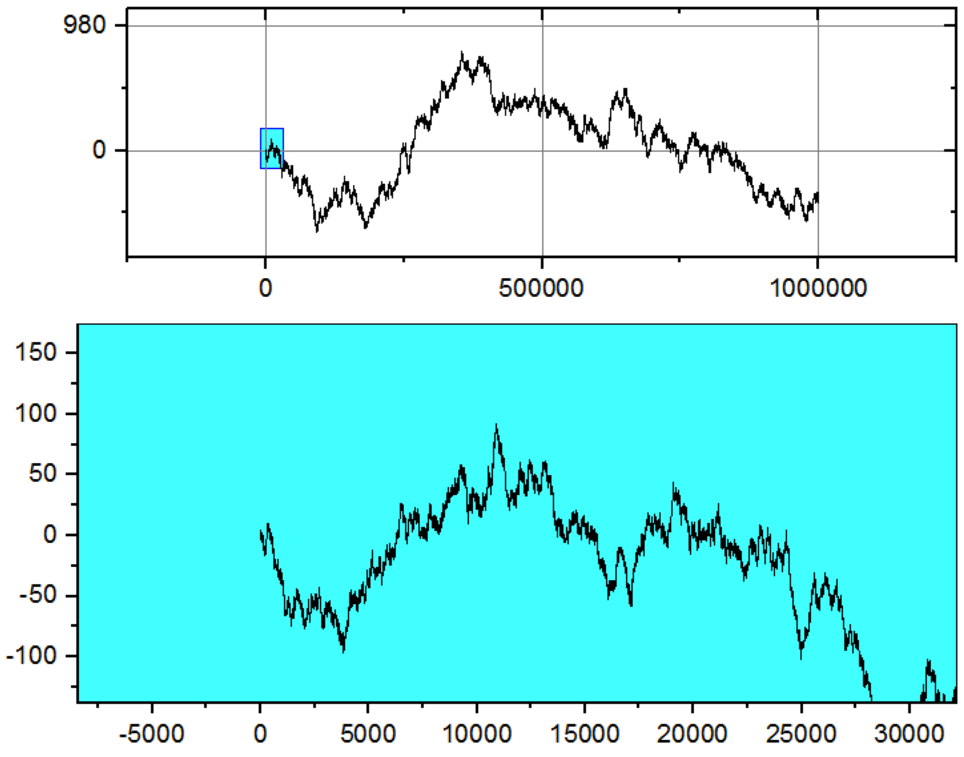
Project 6 – Exploring Basic Stochastics

Stochastic systems or processes are those whose dynamics are seemingly arbitrary, within reasonable limits. They are usually therefore very disorderly, yet can still be analyzed statistically. When the variables of a natural system are not well determined, or are not well correlated, they tend to be accessible only through stochastic modeling. Here we model basic stochastic processes, such as random walks (RWs), self-avoiding walks (SAWs), diffusion, and space-filling curves (SFCs). We also hint at the correlations between stochastic processes and self-similar or "fractal" processes, as well as note how these can be analyzed in terms of their probability distributions, and entropy – but most of the calculations are saved for the next project. Our aim here is principally to explore. We examine the basics, from 1 to 2 to 3 dimensions. Most of our modeling is done via Java, because Java is known to have a very good "Secure Random Number" generator function (RNG), which serves as our essential stochastic element.

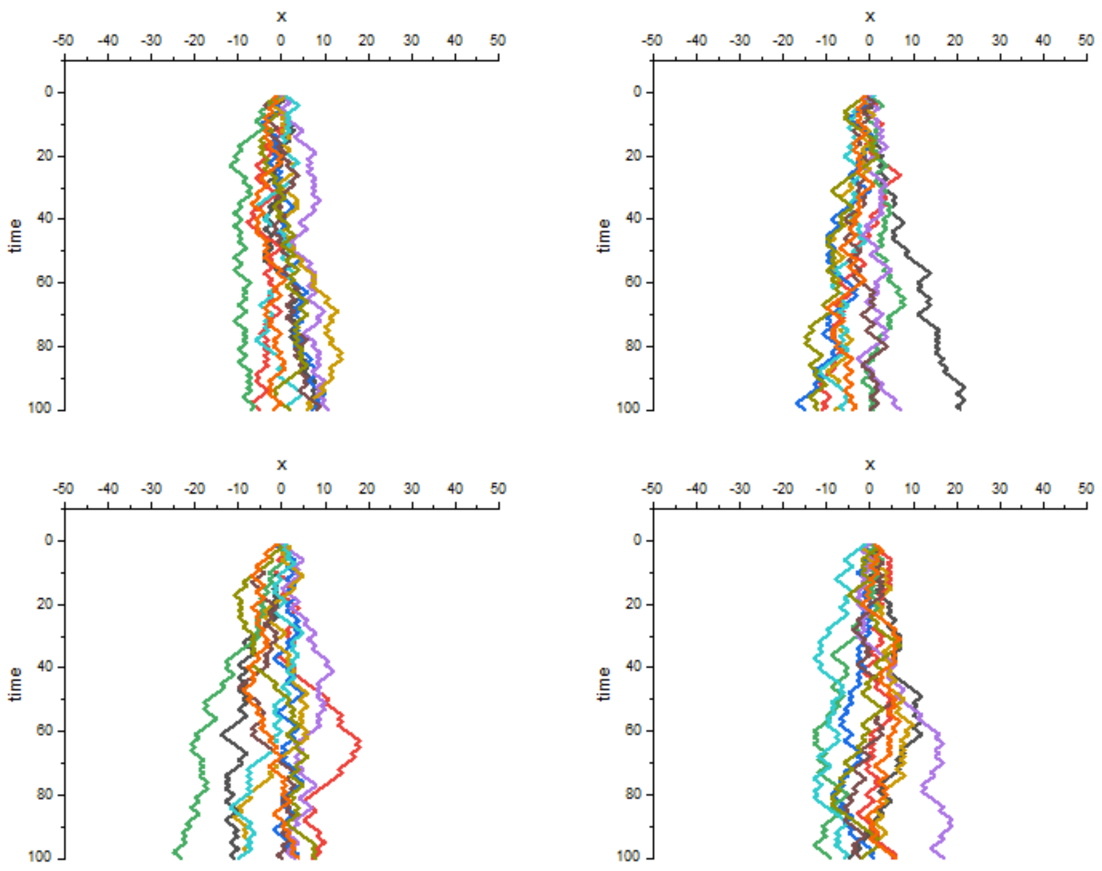
We begin our exploration with a random walk in 1D. This is akin to being able to move only forwards or backwards (like in a narrow hallway), and at each step you flip a coin, roll a die, or make an arbitrary decision as to which way you'll step next. This motion is then plotted over time. Each time-step (sometimes labeled TS) is equivalent to one iteration of our basic program, when a single RNG is chosen.

Below is our first, and perhaps most interesting plot, showing distance traveled forward and backward on the y-axis, and time or number of iterations on the x-axis.

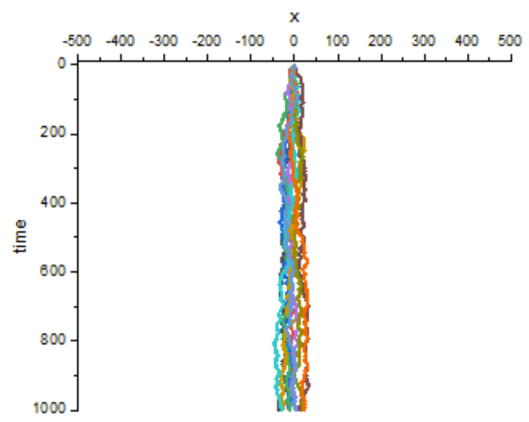


The shaded plot is simply a zoomed-in version of the unshaded plot. One can see that even after 10^6 steps, no obvious pattern has formed. What is also striking to notice is how similar the zoomed plot is to the unzoomed plot. Admittedly, I've chosen a specific area of the overall path for this self-similarity to shine, yet the correlation between fractals and stochastic processes is not unjustified. I leave it to the reader to contemplate this for now, as we will explore this connection more in the next project or two.

Moving on then, we ask how the picture will change when we have multiple *walkers*, who tread the same thin hallway together. For now, these walkers can move through one another, but later we'll see how different things become when RWs become SAWs.



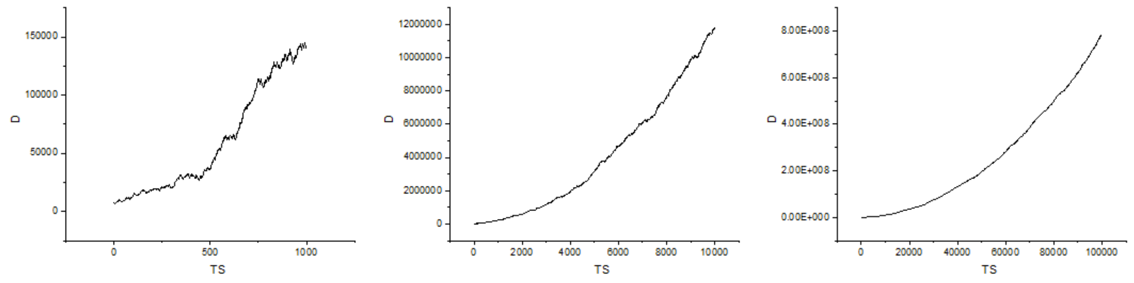
With no intention to confuse you, the axes have been flipped for visual preference – so now we see right and left as the x-axis and time on the y-axis, going downwards. These four plots are of 10 walkers in 100 steps. Each one is quite different, yet they all appear to follow a slight trend: they all expand out from zero and spread slightly as they go. We should be weary though, since stochastic systems are definably those with the least amount of pattern. Nevertheless, we are explorers, so we up the ante and run our calculation for 10^3 time-steps.



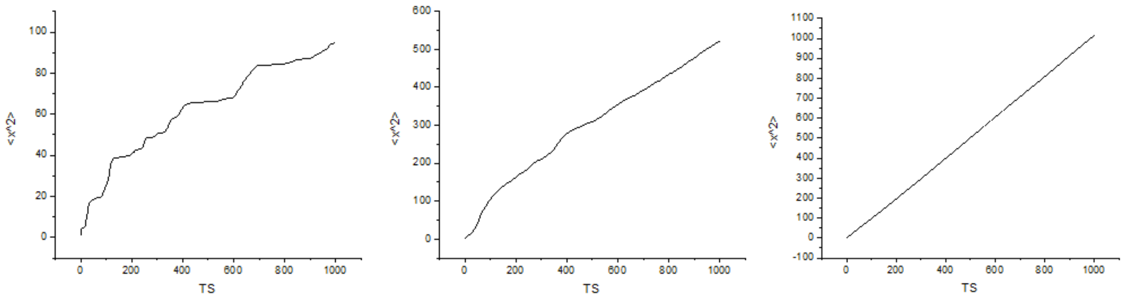
To the left, we see immediately the pattern fizzles out; the spreading doesn't last and eventually the random motion appears to linearize. This can also be understood in terms of the first plot, when we notice that even after 10^6 time-steps, the graph is about as positive as it is negative, and therefore appears to oscillate around the origin.

Such a phenomena is also similar to the analogy of flipping a perfectly random coin, or equivalently, flipping a reasonably random coin an infinite number of times: either way, the chance of heads or tails should be the same: 50/50.

However, if we sum the displacement $\sum|x(t)|$ *over each iteration*, we do get an exponential divergence – as we might expect.

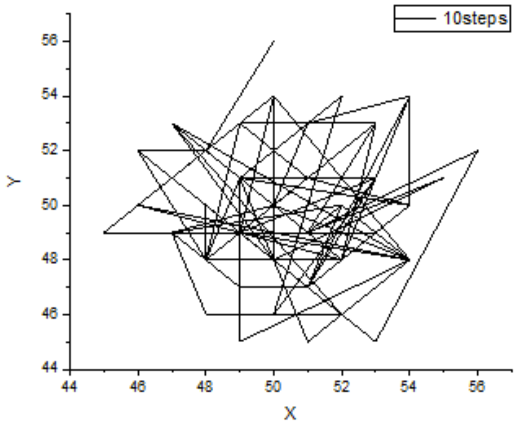


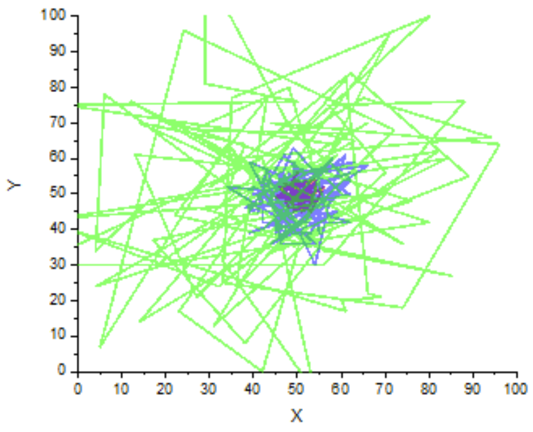
Above three plots shows the accrued displacement $D(t)$, for one walker given 10^3, 10^4, and 10^5 iterations. We see the plot quickly becomes exponential. And to this point, we are motivated to plot an expectation value $\langle x^2\rangle$ as a function of time, as we increase the number of walkers, in order to recover linearity. Below we do just that, showing how accrued distance x actually varies linearly with $\sqrt{t}$.



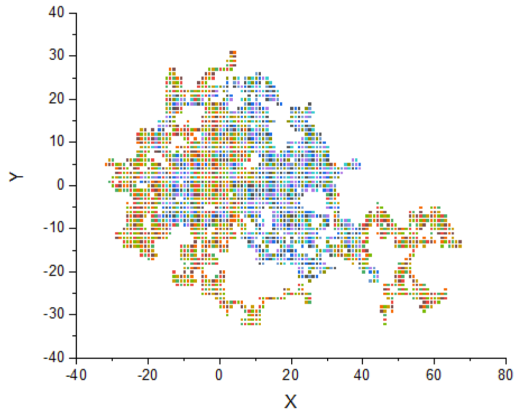
The three plots show increasing number of walkers, $10^{1,2,\&3}$, with their distances x_n averaged, and summed over 10^3 time-steps. Essentially this demonstrates that the dynamics of walkers tend to cancel each other out. Additionally, this demonstration provides us with what to expect, given n number of random walkers. The fact that D is exponential in t, and $\langle x^2 \rangle$ is linear in t, is simply based on how we've accrued our distances – this is important for studying *growth* as we'll see in projects to come.

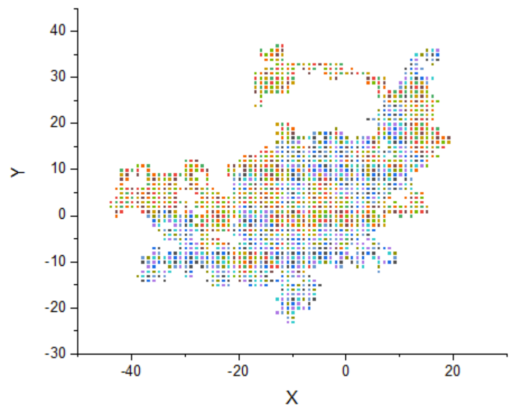
Next in our exploration is none other than the 2nd dimension!





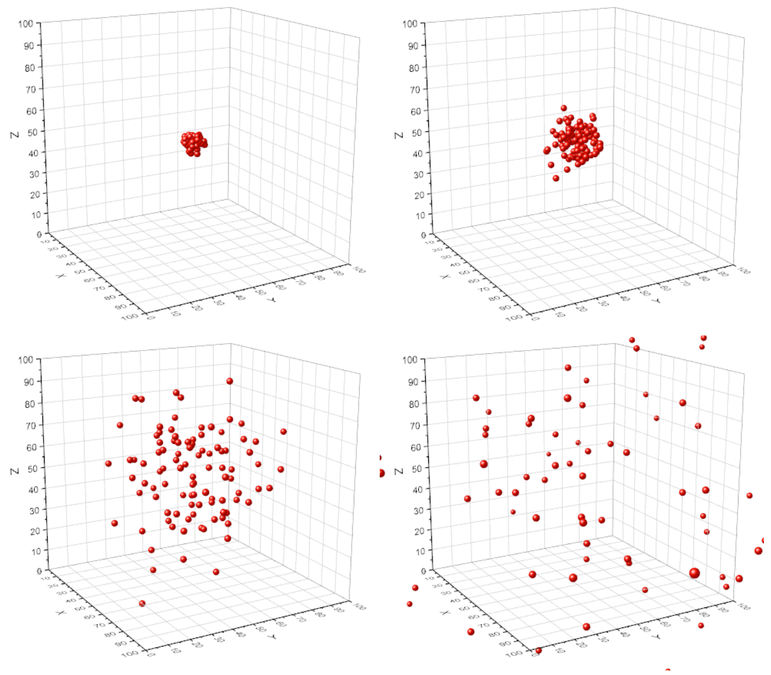
Here we have some rather hideous figures. On the left, one walker randomly scribbles about in 2D. On the right, we've superimposed 3 sets of growing iterations. They aren't pleasant to look at, and there isn't much to learn either, other than perhaps an insight for what's soon to come: More time means a further spreading out, which is itself a kind of *diffusion* – albeit a messy one.





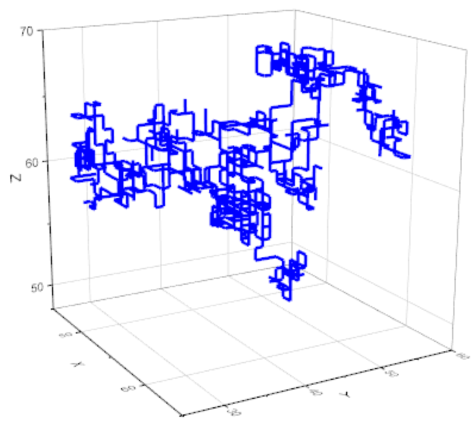
Above are my favorite two plots, they show what happens when instead of scribbling about, you allow the walkers to move only up-down left-right, on the condition that they cannot retrace the same space. Right away we see a massive difference in outcome – not only are these plots much more pleasant to look at, but they also appear to exhibit much more structure. Unfortunately, I'll have to leave these kinds of plots for a future project, since they relate directly to crystal growth and similar structures which *build off of themselves*.

As nice as 2D is, we must move on to 3D!



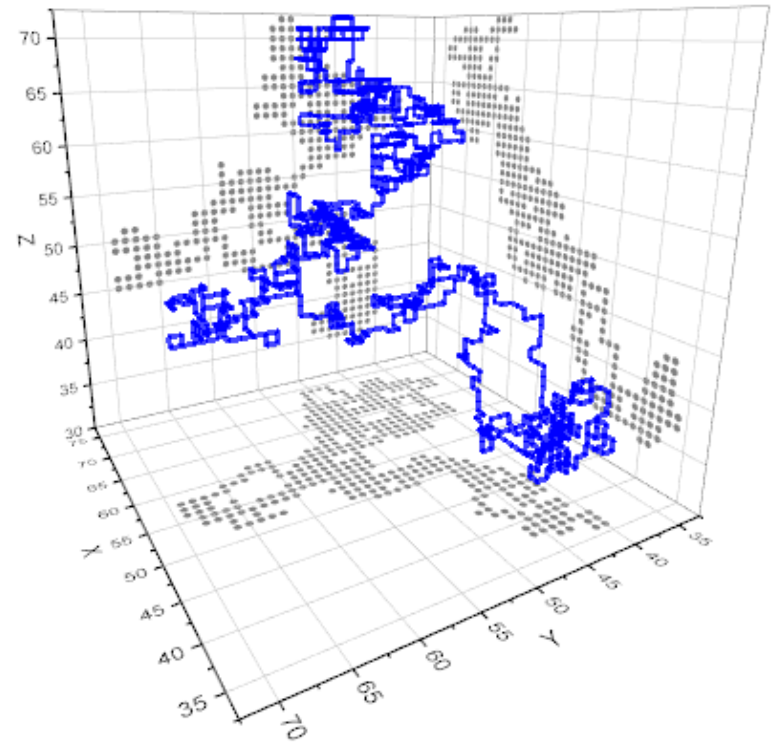
Here we see something just like the hairballs above, but with points instead of lines. This looks just like if you opened a packet of skittles in the ISS – aka, just like diffusion. Whereas previously (minus the last two tile-like images) we've been tracking the trajectory of walkers by incrementing their positional values, here instead we have a grid of 0s (representing no walker) and 1s (for a walker). The code is essentially the same, but instead of tracking an array of walkers and their coordinates, we simply track the grid. Diffusion however will be emphasized more in the next project.

Back to lines, we compare RWs with SAWs – that is, what happens when lines aren't allowed to trace back over themselves? This can of course be done in 2D, but I find it more fun in 3.

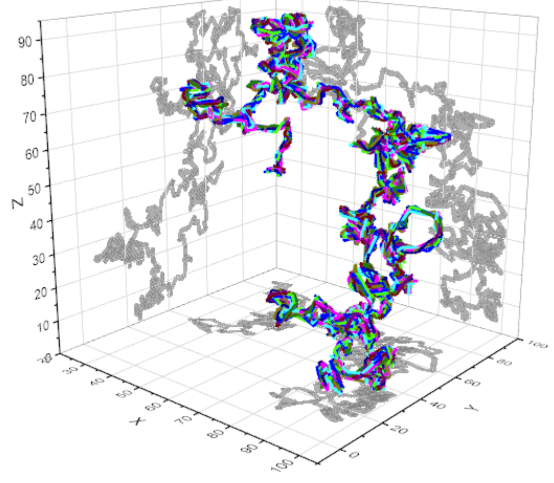


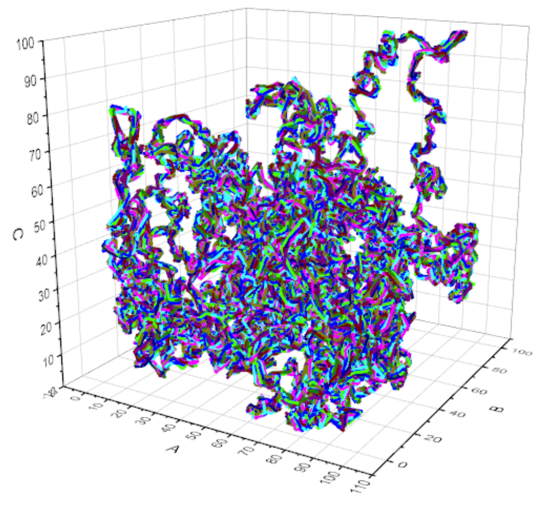
Here we see what looks like a crazy 3D New York subway map. The lines are able to go one way then back that same way. The next few images are large, so I'm trying to give them some space. They will represent SAWs – not in the sense that they cannot cross themselves, but they cannot back-trace, i.e., if they do cross, it's because they've gone off one way and come back again. Notice the spikes and multiple terminal points. For now we represent a single walker, but soon we'll see more.

Below we see the walker avoiding itself, and therefore stretching out more in 3D, creating what looks a bit like ant tunnels. On the walls are projected shadows, to help visualize the structure. Clearly it's different from all sides.



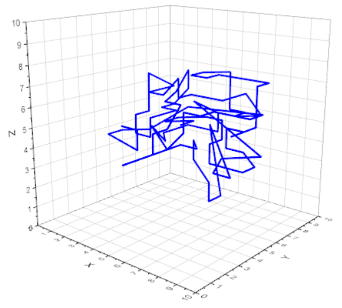
Below is what happens when you add multiple walkers (10 each). The left is run around 200 iterations, whereas the right is run 500 – 700 iterations. With these particular systems, one cannot run above a certain number of iterations ~900, simply because the walkers can trap themselves since they cannot back-trace.

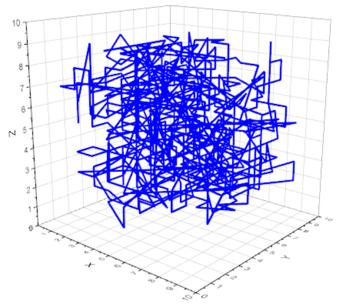


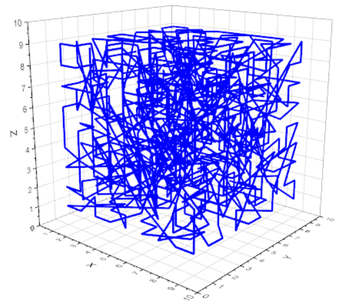


The fact that the walkers appear to be following one another is merely a consequence of how I structured the program – essentially each walker piggybacks the same RNG each iteration. And whereas this might not be ideal (perhaps you may want them to fight over the same space), what I love about this circumstance is that it leads naturally to these bundles and coils – simply because they follow yet cannot cross. Moreover, since they get in their own way, they end up following somewhat linear trajectories and 'bouncing' off walls – all with confined randomness. Anyone who's seen protein structures and their dynamics should be fascinated by the striking similarity.

As fun as this is, we must move on to the final phase. By looking at the above plots one might be inclined to see what happens when one attempts to fill the whole space.







What we've made are stochastic *space-filling curves*, because if you iterate this to its limit, you will eventually hit every point within the 3D cube. Space-filling curves have lots of use in computer science and physics. That's because they provide a dimensional reduction, writing the information in nD space into $(n-1)D$ space, which is great for data compression, optimization, and general simplifications. Usually SFCs are not stochastic, but I don't doubt their utility. Indeed, between well-ordered space-filling curves and stochastic ones are fractal SFCs, which tend to make great models for natural and organic processes – like a slime mold attempting to fill the surface of a petri dish.

To conclude, we've explored stochastic processes / systems in 1, 2, and 3 dimensions. Specifically we looked at random walks, self-avoiding walks, diffusion, and space-filling curves. We also explored a bit about the statistics analysis behind simple systems, yet much more work needs to be done for anything like a 'thorough' account to be made. Nevertheless, the exploration has proven fun and fruitful.

The code I used was primarily Java (again, for the RNG), yet it's been edited and re-edited several times in order to construct the different plots. For that reason, I'll just paste below the code I used for 3D grid-oriented multi-walker SAWs / SFCs – which can easily be adjusted for 2D, or more complicated situations. Please excuse any verbosity – the code isn't as compact as Fortran or C.

```
//Johnathan von der Heyde  2019
import java.security.SecureRandom;
import java.util.Scanner;
import java.io.FileWriter;
import java.io.IOException;

public class xyzgrid {

   static int walks = 10;
   static int max = 100;
   static int grid[][][] = new int[max][max][max];
   static int x = max/2;
   static int y = max/2;
   static int z = max/2;

   public static void main(String[] args){

      int i, steps, count = 1, rand = 0;

      double rSquared = 0.0, MeanrSquared = 0.0;

      setgrid();

      System.out.println("Please Enter # TimeSteps");
      steps = getInt();

      try { FileWriter out = new FileWriter("grid.dat");

         while (count <= steps){

            for (i=0; i<walks; i++){

               while (grid[x][y][z] == 1){

                  rand = returnRandomInt();

                  switch (rand){
                     case 1: x++; break; //right
                     case 2: y++; break; //forward
                     case 3: z++; break; //up
                     case 4: x--; break; //left
                     case 5: y--; break; //backward
                     case 6: z--; break; //down
                  }
                     if (x==max) x--; if (x==0) x++;
                     if (y==max) y--; if (y==0) y++;
                     if (z==max) z--; if (z==0) z++;
                  }

                  grid[x][y][z] = 1;
                  out.write(x + "," + y + "," + z + ",");
               }

               out.write("\n");
               count++;
            }

            out.close();
         }

         catch (IOException e) {e.printStackTrace();}
                  }

   static void setgrid(){
      for (int x=0; x<max; x++){
         for (int y=0; y<max; y++){
            for (int z=0; z<max; z++){
               grid[x][y][z] = 0;
            }
         }
      }
   }

   static int getInt() {
               Scanner scr = new Scanner(System.in);
               while (!scr.hasNextInt())
                  scr.next();
               return scr.nextInt();
   }

   static int returnRandomInt(){
      SecureRandom rand = new SecureRandom();
      return rand.nextInt(6)+1;
   }
}
```