Investigating Diffusion Using Random Walks in C

Candidate Number: 24852
Department of Physics, University of Bath, UK, BA2 7AY

(Dated: April 14, 2022)

DIFFUSION IN ONE DIMENSION

A 1-dimensional diffusion system was modelled by a random walk algorithm which was reliant upon the generation of pseudo-random numbers. With a number of either 0 or 1, a particle can be made to "walk" backwards (if the random number is 0) or forwards (if the generated number is 1). These values, generated by the rand() function from the stdlib header, are labelled "pseudo-random" because they are created through a simple linear congruential generator:

$$I_{i+1} = (aI_i + b) \bmod m \ , \tag{1}$$

where I_0 is the seed value, a is a multiplier, b is an increment, and mod m is a modulus - typically the largest obtainable number to be generated. As this sequence of random numbers is totally dependent on the starting seed value, a sequence can be repeated for the same seed - making it in fact not random, hence the name "pseudo-random". This is made clear in the program as the same seed value will always result in the particle diffusing in the same pattern.

Through varying the seed value, very different results can be produced for $M = 20$ hops. For example, when contrasting seed values of 5 and 6 on a line of 10 points, both produce similar results early on, but for the latter half, a seed of 5 shows oscillation between positions 1 and 3 while oscillation between positions 8 and 9 are very prevalent for the seed equal to 6.

Diffusion was simulated using a pointer as the particle. This pointer was then pointed to the first element in the grid (or first position) and be incremented or decremented to move it along the grid. Periodic boundary conditions could also be implemented by pointing the particle to the final grid element if it was moving backwards from the first element, and the opposite for the particle moving forwards from the final element.

Using a structure consisting of a grid position and a Boolean enumeration (where false = 0 and true = 1) for whether the position has been visited before, the program was altered to enable the particle to diffuse until it had visited every grid point. Whenever a hop was made, the visited variable of the grid element which the particle was pointing to was checked. Using memory arithmetic of (particle - grid), the index the particle was pointing to could be found and changed to a visited value of true if it was previously false. This memory arithmetic worked as it essentially subtracts the address of grid[0] from particle, leaving the number of indexes between the pointer's address and the first grid point's address. Hence, the grid itself could be updated if a new location was visited. After this, the arrayAllTrue() function caused the loop to execute until all points on the grid had a true visited value. Carrying this out for a given random seed, the same number of hops were made for a constant number of sites. Thus it was inferred that through recording the random seed, previous results could be reproduced. As a result of these sequences being periodic, using a single random seed could produce unreliable results.

In order to obtain more meaningful results for counting how many hops were necessary to traverse 10 sites on the grid, an input was incorporated to permit the user to choose the range of seed values they would like to use for an average to be calculated. For example, if the user were to input 1 as a starting seed and 10 as a final seed, the program would calculate the number of hops to traverse each site for seed values 1-10 and take an average of these.

Error handling was implemented using the return value of the scanf function so that a Boolean of 1 is returned if an integer is entered and a Boolean of 0 otherwise. If invalid input is entered, the program will clear the input stream and allow the user to continuously input until something valid is entered. It should also be noted that the upper seed limit is forced to be greater than the lower seed limit (as this would raise errors in looping through seed values otherwise), and that any floats are rounded down by default as scanf takes the input prior to the floating point.

The aforementioned steps could then be looped for an increasing number of sites from 2 up to 200 with random seeds 1 to 100. $L = 2$ was used for the lower bound with an expected number of loops of 1 (as the particle would always move to the other point due to the periodic boundaries), while the whole grid would already be visited for $L = 1$ so there would not be any diffusion occurring. For the upper bound, $L = 200$ was used, but this was an arbitrary value to get a wide range of results and could theoretically be raised at the cost of a large amount of computational time.

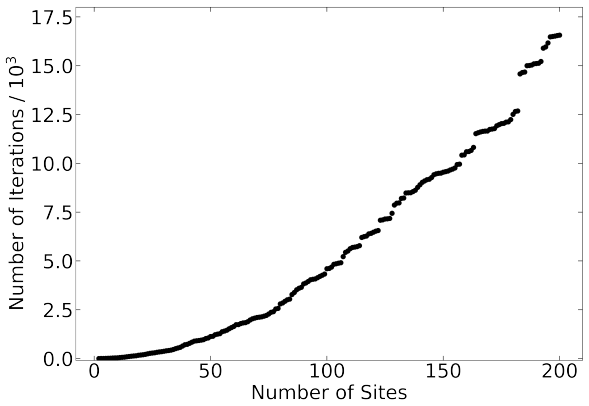


FIG. 1. How the number of iterations necessary to traverse every grid point in a 1-D system varies as the number of sites on the grid increases.

The results for this simulation are visualised in Figure 1. As expected, an increasing number of sites leads to an increasing number of iterations being required to traverse the entire grid. This is because with more points that visiting, more iterations will be necessary to traverse each point, especially with the element of randomness introduced to the system. Physically, this is expected as well, since it takes longer for an air particle to diffuse through a small room in comparison to a larger hallway. Interestingly, there is a leap in number of iterations around $L = 190$, which is likely a mere consequence of a greater jump in iteration value as number of sites surpasses a certain value.

DIFFUSION IN THREE DIMENSIONS

Due to the generalised nature of the C code for a 1-dimensional system, it could easily be adapted to model diffusion for a 3-dimensional system in which the particle can travel in 6 directions. Additional dimensions were added to the Point structure so it could store x, y and z co-ordinates, and the grid was created as a 1-D array of size L^3. When creating the grid, z-direction was first incremented, then y-direction, and finally x-direction. Hence the second point for $L = 3$ would be at (1, 1, 2), fourth point at (1, 2, 1) and tenth point at (2, 1, 1).

The particle was pointed to the grid point (1, 1, 1) and permitted to move forwards or backwards in the x, y or z-direction. To do this for z, the pointer address could simply be incremented or decremented as the z-coordinate was the lowest level of the grid. For x and y, an additional factor of L was needed in the multiplication for incrementing the pointer e.g. the address would move forwards by L when moving forwards in the y-direction and it would need to move forwards by L^2 for the x-direction. Boundary conditions were implemented in a similar manner, multiplying the increment/decrement factor by L and L^2 for y and x, respectively. Otherwise, everything was set up the same as for the 1-D diffusion system.

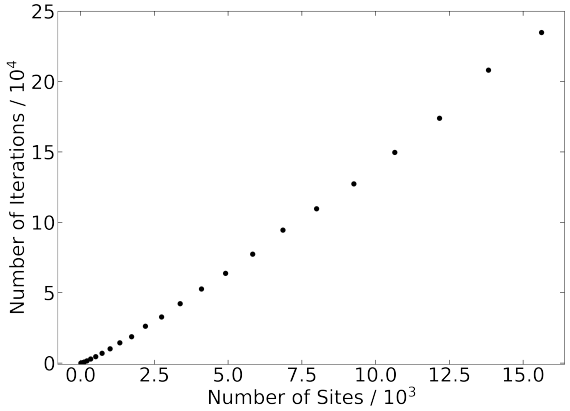


FIG. 2. How the number of iterations necessary to traverse every grid point in a 3-D system varies as the number of sites on the grid increases.

Results for the number of iterations needed to traverse every point in the 3-D system are displayed in Figure 2. These were recorded using an average of random seeds between 1 and 100 for each $L = 2$ up to $L = 25$ (or $N = 8$ to $N = 15625$). Beyond $L = 25$, the program would begin taking a very long time as many iterations would be necessary to traverse each point since computational time scales a lot more for L on the 3-D grid. This is clear from Figure 2, as the average number of hops required to traverse the whole grid increases with the number of sites N.

Each plotted point is further away from the previous (in the x-direction) because it is cubic along this axis, again showing that number of hops needed will be greater since number of sites is not increasing linearly. As explained for the 1-D case, this makes sense physically since a particle will require more time to diffuse across a larger room for both the one and three-dimensional scenarios.

DIFFUSION IN A NETWORK

To model the system as a network, a Node structure containing information regarding connecting points to any vertex and a Graph structure containing an array of adjacency lists were implemented. The graph was created using some networks derivations (see Appendix A) made specifically for the given system where sites within a cluster are all connected and where there is a single connection between each cluster. As the graph was undirected, when a new edge was added, a corresponding edge going in the opposing direction was added as well. This made creating networks of varying sizes much more efficient. Additionally, periodic boundary conditions were

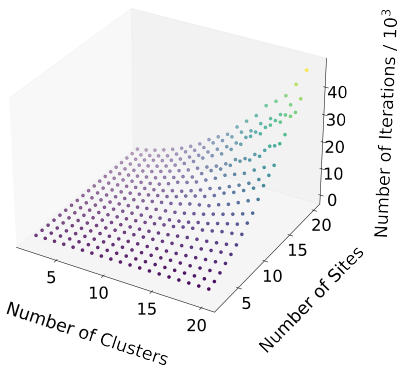


FIG. 3. How the number of iterations necessary to traverse every grid point in a network varies as the number of clusters and sites vary.

automatically considered by the edge between the first node and the final node that was included first in the graph's creation.

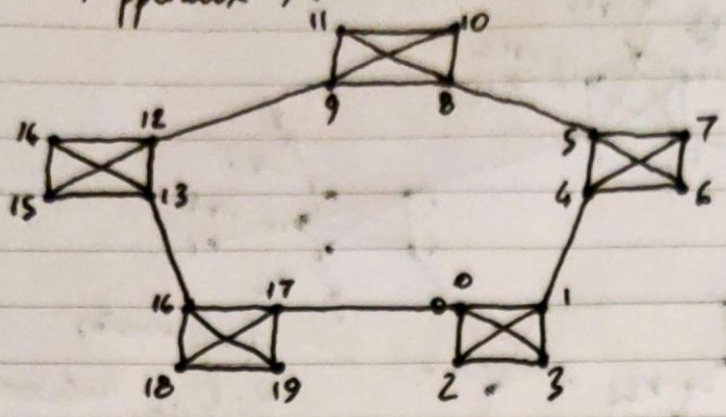
The getNodeAdjacencyList() function existed to return the number of edges that exist for a node along with where these edges lead. This was of great use for producing a random number which would indicate where the particle would diffuse to next. Furthermore, a utility function was made to deallocate memory usage for the graph when a new one was to be created or no more were to be used. Finally, the isAllGraphVisited() function worked in a similar manner to arrayAllTrue() in earlier parts, with a Boolean being returned that depended on whether each node in the network had been traversed.

The particle pointer could be moved around the system quite easily through changing it to point to the graph's adjacency list indexed by the randomly generated number and this same node was then set to have a true visited value. The number of clusters (A) and sites (B) were each increased from 2 to 20 for random seeds from 1 to 100, with an average being taken for each site and cluster number. These results are visualised in Figure 3.

From this plot, the number of iterations will increase with both an increasing number of clusters and sites, but the most significant change will occur when increasing both at the same time. This is to be expected because when both double, the total number of vertices increase by a factor of four, making it take longer to traverse each node. Moreover, it takes longer to traverse the network than it does to traverse an entire three-dimensional system - comparing $L = 7$ for three dimensions and $A = 17$, $B = 20$ for the network, as N is roughly 340 for both - with the network taking approximately 10 times as many iterations to visit each point. This is likely a consequence of adjacent points being very accessible in the three-dimensional system (can move in 6 directions), whereas to reach one node in the network the entire graph may require traversing again. It also takes slightly longer to traverse a one-dimensional system than the network, which is due to each point in the line only being able to travel to 2 other points.

For future simulations, diffusion could be modelled with a random number generator which is more complex than a simple linear congruential generator e.g. a generator that is cryptographically safe such as ISAAC (indirection, shift, accumulate, add, and count). These results could then be contrasted with those obtained using a linear congruential algorithm to see which is the more viable random number generator.

Appendix A



A = 5 clusters
B = 4 sites
V = 5×4 = A×B = 20 nodes

Each 'inner' vertex can make B (4) hops
Each 'outer' vertex can make B-1 (3) hops

Always 2 'inner' nodes per cluster → B-2 'outer' nodes
0 & 1 always 'inner' → vertices B and B+1 will be 'inner'

(Vertex no. % B = 0) → Gives 'inner' node connecting to previous cluster

((Vertex no. - 1) % B = 0) → Gives 'inner' node connecting to next cluster

Any other node is 'outer'

If vertex = 0, can add connection to vertex no. =
(clusters × sites) - (sites - 1) or (A×B) - (B-1)
e.g. A=5, B=4: vertex 1 connects to (20) - (3) = 17

Node connecting to previous will have B connections to:
(v+1), (v+2), ..., (v+B-1) and (v-B) → This can be added automatically for undirected

Node connecting to next will have B connections to:
(v+1), (v+2), ..., (v+B-1) → This is edge connecting to next cluster

'Outer' nodes will connect to (B-1) nodes:
(v+1), ..., (v+B-2) → stop adding when number added to v % B = 0

No need to consider (v-1) edges if undirected graph