Group 8:
Computational Methods & Modelling Report

Name	Percentage contribution (%)	Contribution description
Andrew Arstall	12.5%	- Report introduction - Report for part of Task C - Report for part of Task D
Sean Chen	11.5%	- Report for part of Task C
Lexa Drummond	12.5%	- Report introduction - Report for part of Task B - Report for part of Task C - Report for Task D
Alexander Martins da Silva	11.5%	- Code for part of Task B - Report for part of Task B
Amirul Hakeem Norazam Norazam	11.5%	- Code for part of Task B - Report for part of Task B
William Robertson	11.5%	- First draft of code for Task B
Martin Starkov	65% of code 15% of report	- Code documentation - Code for Task A - Code for Task B - Code for Task C - Code for Task D - Code for Task E - Report illustrations - Report for part of Task C - Report for Task D - Report for Task E
Fred Zikry	35% of code 10% of report	- Code for Task B - Code for Task C - Report diagrams - Report for part of Task B - Report for Task C

Introduction

This report details the way in which Group 8 has solved the tasks for the CMM3 group project. Much of these tasks have been achieved by adapting the Lagrangian approach of advection and diffusion to create a useful and efficient simulation tool. Advection and diffusion are essential principles for describing the movement of fluid particles, which is used in many applications such as the design of aerodynamic surfaces.

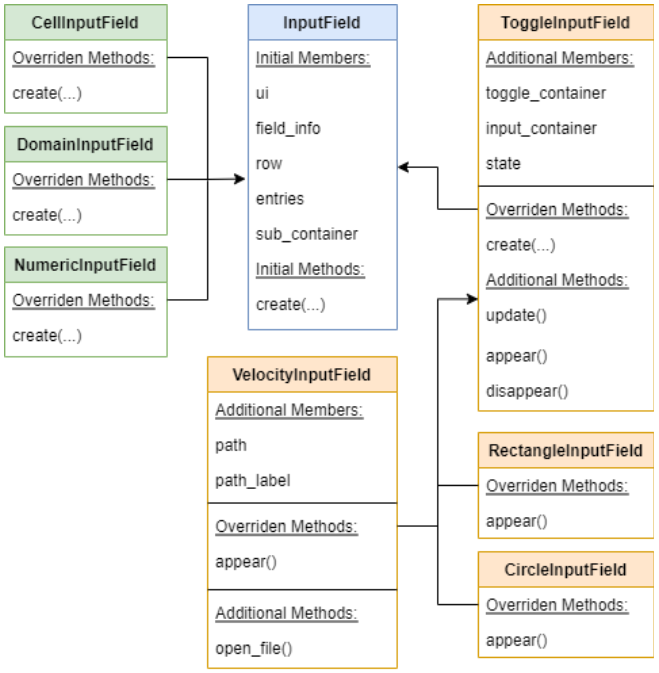
The simulation is started by running the "interface.py" file, which contains the GUI code for the project. This file works synergistically with multiple other files. The "simulation.py" file acts as a repository for functions used to perform calculations such as generating coordinates for the particles and for defining fundamental simulation parameters. The file "utility.py" handles miscellaneous tasks such as setting up plot windows and storing code that would hinder the readability of the interface file. The "config.json" file is a JSON file where parameters, data types, and default entry field input values are stored and defined.

User interface (Task C)

Once the application starts, the user will be greeted with an interactive window created with the Tkinter package. The main menu splits into three submenus.

Button	**Description**
Animated Chemical Spill	- Solves Task D by displaying an animated plot of a chemical spill. - Based on conditions from the "config.json" file as specified by the assignment.
Validation Tasks	- Compares the reference solution to calculated concentrations. - Calculates and plots the RMS error against the number of particles. - Allows for logarithmic and linear plotting over a range of particle numbers and time steps. - Note: the above buttons may take time to respond while they compute the simulation steps.
Custom Conditions	- Allows the user to define their own conditions for advection and diffusion. - Highlights part of all aspects of the project (including Task E). - For non animated plots check the console for simulation progress.

In order to make the user interface dynamic, a class hierarchy (as seen in the right diagram) has been designed to allow for different types of user input fields to be generated. The "InputField" class is an abstract class which defines how user inputs will be stored. This inheritance structure enables there to be many different types of input visuals, as seen in the "Custom Conditions" menu. The creation of input fields consists primarily of generating Tkinter widgets in a formatted grid and appending the interactive ones to an entry list. The entry list is a python list which contains widget instances or variables to be parsed by the code. Once the user presses "Run Simulation", the code will look through all the input field instances, go through all the entries and retrieve data from them and store it in a dictionary.

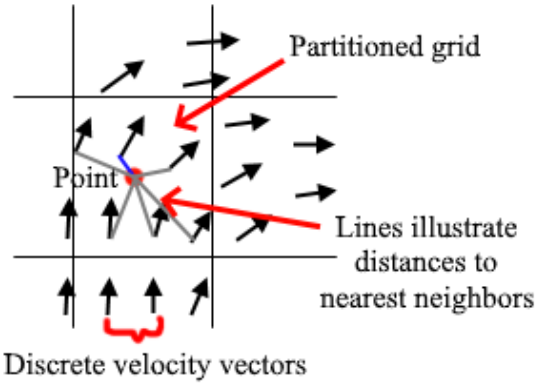


Engineering simulation: Chemical spill (Task D)

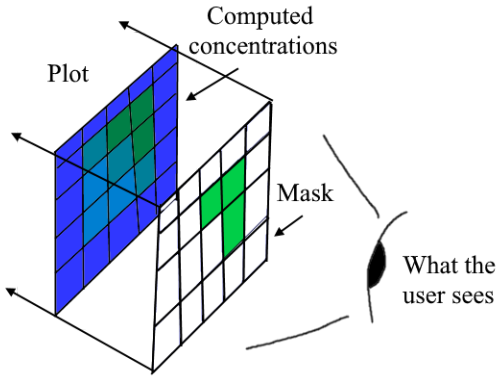
Pressing the "Animated Chemical Spill" button causes the program to read the numerical constants and initial conditions required for the chemical spill from a JSON file. If needed, the user can edit the specifics of the chemical spill through this file. However, the "Custom Conditions" button allows for easier customization of parameters if desired. The parameters read from the JSON file are passed to the "Simulation" class defined in "simulation.py" which creates an instance of a simulation run. This class contains all the mathematical formulas required to model the chemical spill.

Addition of velocity field

One of the early challenges faced in creating the program was the fact that while theoretically a velocity field is supposed to be continuous, the provided "velocityCMM3.dat" data contains a set of velocity vectors for discrete coordinates. To solve this problem, a space-partitioning structure called a k-d tree was chosen. The data structure takes in a set of coordinates and using binary space partitioning algorithms allows the user to query for the nearest neighbor of a particle coordinate. This enables the simulation to find the velocity vector closest to a point which does not lie on any of the discrete velocity coordinates. The image on the right illustrates a simple nearest neighbor algorithm.



Chemical concentration colour mapping



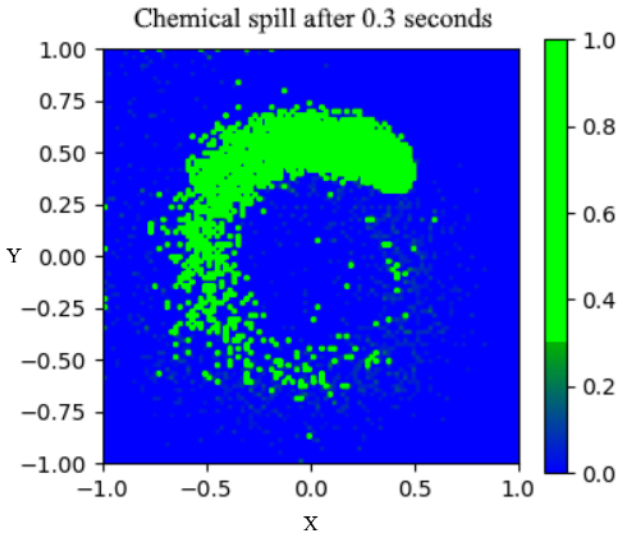
When the chemical spill simulation is initialized, the information about colour mapping (i.e. relevant concentrations and their assigned colour bar colours) are pulled from the JSON file. Two arrays are plotted on the graph at any given time. The first array contains all the concentration values in the grid cells as calculated by the "Simulation" class' "calculate_concentrations()" function. A numpy function (np.where) finds and stores (into the second array) the locations where concentrations have exceeded the defined threshold (0.3). This second array is essentially a mask which is updated each simulation step. By plotting this array on top of the computed concentration grid, the user is able to see a trail of concentrations appear following the spill.

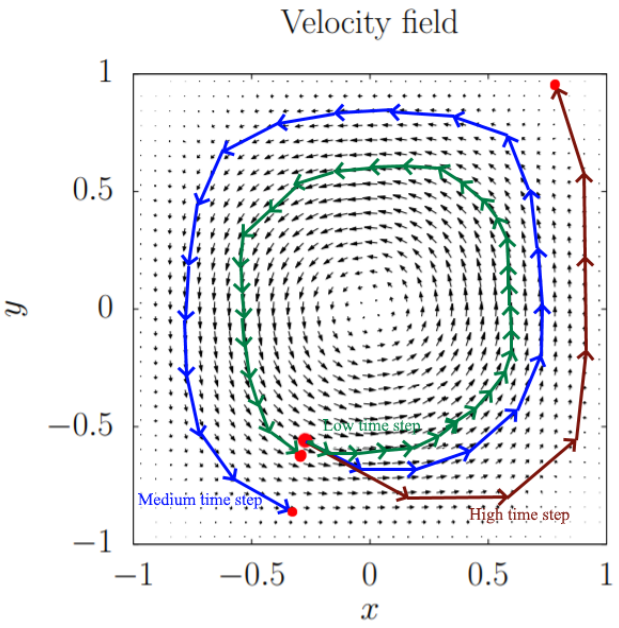
Initial chemical patch positioning

The chemical spill occurs in a circular arrangement so it is essential to control the location and size of the centre of the circle. This is done by a function called "add_circle()" inside of "simulation.py". The function calculates the distance of each particle to the center of the circle using the formula for the equation of a circle:$(x - a)^2 + (y - b)^2 <= r^2$. Then, particles within this radius are set to have a value of 1 (referring to concentration of the chemical).

Engineering applications

As seen in the right plot, an ocean current can very quickly cause a chemical to diffuse and spread around the domain causing many areas to exceed the concentration threshold. This illustrates how, for example, oil spills in the ocean may very quickly be carried long distances and exceed safe concentrations, causing ocean life to be exposed to hazardous chemicals. Tools such as this fluid simulation may be used in real engineering applications as a preliminary model to predict the impacts of these spills on marine habitats and aid in oil spill clean up efforts.





Insights into time step choice

An interesting phenomena observed while designing the simulation was the impact of time step choice on the interaction between particles and the velocity field. Since the velocity is multiplied by the time step it is clear that lower time steps cause less travel between simulation steps. By exaggerating the distance travelled in one time step, the arrows in the diagram to the left demonstrate a spiral phenomenon. The error in a particle's position essentially compounds over time, causing it to spiral outward. This means that the larger the time step, the more clearly the effect is seen. In practice, time steps of higher than 0.05 would cause the patch of chemical to be quickly flung out of the velocity field 'orbit' and pushed to the sides of the container. For lower time steps the particles travelled much longer in the center of the field.

Approaching the same idea from a mathematical perspective, one may consider the simulation as a continuous integration of velocity over time, where the time step is equivalent to the width of a Riemann sum rectangle. From calculus, it is expected that the error of this approximation will fall as the time step approaches 0. However, due to computational limitations this is not possible to achieve in practice.

Validation (Task B):

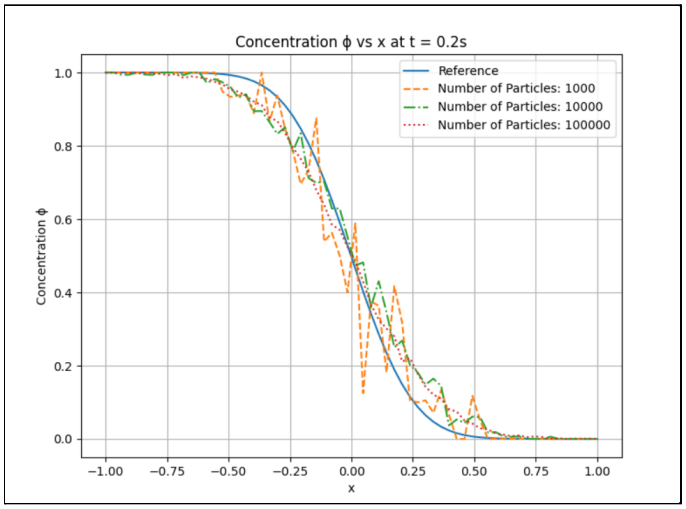
To fulfill the validation requirement for the project, comparisons are made between the reference solution data and the data produced by the simulation. The differences between the two sets of data are quantified by the Root Mean Square Error (RMSE). To show the effect of the time step (dt) and the number of particles on the quality of the simulation data, a range of time steps and particle quantities are inputted into the "Simulation" class, producing a range of data to be used for comparison and validation. To display these disparities and the effect of the parameters on the quality and accuracy of the simulation, three plots are generated:

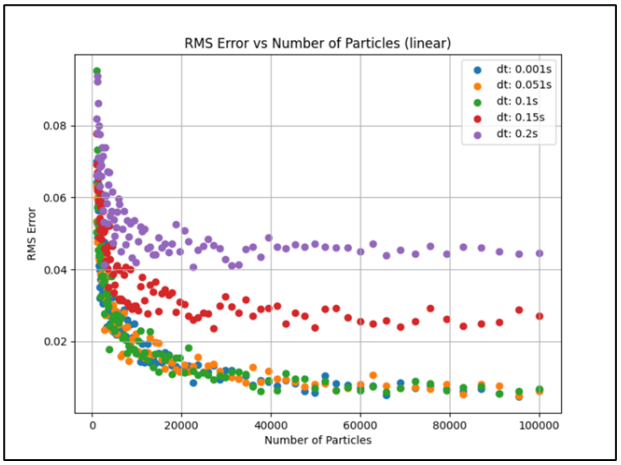
- Concentration (y) vs particle count (x) graph.
- RMSE (y) vs particle count (x) graph at multiple dt values for each particle count.
- The same graph as above but in logarithmic scale with logarithmic regression lines to calculate the constant Beta as requested by the project specification.

Problem faced	Solution
Reference solution data in a different domain than concentration data.	The reference concentration data was given in the (-1, 1) domain, rather than a (0, 63) domain, as it is in the simulation. The solution was to create a linear interpolation function for the reference data which would allow the program to map the simulation concentrations to the reference domain.
How to find constant Beta, β	$Error = a \times N_p^{\beta}$ This equation is plotted for each set of a and βusing scipy.optimize.curve_fit to calculate constants of the regression line. Having lines allows us to more easily compare the effect of the time step on the quality of the data and to see where diminishing returns start to occur. Theβvalues are shown on the relevant plot.
How to reduce noise to enable logarithmic regression to be more accurate	Initially the regression lines were going in wrong directions due to the variance of RMSE values at each particle count, especially at low particle counts where simulation data is less consistent. This was ameliorated by implementing scipy.signal.lfilter to filter out some noise before passing the noise-reduced data set into the curve fitting function.

By plotting three different numbers of particles increasing by a factor of 10 each time, it becomes apparent from the plot on the right that the greater the number of particles used in the simulation, the more the lines tend towards the reference line which is the true value of the solution. The lower the number of particles, the greater the variation from the true value. The Root Mean Square error can be found using the formula:

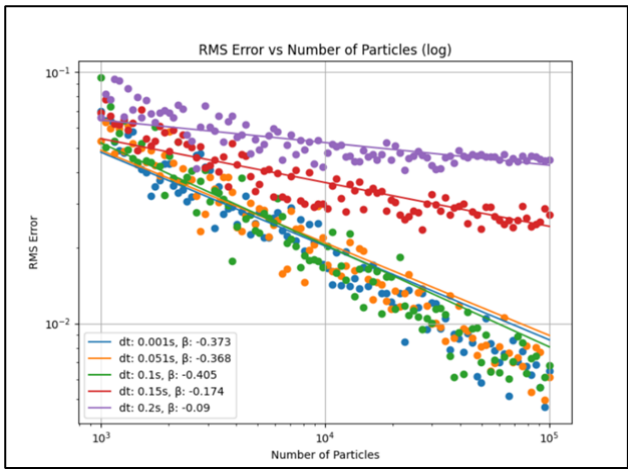
$$RMSE = \sqrt{\frac{\Sigma(predicted - Actual)^2}{N}}$$



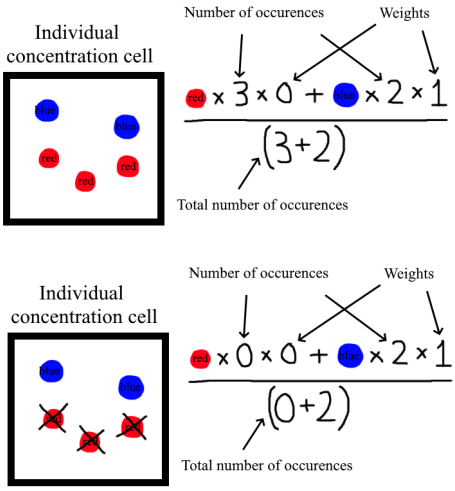


Plotting the global RMS errors linearly against the number of particles, as shown on the left, highlights how the RMS error drops exponentially when increasing the number of particles. Notably, the exponential decay of each set of points plateaus at around 40,000 particles. This indicates that running the simulation at more than 40,000 particles hardly affects the global error and is nonoptimal as it takes significantly longer. Furthermore, the high scattering of particles at low numbers of particles indicates that the simulation is very inaccurate with low numbers of particles as is expected since some concentration cells will be empty. It is evident from the plot that the global error is higher for larger values of dt, as predicted by the discussion in the chemical spill section of the report.

When displaying the same graph with a logarithmic scale, as shown in the right plot, the global errors collate to straight lines proving that the decay is indeed exponential. All lines have a downward trend. There is also very little difference of RMS error between dt=0.001s, 0.051s, and 0.1s as all errors are distributed in the same area. This concludes that running the simulation at extremely low values of dt provides very little additional accuracy and should be avoided. Based on both the plots, the optimal conditions for computation speed and solution accuracy for this fluid simulation seem to be at a dt of about 0.05s with around 40,000 particles.



Method improvement (Task E):



The performance of the code is significantly bottlenecked by the k-d tree search used to determine the velocity vectors at a given particle coordinate. The most direct way to impact this bottleneck is by reducing the number of queries, and hence the number of particles represented in the simulation. The right illustration shows how the concentration of a cell is calculated. Since the weight of red particles is 0, removing all red particles from the container will have no impact on the numerator of the equation, but will reduce the denominator by impacting the total number of particles in the cell. The denominator of this equation can be considered as a relative scale to which the number of blue particles in each cell is compared.

At the beginning of the simulation the particles are distributed in a uniform manner. This means that each cell with any blue particles in it is expected to have around the same amount as the other cells. Computing the average number of blue particles per cell at the beginning of the simulation gives a relative scale which can be used in the denominator. Concentration can now be calculated by comparing the number of blue particles in a cell to the average amount of blue particles per cell at the start. This approach has one major benefit and three minor flaws. The benefit is that the number of particles in the simulation can be reduced by a factor of two, which speeds up the aforementioned bottleneck. One flaw that arises is that concentrations can exceed 1.0 if a cell is more dense than the average cell density in the beginning. To avoid this the concentration is capped at 1.0. An assumption is made that due to uniform diffusion in all directions most cells will maintain around the same number of particles. This of course is not true, which brings up the second minor flaw. Velocity fields which push particles into specific cells will create higher and lower density regions which in turn will make capping the concentration lose information about the field over time. These types of fields would likely not fare well under this improvement model. The third flaw is that if the number of cells is increased without increasing the number of particles, the estimate will become worse and worse. This is because the average number of particles per cell will decrease, meaning that any random variation in the density of particles will be much more pronounced (this manifests itself in a grainy concentration field). The increase in speed may overcome these flaws in some applications.

An entirely different method of improvement would be mapping the velocity field to the data differently. For instance, using standardized coordinates as is done when calculating concentrations. This would eliminate the need to nearest neighbour query a space-partitioning structure as each coordinate would be converted to an index which could easily access a velocity vector field with the same standardization. And lastly, of course, implementing the k-d tree on a GPU would drastically improve its performance and hence speed up the application significantly.