PH30056 Comp Phys B – DLA Coursework Assignment

General guidance on project

This document includes some background information on diffusion limited aggregation (DLA), and the instructions for this coursework assignment. The idea is to use a C++ program to investigate the physical features of a model of DLA. To submit, you will upload your report and a working program to Moodle. I understand that one program will not be sufficient to show your answers to all parts of this coursework: you should hand in a version that illustrates what you did. The report should be written at a level suitable for a third-year physics student who is not taking this unit. On the report, you should identify yourself by your candidate number: *you should not include your name*.

The deadline for the report and the uploads to Moodle is **4pm on Wednesday 23 March**. A report **template** and details of report **marking criteria** are available on Moodle, in separate documents. For the first coursework, the maximum length for the report is 5 pages.

The report should have a clear structure, with a title and abstract, as well as sections dealing with the various exercises within the coursework. You need to explain your methods as well as showing your results. Please do *not* include your entire code. If necessary, you may want to include extracts from the code in your report (it may be easiest to put these into an appendix, which does not count towards the page maximum). Include any plots, tables etc. . . that you consider useful in explaining your findings in the main text. In **week 5**, there will be a lecture about how to write a good report, focusing on structure, content, and flow. I strongly urge you to attend all of the lectures.

Feel free to discuss the coursework and your progress with other students. In all cases, we ask that you submit an **individual report** and you explicitly acknowledge any help that you have received. Plagiarising any part of anyone else's report is forbidden. It is in your own interest to ensure that no one has the opportunity to copy your work.

Part of developing good project management is to write *tidy and readable code*. This will help you as you go, and it will also help anyone else who tries to read the code (including lab demonstrators, and whoever marks your report). Another important project management idea is *version control*. This involves making backup copies of your files as you move forward with the exercises. That way you can always return to the older versions of the codes if you need to.

Version history:
AS, 7 Feb 2022

Section 1 – DLA Introduction

In diffusion limited aggregation, a cluster of particles grows around an initial "seed particle". The simplest method for simulating this process on a computer is:

1. Start with a single stationary particle in the centre of a large grid.
2. Introduce a particle at a random point in the grid, far from the centre. This particle moves around the grid by choosing one of its neighbouring sites at random, and hopping to that site. The particle keeps hopping in this way until it arrives at a site adjacent to a stationary particle. Then, the moving particle becomes stationary and never moves again.
3. Repeat step 2 many times.

This method works but it can take a very long time to build a large cluster, because typical particles have to make many hops before becoming joining the cluster. For this reason, you are provided with a code which achieves almost the same result, much more quickly. The method involves three circles, all centred on the initial seed particle. As shown in Fig. 1, these are called the "cluster circle", the "starting circle", and the "killing circle". The radii of these circles are r_{kill}, r_{start} and r_{max}, respectively. (Beware: within the code, they have slightly different names.)

The radius r_{max} is the distance from the starting particle to the stationary particle that is furthest away from it. This radius will be used as an estimate for the size of the cluster. The radius of the starting circle (r_{start}) is larger than r_{max}: when new particles are introduced into the system, they appear on this starting circle. (If particles are introduced further away than this, it is likely that the program will spend a lot of time simulating their motion, but this motion happens far from the cluster and has no effect on its properties. So including the starting circle makes the program more efficient.)

Also, the radius r_{kill} is larger than r_{start}. If a particle reaches the killing circle, it is removed,

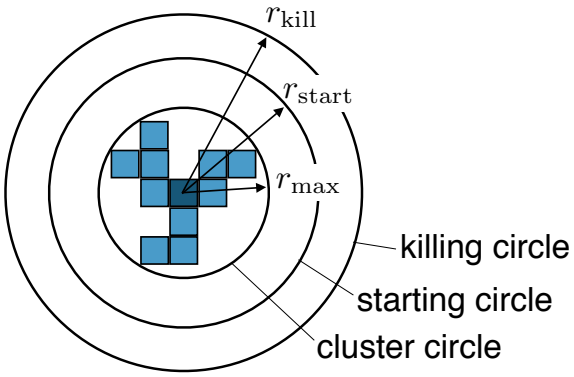

FIG. 1. Diagram of the circles used in the DLA code. The initial particle is shown in dark blue, and all circles are centred on this particle. There are three circles which indicate the approximate cluster size ("cluster circle"), the locations at which particles are introduced to the system ("starting circle"), and the points at which particles are removed from the system, to prevent them from wandering too far from the cluster ("killing circle").

and a new particle is created to replace it. (The new particle appears on the starting circle, as usual.) The killing circle is included because if particles are allowed to diffuse very far from the cluster then the program will a long time simulating their motion and waiting for them to return. The killing circle prevents this from happening.

It is not difficult to show that including the starting circle does not change the properties of the clusters that grow. On the other hand, including the killing circle can lead to changes in cluster properties, but these effects can be controlled by making sure that $r_{\rm kill}$ is not too close to $r_{\rm max}$.

Fractal dimensions

DLA clusters are *fractal* objects. Loosely speaking, this means that they have hierarchical structures where similar patterns repeat, on different length scales. So a DLA cluster with $10,000$ particles looks rather similar to a zoomed-in view of a cluster of $1,000$ particles.

One way to characterise fractal objects is to ask how many particles are needed to build an object whose size is roughly R. If one has square particles of size a, one needs R/a particles to make a line of length R. If I want to make a square cluster of size R, I need $(R/a)^2$ particles. If I want to build a cubic cluster out of cubic particles, I need $(R/a)^3$ of them.

For fractals, the number of particles (N_c) required to build a cluster of size R typically scales as $(R/a)^{d_{\rm f}}$, where $d_{\rm f}$ is called the fractal dimension. From the simple scaling relation $N_c = (R/a)^{d_{\rm f}}$, we can estimate the fractal dimension as

$$d_{\rm f} = \frac{\ln(N_c)}{\ln(R/a)}. \tag{1}$$

From the examples above, a line has $d_{\rm f} = 1$ (one-dimensional object), a square has $d_{\rm f} = 2$ (two-dimensional object), etc.

In the DLA example, we use the length $r_{\rm max}$ in Fig. 1 to estimate the size of our cluster. Since Eq. (1) may not always be accurate, we assume a more general relation

$$N_c(r_{\rm max}) = (\alpha r_{\rm max})^{d_{\rm f}} + \beta \tag{2}$$

where α, β are unknown constants. If we take logs of both sides of Eq. (2) and then differentiate with respect to $\ln r_{\rm max}$ we get

$$\frac{{\rm d}(\ln N_c)}{{\rm d}(\ln r_{\rm max})} = \frac{d_{\rm f}}{1 + \beta/(\alpha r_{\rm max})^{d_{\rm f}}} \tag{3}$$

As $r_{\rm max}$ gets large, you can see that $\frac{{\rm d}(\ln N_c)}{{\rm d}(\ln r_{\rm max})}$ approaches $d_{\rm f}$. Hence, it is possible to estimate $d_{\rm f}$ by plotting $\ln N_c$ against $\ln R$, and measuring the gradient. If $\beta = 0$ then the graph should be a straight line with gradient $d_{\rm f}$. (This should be true whatever is the value of α.)

Section 2 – DLA Program

These are the instructions for the DLA project. As always, remember that you need to manage your time during the semester. If you can't finish both exercises, just complete the parts that you can do.

Make sure that you leave time to write up your report since however good your code might be, you will only get marks if you write up your results clearly, explain how you got them, and what they mean.

Getting started

The first step is to set up a new C++ project in visual studio. The DLA program will be similar in structure to the example code (with one particle) from the lab session in week 2. To set up your new project:

- Create *A New C++ Empty Project*. You can create it on the H: drive, for example. You may also try to create the project on a USB memory (however, note that the USB may work rather slowly...).
- Go to the *Project* menu, click *Add New Item* and choose *C++ File (.cpp)*. Copy-paste the code from *mainDLA.cpp* that is available on Moodle.
- Install the nupengl library as described in previous handouts.
- Create two new classes called `Window` and `DLASystem` (*Project* menu and click *Add Class*).
- Create two new inline classes called `rnd` and `Particle` (to create an inline class you just need to check the 'inline' box in the *Add New Class* form).
- The code that should appear in the relevant C++ files is provided on moodle.

If you copy in all the code and compile and run, you should see a graphics window with a single green particle, and some instructions in the console window. Hit 'u' a few times and you should see the simulation update itself, one step at a time. Hit 'g' and the DLA simulation will start. Hit 'f' to make the simulation run faster.

As the cluster grows, the window will zoom out to follow it. The window is always bigger than the killing circle. If you hit 'z' then it will zoom in so that the window is roughly as big as the starting circle. If you want to include pictures of clusters in your report, it may be useful to hit 'w' so that the black background to the window is replaced with a white one.

Program overview

As in the program from the second lab class, the main program file is fairly simple. It creates a `DLASystem` and it uses a global pointer to access that system. There is a `handleKeyPress` function that controls what happens when you press a key. There is also an `update` function which tells the `DLASystem` to update itself and then tells openGL to wait a while before calling the `update` function again.

You will probably not need to edit the `rnd`, `Window`, and `Particle` classes. The main business happens in the `DLASystem` class:

- The most important *member variables* are called `grid` and `particleList`.

- `particleList` is a C++ vector, which stores the particles and their positions: the idea is that `particleList[i]` is a pointer to the ith particle. The number of particles in the list is stored in a variable `numParticles`. There is a variable `endNum`: if `numParticles` reaches `endNum` the simulation will stop. Particle positions are stored relative to the initial particle which is at the origin.

- The `grid` works like a 2d array, which is used to keep track of where particles are in the system. This array is accessed via `readGrid` and `setGrid` functions. If `pos` is a vector that stores a position then `readGrid(pos)` returns the value 1 if there is a particle at that position and 0 otherwise. Similarly `setGrid(pos,val)` stores the value `val` at position `pos` in the grid (with 1 representing the presence of a particle, 0 otherwise). Note that position vectors may have negative components but indexes for the `grid` array must be positive or 0 (for this reason you should access the grid only through the `setGrid` and `readGrid` functions).

- Within the algorithm, only the last particle ever moves. There is a variable `lastParticle` that takes the value 1 if this particle is free to move, and 0 if the particle is stuck to the cluster.

- There is a single constructor function that takes a `Window` pointer as input. This function mostly just initialises the variables in the class, including `endNum`. (The constructor also sets the random seed and sets up the `grid` array and the `particleList`.) The constructor calls a function called `Reset` which also sets up some variables such as the size of the starting circle.

- The most important functions are `Update` which causes the system to update itself (either move a particle or add a new one); and `DrawSquares`, which draws the system on the screen. If you look inside `Update` you will find that it uses calls to several other functions including `moveParticle` and `addParticleOnAddCircle`. *Note: it is important that these two functions are kept separate: the drawing function should not change any system variables and the update function should not do any drawing.*

- Note the variable `seed` in the main source code. This value is the seed for the random number generator. As long as you do not change the value, the generator will keep producing exactly the same sequence of random numbers (and consequently DLA clusters with exactly the same shape). So you need to change the value of the `seed` in order to see examples of different DLA clusters.

- There are several other member variables which we do not discuss in detail: the `rgen` object is of type `rnd` and deals with random numbers; there are variables like `running` that determine whether the system is running or paused, and `slowNotFast` which determines whether it is running slowly or quickly. There are also various functions that allow these variables to be changed (eg `pauseRunning` and `setSlow`).
  As a general point, it is a good idea to use functions to modify member variables; this tends to make the program more flexible. For example, the `setRunning` function makes the simulation start running, but it first checks that the cluster has not yet reached the edge of the grid.

- The function `updateClusterRadius` deals with the cluster circle (whose radius is `clusterRadius`) as well as the starting circle (radius `addCircle`) and the killing circle (radius `killCircle`).

You are encouraged to take a look at the various functions and see if you can understand how they work. There should be plenty of comments in the code that help with this.

Exercises

There are two exercises in this assignment, however their exact implementation and discussion in the report are up to you.

(*Version control tip*: It is a good idea to make backup copies of your files as you go forward with the exercises. That way you can always return to the older versions of the codes if you need to.)

Exercise 1 – measuring the fractal dimension

In this exercise, you will characterise the shape of the cluster by measuring its fractal dimension. The theory of fractal dimensions is discussed in Section 1 (and in Lecture 1).

As a first approach, edit the code so that when you hit '0', the program prints the number of particles and its size (radius) to the console (`cout`). Grow a few clusters and calculate their fractal dimensions using Eq. (1). Take $a = 1$ as a first guess.

Next, modify the code so that it prints values for N_c and R to a file, as the cluster grows. Some hints as to how to do this are given in part 5 of the 2nd PC lab handout. (As an example, if the cluster grows to size 1000 then you might use the file to record the size when the number of particles in the cluster is $10, 100, 200, 300, \ldots, 1000$.)

After you have generated and saved the data, I suggest to carry out the following data analysis using some other software (e.g. MatLab, Python). By using Eq. (3) and plotting suitable graphs, estimate the fractal dimension of the growing cluster. You might want to think about: (i) if you generate several clusters, do they all give the same value for d_{f}? (ii) what is the best way to present and analyse the data? (iii) can you estimate an error bar on the fractal dimension? Include the graphs in your report and explain what you can conclude from them.

If you use bigger clusters (e.g. 3000 particles or more), do you end up with the same answer for the fractal dimension?

Exercise 2 – What happens if particles do not always stick when they touch the cluster?

So far, you assumed that a particle sticks to the cluster whenever it touches an existing particle. This may not be the case in experimental systems. For example, if a particle arrives with a lot of energy, it might "bounce off" instead of sticking.

Modify the code so that each time the particle touches the cluster, it has a finite probability p of sticking. You will need to modify the `checkStick` function. If the particle is about to attach itself, generate a random number `x` between 1 and some upper cutoff m (for example, take $m = 1000$). If x/m is less than p then the particle should stick.

Since you have changed the algorithm by which the cluster grows, your report should include a precise description of the new algorithm. (If the particle does not stick, what precisely happens on subsequent steps of the algorithm?)

How changing this probability affects the shape of the growing cluster. Does the fractal dimension change? Why? (Note that if you set the sticking probability $p = 1$ then you should recover the original results.)

Your report should discuss your results and your methods; it should also summarise your understanding of the system. Why do the clusters have the shapes that they do? To what extent to the fractal dimensions depend on the algorithms by which the particles move? Why? Can you compare your results with experiments (for example those mentioned in Lecture 1)?.

Version history:
AS, 7 Feb 2022