49^{th} IFF Spring School: Modelling Non-equilibrium, Brownian Particles Using Python

Arvind Ravichandran

Feb 28, 2018

Overview

In this course we will use computer simulations to study Brownian particles out of equilibrium using the Python programming language. By making use of its scientific computational library, we will simulate one-dimensional and two-dimensional random walks as instances of equilibrium systems and compute the distribution of end positions. Next, we will drive particles by providing a directional bias to the noise, and study differences in the mean squared displacement (MSD). Finally, we will simulate a pair of connected harmonic oscillators, coupled to independent temperature baths. In this system energy is perpetually conducted by a spring from the hot bead to the cold bead. By calculating the flux of probabilities in state space, we will identify signatures of non-equilibrium in this system.

At equilibrium, the rates of transition into and out of a system's microstate are balanced. This is to say that the system obeys detailed balance, where *each* elementary process is equilibrated by its reverse process. Systems that obey detailed balance have transition rates between any two microstates that are *pairwise balanced*, see Fig. 1A. At equilibrium, in the entire phase space composed of many such microstates, there cannot exist any pair of states which have a net flux between them [4].

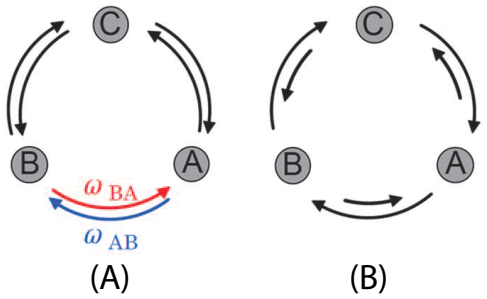

Figure 1: (A) For detailed balance to be satisfied, transitions between microscopic states must be pairwise balanced. (B) A system can break detailed balance, exhibit nonequilibrium behaviour and still be at steady state if there is a flux loop in phase space. Image from Battle *et al.* [1].

Almost all systems found in biology, however, are not in equilibrium, and detailed balance is violated at a molecular scale [5, 9, 6, 3]. Such processes are characterised by the directed fluxes through chemical states. For instance, metabolic and enzymatic processes drive closed-loop fluxes through the system's chemical states, Fig. 1B [1]. Quantifying the extent to which a system breaks detailed balance is useful, because this characterises the extent to which it deviates from equilibrium [7].

Random Walks

Random walks are stochastic processes that describe a path made up of a series of random steps. We can find various examples of random walks in nature such as the path traced by a molecule as it travels in a liquid or a gas, the price of a fluctuating stock, or the search path of a foraging animal. For an in depth perspective on mathematical modelling of random walks in biology, see Codling *et al.* [2].

Unbiased Random Walk

The simple, Markovian, random walk model is isotropic and unbiased. The walker is equally likely to move in any possible direction at any given time. We can derive the probability density function for the location of such a random walker in one-dimension in the limit of a large number of steps.

Consider a walker moving on an infinite one-dimensional lattice described by the coordinate x. The walker starts at $x = 0$ and can only take steps of length l either in the positive or negative direction. The

probability of either choice is 0.5. The probability that a walker is at $x = ml$, where m is positive and even, after taking n steps is given by the binomial distribution,

$$P(m,n) = 0.5^n \binom{n}{\frac{n-m}{2}}. \tag{1}$$

For large n, Eq. 1 converges to a Gaussian distribution after a sufficiently large amount of steps, or time $t = n\tau$. This distribution will have mean 0 and variance $l^2 t/\tau$

$$P(x,t) = \frac{1}{\sqrt{2\pi l^2 t/\tau}} \exp\left(-\frac{x^2}{2l^2 t/\tau}\right). \tag{2}$$

This is equivalent to the solution to the diffusion equation,

$$P(x,t) = \frac{1}{\sqrt{4\pi D t}} \exp\left(-\frac{x^2}{4Dt}\right), \tag{3}$$

where D is the diffusion constant. Comparing the variances of the distribution in Eq. 2 and 3, we find that $D = l^2/(2\tau)$. We can compute time-dependent statistics such as the mean location $\langle X_t \rangle$ and the mean squared displacement (MSD) $\langle X_t \rangle$, defined as,

$$\langle X_t \rangle = \int_{-\infty}^{\infty} x p(x,t) dx, \qquad \langle X_t^2 \rangle = \int_{-\infty}^{\infty} x^2 p(x,t) dx, \tag{4}$$

where X_t is the random time dependent position, and $\langle \quad \rangle$ indicates the mean or expectation value of the quantity. For the one-dimensional problem, we can show that the mean is $\langle X_t \rangle = 0$ and the variance is $\langle X_t^2 \rangle = 2Dt$. This shows us that the walker has no directional bias and its spread increases in space, linearly with time.

Biased Random Walk

We now modify the previous case by including a bias in direction and a possible waiting time between steps. For this exercise, we consider a walker moving on a one-dimensional lattice, where, at each time step τ, the walker can take steps of length l either to the left or to the right with probabilities, p and q, respectively, or stay in the same location. The waiting probability is $1 - p - q$, because the walker needs to choose from one of three possibilities. We can recover the behaviour of the unbiased random walker exactly if we set $p = q = 0.5$. The solution like that in Eq. 3 for such a system is,

$$P(x,t) = \frac{1}{\sqrt{4\pi D t}} \exp\left(\frac{-(x-ut)^2}{4Dt}\right), \tag{5}$$

where,

$$u = \lim_{l,\tau \to 0} \frac{l(q-p)}{\tau}, \qquad D = (p+q) \lim_{l,\tau \to 0} \frac{l^2}{2\tau}. \tag{6}$$

Because of the manner in which the limits are taken, $q - p$ must approach zero in order to obtain a finite value for the drift rate u. Equation 5 is the same as Eq. 3 except for the drift term in the exponential, which shifts the mean of the Gaussian in time. This suggests that the system is not in equilibrium. The first two moments of Eq. 5 are

$$\langle X_t \rangle = ut, \qquad \langle X_t^2 \rangle = u^2 t^2 + 2Dt. \tag{7}$$

In contrast to the unbiased random walk, the MSD of a diffusion process with drift has MSD that is quadratic in time similar to a system where the signal propagates as a wave, or ballistically [8].

Coupled Brownian Harmonic Oscillators

We can show the breaking of detailed balance on a purely stochastic system by numerically modelling two overdamped, tethered beads, coupled by a spring. This example is adapted from that of Battle *et al.* [1].

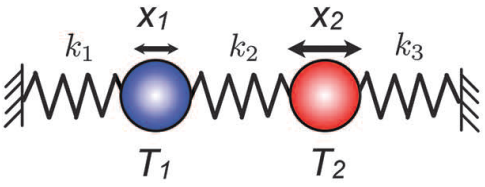

Figure 2: Overdamped, harmonic brownian oscillators coupled to independent temperature baths. Image is adapted from Battle *et al.* [1].

The two tethered beads are described by their displacements x_1 and x_2 from their equilibrium positions respectively, and the heat baths that they are coupled to are at temperatures T_1 and T_2 respectively, Fig. 2. This will lead to a heat flux from the hot bead to the cold bead *i.e.* the system evolves in time according to the stochastic forces from the temperature bath, and systematic forces between the beads, and the beads and the walls. Such an evolution gives a trajectory in state space characterised by the variables x_1 and x_2.

We can prepare a probability flux analysis (PFA) by reducing the state of the system to a coarse grained phase space (CGPS). A trajectory in the two-dimensional phase space is illustrated in Fig. 3A. Fig. 3B, shows that we can discretise the phase space into $N_1 \times N_2$ equally sized boxes, each of which represents a discrete, coarse-grained state α.

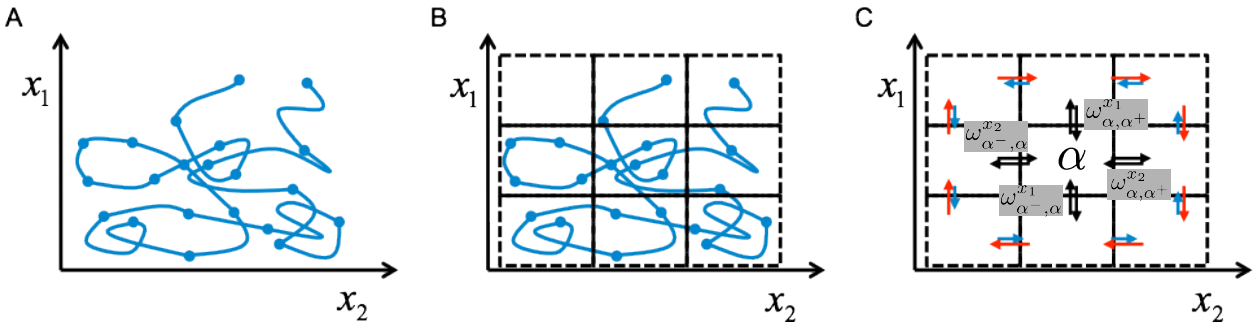

Figure 3: Schematic illustrating the construction of the CGPS. **(A)** Trajectory in phase space described by axes x_1 and x_2. **(B)** Grid illustrating the discretisation of phase space. Transitions between neighbouring discrete states occur when the system trajectory crosses box boundaries in the CGPS. **(C)** The arrows indicate transition rates across various box boundaries. They are determined by counting transitions between boxes throughout the simulation. Arrows in red correspond to clockwise fluxes and arrows in blue correspond to anticlockwise fluxes. In this example, the arrows in black correspond to transitions in relation to state α. The net rates with adjacent boxes to α are given by ω. The image is adapted from Battle *et al.* [1].

Note that even though α represents a continuous set of microstates, we can write the dynamics of the system in the CGPS using the continuity equation

$$\frac{dp_\alpha}{dt} = \omega^{x_1}_{\alpha^-,\alpha} - \omega^{x_1}_{\alpha,\alpha^+} + \omega^{x_2}_{\alpha^-,\alpha} - \omega^{x_2}_{\alpha,\alpha^+}, \tag{8}$$

where p_α is the probability that the system is in coarse-grained state α. As shown in Fig. 3C, α has two neighbouring states in each direction, resulting in four possible transitions. The net transition rates, ω, with respect to state α, used in Eq. 8, are shown in Fig. 3C. For instance the rate $\omega^{x_1}_{\alpha^-,\alpha}$ is the net rate of

transitions into state α from the adjacent state α^- upstream from α, i.e., smaller x_1, while $\omega^{x_1}_{\alpha,\alpha^+}$ denotes the rate of transitions from α downstream to α^+ (larger x_1). The same convention is extended to the x_2 axis. As is the case for fluxes, these transition rates at each box boundary, have a sign. For example, $\omega^{x_1}_{\alpha^-,\alpha} < 0$ if there are more transitions per unit time from α to α^- than the reverse.

By applying PFA, for the non-equilibrium case where $T_1 \neq T_2$ we will find significantly coordinated probability flux loops in the x_1x_2 plane of the CGPS. These flux loops demonstrate broken detailed balance, revealing the nonequilibrium nature of the system's dynamics. When $T_1 = T_2$, the system is in equilibrium and the fluxes disappear.

In order to simulate overdamped oscillators sketched in Fig. 2, we will integrate the equations of motions given by,

$$\begin{aligned} \zeta \frac{dx_1(t)}{dt} &= k\left(x_2(t) - 2x_1(t)\right) + \xi_1(t), \qquad \text{and} \\ \zeta \frac{dx_2(t)}{dt} &= k\left(x_1(t) - 2x_2(t)\right) + \xi_2(t), \end{aligned} \tag{9}$$

where $k = k_1 = k_2 = k_3$ are spring constants, ζ is the drag coefficient on the bead, and ξ_i is noise with mean 0, and variance $2\zeta k_B T_i \delta_{i,j}\delta(t - t')$. We will use the following simulation parameters: $\zeta = 18.849$, $k_B = 1$, $k = 1$, $dt = 0.1$, $T_1 = 1$.

Task 1: Simple Random Walk (SRW)

1. Unbiased Random Walk (single)
   (a) Generate an unbiased random walker in 1-D, for 10,000 steps.
   (b) Histogram all the positions of the random walker. Is this the distribution you expected?
2. Unbiased Random Walk (multiple)
   (a) Generate 10,000 unbiased random walkers in 1-D, each walker must walk 10 steps. But this time just keep track of the end point of the random walkers.
   (b) Histogram these end points, and determine the diffusion constant
3. Random Walk in 2-D
   (a) Plot the trajectory of a two-dimensional random walk
   (b) Plot the end points of many two-dimensional random walk. Do you observe circular symmetry?
   (c) Plot the histograms of positions of many random walks for N=1, N=2 and N=10. Hint: Central Limit Theorem.

Task 2: MSD of Biased Random Walk (BRW)

1. Studying a biased random walker
   (a) Compute the MSD of a 2-D random walker (unbiased)
   (b) Plot the trajectory for a biased 2-D random walker
   (c) Compute the MSD for a biased 2-D random walker
   (d) Change the biased probability to other values (*e.g.* 0.55, 0.60, 0.90), and compute the MSD. What do you observe?

Task 3: Probability Flux in State Space

1. Fluctuating Coupled Harmonic Oscillators
   (a) Set $T_1 = T_2 = 1$ and plot the X_i vs. t
   (b) Histogram X_i the and center of mass of the entire system, and check if the distribution is Gaussian
   (c) Set $T_2 = 1.5T_1$ and plot X_i vs. t
   (d) Histogram X_i and the center of mass of the entire system again. How is it different?
2. Probability Flux
   (a) Set $T_1 = T_2 = 1$ and see how it affects the probability flux in the CGPS
   (b) Set $T_2 = 1.5T_1$. Do you see the flux loops?

References

[1] C. Battle, C. P. Broedersz, N. Fakhri, V. F. Geyer, J. Howard, C. F. Schmidt, and F. C. MacKintosh. Broken detailed balance at mesoscopic scales in active biological systems. *Science*, 352(6285):604–607, April 2016.

[2] E. A. Codling, M. J. Plank, and S. Benhamou. Random walk models in biology. *J. R. Soc. Interface*, 5(25):813–833, April 2008.

[3] D. A. Head, W. J. Briels, and G. Gompper. Nonequilibrium structure and dynamics in a microscopic model of thin-film active gels. *Phys. Rev. E*, 3(89):032705, March 2014.

[4] T. L. Hill. *Statistical Mechanics: Principles and Selected Applications*. Dover Publications Inc., New York, 1987.

[5] G. H. Koenderink, Z. Dogic, F. Nakamurae, P. M. Bendix, F. C. MacKintosh, J. H. Hartwige, T. P. Stossele, and D. A. Weitz. An active biopolymer network controlled by molecular motors. *Proc. Natl. Acad. Sci. U.S.A.*, 106(36):15192–15197, September 2009.

[6] W. Lu, M. Winding, M. Lakonishok, J. Wildonger, and V. I. Gelfand. Microtubule–microtubule sliding by kinesin-1 is essential for normal cytoplasmic streaming in drosophila oocytes. *Proc. Natl. Acad. Sci. U.S.A.*, 113(34):E4995–E5004, August 2016.

[7] V. Mustonen and M. Lässig. From fitness landscapes to seascapes: non-equilibrium dynamics of selection and adaptation. *Trends Genet.*, 25(3):111–119, February 2009.

[8] A. Okubo and S. A. Levin. *Diffusion and ecological problems: modern perspectives*, volume 14. Springer Science & Business Media, 2013.

[9] W. E. Theurkauf. Microtubules and cytoplasm organization during drosophila oogenesis. *Dev. Biol.*, 165(2):352–360, October 1994.

These notes were prepared for the 48th IFF Spring School. In case of errors, questions or comments, please contact the author, a.ravichandran@fz-juelich.de.

An Introduction to Python

Python is a widely used high-level programming language that has a very wide user base. Almost any problem that you face in Python can be quickly looked up on *Stack Overflow*, and is possibly the quickest way to learn the language. Although it is definitely not recommended to use Python to perform large scale molecular dynamics simulations, for the purpose of this lab course, it is an excellent lab space to play with data structures and construct a simple working example of non-equilibrium phenomena.

Python Tools

We will begin by going through some tools available in Python to simulate a 2-D random walk.

1. Press ⌘ + Space to open Spotlight
2. Type in *terminal* in the Spotlight search bar and then Enter ↩
3. Type *spyder* within the terminal window and then Enter ↩

You will see three windows: the editor window, the iPython console and another window with two tabs (Variable explorer and File explorer). Use the iPython console to type in some of the commands that you see in the boxes in the following section and get a feel for the language.

An (Incredibly) Brief Introduction to Python

If you have never used Python before, please follow this section to familiarise yourself with performing some simple tasks in python. We will begin by understanding how to assign values to variables, and manipulate them. This tutorial is written for Python 2.7. Some features, such as the division operator (/), *always* returns a float as of Python 3.6.1.

```
>>> print "hello world" # printing a string
hello world
>>> a = 5               # assigns the integer 5 to variable a
>>> b, c = 10, 12       # assigns 10 to b and 12 to c
>>> a = a*2             # doubles the value of a, and assigns it back to a
>>> print a
10
>>> 10/4                # using python as a calculator (int/int = int) --> Python 2.7
2
>>> 10/4.               # always note the type of your operands (int/float = float)
2.5
```

We can use different data types in Python, and they are categorised into being either mutable or immutable. A list of some instances of these data types are given in Table . We have already seen examples of strings (strs), ints and floats. In this course, we will be making use of lists and something known as NumPy arrays, which as the name "array" suggests is a mutable collection of ints, floats *etc.* We will get to NumPy in the next section.

Immutable Types	Mutable Types
int, float, long	list
str	set
tuple	dict

Lists

One of the most versatile data type that can be used to group together values is known as the list. It can be written as a list of comma-separated values (items) between square brackets. Lists might contain items of different types, but usually the items all have the same type. We can index each of the elements in a list, starting with 0. (Keep in mind that this method of indexing, starting with 0 is commonplace in most languages and is unlike *Matlab*, which begins indexing with 1.) From here on, I will call the 0th element the first element in this tutorial.

```
>>>  squares = [1, 4, 9, 16, 25]  # Define a list of squared values
>>>  squares                      # Another way of printing in your IDE
[1, 4, 9, 16, 25]
>>>  squares[3]                   # Call the fourth element
16
>>> squares[-1]                   # Negative numbers start counting from the back
25
```

Lists allow us to slice them in various ways. If we want certain sections of the list, we can choose them appropriately by using the semicolon operator, which will return another list. [0:3] will give us the first four elements. We can slice entire arrays starting from (or ending at) specified points by using the semicolon function appropriately. For instance, [3:], would be elements in the list starting from the fourth element, until the end. [:3], on the other hand will give all elements from the first to the fourth.

```
>>> squares[0:3]                  # Pick first 4 elements
[1, 4, 9, 16]
>>> squares[-3:]                  # [-3:] would read "Third last element till the end"
[9, 16, 25]
```

Lists also support operations like concatenation, using the + operator. You can also add new items at the end of the list, by using the *append()* method, and the length of the list is given by *len*.

```
>>> squares + [36, 49, 64, 81, 100]
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
>>> squares.append(121)        # add the square of 11 to the list
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121]
>>> len(squares)
11
```

It is possible to nest lists (create lists containing other lists), for example:

```
>>> a = ['a', 'b', 'c']  # list of strs
>>> n = [1, 2, 3]        # list of ints
>>> x = [a, n]           # x is now a compound list of two lists
>>> x
[['a', 'b', 'c'], [1, 2, 3]]
>>> x[0]                 # call for the first element
['a', 'b', 'c']
>>> x[0][1]              # call for the first element of the first element
'b'
```

Control Flow Tools

Loops

Most of the calculations that you do in science will involve working with a recursive loop, known commonly in programming as a "for loop". In Python, it will help to remember for loops don't work like how they do

in other languages. As a thumb rule, whenever you write a for loop in python, try saying: *for each element in the list, perform ...* For instance:

```
>>> colors = ['red', 'blue', 'green']    # list of colors
>>> for color in colors:                 # for each color in the list colors
....:     print color+' tortoise'         # append the word tortoise and print
....:
red tortoise
blue tortoise
green tortoise
```

Let us try to generate the list squares that we clumsily typed in the previous example. A useful tool for this could be the enumerate function, which assigns a number to each element in your list. Note that functions such as enumerate will return an *iterator*. Iterators are constructions in Python that can only be iterated through. You cannot print an iterator like how you print a list. Let's see how it works:

```
>>> numbers = range(4)          # range makes a list of numbers
>>> numbers
[0, 1, 2, 3]
>>> enumerate(numbers)          # enumerate gives you an iterable
<enumerate object at 0x1028a90a0>
>>> for i in enumerate(numbers):
....:     print i
(0, 0)
(1, 1)
(2, 2)
(3, 3)
>>> for (i,n) in enumerate(numbers):
....:     numbers[i] = n*n      # access each element
>>> print numbers
[0, 1, 4, 9]
```

Something that is a little peculiar to Python which will make things much faster is what is known as a list comprehension. Loops written as above is not "Pythonic", and one can use list comprehensions to achieve the same effect:

```
>>> numbers = [ n*n for n in numbers]    # "do n*n for each element
                                         # called n in the list numbers"
>>> print numbers
[0, 1, 4, 9]
```

See how much cleaner this looks?

if Statement

One last thing to learn before we dive into Random Walks. Loops are wonderful to go through or generate large chunks of data, that you might encounter in your simulations. But you need conditions, with which you can categorise your calculations to only do certain things at certain times. You need if constructs to do this. There is nothing too difficult about how Python implements if constructs. Run through the following example that I have adapted from the Python docs tutorial:

```
>>> x = int(input("Please enter an integer: "))
Please enter an integer: 42
>>> if x < 0:
...     x = 0
```

```
...     print('Negative changed to zero')
... elif x == 0:
...     print('Zero')
... elif x == 1:
...     print('Single')
... else:
...     print('This is a really big number') # aka bigger than 1
```

Random Arrays and Plotting

NumPy is the fundamental package for scientific computing with Python. And we will use this package for all simulations in this course. The constituting tools in NumPy that will help you in simulating a random walk are explained in this section. We will use the numpy.random function to generate randomness in our system.

```
>>> import numpy as np                          # import the numpy package
>>> np.random.random()                          # Return random floats in the half-open
                                                # interval [0.0, 1.0).
0.9522182295453697
>>> np.random.choice([-1,1])                    #  make a random choice between -1 and 1
1
>>> np.random.choice([-1,1], size=10)           # prepare an array of
                                                # 10 values of either -1 or 1
array([-1,  1, -1,  1,  1, -1,  1, -1,  1, -1])
>>> a = np.random.choice([-1,1], size=(10,2))# an array of 10x2
>>> a
array([[ 1,  1],
       [ 1, -1],
       [-1, -1],
       [ 1, -1],
       [-1,  1],
       [ 1, -1],
       [-1,  1],
       [-1,  1],
       [-1, -1],
       [-1,  1]])
>>> a.shape          # NumPy arrays have a property called shape
                     # which is the size of the array
(10, 2)
```

We can sum up these numbers using the np.sum or the cumulative sum using np.cumsum functions respectively.

```
>>> b = np.random.choice([-1,1],size=5)
>>> b
array([ 1, -1, -1,  1,  1])
>>> np.sum(b)
1
>>> np.cumsum(b)
array([ 1,  0, -1,  0,  1])
```

In the event that you encounter a function that you are required to use, and you are not familiar with the syntax necessary, type the function with a question mark, to take a look at the documentation for the function.

```
>>> np.random.normal?
'''
Docstring:
normal(loc=0.0, scale=1.0, size=None)

Draw random samples from a normal (Gaussian) distribution.
...
'''
```

This gives you a lot more information on top of telling you that by default, you will get a distribution with mean of 0 (loc=0.0), and a standard deviation of 1.0 (scale=1.0), if you don't specify anything. You can also see that by default you get one value (size=None).

Taking a look at the documentation for np.random.choice, we see that we can bias choices by specifying the probabilities of each option. This will become useful for the biased random walk section of the workshop.

```
>>> np.random.choice?
'''
Docstring:
choice(a, size=None, replace=True, p=None)
...
Parameters
-----------
...
p : 1-D array-like, optional
    The probabilities associated with each entry in a.
    If not given the sample assumes a uniform distribution over all
    entries in a.
...
'''
```

Plotting Histograms

Histogramming, and plotting these histograms will be the last tool necessary for us to proceed to the first task. We can easily generate 1000 (size=1000) gaussian distributed numbers using the np.random.normal function, and histogram them using plt.hist.

```
>>> import matplotlib.pyplot as plt     # for plotting
>>> gaussian_numbers = np.random.normal(size=1000)     # prepare an array of
                                                       # gaussian distributed numbers
>>> plt.hist(gaussian_numbers, bins=10, normed=True)
>>> plt.show()
```

The resulting figure is shown in Fig. 4.

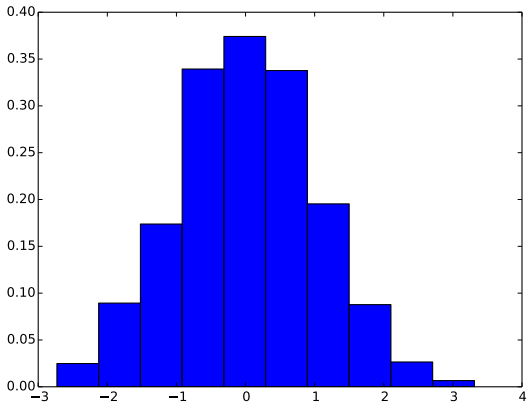

Figure 4: Histogram generated using matplotlib for a set of gaussian numbers.