Exercise 3: Random Numbers and Monte Carlo

Drew Silcock
Physics Department, University of Bristol
(Dated: March 31, 2014)

I. INTRODUCTION

This exercise is split into two parts. The former half deals with generating pseudo-random number datasets that follow particular distributions. This is implemented analytically for a sinusoidal distribution, and for a general distribution using the reject-accept method. The latter half implements a Monte Carlo simulation of an unstable nucleus decaying and emitting a γ-ray, taking advantage of the analytic sinusoidal method used in the first half.

The focus of this exercise is in random number generation, in particular with reference to Monte Carlo techniques. Monte Carlo techniques involve the generation of random or pseudo-random numbers over a certain domain with a given Probability Density Function (PDF), the classic example being calculating pi by calculating what ratio of evenly distributed random numbers fall in the area of a circle to a square[1]. The general random number distributor implemented in the first part of this exercise has the purpose of providing the random number distribution that follows the desired PDF. The latter part applies the Monte Carlo technique as specified here to model the decay of an unstable nucleus travelling towards a detector. The decay time (and therefore position) and γ-ray decay angles are modelled as randomly generated numbers and the resulting detector hits are calculated trigonometrically.

II. RANDOM NUMBERS IN A DISTRIBUTION

A. Pseudo-Random Number Generators (PRNGs)

In practice, PRNGs are used instead of real hardware RNGs. Sawilowsky gave the following characteristics of a correct and useful PRNG, for use in a Monte Carlo simulation[2]:

1. the PRNG has long period before the random values begin to repeat themselves, and
2. the PRNG produces values that pass tests for randomness.

For the purposes of this exercise, NumPy's `numpy.random` functions were used, which use the Mersenne Twister (MT) algorithm as developed by Matsumoto and Nishimura in 1998[3]. These are so-called because their period of repetition is a Mersenne prime; specifically NumPy uses the MT19937 version[4], which has period $2^{19937} - 1 \approx 4.3 \times 10^{6001}$[3]. Hence the period is sufficiently large as to be effectively non-periodic and Sawilowsky's first condition is satisfied. MT also passes the standard empirical tests of the randomness of a PRNG, namely the DIEHARD and TestU01 tests[5, 6]. The MT algorithm therefore satisfies all of Sawilowsky's requirements for a PRNG.

Throughout this exercise, the seed is set by default by accessing the system's `/dev/urandom` on *nix and `CryptGenRandom` on Windows NT, thereby ensuring that values are not repeated on each run.

B. Analytic Sinusoidal Distribution

Two methods are used to produce the desired random number distributions. The first produces a sinusoidal random number distribution analytically by translating evenly distributed random numbers to sinusoidally distributed ones. This is done as follows:

Let the desired distribution, in this case a sinusoid, be $P'(x')$. The problem then is to convert between an even distribution, $P(x)$, and $P'(x')$. If the assumption is made that $P(x)$ and $P'(x')$ are properly normalised, then the cumulative distribution must be the same, so for generated value x_{gen} corresponding to required value x'_{req}:

$$\int_{x_0}^{x_{\text{gen}}} P(x)\,\mathrm{d}x = \int_{x'_0}^{x'_{\text{req}}} P'(x')\,\mathrm{d}x'. \tag{1}$$

Letting $x_0 = 0$, then:

$$\int_{0}^{x_{\text{gen}}} P(x)\,\mathrm{d}x = x_{\text{gen}}. \tag{2}$$

Defining $Q(x'_{\text{req}}) = \int_{x_0}^{x_{\text{gen}}} P(x)\,\mathrm{d}x$, Equation 1 becomes:

$$Q(x'_{\text{req}}) = x_{\text{gen}}, \tag{3}$$

, so the solution for x'_{req} is found by inverting $Q(x'_{\text{req}})$ to obtain:

$$x'_{\text{req}} = Q^{-1}(x_{\text{gen}}). \tag{4}$$

This is applied to a sinusoid, $P'(x') = \sin(x')$ for $0 < x' < \pi$, to obtain:

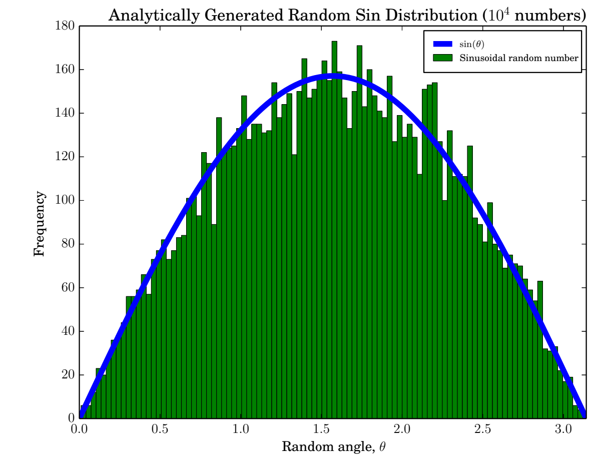
$$\begin{aligned} Q(x'_{\text{req}}) &= \int_{0}^{x'_{\text{req}}} \sin(x')\,\mathrm{d}x' \\ &= \Big[\cos(x')\Big]_{x'=0}^{x'=x'_{\text{req}}} \\ &= \cos(x'_{\text{req}}) - 1 \end{aligned} \tag{5}$$

$$\begin{aligned}\implies x'_{\text{req}} &= Q^{-1}(x_{\text{gen}}) \\ &= \arccos(1 - x_{\text{gen}}),\end{aligned} \tag{6}$$

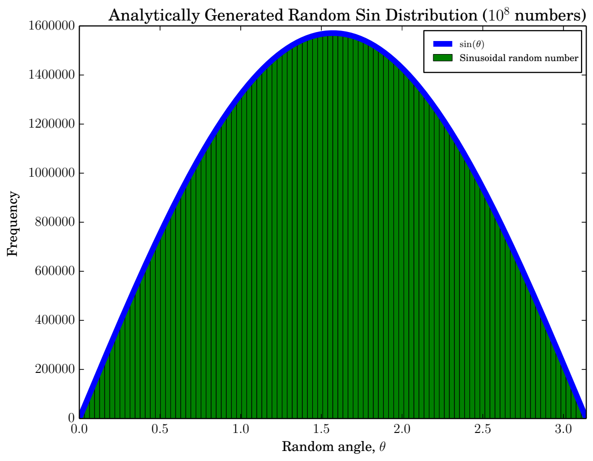
where $0 < x < 2$. Practically this was implemented using `numpy.arccos` function and `numpy.random.uniform`. Note that as throughout NumPy's routines are found to be many times faster than the alternative in-built Python loops. Here `numpy.arccos` applied to the `numpy.ndarray` scales linearly with the number of random values generated. Generating the random numbers takes 0.04 s of processor time for 10^6 numbers, 0.38 s for 10^7 numbers and 3.75 s for 10^8 values (measured to nearest two decimal places). Equally, converting these evenly distributed numbers to sinusoidally distributed ones using the analytic formula given by Equation 5 using `numpy.arccos` took 0.09 s, 0.92 s and 9.04 s of processor time for 10^5, 10^6 and 10^7 random numbers respectively.

This is an astounding factor of 25 times faster than the alternative native Python implementation, which achieves the same function with simple `for` loops and `math.acos`. This drastic difference in performance is as one would expect computationally for several reasons. Firstly, looping in Python natively in very inefficient and time consuming when dealing with large arrays of the order of one million values or so (as we are dealing with here). This is essentially because Python code is dynamic and interpreted, whereas NumPy's functions are heavily optimised `C` code, which is static and compiled. Secondly, whilst Python lists such as those containing the random numbers are dynamics arrays of pointers to objects (even when they're all the same type), NumPy arrays are densely packed homogeneous arrays. This means they have a fixed size and that each element takes up the same size in memory. This allows for heavy optimisations to operations applied to the arrays[7].

The produced random sinusoid is shown for different amounts of random numbers in Figure 1. Note that for the relatively low number 10^4 of random numbers, there is significant deviation from the perfect sin curve overlaid in blue. However, when 10^8 random numbers are produced, all random variations are ironed out and the histogram perfectly matches the overlaid sin curve. The sinusoid curve is normalised to the histogram by multiplying by the bin width and the number of random numbers that are binned. In general one must also divide by the area underneath the curve, but the area underneath $\sin(x)$ between 0 and 1 is 1, so this has no effect.



(a) 10^4 Numbers



(b) 10^8 Numbers

FIG. 1: The sinusoidal random number distribution for ten thousand and one hundred millions random numbers, produced using the analytic method.

C. Reject-Accept Method

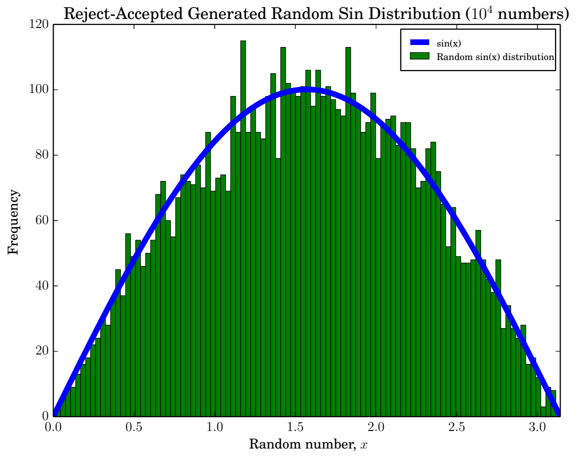
Although the analytic solution is fast, the $Q(x'_{\text{req}})$ cannot generally be analytically inverted for any function. It has to be calculated by hand. Hence another method is needed to be able to produce general distributions of random numbers. This is where the "reject-accept" method is useful. This method works by, as before, generating a random evenly distributed x in the required range. Next, another evenly distributed random number, y, is generated, between 0 and $y_{\text{max}} = P'(x'_{\text{max}})$. If the generated $y < P'(x')$, then the value x is accepted as x', else if $y > P'(x')$ then the value x is rejected. Although y_{max} can be any value $> P'(x'_{\text{max}})$, the higher y'_{max} is, the fewer x numbers are accepted and the less efficient the random distributor is.

Another way of thinking about this reject-accept method is that picking the two values x and y is picking a point in the rectangle of x length π and y length 1. All the numbers that satisfy the aforementioned condition that $y < P'(x')$ are the numbers that lie within the area under the sin curve. This means that the average number ratio of the number of produced x' values to the number of generated x values should be equal to $\frac{2}{\pi} \approx 64\%$, a number confirmed empirically in this program. Indeed this same method could be used to calculate the value of

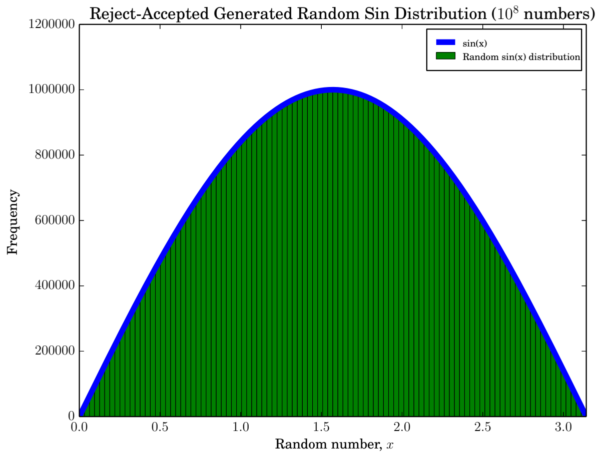
π by Monte Carlo.

This reject-accept technique is generalised to accept any valid real function that returns a single value. The plotted results when `python distributor.py 'sin(x)' -r '(0,pi)' -p` is run is shown in Figure 2. Note that because some (roughly 40%) are rejected, the reject-accept method generates fewer output random custom-distributed numbers, and is hence is less efficient. In addition, the production of these numbers takes slightly (although not orders of magnitude) longer than the analytic solution for the sinusoid: 0.17 s, 1.68 s and 16.82 s for 10^6, 10^7 and 10^8 initial random numbers, respectively. This is almost a factor of 2 more computationally expensive than the analytic counterpart. This is simply due to the incredibly simple nature of the analytic solution that it happens to be $\arccos(1-x)$ and thus can be executed very fast using NumPy's functions. The reject-accept method must not only generate some number of evenly distributed x, but also the same number of y, and hence then have to apply $\sin(x)$ to x and compare it to y. This is not extremely computationally intensive and when compared to other more complicated analytic solutions it would be significantly faster (imagine comparing this reject-accept method to the analytic inversion arising from $Q(x') = x'e^{x'}$ or another similarly complicated PDF). In addition, not all functions are analytically invertible or integrable, meaning the reject-accept method is the only viable method of random number distribution.

A reject-accept algorithm that produces a fixed number of output random numbers is also implemented. Instead of checking to see whether the number cap has been reached after each random number generation, 9 times the number needed were generated, meaning on average $\tilde{6}4\%$ of these (for the sin function) pass the test and can be used for the final distributed random numbers. This should suffice in over 99% of all runs, whilst not adding significant runtime to the program execution. Different functions vary differently as the efficiency of random user-distributed numbers produced goes is equal to the ratio of the areas under the function and in the rectangular grid of the plot in the specified range, however this proves empirically sufficient for almost all functions.



(a) 10^4 Numbers



(b) 10^8 Numbers

FIG. 2: The sinusoidal random number distribution for ten thousand and one hundred millions random numbers, produced using reject-accept method.

III. SIMULATIONS

In the final part of this exercise, a Monte Carlo technique is applied to the problem of a beam of unstable nuclei injected travelling towards a detector 2 m away at 2000 km s^{-1} with average lifetime 520 µs. The resolution of the detector is also modelled using Gaussian smearing.

A. Decay Times

As this is a discrete event occurring a known average number of times in unit time interval, the radioactive decay of these nuclei can be modelled as a Poisson distribution. The average number of decays per second, f, is the reciprocal of the average lifetime, τ, $f = \frac{1}{\tau}$. The decay times are then produced by generated a number of Poisson-distributed decays giving the number of decays in that second. These are inverted to find the lifetimes of the nuclei. As the nuclei are travelling at a constant speed of 2000 km s^{-1}, the distance from the injection point at which the nuclei decay can be easily calculated using the equation decay position = lifetime × speed. The distance travelling before the decay is thus also Poisson-distributed, with mean of 1.04 m, as shown in Figure 3.

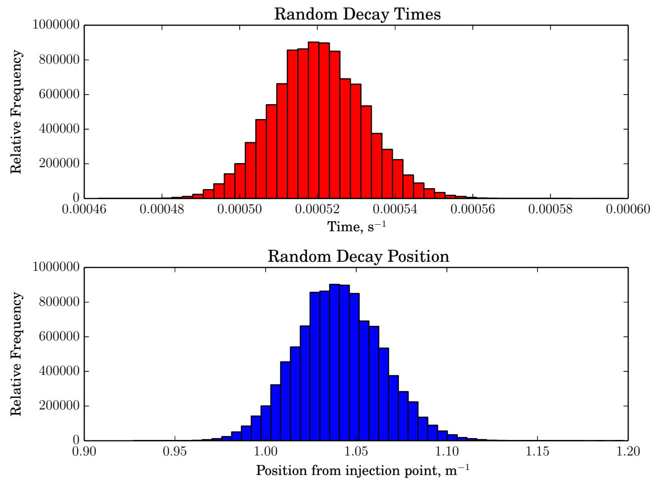


FIG. 3: Poisson-distributed decay times and positions.

B. Decay Angles

Next the angles at which the emitted γ-rays decay need be calculated. Although it may seem intuitive, it is incorrect to choose both the azimuth, φ, and the altitude, θ, to be evenly distributed. This is because the aim is to generate an area *density* of angles, or an even number of angles in unit area; the area of a small area element of a sphere is given by[8]:

$$\mathrm{d}\Omega = \sin(\theta)\,\mathrm{d}\varphi\,\mathrm{d}\theta. \tag{7}$$

Hence, whilst φ should be a uniformly distributed random number on interval $[0, 2\pi)$, θ should be sinusoidally distributed on interval $[0, pi]$. As calculated previously, φ can be distributed sinusoidally by using the analytic inversion formula given in Equation 6. In other words, if u, v are random evenly distributed variables on $(0, 1)$ as given by `numpy.random.random`:

$$\varphi = 2\pi u, \tag{8}$$
$$\theta = \arccos(2v - 1). \tag{9}$$

This effect of clumping at the poles when the points are picked incorrectly is shown in Figure 4, taken as a snapshot of the interactive Mayavi visualisation accessible via the Python script `gamma/sphere_mayavi.py`.

Now that the decay angles for the γ-ray have been correctly chosen so as to prevent clumping, all that need be done is to calculate the X position and Y position of the γ-rays when (or if) they intersect with the detector screen.

C. Detector Reading Calculations

The spherical coordinate system is set up such that x points directly towards (perpendicular to) the detector, y points parallel to the detector in the direction of $-X$

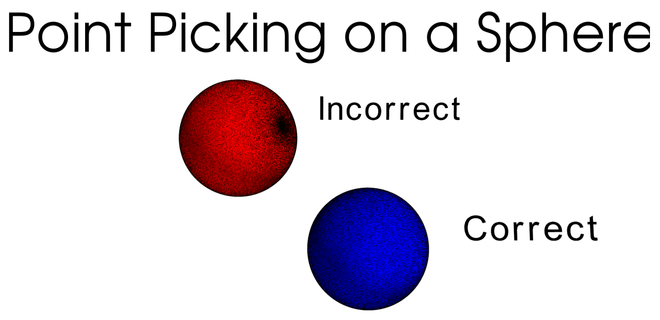


FIG. 4: An illustration of the clumping that occurs when incorrectly picking points on a sphere. The black dots represent random points. Blue sphere shows correctly distributed points; red sphere shows incorrectly distributed points.

on the plane of the detector screen and z points parallel to the plane of the detector in the $+Y$ direction on the plane of the detector screen. The standard equations for converting spherical coordinates to Cartesian are:

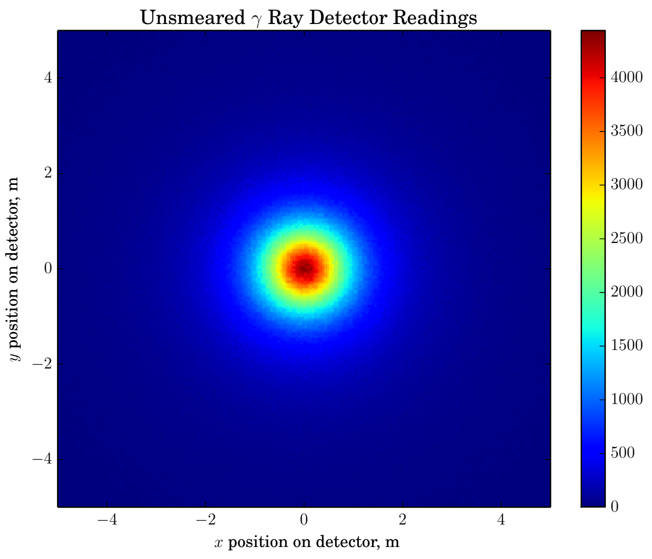
$$x = r\cos(\varphi)\cos(\theta), \tag{10}$$
$$y = r\cos(\varphi)\sin(\theta), \tag{11}$$
$$z = r\sin(\varphi), \tag{12}$$

where r is the radial coordinate. To find the position the γ-ray hits the detector, x is set to a constant, $d =$ distance from decay position to detector screen. Thus:

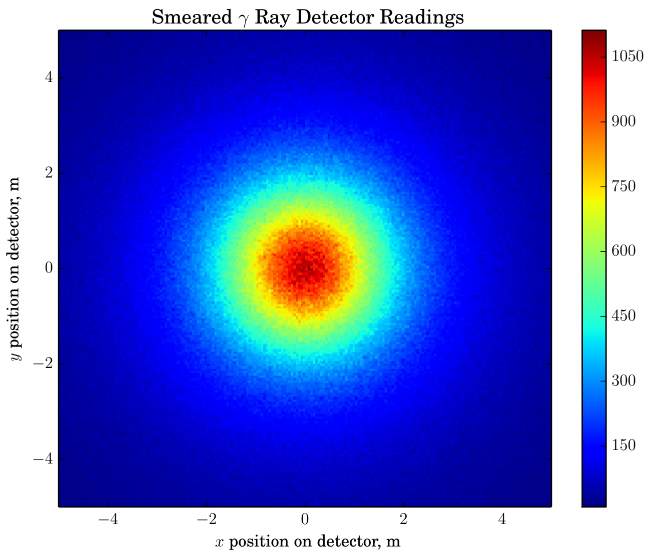
$$r = \frac{d}{\cos(\varphi)\cos(\theta)}$$
$$\implies y = d\tan(\theta) \tag{13}$$
$$\text{and } z = \frac{d\tan(\varphi)}{\cos(\theta)}. \tag{14}$$

Note that these equations require $\varphi = 0$ to be pointing along the x-axis, so $\frac{\pi}{2}$ is taken from the sinusoidally distributed value of φ so that this is the case.

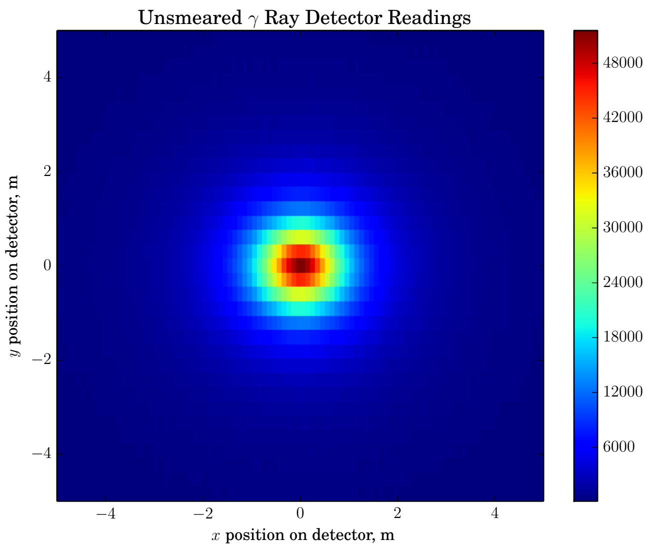
Figure 5a shows the positions of the γ-rays as they hit the detector. This clearly shows a circular pattern on the centre of the detector screen, where most rays hit the centre of the screen, and fewer hit the edges. This is exactly what one would expect from Equations 13 and 14, which essentially project the spherical envelope of randomly directional γ-rays onto the planar surface of the detector; at the centre of the screen the distance to the decay is minimal and the concentration of radiation is approximately the same as that in the small area pointing directly at the detector. In other words, map from spherical to planar surfaces has little effect. However, as the angles φ and θ increase, the corresponding angle on the surface of the detector increases yet more so that changing φ or θ a small amount causes a massive change in the position (X, Y) on the detector.



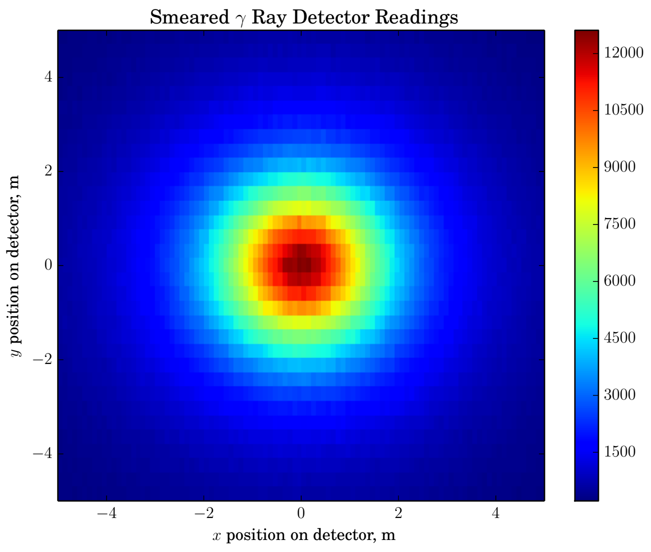
(a) 200 bins



(a) 200 bins



(b) Bins reflecting resolution



(b) Bins reflecting resolution

FIG. 5: 2d histogram of positions of γ-rays as they hit the detector, first with 200 bins, showing the real distribution of γ-ray hit locations, and secondly with bins reflecting the resolution of the detector.

FIG. 6: 2d histogram of positions of γ-rays as they hit the detector, after smearing, first with 200 bins, showing the real distribution of γ-ray hit locations, and secondly with bins reflecting the resolution of the detector.

D. Detector Resolution

The detector being modelled is said to have X and Y resolutions of 10 cm and 30 cm respectively. One representation of this is to bin the γ-ray hit locations in bins of size 10 cm by 30 cm. This is shown in Figure 5b. The positions are still clearly circular contours of decreasing number of counts.

In addition to this, the resolution of the detector, as a fundamental limiting factor to its ability to accurately record position data, is modelled by smearing the hit location data. This is done by adding a random normally distributed value to each hit location before binning, with mean equal to that hit location and standard deviation in X and Y directions equal to the resolution in that axis. The effect of this is to spread out the hits so that the counts at the centre are lowered, but the counts around the edges are increased. This is shown in Figure 6, which shows both the smeared locations with a large number of bins (Figure 6a), and with the bins reflecting the resolution (Figure 6b).

E. Improvements

There are several unaccounted for factors in this simulation that would improve the accuracy and realism of the simulation. For instance, real particle detectors have a dead time, meaning that there is a short period of time after a particle is detected that the detector cannot recognise any more signals in that area. Given the γ-rays in this simulation all travel at c, it would be possible to improve this simulation by calculating the time each particle hits the detector, and ignoring any hits a period τ after the detector recognises a signal. Silicon detectors also suffer from radiation damage, which in a limited sense could be modelled in this Monte Carlo simulation by introducing some random variables representing the changes to the characteristics of the detector and loss of hits due to the detector damage[9].

IV. CONCLUSION

In conclusion, random numbers distributed as functions are essential for Monte Carlo simulations. An analytic method of producing random numbers distributed as sin is presented, with a general non-analytic reject-accept method providing random numbers distributed as any valid user-input function. The analytic solution is then used to produce correctly evenly distributed random points on a sphere, which is used to model the random decay of a beam of nuclei, releasing γ-rays in random directions. These random angles are combined with the positions of decay to calculate the reading a detector would find, accounting for the limiting resolution of the detector.

[1] J. H. Mathews and K. K. Fink, *Numerical Methods Using Matlab (4th Edition)* (Pearson, 2004), 4th ed., ISBN 0130652482, URL `http://mathfaculty.fullerton.edu/mathews//n2003/montecarlopimod.html`.

[2] S. S. Sawilowsky, Journal of Modern Applied Statistical Methods **2**, 218 (2003), URL `http://digitalcommons.wayne.edu/coe_tbf/21`.

[3] M. Matsumoto and T. Nishimura, ACM Trans. Model. Comput. Simul. **8**, 3 (1998), ISSN 1049-3301, URL `http://doi.acm.org/10.1145/272991.272995`.

[4] P. S. Foundation, Tech. Rep. (2014), URL `https://docs.python.org/2/library/random.html`.

[5] P. Ecuyer and A. Owen, *Monte Carlo and Quasi-Monte Carlo Methods 2008*, Mathematics and Statistics (Springer, 2010), ISBN 9783642041075, URL `http://books.google.co.uk/books?id=5Y9cnmtLGQYC`.

[6] A. Jagannatam, *Mersenne Twister - A Pseudo Random Number Generator and its Variants* (2009), URL `cryptography.gmu.edu/~jkaps/download.php?docid=1083`.

[7] T. S. Community, Tech. Rep. (2014), URL `http://docs.scipy.org/doc/numpy/user/whatisnumpy.html`.

[8] E. W. Weisstein, *Sphere Point Picking*, from Mathworld – A Wolfram Web Resource, URL `http://mathworld.wolfram.com/SpherePointPicking.html`.

[9] M. Moll (1999).