PHY407 Lab 5

Monday, December 2, 2024

Jace Alloway - 1006940802 - alloway1

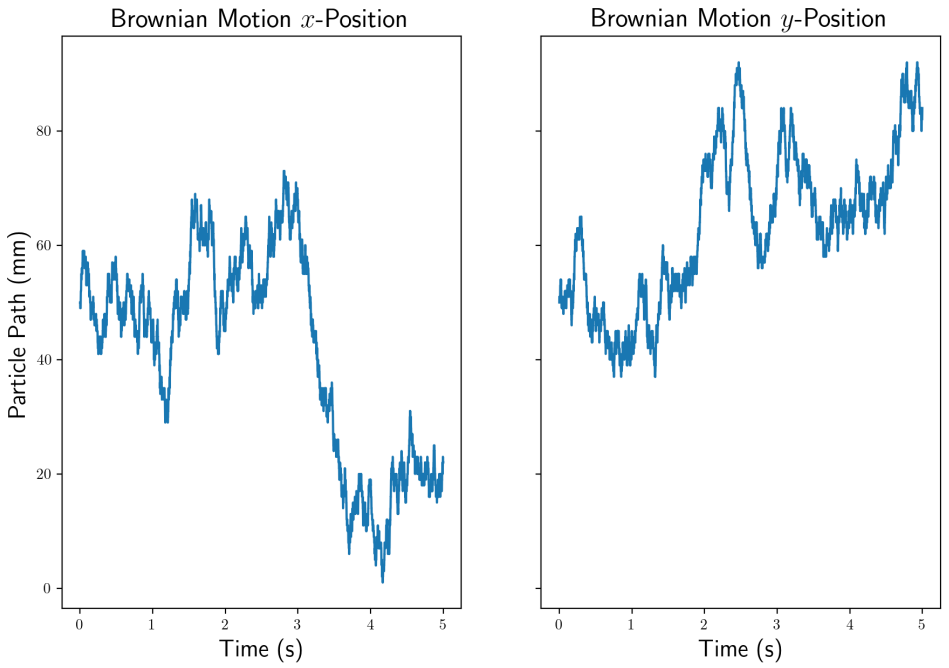
This assignment was compiled in VS Code using the Python and LaTeX extensions. Matplotlib does not produce inline plots in terminal, so the `%matplotlib inline` command was commented out. **Collaborators for all questions: none.**

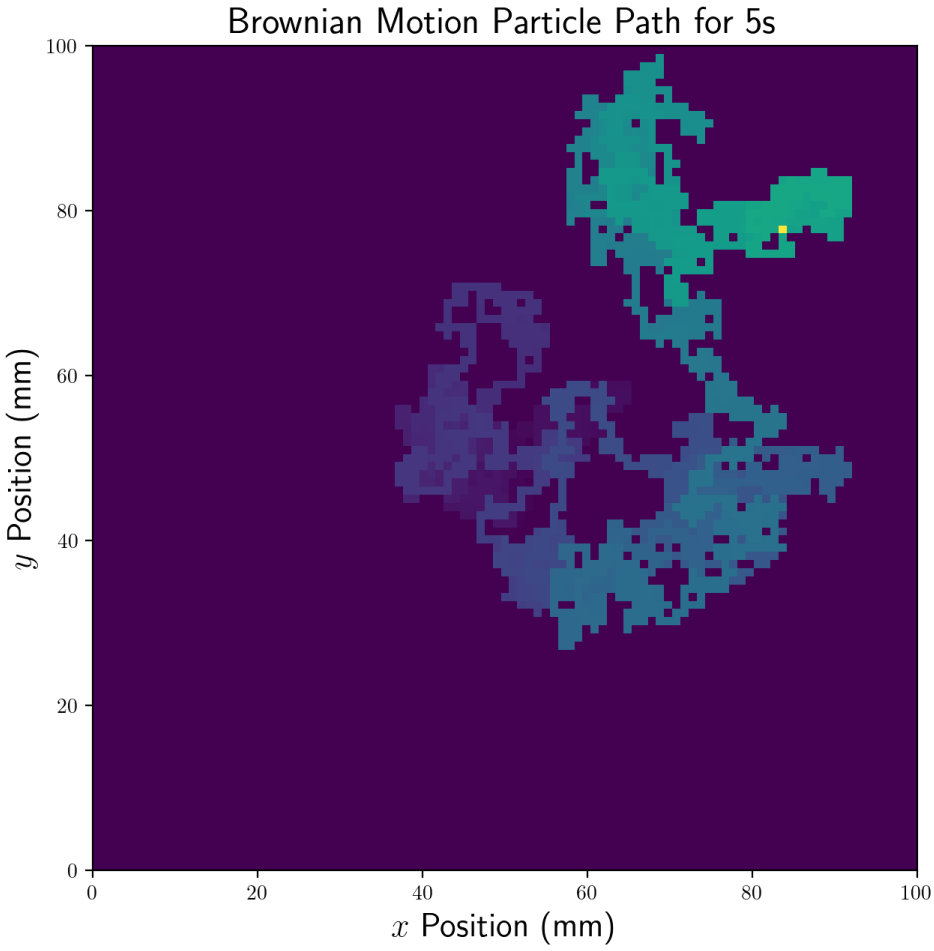
Problem 1

(a) The Diffusion-Limited Aggregation (DLA) process involves generating random steps at each timestamp to move a particle over grid cells in an $N \times N$ domain. The order of steps (pseudocode) was implemented as

1. Generate axis grids using `np.linspace` and `np.meshgrid`
2. Define a function '`get_cell_action`' which takes in an old cell position and returns the list of possible moves.
3. Define a function '`next_cell`' which takes in the current cell and returns a random move in a direction given by `get_cell_action`.
4. For animation, define a '`draw_pos`' function which takes in the current cell and list of previous cell placements. It will return a grid to plot the particle position as a function of frames using `plt.imshow`.

(1.) was completed using `np.meshgrid()`. We begin (2.) by indexing the current cell and getting the lengths of each sides of the grid (in the case we are not working in an $N \times N$ grid). If the input cell is outside the bounds of the specified grid, a valueError is raised. For the x and y axes independently, we check whether the cell is along an edge/corner or in the center. The function returns the corresponding possible moves as an array of $(x_{\text{out}}, y_{\text{out}})$ values. Hence, if the total length of the output is 4, the particle is in the center. If it is 3, it is along an edge, and if it is 2, it is in a corner.
For (3.), we first take in the current cell then use a nested function call by calling the `get_cell_action` function to return the list of possible moves. We label the total number of possible moves (in any direction) K, so that a random integer $[0, K]$ inclusive is generated and yields a direction to move. If K is even, it is either 2 (corner) or 4 (center). If K is odd, it is 3, and we must compare the lengths of the x and y returned arrays to determine which is greater. For instance, 0 implies a move north, 1 implies east, 2 south, and 3 west. The new index determined by `get_cell_action` replaces the old index, and only one move is allowed at a time.
For (4.), the drawing function is written by defining a new grid with the same dimensions, and populating the (i, j) position with a 1 for current position, and weighting the previous positions of the particle by using the length of the history array. The grid is returned so that is can be plotted/animated with `plt.imshow`.
The process was animated using 5000 frames and using `matplotlib.animation`, however only the final result is shown here. The x and y histories of the particle were generated and plotted, using $(50, 50)$ as an initial particle position with a 100×100 grid. Lastly, the total event history of the particle was plotted by calling the weighted draw function. Sample plots are included here for 5s of simulating:

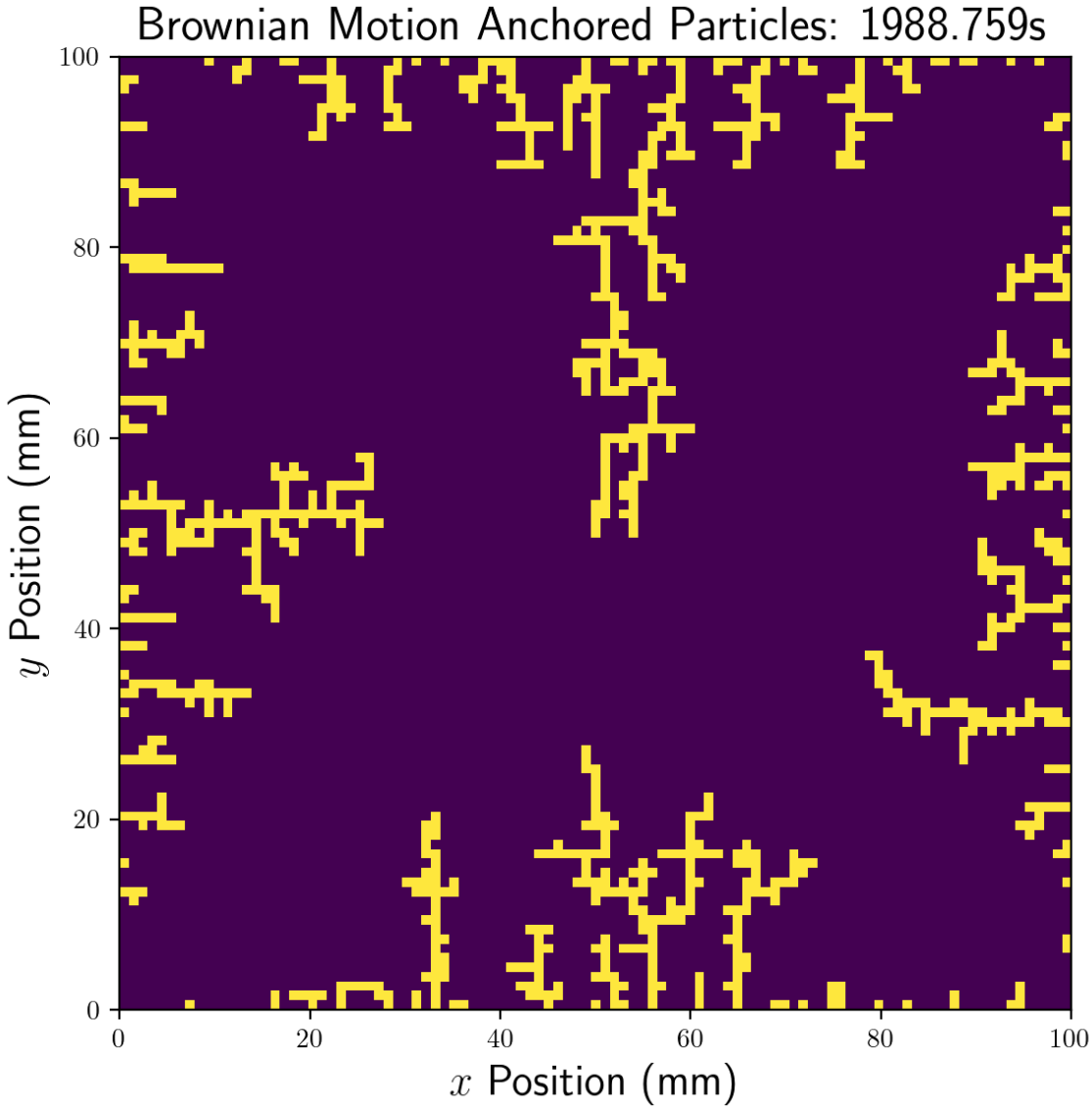




(**b**) For part (b), a similar process was implemented, using `get_cell_action` and `next_cell` func-

tions to determine the particle evolution. The difference between the functions defined in part (b) and in part (a) is that we now introduce another grid, called the 'anchored' grid, which records the (i, j) positions of all the anchored particles. In the `get_cell_action` function, we also introduce a bool value, `anchored = True | False`. The default is False. Now, if the particle is beside wall or corner, or if the cell (i, j) is located next to an `anchored[i +/- 1][j +/- 1] = 1` cell, then the 'anchored' bool returns True. The `next_cell` function is identical to that as the function defined in part (a).

To present the data, we use the same draw function as in (a), but draw every anchor using the anchored bool key. For each particle, we loop the randomization step process until it becomes anchored. It is then added to the anchored grid. It is then repeated for another particle, until `anchored[50][50] = 1` (center cell contains an anchored particle). The history of each particle is also recorded. For the 100×100 grid, the resulting anchor grid was plotted. A sample process is located below:



Problem 2

(a) The function

$$f(x, y) = x^2 - \cos(4\pi x) + (y - 1)^2 \tag{2.1}$$

was defined in Python. To implement the stepping process, a '`generate_step`' function was defined to draw two random numbers from a normal distribution with 0 mean and unit deviation and added to each input i and j coordinates. The cooling schedule and it's inverse were defined according to

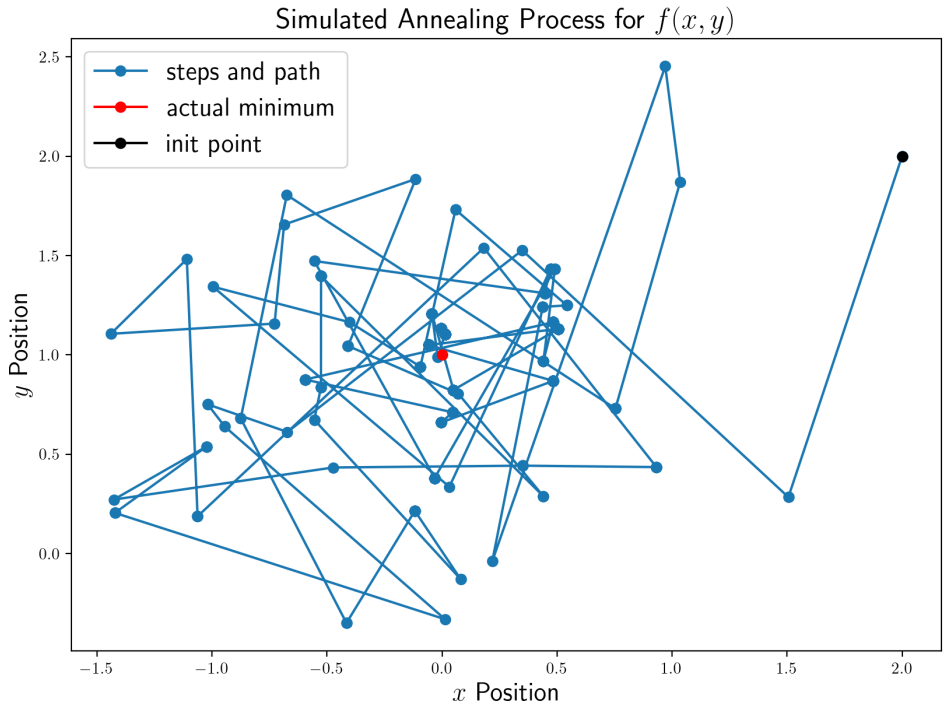
$$t = \tau \log(T_0/T_f) \tag{2.2}$$

$$T(t) = T_0 e^{-t/\tau} \tag{2.3}$$

and the time array was defined using `np.linspace` from 0 to t at the given parameters of (T_0, T_f, τ). We note that a finite number of time points was required for discretization (more on this later), and each time point implied one iteration of choosing new coordinates, then computing the acceptance probability. The acceptance probability function was defined using the functional values of f as the energy values:

$$\mathbb{P}(E_i, E_j, T) = \begin{cases} 1 & \text{if } E_i \geq E_j \\ e^{-(E_j - E_i)/T} & \text{if } E_i < E_j \end{cases}$$

where E_i is the energy prior to the move, and E_j is the energy after the move. T is the temperature at the timestep t. Note that the parameters (T_0, T_f, τ) govern the acceptance probability in the case the system desires to make a jump to a greater energy level. For low values of T, this is essentially 0. But for high T, nearly any move will be accepted. The number of time points set by `np.linspace` is important because the more time points, the more likely the system will eventually move to the minimum depending on the cooling rate. For part (a), I took $N = 1000 = \tau$ time points in the time array (I found that for smaller values of N, say, 100-200, the minimum located was not accurate). A sample plot of the step history is shown below

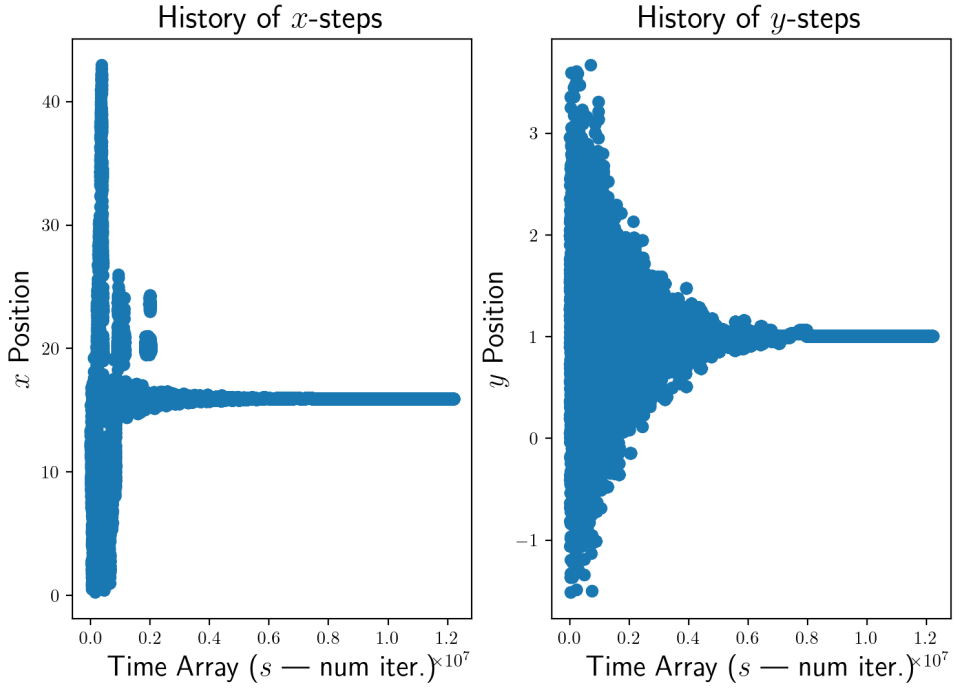


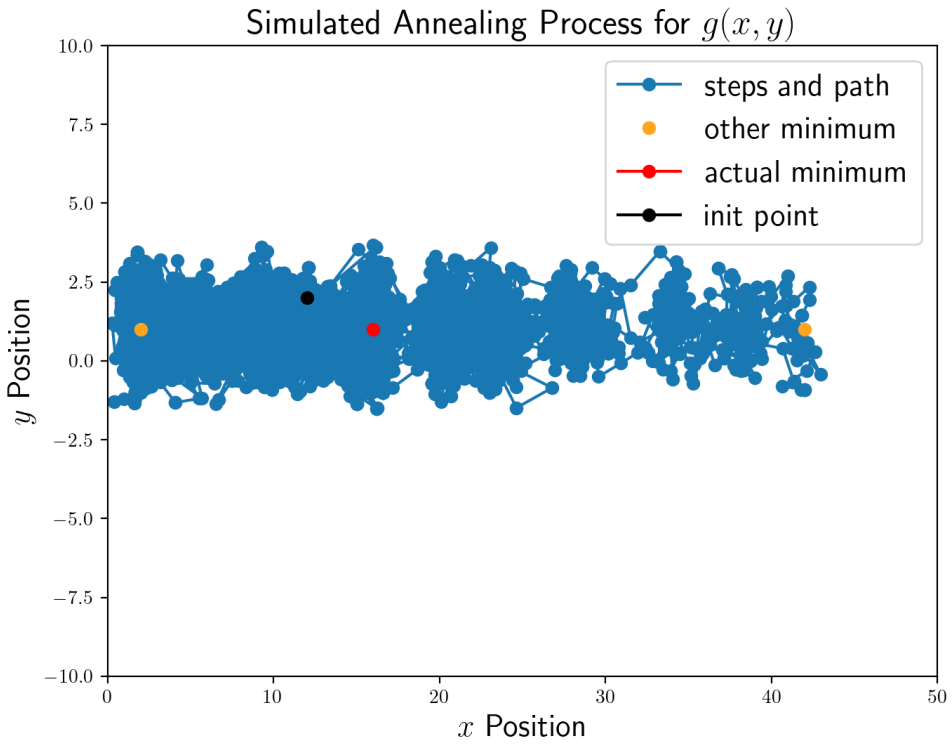
with output $(T_0, T_f, \tau) = (1, 10\text{e}-6, 1000)$. The corresponding output was $(-0.0017, 0.9405)$ which is approximately $(0, 1)$.

(b) The same process was implemented as in part (a), but the new function that was defined as the energy distribution was

$$g(x, y) = \cos(x) + \cos(x\sqrt{2}) + \cos(x\sqrt{3}) + (y-1)^2. \tag{2.4}$$

and any values outside the $0 < x < 50$, $-20 < y < 20$ domain were rejected in the loop. The initial point used was $(12, 2)$ because it was close to the actual minimum. After a lot of trial and error, it was eventually found that a very large time array must be used, to 'anneal' the function very slowly. I spent a lot of time on this problem not realizing how slow this process actually needed to be. Eventually, it was found that $\tau = 10 \times 10^8$ with $T_0 = 2$, $T_f = 10 \times 10^{-6}$ (same as part (a)) and 10×10^7 timesteps produced the same result and the actual minimum nearly every time. Two plots were produced due to the sheer number of iterations: one which records x and y positions over time, and the searching position such as the plot in part (a). A sample output is included.





The corresponding minimum was found to be $(15.9494..., 1.0050...)$.

Problem 3

(**a**) To compute the integral

$$\int_0^1 dx\, f(x) \tag{3.1}$$

where

$$f(x) = \frac{1}{\sqrt{x}(1+e^x)} \tag{3.2}$$

is the integrand. We first use the mean value method to compute the integral for 1000 iterations using $N = 10'000$ sample points, then appending them to an array and using `np.mean` to return an average value. The method was implemented via drawing a uniform random number $x_i \in [0, 1)$ for each sample point, the summing the integral

$$I = \frac{b-a}{N} \sum_{i=1}^{N} f(x_i) \tag{3.3}$$

where $a = 0$ and $b = 1$ were the bounds of the integral. This value of I was appended to an array, from which the mean was taken after 1000 iterations were taken out. The returned value from the mean was $I \approx 0.839395$.

(**b**) We now compute the same integral in (3.1) with the same f as in (3.2) using the same number of iterations and sample points, but now implementing the importance sampling calculation. The weighted function

$$w(x) = \frac{1}{\sqrt{x}} \tag{3.4}$$

was defined so that

$$\frac{f(x)}{w(x)} = \frac{1}{1+e^x} \tag{3.5}$$

did not contain a singularity at $x = 0$. We note that

$$\int_0^1 dx\, w(x) = \int_0^1 \frac{dx}{\sqrt{x}} \tag{3.6}$$

$$= 2\sqrt{x}\Big|_0^1 \tag{3.7}$$

$$= 2 \tag{3.8}$$

so that we can define a probability density function

$$p(x) = \frac{w(x)}{\int_0^1 dx\, w(x)} = \frac{1}{2\sqrt{x}}. \tag{3.9}$$

Since the probability of finding a number between $(x, x+dx)$ is $p(x)dx$, then the cumulative probability over the whole interval is given by the cumulative density function,

$$\int dx\, p(x) = \sqrt{x} \tag{3.10}$$

evaluated between 0 and 1, which is 1. Hence any random number a drawn from a uniform distribution maps to the probability as $\sqrt{x}$, which implies that the required x for drawing a random number from the probability density function (3.9) to evaluate the integral is given by $x(a) = a^2$.

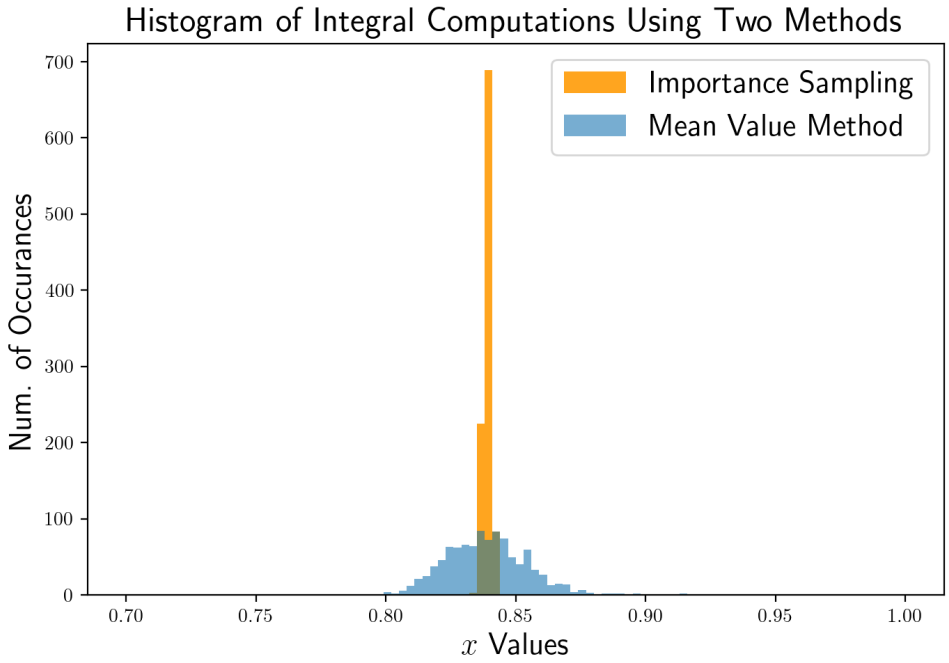
Hence, this mapping was defined in Python to map uniformly distributed numbers onto the cumulative probability function, and hence the probability density. The integral was computed as

$$I = \frac{1}{N}\sum_{i=1}^{N}\frac{f(x_i)}{w(x_i)}\int_0^1 dx\, w(x) \tag{3.11}$$

$$= \frac{2}{N}\sum_{i=1}^{N}\frac{f(x_i)}{w(x_i)} \tag{3.12}$$

which were also appended to an array for 1000 iterations and the mean was taken. It was found that $I \approx 0.839045$.

(c) A histogram of the integral arrays IA from part (a) and IB from (b) were plotted with 100 bins. I took the range of points to be $(0.7, 1)$ since I found that not many integral values fell outside this range and wanted to give room for the plot. The histogram plotted is shown below.



We note the difference between the two methods: the variance from the importance sampling computations was much smaller than that of the mean value method computations. This is most likely because the mean value method is uniformly random, which can give a greater variance due to unbiased randomness. However, taking points from a weighted average reduces variance since the values from the chosen distribution may be different than values drawn from the original distribution, which can remove bias and yield a more accurate answer. In the case of our integral, the initial mapping favoured points closer to 0 due to the divergence of the integrand, so these values would be considered of 'greater weight' than desired. It is for this reason some integral values may be very large when estimating a divergent distribution. However, using the weights and dividing out the singularity (later re-introducing the weights in the sum via the means of an average) removes the bias of divergence for x values close to 0. Hence the integral will be more accurate.

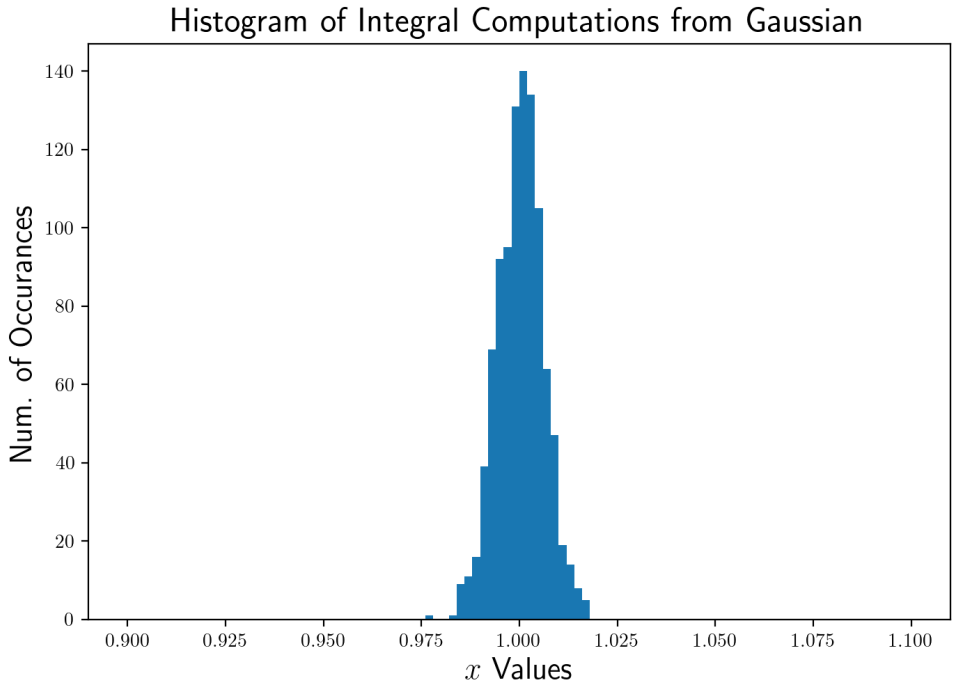
(d) We now compute the integral

$$\int_0^{10} dx\, e^{-2|x-5|} \tag{3.13}$$

using the weighted function

$$w(x) = \frac{1}{\sqrt{2\pi}} e^{-(x-5)^2/2}. \tag{3.14}$$

Using the same number of iterations and sample points as in parts (a) and (b) to average, we draw random numbers from the sample distribution `np.random.normal(5, 1)` (mean 5, variance 1), since (3.14) integrates to approximately 1 over the whole domain $[0, 10]$, so we can take it to be $p(x)$. This implies that we need not define another map from a uniform distribution to non-uniform, since the Gaussian is already non-uniform and governs the weighting. Using the exact same code as in (b), we find the averaged integral from the 1000 iterations to be approximately ≈ 1.000320. A histogram of the integral arrays was plotted, again using 100 bins over an appropriate range:



Again, I chose a larger range to account for some variances in the integral calculations. That being said, the overall variance of the integral calculations are small and all centered around 1.