Explanation-of-algorithm

Jens Xin Hyldgaard

January 2025

1 Introduction

This document will explain my solution to the IPS programming challenge.

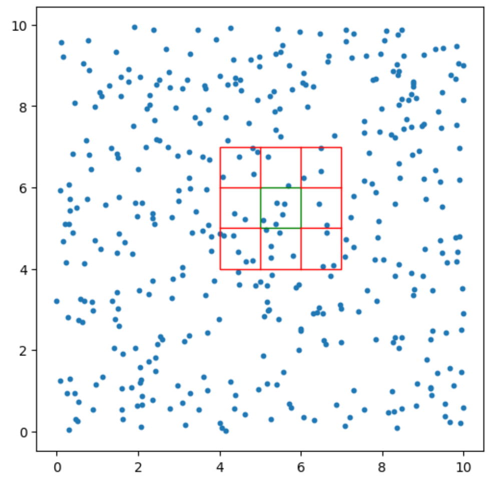
2 Problem

Given X, Y, Z coordinates of points in 3D space, calculate how many pairs are within the distance of r = 0.05.

3 Algorithm

3.1 Initial Approach

There are multiple ways to attack the problem, KD trees, Ball trees and probably other algorithms could be implemented. The idea I choose was to use the fact that the 3D space could be partitioned into cubes of side lengths 0.05, any given cube will then have no points that can reach (from here on we will refer to two dots being able to reach each other if: $|\vec{v_{dotA}} - \vec{v_{dotB}}| < r = 0.05$) another cube that is not within the cubes neighbours. That sounded complex, but is not, look at the figure below:



For simplicity we illustrate it in 2D. Lets say $r = 1$, and each square has side length 1. Any given dot inside the green square can only reach dots inside the red square. That means we don't need to compare dots inside the green square against any dots outside the red squares, saving us lots of time. We apply this idea in 3D space where squares become cubes. We partition 3D space into cubes. Neighbouring cubes will by definition be all cubes with infinitesimal distance to the home cube.

We now want to implement our idea. The main problem is to avoid counting a cube twice. Here is the algorithm we finally choose:

Iterate through all cubes, for all cubes:

1. **Counting process 1:** Count the number of pairs that can reach each other where one point is inside the home cube, and another point is inside *relevant neighbours**.
2. **Counting process 2:** Count the number of pairs that can reach each other within the cube (counting process 2).
3. Sum the two counts.

Definition*: Relevant neighbours Given home cube with index: (i, j, k). Relevant neighbours would be:

- $(i, j + 1, k)$
- $(i + 1, j + 1, k)$
- $(i - 1, j + 1, k)$
- $(i, j + 1, k + 1)$
- $(i, j + 1, k - 1)$

- $(i+1, j+1, k+1)$
- $(i+1, j+1, k-1)$
- $(i-1, j+1, k+1)$
- $(i-1, j+1, k-1)$
- $(i, j, k+1)$
- $(i+1, j, k)$
- $(i+1, j, k+1)$
- $(i+1, j, k-1)$

This choice of "relevant neighbours" might seem strange, but this ensures we do not commit any double counting.

For ease of visualisation, view the figure below:

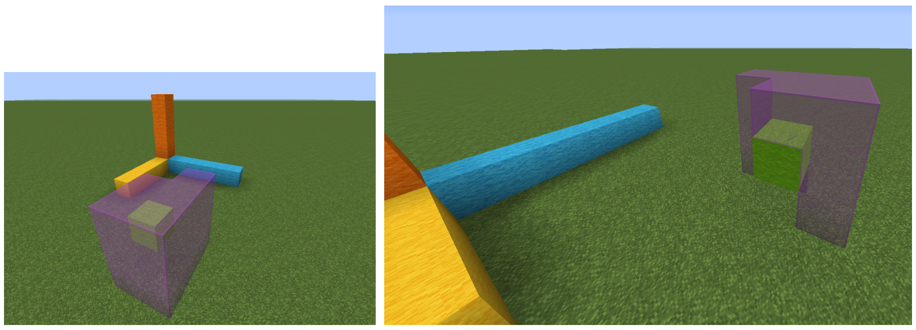


Figure 1: 3D visualization of relevant neighbours.

The yellow blocks represents the i-axis, the cyan blocks represents the j-axis, and the orange blocks represents the k-axis. The green block represents the home cube, the glass around the green cube represents *relevant neighbours*. In order to convince yourself that this ensures no double counting, imagine a $cube_b$ placed in a neighbouring position to the home cube (the green cube). If $cube_b$ is placed in a relevant neighbouring position, $cube_b$ will not have the home cube as a relevant neighbour. If however $cube_b$ is placed in a non-relevant neighbouring position, $cube_b$ will have the home cube as a relevant neighbour.

3.2 Initial Implementation

We now have a choice to make when iterating through every cube, one could either create each cube and its relevant neighbours ad-hoc as we iterate through all cubes. Or one could create a data structure to store all cubes and its points prior to iterating everything. Both have pros and cons. The cons of creating cubes ad-hoc is the fact that we will create each cube more than once, on the other hand side, it will save a lot of memory. The cons of creating a data-structure prior to iteration is that it will take up memory, it will however be faster, since we will not create any cube more than once. I choose to go by the later option, creating a data-structure prior to iteration. The data-structure we will use is a dictionary, where the key is a tuple of integer indices of the cube, (i, j, k), and value is a numpy matrix for all dots and their X, Y, Z coordinate inside the cube.

It would turn out that the creation of dictionaries takes a lot of time, and we could save a large amount of time by partitioning the 3D space with larger cubes, thus, lowering the total number of cubes created. This however, makes both counting processes slower. Here we need to do parameter optimisation, which we discuss later.

3.3 Cluster handling

By plotting the data initially, we see that data points are clearly concentrated in certain areas:

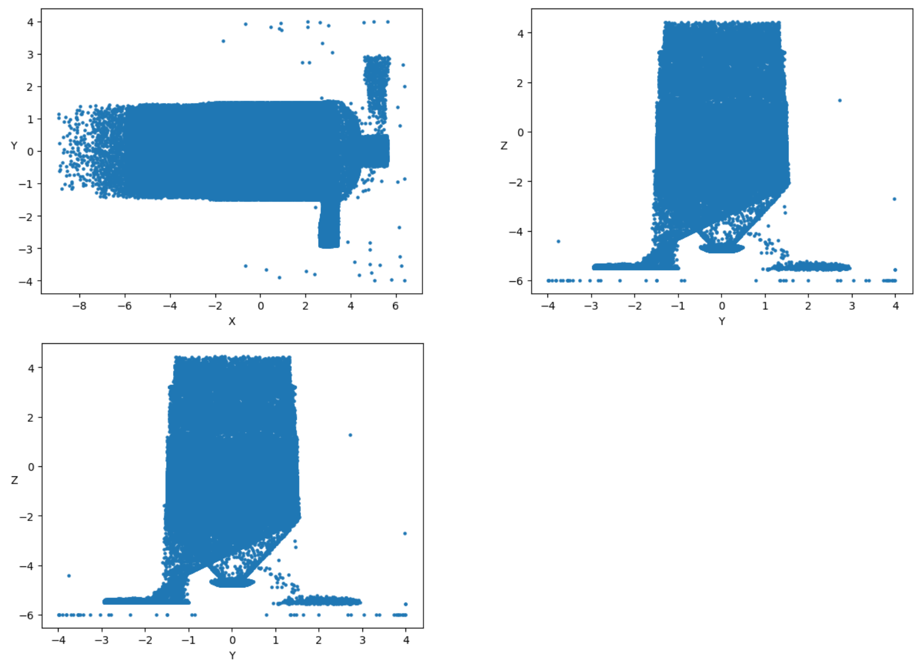


Figure 2: Data from positions_large.xyz plotted.

If our algorithm iterates over a large chunk of empty space, it takes lots of time since it needs to create a bunch of empty cubes. What we implement here is for the algorithm to check whether a larger chunk of space is empty, if empty, don't bother to create cubes, if not empty, create cubes. Example:

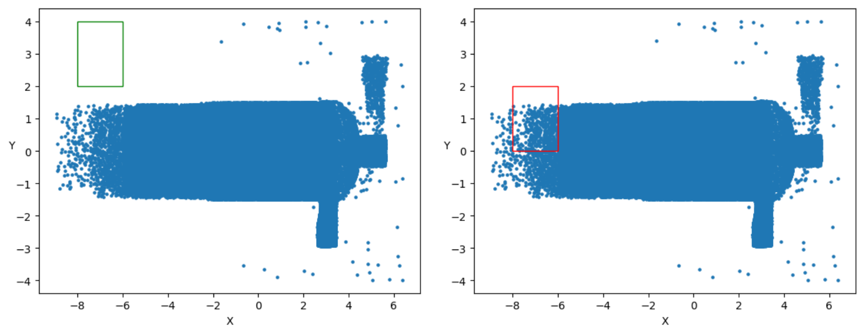


Figure 3: Examples of empty, and non empty cube.

In Figure 3, the green box would not raise a need to create sub boxes and would then skip to the next box. The red box, would however raise a condition to create sub boxes and would then create sub boxes for the entire red area. This implementation lead to an improvement in runtime. This larger cube that decides whether or not to partition itself into smaller cubes, will be called *sorting cube*.

We are now faced with an optimisation problem: How big should the larger cubes be (sorting cubes), and how big should the normal cubes be? Simulation for this problem was runned and the following data was found:

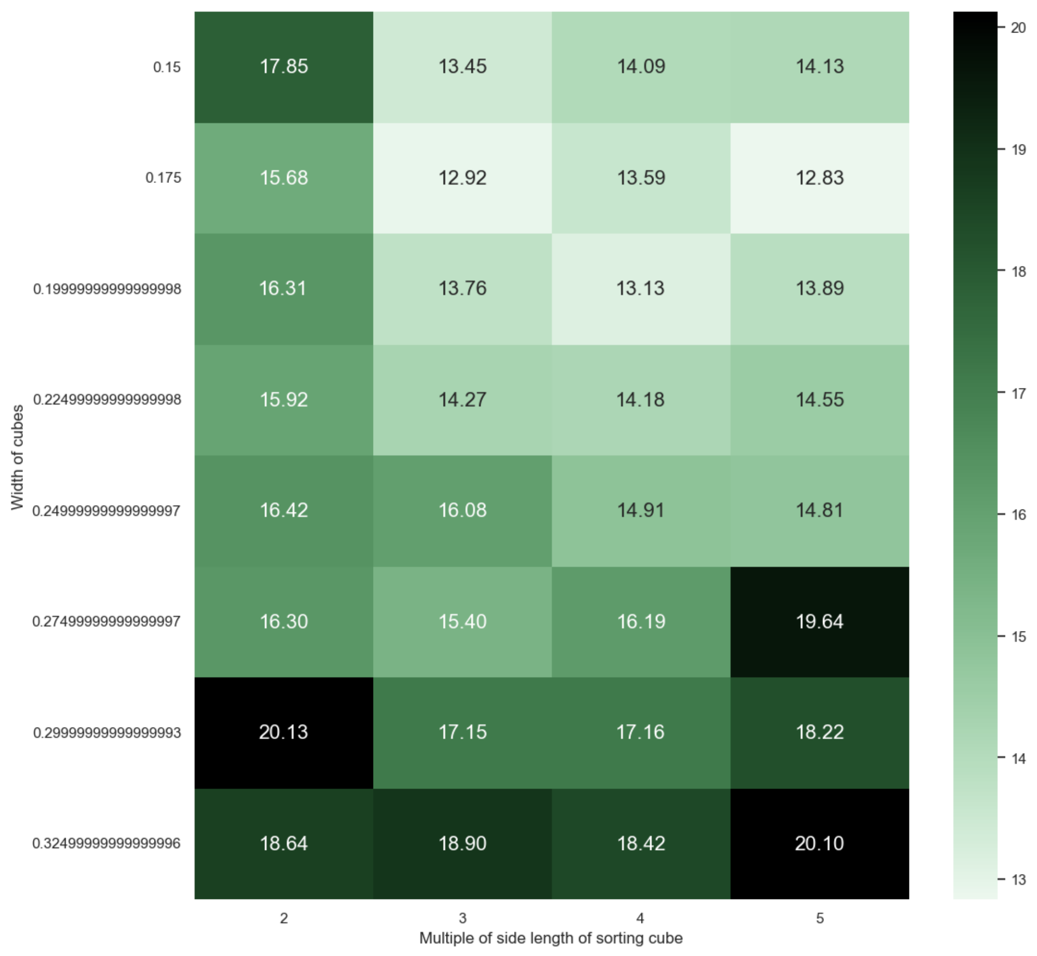


Figure 4: Simulations of parameters: Side length, and sorting cube multiple.

In the y-axis, we have the width of the cubes, and in the x-axis, we have the multiple for how many times larger the sorting cube width should be, compared to the width of the normal cube. As we can see, the optimal width for the normal cube seemed to be 0.2, and the optimal multiple for width of the *sorting cube* seemed to be 4 (the sorting cube would then have width $0.2 * 4 = 0.8$). Other combinations had lower runtime, but this combination had the best neighbours with low runtime, leading to the algorithm being more stable for any given dataset.

4 Complications

Through out the development of the algorithm, some complications where found:

- The creation of the dictionary was inefficient since, we had to partition the dictionary storing 3D space into 12 (number of threads on CPU) equal sized dictionaries, in order for parallel-processing to work, and then merge the 12 dictionaries at the end. Theoretically, this shouldn't be needed, since we would not create any race condition since the function should only iterate over each index once. But for some reason, it iterates over some indices twice, which leads to race condition. I suspect this is due to *numpy.arange()* being inaccurate, and creating overlapping partitions.
- The algorithm optimal for the larger dataset *positions_large.xyz*, was not optimal for *positions.xyz*.
- During the creation of the algorithm, we utilized simulations in order to see what algorithm gave the best runtime. It could be argued that this might lead to overfitting, since the next dataset we apply the algorithm on, might not have similair attributes to *positions_large.xyz*. It might not have clusters, it might not have the same density of points.

5 Further Ideas

This algorithm can of course be even more optimised. Ways to optimise future algorithm could be:

- In our algorithm, we have implemented CPU-parallelisation but not GPU-parallelisation.
- Take into account the amount of points in the given dataset. Since we saw that the optimal algorithm for *positions_large.xyz* was not optimal for *positions.xyz*.