34].

Since MCNP6 is a stochastic code, a major concern was achieving accurate results with low uncertainty. This means that certain problems can require long runtimes – hours or even days – depending on the desired results of the simulation. MCNP6 is
embarrassingly parallelizable, which means many processors can be used to run a single
simulation, thereby reducing runtime. For this XBI work, the use of the University of
Florida’s (UF) high performance computing (HPC) cluster was essential. As will be
discussed in the next section, a vast number of simulations and processors were necessary
in order to achieve results within a reasonable time frame.

XBI is not a common use for MCNP6, so there are no proper validation studies
that have been conducted for it. Therefore it was important to first perform a validation
study, albeit a crude one, on previous XBI works before moving forward with new imaging
techniques. The test cases discussed in section 2.3 offer a more qualitative rather than
quantitative analysis of XBI in MCNP6.

The primary reason for this is the lack of information about these older studies.
Since almost all XBI systems are prototypes, there was missing information about the
physical dimensions. Such dimensions include the distance between X-ray source and
radiation detectors and the distance between the system and the surface of the object
being scanned. Another piece of valuable information missing was the energy spectra
of the X-ray tubes being used. All of this information has a great influence on the kind
of signal an XBI system would acquire. Furthermore, photon counts are not provided
for each pixel in the images presented in these studies, so there was no definitive way to
validate MCNP’s results to that of the experiments.

The main focus on these validation tests were to determine whether simulated
images show similar color contrasts to that of the real images, as well as test the depth
discrimination abilities of XBI proven in a paper by Towe and Jacobs [15].

2.2 Methods for Pencil-Beam XBI in MCNP6

For both validation tests, the XBI systems used were based on a point-by-point
scanning method utilizing a pencil-beam X-ray source and common, cylindrical detectors.
The XY plane is parallel to the surface of the object being imaged while the Z dimension
represents the depth dimension going into the object. Therefore a pencil-beam XBI system
will raster scan in the XY dimensions over the object surface. Each pixel in the resulting image refers to a point the system stopped at and collected scattered X-rays with its detectors. The number of detectors used in a design such as this does not increase the number of pixels measured at one scan step, it only improves the count rate (reduced scan times) and removes streaking artifacts due to measuring scattered X-rays from a single angle. Streaking artifacts will be further discussed in section 2.4.

A pencil-beam was formed in MCNP6 by biasing a point source so that it projected a cone of photons towards the phantom. Then by placing a cell of importance 0 with a square whole in front of the cone, the cone beam is shaped into a square pencil beam of a desired size. Since photons are killed and no longer tracked in MCNP when they pass through a cell of importance 0, this system is an ideal pencil-beam of radiation with no photons leaking out towards the detectors. It’s as if the X-ray tube is perfectly collimated so that only X-rays can come from the pencil-beam. This can be easily achieved in a real system as well by shielding both the X-ray tube and detectors so that no leakage photons add erroneous signal to the detectors.

Since a pencil-beam XBI system goes point-by-point, there must be a separate MCNP6 simulation for each pixel imaged, since the system is physically moving each time. This means hundreds of processors are being used at once, with at least one dedicated to each simulation scan point. Python scripts were written to aid in the rapid duplication and editing of MCNP6 input files in order to simulate this system scanning method. This process is simplified by the transformation (TR) card available in MCNP6. This allows the objects assigned to the TR card to be shifted in space without shifting all the other objects in the simulation.

The process for which the simulations were prepared only required three files in a root directory: the Python script that creates everything; a template of the MCNP6 input file, modeling the XBI system and the phantom to be imaged, with the system starting at the
center of the phantom; and a shell script template which submits the simulations to be run
on the HPC. For every pixel in the image, the Python script will:

- Create a new sub-directory
- Copy MCNP6 template to each sub-directory
- Edit the copied template in order to shift the XBI system inside the model so that
  its physical location is where the pixel is located for that sub-directory
- Copy the shell script template to the sub-directory and use it to submit the edited
  MCNP6 file to start running on the HPC

Once all the simulations are finished, the Python script can then go into each
sub-directory, pull out the results from the output file, and aggregate them into a single
array with the same dimensions of the desired image. From there, the array can be
analyzed, processed, or visualized like any other image.

For this study and all others discussed in this work, MCNP6 was set to measure
the fluence of photons crossing the face of each detector; detectors were not given any
material properties. In this way, the data acquired by these simulations are not influenced
by specific materials and their detection efficiencies. The results can be interpreted as if
the detectors had perfect collection efficiency, thereby only the detector’s dimensions can
influence the photons collected. If specific detector materials were of interest, it’s possible
to post-process the data in order to account for detector efficiency effects.

2.3 Test Cases

Two test cases were considered for the MCNP6 validation. The first case was based
off the prototype XBI RSD system developed in the early 2000’s at UF to create 2D
images for the detection of voids and defects in industrial materials, such as the spray on
foam insulation (SOFI) used on the NASA space shuttle [33]. Figure 2-1 shows an image
of the RSD prototype, which consisted of a 1.5 mm diameter pencil-beam using a 60 kVp
X-ray tube spectra, with four NaI scintillator detectors forming a ring around the beam.
Each detector had a 0.2 mm thick lead collimation sleeve placed around it. A 2.9 cm thick
aluminum calibration plate with five 0.6 cm diameter holes drilled halfway into it served as the validation image.

The MCNP6 model sought to replicate the RSD system as closely as possible. One assumption made was the system was placed 1 cm away from the object surface. The length of the collimation sleeves were never mentioned either, so they were set so the detectors could see scattered X-rays at 0.7 cm and deeper inside the phantom. Also, a Matlab code called Spektr was used to try and mimic a 60 kVp X-ray spectra [35]. However, without knowing the exact spectrum used by the prototype, it is uncertain how accurate the Spektr model was.

The second case was based off of a similar XBI design but with a more advanced concept behind it: to scan an image multiple times, at different X-ray tube voltages, in order to acquire images with different depth information, thereby achieving a form of single-sided tomography [15]. A schematic of the system set up from the paper can be seen in Figure 2-2. The test image was based off of an experiment described in Figure 2-3. The physical dimensions recorded in the figure were used to create a duplicate model in MCNP6. Other than the pencil-beam being 1 mm in diameter, the rest of the system parameters were unknown. Therefore the same system model was used from the first test case, except all the collimation was removed and the pencil-beam was adjusted to 1 mm. The technique involved multiple scans of the object at different X-ray tube voltages, so multiple spectra were created in Spektr for the different MCNP6 images. The paper mentioned using 40, 50, 70, and 100 kVp beam energies.

2.4 Results

The experimental image of the aluminum calibration plate for the first test is shown in Figure 2-4, followed by the MCNP6 replication in Figure 2-5. Data from only one of the four detectors was shown in the experimental test, so the corresponding detector from the MCNP6 model was chosen for comparison.
A visual comparison shows good agreement between the images, with the holes in the aluminum creating dark spots in the image. They’re darker because there is a lack of material there for which X-rays can scatter and contribute a signal for those pixels. Likewise, the same streaking artifacts are present in both images, although it is more severe in the simulated image. This is likely due to the system being much closer to the object surface in the model than what was actually done in the real scan. Streaks occur when there is information collected from only one direction during XBI. This is because of the two separate paths the photon must travel. When the primary beam is directly over the void, it is a dark spot due to lack of scattering. However when the system moves away – in this case towards the bottom-left – the void is now in the way of the secondary beam path since the detector is located towards the upper-right direction of the primary beam. This void provides a lack of attenuation for the secondary, scattered photons and thus allows for more photons to reach the detector, resulting in a brighter signal. The closer the system is to the object’s surface, the longer the secondary path is inside the object, and so the longer the streaking artifact will be.

Figure 2-6 shows two of the four images presented by Towe and Jacobs, one at 40 kVp and the other at 100 kVp beam energy. Figure 2-7 shows the attempted MCNP6 replication. The biggest obstacle was that the simulated X-ray spectra provided significantly more image penetration than what was shown by Towe and Jacobs. Therefore lower, mono-energetic X-ray images were taken in order to still prove that it’s possible to perform depth sensitive imaging with X-ray backscatter inside MCNP6. Two possible explanations for the discrepancies in required X-ray spectra: the effect of unknown system geometry and detector efficiencies inside the experimental images required greater X-ray energies; and/or the energy spectra emitted from the experiment’s X-ray tube is far different than what was attempted in the simulation model, using Spektr. Despite the one difference, it was clearly shown that modulating the X-ray energy enables depth sensitive imaging even in simulation.
There was enough evidence from both validation tests to suggest that it is safe and beneficial to use MCNP6 as a tool to design and test new XBI techniques for beyond 2D imaging.

Figure 2-1. Photos of the RSD scanning system at UF. Pencil-beam collimator surrounded by four detectors [32].

Figure 2-2. Experimental setup used to make X-ray scatter images by Towe and Jacobs [15].
Figure 2-3. Composite test object used to demonstrate three-dimensional imaging properties of the backscatter technique [15].

Figure 2-4. Experimental image of the aluminum calibration plate used in RSD imaging [32].
Figure 2-5. MCNP6 simulated image of the aluminum calibration plate used in RSD imaging.
Figure 2-6. Backscatter radiographs of the composite test object shown in Figure 2-3 made at A) 40 kVp and B) 100 kVp beam energy [15].
Figure 2-7. MCNP6 simulated backscatter radiographs of the composite test object shown in Figure 2-3 made at A) 23 keV (mono-energetic) and B) 40 kVp beam energy.
CHAPTER 3
PARAMETRIC STUDY AND PROOF OF CONCEPT FOR XBI TOMOSYNTHESIS USING SPECTRAL DETECTION

3.1 Overview

Previous works have attempted to use Compton scattering as a means for 3-dimensional (3D) imaging, but they rarely just used backwards scattering radiation and they achieved varying degrees of success [12–14]. Typically, the systems utilized highly collimated detectors and photon sources, with and without the image processing to extract depth info [24].

Although the majority of backscatter modalities have been designed to acquire 2-dimensional (2D) images, Towe and Jacobs [15] showed that it is possible to obtain depth-sensitive information (the 3rd dimension in this case) by acquiring multiple images, each at a different X-ray tube voltage. When the average energy of the X-ray source (and therefore penetration depth) is changed, the signal, which is generated from a backscatter design, changes depending on the materials present in the object of interest. A form of image tomosynthesis can be achieved by taking multiple scans at different X-ray energies and observing the change in the image produced; however, the depth resolution of this image is neither precise nor uniform, since it relies on the statistical nature of photon interactions. It also requires multiple exposures/scans, which increases both imaging time and dose.

This objectives of the work presented in this Chapter were 2-fold. First, to perform a parametric study on X-ray energy and its associated backscatter signal depth for materials relevant to medical imaging and non-destructive testing. This will provide a better understanding of the capabilities of this backscatter tomosynthesis technique. Second, to perform a proof of concept for an improvement upon this technique, which inverts the process. Instead of multiple scans at different tube voltages, the technique will require just one scan at a set voltage and the detectors will use energy discrimination in order to
measure the signal generated from different depths within the object. This improvement will reduce both scan time and radiation dose.

3.2 Methods

3.2.1 Imaging System

This study was conducted using the radiation transport code Monte Carlo N-Particle (MCNP) 6 [34]. The modeled backscatter system was a simple pencil-beam X-ray source surrounded by 4 cylindrical detectors, each 5 cm in diameter. The center of each cylindrical detector was located 7 cm away from the center of the pencil-beam and 1.88 cm away from the surface of the phantom being imaged. Detector faces were parallel to phantom surface. The same methods discussed in section 2.2 were used to form simulated images.

This XBI system has the slowest image acquisition but it also has the least complicated design, allowing for fine-tuning of image resolution with the fewest factors to complicate a backscatter signal. For a pencil-beam system, the image resolution across the scanning dimensions (X and Y in this study) is determined by the beam diameter (1 cm for the parametric study) and less so by the detector size. Since the system scans in 1 cm steps, the depth discrimination resolution was at best 1 cm for this study. By simply using a smaller beam, one can increase image resolution at the cost of increased scan time.

The MCNP6 simulations photon source was biased so that it modeled a parallel beam, generating photons only in the negative Z direction towards the test phantoms. Thus removing the signal degradation caused by an isotropic source term.

An image was formed by tracking the combined surface current tallies of the four detectors for each scan point (pixel); therefore, each pixel had the units of photons crossing the surface per starting source photon. Thus, for a known source intensity, the pixel data provided an estimate for the number of photons entering the detectors. By measuring the photon current over the surface of a detector face, information is acquired independently from detector material, volumes, and efficiencies. However, common

38
scintillation detectors (e.g., sodium iodide) have been effectively used in past experimental systems [15].

The four detectors were arranged in a “cross-hair” geometry, and their signals were summed in order to remove the shadowing effects that occur in backscatter imaging when images are collected from only one scatter direction. When images are collected from orthogonal positions around the beam, the cumulative signal removes the shadows apparent from just one detectors image. Figure 3-1A shows a cross-sectional view of the four detectors, where the pencil-beam source would be at the center of the ring of detectors.

Since a pencil-beam system has to raster scan over its object to form 2D images, a separate MCNP6 simulation is needed for each position of the system. In these simulations, the system scanning plane was the XY dimension (depth is Z dimension). For the purposes of speed, a 14 cm by 5 cm scan was used (70 pixels per image). One image was formed for each of three test phantoms for X-ray energies ranging from 10 keV to 200 keV (in steps of 5 keV). Mono-energetic X-ray sources were used in order to accurately define depth penetration.

For the subsequent proof of concept, the pencil-beam was reduced to 5mm in size while detector dimensions were kept retained. The X-ray spectra were modeled after a 150 kVp unfiltered lead target X-ray tube, using the computational tool Spektr [35]. The system raster scanned a 28 cm by 18 cm (56 by 36 pixels) region over the test phantom.

### 3.2.2 Test Phantoms

Three test phantoms were used for the parametric study: (1) a block of soft tissue with a cylinder of air; (2) a block of soft tissue with a cylinder of compact bone [36]; (3) a block of aluminum with a cylinder of air. Figure 3-1B shows a cross-sectional view of one of the phantoms. In each case, the cylinder feature (having 0.35355 cm radius) is embedded inside the surrounding material at a 45° angle so that the top of the cylinder is at the surface and the bottom is located 10 cm deep inside the block. In this way, each
scan step (pixel) of 1 cm in the X direction corresponds to the feature being located 1 cm deeper in the block, allowing for a controlled measurement of depth sensitivity for a given X-ray energy. The phantom blocks were made sufficiently large enough to ensure uniform scattering at the imaging volume boundaries. The XZ plane shown in Figure 3-1B corresponds to the signals collected in the middle row (row 2) of the acquired images. The 4 other rows correspond to when the system is scanning over the bulk, homogeneous section of the phantoms (i.e., no feature is present underneath the systems pencil-beam).

The materials used in these phantoms cover a range of imaging situations. Soft tissue, air, and bone are what primarily make up the components found in diagnostic images of bone, lungs, or gastrointestinal tract. Aluminum is a common material used in products and serves a good first example of the non-destructive testing of metallic objects, scanning for voids and defects.

A tissue test phantom was used to test the tomosynthesis proof of concept. The letters U and F were embedded 6 cm and 8 cm deep in the tissue, respectively. Each letter was 2 cm thick and spanned a depth of 1 cm inside the phantom. The letter U was made of air, whereas the letter F was made of compact bone. These depths were chosen based on the results of the parametric study and were another way to verify the depth penetration of the X-ray source. Figure 3-2 shows a cross-sectional view of the phantom in MCNP6.

### 3.2.3 Determining Image Depth

For the parametric study, a simple algorithm was developed to analyze the acquired images and determine how deep the feature in a given phantom is distinguishable from the surrounding bulk material. The algorithm defines distinguishable as any pixel intensity that is greater or less than three standard deviations away from the mean pixel intensity. Mean pixel intensity and standard deviation are calculated from all the pixels located in the first and last two rows of the image (i.e., rows 0, 1, 3, and 4). These rows represent the background for the image (i.e., signal from bulk homogenous material of a phantom).
Because the feature inside the imaged phantom is located in the middle row (row 2), only the pixels in row 2 are checked to determine which intensities are greater than three standard deviations away from the mean. The location of the pixel farthest along this row (subtract 2 pixels) is the depth in centimeters in which the feature is still distinguishable from its surrounding material. The pixel index starts at 0, and the first three pixels in the row either do not contain the feature or it is at the surface of the phantom, which is a depth of 0 cm, and should not be counted; thus, 2 pixels are subtracted. Figure 3-3 illustrates this theory using the image obtained from the tissue/bone phantom with 50 keV photons. Pixel location 7 is the farthest pixel outside the statistical fluctuations of the soft tissue background. Therefore, 50 keV photons can distinguish bone from soft tissue up to a depth of 5 cm.

3.3 Results and Analysis

3.3.1 Parametric Study

Figures 3-4A and 3-4B show the raw images of the soft tissue phantom with a cylinder of air at 10 keV and 70 keV X-rays, respectively. The system scan began 2 cm (2 pixels) behind the cylinder of air. The interrogation source energy affected what was visible inside the object being imaged. As Figure 3-4B shows, the 70-keV image is noisier with no clear depth cutoff; this is partly due to the system resolution and limited number of pixels in the image. However, by using the algorithm described previously, the signal from the air feature dropped off to background levels at approximately 6 cm. The air inside the phantom caused a lower signal than the bulk material because of its low density; thus, photons were less likely to interact with it. This behavior was also observed inside the aluminum/air phantom.

Figure 3-5 shows the association between signal depth and X-ray energy for each phantom. The low-energy photons (40–90 keV) penetrated deeper into the soft tissue when air rather than compact bone was present. Notably, because bone is denser than the material surrounding it, the penetration depth for the soft tissue phantom with
bone dropped to zero at 80–85 keV and then spiked again. For low-energy ranges, the photoelectric effect absorbs X-rays more frequently than it scatters them, thereby reducing bone signal. However, beyond 50 keV, Compton scatter becomes the dominant interaction and bone will begin emitting more signal than the surrounding soft tissue. During this transition period, bone eventually generates a signal intensity comparable to soft tissue, causing it to be indistinguishable from its surroundings. Figure 3-6 captures this phenomena of transitioning signal intensity. The 80 keV pixels clearly have much smaller changes in intensity as compared to other energies.

This shifting of a features signal from low to high intensities relative to its surroundings, with increasing X-ray interrogation energy, is not exclusive to bone embedded in soft tissue. This phenomena should be inherent in any backscatter imaging scenario where high electron density materials are surrounded by a lower electron density medium. Our results showed that there is a limited depth at which bone and air can be distinguished from soft tissue and increasing photon energies yield no more information: 5 cm and 7 cm for air and bone, respectively. Deeper depth information was achieved for specific ranges of photon energies: 6 cm at 50–80 keV for air and 8 cm at 95–100 keV for bone.

An interesting result seen in Figure 3-5 is the seeming plateau of depth distinguishability for tissue/air and tissue/bone. The hypothesized explanation for this occurrence is a combined effect from the photon’s mean free path in the tissue and the detector’s placement. The depth at which the higher energy photons interact inside tissue, combined with the detector’s distance from the incident beam, causes signals to only be collected when photons scatter at angles greater than 140°. These kinds of scatter angles are increasingly unlikely at these higher photon energies, where forward scattering dominates. By this logic, the plateau has not yet occurred in the aluminum phantom because even at high photon energies, the mean free path is still shallow enough that scattering angles aren’t so severe.
Image penetration was more consistent, with a gradual incline for the air-filled aluminum phantom; overall it was lower than the other phantoms at the photon energies considered due to aluminum's much higher electron density. Significantly higher X-ray energies are necessary for depth-sensitive information in industrial applications containing aluminum and other high electron density materials, depending on the desired depth.

3.3.2 Tomosynthesis Proof of Concept

Because the Spektr tool can only model an X-ray spectrum up to 150 kVp, a tissue test phantom was used; it has lower attenuation coefficients than aluminum at those energies. The average energy for the spectrum was 57.6 keV. At that energy, photons have a mean free path of 4.83 cm in soft tissue (based on mass attenuation data from NIST). The air and bone letters were imbedded in the tissue their max depth based on the results in Figure 3-5 (6 and 8 cm, respectively). At approximately 90 keV there was a clear cutoff between air and bone image depth penetration. Therefore, two energy bin groups were used for the detector: (1) any signal less than 90 keV and (2) any signal above 90 keV.

Figure 3-7 shows the resulting image after the phantom was scanned and signals were binned into their respective energy groups. When the system measured photons less than 90 keV, only the letter U was present, which means the system only sees up to 6–7 cm deep. But when it measured photons greater than 90 keV, the letter F was also visible, so the system can now see up to 8–9 cm deep.

The letter U produces a low signal because of the low density of air. The letter F, which is composed of compact bone, provides a brighter signal, as was observed in the parametric study at those energy ranges.

The average backscatter signal in the lower energy and higher energy groups was 2.95% and 0.0245% of the primary beams particle flux, respectively. This corresponded to an energy fluence of 1.45 keV per starting source particle for the lower energy group and 0.02311 keV per starting source particle for the higher energy group. The total energy fluence coming from the incident beam was 57.6 keV per starting source particle. By
dividing the energy fluence at the detector face by the energy fluence projected at the phantom, one obtains a collection efficiency of 2.55%.

It can also be seen that the range of signals in the high energy group is quite small, only a 1.5% change from the lowest signal to the highest signal in the image. If this were to be the case for a real system acquiring this image, then a photon source with an intensity of $10^9$ photons per scan point would be necessary in order to acquire enough counts so that the image contrast between the lowest and highest signal are greater than their associated uncertainties.

This drastic reduction in signal for the higher energy group can be explained by Figure 3-8. The bottom distribution represents the normalized energy spectra that was collected at the detector face during the image acquisition. As can be seen by the black vertical lines, applying a 90 keV threshold value means only 11% of the initial spectra (and 1% in the scattered spectra) is placed in the high energy group. It was found that shifting the threshold down to 60 keV maintained image quality and depth discrimination while increasing the signal collected in the high energy group by an order of magnitude.

It was found that the measured energy spectra aligns well with the distribution one would get if the initial 150 kVp spectra was scattered by 122°, and then had its low energy components filtered. This filtering is what would be expected to happen as these scattered photons have to leave the phantom, passing through more material on its way to the detector faces.

### 3.4 Conclusions

X-ray backscatter imaging only needs single-sided access to an object and has the ability to acquire images beyond its normal 2D capabilities with just one scan of an X-ray source and spectral detector. This tomosynthesis technique is a marked improvement over its predecessor, which required multiple scans at different tube voltages. By performing a parametric study of photon energies on the depth sensitivity of images acquired for specific
materials, a database can be developed to aid in the design and use of backscatter imaging for 3D interrogation.

The materials used in this study were chosen for their relevance to applications in medical imaging and non-destructive testing. Significant penetration was acquired (up to 5–6 cm) with moderate X-ray energies (40–55 keV) in organic materials, such as compact bone, soft tissue, and air; however, bone was most visible at energies greater than 90 keV. These energy ranges are easily achievable with standard X-ray sources. However, there appears to be a limited depth at which air and bone can be distinguished from the surrounding soft tissue. For industrial materials, such as aluminum, much higher X-ray energies are needed for significant depth information. Although this study used a collimated pencil-beam of radiation, other source/detector geometries could possibly be used in order to reduce scanning times.

To further improve this study, photon energies beyond 200 keV for aluminum phantoms should be evaluated. At higher energies, the photoelectric absorption becomes negligible but the probability for backwards scattering photons also decreases. It would be beneficial to investigate the absorbed dose received to organic materials for images acquired in this study and compare that with standard imaging techniques, such as radiographs and CT scans. The parametric study performed here can be expanded to many more materials, such as wood, concrete, and other organs. Also, entirely new parametric studies can be performed that investigate how depth penetration and depth resolution are affected by detector size, placement, and materials.
Figure 3-1. Cross-sectional views of simulation geometry in MCNP6. A) Detector ring in the XY plane. B) Parametric study phantom in the XZ plane. Blue represents block of soft tissue; yellow represents the compact bone feature; green represents air; white represents void (i.e. no material). Detector volumes were void (white) since only photon currents over bottom surface was measured. The system scans over the phantom in the XY dimension.
Figure 3-2. Cross-sectional views in the XY dimension of the tomosynthesis test phantom in MCNP6. A) At 6.5 cm deep. B) At 8.5 cm deep. Green represents air and yellow represents compact bone.
Figure 3-3. Depth distinguishability using image of tissue/bone phantom with 50 keV photons. Mean and standard deviation of pixel intensities determined by the bulk material imaged. Farthest pixel location in the middle row that has an intensity greater than 3 standard deviations from the mean determines how deep a feature is distinguished from its bulk material.
Figure 3-4. Raw images of soft tissue phantom with cylinder of air. A) Interrogated with 10 keV X-rays. B) Interrogated with 70 keV X-rays. For each image, pixel rows correspond to Y scanning dimension and pixel columns correspond to X scanning dimension.
Figure 3-5. Image penetration depth versus X-ray interrogation energy. Image penetration depth represents the deepest a feature was distinguishable from the signal measured in the surrounding material. Surrounding material was either soft tissue or aluminum and feature material was either air or compact bone.
Figure 3-6. Normalized pixel intensity vs pixel location along the middle row (row 2) of images acquired from the tissue/bone phantom. As Interrogation energy increases, the signal from the bone transitions from lower to higher signal than that emitted by surrounding tissue.
Figure 3-7. Images acquired in MCNP6 using spectral detection in backscatter imaging system. A) Signal from photons less that 90 keV. B) Signal from photons greater than 90 keV. Units are number of photons crossing detector face normalized per source intensity. For each image, pixel rows correspond to Y scanning dimension and pixel columns correspond to X scanning dimension.
Figure 3-8. Comparison of normalized X-ray energy spectra. Top spectra is the initial source spectra used to investigate the test phantom. The bottom distributions show the measured spectra at the detector faces and the initial source spectra scattered by 122 degrees. Black vertical lines represent energy threshold that was used for depth/energy discrimination.
CHAPTER 4
2D BACKSCATTER IMAGING: PYTHON TOOLKIT

4.1 Motivation

As was discussed in section 1.2.1, the choice of source and detector geometry for a backscatter system has a set of tradeoffs between acquisition time, resolution, and signal to noise ratio (SNR). Chapters 2 and 3 used pencil-beam geometries, which has good resolution but slow acquisition time. The remaining work discussed will be about fan-beam geometries, specifically plane-by-plane imaging, which should provide faster imaging but at the cost of resolution and SNR.

The system under consideration is that of a fan-beam and a 2D detector array with a collimation grid, which is henceforth referred to as a 2D backscatter imaging system. In this case the term 2D refers to the two dimensional detector array, which enables the acquisition of a 2D signal for every step the system takes: one dimension across the object’s surface; one dimension into/inside the object; the third dimension being the mechanical scanning of the system across the object. This system is likely to be more useful than a pencil-beam source in applications where relatively large areas must be scanned in a reasonable amount of time. However, formulating a collimation grid that is optimized for depth sensitive imaging is time consuming and prone to error if it has to be done manually for every design iteration. Therefore this is an ideal situation for the development and use of a code package that will do the heavy lifting and allow the user to focus on the system’s design parameters, such as resolution and detector dimensions.

This chapter serves as an overview of the Python toolkit, backscatter2D, that was developed for the design of 2D backscatter imaging systems, conversion into MCNP6 inputs, and subsequent image formation and analysis of the simulation outputs. A simple test is performed before moving on to the investigation of different scanning methods for 3D imaging, discussed in the following chapters.
4.2 Code Description

The toolkit/application package is written in Python and is called backscatter2D. Within the package, there are currently seven modules:

- system_design
- mcnp
- visualize
- matplotlibting
- tally_reader
- hpc_utils
- reconstruct

The focus of this code description will be about the system_design module, with brief mentions of the other modules when it is necessary. The reconstruct module will be further discussed in Chapter 6, when it is specifically used. The full toolkit can be found in the Appendix.

4.2.1 Backscatter System

The system_design module utilizes object-oriented programming, creating a class for each of the components in the 2D backscatter system: fan-beam source, detector array, and collimation grid. The collimation grid can either be an optimized grid or a uniform grid. There is also a class for the system as a whole called BackscatterSystem, which is just a composition of the previously mentioned classes except for added attributes, such as step_size. This attribute determines the distance of each step the system takes when it is scanning over an object, which describes the resolution in the X dimension of the images acquired. In this toolset, X dimension refers to the scanning dimension, Y dimension is the breadth of the fan-beam and detector width, and Z dimension is the height/depth. Step size also plays a role during MCNP input formulation because a collimator for the photon source is needed in order to form it into a fan-beam. Since the source is divergent, the
fan-beam broadens in the X dimension as it penetrates deeper into the object. Therefore collimators are designed so that the fan-beam is as broad as the system step size when it has penetrated a maximum depth inside the object. This maximum depth is an attribute of the collimator class, discussed later. Tables 4-1 through 4-4 give brief descriptions of the attributes required for each class in the design of a 2D backscatter system.

When all the components of the backscatter system has been designed and placed inside of a BackscatterSystem instance, BackscatterSystem can perform some simple parameter calculations and store them in a dictionary called depths. Such parameters include: depth range a pixel can see for each row in 2D array; estimated resolution of pixels in the Y dimensions, for each row; and depth location that each pixel center is looking at inside object under investigation.

4.2.1.1 X-ray fan-beam

The photon source is handled in the toolkit by a class called XRayFanBeam, its required attributes listed in Table 4-1. X-ray is used in the naming convention because it is assumed that X-ray sources will be the most common type of radiation source used in backscatter imaging. However it is entirely acceptable to use a mono-energetic source that is more akin to imaging with gamma emitting radioisotopes, which have been used before in Compton scatter imaging and densitometry [8, 11–13]. You can either pass it a single energy (in MeV) or you can give it a path to a file name containing an energy distribution. The distribution must be written in the file as one entry per line and each entry contains two space delimited values. The first value is the energy of the photon (again in MeV) and the second value is the frequency in whatever units desired. Once the file is read and converted into a 2D array, it will create another array that normalizes the distribution based on the frequencies provided. This array will be used to create a source definition in MCNP with a varying energy distribution.

The surface offset distance and half angle attributes determine the shape of the fan-beam. A larger half angle means a wider fan and a half angle of 0° creates a
pencil-beam. The offset distance merely describes the location/height of the photon source, which is needed when creating a source definition in MCNP. It should be noted that if the fan half angle and surface offset are not correct, it’s possible the breadth of the fan-beam at the object’s surface will not be as long as the breadth of the desired detector width, meaning not all the pixels can be correctly imaged inside the object. This can be mitigated by first designing the detector in the toolset and then passing it to an instance of the XRayFanBeam class while calling its function that can optimize either its surface offset or half angle based on the detector’s dimensions.

The matplotting module provides an xray.spectrum method that lets the user view their imported energy distribution to see what it looks like.

4.2.1.2 Detector

The Detector class contains the data regarding the dimensions and geometry of the 2D detector array. It is a simple class and is the basis for the design of the collimators in the collimation grid. Table 4-2 summarizes the class’ required attributes.

Besides containing information on the size and number of pixels in its array, it contains an important attribute called the detector angle. An angle of 0° means the detector face is parallel to the object surface while an angle of 90° has the detector face perpendicular to it. The angle influences collimator length as well as the scattered photon signal. Detector angle should never be set below 45° because at that point it is impossible for Z collimators to block signals originating from certain depths. To be more clear, the shallower angles actually invert the concept of Z collimators; they would instead block signal from a certain depth and above rather than a certain depth and below. This code was designed to only consider the latter situation and therefore would provide wrong answers or break if the detector angle was below 45°.

It’s always a good idea to plot and investigate the systems design before implementing it into MCNP and collecting results.
4.2.1.3 Collimation grid design

The collimator classes are the workhorses in the system design module since they are the component that enables enhanced imaging in two dimensions at once. Without collimation, detector pixels collect scattered photons coming from too many different locations within the fan-beam and imaging would be nearly impossible due to poor resolution. By applying collimation to the detector, it’s possible to block scattered photons originating from unwanted locations inside the object, allowing for better resolution and more detailed information about the points of scatter at the cost of reduced signal. The spatial discrimination in signal is useful for depth sensitive backscatter images.

A 2D detector array has two primary forms of collimation in its collimation grid: the depth (Z) collimation and the surface (Y) collimation. Each form of collimation can be solved for independently from the other, since they influence two separate dimensions of the image. There is a unique length for both Y and Z collimators for each row of pixels in the detector array, since each row is aimed at measuring a specific depth inside the object. The fundamental relationships between system parameters and collimator length for Z and Y collimators are shown by Equations 4–1 and 4–2, respectively. In each case, the equations are iterated over each location \((X, Y, Z)\) of a row of pixels on the detector array \((P_i)\) and its associated depth \((d)\) its trying to image. A collimator’s Length is dependent on the following system parameters given by the user: height of a single detector pixel \((P_H)\); width of a single detector pixel \((P_W)\); thickness of the collimators in the grid \((C_t)\); detector angle with respect to the object surface \((\theta_D)\); and the desired resolution in the \(y\) dimension \((y_{res})\).

\[
\text{Length}_z = \frac{P_H - C_t}{\tan(\zeta)} \quad (4\text{-}1a)
\]
\[
\zeta = \theta_D - \phi \quad (4\text{-}1b)
\]
\[
\phi = \tan^{-1}\left(\frac{P_{i,X}}{d + P_{i,Z}}\right) \quad (4\text{-}1c)
\]
For $Z$ collimators, a specific length is calculated so that its row of pixels can not see scattered photons beyond a certain specified depth. However, since the $Z$ collimator above the current row of pixels is being optimized for the row of pixels above it, it is not possible to specifically block photons from a specified “upper” depth. Therefore it’s possible, for example, that even though a pixel row in the detector is collimated to see nothing deeper than 4 cm, it could potentially see scattered photons from any point above that depth, all depending on how long the $Z$ collimator above it is.

\begin{align*}
L &= \sqrt{ \left( P_{i,X} + \frac{P_H}{2} \cos(\theta_D) \right)^2 + \left( d + P_{i,Z} - \frac{P_H}{2} \sin(\theta_D) \right)^2 } \\
Length_y &= \frac{2L}{\frac{y_{res}}{PW-C_t} + 1} 
\end{align*}

(4-2a)

(4-2b)

The $Y$ collimators are calculated at the center of a pixel and are as long as they have to be to insure that each pixel has a resolution of $y_{res}$ as specified by the user. The downside is that it’s impossible for $y_{res}$ to equal the size of the visible pixel width ($PW-C_t$) because that would mean the collimator length is as long as the path length of the scattered photon ($L$), which is mostly inside of the object. Having a collimator penetrate the object defeats the purpose of “non-destructive” testing. Even when $y_{res}$ is 3-4 times the size of the pixel width, collimator lengths can be prohibitively long and greatly reduce SNR. This is an inherent drawback to backscatter systems that utilize photon sources and detectors greater than one dimension.

### 4.2.1.4 Optimized collimation grid

The first type of collimator designed for this toolkit was the optimized collimation grid, which was thought to be an advancement over the uniform collimation designs, discussed in the next section. Table 4-3 describes the key attributes needed to design an optimized collimation grid.

The optimized collimation grid class is simply called Collimator. This class contains attributes like max depth, collimator thickness, and offset distances from the surface and
fan-beam. Max depth is the deepest point the entire system will see and is the value that the bottom row of detector pixels (closest to surface) are set to view. It also is the depth used to calculate the fan-beam collimators discussed earlier. Since the thickness of a collimator will block a portion of each detector pixel, it’s important to watch this attribute in combination with pixel dimensions. Make the collimators too thick or the pixels too small and you’ll entirely block signal because you essentially made a lead sheet over the detector. Offset distances provide a hard limit for how close the detector and its collimation grid can get to the fan-beam and object, since you want neither the collimator blocking the fan-beam nor scraping the object surface.

The methods contained inside Collimator take a Detector instance and uses its information to help create the correct lengths for each row of Z and Y collimators. It utilizes Equations 4-1 and 4-2 as well as others in order to place the detector/collimator system as close as possible to the surface and fan-beam while at the same time ensuring the proper resolution and viewing depths are achieved.

The material attribute aids in the MCNP input deck creation, since different materials can be used to create collimators; two of the most common being lead and tungsten.

4.2.1.5 Uniform collimation grid

The uniform collimation grid class, named UniformCollimator, is a simplified version of an optimized collimation grid. The attributes necessary for this class are similar to that of the Collimator class, with a few exceptions, as seen in Table 4-4. The max_depth and y_res attributes are no longer required, since they are no longer optimized and controlled by the user. Instead, they get calculated once the collimators have all been created with the same requested length. This kind of collimation grid design should be easier to manufacture and can sometimes create results just as good as an optimized collimation grid.
4.2.2 MCNP Input Conversion

The entire design process is done inside Python but it is MCNP that will actually simulate an image using that design. It would be extremely tedious to manually input the proper values in order to replicate the system inside MCNP. Therefore the mcnp module is supplied, which contains a method specifically for converting the designed system into proper MCNP format as well as some generic methods for inputting the most common cards used in MCNP such as cell cards, surface cards, tally cards, etc.

Data for the detector and collimators in the grid are stored in their classes in a style that matches the BOX Macrobody format of MCNP. Therefore all the collimators are written as such inside the input deck, while the detector is also turned into an FMESHn4 tally. Each element in the mesh tally matches the number and dimensions of the detector pixels.

Collimator cells are made of whatever material is requested by the user while the detector cell is a void, since the analysis seeks to be independent of detector composition. Properties such as detector efficiency can be applied during post processing of the results if desired. Since the FMESH tally is a measure of the photon flux over all surfaces of a volume, the detectors are made sufficiently thin so that the results will approach that of a surface current over just the face of the detector pixels and not their other sides.

The cards generated to make the system are all assigned a coordinate transformation (TR) card. This enables the easy creation of multiple input files, modeling the scanning motion of the system over an object. Only the TR cards need to be adjusted by the system step size instead of recomputing and rewriting every surface and cell card.

4.2.3 Visualization

Both the visualize and matplotting modules provide options for viewing the designed BackscatterSystem. These tools are purely for qualitative purposes, such as observing the scale of the collimator lengths with respect to other system properties and checking that
no errors occurred. Quantitative information about the system is stored inside the classes of the backscatter design.

The tally_reader module contains a CartesianMeshTally class that is able to read meshtally output files that are generated by MCNP when FMESH tallies are used. The class compiles the tally data into arrays which can then be visualized with common tools available to Python.

4.3 Testing of Python Toolkit

To test the capabilities of the toolkit, as well as provide a trial for depth sensitive imaging using a 2D backscatter system, a former MCNP simulation was replicated from a Master’s Thesis at UF [37]. In the thesis, they briefly discuss the possibility of 2D backscatter imaging and attempt a simple image in MCNP of an aluminum block with a ring of air embedded in it.

4.3.1 Test Simulation Setup

The MCNP simulation used in the thesis described a 2D detector array that was 6 rows with 40 pixels each, for a total of 240 pixels in the array. Each pixel was 2.5 mm high and 1 mm wide. The detector array was angled at 45° and a 120 keV fan-beam was used. A 1.5 cm long and 0.4 mm thick tungsten collimation grid was placed over the detector. Therefore a uniform collimation grid was used. Since the offset distance of the collimators/detector from the object surface and fan-beam were not provided, a best guess was used (0.9 cm from surface and 2.55 cm from fan-beam). The X-ray fan-beam was set to 5 cm above the object surface.

The phantom being imaged was a solid block of aluminum with a ring of air placed 1 cm deep inside of it. The ring of air was 1 cm thick, had an inner diameter of 0.5 cm, and an outer diameter of 1.5 cm.

4.3.2 Test Simulation Results

Figure 4-1 below shows the replicated system that was designed using the toolkit and then visualized in 3D using the visualize module. The system was then exported to
an input deck for MCNP and then the test phantom was input by hand into the deck. Figure 4-2 shows cross sectional views of the geometry plotted in MCNP, where the test phantom and system can be seen. These images show the success of the toolkits ability to model a 2D backscatter system and subsequently export it to an MCNP file.

Since MCNP simulations of a 2D backscatter imaging system can be computationally expensive, only two scan points were simulated. The first scan point (i.e. fan-beam position) was directly over the center of the ring of air and the second point was over solid aluminum. Obtaining an image of the solid aluminum acts as a form of baseline/background signal that can be used as a filter to remove the disparities in pixel intensity due to a divergent fan-beam source and photon attenuation. When you divide your 2D image slices by the aforementioned filter, pixels that had lower signals near the edges and bottom of the scans will be given more weight since they’re being divided by smaller values in the filter (because signal was also lower in that image due to the same reasons of source divergence and attenuation). Any changes in density/material will stand out because it will have a noticeably different signal than what was in the scan of the bulk homogenous material of the filter image.

Figure 4-3 shows the filtered image of the ring of air, which has had its pixel values rescaled, to help improve the contrast of the air and aluminum. Each row in the image represents a row in the detector array looking at a different depth in the phantom. This figure shows that depth sensitive imaging is a possibility, since the two black regions, representing the cross-sectional view of the air ring, are located near the middle of the image. This corresponds to where the air ring was located inside the phantom.

Figure 4-4 shows the pixel values for the filtered image along the middle row of the 2D detector array (row 2 in the Y axis of the image in Figure 4-3. The physical locations of the phantom materials have been superimposed over the pixel values to help show where the contrast in signal should appear. There seems to be good agreement with the line plot of pixel values along with the physical change in density/material inside the test.
phantom. It was estimated that it takes a span of roughly two pixels to shift from the aluminum signal to the air ring signal. Since the pixels in the 2D detector array were set to have a physical width of 1 mm, this means the imaging resolution for this system is estimated to be 2 mm. This resolution is for the fan-beam dimension (Y dimension in the MCNP simulation). The resolution in the scanning dimension (X dimension in this simulation) would be determined by how thick the fan-beam was as well as the scan step of the system.

Table 4-1. List of variables required from user for XRayFanBeam class.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description (Units)</th>
</tr>
</thead>
<tbody>
<tr>
<td>energy</td>
<td>Either a mono-energetic source (MeV) or an energy distribution (array[MeV, Frequency])</td>
</tr>
<tr>
<td>$O_F$</td>
<td>Offset distance of the photon source from the object surface (cm)</td>
</tr>
<tr>
<td>$\Phi_F$</td>
<td>Half angle describing the breadth of the fan-beam (degrees)</td>
</tr>
</tbody>
</table>

Table 4-2. List of variables required from user for Detector class.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description (Units)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$N_H$</td>
<td>Number of pixels (rows) in the height of the detector array</td>
</tr>
<tr>
<td>$N_W$</td>
<td>Number of pixels (columns) in the width of the detector array</td>
</tr>
<tr>
<td>$P_H$</td>
<td>Height of a single pixel in the detector array (cm)</td>
</tr>
<tr>
<td>$P_W$</td>
<td>Width of a single pixel in the detector array (cm)</td>
</tr>
<tr>
<td>$\theta_D$</td>
<td>Angle of detector array from the surface of the object (degrees)</td>
</tr>
</tbody>
</table>
Table 4-3. List of variables required from user for Collimator class (optimized collimation grid).

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description (Units)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C_t$</td>
<td>Thickness of the collimators in the collimation grid (cm)</td>
</tr>
<tr>
<td>$\text{max_depth}$</td>
<td>Maximum depth the system can see (cm)</td>
</tr>
<tr>
<td>$O_B$</td>
<td>Offset distance of the detector array from the fan-beam (cm)</td>
</tr>
<tr>
<td>$O_S$</td>
<td>Offset distance of the detector array from the object surface (cm)</td>
</tr>
<tr>
<td>$y_{res}$</td>
<td>Resolution of the image pixels in the Y dimension (cm)</td>
</tr>
<tr>
<td>material</td>
<td>Type of material the collimator is made out of (ex: Tungsten)</td>
</tr>
</tbody>
</table>

Table 4-4. List of variables required from user for UniformCollimator class (uniform collimation grid).

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description (Units)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C_t$</td>
<td>Thickness of the collimators in the collimation grid (cm)</td>
</tr>
<tr>
<td>$\text{length}$</td>
<td>Length of all the collimators in the collimation grid (cm)</td>
</tr>
<tr>
<td>$O_B$</td>
<td>Offset distance of the detector array from the fan-beam (cm)</td>
</tr>
<tr>
<td>$O_S$</td>
<td>Offset distance of the detector array from the object surface (cm)</td>
</tr>
<tr>
<td>material</td>
<td>Type of material the collimator is made out of (ex: Tungsten)</td>
</tr>
</tbody>
</table>
Figure 4-1. Visualization of 2D backscatter imaging system using a uniform collimation grid (gray). Fan-beam is red, detector is light blue, and object surface is black.
Figure 4-2. Cross-sectional views of test simulation geometry inside MCNP6. A) XZ plane showing backscatter system and test phantom with air ring. B) XY plane showing ring of air inside aluminum phantom. C) Detector plane showing collimation grid.
Figure 4-3. Simulated backscatter image of air ring inside aluminum block. Y axis is depth and X axis is breadth of fan-beam. Dark regions caused by cross-section of ring.

Figure 4-4. Line plot of pixel intensities along middle row of 2D detector array for the simulated image in Figure 4-3. Red points/pixels were used to calculate the mean signal for aluminum, green points/pixels for air. Takes roughly two pixels to change from aluminum signal to air, meaning an estimated imaging resolution of 2 mm.
CHAPTER 5  
2D BACKSCATTER IMAGING: LINEAR SCANNING

The test results of Chapter 4 showed that 2D backscatter imaging systems have the potential to achieve depth sensitive imaging, like that of the XBI system discussed in Chapter 3. However, instead of using energy discrimination to separate depth information, a higher dimension of detectors are used; a 2D array of pixels. Each row in the array looks at a different depth, the resolution of each depth slice is dependent on the length of the collimation used (specifically the Z collimators mentioned in section 4.2.1.3). As long as the collimation grid does not become too long, then a 2D backscatter system can acquire images at a faster pace then that of a pencil-beam system, at a reduced resolution. If the grid becomes too long, then too much signal is filtered out and the dwell time for each scan point becomes prohibitively long, thereby becoming no faster than a pencil-beam design.

This chapter and the next look at two different methods of scanning for a 2D backscatter system. This chapter discusses the simpler, linear scanning method. The system starts at one end of the object and scans across in one dimension, until it reaches the other end. If the system is scanning in the X dimension, then at each point in the scan the system is acquiring scattered signals in the Y dimension (for every column of pixels) and in the Z dimension (for every row of pixels). In the end, a 3D array of pixel data is collected: a series of 2D radiographs, one at each depth/row in the object/detector. The next chapter will discuss a rotational method that requires an image reconstruction algorithm.

In order to make an accurate comparison between the two methods, the same system design and test phantom are used.

5.1 Imaging System

Using the Python toolkit, a 2D backscatter system with an optimized collimation grid was made. The X-ray fan-beam was placed 25 cm above the object surface and used a 150
kVp, unfiltered, lead target spectrum, produced by Specktr. With an average energy of 57.6 keV, the photons have a mean free path of 1.50 cm in the concrete that was modeled (based on mass attenuation data from NIST). The 2D detector array was placed at a 45° angle and was 5 rows by 56 columns. Each pixel in the array was 5.5 mm high and 5.5 mm wide, so at each scan point the detector array imaged a 2.75 cm deep by 30.8 cm wide area. The detector was made 1 mm thick. The optimized collimation grid was set to have a max depth of 6 cm and a desired y-resolution of 8.6 mm, made of tungsten collimators that are 0.5 mm thick. The detection/collimation was offset from the fan-beam by 2.7 cm and from the object surface by 1 cm. The systems step size was set to 5 mm.

5.2 Test Phantom

The test phantom was very similar to the one used in the proof of concept study discussed in section 3.2.2, and therefore are referred to Figure 3-2 for a picture of the phantom in MCNP6. It was a block of solid, regular concrete, spanning 80 cm in the X and Y dimension and 20 cm deep (Z dimension). The letters U and F were still 2 cm wide, except the U was made of water and the F of 316 stainless steel. Since these materials are much more attenuating than tissue but the X-ray energy spectrum was still 150 kVp, the letters were moved much closer to the surface. The letter U was embedded at a depth that spanned from 0.6 cm to 1.3 cm deep. This range corresponded to the depth that the second row (from the top) of detector pixels was calculated to image. The letter F was placed in the range for the next row beneath it (3rd row from top). It spanned a depth of 1.38 cm to 2.08 cm.

The system scanned the phantom in 36 steps. With 5 mm step size, this corresponds to a scan length of 18 cm. A complete scan consists of a 3D image that is 36 x 56 x 5, with a field of view spanning 18 x 30.8 x 2.75 cm.

5.3 Imaging Results

Figure 5-1 shows the resulting 3D image after it was simulated in MCNP6. Each 2D image is of a different depth/detector row. As was done with the test image in
section 4.3.2, a background filter was applied to every scan point in order to remove effects of attenuation and fan-beam divergence. The 2D (YZ plane) image slice at the first scan point (bottom row in images) was used since it was when the system measured a uniform concrete signal. By dividing every other scan point/row in the images, the imaging artifacts are removed and the contrast in signal from other materials is enhanced, making it easier to see. Since the changes in signal can be very subtle between materials, readjusting the pixel gray-scaling is typically needed in order to better see the features; this was done in Figure 5-1. The letter U produces a brighter signal because it scatters photons more frequently than it absorbs them, with respect to the other materials present, for this range of X-ray energies. The letter F still has a higher likelihood to absorb photons at these energies and therefore produces a darker signal.

With respect to the phantom, the letter U should be visible in the second image and deeper (image at 0.958 cm), while the letter F should be visible in the third image and deeper (image at 1.735 cm). The reason they will be visible for every image deeper is because the feature is affecting the primary beam path of the X-rays, so they will always influence the signal collected whether or not they’re actually scattering from that feature at that measured depth. But if signal is being measured from scattered X-rays above the feature, then it will not be visible since it’s not influencing any part of the beam’s path at that point.

To an extent, this concept holds true for Figure 5-1, except some shadows can be seen of the letters at depths above which they don’t belong to. There is also the problem of very poor image quality in this figure, which is believed to be caused by a few factors. First, the X-ray energy spectra is likely too weak to penetrate the phantom well and provide significant signal. Second, the optimized collimation grid that was used to achieve the desired depths and resolutions caused extremely long fins; they ranged from 8-26 cm long. These exceptionally long collimators create a very small solid angle from which photons can reach the pixels, which leads to longer scan times. This is reflected in the
MCNP6 simulations. After 36.5 billion particles were ran, which took over a week to finish the full image, uncertainties in the meshtallies were still unacceptably high. These high uncertainties affect the image quality in Figure 5-1. Continuing to run simulations even longer until all uncertainties are sufficiently low (less than 10%) is far too time consuming to be practical. This problem seems to be inherent to fan-beam designs. Variance reduction techniques will be necessary if fan-beam imaging using MCNP6 is to be continued.

This complexity of creating real 3D images using a 2D backscatter system comes from the fact that it’s single-sided imaging. Each row in the detector that sees deeper in the object will inherently have worse resolution than the depths/rows above, simply because the object being imaged is farther away from the detector pixels. This means those pixels seeing deeper have a larger field of view and can receive scatter signals from depths deeper and shallower than what would be desired, resulting in detecting features in an object that are located at different depths than what that row of pixels is intended to see.

The depth resolution of a 2D backscatter system is not equivalent to the resolution set for each row in the detector array, it’s a few times larger; spanning multiple rows. This is an inherent drawback to single-sided imaging and requires careful attention when designing a system. It may be possible to overcome some of the resolution setbacks by using more rows and smaller pixels.
Figure 5-1. Images acquired in MCNP6 of a linear scanning method for 2D backscatter system. Each image is focused at a different depth in the object.
A new technique is proposed for 2D backscatter imaging that involves rotating the system instead of linearly scanning it across an object. During a linear scan, the object is being viewed from only one direction, which can cause shadows and streaks, like what was discussed in section 2.4 and was visible in Figures 2-4 and 2-5. One way to mitigate this in linear scanning 2D backscatter systems is to have two 2D detector arrays, one on each side of the fan-beam. Combining the signal from both sides would provide better count rates and help remove the potential streaking effects.

The idea of rotating the system instead would solve the same problem while still only needing one detector array. Although a second detector array can still be added; it can be used to either cut scan times in half or can help improve count rates by combining the signals from both detectors. By rotating the system around a center point, the features in an object can be imaged from different viewpoints and beam paths, creating a cleaner image with less artifacts that would be inherent in a linear scanning method.

For fair comparison, the same system design and test phantom from Chapter 5 were used. The number of scan points (36) were kept the same as well as the number of particle histories for the simulations (36.5 billion). In the rotating technique, each scan point is a different angle, to a maximum of 360°. 36 scan points means the system rotated in 10° increments, rather than the 5 mm steps described for the linear scan.

### 6.1 Image Reconstruction Model

Since a rotating technique collects signals of the same points in an object from multiple positions, image reconstruction algorithms are needed in order to correctly map all of the data back into a common 3D space that describes the object. The use of reconstruction algorithms is the hallmark of computed tomography (CT), which is widely used in medical imaging. However, all current methodologies are designed around
transmission imaging, where full 360° access is available. If image reconstruction is to be feasible for a single-sided backscatter system, then a novel algorithm must be developed.

6.1.1 Theory

Inspiration for a backscatter image reconstruction algorithm was taken from the analytical and iterative reconstruction algorithms used in single-photon emission computed tomography (SPECT) \[38\]. In SPECT, the objective is to measure the 3D source distribution inside of a patient that has been administered a gamma emitting pharmaceutical. The analogy to backscatter imaging is to measure the unknown, 3D distribution of photon scatter intensities inside the object being irradiated. In this sense, the attenuation of the primary beam path is being ignored and each pixel in the detector array is assumed to be measuring the intensity of photons being emitted from within the object, as if photon emitters were present inside.

The approach that was designed used the analytical method of ray-driven backprojection. The implementation of a discrete backprojection for backscatter imaging is given by:

\[
\tilde{b}(x, y, z) = \sum_{k=1}^{p} g(s_k, z, \theta_k)w_r(x, y, z, \theta_k)\Delta\theta
\] (6-1)

Where \( p \) is the number of projections acquired over \( 2\pi \) radians, \( \theta_k \) is the \( k \)th angular position of the detector, and \( \Delta\theta \) is the angular step between two successive projections \( (\Delta\theta = 2\pi/p) \). Let the imaged region of an object be thought of as a 3D matrix of voxels \( M(x, y, z) \). The goal is to find \( \tilde{b}(x, y, z) \), the result of the backprojection at voxel \( M(x, y, z) \). \( g(s_k, z, \theta_k) \) is the signal from the imaging pixel located in column \( s_k \) and row \( z \) in the 2D detector array, at projection angle \( \theta_k \), weighted by the path length \( w_r(x, y, z, \theta_k) \) of the ray traced through voxel \( M(x, y, z) \). The location \( s_k \) is a function of the projection angle \( \theta_k \) and the \((x, y)\) coordinates of the voxel \( M(x, y, z) \):

\[
s_k = x \cos \theta_k + y \sin \theta_k
\] (6-2)
The $z$ dimension is the same for both projection images $g(s_k, z, \theta_k)$ and voxel matrix $M(x, y, z)$. Another way of describing Equation 6-1 is for each angle $\theta_k$, using Equation 6-2, find the location $s_k$ (on the detector) that is the projection location of point $M(x, y, z)$. Add the value $g(s_k, z, \theta_k)$, weighted by its associated ray length $w_r(x, y, z, \theta_k)$, to the current value of point $M(x, y, z)$ (initial value should be 0). Repeat this process for all angles, and divide the sum by the number of projection angles.

For SPECT and other transmission tomographies, the number of projections is acquired over $\pi$ radians. Projections beyond $\pi$ do not provide new information, since they are symmetric to previous projections made at angles less than $\pi$. This is not the case for backscatter, where every angle provides new information because it’s rotating around a point on the same side of the object rather than rotating around the entire object.

An issue that arises from images by rotating a 2D detector array around a fixed axis on one side is the disparity of signal contribution to voxels farther away from the rotation axis and deeper within the object. Figure 6-1 shows an illustration of how rays traced from the scatter point in a fan-beam travel much farther distances if they’re located deeper in the object. This triangular shape of traced voxels means there’s more sampling of voxels closer to the surface then at the bottom. Likewise in the other dimension, voxels closer to the axis of rotation are sampled more than ones on the edge of the fan-beam. This uneven distribution of sampling is illustrated by Figures 6-2 and 6-3, for the top/surface of the voxel map and the bottom/deepest of the map, respectively. As more projection angles are added, more information is obtained and a better image can be formed. However, the information contained in these reconstructed images will always degrade the deeper and farther away it is from the axis of rotation. Furthermore, disparities in detector signal also occur in each projection image due to the divergent fan-beam source and photon attenuation in the medium.

In order to mitigate all these sources of image artifacts, a very similar approach that was discussed in section 4.3.2 is taken. For a linear scan, all that was necessary was to
take one scan of a bulk homogenous material and use that as a form of filter. When you divide your scanned image slices by the aforementioned filter, the effects of a divergent fan-beam source and photon attenuation are mitigated. Pixels that had lower signals near the edges and bottom of the scans will be given more weight since they’re being divided by smaller values in the filter (because signal was also lower in that image due to the same reasons of source divergence and attenuation). Any changes in density/material will stand out because it will have a noticeably different signal than what was in the scan of the bulk homogenous material of the filter image. We can use this same procedure for the rotational scan. All that is needed is one projection of a bulk homogenous material. One can then feed that projection to the reconstruction algorithm and it will construct an image as if the system rotated/scanned over a homogenous object. This image now contains those same artifacts due to fan-beam source divergence, attenuation, and the un-symmetric sampling from the reconstruction algorithm. This now can be used as a 3D filter just like the 2D one used in the linear scanning. By diving any reconstructed image by this 3D filter, all of the artifacts mentioned will be smoothed out and result in a much more balanced image. The division is simply taking every voxel in the 3D image and dividing it by the same corresponding voxel in the filter.

6.1.2 Python Image Reconstruction Module

The reconstruct module inside the backscatter2D Python toolkit, described in Chapter 4, was developed to handle the collection, reconstruction, and viewing of projection angles acquired during a rotational scan. Two main classes describe this module: SignalMap and VoxelMap.

The MCNP simulations are created and set up using the methods provided in the backscatter2D toolkit. In this scenario it’s easier to leave the system stationary and instead rotate the phantom geometry inside the input file. Once all of the MCNP simulations have produced their meshtally data for each scanning angle, they are collected and placed in a folder, with each meshtally file being relabeled to reflect which angle
it was taken at. The SignalMap class is essentially a container that organizes all of the projection data. Given the path to the aforementioned meshtally folder, it will read each meshtally data file and store it into a dictionary (called “map”), with its key value being the angle the projection was taken at.

The VoxelMap class is what develops the framework that allows for image reconstruction. It creates the size and dimensions of the voxel map that is to be reconstructed; the matrix $M(x, y, z)$ defined earlier. VoxelMap contains the ray tracing algorithm needed for weighting the signals in each voxel. To create a VoxelMap instance, it must be passed an instance of the BackscatterSystem that describes the system used to take the projection images. It also must be passed either an integer describing the desired number of angles to ray trace, or a SignalMap instance which contains a list of angles that the meshtallies were taken at (recommended approach).

With the information about the system dimensions and number of angles being imaged, VoxelMap steps through the ray tracing process. It stores all of the information in a dictionary called “map”. The keys for this dictionary are the projection angles, identical to that of the “map” dictionary in SignalMap. The values for each key is another dictionary. In this sub-dictionary, the key is a tuple describing the column and row location of a pixel in the 2D detector array of the backscatter system. The value for these keys are a list of voxels that the ray has passed through on its way from the scatter location within the object to its associated detector pixel. Each voxel in the list is a Voxel class containing the following attributes: x,y,z index of the voxel with respect to the voxel matrix $M(x, y, z)$ and the path length of the ray that was traced through it. The process for this map creation is as follows:

- Based on detector dimensions and maximum distance traveled by scattered photons, determine maximum dimensions of the voxel matrix $M(x, y, z)$.
- For an angle $(\theta_k)$ in the list of projection angles:
transform the radial distance \( s_k \) of the detector column location into \( x \) and \( y \) coordinates in the \( M(x, y, z) \) reference frame. Detector row location is already in \( z \) coordinate.

- For every \( x, y, z \) location of a scatter point in the voxel matrix \( M(x, y, z) \), trace the ray towards the associated detector pixel that images that scatter point:
  - record the coordinates and path lengths of the voxels being traversed by the ray.

A separate method inside reconstruct takes the created SignalMap and VoxelMap and turns them into the final \( \tilde{b}(x, y, z) \) matrix which is the resulting reconstructed image. Since the system is rotating around a fixed point, the final image is in a cylindrical shape but it is mapped to a 3D Cartesian space. This results in a lot of empty space (0 values) around the edges of the matrix, as can be seen in Figures 6-2 and 6-3. They are to be ignored and do not provide any information about the object that was imaged.

### 6.2 Imaging Results

The 2D backscatter system rotated around the center of the industrial phantom in 10° increments, for a total of 36 scan points. This was the same number of scan points used in the linear scanning method. Besides the normal scan, a separate image was reconstructed based on a solid concrete block. Since the block is uniform from all angles, only a single scan point is needed. The single scan is repeatedly stored in the SignalMap at the same angular positions as the actual scan of the true phantom. This enables the creation of a baseline image that can be used to filter the actual image and remove the reconstruction artifacts that occur.

Figure 6-4 shows the raw image of the phantom after it has been reconstructed and Figure 6-5 is the same image after it was divided by the baseline image developed from pure concrete. The original, raw image is heavily obscured by the scan’s over sampling near the center of the image, making it difficult to see anything. Once the image is filtered by the baseline concrete image, however, the images become more uniform and the letters U and F become much more visible.
The first image in Figure 6-5 is a relatively uniform signal, as it should be since that
detector row was expected to be viewing only concrete. However, the next depth/row
down, both letters become visible when only the letter U should be. Although the letter
F becomes more prominent at the third depth when it should be, since that’s where it
was placed in the phantom. The bleeding of scatter signal from features outside of a given
depth plane are due to a lack of depth resolution in the rows of detector pixels. The final
image depth in the reconstructed image clearly shows the lack of sampling at that depth,
as there are entire sections missing. More scan points are necessary in order to improve
the image at the deeper locations.

Figure 6-6 shows a visual comparison of the two scanning techniques, as well as the
slice of the phantom that was imaged at that each depth/row in the 2D detector array.
Overall, a rotating scheme shows the potential for improvement over the linear scanning
method. Even though both image scans suffered from high uncertainties in the MCNP6
tally data, smoother images were formed using the rotational scan; high noise from the
linear scans at deeper depths were somewhat suppressed in the rotational scan. With the
aid of computational image reconstruction, scan/processing times would be comparable to
linear scanning. The only drawback is the lack of information at the deepest scan depths
of the system. Rotational scans would require more scan points to improve the deepest
images while a linear scan theoretically wouldn’t need to.
Figure 6-1. Illustration of ray tracing performed for each row in the detector array (depth dimension of image). Fan-beam and pixel columns in detector array would be in and out of the page.
Figure 6-2. Illustration of image sampling disparities with increasing projection angles for the top detector row measuring near object surface.
Figure 6-3. Illustration of image sampling disparities with increasing projection angles for the bottom detector row measuring the deepest within object.
Figure 6-4. Raw reconstructed image of UF industrial phantom from MCNP6 simulations. No filtering.
Figure 6-5. Filtered reconstructed image of UF industrial phantom from MCNP6 simulations. Filtered using reconstructed image of solid concrete.
Figure 6-6. Comparison of linear (top row) and rotational (middle row) scanning methods. Bottom row shows an image of the MCNP6 phantom at the depth that was scanned by the backscatter system.
CHAPTER 7  
CONCLUSIONS AND FUTURE WORK

7.1 Summary and Conclusions  

Compton scatter imaging is a technique similar to that of transmission radiography except that it utilizes the inelastic scattering of photons to acquire spatial information about the object under investigation. X-ray backscatter imaging can be considered a subset of CSI, where X-rays are the photon source and the detectors are placed on the same side as the X-ray source. The ability to image the internal structure of objects while only needing access to one side has useful applications in non-destructive testing and the potential for use in medical imaging. Situations that XBI excels at are when the object being imaged is either too voluminous or access to its opposite side is impossible/impractical.

X-ray backscatter designs typically acquire only 2D images and previous attempts to go beyond that involved multiple scans. In this study, two new techniques were developed for imaging tomosynthesis using X-ray backscatter systems.

The first technique uses energy discrimination in the detectors. By binning the signals collected into different energy groups, depth sensitive information is obtained about the object, since the energy of an X-ray affects what depth they’re most likely to scatter at. The second technique developed was the use of fan-beam geometries and collimated 2D detector arrays, called the 2D backscatter system. Through proper calculation and design, the pixels in the 2D detector array can be collimated so that they see specific depths inside the object being scanned. Both linear scanning and rotational scanning, with the aid of reconstruction algorithms, were used to acquire depth sensitive images.

The development and analysis of these two techniques was made possible through the use of radiation transport simulations in MCNP6 and the extensive use of a Python toolkit that was developed. The toolkit allows for rapid development and testing of
2D backscatter imaging systems, as well as novel image reconstruction algorithms for rotational image scanning with that system.

Test cases were performed to validate the use of MCNP6 for designing new XBI techniques. Old experimental tests were modeled in MCNP6, simulated, and converted into images to be compared against. Considering the lack of information about certain geometric properties of the experimental set ups and X-ray spectra, the simulated images agreed quite well with the physical images. This suggests that MCNP6 is a valid tool to be used for the design and analysis of X-ray backscatter imaging technologies, offering a cheap alternative for creating proof of concepts before ultimately moving to prototyping of designs.

A parametric study was performed for photon energies and their influence on the depth sensitivity of backscatter images for organic and industrial materials. It created the beginnings of a database that can be developed to aid in the design and use of backscatter imaging for 3D interrogation. Overall, greater depth penetration was achieved in tissue than in aluminum but photons were only investigated up to an energy of 200 keV. Higher energies are expected to yield greater penetration in industrial materials.

Building off of the results from the parametric study, a proof of concept for the first tomosynthesis technique was successfully achieved. This technique was a direct improvement over an old study which proved X-ray backscatter imaging tomosynthesis by scanning an object multiple times with different X-ray tube voltages. The improved technique achieved the same success while only needing to scan the object once, by using spectral detectors and energy binning.

The second tomosynthesis technique, involving a fan-beam and 2D detector array, offers a faster scanning system than previous pencil-beam designs at the cost of imaging resolution. This technique requires precise use of collimation grids on the detector arrays. With extensive use of trigonometry, the Python toolkit is able to design either uniform or optimized collimation grids for these systems, reducing the effort needed by designers
of 2D backscatter systems. Both linear and rotational scanning were tested. Even though image reconstruction algorithms are necessary, the rotational scanning method created cleaner images than that of linear scanning.

Both techniques introduced in this work open up new territory for X-ray backscatter imaging. The methods developed can be applied to pre-existing systems or used to create new ones, impacting a wide range of non-destructive testing and non-invasive imaging.

7.2 Future Work

It is difficult to predetermine how an imaging system will handle new materials because of the complex nature in which a backscatter signal is collected. This is especially true when the object is highly heterogeneous and has many different materials. It is believed the next step to take in this work would be to image more complex, realistic objects. This could be either industrial or medical related. It would also be important to start directly comparing these simulation results with an existing backscatter system, in order to truly validate the capabilities of MCNP6 to model backscatter imaging and help with designing new systems.

The depth resolution of a backscatter imaging system utilizing the spectral detection technique is difficult to define because it is dependent upon many factors and would need to be investigated with more parametric studies. Factors that influence depth resolution include: energy resolution of the detector being used, size and placement of the detectors, collimation (if any) used on the detectors, x-ray energy spectra used, resolution/dimension of the pencil beam used, and the materials being imaged. This is a significant amount of variables to consider and may be best suited to solve for a specific system that is already physically built, rather than backscatter imaging as a whole.

Only the simplest of reconstruction algorithms have been developed for the rotating 2D backscatter system: a ray-driven backprojection method. Other analytical methods, such as filtered backprojection, as well as more advanced iterative reconstruction methods, may improve the imaging capabilities of this rotational technique. Although it is currently
unknown if some of these methods can be effectively translated to a system that rotates about an axis on one side of an object.

The long runtimes and poor image quality of fan-beam systems in MCNP6 necessitates that some form of variance reduction should be incorporated into the simulations. For pencil-beam geometries, simulations were ran in a reasonable amount of time, due to the larger surface area/solid angle of the detectors to acquire more signal. Translating the same mentality to a fan-beam design did not succeed. The solid angle of the pixels in the detector array were too small when trying to achieve mm resolutions, making it difficult to acquire enough signal in a reasonable time frame. Both cell importance and weight windowing are two possible forms of variance reduction that may speed up fan-beam simulations in MCNP6. The one issue that may arise is that the backscatter system shifts for each simulation, as it’s scanning the phantom. Therefore, variance reduction set up to reduce the run time of one simulation may not be as effective for the other simulations. Potentially this can be incorporated into the Python toolkit.
APPENDIX

2D BACKSCATTER PYTHON TOOLKIT

The code for each module in the backscatter2D package is transcribed in each section listed here. Each module was written as a separate file contained within the backscatter2D package. In order to work as a whole, an __init__.py file also needs to be present within the folder containing all of the modules.

A.1 system_design.py

Contains all the classes and functions necessary to develop an X-ray Backscatter System that utilizes a fan beam X-ray source, a 2D pixelated detector array, and a collimation grid that rests on the face of the detector array. This allows for 2D image acquisition for a single position of the backscatter system. By scanning the system in one dimension, it’s possible to develop a 3D image of an object.

```
import numpy as np

class BackscatterSystem(object):
    
    Object that is a composition of all components in the backscatter system:

    1) X-Ray Fan Beam source
    2) 2D Pixelated detector array
    3) Collimation grid for detector array

    Parameters Needed from User:
    ======

    step_size : distance taken by each step in the X dimension as the system scans over an object (cm)

    Default Values:
```
step_size : None

Parameters:

depths : A dictionary containing useful information on the system. Gets created after calling the system_check method. Each Key contains a list that’s as long as the number of pixels in the height of the detector.

Keys:

pix_center : depth position (cm) determined by tracing a line from the center of a pixel to where it intersects with the fan beam. This line is orthogonal to the surface of the pixel. A negative value indicates it is above the surface of the object being imaged.

full_pix_range : tuple (cm), containing the (deepest depth, shallowest depth) that the pixel has full sight of. Full sight meaning any photons that scatter in this range can pass through any part of the pixel’s face. A negative value indicates it is above the surface of the object being imaged.

range : tuples (cm), containing the (deepest depth, shallowest depth)
that the pixel can possibly see. So this range is always greater than the full_pix_range. It includes areas of the fan beam where photon scatters are only partially visible to the pixel face. A negative value indicates it is above the surface of the object being imaged.

\[
y_{\text{res}} : \text{resolution in the y dimension (cm).}
\]

Determined by using the path length of a photon scattering from the fan beam and going straight to the center of the pixel. This path length is the length of the line traced when calculating \( \text{pix\_center} \)

You have the option to fully build a backscatter system from this object or you can build as many components as you want separately and then add them to an instance of this object.

Examples:

1) Build from scratch with this object:

```python
import system_design as sysd
your_system = sysd.BackscatterSystem()
```

2) Build a detector separately and add it to your system, and build the remaining components during instantiation of your system:

```python
import system_design as sysd
your_detector = sysd.Detector(your_settings)
```
your_system = sysd.BackscatterSystem(your_detector)

3) Build all components beforehand and then put in a system

```python
import system_design as sysd
your_xray = sysd.XRayFanBeam(your_xray_settings)
your_detector = sysd.Detector(your_detector_settings)
your_collimator = sysd.Collimator(your_collimator_settings)
your_system = sysd.BackscatterSystem(your_xray, your_detector, your_collimator)
```

```python
def __init__(self, beam_input=None, det_input=None, collimator_input=None, step_size=None):
    self.step_size = step_size

    # Load or set up X-ray Fan Beam
    if isinstance(beam_input, XRayFanBeam):
        self.XRayFanBeam = beam_input
    else:
        responded = False
        print "X-Ray Fan Beam source doesn’t exist, would you like to" +
        " build one?"
        print "Options: yes, no, use defaults \n"
        response = raw_input('>> ')
        while not responded:
            if response == 'y' or response == 'yes':
                self.XRayFanBeam = XRayFanBeam(build_manually=True)
                responded = True
                print "\nX-Ray Fan Beam source doesn’t exist, would you like to build one?"
                print "Options: yes, no, use defaults \n"
```
elif response == 'n' or response == 'no':
    print "\nYou can add an X-Ray Fan Beam source later by" +\
    " creating an XRayFanBeam instance and" +\
    " assigning it to your BackscatterSystem instance."
    self.XRayFanBeam = None
    responded = True
elif response == 'use defaults':
    print "\nUsing defaults established in XRayFanBeam" +\
    " Class."
    print "See XRayFanBeam documentation to learn more."
    self.XRayFanBeam = XRayFanBeam()
    responded = True
else:
    print "\nYour response was not a valid option."
    print "Options: yes, no, use defaults \n"
    response = raw_input('>> ')
while not responded:
    if response == 'y' or response == 'yes':
        self.Detector = Detector(build_manually = True)
        responded = True
    elif response == 'n' or response == 'no':
        print "\nYou can add a 2D detector array later by" +\
        " creating a Detector instance and" +\
        " assigning it to your BackscatterSystem instance."
        self.Detector = None
        responded = True
    elif response == 'use defaults':
        print "\nUsing defaults established in Detector" +\
        " Class."
        print "See Detector documentation to learn more."
        self.Detector = Detector()
        responded = True
    else:
        print "\nYour response was not a valid option."
        print "Options: yes, no, use defaults \n"
        response = raw_input('>> ')

# Load or set up Collimation Grid
# ==============

if isinstance(collimator_input, Collimator):
    self.Collimator = collimator_input
    if self.step_size is None:
        responded = False

96
print "Do you want system step size to match "
print "Collimator Y resolution?"
print "Options: yes, no"
response = raw_input('>> ')
while not responded:
    if response == 'y' or response == 'yes':
        self.step_size = collimator_input.y_resolution
        responded = True
    elif response == 'n' or response == 'no':
        print "Enter a step size: "
        print "(in centimeters)"
        self.step_size = get_value_from_user('float', float)
        responded = True
    else:
        print "Your response was not a valid option."
        print "Options: yes, no"
        response = raw_input('>> ')

if self.Detector is not None and 
    self.Collimator.starting_point is None:
    responded = False
    print "Since there is a Detector present, do you want"
    print "to construct the collimation grid?"
    print "Options: yes, no"
    response = raw_input('>> ')
    while not responded:
        if response == 'y' or response == 'yes':
            self.Collimator.construct_grid(self.Detector)
responded = True

elif response == 'n' or response == 'no':
    responded = True
else:
    print '\nYour response was not a valid option."
    print "Options: yes, no\n"
    response = raw_input('>> ')

elif isinstance(collimator_input, UniformCollimator):
    self.Collimator = collimator_input
else:
    responded = False
    print "\nDetector Collimation grid doesn’t exist, " +\n    "would you like to build one? "
    print "Options: yes, no, use defaults \n"
    response = raw_input('>> ')
while not responded:
    if response == 'y' or response == 'yes':
        print "\nBuild a Uniform or Optimized COLLimator?"
        print "Options: u (uniform), o (optimized) \n"
        coll_type = raw_input('>> ')
        if coll_type == 'o' or coll_type == 'optimized':
            self.Collimator = Collimator(build_manually = True)
        if coll_type == 'u' or coll_type == 'uniform':
            self.Collimator = UniformCollimator(build_manually = \            True)
    responded = True
    if self.Detector is not None and (coll_type == 'o' or \
coll_type == 'optimized'):

responded = False

print "\nSince there is a Detector present, do you want"
print "to construct the collimation grid?"
print "Options: yes, no\n"
response = raw_input('>> ')

while not responded:
    if response == 'y' or response == 'yes':
        self.Collimator.construct_grid(self.Detector)
        responded = True
    elif response == 'n' or response == 'no':
        responded = True
    else:
        print "\nYour response was not a valid option."
        print "Options: yes, no\n"
        response = raw_input('>> ')

elif response == 'n' or response == 'no':
    print "\nYou can add a detector collimation grid later " +\
    "by creating a Collimator instance and " +\
    "assigning it to your BackscatterSystem instance."
    self.Collimator = None
    responded = True

elif response == 'use defaults':
    print "\nUsing defaults established in Collimator Class"
    print "See Collimator documentation to learn more."
    self.Collimator = Collimator()
    responded = True
if self.Detector is not None:
    responded = False
    print "Since there is a Detector present, do you want"
    print "to construct the collimation grid?"
    print "Options: yes, no\n"
    response = raw_input('>>
')
    while not responded:
        if response == 'y' or response == 'yes':
            self.Collimator.construct_grid(self.Detector)
            responded = True
        elif response == 'n' or response == 'no':
            responded = True
        else:
            print "Your response was not a valid option."
            print "Options: yes, no\n"
            response = raw_input('>>
')
    else:
        print "Your response was not a valid option."
        print "Options: yes, no, use defaults \n"
        response = raw_input('>>
')

def system_check(self):
    ""
    Analyzes the system and calculates the following features:
    
    depths : A dictionary containing useful information on the
    system. Gets created after calling the system_check
    method. Each Key contains a list that's as long as the
number of pixels in the height of the detector.

Keys:

pix_center : depth position (cm) determined by tracing a line from the center of a pixel to where it intersects with the fan beam. This line is orthogonal to the surface of the pixel. A negative value indicates it is above the surface of the object being imaged.

full_pix_range : tuples (cm), containing the (deepest depth, shallowest depth) that the pixel has full sight of. Full sight meaning any photons that scatter in this range can pass through any part of the pixel’s face. A negative value indicates it is above the surface of the object being imaged.

range : tuples (cm), containing the (deepest depth, shallowest depth) that the pixel can possibly see. So this range is always greater than the full_pix_range. It includes areas of the fan beam where photon scatters are only partially visible to the pixel face. A negative value indicates it is above the surface of the object being imaged.
the object being imaged.

\[ y_{res} \] : resolution in the y dimension (cm).
Determined by using the path length of a photon scattering from the fan beam and going straight to the center of the pixel. This path length is the length of the line traced when calculating \( \text{pix\_center} \)

```
self.depths = {'pix_center' : [], 'full_pix_range' : [],
              'range' : [], 'y_res' : []}

pix_height = np.array([-self.Detector.pixel_height*
                       np.cos(self.Detector.angle_radians),
                      0 ,
                      self.Detector.pixel_height*
                       np.sin(self.Detector.angle_radians)
                      ])

pix_width = self.Detector.pixel_width - \\
            self.Collimator.collimator_thickness

y_index = self.Detector.num_pixel_W + 1

for i in range(self.Detector.num_pixel_H):
    # Top of Pixel
    Xp,Yp,Zp = self.Collimator.z_fins[i+1].v_vector
    # Bottom of Pixel
    xp,yp,zp = self.Collimator.z_fins[i].v_vector + \\
               self.Collimator.z_fins[i].a_vector
    # Center of Pixel
xcp, ycp, zcp = self.Collimator.z_fins[i].v_vector + 
    self.Collimator.z_fins[i].a_vector/2. + 
    pix_height/2.
# Collimator above pixel
Xc, Yc, Zc = self.Collimator.z_fins[i+1].v_vector +
    self.Collimator.z_fins[i].c_vector
# Collimator below pixel
xc, yc, zc = self.Collimator.z_fins[i].v_vector + \
    self.Collimator.z_fins[i].a_vector + \
    self.Collimator.z_fins[i].c_vector
# Calculate pix_center
d = xcp/np.tan(self.Detector.angle_radians) - zcp
path_length = xcp/np.sin(self.Detector.angle_radians)
self.depths["pix_center"].append(d)
# Calculate full_pix_range
lower_limit = -find_intercept((xp, zp), (xc, zc))
upper_limit = -find_intercept((Xp, Zp), (Xc, Zc))
self.depths["full_pix_range"].append((lower_limit, upper_limit))
# Calculate range
lower_limit = -find_intercept((Xp, Zp), (xc, zc))
upper_limit = -find_intercept((xp, zp), (Xc, Zc))
self.depths["range"].append((lower_limit, upper_limit))
# Calculate y_res
length = self.Collimator.y_fins[y_index*i].length
y_res = pix_width*(path_length - (length/2.))/(length/2.)
self.depths["y_res"].append(y_res)
def optimize_fan_beam(self, fan_angle=True):
self.XRayFanBeam.optimize_with_detector(self.Detector, fan_angle)

def optimize_collimator(self):
    self.Collimator.construct_grid(self.Detector)

class XRayFanBeam(object):
    
    Parameters Needed from User:
    
    ================
    
    energy : either pass it a single value or a file path containing
    an X-ray energy distribution. It will then store it
    as an array. See example below
    (units: MeV)
    
    surface_offset : how high up (+Z) the source is positioned, where
    x-rays are emitted
    
    fan_angle : angle between vector pointing directly down (-Z) and
    vector pointing along edge of fan
    so fan_angle = total fan spread / 2
    See figure below
    (units: degrees)
    
    fan_angle = 0 is a pencil beam
    
    Example energy spectrum:
    
    energy = array([[ 1.00000000e-02,  1.24156743e+01],
                   [ 1.10000000e-02,  1.04968200e+02],
                   [ 1.20000000e-02,  3.89738325e+02]])
    
    The first column contains the energy values in MeV
    The second column contains the photon intensity or frequency
    The second column is what gets normalized and placed inside the
normalized_energy attribute

Fan Beam Figure:

Point Source

X

/|\|
/ | \ \
/ | __<---- Fan Angle
/ | \ 
/ | \ <- Edge of Fan Beam
/ | \
/ | \\ \
/ | \\ \
/ | \\ \
/ | \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \",

Default Values:

=============

Energy : 0.075 MeV
surface_offset : 16. cm
fan_angle : 43.0 degrees

Attributes:

---------

normalized_energy : If the energy attribute is an array containing an energy spectrum, then there will also be a copy of that spectrum that has been normalized. This is necessary for importing the spectrum into MCNP
def __init__(self, energy = 0.075, surface_offset = 16.0,
    fan_angle = 43.0, build_manually = False):
    if build_manually == True:
        responded = False
        print '
Building fan beam manually'
        print 'Are you importing the X-Ray energy spectra from a file?'
        print "Options: yes, no\n"
        response = raw_input('>> ')
        while not responded:
            if response == 'y' or response == 'yes':
                print "\nEnter file path\n"
                file_path = raw_input('>> ')
                self.energy = convert_file_to_array(file_path)
                responded = True
            elif response == 'n' or response == 'no':
                print "\nEnter an X-ray energy: "
                print "(in units of MeV)"
                self.energy = get_value_from_user('float', float)
                responded = True
            else:
                print "\nYour response was not a valid option."
                print "Options: yes, no\n"
                response = raw_input('>> ')
        print "\nSurface offset of X-ray source point"
        print "(in centimeters)"
        self.surface_offset = get_value_from_user('float', float)
        print "\nAngle of the fan beam (the angle between the edge of"
print "the beam and the -Z direction)"
print "(in degrees)"
self.fan_angle = get_value_from_user('float', float)
else:
    if isinstance(energy, str):
        self.energy = convert_file_to_array(energy)
    else:
        self.energy = energy
self.surface_offset = surface_offset
self.fan_angle = fan_angle
if isinstance(self.energy , np.ndarray):
    self.normalized_energy = normalize_spectrum(self.energy)
    self.__get_spectra_stats()
else:
    self.normalized_energy = None

def optimize_with_detector(self, detector, opt_fan_angle=True):
    # if not isinstance(detector, Detector):
    #     print "Error: Did not pass an instance of system_design.Detector"
    #     return
    if opt_fan_angle:
        fan_angle = np.arctan((detector.width/2.)/self.surface_offset)
        self.fan_angle = fan_angle*180/np.pi
    else:
        self.surface_offset = detector.width/(2.*
            np.tan(self.fan_angle*np.pi/180))
```python
def __get_spectra_stats(self):
    # Find the mean energy of spectra
    self.mean_energy = np.sum([energy*prob for energy, prob in self.normalized_energy])
    # Find the mode energy of spectra (most common energy)
    row, col = np.where(self.normalized_energy == self.normalized_energy[:,1].max())
    self.mode_energy = self.normalized_energy[row[0],0]
    # Find the median energy of spectra (energy where cumulative density function is .5)
    cumulative_prob = 0
    for index, prob in enumerate(self.normalized_energy[:,1]):
        if cumulative_prob >= 0.5:
            self.median_energy = self.normalized_energy[index,0]
            break
        else:
            cumulative_prob += prob
```

class Detector(object):
    '''
    Object that contains all the necessary information to build a
    2D pixelated detector array.
    The "height" of the detector determines the resolution and number
    of pixels seeing into the object (Z dimension) being imaged.
    The "width" of the detector determines the resolution and number
    of pixels seeing accross the object (Y dimension) being imaged.
    The system therefore mechanically moves in the X dimension, scanning
    '''
```
over the object to form a 3D image.

2D Detector Array

| | | | | | | |
| |___|___|___|___|___|___|___|
|--|--|--|--|--|--|--|

height (num_pixel_H)

| | | | | | | |
| |___|___|___|___|___|___|___|

| | | | | | | |
|___|___|___|___|___|___|___|___|

<-------- width -------->

(num_pixel_W)

Individual Pixel

| |
| |
|--|--|--|--|--|--|--|

pixel

height (num_pixel_H)

| |
| ___ |

pixel

width

Parameters Needed from User:

num_pixel_H : number of pixels in the height dimension of detector array

num_pixel_W : number of pixels in the width dimension of detector array
pixel_height : height of a single detector pixel (units: cm)
pixel_width : width of a single detector pixel (units: cm)
age : angle of detector array from object surface

EX: 0 degrees means detector is parallel to surface
    90 degrees means detector is parallel to fan beam

(units: degrees)

Default Values:

num_pixel_H : 8 pixels
num_pixel_W : 30 pixels
pixel_height : 1.0 cm
pixel_width : 1.0 cm
angle : 40.0 degrees

Attributes:

height : total height of the detector (units: cm)
width : total width of the detector (units: cm)
age_rad : angle of the detector array in radians (units: radians)
thickness : how thick the detector is. This is held constant and
            should be changed with caution, since it will affect
            Tallying (image results) in the MCNP simulations.
            (units: cm)

The following attributes are modeled after the BOX Macrobody used in
MCNP6. See MCNP manual for more information on Macrobodies.

See Collimator.construct_grid or Fin documentation for more information
on these vectors and MCNP format.

v_vector : vector pointing from the origin to the corner of the
detector. This is helpful for plotting and MCNP input
generation (units: cm)

\[ \text{a\_vector} \] : vector pointing from \( v\_vector \) to a corner of the
detector, defining its thickness. This is helpful for
plotting and MCNP input generation (units: cm)

\[ \text{b\_vector} \] : vector pointing from \( v\_vector \) to a corner of the
detector, defining its width. This is helpful for
plotting and MCNP input generation (units: cm)

\[ \text{c\_vector} \] : vector pointing from \( v\_vector \) to a corner of the
detector, defining its height (length). This is helpful
for plotting and MCNP input generation (units: cm)

```python
def __init__(self, num_pixel_H = 8 , num_pixel_W = 30 ,
pixel_height = 1.0 , pixel_width = 1.0 ,
angle = 40.0 , thickness = 0.35 ,
build_manually = False):
    if build_manually == True:
        print "\nBuilding 2D Detector array manually"
        print "Number of pixels in the height dimension"
        self.num_pixel_H = get_value_from_user('integer', int)
        print "\nNumber of pixels in the width dimension"
        self.num_pixel_W = get_value_from_user('integer', int)
        print "\nHeight of a single pixel in the detector array"
        print "(in centimeters)"
        self.pixel_height = get_value_from_user('float', float)
        print "\nWidth of a single pixel in the detector array"
        print "(in centimeters)"
```
self.pixel_width = get_value_from_user('float', float)
print "\nAngle of detector array from object surface"
print "0 degrees means detector is parallel to object surface"
print "90 degrees means detector is parallel to fan beam"
print "Recommended values are between 50 and 70 degrees"
print "(in degrees)"
self.angle = get_value_from_user('float', float)
print "\nDetector thickness"
print "(in centimeters)"
self.thickness = get_value_from_user('float', float)

else:
    self.num_pixel_H = num_pixel_H
    self.num_pixel_W = num_pixel_W
    self.pixel_height = pixel_height
    self.pixel_width = pixel_width
    self.angle = angle
    self.thickness = thickness

self.height = self.num_pixel_H * self.pixel_height
self.width = self.num_pixel_W * self.pixel_width
self.angle_radians = self.angle *np.pi/180

self.v_vector = 0
self.a_vector = 0
self.b_vector = 0
self.c_vector = 0

def create_detector_vectors(self, starting_point, collimator_thickness):
    collimator_thickness_vec = np.array([
-collimator_thickness/2. * np.cos(self.angle_radians),
collimator_thickness/2. ,
collimator_thickness/2. * np.sin(self.angle_radians)
])

self.v_vector = starting_point + collimator_thickness_vec
self.a_vector = np.array([[self.thickness * np.sin(self.angle_radians),
                          0 ,
                          self.thickness * np.cos(self.angle_radians)]
])

self.b_vector = np.array([[ 0, self.width, 0]])
self.c_vector = np.array([[-self.height * np.cos(self.angle_radians),
                          0 ,
                          self.height * np.sin(self.angle_radians)]
])

class Collimator(object):
    
    Object that contains information and modules necessary to build a
collimation grid to accompany the 2D pixelated detector array.
In order to construct the grid, a fully built Detector object
must be passed to the appropriate modules. Design data located
inside the Detector class are necessary to correctly design the
collimators.
The grid is broken into two categories: Z collimators and Y collimators
Z collimators are what provide depth discrimination (Z dimension in this
framework) and are placed along the height of the detector, looking like
fins (thin rectangular parallelepiped) extending in the Y dimension.
They blind a detector pixel from X-ray backscatter signals generated at a specific depth in the object and deeper. Therefore the pixels only receive signal from positions inside the object above that specified depth. There are as many Z collimators as there are pixels in the "height" dimension of the detector, one for each row. Y collimators are what provide the resolution and discrimination in signal across the object being imaged (Y dimension in this framework) and are placed along the width of the detector, looking like fins (thin rectangular parallelepipeds) extending in the Z dimension. They blind a detector pixel from X-ray backscatter signals generated from other areas of the object that are supposed to be received by a pixel closer to that position of scatter. These X-rays that scatter at wrong angles, causing them to contribute signal to the wrong pixel, contribute to the reduction in image resolution in the XY plane. There are as many Y collimators as there are pixels in the "width" dimension of the detector plus one. See Detector class documentation for diagrams and explanations of detector "height" and "width".

When constructing the Z collimators, the lowest Z collimator is designed based around the maximum depth (max_depth) input. From there, each collimator above is designed to see an increment above that maximum depth. The increment is uniform across all depths and is calculated based on the max depth, the number of pixels in the height dimension of the detector array, and the height of a single pixel.

Parameters Needed from User:

```
collimator_thickness : thickness of the detector collimators
```
max_depth : maximum depth the detector can see inside the object (units: cm)
surface_offset : closest point/distance the system can be to the object (units: cm)
beam_offset : closest point/distance the system can be to the fanbeam (units: cm)
y_resolution : the resolution of the pixels in the Y dimension. Must be larger than the detector pixel_width, since that is the limiting resolution (units: cm)
material : the type of material that the collimator is made out of.

Default Values:
=*
collimator_thickness : 0.2 cm (2 mm)
max_depth : 8.0 cm
surface_offset : 2.0 cm
beam_offset : 2.0 cm
y_resolution : 1.2 cm
material : 'lead'

Attributes:
********
z_fins : List containing each fin in the height direction. Length of list will be equal to the number of pixels in the height dimension of the detector plus one.
y_fins : List containing each fin in the width direction.
Length of list will be equal to the number of pixels in the width dimension of the detector plus one, multiplied by the number of pixels in the height dimension of the detector.

depths : List containing the points at which depth each z.fin can stop seeing at. First entry is always max_depth. (units: cm)

starting_point : Vector pointing to the start point when creating the list of fins for the collimation grid (units: cm)

paths : List containing the minimum path lengths. Each path length is traced from each depth position to each corresponding detector pixel in the height dimension (so the list contains as many elements as there are pixels in the height dimension). (units: cm)

max_z_length : Longest fin in the list of Z collimation fins (units: cm)

max_y_length : Longest fin in the list of Y collimation fins (units: cm)

```python
def __init__(self, collimator_thickness = 0.2 , max_depth = 8.0 ,
             surface_offset = 2.0 , beam_offset = 2.0 ,
             y_resolution = 1.2 , material = 'lead' ,
             build_manually = False):
    if build_manually == True:
        print "\nBuilding collimator manually"
print "Thicknes of the collimation grid"
print "(be mindful that the thickness of the grid will be blocking"
print "that much of each pixel face in the detector array)"
print "(in centimeters)"
self.collimator_thickness = get_value_from_user('float', float)
print "\nMaximum depth the detector will see inside the object"
print "(in centimeters)"
self.max_depth = get_value_from_user('float', float)
print "\nClosest point/distance the system can be to the object"
print "(offset distance of detector from surface of object)"
print "(in centimeters)"
self.surface_offset = get_value_from_user('float', float)
print "\nClosest point/distance the system can be to the" 
print "X-ray fan beam source (offset distance of detector" 
print "from X-ray fan beam plane)"
print "(in centimeters)"
self.beam_offset = get_value_from_user('float', float)
print "\nPixel resolution of your images in the Y dimension."
print "This must be larger than the value you set for your"
print "Detector pixel width. If there’s any problems during"
print "collimation grid construction, it will prompt you to"
print "make appropriate changes."
print "(in centimeters)"
self.y_resolution = get_value_from_user('float', float)
print "\nMaterial the collimator is made out of"
self.material = raw_input('\n>> ') 
# These attributes are filled using methods in this class
self.z_fins = []
self.y_fins = []
self.depths = []
self.starting_point = None
self.paths = []
self.max_z_length = 0
self.max_y_length = 0
else:
    self.collimator_thickness = collimator_thickness
    self.max_depth = max_depth
    self.surface_offset = surface_offset
    self.beam_offset = beam_offset
    self.y_resolution = y_resolution
    self.material = material
    self.z_fins = []
    self.y_fins = []
    self.depths = []
    self.starting_point = None
    self.paths = []
    self.max_z_length = 0
    self.max_y_length = 0

def construct_grid(self, detector, depths = None):
    
    Pass the function a list of desired depths if you don’t want it to automatically calculate the depths based on max_depth and the number of pixels in the height dimension of detector array.
fins contain data in a format that follows that of
MCNP’s BOX Macrobody (See MCNP Manual for more details)

```
BOX v_x v_y v_z a_x a_y a_z b_x b_y b_z c_x c_y c_z
```

where the v vector points to a corner of the box (from the origin).
In this situation, a box is one of the fins that form the collimation grid.

vectors a, b, and c each are orthogonal to eachother and point to
the other corners of the box FROM THE STARTING POINT OF THE CORNER
ESTABLISHED BY VECTOR v. To say it another way, for these the vectors
the origin is located at the corner of the fin pointed at by vector v.
z_fins will be a list containing each fin in the height direction.
Length of list will be equal to the number of pixels in the height
dimension of the detector plus one.
y_fins will be a list containing each fin in the width direction.
Length of list will be equal to the number of pixels in the width
dimension of the detector plus one, multiplied by the number of
pixels in the height dimension of the detector.

a single fin is an object of the Fin class containing
those 4 arrays which correspond to the vectors mentioned above, as well
as other data, such as fin length, width, and thickness.
See Fin documentation for more information.

Example:

```
z_fins = [fin, fin, fin, fin, fin, ....]
```

```python
# Check that Collimator resolution and Detector Pixel agree
# =============================================================
```
```python
while self.y_resolution <= (detector.pixel_width - \
    self.collimator_thickness):
    responded = False
    print "\nPixel resolution in the Y dimension is less than" +\
    " or equal to detector pixel width."
    print "Would you like to increase y_resolution (poorer " +\
    "image resolution) or decrease detector"
    print "pixel width? "
    print "Options: increase, decrease, quit \n"
    response = raw_input('>>  ')
    while not responded:
        if response == 'i' or response == 'increase':
            print "\nEnter new Y resolution:"
            self.y_resolution = get_value_from_user('float', float)
            responded = True
        elif response == 'd' or response == 'decrease':
            print "\nEnter new detector pixel width:"
            detector.pixel_width = get_value_from_user('float',
                float)
            responded = True
        elif response == 'q' or response == 'quit':
            return
        else:
            print "\nYour response was not a valid option."
            print "Options: increase, decrease, quit \n"
            response = raw_input('>>  ')
# ==================================================================
```
# Clear the lists and variables
# -=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-
self.z_fins = []
self.y_fins = []
self.depths = []

# If desired depths were given, update the max_depth of system
if depths:
    self.depths = depths
    self.max_depth = depths[0]

depth_pos = self.max_depth
# -=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-

# Fill the self.starting_point attribute
# -=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-
self.find_starting_point(detector)
# -=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-

# this vector will be used to increment the starting point (v vector)
# to point to each fin needed in the height dimension.
# It is determined by pixel height and detector angle.
# Called increment_xz because it only affects the vector in the
# x and z dimension.
# -=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-
increment_xz = np.array(
    [-detector.pixel_height*np.cos(detector.angle_radians),
     0,
     detector.pixel_height*np.sin(detector.angle_radians)])
# -=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-

# this vector will be used to increment the starting point (v vector)
# to point to each fin needed in the width dimension.
# It is determined by pixel width. Called increment_y because it
# only affects the vector in the y dimension.

# ==============================================================
increment_y = np.array([0, detector.pixel_width, 0])
# ==============================================================

# This defines the resolution in the Z (depth) dimension for the
# system. Each iteration will increment closer to the surface of
# the object being imaged. This is the default increment if no
# desired depths were provided.

# ==============================================================
if not depths:
    increment_depth = float(self.max_depth) / detector.num_pixel_H
# ==============================================================

# Start iterating over all the fins

# ==============================================================
for i in range(detector.num_pixel_H + 1):
    if depths:
        depth_pos = depths[i]
        # Update the position for the starting point of a Z fin
        current_point_xz = self.starting_point + i * increment_xz
        next_point_xz = current_point_xz + increment_xz
        # Create a Z fin and append it to the list
        self.z_fins.append(self.create_z_fin(detector, i+1,
                                              current_point_xz,
                                              next_point_xz, depth_pos))
    # ==============================================================
# After the first iteration, start creating the y fins too
# ==============================================================

if i>0:
    if depths:
        increment_depth = depths[i-1] - depths[i]

    # Find center coordinate of current pixel
    X_center, Y, Z_center = self.find_pixel_center(
        current_point_xz,
        detector.pixel_height,
        detector.angle_radians)

    # Find the distance from depth position to
    # current pixel center
    path_length = np.sqrt( (X_center)**2 + \ 
        (Z_center + depth_pos + increment_depth/2.)**2 )

    # Start iterating over y fins
    for j in range(detector.num_pixel_W + 1):
        current_point_xyz = current_point_xz + j*increment_y
        self.y_fins.append( self.create_y_fin( detector,
            (i-1)*(detector.num_pixel_W+1) + (j + 1),
            current_point_xyz,
            path_length
        )

    if not depths:
        self.depths.append(depth_pos)
        depth_pos -= increment_depth

self.max_z_length = np.max([fin.length for fin in self.z_fins])
self.max_y_length = np.max([fin.length for fin in self.y_fins])

def find_starting_point(self, detector):
    '''
    finds vector starting point for the construction of the collimation
    grid
    Heavy use of trigonometry
    Determines whether Z or Y dimension is limiting factor and then
    creates the proper staring point that makes sure collimators won’t
cross in front of the x-ray beam or touch the object surface
    Also creates the vectors that define the dimensions of the Detector
    instance being used.
    '''

    # ==============================================================
    # First calculate max collimator length for Z (Lz)
    # ==============================================================
    # The pixels closest to the surface will always have the longest path
    # and will need the longest collimator, so no need to check every
    # path (I think... maybe one day this will cause problems)
    phi = np.arctan( (detector.height*np.cos(detector.angle_radians) + \
                      self.beam_offset) / (self.max_depth + \
                      self.surface_offset) )
    eta = detector.angle_radians - phi
    Lz = (detector.pixel_height - self.collimator_thickness)/np.tan(eta)

    # ==============================================================
    # Next calculate max collimator length for Y (Ly)
    # ==============================================================

124
self.calculate_minimum_path_lengths(detector)

index = self.paths.index(np.max(self.paths))

max_p_min = self.paths[index]

Ly = 2*max_p_min / (self.y_resolution/detector.pixel_width - 1)

# Determine which is larger and then create the starting point
L = np.max([Lz, Ly])

start_x = self.beam_offset + \
    detector.height * np.cos(detector.angle_radians) + \
    L * np.sin(detector.angle_radians) + \
    self.collimator_thickness * np.cos(detector.angle_radians)

start_y = -(detector.width + self.collimator_thickness)/2.

start_z = self.surface_offset + \
    L * np.cos(detector.angle_radians) - \
    self.collimator_thickness * np.sin(detector.angle_radians)

self.starting_point = np.array([start_x, start_y, start_z])

detector.create_detector_vectors(self.starting_point, 
                                 self.collimator_thickness)

def create_z_fin(self, detector, index, position, pixel_height_vec, depth):
    # y_hold just grabs the value but is not used
    pix_x, y_hold, pix_z = pixel_height_vec

    # =================================================================
    # Instantiate a Fin object and assign values to its attributes
    # =================================================================
    fin = Fin('Z', index, self.collimator_thickness)
    fin.width = detector.width + self.collimator_thickness

    # Fin starting position (v_vector)

fin.v_vector = position  
# Fin thickness vector (a_vector)
fin.a_vector = np.array([  
    -self.collimator_thickness*np.cos(detector.angle_radians),
    0,
    self.collimator_thickness*np.sin(detector.angle_radians)
])
# Fin width vector (b_vector)
fin.b_vector = np.array([ 0,
    detector.width + self.collimator_thickness,
    0])

# 3 Steps to solve for fin length
phi = np.arctan( pix_x / (depth + pix_z) )
eta = detector.angle_radians - phi
L = (detector.pixel_height - self.collimator_thickness)/np.tan(eta)
# Fin length vector (c_vector)
fin.length = L
fin.c_vector = np.array([ -L*np.sin(detector.angle_radians),
    0,
    -L*np.cos(detector.angle_radians) ])

return fin

def create_y_fin(self, detector, index, position, path_length):
    # Create useful variables from input arguments
actual_pixel_height = detector.pixel_height - self.collimator_thickness
actual_pixel_width = detector.pixel_width - self.collimator_thickness

# Instantiate a Fin object and assign values to its attributes
fin = Fin('Y', index, self.collimator_thickness)
fin.width = actual_pixel_height
fin.v_vector = position
fin.a_vector = np.array([ 0, self.collimator_thickness, 0 ])  
fin.b_vector = np.array([  
    actual_pixel_height*np.cos(detector.angle_radians),
    0,
    -actual_pixel_height*np.sin(detector.angle_radians),
  ])  
L = 2*path_length / (self.y_resolution / actual_pixel_width + 1)
fin.length = L
fin.c_vector = np.array([ -L*np.sin(detector.angle_radians),  
                          0,  
                          -L*np.cos(detector.angle_radians) ])

return fin

def find_pixel_center(self, position, pixel_height, theta):
  actual_pixel_height = np.array([[  

127
-(pixel_height + self.collimator_thickness)*np.cos(theta),
0 ,
(pixel_height + self.collimator_thickness)*np.sin(theta)
]

pixel_center = position - 0.5*actual_pixel_height

return pixel_center

def calculate_minimum_path_lengths(self, detector):

    # These path lengths will be necessary to calculate fin Lengths for
    # the Y collimators
    # Minimum path length is defined as the ray traced from the current
    # depth position and the current detector pixel that is positioned
    # as close as possible to the object surface and X-ray beam
    # Initializing some helpful variables
    # Only need to worry about XZ dimension so array([ X, Z ])
    self.paths = []
    increment_depth = np.array([0, self.max_depth/detector.num_pixel_H])
    increment_pixel = np.array([
        [-detector.pixel_height*np.cos(detector.angle_radians),
         detector.pixel_height*np.sin(detector.angle_radians) ],
    ])
    depth_start = np.array([0, -self.max_depth + increment_depth[1]/2.])
    pixel_start = np.array([ self.beam_offset + (detector.height -
                           detector.pixel_height/2.) * \
                           np.cos(detector.angle_radians),
                           self.surface_offset + detector.pixel_height * \
                           np.sin(detector.angle_radians)/2.])
for i in range(detector.num_pixel_H):
    depth_pos = depth_start + i*increment_depth
    pixel_pos = pixel_start + i*increment_pixel
    self.paths.append(np.sqrt(np.sum((pixel_pos - depth_pos)**2)))

class UniformCollimator(object):

    Object that contains information and modules necessary to build a uniform collimation grid to accompany the 2D pixelated detector array. This collimator will have a grid that is all a single length, unlike the Collimator class which makes an optimized collimation grid to meet desired resolutions and viewing depths. Therefore you will need to run a system check using the method inside the BackscatterSystem class to find out what the resolutions and viewing range is for this collimator. In order to construct the grid, a fully built Detector instance must be passed to the appropriate modules. Design data located inside the Detector class are necessary to correctly design the collimators.

The grid is broken into two categories: Z collimators and Y collimators. Z collimators are what provide depth discrimination (Z dimension in this framework) and are placed along the height of the detector, looking like fins (thin rectangular parallelepipeds) extending in the Y dimension. They blind a detector pixel from X-ray backscatter signals generated at a specific depth in the object and deeper. Therefore the pixels only receive signal from positions inside the object above that specified
depth. There are as many Z collimators as there are pixels in the
"height" dimension of the detector, one for each row.

Y collimators are what provide the resolution and discrimination in
signal across the object being imaged (Y dimension in this
framework) and are placed along the width of the detector, looking like
fins (thin rectangular parallelepipeds) extending in the Z dimension.

They blind a detector pixel from X-ray backscatter signals generated
from other areas of the object that are supposed to be received by a pixel
closer to that position of scatter. These X-rays that scatter at wrong
angles, causing them to contribute signal to the wrong pixel, contribute
to the reduction in image resolution in the XY plane. There are as many
Y collimators as there are pixels in the "width" dimension of the detector
plus one.

See Detector class documentation for diagrams and explanations of
detector "height" and "width".

Parameters Needed from User:

============================
collimator_thickness : thickness of the detector collimators
    (units: cm)
length : How long all the fins will be inside the grid
    (units: cm)
surface_offset : closest point/distance the system can be to
    the object (units: cm)
beam_offset : closest point/distance the system can be to
    the fanbeam (units: cm)
material : the type of material that the collimator is made
    out of.
Default Values:

===============
collimator_thickness : 0.2 cm (2 mm)
length : 5 cm
surface_offset : 2.0 cm
beam_offset : 2.0 cm
material : 'lead'
Attributes:
============
depths : maximum depth a row of detector pixels can see inside the object. Will be a list with the max depth for each row in grid array (units: cm)
y_resolution : the resolution of the pixels in the Y dimension. Will be a list with the y resolution for each row in grid array (units: cm)
z_fins : List containing each fin in the height direction. Length of list will be equal to the number of pixels in the height dimension of the detector plus one.
y_fins : List containing each fin in the width direction. Length of list will be equal to the number of pixels in the width dimension of the detector plus one, multiplied by the number of pixels in the height dimension of the detector.
starting_point : Vector pointing to the start point when creating the list of fins for the collimation grid (units: cm)
```python
def __init__(self, collimator_thickness = 0.2 , length = 5.0 ,
surface_offset = 2.0 , beam_offset = 2.0 ,
material = 'lead' , build_manually = False):
    if build_manually == True:
        print "Building collimator manually"
        print "Thickness of the collimation grid"
        print "(be mindful that the thickness of the grid will be blocking"
        print "that much of each pixel face in the detector array)"
        print "(in centimeters)"
        self.collimator_thickness = get_value_from_user('float', float)
        print "Length of the the collimators in the grid"
        print "(in centimeters)"
        self.length = get_value_from_user('float', float)
        print "Closest point/distance the system can be to the object"
        print "(offset distance of detector from surface of object)"
        print "(in centimeters)"
        self.surface_offset = get_value_from_user('float', float)
        print "Closest point/distance the system can be to the X-ray fan beam source (offset distance of detector"
        print "from X-ray fan beam plane)"
        print "(in centimeters)"
        self.beam_offset = get_value_from_user('float', float)
        print "Material the collimator is made out of"
        self.material = raw_input('
>> ')
    # These attributes are filled using methods in this class
```
self.z_fins = []
self.y_fins = []
self.depths = []
self.starting_point = None
self.max_depth = None

else:
    self.collimator_thickness = collimator_thickness
    self.length = length
    self.surface_offset = surface_offset
    self.beam_offset = beam_offset
    self.material = material
    self.z_fins = []
    self.y_fins = []
    self.depths = []
    self.starting_point = None
    self.max_depth = None

def construct_grid(self, detector):

    fins contain data in a format that follows that of
    MCNP's BOX Macrobody (See MCNP Manual for more details)
    BOX v_x v_y v_z a_x a_y a_z b_x b_y b_z c_x c_y c_z
    where the v vector points to a corner of the box (from the origin).
    In this situation, a box is one of the fins that form the collimation
grid.
    vectors a, b, and c each are orthogonal to eachother and point to
    the other corners of the box FROM THE STARTING POINT OF THE CORNER
ESTABLISHED BY VECTOR v. To say it another way, for these the vectors
the origin is located at the corner of the fin pointed at by vector v.
z_fins will be a list containing each fin in the height direction.
Length of list will be equal to the number of pixels in the height
dimension of the detector plus one.
y_fins will be a list containing each fin in the width direction.
Length of list will be equal to the number of pixels in the width
dimension of the detector plus one, multiplied by the number of
pixels in the height dimension of the detector.
a single fin is an object of the Fin class containing
those 4 arrays which correspond to the vectors mentioned above, as well
as other data, such as fin length, width, and thickness.
See Fin documentation for more information.
Example:
    z_fins = [fin, fin, fin, fin , fin, ....]

''''

# ==================================================================
# Clear the lists and variables
# ==================================================================
self.z_fins = []
self.y_fins = []

# ==================================================================
# Fill the self.starting_point attribute
# ==================================================================
self.find_starting_point(detector)

# this vector will be used to increment the starting point (v vector)
# to point to each fin needed in the height dimension.
# It is determined by pixel height and detector angle.
# Called increment_xz because it only affects the vector in the
# x and z dimension.
# ==============================================================
increment_xz = np.array([
    -detector.pixel_height*np.cos(detector.angle_radians),
    0 ,
    detector.pixel_height*np.sin(detector.angle_radians) ])
# ==============================================================
# this vector will be used to increment the starting point (v vector)
# to point to each fin needed in the width dimension.
# It is determined by pixel width. Called increment_y because it
# only affects the vector in the y dimension.
# ==============================================================
increment_y = np.array([ 0 , detector.pixel_width , 0 ])
# ==============================================================
# Start iterating over all the fins
# ==============================================================
for i in range(detector.num_pixel_H + 1):
    # Update the position for the starting point of a Z fin
    current_point_xz = self.starting_point + i*increment_xz
    # Create a Z fin and append it to the list
    self.z_fins.append( self.create_z_fin( detector, i+1,
                                          current_point_xz) )

    # ==============================================================
    # After the first iteration, start creating the y fins too
if i > 0:
    
    # Start iterating over y fins
    for j in range(detector.num_pixel_W + 1):
        current_point_xyz = current_point_xz + j*increment_y
        self.y_fins.append( self.create_y_fin( detector,
            (i-1)*(detector.num_pixel_W+1) + (j + 1),
            current_point_xyz )
    
    # Need to calculate depth now that the fin was made
    if i < detector.num_pixel_H:
        Xp,y,Zp = self.starting_point + (i+1)*increment_xz
        xl,y,zl = current_point_xz + self.z_fins[i].a_vector + \
                 self.z_fins[i].c_vector
        self.depths.append( (Zp - zl*Xp/xl)/(Xp/xl - 1) )
        self.max_depth = self.depths[0]

def find_starting_point(self, detector):
    
    finds vector starting point for the construction of the collimation grid
    Heavy use of trigonometry
    Also creates the vectors that define the dimensions of the Detector instance being used.
    
    start_x = self.beam_offset + \
        detector.height * np.cos(detector.angle_radians) + \
self.length * np.sin(detector.angle_radians) + \
self.collimator_thickness * np.cos(detector.angle_radians)

start_y = -(detector.width + self.collimator_thickness)/2.
start_z = self.surface_offset + \
    self.length * np.cos(detector.angle_radians) - \
    self.collimator_thickness * np.sin(detector.angle_radians)

self.starting_point = np.array([ start_x, start_y, start_z ])
detector.create_detector_vectors(self.starting_point,
                                self.collimator_thickness)

def create_z_fin(self, detector, index, position):
    # ===============================================================
    # Instantiate a Fin object and assign values to its attributes
    # ===============================================================

    fin = Fin(‘Z’, index, self.collimator_thickness)
    fin.width = detector.width + self.collimator_thickness

    # Fin starting position (v_vector)
    fin.v_vector = position

    # Fin thickness vector (a_vector)
    fin.a_vector = np.array([-
                             self.collimator_thickness*np.cos(detector.angle_radians),
                             0,
                             self.collimator_thickness*np.sin(detector.angle_radians)]
                        )

    # Fin width vector (b_vector)
    fin.b_vector = np.array([ 0 ,
                              detector.width + self.collimator_thickness ,
                              137])
# Fin length vector (c_vector)
fin.length = self.length

fin.c_vector = np.array([ -self.length*np.sin(detector.angle_radians),
        0,
        -self.length*np.cos(detector.angle_radians)])

return fin

def create_y_fin(self, detector, index, position):
    # ==============================================================
    # Create useful variables from input arguments
    # ==============================================================

    actual_pixel_height = detector.pixel_height - self.collimator_thickness
    actual_pixel_width = detector.pixel_width - self.collimator_thickness

    # ==============================================================
    # Instantiate a Fin object and assign values to its attributes
    # ==============================================================

    fin = Fin('Y', index, self.collimator_thickness)
    fin.width = actual_pixel_height

    # Fin starting position (v_vector)
    fin.v_vector = position

    # Fin thickness vector (a_vector)
    fin.a_vector = np.array([ 0, self.collimator_thickness, 0 ])

    # Fin width vector (b_vector)
    fin.b_vector = np.array([
        actual_pixel_height*np.cos(detector.angle_radians),
        0,
        138])
class Fin(object):
    
    This is a class specifically to be used by the Collimator class. The Collimator is a collimation grid that is composed of many individual fins. a Fin is characterized by its dimensions: thickness, width, and length thickness is determined by the Collimator class and will be the same for every Fin in a Backscatter System. width is determined by detector dimensions:
    
    detector width if it’s a horizontal fin collimating for depth (Z) discrimination
    
    detector pixel height if it’s a vertical fin collimating for surface (Y) discrimination
    
    length is the factor that must be calculated based on the desired system properties such as maximum depth, number of pixels, etc.

    Detector Face
    
    / \
    |
    -------- --

    fin.length = self.length
    fin.c_vector = np.array([-self.length*np.sin(detector.angle_radians), 0,
                             -self.length*np.cos(detector.angle_radians)])

    return fin
Attributes:

--------

collimation_dimension : The dimension in which the fin is trying to
collimate its detector pixel. Z is collimating the depth dimension and Y is collimating across
the surface dimension

index_number : The index of this specific fin amongst the other
definitions that share the same collimation_dimension. 1 (Not 0 like most python indexes) is the first
in the collection.

thickness : Determined by the Collimator class and will be
the same for every Fin in a Backscatter System. (units: cm)

width : Determined by detector width if it’s a Fin
with collimation_dimension = Z
Determined by detector pixel height if it’s a Fin with \texttt{collimation\_dimension = Y} (units: cm)

\textbf{length}: Must be calculated based on the desired system properties such as maximum depth, number of pixels, etc. (units: cm)

The following attributes are modeled after the BOX Macrobody used in MCNP6. See MCNP manual for more information on Macrobodies.

See \texttt{Collimator.construct\_grid} documentation for more information on fin attributes and MCNP format.

\textbf{v\_vector}: Base vector. Points from origin to a corner of the box that defines this fin (units: cm)

\textbf{a\_vector}: Dimension vector describing fin thickness. Provides direction and therefore magnitude (size) of a side of the box. This vector’s components are relative to the corner pointed at by \textbf{v\_vector} (units: cm)

\textbf{b\_vector}: Dimension vector describing fin width. Provides direction and therefore magnitude (size) of a side of the box. This vector’s components are relative to the corner pointed at by \textbf{v\_vector} (units: cm)

\textbf{c\_vector}: Dimension vector describing fin length. Provides direction and therefore magnitude (size) of a side of the box. This vector’s components are relative to the corner pointed at by \textbf{v\_vector}
def __init__(self, collimation_dimension, index_number, thickness):
    self.collimation_dimension = collimation_dimension
    self.index_number = index_number
    self.thickness = thickness
    self.width = 0
    self.length = 0
    self.v_vector = 0
    self.a_vector = 0
    self.b_vector = 0
    self.c_vector = 0

def get_value_from_user(dtype, convert_func):
    
    helper function when system components are being built manually and require user input.
    Tries to convert the user input into the right data type based on the conversion function that was passed. If successful, it loads it into variable
    Example: passing float, will try converting user input into a float
    If an error is raised, it will tell the user and ask them to try again.
    
correctly_input = False
while not correctly_input:
    try:
        response = convert_func(raw_input('>> '))
correctly_input = True
except ValueError:
    print '\nIncorrect, the input must be a ' + dtype
    print 'Try again: '
return response

def convert_file_to_array(file_path):
    opened = False
    spectrum = []
    while not opened:
        try:
            f = open(file_path, 'r')
            opened = True
        except IOError:
            print '\nFile and/or path does not exist'
            print 'Enter a file path or quit\n'
            response = raw_input('>> ')
            if response == 'q' or response == 'quit':
                return 75.0
            else:
                file_path = response
        contents = f.readlines()
        t = []
        for line in contents:
            # Ignores lines beginning with '#' and any blank lines
            if '#' not in line and line != '\n':
                # Remove 'newline' character and split up values
spectrum.append([float(value) for value in 
    line.rstrip('
').split()])

return np.array(spectrum)

def normalize_spectrum(spectrum):
    norm_spectrum = np.copy(spectrum)
    norm_spectrum[:,1] = norm_spectrum[:,1]/norm_spectrum[:,1].sum()
    return norm_spectrum

def find_intercept(point1, point2):
    x1, y1 = point1
    x2, y2 = point2
    m = (y2 - y1)/(x2 - x1)
    b = y1 - m*x1
    return b

A.2 mcnp.py

Contains classes and modules for the creation of a 2D backscatter system for MCNP input files and the creation of images using said system.

import system_design as design
import numpy as np
import os

# This is a dictionary that contains materials that can be used for the Material class. Keys are common names for the material and Values are a tuple containing the material density and then the ZAID identifier and fraction of that isotope in the material
# (See MCNP manual for more details about material card specifications)

```python
LIST_OF_KNOWN_MATERIALS = {'lead': ('-11.34', '82000 -1.00'),
                           'tungsten': ('-19.30', '74000 -1.00'),
                           'air': ('-0.0012041', '6000 -0.000124 ' + 
                                   '7000 -0.755268 ' + 
                                   '8000 -0.231781 ' + 
                                   '18000 -0.012827')}
```

# Classes

```python
class Cell(object):
    # Contains the properties describing a cell card in MCNP
    def __init__(self, name=None, material_num=None, material_density=None,
                 surfaces=None, importance='P=1'):
        self.name = name
        self.material_num = material_num
        self.material_density = material_density
        self.surfaces = surfaces
        self.importance = importance

class Surface(object):
    # Contains the properties describing a surface card in MCNP
    def __init__(self, name=None, TR=None, type=None, type_variables=''):
        self.name = name
```
self.TR = TR
self.type = type
self.type_variables = type_variables

class SourceDefinition(object):
    '''
    Contains the properties describing a source definition (SDEF) card in MCNP
    '''
    def __init__(self, position = None, energy = None, particle = '2',
                  vector = None, direction = None, TR= None,
                  weight = None):
        self.name = 'SDEF'
        self.position = position
        self.energy = energy
        self.particle = particle
        self.vector = vector
        self.direction = direction
        self.TR = TR
        self.weight = weight
        self.distributions = {}

    def make_energy_distribution(self, energies):
        name = self.energy.split('d')[-1]
        SI = 'SI'+name + ' A ' + ' '.join(map(str, energies[:,0]))
        SP = 'SP'+name + ' ' + ' '.join(map(str, energies[:,1]))
        self.distributions.update({self.energy : [SI, SP]})

    def make_direction_distribution(self, angle):
        name = self.direction.split('d')[-1]
        angle_bins = [-1, round(np.cos(angle*np.pi/180), 5), 1]
SI = 'SI'+name + ' ' + ' '.join(map(str, angle_bins))
probabilities = [ 0,
    round((1 - angle_bins[0] - 1 + angle_bins[1])/2., 5),
    round((1 - angle_bins[1] - 1 + angle_bins[2])/2., 5) ]
SP = 'SP'+name + ' ' + ' '.join(map(str, probabilities))
SB = 'SB'+name + ' 0 0 1'
self.weight = str(round(1./probabilities[-1], 5))
self.distributions.update({self.direction : [SI, SP, SB]})

class CoordinateTransformation(object):
    '''
    Contains the properties describing a coordinate transformation (TR) card
    in MCNP
    '''
    def __init__(self,name=None,degrees=True,displacement_vector=None):
        self.name = name
        self.degrees = degrees # If True, then name has *TR instead of TR
        self.displacement_vector = displacement_vector
        if self.degrees:
            self.rotation_matrix = np.array([ [0, 90, 90],
                                               [90, 0, 90],
                                               [90, 90, 0]])
        else:
            self.rotation_matrix = np.array([ [1, 0, 0],
                                               [0, 1, 0],
                                               [0, 0, 1]])
        self.M = '1'

class Material(object):
def __init__(self, description = None, name = None ):
    self.description = description
    self.name = name
    if LIST_OF_KNOWN_MATERIALS.has_key(self.description):
        self.composition = LIST_OF_KNOWN_MATERIALS[description][1]
    else:
        self.composition = None

class Tally(object):
    def __init__(self, type, name, radiation = 'P', type_variables = ''):
        self.type = type
        self.name = name
        self.radiation = radiation
        self.type_variables = type_variables

# ==========================================================================
# Modules
# ==========================================================================
# Modules Specific to Creating this 2D Backscatter System
# ==========================================================================
def create_input_deck(system, file_path = 'system_template.i',

Takes the backscatter system and exports it into a readable format in an MCNP input file. The system is an instance of `system_design.BackscatterSystem`. The `file_path` is either a file name or a path to (and including) the file. The `TR` parameter is the coordinate transformation name that will be used when creating surface cards and source definition cards.

```python
if not isinstance(system, design.BackscatterSystem):
    print "ERROR: must pass an instance of"
    print "system_design.BackscatterSystem"
    return

# Either create a new file or overwrite one. Check if new directories were requested too.
# if os.path.isfile(file_path):

responded = False

print "\nWARNING: This file already exists, do you wish to proceed?"
print "Doing so will overwrite the file."
print "(yes/no)"
response = raw_input('>>  ')

while not responded:
    if response == 'y' or response == 'yes':
        input_deck = open(file_path, 'w')
        responded = True
```
elif response == 'n' or response == 'no':
    return
else:
    print "\nYour response was not a valid option."
    print "Options: yes, no \
    response = raw_input('>> ')

elif len(file_path.split('/')) > 1:
    directory = file_path.rpartition('/')[0]
    if not os.path.isdir(directory):
        os.makedirs(directory)
        input_deck = open(file_path, 'w')
    else:
        input_deck = open(file_path, 'w')
else:
    input_deck = open(file_path, 'w')

# Create Cell, Surface, and Material cards for system
# Collimator Data
# collimator_cells, collimator_surfaces, collimator_material = \
#     create_collimation_cards(system,TR)

# If it’s a fan beam, need to create collimators that will shape the
# x-ray source into a fan beam
if system.XRayFanBeam.fan_angle > 0.0:
    xray_collimator_cells, xray_collimator_surfaces = \
        create_xray_collimation_cards(system,TR)

# Create Detector Cell where mesh tally will be placed inside of
detector_cell, detector_surface = create_detector_cards(system,
     TR=str(int(TR)+1))

# Figure out maximum dimension for determining the size of the problem
x_max, y_max, z = system.Detector.v_vector + system.Detector.a_vector
x, y, z_max = system.Detector.v_vector + system.Detector.a_vector + \
    system.Detector.c_vector
max_dimension = np.max([system.XRayFanBeam.surface_offset,
    x_max, y_max, z_max ])

problem_boundary_surface = Surface(name=’99998’, type=’so’, TR='’,
    type_variables=str(round(max_dimension*2.,3)))
inside_boundary_material = Material(description=’air’, name=’900’)
inside_boundary_cell = create_inside_boundary_cell(collimator_cells +
    xray_collimator_cells +
    [detector_cell],
    problem_boundary_surface, inside_boundary_material)
outside_boundary_cell = Cell(name = ‘99999’,
    material_num = ‘’,
    material_density = ‘0’,
    surfaces = problem_boundary_surface.name,
    importance = ‘P=0’)

# Source Definition (X Ray Fan Beam Data)
source_def = create_source_definition(system, TR)

# Coordinate Transformation (will be used when scanning over an object)
transform = CoordinateTransformation(name = TR,
    displacement_vector = ’0 0 0’)

# Create Mesh Tally for Detector
mesh_tally = create_detector_tally(system, TR=str(int(TR) + 1),
                                 energy_mesh=True)

# Create Transform for detector
tally_transform = create_detector_transform(system, TR=str(int(TR) + 1))

# Start writing the contents for the input file
# ====================================================================
# Write Cell Cards
# ================
write_header('Cells', input_deck)
write_subheader('Collimation Grid', input_deck)
for cell in collimator_cells:
    write_cell_card(cell, input_deck)
if system.XRayFanBeam.fan_angle > 0.0:
    write_subheader('X-Ray Fan Beam Collimators', input_deck)
    for cell in xray_collimator_cells:
        write_cell_collimator_card(cell, input_deck)
write_subheader('Detector', input_deck)
write_cell_card(detector_cell, input_deck)
write_subheader('Phantom', input_deck)
write_subheader('Outside of Problem', input_deck)
write_cell_card(outside_boundary_cell, input_deck)
write_subheader('Inside of Problem', input_deck)
write_cell_card(inside_boundary_cell, input_deck)

# Write Surface Cards
# ===================
input_deck.write('
')
write_header('Surfaces', input_deck)
write_subheader('Collimation Grid', input_deck)
for surface in collimator_surfaces:
    write_surface_card(surface, input_deck)
if system.XRayFanBeam.fan_angle > 0.0:
    write_subheader('X-Ray Fan Beam Collimators', input_deck)
    for surface in xray_collimator_surfaces:
        write_surface_card(surface, input_deck)
write_subheader('Detector', input_deck)
write_surface_card(detector_surface, input_deck)
write_subheader('Phantom', input_deck)
write_subheader('Problem Boundary', input_deck)
write_surface_card(problem_boundary_surface, input_deck)
#
# Write Data Cards
#
# ================
input_deck.write('
')
write_header('Data', input_deck)
write_subheader('Coordinate Transformation', input_deck)
write_coordinate_transform_card(transform, input_deck)
write_coordinate_transform_card(tally_transform, input_deck)
write_subheader('Materials', input_deck)
write_material_card(collimator_material, input_deck)
write_material_card(inside_boundary_material, input_deck)
write_subheader('Source Definition', input_deck)
write_subheader('Source Definition', input_deck)
write_source_definition_card(source_def, input_deck)

# Write Tallies
write_subheader('Tallies', input_deck)
write_tally_card(mesh_tally, input_deck)

# Write Ending
input_deck.write('c
')
input_deck.write('MODE ' + mode + ' \n')
input_deck.write('NPS ' + str(nps) + ' \n')
input_deck.write('PRINT 110 160 161 162')
input_deck.close()

def create_collimation_cards(system, TR):
    
    Creates surface, cell, and material cards for the collimation grid
    
    if not isinstance(system.Collimator, design.Collimator) and \
        not isinstance(system.Collimator, design.UniformCollimator):
        print '\nError: did not receive an instance of system_design.Collimator'
        print "or system_design.UniformCollimator"
        return
    
    coll = system.Collimator
    cells = []
    surfaces = []
    material = Material(description = coll.material.lower(), name = '200')
    num_z = coll.z_fins[-1].index_number
    for fin in coll.z_fins:
        surface = Surface(name = str(fin.index_number), 
            TR = TR ,
type = 'BOX')

# Round values to 3 decimal places
values = map(lambda x: round(x, 5), np.hstack([fin.v_vector,
                                            fin.a_vector*0.99,
                                            fin.b_vector,
                                            fin.c_vector]))

# Combine values into a string
surface.type_variables = ' '.join(map(str, values))
cell = Cell(name=str(fin.index_number),
            material_num=material.name,
            material_density=LIST_OF_KNOWN_MATERIALS[material.description][0],
            surfaces='-' + surface.name)
surfaces.append(surface)
cells.append(cell)
for fin in coll.y_fins:
    surface = Surface(name = str(num_z + fin.index_number) ,
                       TR = TR ,
                       type = 'BOX')

    # Round values to 3 decimal places
    values = map(lambda x: round(x, 5), np.hstack([fin.v_vector,
                                                 fin.a_vector,
                                                 fin.b_vector*.99,
                                                 fin.c_vector]))

    # Combine values into a string
    surface.type_variables = ' '.join(map(str, values))
cell = Cell(name=str(num_z + fin.index_number),
            material_num=material.name,
material_density=LIST_OF_KNOWN_MATERIALS[material.description][0],
surfaces='-' + surface.name)
surfaces.append(surface)
cells.append(cell)
return cells, surfaces, material

def create_inside_boundary_cell(other_cells, surface, material):
    problem_cell = Cell(name=surface.name,
                        material_num=material.name,
                        material_density=LIST_OF_KNOWEN_MATERIALS[material.description][0])

    problem_cell.surfaces = '- ' + surface.name
    for cell in other_cells:
        problem_cell.surfaces += ' #' + cell.name

    return problem_cell

def create_xray_collimation_cards(system, TR):
    angle = system.XRayFanBeam.fan_angle *np.pi/180.
    zmin = system.XRayFanBeam.surface_offset - 2
    zmax = zmin + 1
    ymin = -np.tan(angle)*2 - 1
    ymax = np.abs(ymin)
    xmin = system.step_size/(system.XRayFanBeam.surface_offset + 
                            system.Collimator.max_depth)
    xmax = ymax

    values = map(lambda x: round(x, 5), [xmin, xmax, ymin, ymax, zmin, zmax])
    surface1 = Surface(name = '99991', TR = TR, type = 'RPP')
    surface1.type_variables = ' '.join(map(str, values))
    surface2 = Surface(name = '99992', TR = TR, type = 'RPP')
    values = map(lambda x: round(x, 5), [-xmax, -xmin, ymin, ymax, zmin, zmax])
surface2.type_variables = ' '.join(map(str, values))
cell1 = Cell(name = '99991', material_num = '', material_density = '0',
    surfaces = '-'+surface1.name, importance = 'P=0')
cell2 = Cell(name = '99992', material_num = '', material_density = '0',
    surfaces = '-'+surface2.name, importance = 'P=0')
return [cell1, cell2], [surface1, surface2]
def create_source_definition(system, TR):
    if not isinstance(system.XRayFanBeam, design.XRayFanBeam):
        print "Error: did not receive an instance of \n" +
        "system_design.XRayFanBeam"
        return
    fan = system.XRayFanBeam
    sdef = SourceDefinition(position = '0 0 ' + str(fan.surface_offset),
        vector = '0 0 -1', TR = TR)
    if isinstance(fan.energy, np.ndarray):
        sdef.energy = 'd1'
        sdef.make_energy_distribution(fan.normalized_energy)
    else:
        sdef.energy = str(fan.energy)
    if fan.fan_angle > 0:
        sdef.direction = 'd2'
        sdef.make_direction_distribution(fan.fan_angle)
    else:
        sdef.direction = '1'
    return sdef
def create_detector_tally(system, TR, energy_mesh = False):
    if not isinstance(system.Detector, design.Detector):
        # Remaining code
print "\nError: did not receive an instance of " +\n"system_design.Detector"

return
det = system.Detector
tally = Tally(type = 'FMESH', name = '14')
tally.type_variables += 'GEOM=xyz ORIGIN=0 0 0'
tally.type_variables += ' TR=' + TR + ' \n'
i = ' '.join([' '*8, 'IMESH=' + str(det.thickness), 'IINTS=1\n'])
j = ' '.join([' '*8, 'JMESH=' + str(det.width),
               'JINTS=' + str(det.num_pixel_W) + ' \n'])
k = ' '.join([' '*8, 'KMESH=' + str(det.height),
               'KINTS=' + str(det.num_pixel_H) + ' \n'])
tally.type_variables += ' '.join([', i, j, k])
if energy_mesh:
    if isinstance(system.XRayFanBeam.energy, np.ndarray):
        e = ' '*10 + 'EMESH=' + str(system.XRayFanBeam.energy[-2,0])
        e += ' EINTS=' + str(int(system.XRayFanBeam.energy[-2,0]/0.005)) +\n             ' \n'
    else:
        e = ' '*10 + 'EMESH=' + str(system.XRayFanBeam.energy)
        e += ' EINTS=' + str(int(system.XRayFanBeam.energy/0.005)) + ' \n'
tally.type_variables += e
return tally
def create_detector_transform(system, TR):

transform = CoordinateTransformation(name = TR)
values = map(lambda x: round(x, 5),
system.Detector.v_vector + np.array([0.01, 0, 0.01])

transform.displacement_vector = ' '.join(map(str, values))

angle = system.Detector.angle

transform.rotation_matrix = np.array([[[90-angle, 90, angle],
            [90, 0, 90],
            [180-angle, 90, 90-angle]]]

return transform

def create_detector_cards(system, TR):
    if not isinstance(system.Detector, design.Detector):
        print "Error: did not receive an instance of " +
        "system_design.Detector"

        return

    det = system.Detector
    xmin, ymin, zmin = det.v_vector

    surface = Surface(name = '99990', TR = TR, type = 'RPP')
    values = map(lambda x: round(x, 5), [0, det.thickness,
                                           0, det.width,
                                           0, det.height])

    surface.type_variables = ' '.join(map(str, values))

    cell = Cell(name = '99990', material_num = '', material_density = '0',
                 surfaces = '-'+surface.name)

    return cell, surface

# ==========================================================================
# General Modules for Writing to an MCNP Input File
# ==========================================================================

def write_subheader(text, deck):
    text = ' ' + text + ' '
def write_header(text, deck):
    text = ' ' + text + ' ' 
    deck.write('c ' + '-'*78 + '\n')
    deck.write('c ' + text.center(78, '-') + '\n')
    deck.write('c ' + '-'*78 + '\n')
    deck.write('c
')

def write_cell_card(cell, deck):
    if not isinstance(cell, Cell):
        print "Error: did not receive an instance of mcnp.Cell"
        return
    head = ' '.join([cell.name, cell.material_num, cell.material_density])
    tail = ' '.join([cell.surfaces, 'IMP:' + cell.importance])
    write_line(head, tail, deck)

def write_surface_card(surface, deck):
    if not isinstance(surface, Surface):
        print "Error: did not receive an instance of mcnp.Surface"
        return
    head = ' '.join([surface.name, surface.TR, surface.type])
    tail = surface.type_variables
    write_line(head, tail, deck)

def write_coordinate_transform_card(transform, deck):
    if not isinstance(transform, CoordinateTransformation):
        print "Error: did not receive an instance of " + "mcnp.CoordinateTransformation"
return
rotation = ' '.join(map(str, np.hstack(transform.rotation_matrix)))
line = ' '.join(['TR' + transform.name, transform.displacement_vector,
                   rotation, transform.M])
if transform.degrees:
    deck.write('*' + line + '\n')
else:
    deck.write(line + '\n')
def write_material_card(material, deck):
    if not isinstance(material, Material):
        print "\nError: did not receive an instance of " +\
            "mcnp.Material"
    return
    deck.write('c\nc ' + material.description.capitalize() + '\nc\n')
name = 'm' + material.name + ' '
zaids = material.composition.split()
zaids.reverse()
    deck.write(name + ' ' + ' '.join([zaids.pop(), zaids.pop()]) + '\n')
for i in range(len(zaids)/2):
    zaid_and_fraction = ' '.join([zaids.pop(), zaids.pop()])
    deck.write( zaid_and_fraction.rjust(len(name)+len(zaid_and_fraction))\
                  + '\n')
def write_source_definition_card(sdef, deck):
    if not isinstance(sdef, SourceDefinition):
        print "\nError: did not receive an instance of mcnp.SourceDefinition"
    return
head = ' '.join(['SDEF',

'POS'='+sdef.position,
'ERG'='+sdef.energy])

if sdef.weight:
    head += ' WGT='+'+sdef.weight

tail = ' '.join(['PAR='+'+sdef.particle,
                'VEC='+'+sdef.vector,
                'DIR='+'+sdef.direction,
                'TR='+'+sdef.TR])

write_line(head, tail, deck)

for distribution in sdef.distributions.keys():
    for line in sdef.distributions[distribution]:
        head, temp, tail = line.partition(' ')
        # In case the distribution is specified with a character,
        # like A,D,L etc.
        if tail.split()[0].isalnum():
            head += ' '+tail[0]
        tail = tail[1:]
        else:
            head += ' ' + tail[0]
            tail = tail[1:]

        write_line(head, tail, deck)

def write_tally_card(tally, deck):
    
    if not isinstance(tally, Tally):
        print "Error: did not receive an instance of mcnp.Tally"
        return

    if 'FMESH' in tally.type.upper():
        deck.write(tally.type + tally.name + ': ' + tally.radiation)
        deck.write(' ' + tally.type_variables)
else:
    head = tally.type + tally.name + ': ' + tally.radiation
    tail = tally.type_variables
    write_line(head, tail, deck)

def plot_basis(system):
    
    Helps you visualize collimation grid in the MCNP plotter
    by telling you what to input for the BASIS command in the
    plotter.
    
    angle = system.Detector.angle_radians
    x = str(-np.cos(angle))
    y = '0'
    z = str(np.sin(angle))
    print 'For Y collimators:'
    print 'basis = 0 1 0 ' + ' '.join([x,y,z])
    print 'For Z collimators:'
    print 'basis = 1 0 0 0 0 1'

def write_line(head, tail, deck):
    # Check if line needs to be written over multiple lines
    if len(head + tail) > 80:
        pad = len(head) + 1
        # Find string length of element with longest name
        max_length = np.max(map(lambda x: len(x), tail.split())) + 1
        # determine how many surfaces can fit on a single line
        num_per_line = (80 - pad)/max_length
        elements = tail.split()
A.3 visualize.py

```python
import vtk
import system_design
import numpy as np

def system(system, show_surface=True, show_beam=True):
    # Create the graphics structure. The renderer renders into the render
    # window. The render window interactor captures mouse events and will
    # perform appropriate camera or actor manipulation depending on the
    # nature of the events.
    ren = vtk.vtkRenderer()
    renWin = vtk.vtkRenderWindow()
    renWin.AddRenderer(ren)
    iren = vtk.vtkRenderWindowInteractor()
    iren.SetRenderWindow(renWin)
    # Add the Z fins to the renderer
    for fin in system.Collimator.z_fins:
        fin_coords = [fin.v_vector, fin.b_vector, fin.a_vector,
                      fin.c_vector]
        finAct = create_hexahedron_actor(fin_coords)
        ren.AddActor(finAct)
```
# Add the Y fins to the renderer
for fin in system.Collimator.y_fins:
    fin_coords = [fin.v_vector, fin.a_vector, fin.b_vector, fin.c_vector]
    finAct = create_hexahedron_actor(fin_coords)
    ren.AddActor(finAct)

# Add the Detector to the renderer
det_coords = [system.Detector.v_vector, system.Detector.b_vector,
              system.Detector.c_vector, system.Detector.a_vector]
ren.AddActor(create_hexahedron_actor(det_coords, color=(92, 144, 249)))
if show_beam:
    # Add the X-ray Beam to the renderer
    beam_coords = get_beam_vectors(system)
    ren.AddActor(create_pyramid_actor(beam_coords))
if show_surface:
    # Add plane showing Object surface
    center = (system.Detector.width, 0, 0)
    point1 = (0, 4*system.Detector.width, 0)
    point2 = (4*system.Detector.width, 0, 0)
    ren.AddActor(create_plane_actor(center, point1, point2,
                                     color=(100,100,100)))

# Set the background and size
ren.SetBackground(0.9, 0.9, 0.9)
renWin.SetSize(500, 500)

# This allows the interactor to initialize itself. It has to be
called before an event loop.
iren.Initialize()

# We’ll zoom in a little by accessing the camera and invoking a "Zoom"
# method on it.
ren.ResetCamera()
ren.SetActiveCamera().Zoom(1.5)
renWin.Render()
# Start the event loop.
iren.Start()

def create_hexahedron_actor(hex_coords, color = (186, 200, 214)):
    if np.max(color) > 1.0:
        color = (color[0]/255., color[1]/255., color[2]/255.)
    src_vector, width_vector, thickness_vector, length_vector = hex_coords
    P0 = src_vector
    P1 = src_vector + width_vector
    P2 = src_vector + width_vector + length_vector
    P3 = src_vector + length_vector
    P4 = src_vector + thickness_vector
    P5 = src_vector + thickness_vector + width_vector
    P6 = src_vector + thickness_vector + width_vector + length_vector
    P7 = src_vector + thickness_vector + length_vector
    # Create the points
    points = vtk.vtkPoints()
    points.InsertNextPoint(P0)
    points.InsertNextPoint(P1)
    points.InsertNextPoint(P2)
    points.InsertNextPoint(P3)
    points.InsertNextPoint(P4)
    points.InsertNextPoint(P5)
    points.InsertNextPoint(P6)
points.InsertNextPoint(P7)

# Create a hexahedron from the points
hexahedron = vtk.vtkHexahedron()
for i in range(8):
    hexahedron.GetPointIds().SetId(i,i)

# Add the hexahedron to a cell array
hexs = vtk.vtkCellArray()
hexs.InsertNextCell(hexahedron)

# Add the points and hexahedron to an unstructured grid
uGrid = vtk.vtkUnstructuredGrid()
uGrid.SetPoints(points)
uGrid.InsertNextCell(hexahedron.GetCellType(), hexahedron.GetPointIds())

# Create mapper and Actor
mapper = vtk.vtkDataSetMapper()
mapper.SetInputData(uGrid)
actor = vtk.vtkActor()
actor.SetMapper(mapper)
actor.GetProperty().SetColor(color)
return actor

def create_pyramid_actor(pyramid_coords, color = (239, 57, 57)):
    if np.max(color) > 1.0:
        color = (color[0]/255., color[1]/255., color[2]/255.)

    # Using vectors in the format of get_beam_vectors() function
    # source_vector, height_vector, step_vector, width_vector
    src_vector, h_vector, step_vector, w_vector = pyramid_coords
    P0 = src_vector
    P1 = src_vector + h_vector + step_vector + w_vector
P2 = src_vector + h_vector + step_vector - w_vector
P3 = src_vector + h_vector - step_vector - w_vector
P4 = src_vector + h_vector - step_vector + w_vector

# Create the points
points = vtk.vtkPoints()
points.InsertNextPoint(P0)
points.InsertNextPoint(P1)
points.InsertNextPoint(P2)
points.InsertNextPoint(P3)
points.InsertNextPoint(P4)

# Create a pyramid from the points
pyramid = vtk.vtkPyramid()
for i in range(8):
    pyramid.GetPointIds().SetId(i, i)

# Add the pyramid to a cell array
cells = vtk.vtkCellArray()
cells.InsertNextCell(pyramid)

# Add the points and pyramid to an unstructured grid
uGrid = vtk.vtkUnstructuredGrid()
uGrid.SetPoints(points)
uGrid.InsertNextCell(pyramid.GetCellType(), pyramid.GetPointIds())

# Create mapper and actor
mapper = vtk.vtkDataSetMapper()
mapper.SetInputData(uGrid)
actor = vtk.vtkActor()
actor.SetMapper(mapper)
actor.GetProperty().SetColor(color)
def get_beam_vectors(system):
    # Point where X-rays are generated
    source_vector = np.array([0.0, 0.0, system.XRayFanBeam.surface_offset])

    # Vector Pointing from source to max depth of imaging
    height_vector = np.array([0.0, 0.0, -system.XRayFanBeam.surface_offset -
                              system.Collimator.max_depth])

    # Vector pointing from source to half of system step size (x direction)
    step_vector = np.array([system.step_size/2., 0.0, 0.0])

    # Vector pointing from source to half of detector width (y direction)
    angle = system.XRayFanBeam.fan_angle * np.pi/180.
    half_width = (system.XRayFanBeam.surface_offset +
                  system.Collimator.max_depth) * np.tan(angle)
    width_vector = np.array([0.0, half_width, 0.0])

    return [source_vector, height_vector, step_vector, width_vector]

def create_plane_actor(center, point1, point2, color=(0,0,0)):
    if np.max(color) > 1.0:
        color = (color[0]/255., color[1]/255., color[2]/255.)

    source = vtk.vtkPlaneSource()
    source.SetPoint1(point1)
    source.SetPoint2(point2)
    source.SetCenter(center)

    # mapper
    mapper = vtk.vtkPolyDataMapper()
    mapper.SetInputConnection(source.GetOutputPort())

    # actor
    actor = vtk.vtkActor()
actor.SetMapper(mapper)
actor.GetProperty().SetColor(color)
return actor
def hello_world():
    from vtk.util.colors import tomato
    # This creates a polygonal cylinder model with eight circumferential
    # facets.
cylinder = vtk.vtkCylinderSource()
cylinder.SetResolution(8)
    # The mapper is responsible for pushing the geometry into the graphics
    # library. It may also do color mapping, if scalars or other
    # attributes are defined.
cylinderMapper = vtk.vtkPolyDataMapper()
cylinderMapper.SetInputConnection(cylinder.GetOutputPort())
    # The actor is a grouping mechanism: besides the geometry (mapper), it
    # also has a property, transformation matrix, and/or texture map.
    # Here we set its color and rotate it -22.5 degrees.
cylinderActor = vtk.vtkActor()
cylinderActor.SetMapper(cylinderMapper)
cylinderActor.GetProperty().SetColor((0.5137254901960784,
                                      0.5490196078431373,
                                      0.01568627450980392))
cylinderActor.RotateX(30.0)
cylinderActor.RotateY(-45.0)
    # Create the graphics structure. The renderer renders into the render
    # window. The render window interactor captures mouse events and will
    # perform appropriate camera or actor manipulation depending on the
# nature of the events.
ren = vtk.vtkRenderer()
renWin = vtk.vtkRenderWindow()
renWin.AddRenderer(ren)
iren = vtk.vtkRenderWindowInteractor()
iren.SetRenderWindow(renWin)

# Add the actors to the renderer, set the background and size
ren.AddActor(cylinderActor)
ren.SetBackground(0.1, 0.2, 0.4)
renWin.SetSize(200, 200)

# This allows the interactor to initialze itself. It has to be
# called before an event loop.
iren.Initialize()

# We'll zoom in a little by accessing the camera and invoking a "Zoom"
# method on it.
ren.ResetCamera()
ren.GetActiveCamera().Zoom(1.5)
renWin.Render()

# Start the event loop.
iren.Start()
from mpl_toolkits.mplot3d import Axes3D
from mpl_toolkits.mplot3d.art3d import Poly3DCollection
import matplotlib.pyplot as plt
import matplotlib.patches as patches
import matplotlib as mpl
import numpy as np

def xray_spectrum(xray_object, norm = True, title = 'X-ray Spectrum'):
    
    Plots the X ray spectrum from an XRayFanBeam instance
    Will only work if the XRayFanBeam instance actually has a spectrum to plot
    and is not a mono-energetic source.
    Can choose to plot either the original spectrum or the normalized spectrum,
    default is the normalized spectrum.
    
    if isinstance(xray_object, design.BackscatterSystem):
        if isinstance(xray_object.XRayFanBeam, design.XRayFanBeam):
            if isinstance(xray_object.XRayFanBeam.energy, np.ndarray):
                if norm:
                    spectrum = xray_object.XRayFanBeam.normalized_energy
                else:
                    spectrum = xray_object.XRayFanBeam.energy
            else:
                print "X-ray energy source is not a spectrum; nothing to plot"
                print "XRayFanBeam.energy needs to be a numpy.ndarray"
                return
        else:
            print "BackscatterSystem instance does not contain an"
print "XRayFanBeam instance"
return

eelif isinstance(xray_object, design.XRayFanBeam):
    if isinstance(xray_object.energy, np.ndarray):
        if norm:
            spectrum = xray_object.normalized_energy
        else:
            spectrum = xray_object.energy
    else:
        print "\nX-ray energy source is not a spectrum; nothing to plot"
        print "XRayFanBeam.energy needs to be a numpy.ndarray"
        return
else:
    print "\nError: Did not pass an instance of XRayFanBeam or an"
    print "instance of BackscatterSystem containing an XRayFanBeam."
    return

figure = plt.figure()
ax = figure.add_subplot(111)
ax.set_title(title)
ax.set_xlabel('Energy (MeV)')
if norm:
    ax.set_ylabel('Probability')
else:
    ax.set_ylabel('Frequency')
plt.plot(spectrum[:,0], spectrum[:,1])
plt.show()
return figure, ax
def create_figure_2D(title, limits):
    
    Creates a 2D figure instance
    plotting limits are determined by the maximum dimensions of the system
    
    figure = plt.figure()
    ax = figure.add_subplot(111)
    xlims, ylims = limits
    ax.set_xlim( xlims )
    ax.set_ylim( ylims )
    ax.set_title(title)
    return figure, ax

def xz_system(system, title = 'Depth Collimation', show_ray_limits = True):
    
    Plot a 2D view of just the Z collimators.
    Have the option to view rays being traced from the detector pixels
to their respective maximum depths of view for each collimator
Due to the limitations of plotting rectangle patches in matplotlib,
the view is from the angle of the detector, so everything is rotated
by the degree that the detector was set up at.

    rectangle_patches = []
    lines = []
    angle = system.Detector.angle_radians
    rotate_matrix = np.array([[ np.sin(angle), np.cos(angle) ],
                              [-np.cos(angle), np.sin(angle) ] ])
    x,y,z = system.Detector.v_vector + system.Detector.a_vector
xx, yy, zz = system.Detector.v_vector + system.Detector.c_vector + \
    system.Detector.a_vector

x_r = np.dot(rotate_matrix, np.array([[[-1., x*1.1, x*1.1],
                                      [0., 0., z*1.1]]]))

z_r = np.dot(rotate_matrix, np.array([[0., 0., xx*1.1],
                                      [-system.Collimator.max_depth*1.1,
                                       system.XRayFanBeam.surface_offset*1.1,
                                       zz*1.1]]))

xlims = [np.min([x_r[0], z_r[0]]), np.max([x_r[0], z_r[0]])]
zlims = [np.min([x_r[1], z_r[1]]), np.max([x_r[1], z_r[1]])]

xlims = [round(x) for x in xlims]
zlims = [round(z) for z in zlims]

if zlims[1] - zlims[0] > xlims[1] - xlims[0]:
    limits = [[xlims[0], xlims[0] + (zlims[1] - zlims[0])], zlims]
else:
    limits = [xlims, [zlims[0], zlims[0] + (xlims[1] - xlims[0])]]

fig, ax = create_figure_2D(title, limits)

# Detector
det_x_start, det_y_start = np.dot(rotate_matrix, 
    [system.Detector.v_vector[0],
     system.Detector.v_vector[2]])

det_patch = patches.Rectangle((det_x_start, det_y_start),
                           175
system.Detector.thickness ,
system.Detector.height ,
fc='#22F0E6' ,
alpha=1.0 ,
)
rectangle_patches.append(det_patch)

# Fins
for fin in system.Collimator.z_fins:
    f_x, f_y = np.dot(rotate_matrix, [fin.v_vector[0], fin.v_vector[2]])
    f = patches.Rectangle((f_x, f_y), -fin.length, fin.thickness,
                          fc='k', alpha=1.0 )
    rectangle_patches.append(f)

# Fan Beam
fan_x_start, fan_y_start = np.dot(rotate_matrix, [0 ,
                      system.XRayFanBeam.surface_offset])
fan_x_finish, fan_y_finish = np.dot(rotate_matrix, [0 ,
                      round(-system.Collimator.max_depth*1.1)])

lines.append( plt.Line2D( (fan_x_start, fan_x_finish),
                          (fan_y_start, fan_y_finish),
                          color='r', lw=1.5 ) )

# Object Surface
surf_x_start, surf_y_start = np.dot(rotate_matrix, [-1 , 0])
surf_x_finish, surf_y_finish = np.dot(rotate_matrix,
                      [round(system.Detector.v_vector[0]), 0] )

lines.append( plt.Line2D( (surf_x_start, surf_x_finish),

176
(surf_y_start, surf_y_finish),
color='b', lw=1.5) )

# If desired, create Line patches and add to axes
if show_ray_limits:

    for i in range(system.Detector.num_pixel_H):
        depth = system.Collimator.depths[i]
        pixel = system.Collimator.starting_point + (i + 1)*np.array([-system.Detector.pixel_height * np.cos(system.Detector.angle_radians), 0, system.Detector.pixel_height * np.sin(system.Detector.angle_radians)])
        ray_x_start, ray_y_start = np.dot(rotate_matrix, [pixel[0], pixel[2]])
        ray_x_finish, ray_y_finish = np.dot(rotate_matrix, [0, -depth])
        lines.append( plt.Line2D( (ray_x_start, ray_x_finish), (ray_y_start, ray_y_finish), color='r') )

    # Add to axes
    for patch in rectangle_patches:
        ax.add_patch(patch)
    for line in lines:
        ax.add_line(line)
    ax.set_aspect('equal')

177
plt.show()
return fig, ax
def yz_system(system, row = 0, title = 'Surface Collimation',
            show_ray_limits = True):
    
    Plot a 2D view of just the Y collimators, at a specific row in the
detector.
Have the option to view rays being traced from the detector pixels
to their respective maximum depths of view for each collimator
    
    rectangle_patches = []
    lines = []

    yspan = (system.Detector.width + system.Collimator.collimator_thickness +
             system.Collimator.y_resolution)
    for fin in system.Collimator.y_fins:
        if fin.index_number <= (row + 1)*(system.Detector.num_pixel_W + 1) \
            and fin.index_number > row*(system.Detector.num_pixel_W + 1):
            fin_length = fin.length
            break

    zspan = (system.Collimator.paths[row] + fin_length)*1.5
    limits = [ [-round(yspan/2.), round(yspan/2.)],
                [-round(zspan/2.), round(zspan/2.)]  ]
    fig, ax = create_figure_2D(title, limits)
    # Detector
    det_y_start, det_z_start = (-system.Detector.width/2., fin_length)
    det_patch = patches.Rectangle( ( det_y_start , det_z_start ) ,
                                system.Detector.width ,
rectangle_patches.append(det_patch)

# Fins

for fin in system.Collimator.y_fins:
    if fin.index_number <= (row + 1)*(system.Detector.num_pixel_W + 1) \
    and fin.index_number > row*(system.Detector.num_pixel_W + 1):
        f_y = -fin.v_vector[1]
        f_z = fin.length
        f = patches.Rectangle((f_y, f_z), -fin.thickness, -fin.length,
                               fc='k', alpha=1.0)
        rectangle_patches.append(f)

    # If desired, create Line patches and add to axes
    if show_ray_limits and \
    fin.index_number < (row + 1)*(system.Detector.num_pixel_W + 1):
        y1_start, z1_start = ( f_y - fin.thickness, f_z )
        y1_finish = f_y - (system.Collimator.y_resolution + \
                          system.Detector.pixel_width + \n                          system.Collimator.collimator_thickness)/2.
        z1_finish = -system.Collimator.paths[row]

        y2_start, z2_start = ( f_y - system.Detector.pixel_width, f_z )
        y2_finish, z2_finish = ( y1_finish + \
                                system.Collimator.y_resolution, 
                                -system.Collimator.paths[row] )
def plot_system_parameters(system, variable, values, fs = 20):
    
    ... 

    Pass it a fully designed system and tell it which parameter to vary, along
    with a range of values to vary it by

    variable : Type string, attribute to vary inside system.

    sugested variable options: "num_pixel_H", "num_pixel_W", "pixel_height", 

values : (Type: array or list) range of values for variable.

depths = []
if variable in system.Detector.__dict__:
    parameter = system.Detector
elif variable in system.Collimator.__dict__:
    parameter = system.Collimator
elif variable in system.XRayFanBeam.__dict__:
    parameter = system.XRayFanBeam
else:
    print "\nRequested variable does not exist in System\n"
    return
for value in values:
    if variable == "angle":
        parameter.__setattr__(variable, value)
        parameter.__setattr__("angle_radians", value*np.pi/180.)
    elif variable == "angle_radians":
        parameter.__setattr__(variable, value)
        parameter.__setattr__("angle", value*180./np.pi)
    else:
        parameter.__setattr__(variable, value)
    system.optimize_fan_beam()
    system.optimize_collimator()
    system.system_check()
depths.append(system.depths)
var_title = ' '.join(variable.split('_')).title()
fig1, ax1 = plt.subplots()
fig2, ax2 = plt.subplots()
ax1.set_title('System View vs. ' + var_title, fontsize=1.5*fs)
ax1.set_xlabel(var_title, fontsize=fs)
ax1.set_ylabel('Depth (cm)', fontsize=fs)
pix_centers = [depth['pix_center'] for depth in depths]
full_pix_range = [depth['full_pix_range'] for depth in depths]
pix_range = [depth['range'] for depth in depths]
for i,(pixels,fpranges,ranges) in enumerate(zip(pix_centers,full_pix_range,
                                       pix_range)):
    lower_depth_fpr = [fpr[0] - pix for pix,fpr in zip(pixels,fpranges)]
    upper_depth_fpr = [pix - fpr[1] for pix,fpr in zip(pixels,fpranges)]
    ax1.errorbar([values[i]]*len(pixels), pixels,
                 yerr=[lower_depth_fpr, upper_depth_fpr],
                 fmt='ob',ecolor='g', capthick=2)
    lower_depth_r = [r[0] - pix for pix,r in zip(pixels,ranges)]
    upper_depth_r = [pix - r[1] for pix,r in zip(pixels,ranges)]
    ax2.errorbar([values[i]]*len(pixels), pixels,
                 yerr=[lower_depth_r,upper_depth_r],
                 fmt='ob',ecolor='g', capthick=2)
plt.show()
return depths

A.5 tally_reader.py

import numpy as np
class CartesianMeshTally(object):

Mesh Tally that uses x y z coordinates
(as opposite to cylindrical coordinates)
Initialize it by passing a file_path to a mesh tally output file
Matrix Shape: [num_Z, num_Y, num_X, num_energy_bins]
last energy bin is the 'Total' (over all energies)

```python
def __init__(self, file_path):
    #
    num_histories
    tally_number
    x_bin_boundaries
    y_bin_boundaries
    z_bin_boundaries
    energy_bin_boundaries
    energy
    X
    Y
    Z
    flux
    relative_error
    
    self.file_path = file_path
    f = open(file_path, 'r')
    lines = f.readlines()
    f.close()
    lines = [line.strip('
') for line in lines]
```
# grab meta data for the mesh tally
for line in lines:
    if 'Number of histories' in line:
        self.num_histories = float(line.split()[-1])
    if 'Mesh Tally Number' in line:
        self.tally_number = line.split()[-1]
    if 'X direction:' in line:
        self.x_bin_boundaries = map(float, line.split()[2:])
    if 'Y direction:' in line:
        self.y_bin_boundaries = map(float, line.split()[2:])
    if 'Z direction:' in line:
        self.z_bin_boundaries = map(float, line.split()[2:])
    if 'Energy bin boundaries:' in line:
        self.energy_bin_boundaries = map(float, line.split()[3:])
    if 'Energy' in line and 'X' in line:
        start_pos = lines.index(line) + 1

    # Start grabbing all the data for the mesh tally
energy, X, Y, Z, flux, rel_err = [], [], [], [], [], []
for line in lines[start_pos:]:
    # If there were energy bins, then there will be a "Total" over all energies at the end
    if 'Total' in line:
        total = True
        x,y,z,r,re = map(float, line.split()[1:])
        e = 1.00e36
    else:
        total = False
e, x, y, z, r, re = map(float, line.split())
energy.append(e)
X.append(x)
Y.append(y)
Z.append(z)
flux.append(r)
rel_err.append(re)

# Get dimensions for the shape of the arrays
if total:
    size_E = len(self.energy_bin_boundaries)
else:
    size_E = len(self.energy_bin_boundaries) - 1
size_x = len(self.x_bin_boundaries) - 1
size_y = len(self.y_bin_boundaries) - 1
size_z = len(self.z_bin_boundaries) - 1
shape = [size_E, size_x, size_y, size_z]

# Create Arrays
self.energy = np.flipud(np.array(energy).reshape(shape).transpose())
self.X = np.flipud(np.array(X).reshape(shape).transpose())
self.flux = np.flipud(np.array(flux).reshape(shape).transpose())
self.relative_error = \
    np.flipud(np.array(rel_err).reshape(shape).transpose())

A.6  hpc_utils.py

import system_design
import numpy as np
import os, shutil
import re
import time

def setup_rotation_simulation_dirs(mcnp_input, num_theta, TR_cards = [3],
                                  axis='Z'):
    
    mcnp_input : name of mcnp input file to be copied and manipulated
                 for each rotation (type string)

    num_theta : The number of steps to take during the scan rotation.
               If num_theta = 0 then there is only the one scan, at its
               original positon.
               If num_theta = 1 then there two total scans, one at the
               starting positon and one at 180 degrees from start.
               If num_theta = 3 then four total scans, one at every 90
               degrees
               etc....
               (type int)

    TR_cards : list of the TR card names to be altered inside the mcnp
               input file (type list, [type int] inside list)

    axis : the axis which the rotation will occur.
           Example: if axis Z then the X and Y axes will be altered
                    so that system is rotating about Z axis
           (type string)

    options: 'Z', 'X', 'Y'

    system_angles = np.linspace(0 ,
                                 360.0*(1 - 1./(num_theta + 1))),
num_theta+1)

pwd = os.getcwd()
dir_list = []

for degree in system_angles:
    path_name = os.path.join(pwd, str(int(degree))+'_degrees')
    mcnp_file = os.path.join(path_name, str(int(degree))+'_degrees.i')
    dir_list.append(path_name)
    try:
        os.mkdir(path_name)
        shutil.copyfile(mcnp_input, mcnp_file)
        os.chdir(path_name)
    except OSError:
        shutil.copyfile(mcnp_input, mcnp_file)
        os.chdir(path_name)
    f = open(mcnp_file, 'r')
    contents = f.read()
    f.close()
    for TR in TR_cards:
        regex = re.compile('\n\*TR' + str(TR) + '.*')
        original_TR = regex.findall(contents)[0]
        rotated_TR = __rotate_TR(original_TR, degree, axis)
        contents = contents.replace(original_TR, rotated_TR)
    contents = 'c\nc Imaging System Rotated by ' + str(degree) + ', degrees/c\nc

f = open(mcnp_file, 'w')
f.write(contents)
f.close()
os.chdir(pwd)
f = open('list_of_directories','w')
for line in dir_list:
    f.write("%s\n" % line)
f.close()

def __rotate_TR(TR_string, degree, axis):
    split_TR = TR_string.split(' ')
    name = split_TR[0:4]
x_axis = np.array(map(float, split_TR[4:7]))
y_axis = np.array(map(float, split_TR[7:10]))
z_axis = np.array(map(float, split_TR[10:13]))
end = [split_TR[-1]]

if axis == 'Z':
    new_x_axis = x_axis + np.array([degree, -degree, 0.])
    new_y_axis = y_axis + np.array([degree, degree, 0.])
    return ' '.join(name + map(str, np.hstack([new_x_axis,
                                               new_y_axis,
                                               z_axis ])) + end)

if axis == 'X':
    new_y_axis = y_axis + np.array([0., degree, -degree])
    new_z_axis = z_axis + np.array([0., degree, degree])
    return ' '.join(name + map(str, np.hstack([x_axis ,
                                               new_y_axis,
                                               new_z_axis ])) + end)

if axis == 'Y':
    new_x_axis = x_axis + np.array([degree, 0., -degree])
    new_z_axis = z_axis + np.array([degree, 0., degree])

return ' '.join(name + map(str, np.hstack([new_x_axis, y_axis, new_z_axis])) + end)

def add_slurm_files_to_dirs(job_sub, directory_list, num=None, resub=None, con_file='continue.i'):
    
    job_sub = Name of .slurm file template to use to submit jobs with on the HPC

directory_list = Name of file that contains a listing of all the directories that each simulation of interest is in

num = If you only want a certain index of directories from 'directory_list' to add slurm files to, then specify the start and end index with num Type: tuple (start int, end int)

resub = Type: string

    options: 'resub' (If run didnt finish. Ex: because walltime expired)

    'new_nps' (If you need to simulate more particles or want to stop at a new particle limit 'NPS')

con_file = If you choose to do 'new_nps' for resub, you need an input file telling MCNP how many particles to simulate

f = open(directory_list, 'r')
dirs = [line.strip('\n') for line in f.readlines()]
f.close()

pwd = os.getcwd()
if num == None:
    num = (0, len(dirs))

if resub == None:
    for path in dirs[num[0]:num[1]]:
        job_name = path.split('/')[1][-1]
        os.system("cp %s %s % (pwd""/+job_sub, path))
        os.system("sed -i 's/name.job""/""+job_name+".job/g' ""+
                    path""/+job_sub)
        os.system("mv "+path""/+job_sub+ "out1 "+path""/old1")
        os.system("mv "+path""/+job_sub+ "out1 "+path""/old1")
        os.system("mv "+path""/+job_sub+ "out1 "+path""/old1")
```python
def submit_slurm_jobs(directory_list, num=None, resub=None):
    
    directory_list = Name of file that contains a listing of all the
    directories that each simulation of interest is in

    num = If you only want a certain index of directories from
    'directory_list' to add slurm files to, then specify
    the start and end index with num
    Type: tuple (start int, end int)

    resub = Type: string
    options: 'resub' (If run didn't finish.
    Ex: because walltime expired)
    'new_nps' (If you need to simulate
    more particles or want to stop at
    a new particle limit 'NPS')

    
    f = open(directory_list)
    dirs = [line.strip('
') for line in f.readlines()]
    f.close()

    pwd = os.getcwd()

    if num == None:
        num = (0, len(dirs))

    if resub == None:
        for path in dirs[num[0]:num[1]]:
            job_name = path.split('/')[1][-1]
            os.chdir(path)
```

os.system("sbatch --export=file_name="+job_name+".i "+
job_name+".slurm")

print('Submitting ' + path + '
')
time.sleep(1.0)

if resub == 'resub':
    for path in dirs[num[0]:num[1]]:
        job_name = path.split('/')[-1]
        os.chdir(path)
        os.system("sbatch resub_"+job_name+".slurm")
        print('Resubmitting ' + path + '
')
        time.sleep(1.0)

if resub == 'new_nps':
    for path in dirs[num[0]:num[1]]:
        job_name = path.split('/')[-1]
        os.chdir(path)
        os.system("sbatch --export=file_name=con"+job_name+"
"+.i new_nps_"+job_name+".slurm")
        print('Continuing Simulation' + path + '
')
        time.sleep(1.0)

os.chdir(pwd)
def check_for_keyword_in_simulation_dirs(keyword=None, f_name=None,
                                          look_in='*.out'):
    keyword = The word you want to find in the 'look_in' file
    f_name = Name of file you want to save the results in. If None,
    then it will just print the matches found inside the terminal
    look_in = the file (or type of file) you want to look inside of for the
'keyword' inside all the simulation directories

if f_name == None:
    os.system('grep "' + keyword + '" ./*/look_in')
else:
    os.system('grep "' + keyword + '" -l ./*/look_in+ > temp')
    ft = open('temp', 'r')
    dirs = ft.read().split('
')[:-1]
    ft.close()
    f = open(f_name, 'w')
    pwd = os.getcwd()
    for path in dirs:
        job_name = path.split('/')[-2]
        f.write(pwd + '/' + job_name + '
')
    f.close()
    os.system('rm temp')

def remove_files_in_simulation_dirs(directory_list, file_name, num=None):
    directory_list = Name of file that contains a listing of all the
directories that each simulation of interest is in
file_name = Name of file you wish to delete
num = If you only want a certain index of directories from
'directory_list' to add slurm files to, then specify
the start and end index with num
    Type: tuple (start int, end int)
f = open(directory_list)
dirs = [line.strip(\'\n\') for line in f.readlines()]
f.close()
pwd = os.getcwd()
if num == None:
    num = (0,len(dirs))
for path in dirs[num[0]:num[1]]:
    try:
        os.chdir(path)
        os.remove(file_name)
        os.chdir(pwd)
    except OSError:
        continue

def rename_files_in_simulation_dirs(directory_list,old_name,new_name,num=None):
    
    directory_list = Name of file that contains a listing of all the
directories that each simulation of interest is in

    old_name = Current name of the file you wish to rename

    new_name = New name you want to call the file

    num = If you only want a certain index of directories from
          'directory_list' to add slurm files to, then specify
          the start and end index with num

    Type: tuple (start int, end int)

    f = open(directory_list)
dirs = [line.strip(\'\n\') for line in f.readlines()]
f.close()
pwd = os.getcwd()
if num == None:
    num = (0, len(dirs))
for path in dirs[num[0]:num[1]]:
    try:
        os.chdir(path)
        os.rename(old_name, new_name)
        os.chdir(pwd)
    except OSError:
        continue

def print_last_dump_of_MCNP_simulations():
    pwd = os.getcwd()
    fdir = open('list_of_directories', 'r')
    dirs = fdir.readlines()
    fdir.close()
    dirs = [dir.strip('
') for dir in dirs]
    for dir in dirs:
        os.chdir(dir)
        if os.path.isfile('out1'):
            f = open('out1', 'r')
            lines = f.readlines()
            f.close()
            print('
' + dir)
            print(lines[-3])
        os.chdir(pwd)

def collect_meshtallies_from_simulation_dirs(meshtally_file_name = 'meshtal',
                                            meshtally_file_dir = 'meshtally_files'):
    

meshtally_file_name = Name of the meshtally files located in each simulation directory to be moved to the folder 'meshtally_file_dir' and renamed
meshtally_file_dir = Name of directory to store all the collected meshtally files

f = open('list_of_directories', 'r')
dirs = [line.strip('
') for line in f.readlines()]
f.close()
pwd = os.getcwd()
for path in dirs:
    mesh_name = path.split('/')[-1]
    file_location = os.path.join(path, meshtally_file_name)
    if os.path.isfile(file_location):
        new_file_location = os.path.join(pwd, meshtally_file_dir,
                                         '_'.join([mesh_name, 'meshtally']))
        try:
            shutil.copy2(file_location, new_file_location)
        except IOError:
            os.mkdir(meshtally_file_dir)
            shutil.copy2(file_location, new_file_location)

def recreate_list_of_directories_file(num_theta):
    num_theta : The number of steps to take during the scan rotation.
    If num_theta = 0 then there is only the one scan, at its original position.
    If num_theta = 1 then there two total scans, one at the
starting position and one at 180 degrees from start.
If num_theta = 3 then four total scans, one at every 90
degrees
etc....
(type int)

system_angles = np.linspace(0,
    360.0*(1 - 1./(num_theta + 1)),
    num_theta+1)

pwd = os.getcwd()
dir_list = []
for degree in system_angles:
    path_name = os.path.join(pwd, str(int(degree))+'_degrees')
dir_list.append(path_name)
f = open('list_of_directories','w')
for line in dir_list:
    f.write("%s
" % line)
f.close()
function that performs everything above with just this call and returns
the array and possibly plots the image

signal = SignalMap(folder_location, full_path)
voxel_map = VoxelMap(BackscatterSystem, signal)
image_array = construct_image_array(voxel_map, signal, energy=energy,
                                use_background=use_background)
plot_image_array(image_array, BackscatterSystem, fs=10)

def construct_image_array(VoxelMap, SignalMap, use_background=False,
                           energy=None, angles=None):
    VoxelMap used with SignalMap to reconstruct an image.
can request to use signals from a specific energy in the array
can request to first filter signal by dividing it by the
background_meshtally

if energy:
    e = SignalMap.map[0].energy_bin_boundaries.index(energy)
else:
    e = -1
if angles is None:
    angles = SignalMap.system_angles_degrees
array_shape = [len(VoxelMap.x_range) - 1, # Number Pixels in X dimension
               len(VoxelMap.y_range) - 1, # Number Pixels in Y dimension
               len(VoxelMap.z_range) - 1] # Number Pixels in Z dimension
image_array = np.zeros(array_shape)
for theta in angles:
    for i,column in enumerate(VoxelMap.pix_columns):
        for j,row in enumerate(VoxelMap.pix_rows):
            for voxel in VoxelMap.map[theta][(column, row)]:
                x, y, z = voxel.coordinate
                if use_background:
                    image_array[x, y, z] += \
                    (SignalMap.map[theta].flux[j, i, 0, e]/ \
                    SignalMap.map[‘background’].flux[j, i, 0, e])*\
                    voxel.path_length
                else:
                    image_array[x, y, z] += \
                    SignalMap.map[theta].flux[j, i, 0, e]*\
                    voxel.path_length

return image_array

def construct_basis_image_array(VoxelMap, angles=None):
    
    Reconstruct an image if every signal was 1.0
    Used to help make filters for images and also see how the
    reconstruction algorithm works.
    
    if angles is None:
        angles = VoxelMap.system_angles_degrees
    
    array_shape = [len(VoxelMap.x_range) - 1, # Number Pixels in X dimension
                    len(VoxelMap.y_range) - 1, # Number Pixels in Y dimension
                    len(VoxelMap.z_range) - 1]# Number Pixels in Z dimension
    
    image_array = np.zeros(array_shape)
for theta in angles:
    for i, column in enumerate(VoxelMap.pix_columns):
        for j, row in enumerate(VoxelMap.pix_rows):
            for voxel in VoxelMap.map[theta][(column, row)]:
                x, y, z = voxel.coordinate
                image_array[x, y, z] += 1# voxel.path_length

return image_array

def divide_two_images(image1, image2):
    
    Take image1 and divide every value in its array by the corresponding
    value in image2
    
    div_image = []
    for val1, val2 in zip(image1.flatten(), image2.flatten()):
        if val1 == 0.0 and val2 == 0.0:
            div_image.append(0.0)
        else:
            div_image.append(val1/val2)
    div_image = np.array(div_image)
    return div_image.reshape(image1.shape)

class SignalMap(object):
    
    Opens folder_location and gets all the data from MCNP6 simulations and
    compiles them into a signal map that associates a system angle posiitio
    with the 2D detector array signals collected.
    
    def __init__(self, folder_location, full_path = False):
CartesianMeshTally | SignalMap/VoxelMap

0 < Z < max_Z - values < pix_rows < 0
(closer to object surface)

0 < Y < max_Y - values < pix_columns < + values

self.system_angles_degrees = []
self.map = {}

# If the full path was provided for folder
if full_path:
    self.folder_path = folder_location

# If only a path relative to the current working directory was provided
else:
    self.folder_path = os.path.join(os.getcwd(), folder_location)
tally_files = os.listdir(self.folder_path)

# Mac problems... hidden file inside folders on Mac. Get rid of it.
if '.DS_Store' in tally_files:
    tally_files.remove('.DS_Store')

for meshtally in tally_files:
    meshtally_path = os.path.join(self.folder_path, meshtally)

    # if there's a meshtally that imaged just a solid background
    # (background images help improve image quality of actual images)
    if 'background' in meshtally:
        self.map['background'] = \
        tally_reader.CartesianMeshTally(meshtally_path)

    # If it's not a background meshtally, then it should be one of the
else:
    # Store the angle in degrees
    theta = float(meshtally.split('_')[0])
    self.system_angles_degrees.append(theta)
    self.map[theta] = \
        tally_reader.CartesianMeshTally(meshtally_path)

    # Sort the angles into ascending order
    self.system_angles_degrees.sort()

    # Store the angles in an array (as opposed to a list)
    # Since VoxelMap stores its angles as an array too
    self.system_angles_degrees = np.array(self.system_angles_degrees)

class VoxelMap(object):
    
    constructs a voxel_map for a given BackscatterSystem and for the number of
    angular positions desired, provided by either an existing SignalMap or
    num_theta.

    For example, you have/want 36 unique images from the system, then request
    a num_theta=35, since theta=0 is one of the 36 images. This would be a
    system incrementing in steps of 10 degrees when rotating.
    Contains the location of all the voxels in the image to be constructed as
    well as the detector pixels contributing to that voxel and the ray traced
    path length for that specific signal.

    The coordinate system used when ray tracing looks as follows:

    XY Dimension:

        + Y  y_range[0]

        |
<-/- System rotation
/
/
/
/
/ (Theta)

-X ----------------------------------------------- + X

x_range[0] /| (X=0, Y=0) x_range[max_x]
/ |
/ |
/ |
/ |
/-> |

- Y  y_range[max_y]

XZ Dimension:

xy_length_scatter

<------------>

-X ----------------------------------------------- + X

z_range[0] /
(X=0, Z=0) /

x_range[0] / /
/ x_range[max_x]
Attributes:

 det_angle : angle the detector is at with respect to object surface

 EX: 0 degrees means detector is parallel to surface

 90 degrees means detector is parallel to fan beam
 (units: radians)

 system_angles : the angles at which the system is rotated and ray traces are solved for
 (units: radians)

 system_angles_degrees : same as system_angles but in units of degrees
 (units: degrees)

 xy_length_scatter : the max distance in the XY dimension that a photon travels when scattering from the fan beam and
heading towards the detector. This needs to be calculated because it’s possible that this distance could be greater than the width of the detector face (units: cm)

**num_pix_xy**
- number of pixels in the X and Y dimension of the image that will be reconstructed. Therefore image slices are always square

**xy_lim**
- half the physical distance the image spans. It’s half since the center of the image is at the origin (0,0) (units: cm)

**pix_columns**
- The radial location of the center of each column of pixels located in phantom/image that corresponds to the columns of pixels in the detector array. (units: cm)

**pix_rows**
- The physical location (depth) of the center of each row of pixels located in phantom/image that corresponds to the rows of pixels in the detector array. These values never change since the system's rotation is independent of the Z dimension (units: cm)

**x_range**
- The physical boundaries of the voxels that make up the reconstructed image in the X dimension (units: cm)
y_range : The physical boundaries of the voxels that make up
the reconstructed image in the X dimension
(units: cm)

z_range : The physical boundaries of the voxels that make up
the reconstructed image in the X dimension
(units: cm)

map : Dictionary containing all the different ray traces
for the image reconstruction
structure:
map = {system_angles}

| |
| |
--->{unit_v,
  unit_voxel_v,
  (column_position, row_position)}

| |
| |
---> [list of Voxels]


def __init__(self, BackscatterSystem, SignalMap=None, num_theta=None):
    # Grab relevant variables from the BackscatterSystem instance
    # Mostly doing this so that the variables being used aren’t super long
    # and take up a bunch of lines
    det_width = BackscatterSystem.Detector.width
    det_height = BackscatterSystem.Detector.height
    pix_width = BackscatterSystem.Detector.pixel_width
    pix_height = BackscatterSystem.Detector.pixel_height
num_pix_width = BackscatterSystem.Detector.num_pixel_W
num_pix_height = BackscatterSystem.Detector.num_pixel_H
self.det_angle = BackscatterSystem.Detector.angle_radians

# Determine Array of angles that system will be rotated by
if SignalMap is None:
    if num_theta is None:
        print 'Can not create VoxelMap:'
        print 'Did not provide a SignalMap or num_theta '
        return
    else:
        self.system_angles = np.linspace(0 ,
        2*np.pi*(1 - 1./(num_theta + 1)),
        num_theta+1)
        self.system_angles_degrees = np.round(self.system_angles*180./np.pi,4)
    else:
        self.system_angles_degrees = SignalMap.system_angles_degrees
        self.system_angles = self.system_angles_degrees*np.pi/180.

# Max Boundaries for the physical dimensions of the image/array of
# voxels
# Extent of image if only considering detector width
xy_length_det = det_width/2.

# Extent of image if only considering the farthest travelling scattered
# signal
self.xy_length_scatter = \
    (BackscatterSystem.depths['full_pix_range'][0][0]-
     BackscatterSystem.depths['full_pix_range'][-1][1])
\
*np.tan(self.det_angle)

# Combine lengths of detector and scattered path to get the maximum
# dimension that a signal will contribute to in the voxel array
xy_radius = np.sqrt(xy_length_det**2 + self.xy_length_scatter**2)

# round up the above value so that it is divisible by the dimensions of
# the pixels in the detector
self.num_pix_xy = int(np.ceil(xy_radius/pix_width))*2
self.xy_lim = pix_width*self.num_pix_xy/2.

# finding max depth for image
z_lim = det_height/np.cos(self.det_angle)

# array of pixel centers. One for the xy position (columns) and one for
# the z position (rows)
self.pix_columns = np.linspace(-xy_length_det + pix_width/2.,
                               xy_length_det - pix_width/2.,
                               num_pix_width)
self.pix_rows = np.linspace(0 -pix_height/(2*np.cos(self.det_angle)),
                           -z_lim+pix_height/(2*np.cos(self.det_angle)),
                           num_pix_height)

# array of boundaries that will define all the voxels that make up the
# image to be constructed
self.x_range = np.linspace(-self.xy_lim, self.xy_lim, self.num_pix_xy + 1)
self.y_range = np.linspace(self.xy_lim,-self.xy_lim, self.num_pix_xy + 1)
self.z_range = np.linspace(0, -z_lim, num_pix_height+ 1)

self.__create_map()

def __create_map(self):
    self.map = {}

    for theta, theta_deg in zip(self.system_angles,

208
self.system_angles_degrees):

# Update Unit Vector
# Unit Vector points in the direction towards detector face,
# orthogonal to the detector’s surface
unit_v = np.round(np.array( \
    [np.sin(self.det_angle)*np.cos(theta + np.pi/2),
     np.sin(self.det_angle)*np.sin(theta + np.pi/2),
     np.cos(self.det_angle)]) , # X,Y,Z coordinates
10 ) # Number if sig figs to round to

# Update the "vector" defining the direction voxels are being
# crossed. Initialize the vector to all zeroes
unit_voxel_v = np.zeros([3])

# If the unit vector has a dimension pointing in the negative
# or posistive, update the voxel vector to reflect that
# Since x_range goes from negative to positive, a positive
# unit vector will go with increasing x_range
if unit_v[0] > 0 : unit_voxel_v[0] = 1
if unit_v[0] < 0 : unit_voxel_v[0] = -1

# Since y_range goes from positive to negative, a positive
# unit vector will go with decreasing y_range
if unit_v[1] > 0 : unit_voxel_v[1] = -1
if unit_v[1] < 0 : unit_voxel_v[1] = 1

# Since z_range goes from positive to negative, a positive
# unit vector will go with decreasing z_range
# Unit vector will always be pointing in the positive direction
# (decreasing z_range)
unit_voxel_v[2] = -1

209
unit_voxel_v = map(int, unit_voxel_v)

# Initialize the theta key in self.map
self.map[theta_deg] = {'unit_v':unit_v,
                      'unit_voxel_v': unit_voxel_v}

for pix_pos_r in self.pix_columns:
    for pix_pos_z in self.pix_rows:
        # Initialize the next key in self.map
        self.map[theta_deg][(pix_pos_r, pix_pos_z)] = []

        # Initialize starting point of ray tracing
        start_coord = np.round(np.array([pix_pos_r*np.cos(theta),
                                          pix_pos_r*np.sin(theta),
                                          pix_pos_z]),
                               10)

        # Initialize starting voxel location of ray tracing
        current_voxel_id = self.__get_start_voxel(unit_voxel_v,
                                                    start_coord)

        # Start ray tracing while within voxel array
        # Once any voxel id is less than 0, it’s outside of
        # the system
        r_previous_path = 0

        while(np.min(current_voxel_id) >= 0):
            # 1) find path length (using current voxel id and voxel
            # vector)
            next_boundary = self.__get_next_boundary(
                                           unit_voxel_v,
                                           current_voxel_id)

            r_possible_paths = self.__calculate_paths(
                                           unit_voxel_v,
                                           current_voxel_id)

            # 2) Check for possible paths
            if r_possible_paths:
                # 3) Select the best path
                selected_path = r_possible_paths[0]

                # 4) Update current_voxel_id
                current_voxel_id = next_boundary[0]

                # 5) Update r_previous_path
                r_previous_path = selected_path

            # 6) Move to next voxel
            if r_previous_path == 0:
                current_voxel_id = next_boundary[1]
            else:
                current_voxel_id += r_previous_path
r_path = np.min(r_possible_paths[r_possible_paths > 0])

r_index = np.where(r_possible_paths == r_path)[0][0]

# 2) store this info (store voxel id and associated path length)
voxel = Voxel(current_voxel_id, r_path-r_previous_path)
self.map[theta_deg][(pix_pos_r,pix_pos_z)].append(voxel)

# 3) update current_voxel_id
current_voxel_id[r_index] += unit_voxel_v[r_index]
current_voxel_id = map(int, current_voxel_id)
r_previous_path = r_path

def __get_start_voxel(self,unit_voxel_v, coordinate):
    # FINDING X LOCATION
    # if photon is traveling in negative X direction and point happens to land on intersection of voxels,
    # you don’t want to count the voxel where start_point is equal to the boundary/intersection
    if unit_voxel_v[0] < 0:
        X = np.where(coordinate[0] > self.x_range)[0][-1]
    # if photon is traveling in positive X direction and point happens to land on intersection of voxels,
    # you DO count the voxel where start_point is equal to the boundary/intersection
    else:
X = np.where(coordinate[0] >= self.x_range)[0][-1]

# FINDING Y LOCATION
# Have to switch to 'less than' sign since the y_range
# starts at positive number and ends in negative
if unit_voxel_v[1] < 0:
    Y = np.where(coordinate[1] < self.y_range)[0][-1]
else:
    Y = np.where(coordinate[1] <= self.y_range)[0][-1]

# FINDING Z LOCATION
# Photon will always be traveling in the positive Z direction
# z_range is like y_range, so it switched to 'less than' sign
Z = np.where(coordinate[2] < self.z_range)[0][-1]

return [X,Y,Z]

def __get_next_boundary(self, unit_voxel_v, voxel_coordinate):
    # FINDING X LOCATION
    # if photon is traveling in negative X direction,
    # you don’t need to move the X coordinate
    if unit_voxel_v[0] < 0:
        X = self.x_range[voxel_coordinate[0]]
    # if photon is traveling in positive X direction,
    # point happens to land on intersection of voxels,
    # you DO need to move the X coordinate by 1
    else:
        X = self.x_range[voxel_coordinate[0] + unit_voxel_v[0]]

    # FINDING Y LOCATION
    if unit_voxel_v[1] < 0:
        Y = self.y_range[voxel_coordinate[1]]
else:
    Y = self.y_range[voxel_coordinate[1] + unit_voxel_v[1]]

# FINDING Z LOCATION
Z = self.z_range[voxel_coordinate[2]]
return [X,Y,Z]

def __calculate_paths(self, start_coord, unit_v, next_boundary):
    
    Solution of equation:
    o_v + r*u_v = f_v

    where o_v is the vector indicating the start of the ray
    r is the ray length (scalar)
    u_v is the unit vector indicating ray’s trajectory
    f_v is the vector indicating the stop point of the ray

    solving for r:
    r = (f_v - o_v)/u_v

    Must choose the smallest path since that is where it crosses into
    the nearest voxel

    r_possible_paths = []
    for start_pos, stop_pos, unit_dir in zip(start_coord, next_boundary, unit_v):

        # Must check if vector even has a component in a dimension
        # if not ( component = 0), then it will never cross a
        # surface in this dimension.
        if unit_dir == 0:
            r_possible_paths.append(0)
        else:
r_possible_paths.append((stop_pos - start_pos)/unit_dir)
return np.array(r_possible_paths)

class Voxel(object):
    '''Voxel class containing info for a voxel used in image reconstruction.
    coordinate is the x,y,z location of the voxel in an array being
    reconstructed.
    path_length is the path length of a specific ray that was traced through
    this voxel when creating the voxel map for image reconstruction
    '''
    def __init__(self, coordinate, path_length):
        # Copy the list 'coordinate' (Don't just reference it)
        self.coordinate = list(coordinate)
        self.path_length = path_length

def plot_image_array(image_array, BackscatterSystem, fs=20, vmin=None,
                      vmax=None, savename=None,
                      rotate_counter_clockwise=False,
                      num_angles=None, cbar=True, **kwargs):
    num_slices = image_array.shape[-1]
    rows = (num_slices + 1)/3
    columns = (num_slices + 1)/2
    if vmin is None or vmax is None:
        vmin = np.min(image_array)
        vmax = np.max(image_array)
    fig, axarr = plt.subplots(rows, columns)
    fig.set_size_inches(columns*5, rows*5)
    for i, axe in enumerate(axarr.flat):
        # In case there's an odd number of image slices, don't try and
plot an image array that doesn't exist

if i <= (num_slices-1):
    axe.set_title('Image at ' + \
    str(np.round(BackscatterSystem.depths['pix_center'][-(i+1)], 3)) +\n    ' cm Deep', fontsize=fs)

# Some images were taken in a different dimension, so image needs
# to be rotated first (by transposing the array and then
# flipping the matrix from top to bottom)
if rotate_counter_clockwise:
    cax = axe.imshow(np.flipud(image_array[:,:,i].transpose()),
    cmap='gray',
    vmin=vmin,
    vmax=vmax,
    **kwargs)
else:
    cax = axe.imshow(image_array[:,:,i], cmap='gray',
    vmin=vmin, vmax=vmax,**kwargs)

else:
    fig.delaxes(axe)

if cbar:
    cbar = fig.colorbar(cax, ax=axarr[i,j].ravel().tolist(),
    ticks=[vmin,vmax])
    #cbar.ax.set_yticklabels(["Low Signal", "High Signal"])
plt.show()

if savename:
    if num_angles:
        fig.savefig(savename + '_' + str(num_angles) + 'angles.eps',
                    bbox_inches='tight')
    else:
        fig.savefig(savename + '.eps', bbox_inches='tight')

def smooth_image_values_across_depths(image_array):
    
    Take the 2D image at each depth and divide it’s values by the mean signal for that depth. This way the values in each depth will be similar and can be correctly visualized when applying grayscale values. Should help remove the decrease in signal with depth due to attenuation.

    norm_array = np.zeros(image_array.shape)
    for i in range(image_array.shape[-1]):
        norm_array[..., i] = image_array[..., i] / image_array[..., i].mean()
    return norm_array

def normalize_images_across_depths(image_array, specific_depth = None):
    
    Take the 2D image at each depth and normalize it’s values for that depth. This way the values in each depth will be similar and can be correctly visualized when applying grayscale values. Should help remove the decrease in signal with depth due to attenuation.

    norm_array = np.zeros(image_array.shape)
    if specific_depth:
for i in range(image_array.shape[-1]):
    norm_array[..., i] = image_array[..., i]/ \n    np.sum(image_array[..., specific_depth].flatten())
else:
    for i in range(image_array.shape[-1]):
        norm_array[..., i] = image_array[..., i]/ \n        np.sum(image_array[..., i].flatten())
return norm_array

def setup_test_files(test_file='oliv_mesh_air_24B',
                     background_file='oliv_mesh_al_24B',
                     num_theta = 71, dir_name = 'reconstruction_test'):
    angles = np.linspace(0, 360*(1 - 1./(num_theta + 1)), num_theta+1)
    print 'Angles being generated: ', angles
    try:
        os.mkdir(dir_name)
    except OSError:
        shutil.rmtree(dir_name)
        os.mkdir(dir_name)
    for theta in angles:
        shutil.copy(test_file, dir_name + '/' + str(theta) + 
                    '_degrees_meshtally')
        shutil.copy(background_file, dir_name + '/meshtally_background')
REFERENCES


BIOGRAPHICAL SKETCH

Lucas Rolison was born in Kissimmee, Florida and grew up in the small town of St. Cloud, Florida. After graduating from St. Cloud High School, he attended the University of Florida and received a Bachelor of Science in nuclear engineering, graduating Summa Cum Laude; the first in his family to attend college. He then stayed at UF to receive his Master of Science and Doctor of Philosophy in nuclear engineering, as well as a certificate in scientific computing.