UiT The Arctic University of Norway

Faculty of Science and Technology
Department of Computer Science

# Particular: A Functional Approach to 3D Particle Simulation

Marius Indreberg

Master Thesis Spring 2021

# Abstract

Simulating large bodies of entities in various environments is an old science that traces back decades in computer science. There are existing software frameworks with well built mathematical models for approximating various environments. These frameworks are however built on imperative programming fundamentals often following a object oriented paradigm.

This thesis presents Particular a 3d particle simulator software library for simulating movements of independent entities on a time dependant three-dimensional vector field using a functional approach. Particular uses functional programming paradigms to create a quite customizable, flexible and maintainable library based on lambda functions with all relevant parameters encapsulated in closures. Particular uses a functional implementation of a Entity Component System software architecture usually found in game development to create a highly performant, flexible, data oriented design. Which uncouples the data with the aforementioned lambda functions that predicate particle behaviour.

According to evaluations particular shows a significant performance increase with regards to execution time compared to comparison to other contemporary trajectory simulation frameworks such as opendrift. With some evaluations showing a 900% faster execution time under certain conditions.

# Acknowledgements

I would like to thank Jonas Juselius and John Markus Bjørndalen for supporting me with the technical aspects of this thesis, I could not have completed this thesis without either of you.

# Contents

# List of Figures

# List of Tables

# 1 Introduction

In computational oceanography there is often a need to simulate the drift,spread and fate of physical entities in the ocean. Examples include spreading and dispersion of pollutants or chemicals, spreading of parasites or viruses and sedimentation processes. These entities may grow very large in numbers with varying complexities in calculating their drift within their respective lifecycles.

There already exists numerous software frameworks for modelling trajectories of objects drifting in the ocean. Popular Lagrangian packages are OpenDrift [1]and OceanParcels [2], both written in the Python programming language [3].In the Eulerian space, FABM [6] is an example of a biogeochemical transport model written in Fortran.

In this thesis we will introduce particular. A functional first 3d particle simulator prototype. Particular will use a functional approach to particle simulation, which lends itself very nicely to parallelization. Particular also gets inspiration from game development software architecture to ensure performance and maintainability without giving up flexibility. Maximizing the use of the generic lambdas to ensure a user can create their own simulation with minimal amounts of code while getting a performant output.

## 1.1 Thesis Statement

This thesis shall investigate the feasibility of developing a fully functional particle simulator in the functional programming paradigm. Which operates as performant and extendable as other contemporary libraries developed in the object oriented programming paradigm, without sacrificing functional programming cornerstones such as immutability and type safety.

## 1.2 Scope and Assumptions

This particle simulation will rely entirely on oceanographic mathematics provided by the opendrift library, and will be focusing on the mathematical aspects of the particle simulation in the smallest degree possible. The end goal and scope of this thesis is to create a functioning particle simulator which may accurately approximate oceanographic trajectories to the extent that it is usable in a real world application.

There will not be any detailed mathematical proofs or test of correctness with the mathematical advection functions, as it requires a level of mathematical prowess that we simply do not currently possess.

## 1.3 Context

This thesis was written in the context of Serit IT partner who is in the process of developing a technology stack for analyzing oceanographic data. With this particle simulator being but one layer of the stack.

## 1.4 Method and Approach

The Task Force on the Core of Computer Science presented in their final report a way to divide the discipline of Computing into distinct paradigms. The three major paradigms are:

**Theory** which is rooted in mathematics and consists of four steps, followed in the development of a coherent and valid theory:

- First, one characterizes the objects of study, or definition.

- Then, hypothesize possible relationships between them, or theorem.
- Further, determine whether the relationships are true, or proof.
- Lastly, interpret the results.

A mathematician expects to iterate on these steps, as they encounter errors or inconsistencies.

**Abstraction** which is rooted in the experimental scientific method and follows four steps when investigating a phenomenon:

- Form a hypothesis on the phenomenon.
- Construct a model to make a prediction.
- Design an experiment to collect data.
- Analyze the results.

A scientist expects to iterate these steps, as they encounter problems such as when a model's predictions disagree with experimental evidence. Modeling is another word for this paradigm.

**Design** which is rooted in engineering and consists of four steps, followed in the construction of a system to solve a problem:

- State the requirements.
- State the specifications.
- Design and implement the system.
- Test and evaluate the system.

An engineer expects to iterate on these steps, as they encounter issues such as the system not upholding requirements to a satisfactory level.

In this thesis, the last paradigm will be worked, design. We state the requirements and specifications of the system associated with our conjecture. Further, we present a design for a system, implement the requirements needed for our system, and then create a prototype based on the design. We then evaluate our prototype through a series of evaluations and benchmarks.

## 1.5 Contributions

This thesis contributes by providing a prototype implementation of a 3d particle simulator written in the F# programming language, following the functional programming paradigm. This prototype is open source and may be found at the owners github account and is also provided as a zipped attachment as a part of this thesis.

## 1.6 Outline

The thesis is structured as follows:

**Chapter 2: Background** Details the technical background of the mathematical concepts encountered during this thesis as well as other computer science based concepts and data structures. Background Also contains an overview of related works.

**Chapter 3: Design** Describes the architecture and design choices of particular.

**Chapter 4: Implementation** Describes the implementation specific details which were encountered when creating the particular prototype.

**Chapter 5: Alternate Approaches** Details all the alternate approaches which could have been used. Which will be referenced in the evaluation chapter as basis for comparisons.

**Chapter 6: Evaluation** Details the findings of the experiments, how they were conducted and what the experiment results may imply.

**Chapter 7: Discussion** Describes the findings of the evaluations, comparisons to opendrift and other relevant frame works.

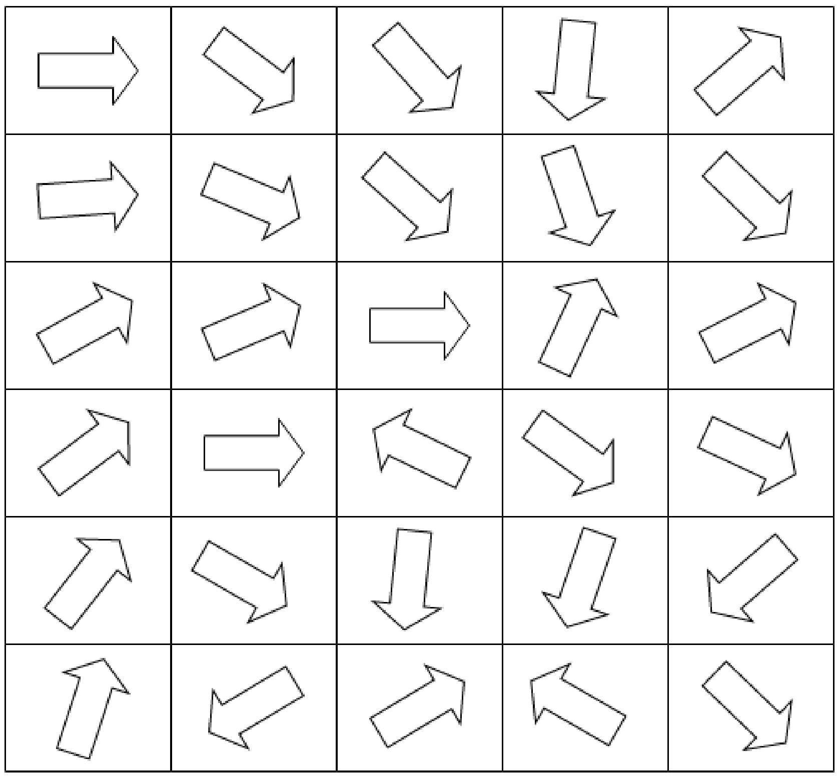
**Chapter 8: Conclusion** Summarizes the thesis and describes potential avenues for hfuture work.

# 2 Background

This chapter will detail all the technical background required to understand all the concepts provided with this thesis.
Section 2.1 Will explain vector fields.
Section 2.2 will go through various ways of solving ordinary differential equations.
Section 2.3 will detail the concept of netCDF formatted files.
Section 2.4 describes KD-trees.
Section 2.5 outlines the .net garbage collector.
Section 2.6 will explain the ECS data architecture.
Section 2.7 describes and defines concepts with regards to particle simulation which will be referred to in the rest of the thesis.
Section 2.8 Will go through related works.

## 2.1 Vector Fields

**Figure 2.1:** Vector Field Example. Arrows signify arbitrary vectors within the field.



A vector field is an assignment of a vector to each point in a subset of space. A vector field can be thought of as a collection of vectors over a set bound, each vector independent of each other pointing in arbitrary directions. Vector fields are often used to model speed and direction of a moving fluid through space. A visualization of an example vector field composed of arbitrary vectors can be viewed in figure 2.1.

## 2.2 Solving Ordinary Differential Equations

Both Particular and Opendrift use numerical procedures to solve Ordinary Differential Equations to calculate advection of particles. In Opendrift and Particular there are two different approaches of varying order to solve these differential equations.

### 2.2.1 Euler's method

$$Y(i+1) = Yi + f(Ti, Yi)\Delta t \tag{2.1}$$

Euler's method[1] is a first order numerical procedure for solving ordinary differential equations with a given initial value. As the Euler method is a first order method the error per step is directly proportional to the square of the step size and the error at any time is proportional to step size.

In the context of vectors the equation 2.1 can be read as the equation 2.2. With W as the position vector, V the velocity vector and delta t as the time step scalar.

$$w(new) = w(old) + (v * \Delta t) \tag{2.2}$$

### 2.2.2 Runga-Kutta

The Runga-Kutta[2] method attempts to overcome the problem of Euler's method namely the choice of a sufficiently small step size to reach a reasonably accuracy in the problem solution. The drawback being the method is more complex thus requires more computational power.

$$Y(n+1) = Yn + 1/6(K1 + 2K2 + 2K3 + K4)h \tag{2.3}$$

$$1/6(K1 + 2K2 + 2K3 + K4) \tag{2.4}$$

$$K1 = f(Tn, Yn) \tag{2.5}$$

$$K2 = f(Tn + h/2, Yn + h/2 * K1) \tag{2.6}$$

$$K3 = f(Tn + h/2, Yn + h/2 * K2) \tag{2.7}$$

$$K4 = hf(Tn + h, Yn + h * K3) \tag{2.8}$$

Runga-Kutta which is a euler method with a higher order than 1, involves slope calculation at multiple steps between the current and next discrete time values. The next dependant variable is calculated by taking a weighted average of these multiple stages based on a Taylor Series approximation of the solution. The weights in this weighted average is derived by solving non-linear algebraic equations which are formed by requiring cancellation of error terms in the Taylor series. In this thesis it will be focused on Runga-Kutta of order 4 which is the most popular method as it is a good compromise of accuracy and cost of computation.

As can be seen in the equation 2.3, is the equation of the Runga-Kutta. 2.4 Finds the weighted average slope. Equations between 2.5 and 2.8 are the 4 values of a 4th order runga kutta equation. With H defining the step size.

### 2.2.3 Geographic Projection vs Coordinate projection

In this thesis when referring to updating particle positions will be in reference to the Cartesian coordinate system, however when saving the results it is saved in a geographic projection system (longitude latitude). The main differentiation between the geographic and coordinate projection is the fact that Coordinate projection is defined on a flat two-dimensional surface. Geographic coordinate systems however are based on spheres which is much more appropriate when handling particle simulations on or around the earth as the curvature of the earth will significantly change the outcome of the projection when comparing the two approaches.

## 2.3 NetCDF

NetCDF[1] or network common data form is a data format that contain array oriented scientific data. NetCDF attempts to be a data format which can package large sizes of data which can be quickly fetched as arrays in an optimal fashion. As NetCDF is an array oriented format, the arrays are defined as variables, (e.g. Water currents or temperature), being able to have different amounts of dimensions. E.g. A variable of the temperature for a specific area in a vector field may have dimensions such as the time step, the depth and the actual area. With each index of the dimension pointing to a different value.

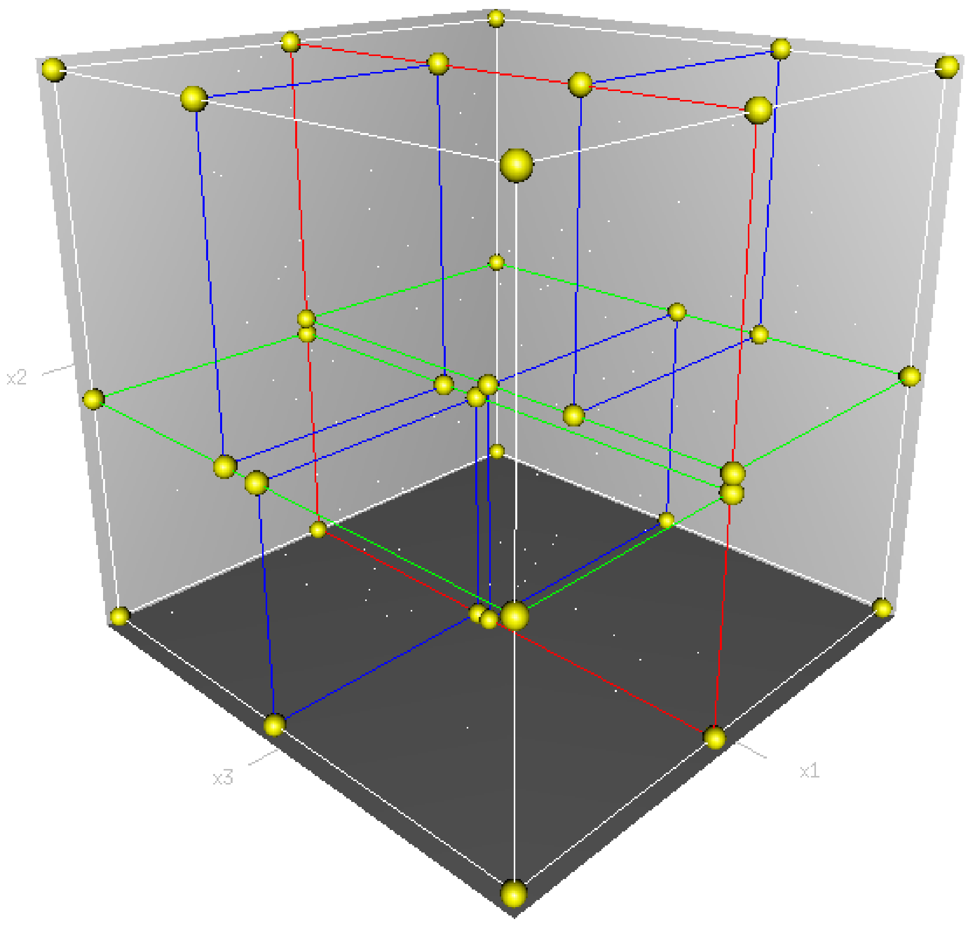
## 2.4 K-D Tree

A k-d tree is a space partitioning data structure that allows for points to be organized in a k-dimensional space. K-d trees are generally applied in situations where searches over an arbitrary geographical range occurs often.

The K-d tree is a binary tree in which each leaf node is a k-dimensional point.

1. https://www.unidata.ucar.edu/software/netcdf/

**Figure 2.2:** K-d tree in 3 dimensions.
(By Btyner - Own work, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=37229011)



Each leaf node can then be thought of as a subset of a larger area in a k-dimensional plane. These leaf nodes then may contain points on the plane which are available for searching.

The K-d tree promises a linear worst case for look-ups and insertions/removals as well as space complexity. A visualization of a K-d tree may be viewed in figure 2.2.

## 2.5 Garbage Collection in .Net

The .net Garbage collector[3], or GC, is the autonomous garbage collector which runs in the background in managed memory .net space. It allows developers to refrain from manually allocating and releasing memory, as well as allocating objects on the heap efficiently. It also provides memory safety by not allowing an object to use the content of another object.

When a new process is initialized the CLR[4] reserves a contiguous regions of address sapce for the process. This is referred to as the managed heap. This managed heap has a pointer which points to the next object in the heap to be allocated. This is crucially only for referenced types, as value types are kept on the stack and outside of the managed heap. When the application creates this next object the GC is tasked with allocating the memory and address space. The pointer then moves to the next object available for allocation.

When releasing memory the GC determines the optimal time to perform a collection of objects no longer used by the application. The GC determines which objects are no longer used by creating a graph from the applications root and each object that is not reachable from this root that is currently in managed memory is deemed unreachable and can safely be freed from the heap.

The GC is typically incurred in three scenarios; The system has low physical memory, the allocated memory on the heap surpasses a set threshold, the collect method is called. This is done automatically as the GC runs contiously in the background while the application is running.

The GC also has a concept known as generations. The concept is borne out of some considerations: Compressing memory is faster for a portion of the managed heap than the entire heap. Newer objects have shorter lifetimes and older objects have longer lifetimes. Newer objects tend to be related to each other thus have temporal locality.

For optimization purposes the managed heap is divided into three generations; generation 0, 1 and 2. This allows the GC to handle long and short lived objects separately. Each new object that is allocated starts at generation 0, if the object "survives" a collection are promoted to further up the generations. This allows the GC to release the memory in a specific generation rather than the entire managed heap for every collection.

This means that generation 0 contains objects that are newly created and are the most likely to be freed, in generation 0 you would find objects such as temporary variables etc. Generation 1 acts as a buffer between 0 and 1, it prevents the GC from having to reexamine objects in generation 1 when it performs a collection in generation 0. Generation 2 is where the long lived objects are located, and example of this is static data in a server application.

## 2.6 Entity Component System

Entity Component Systems[5] or ECS is an architecture for flexible high performance data computing often used in game development. The main selling point of an ECS architecure is the data oriented design which promises flexibility and maintainability while also promising good performance due to the locality of reference of the data which translates to using the CPU cache well. There are three main aspects to an ECS architecture; Entities, Components and Systems.

### Components

Components in ECS are the pure data, as such components does not have any methods or behaviour tied to them, only values. These values are what define entities and what form they actually take in the world.

### Entity

An entity is what ties a various component types together as a vertical slice of data. Thus a component can be as simple as a integer or some other unique identifier to identify which different component types creates an arbitrary group of data.

An example from the game development world would be a playable character entity having a position component, a sprite component and a collision component. Another example being an Entity being a foreign key with components being tables in a regular relational database.

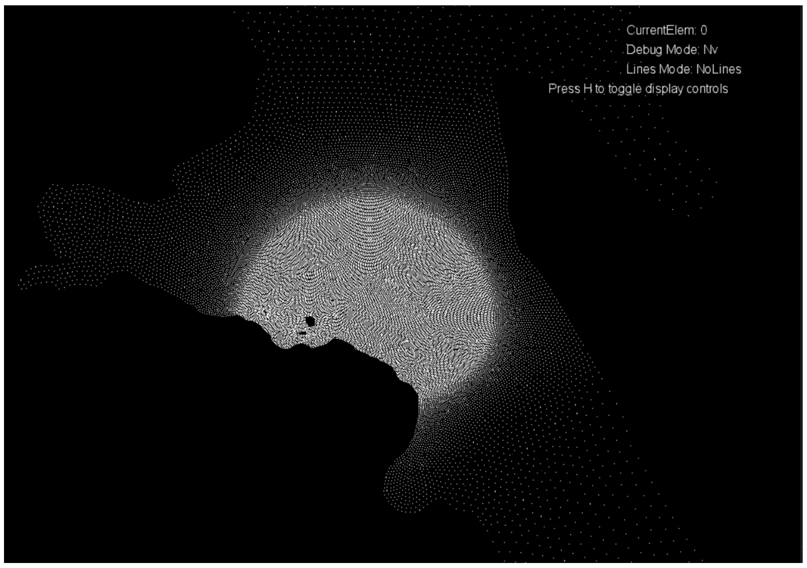
### System

The systems are the conduit for the programmer to enact change upon the world in a ECS architecture. A system is a function which runs a transformation on a subset of the components within the ECS. These systems are ran every "frame" usually in the context of game development.

## 2.7 Particle Simulation

As Particular is one stage of a larger pipeline of systems for handling simulated data. This section will detail some of the mathematical background for the particle simulation as well as defining various characteristics of the generated grids provided as input for Particular. In addition an example of how to mathematically calculate advection of ocean drift of particles will be explained in this section as it will be used as an example throughout this thesis in both the design and implementation chapters.

**Figure 2.3:** Visualization of unstructured grid data, dots are nodes creating triangular cells in the vector field.

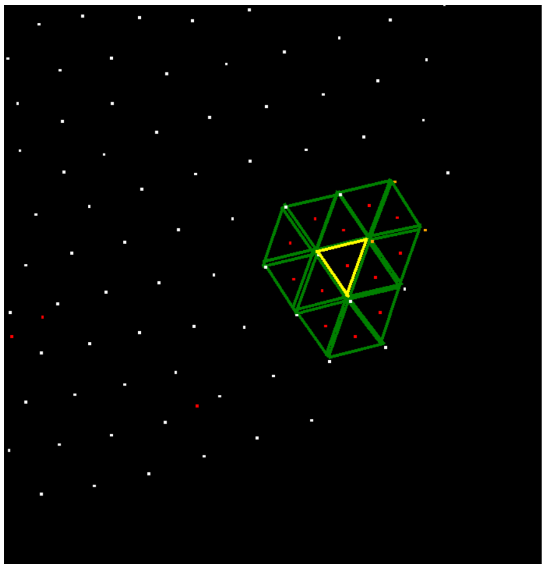

CurrentElem: 0
Debug Mode: Nv
Lines Mode: NoLines
Press H to toggle display controls

### 2.7.1 Grid Mesh

Further ahead in the pipeline a grid mesh file is generated which defines a vector field of positions and values for a unstructured grid mesh of triangles with each triangle having specific discrete properties and values defined such as temperature as an example. These values are calculated using the FVCOM[**FVCOM**] model. A visualization of the grid can be viewed in figure 2.3.

This unstructured grid has two components; nodes and elements.

**Figure 2.4:** Element and its neighbours, Neighbours have green borders.



### Elements

Elements can be thought of as triangular zones or cells in the mesh grid that define specific values for a specific area of the vector field. The element has a position that is the centroid of a triangle of nodes, with the nodes acting as bounds for the triangle. Elements can be viewed in figure 2.4 as the red dots in the center of every triangle.

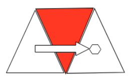
### Nodes

Nodes are points in the vector field that defines the boundaries of each element in the mesh. Nodes also contain data of all the elements that it is currently part of defining the bounds of. This is primarily used when mapping particles to the correct element for advection. Nodes can be viewed in figure 2.4 as the white dots which define the borders of the triangles.

## 2.7.2 Time steps

Time stepping is every advection or state update of a particle. E.g. If there is ocean current advection with the unit of measure of M/S, then every advection is one second of real time. The issue with this approach is performance as a simulation of 24 hours would require 86400 updates, which scales poorly with the number of particles. Therefore it is not unusual to increase the time step by

**Figure 2.5:** Particle skipping red element due to time step being too large.



multiple factors, multiplying the vector advection by whichever factor the time step is. A timestep of advection every 30 seconds approximates the amount of distance asomething has traveled within that time period. This also reduces the amount of updates in a day. E.g using a 30 second time step we go to 2880 which is 1/30th of having the timestep of 1 second.

There are issues with choosing too large time steps however as the larger the step the higher the probability of simulation inaccuracy. Consider a particle in a specific element advected by a timestep of 1 minute (60 seconds) which means the particle jumps across the neighbouring element to the element next to the immediate neighbour, missing the advection of the skipped element completely. This changes the trajectory of the particle compared to what it would have been with a time step of 1 second. A visualization of this can be seen in figure 2.5. This means setting the time step is often a practice of trial and error to see if the result is within an acceptable threshold of accuracy and performance. There are however ways to minimize the potential inaccuracy such as performing interpolation.

### 2.7.3 Interpolation

Interpolation in the context of trajectory modelling over a vector field is making discrete values continuous over a set area bound based upon a set amount of contributions. These contributions have weights attached to them and are taken into account as well as the distance from a arbitrary point within the

$$P_X = W_v1X_v1 + W_v2X_v2 + Wv3Xv3 \tag{2.9}$$

$$P_Y = W_v1Y_v1 + W_v2Y_v2 + Wv3Yv3 \tag{2.10}$$

$$W_v1 + W_v2 + W_v3 = 1 \tag{2.11}$$

**Figure 2.6:** System of equations to find the weights of V1, V2 and V3 in a barycentric coordinate system and an arbitrary point

$$Value_p = \frac{W_v1Value_v1 + W_v2Value_v2 + W_v3Value_v3}{W_v1 + W_v2 + W_v3} \tag{2.12}$$

**Figure 2.7:** Applying values to the calculated weights

bound and the contribution positions. An example of this could be a triangle with its vertexes having temperature values that acts as weights. If point A is at position (0,0) and point B is at position (10,0) with A having a temperature of 10 and B a temperature of 20 then each increment of position from a to b would increase the temperature by 1. With the mid point (5,0) having the temperature value of 15.

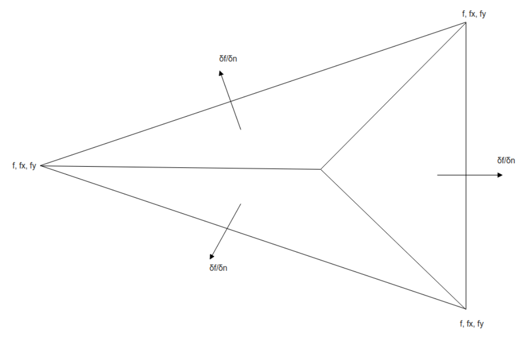
In the context of oceanography this makes intuitive sense as discrete values are not very accurate to real world behaviour within fluids. Where there is an expectation of continuous values not discrete. Thus to achieve a somewhat realistic simulation result one must interpolate the vector field.

### Barycentric Coordinates

Barycentric coordinates is a coordinate system where a point may be located within a simplex with a specific value. With each vertex of the simplex having some corresponding weight attached to it. A given points distance from each vertex in the simplex will then dictate the value of the point.

As can be seen in equation 2.6, barycentric coordinate systems sets up a system of equations which when solved will give the weights for each vertex and its effect on a potential tuple of values. The actual numerical value for an arbitrary point is then calculated by using the equation displayed at 2.7.

**Figure 2.8:** Clough-Tocher triangle



### Clough-Tocher Interpolation

Another interpolation approach is the Clough-Tocher implementation. The Clough-Tocher[6] implementation method is a finite element method. Clough-Tocher initiates with splitting its triangle in 3 smaller micro triangles, a bivirate polynomial is then define for each triangle which forms a bezier surface patch[7].

These patches are a cubic polynomial defined by twelve control points that acts as parameters. A visualization of this can be viewed in figure2.8. The function values of the f and first derivatives $f_x$ and $f_y$ at each original triangle vertex and the normal derivatives $\delta f$ at the mid point of each edge in the original triangle. The first derivatives at the vertices are estimated using the average slopes of the surrounding triangles. The triangle itself is partitioned into three micro-triangles along the seams defined by the centroid and the vertices of the original triangle.

The main selling point of clough tocher is the fact that is a local scheme i.e it only takes contributes from the vertexes of the triangle and no outside forces which makes it very performant. An innate weakness to this is the fact that it is not as accurate as other schemes which also takes non-local contributions, as the edges of the triangles with regards to neighbours. This means there is no guarantee of abiding by a completely continuous value range at these borders.

## 2.7.4 Ocean Drift

This section will go through the mathematics behind all the forces contributing to the advection when simulating particles in the ocean. This example will be used throughout the thesis as a basis of comparison between Particular and

OpenDrift, as these functions are directly ported from OpenDrifts base model physics methods.

### Advect Ocean Currents

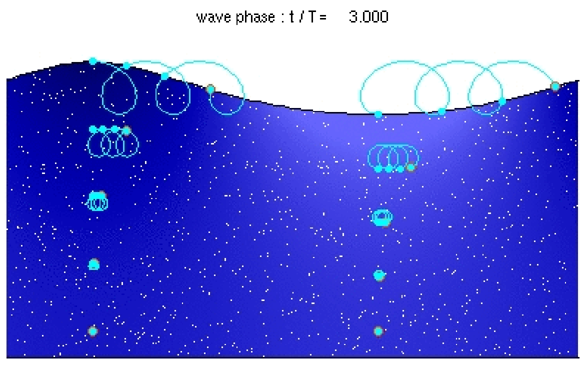
This advection function is a matter of advecting the particle based upon the ocean currents of the particles current position. This means advecting the particle position with either the Euler Method or Runga-Kutta which is described in detail previously in this chapter.

### Advect Wind

When calculating the wind advection of the particles, only particles near a set threshold of the surface should be affected by the wind, however only the differential of the wind and the ocean current. As intuitively particles on the bottom of the ocean is not affected by the winds advection force. The advection itself is done with the Runga Kutta method.

### Stokes Drift

**Figure 2.9:** Stokes drift wave after three periods. (By Kraaiennest - Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=37229011)



Stokes drift[8] velocity is the difference between the average Lagrangian flow velocity of a fluid parcel and the average Eulerian flow velocity of the fluid. Specifically used for movements of particles after waves. As a wave will take the particle in a almost circular motion but will usually displace the particle somewhat for each wave. This displacement is what stokes drift calculates and is reliant on data such as wave height and frequency. A visualization of stokes drift can be viewed with figure 2.9, with the teal points being the trajectories affected by stokes drift and the overall displacement for each wave.

#### Vertical Buoyancy

In fluid dynamics, an object is moving at its terminal velocity if its speed is constant due to the restraining force exerted by the fluid through which it is moving. When the buoyancy effects are taken into account, an object falling through a fluid under its own weight can reach a terminal velocity (settling velocity) if the net force acting on the object becomes zero. When the terminal velocity is reached the weight of the object is exactly balanced by the upward buoyancy force and drag force.

#### Vertical Advection

Moves the particle vertically based upon the environments velocity, much like the ocean currents except vertically instead of horizontally.

## 2.8 Related works

### 2.8.1 Opendrift

Opendrift[2] is a software package for modeling trajectories and fate of objects or substances in various environments such as oceans or the atmosphere. As such it promises a flexible, stochastic, robust, modular and fast enough trajectory model.

Opendrift is implemented in the Python programming language, and is seen as a framework and provides a core model which may be extended by the framework user using the classic OOP paradigm of inheritance.

Opendrift also comes with a lot of physics methods out of the box as well models for various environments pre-built like oceanographic drift of particles. As well as custom readers of files and writers which outputs to file and various visualizations of the trajectories simulated. A Lagrangian array is used to track particles, which is a multi-dimensional numpy[3] array describing the particle type and its properties.

2. https://opendrift.github.io/
3. https://numpy.org/

Particular and Opendrift both share similarities in the sense that both model trajectories. However implementation wise opendrift and particular does not share significant similarities with the exception of Particular using most of the trajectory mathematics with regards to oceanographic simulations from Opendrift.

### 2.8.2 Garnet

The garnet framework[4] is an ECS framework implemented in F#. Garnet functions as a simplified in-memory database and messaging system.

Garnets aims at being as fast as possible. In this case it means avoiding the .net garbage collector, (or GC), as much as possible as it may introduce spikes and inconsistent performance. Garnet avoids the Garbage Collector by using object pooling to pool objects within a container or on the stack. As well as avoiding closures.

Entities within garnet are simple 32 bit ids which are generated by re-use buffers to avoid the Garbage Collection. Components are any arbitrary data type that is associated with some entity. Components are stored in 64 element segments with a sparse mask, which provides a CPU-friendly iteration.

Systems are event subscriptions, with .Net event handlers being called every update which have been subscribed to a specific system. As these systems are just event subscriptions it is possible to compose smaller systems into larger systems.

Particular takes inspiration from the ECS architecture when attempting to model trajectories in a performant manner. While garnet and particular has significant differences in their respective implementations, (Which will be discussed in section 7.2). Particular did take inspiration from the garnet API which the user of the library would interact with to add/modify the entities and systems.

4. https://github.com/bcarruthers/garnet

# 3 Design

This chapter will give insight into the requirements, architecture and design choices of the components of Particular.
section 3.1 will detail the requirements that Particular needs to comply with, but in a functional and non-functional sense.
Section 3.2 will show the overarching architecture of Particular and define the system of components that make up Particular as a whole.
Section 3.3-3.7 will go through each defined component and detail their proposed function and how they aim to accomplish this.

## 3.1 Requirements

Particular has a fair amount of requirements it is expected to fulfill, both functional and non-functional.

### 3.1.1 Non-Functional Requirements

- Maintainability
  - Must have an easy to use interface to customize simulation
  - Should follow functional programming best practice to the best of its ability
  - Should prioritize type safety
- Robustness
  - Extensive logging if there is an error
  - Keep statistics over potential simulation inaccuracies
  - Gracious error handling
  - Debugging tools in visualization
- Performance
  - Should be able to scale in linear time if the vector field increases
  - Able to handle large quantities of particles (1 million+ concurrently)
- Flexibility
  - Should be able to run with any vector field
  - Choosing what data to read from disk to memory for the simulation should be determined by the user
  - Behaviour of particles should be completely determined by the user through functions
  - Choosing what particle trajectory data written to disk should be

determined by the user

– Must be able to handle simulations of arbitrary lengths

### 3.1.2 Functional Requirements

- Must be able to read NetCDF4 formatted files to memory.
- Must be able to write NetCDF4 formatted files to disk.
- Must have option to display graphs and visualizations of the particle trajectory simulation when complete.
- Particles has to be able to have any number of properties tied to them.
- Should produce the same output if the input is equivalent

## 3.2 Architecture

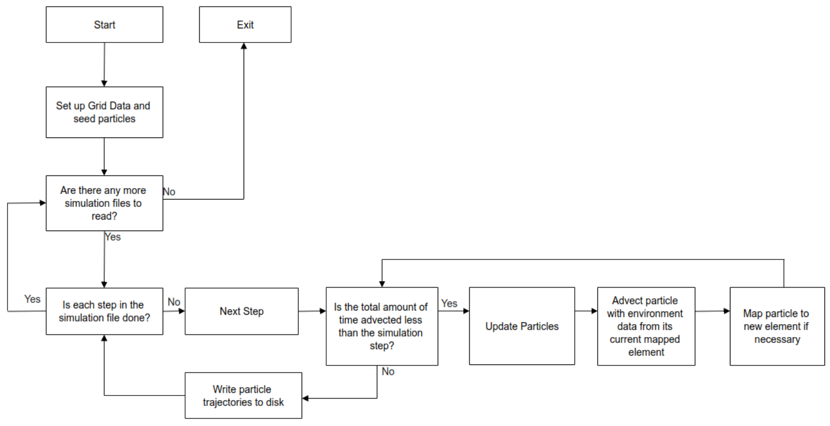
### 3.2.1 Overarching approach

The overarching approach to the design is to have a collection of particles with positions that are mapped onto a provided vector field. Each unit of the vector field will have its corresponding values within it that may affect the particles trajectory position in some way. These values have to be decided by the library user as well as how the particle advects and how a particle reacts to user defined events.

As the vector field will remain unchanged across the simulation, the elements and nodes can be read into memory. The elements can also keep a record of each elements nearest neighbours to aid with computation when particles are advected away from their current elements onto a new one and needs to be mapped to the new element correctly.

There is also a time stepping concept, with both advection of the particle approximated over a specific range of time. As well as another type of time step which is when the particle trajectories are saved and written to disk. An example of this is that particles updates every 10 minutes, but every hour the particle trajectories is appended to a NetCDF formatted file on disk. Thus a typical simulation cycle would look like the flow diagram at 3.1, with setting up the grid data and seeding particles before starting the outer loop of reading a list

**Figure 3.1:** Particular execution flow.


Start
Exit
Set up Grid Data and seed particles
Are there any more simulation files to read?
No
Yes
Is each step in the simulation file done?
Yes
No
Next Step
Is the total amount of time advected less than the simulation step?
Yes
No
Update Particles
Advect particle with environment data from its current mapped element
Map particle to new element if necessary
Write particle trajectories to disk

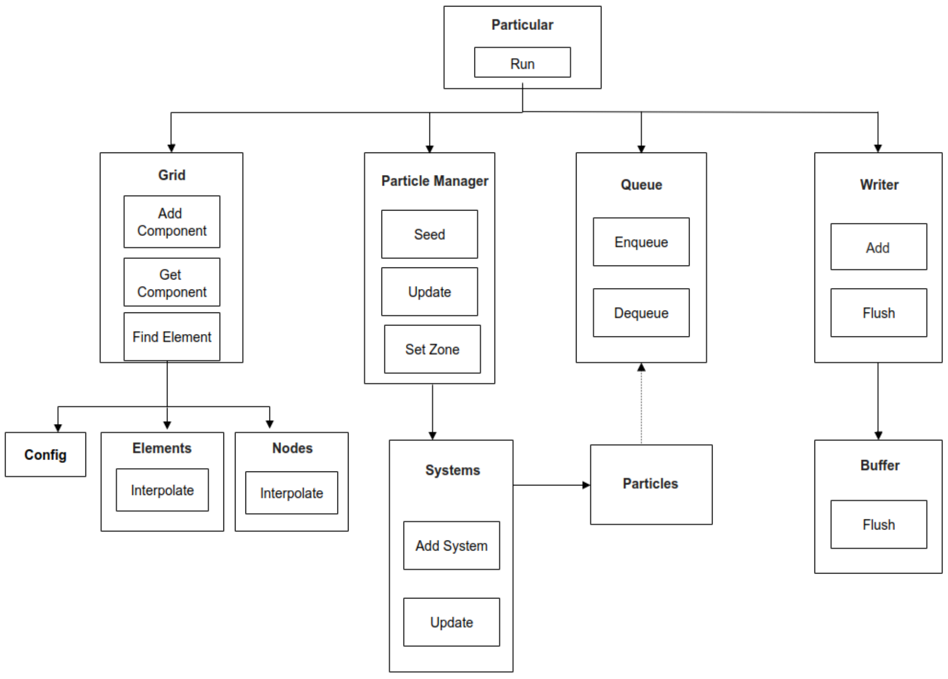
of environment files containing simulation data. Inside this loop another loop would go through every time step of the simulation file. (e.g if a simulation file has 24 hours of data then the time step is also 1 hour, which equals 24 iterations). Within each time step a third and final loop is ran which advects the particles with the given amount of time step advection. Where it also maps particles to their corresponding elements in the provided vector field. Importantly, the advection step cannot be larger than the overarching time step.

### 3.2.2 Splitting the system into components

While planning this particle simulation system, there was a fairly intuitive split of the system into a collection of components, in a layered structure. An illustration of this is provided with figure 3.2. As the illustration shows the data will flow only one way with the one exception of an asynchronous call from particles to the queue which will be described in detail in later sections of this chapter. This is an optimal data flow for the functional paradigm and overall this sort of data flow is almost always preferred for maintainability especially when parallel processing is introduced and it is almost forced upon you due to immutability.

The proceeding sections will describe each of these components, from the top-down, starting with the base component of particular.

**Figure 3.2:** Particular Component Relationships and layers



## 3.3 Particular

The particular component which is the component on the top layer is mostly responsible for composing all the lower tier components together in a simulation loop. The simulation loop would be very similar to the flow diagram provided 3.1 in both program logic and responsibilities. With particular calling all the proceeding components for each loop at various stages of the update cycle. This component also takes three lambda functions as input which will govern how environmental data is read and how trajectory data is written and how to handle user defined events. Which provides the flexibility required as this is a library which needs that generality. This will be described in further detail later on this chapter.

Particular also handles registering new systems to the simulation which is mathematical functions that are ran every update loop on a specifically designed sub set of the particles currently being simulated. This will be described in further detail in the systems sub section.

## 3.4 Grid

The grids main responsibility is handling the environment data, such as the vector field boundaries, positions of elements and nodes and their neighbours. To do this the grid must be able to parse specific formatted files which detail the positions of elements and nodes in a vector field.

#### Initialize Grid Data

When initializing grid data, we assume there to be vector positions for both elements and nodes, as well as some indexes which connects the given nodes to a given element or vice versa. From this information one can generate a nearest neighbour map and use this to generate a list of nearest neighbouring elements for each element which can be used to help particles map to the correct element during the simulation in an efficient manner.

The grid also handles the simulation data provided which sets values to each unit of the vector field. These values are governed by the user of the particular library thus needs to be generic, and easily accessible, thus the grid has two behaviours add and get component.

#### Add Component

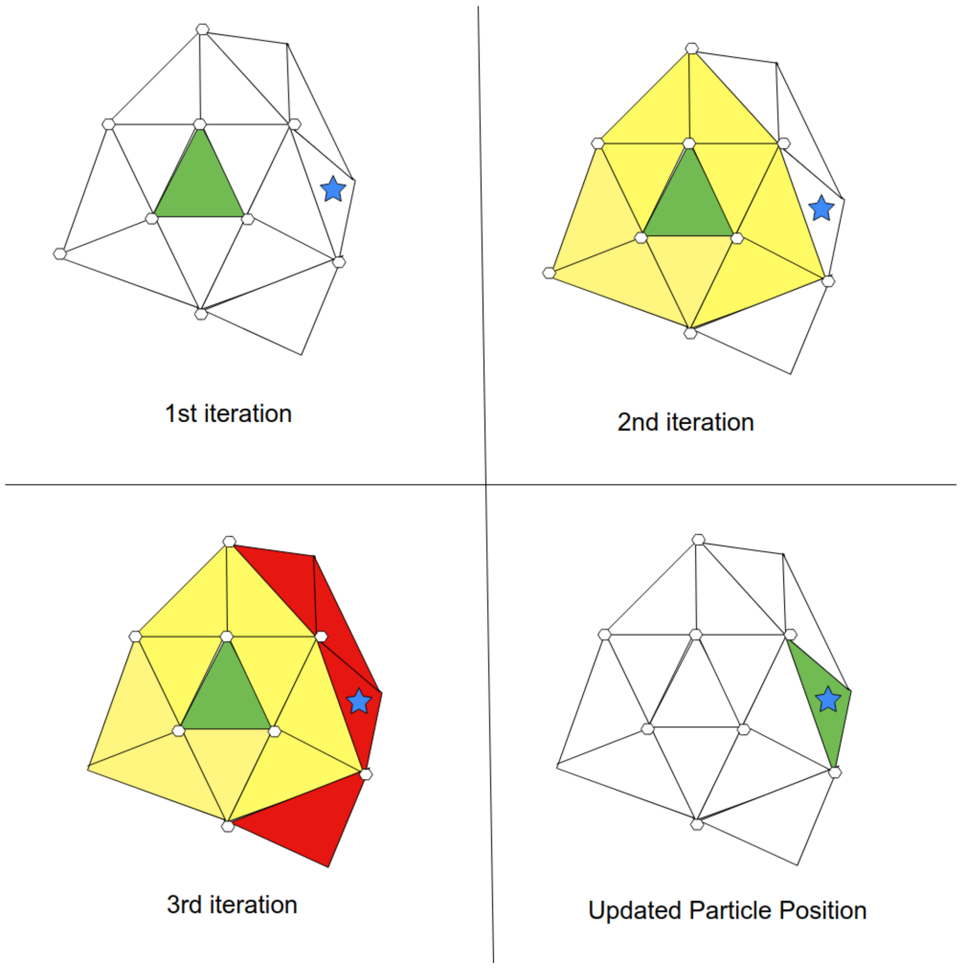
The add component behaviour will let the library users add simulation data from netcdf sources onto memory, which can later be fetched. This data may have different dimensions, 1d, 2d or 3d, with 1d being a constant array of data with the index being the node or element the value corresponds to. 2d having two indexes, one index corresponding to either the current time step or the depth, and the other index to the node or element the value corresponds to. With 3d having all three indexes previously mentioned. They are all however, assumed to be floating point numbers. So add component must be able to take any dimension of floating point arrays, store them in an efficient way to be fetched later when particles are being updated. The data itself is stored on either the nodes or elements depending on which the data is tied to. This will be described in further detail later on in this chapter.

#### Get Component

Get component is the opposite operation of the previously mentioned add component and will fetch a specific value from the grid of elements and nodes which can then be used by the application developer to influence the particles mapped onto that specific element in some user-defined way.

### Find Element

**Figure 3.3:** Search visualization, blue star represents the particle, green triangle the previously known position, yellow the nearest neighbouring elements, red the next nearest neighbours



After particle advection it is not uncommon for the particle to have moved away from its previous element to a new one. Thus the find element behaviour is necessary. Find Element's function is to search the nearby neighbours of the a chosen element to see if a particle is within any of those elements. A visualization of search can be seen at figure 3.3, which illustrates a breadth first search through the neighbours.

### 3.4.1 Elements and Nodes

Elements and nodes within the grid are structures that are designed to primarily hold data. Such as positions of each element and node, and a list of values for each element and node. These values may be added or fetched by the get and add component behaviours previously described. Some of these values may be set to be interpolated however.

#### Interpolate

The interpolate behaviour will interpolate a given value when fetched by applying the barycentric algorithm described in the background chapter section 2.8.3. The position of the vertices will be defined already by the elements and nodes. Thus all that is needed is a particle position within the triangle to complete the interpolation which will return a floating point number based on the weights of each vertices relative to the particles position.

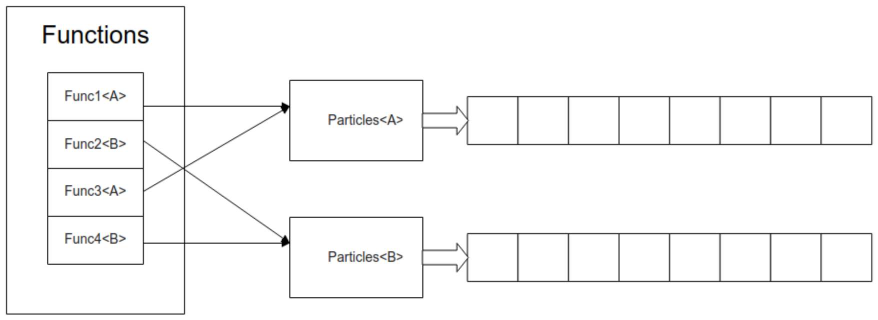
### 3.4.2 Config

Config is a read-only meta data structure that contains meta data of the simulation that is not directly applicable to the particle trajectory simulation, as well as other settings not found in the environmental files. Examples of this is, advect time step size which sets the amount of time delta approximated for each advection. Another example include wind drift depth, which sets a given threshold for how deep the particle has be submerged to avoid being affected by wind. A given requirement for these values is that they must be globally applied.

## 3.5 Particle Manager

The particle manager is the component chiefly responsible for advecting and keeping track of all particle trajectories during the simulation. In addition the particle manager is also responsible for advecting the collection of particles at every time step. Since the requirements of particular promises a degree of maintainability and flexibility, each particle may have specific characteristics not shared with other particles. There is also the possibility of a subset of the particles not being advected by a given update function. The particle manager should also support dynamic adds during the simulation run. To solve this an architecture was designed inspired by the ECS architecture described in the background chapter. With a given characteristic of a particle being

**Figure 3.4:** Particle Design: Visualization of a simulation of two particles types, A and B, with two different arrays of data. Also pictured a list of functions which are mapped onto the correct array based on the type in each update loop.



the determinant factor in which advection updates are applied to that set of particles. A diagram of the proposed architecture may be viewed at figure 3.4. With the systems stored in the particle manager being iterated through and applied to a corresponding particle set that matches the type of the function.

The aim of this being flexibility with supporting multiple types of particles simulated concurrently without sacrificing performance as these arrays ensure fast iterations due to contiguous memory which is CPU friendly. Another boon is increased maintainability as this splits the data and behaviour with systems containing behaviour and the particles containing the data. This makes it easy for the library user to simply register systems to a corresponding type and if there are any particles of such type currently being simulated these functions will be applied to the particles at every update call.

### Seed Particle

The particle managers seed particle behaviour will spawn a set of particles of a given number over a given 3 dimensional area set by the caller of the seed particle function.

### Update

When update is called the particle manager will iterate over all available systems(functions) and apply them to the particle trajectory data where applicable. This will be described in further detail in section 3.5.2.

#### Set Zone

After all particles have been advected the particle manager will iterate through every updated particle and check whether the particle is still at its registered element, or if its has advected beyond the borders of its previous element. If it has advected away the particle manager will call the find element behaviour described in section 3.4 to find the new element the advect particle is currently within and register it for the next update.

### 3.5.1 Particles

The particles themselves are a collection of pure data with no behaviour attached to them. All particles contain data such as position in longitude and latitude, the Z depth and the current element the particle is current within. In addition since particular has a requirement of flexibility and maintainability every particle also has some user provided properties. Not all particles in the simulation may have the same properties as these properties is what governs which functions are applied to that given particle at each advection update.

### 3.5.2 Systems

**Listing 3.1:** Update Function Signature

```
Particle <'a> -> Grid -> ('msg -> unit) -> Particle <'a>
```

The systems component represents the behaviour portion of the particle manager and contains a collection of lambda functions. All the functions that are applied to the aforementioned particles during an update. The functions themselves follows a set function signature which takes a particle, a reference to the grid and a dispatch function which asynchronously en-queues a messages into the queue which will be further described in section 3.6.

#### Add System

When adding a system it must obey the function signature displayed in listing 3.1, where a particle with a generic property annotated as $'a$ will be input with a reference to the grid which allows the particle to fetch values from their specific element with the get component behaviour as described in section 3.4. The dispatch function operates as a way to define your own events. An example would be if a particle hits the bottom of the ocean a message could be sent to the queue with a specific bottom hit message which spawns some smaller particles around the area of the impact or maybe removes the particle that hit the ocean floor. This is all up to the user and how he defines his message type and the reducer type which will be further described in section 3.6.

## 3.6 Queue

**Listing 3.2:** Queue psuedocode

```
(*Message type defined by the user*)
type Msg =
    | Sink of int

(*Emptying Queue after particle update*)
for each msg in queue:
    let reduce (msg : Msg) (pm : ParticleManager) =
        match msg with
        | Sink pId ->
            //Particle with the id pId has sunk

            //Do some operation on the particle manager
            //based on this message.

(* Function called in the particle manager update stage *)
let sink particle grid dispatch =
    //dispatch enqueues this message onto the queue which is emptied
    //after update of all particles.
    dispatch (Sunk particle.Id)
```

The queue is a component which is used to facilitate events that occur at the particle level which has ramifications for components above them. Example being a particle that emits some event which spawns more particles at its position. Then an event with a corresponding message is emitted during the advection asynchronously to the queue. After the particle manager is done updating, the queue is emptied of messages which are all passed into a given message reducer. The reducer takes the particle manager as a input and pattern matches all the messages and performs some operation on the particle manager based on the message. All these parts can be viewed with the psuedocode at listing 3.2 which attempts to help illustrate how the user may define the messages.

## 3.7 Writer

The writer component is responsible for transforming the trajectory data in memory to netcdf formatted data which is written to disk. The particle trajectory variables written to disk is decided by the user through passing in a function when creating the simulation. The writer also keeps a buffer of particle trajectory entries which gets written in bulk for performance reasons as conventional wisdom dictates that its better to rarely write a lot of data to disk as opposed to writing often small amounts of data. When writing the data to disk the positions should be recorded in geographic coordinate system (longitude and latitude), so the positions have to be projected from a Cartesian coordinate system to a geographic coordinate system.

### 3.7.1 Buffer

The buffer itself is a list of trajectory data for a single time step, so the total size of the buffer scales linearly with the amount of particles simulated. When the buffer is registered as full the entire buffer is flushed and written to disk in bulk.

# 4 Implementation

This chapter will detail the implementation of a prototype of Particular following the design layed out in the previous chapter.
Section 4.1 will detail the language choice and why it was made.
Section 4.2 will set up the domain specific language for this particular prototype
Section 4.3 will detail the implementation specifics for each individual component of particular.
Section 4.4 will show an example of how to set up an ocean drift simulation with particles using the described prototype.

## 4.1 Language choice

### 4.1.1 Choosing F#

The implementation of Particular was done in the F# programming language[1] using the .Net 5.0 compiler. F# was chosen primarily because its a functional programming language which is the main selling point of particular. F# a ML functional language which also defined as a functional first language as opposed to purely functional. This means F# supports both the functional language with more of an ML style as well as the OOP paradigm.

With being an ML language it means f# contains features such as static typing, algebraic data types, pattern matching, garbage collection, call by value and currying. In addition to this as mentioned, f# also supports the object oriented paradigm with classes and interfaces so it can interop with other first class citizens in the .net environment such as the C#[2] programming language.

#### What .Net Provides

The .net environment provides a great infrastructure with multiple libraries and frameworks which was used for handling simulation data. Which will be discussed in future sections in this chapter. This in addition to good concurrency support through asynchronous workflows and message passing. In addition there is also good parallelism suppport through the task parallel library, which abstracts the concepts of threads and thread management from the programmer as well as all the inherent problems with managing threads such as partition of work, scheduling threads from the thread pool and scaling the degree of concurrency dynamically to exploit all available processors in the most efficient way.

### 4.1.2 Potential drawbacks of functional programming and f#

While there are a lot of good things about functional programming and f#, there are also some inherent cons. Firstly functional programming is immutable, while great for correctness has the unfortunate drawback of performance as every update creates an entirely new object in memory. This is especially true in the .net environment which has a garbage collector which the programmer cannot control in any way, only influenced. While the garbage collector is

1. Fsharp.org
2. csharp.org

efficient and is often preferred rather than the programmer manually allocating memory, it is still a significant performance hit when the garbage collector is too busy. So there may be need to discard some immutability during the simulations to uphold performance and prevent the garbage collector from being too involved.

Another potential problem is the static type system, while ensuring correctness it is hard to create a generic library while taking full advantage of the type safety provided by static typing. Thus there may be need to use the generic object casting at times to fulfill the requirement of the library being generic. However ideally this should be done as rarely as possible, and the implementation should be built on a robust domain specific language.

## 4.2 Setting up the domain specific language

When setting up a domain specific language (DSL), it is important to mirror the types with the function of the program as a whole. If done correctly the types will document the code for the programmer, making the code easy to read and maintainable.

**Listing 4.1:** Particular DSL

```
type Interpolate = bool
type EnvironmentType =
    | Element
    | Node
type Dimension =
    | OneDim of Single []
    | TwoDim of Single [,]
    | ThreeDim of Single [,,]
type EnvironmentData = Dictionary<Type, Dimension*Interpolate>
type FilePath = string
type NNeighbours = int
type NodePoint = NodePoint of (Vector2*int)
type Seconds = int
type Hour    = int
type TimeStep = int
type Node = {
    Position : Vector2
}

type Element = {
    Vertices : (Node*Node*Node)
    Position : Vector2
    Neighbours : Element []
    Interpolation : (Vector2*Vector2*Vector2)
    Edge : bool
}
    member this.GetInterpolWeights :
        Vector2 -> EnvironmentType -> (float*float*float)

type Config = {
    AdvectTimeStep : Seconds
    TimeStep       : Hour
    SimFiles       : FilePath []
}

type Grid = {
    CurrentTs : TimeStep
    Elements : Element []
    Config   : Config
    EnvironmentData : EnvironmentData
}
    member this.Init : FilePath -> NNeighbours -> Grid
    member this.GetComponent<'a> : 'a -> single
    member this.AddComponent<'a> : string -> Interpolate -> unit
    member this.FindTriangle : int -> Vector2 -> Vector2 -> int -> C

[<Struct>]
type Particle<'a> = {
    Position : Vector2
    Z        : int
    Props    : 'a
}

type ParticleManager = {
    Particles : Dictionary<Type, obj>
}
    member this.Seed : Grid -> 'props -> int -> Area -> unit
    member this.Update : Grid -> (msg -> unit) -> unit
    member this.SetZone : Grid -> Vector2 -> Particle<'props>

type ISystem :
        -> (Particle<'props
        -> Grid
        -> ('msg -> unit)
```

```
          -> Particle <'props>)

    type Systems = {
        Systems : Dictionary<Type, obj>
    }
        member this.AddSystem<'a> : ISystem -> unit
        member this.GetSystem<'a> : ISystem<'a>
        member this.Run<'props..> : Systems<'props..> -> ParticleManager -> Part

    type Queue<'msg> = Queue of ConcurrentQueue<'msg>()

    Writer = Writer of (Dictionary<Type,obj> -> unit)

    type Simulation = {
        ParticleManager : ParticleManager
        Grid            : Grid
        Queue           : Queue
        Writer          : Writer
        Systems         : Systems
    }
        member this.Start : unit -> unit
```

The DSL showed at 4.1 is a psuedocode representation of the real DSL of particular in its entirety. In reality some of these objects are f# classes instead of f# records for performance reasons. However this illustrates the relationships between each component used in particular as well as the data contained in a terse way.

## 4.3 Implementing the Particular components

In design we went through all the components from the top down, to get a familiarity with their roles and relationships. In this chapter we will start from the opposite end, going bottom to the top. To ensure that all the functions have already been defined and described from the lower layer components as they are used by the upper layer components.

### 4.3.1 Implementing Nodes And Elements

```
[<Struct>]
type Node = {
    Position : Vector2
}
[<Struct>]
type Element = {
    Vertices : (Node*Node*Node)
    Position : Vector2
    Neighbours : Element []
    Interpolation : (Vector2*Vector2*Vector2)
    Edge : bool
}
    member this.GetInterpolWeights :
        Vector2 -> EnvironmentType -> (float*float*float)
```

The nodes and elements types are pure f# record data structures that are responsible for keeping track of the vector field characteristics. They are defined as a .net struct value type to lessen the amount of references to increase the locality of reference when iterating through a collection of elements. Both node and element have a position in the world. That is the extent of the node data however. The element data type is a more complex structure as it has a list of neighbouring element vertices, the positions of its own vertices and the position of the elements that create its interpolation zone. The edge flag is set if any of the vertices surrounding the element only has 3 or less elements surrounding it, meaning this element must be on the edge.

#### Getting Interpolation Weights

When fetching a value for use in the particle advection that has to be interpolated, this function is called. The purpose of this function is taking in a vector position within the element where the particle in question is currently residing as input, then returning the weights of each vertex. There is a complication however, as values can be tied to both elements and nodes. This means that there are two different triangles that are interpolated based on whether the value is an element value or a node value. Thus when calling to get the interpolation weights a union type signifying whether the value is a element or node value is necessary.

### 4.3.2 Implementing the Grid

```
type Grid = {
    CurrentTs : TimeStep
    Elements : Element []
    Config   : Config
    EnvironmentData : EnvironmentData
}
member this.Init : FilePath -> NNeighbours -> Grid
member this.Next : FilePath -> (Grid -> Grid) -> Grid -> Grid
member this.GetComponent<'a> : 'a -> single
member this.AddComponent<'a> : string -> Interpolate -> unit
member this.FindTriangle : int -> Vector2 -> Vector2 -> int -> Option<int>
```

The grid is a data structure in the form of a f# record that reads grid data and transforms it to a use-able vector field in memory which can add and fetch values corresponding to a given element.

## Init

The initializing function takes a file path, the number of neighbours each element should store and returns a new grid with all the nodes and elements within a specifically formatted grid file.

**Listing 4.2:** Grid data file format example

```
Node Number = 3
Element Number = 1
1 1 2 3
1 20.54 18.04
2 18.20 24.52
3 15.13 18.45
```

When parsing a grid file it is a specific format for the grid file is expected. Respectively a header of the total amount of nodes and elements. Then a list of element indexes and the node indexes that surrounds the element. Then finally a list of node indexes and their x and y positions. A small sample can be viewed in listing 4.2. This data can then be used to find the nodes surrounding an element and the elements surrounding a node.

With this info it is possible to traverse the vector field from a given triangle to another triangle. Which is a necessary feature for when you need to map a particle to another element after advection. However doing this every time you need to map a particle to another element when advecting would be costly, especially if the advection places the particle further away than the current elements closest neighbours. This would then require a recursive breadth first search, which has an exponential search time for every "ring" it has to search. This will be evaluated and investigated in more detail in the evaluation chapter 6.3.

Taking all this into account a kd-tree solution was implemented, where every element position was inserted as a key with the element index as the value. After all elements had been added to the tree, the tree was queried for the n nearest neighbours for every element position an array of indexes would be returned signifying these elements were the n closest to a given element. This would result in a nested array of vectors which would be applied to the existing element array to set the neighbours array for all elements. This would result in a O(1) lookup time whenever one needed to find the closest neighbours to a given element.

The kdtree used originated from a library named "kdtree"[3] created by github user "codeandcats" and was implemented in C#, which is obviously mutable. However as this is ran sequentially and only ran once before the simulation loop even begins it is not considered an issue.

### Add Component

Adding a component is called when the grid needs data from an environmental file read to memory. To read the environmental files in netcdf format the SDSLite library was used, which is a Microsoft developed .net library for handling scientific data.

SDSLite reads data lazily from disk to memory, which is sometimes considered good behaviour. Particular however will be expected to run on machines with massive amounts of memory thus should always sacrifice space complexity for time or I/O complexity if possible.

The add component function takes a generic type annotation and the environmental type signifying if its an element or node as well as the variable name stored in the netcdf file on disk and a boolean as an interpolation flag used when data is fetched. Once this is done it gets the type of the type annotation and inserts it into the environment data dictionary structure with the type as key and the array of floating point numbers read from disk as values.

There is a complexity that arises from this however, as the data read from disk can come in three different formats. The data can be constant, i.e one dimensional so the index just points to an element or node. It can be two dimensional where one index points to either the depth of the vector field or the time step whereas the other index points to the element or node index. It can also be three dimensional which has a time step index, z depth index and a node/element index. To solve this a union type was created with cases for

3. https://github.com/codeandcats/KdTree

both one dimensional, two dimensional and three dimensional single arrays. Thus the dictionary stores a union type instead of a specific single array type, this upholds the type safety with for the price of a bit more branching in the code as a pattern match is required when fetching the component.

### Get Component

The get component function takes a type annotation of the type that the user wants fetched, uses it as a key in the environmental data dictionary to fetch the union environmental data type. In addition a particle is also passed in as get component is a function that is called during particle advection.

```
match envData with
| OneDim s -> s.[particle.CurrentElem]
| TwoDim (s,tag) ->
    match tag with
    | Depth -> s.[int particle.Z, particle.CurrentElem]
    | Time -> s.[this.CurrentTs, particle.CurrentElem]
| ThreeDim s ->  s.[this.CurrentTs, int particle.Z, particle.CurrentElem]
```

The data type is pattern matched and indexed accordingly to the dimensions of the environmental data. An example of this can be seen in listing 4.3.2, where each case is indexed into based on the particle passed in.

There is a complexity to this however, if the value is marked as interpolated it must be interpolated by the element the provided particle is currently on. Thus get component will call get interpolation weights with the particles position which has been described in subsection 4.3.1. This will return a tuple of three weights that the corresponding values are multiplied against which results in a unique value based on the position of the particle in relation to the distance to each vertex in the interpolated zone.

### Next

**Listing 4.3:** Example of grid binding function called in Grid.Next

```
let bindingFunc (grid : Grid) =
    AddComponent<U, single[,,]> Element "u" true grid
    |> AddComponent<V, single[,,]> Element "v" true
```

The next function is responsible for reading environmental netcdf files. Once a netcdf file is read, a binding function that is user defined is applied to the grid. An example of such an binding function defined by the user can be viewed with listing 4.3. This function is then ran after every new environmental file is read during the simulation loop which results in reading the environmental data to memory from the new file, essentially resetting the grid with new values.

### Find Element

The find element function is responsible for finding a element that the provided position is currently within, once that element has been found the index of that element should be returned. If no element is found a none type should be returned signifying that there is no element that the current position is within which usually means the vector is out of bounds of the vector field.

**Listing 4.4:** FindElement psuedocode

```
let rec findElement =
    (vel : Vector2)
    (current : int)
    (steps : int)
    (currentElem : int)
    (currentPos : Vector2) =

    if currentPos + vel is in triangles.[currentElem] then

        if the steps are equal to 1 we can exit with the
        current elem.

        if not we recursively call innerFn again with a
        update position of the velocity

    else

    for every triangles.[currentElem].Neighbours
    try to find an element that (currentPos + vel) is within.

    if none can be found and we are on a registered edge return none.

    if none can be found otherwise split the velocity in two and double
    the steps.

    if a match is found and the steps is equal to one return
    currentElem

    if a match is found and the steps is not equal to one
        recursively call findElement with a decremented step
        and a incremented (currentPos + vel).
findElement originalVel 0 1 originElem oldPos
```

Psuedocode is provided with listing 4.4, where the general idea is to first search in the current element to see if the particle is still within it, if it is not search through the neighbours to try to find an element the particle is now currently within. Usually this is good enough if the time step advection is not too large or the elements are not too small. However there can be situations where a particle has advected so far that it is beyond all the nearby neighbours of the previously recorded particle element position. In this case we have to recursively slash the velocity provided in half, and double the steps remaining to complete the velocity advection, until a neighbour or the current element is a match. Once a match is found you cannot quit yet as there is some velocity to still add which is kept track of by the steps remaining value. So the particles current element is temporarily updated to the matched neighbour before adding the remaining velocity to its position repeating this process until the entire velocity vector has been advected which is when the remaining steps value is equal to 1.

### 4.3.3 Particles

```
[<Struct>]
type Particle<'a> = {
    Position    : Vector2
    Z           : int
    CurrentElem : int
    Props       : 'a
}
```

The particles are defined as a .net record value type for locality of reference reasons as it is more CPU friendly to iterate through which happens frequently during a simulation. The particle itself has some general values all particles possess such as position in the x,y and z axis. As well as a index to the current element the particle was previously registered as being within.

In addition to this there are some properties that are passed in as a generic annotated type by the library user which differentiates particle types from one another. Particles themselves have not any behaviour or method attached to them, this is by design as we want to uncouple data from functions as much as possible for maintainability as well as performance reasons. Which will be made clear in the evaluation section 6.5.

### 4.3.4 Particle Manager

```
type ParticleManager = {
    Particles : Dictionary<Type, obj>
}
member this.Seed : Grid -> 'props -> int -> Area -> unit
member this.Update<'prop> : ISystem -> Grid -> (msg -> unit) -> unit
member this.MapGrid : Grid -> Vector2 -> Particle<'props>
```

The particle manager is represented as a .net class and manages all the particles currently being simulated, it keeps particle arrays in a dictionary based on the props type of the particle defined by the user with a generic annotation. This ensures a CPU friendly iteration even if there any number of different types of particles being simulated concurrently. The particles themselves are upcast to .net objects, the reasoning for this is that the f# type system is limited in the regards of having differing types in the same collection. Thus upcast it to the root object is often the most straightforward solution. The downside is that we are essentially turning off the type system and we open ourselves up to an unsafe operation which may throw an exception as we have to down cast the object back to its proper type when it comes time to fetch the particles.

#### Seed Particles

The seed particle function appends an array of newly spawned particles with a specific property to other existing particles with the same property in the particle manager dictionary. These particles have a starting point defined by the area type where particles are psuedo randomly spawned within this defined 3 dimensional area at a given time frame interval.

#### MapGrid

Map Grid is called after the main update advection is done for each particle. Map grid mainly calls the find triangle function described in section 4.3.2. Which will return an updated index for which signifies which element the particle is now on.

#### Update

The update function takes a particle array with a specific property from the dictionary and dynamically down casts it to the correct particle type. This is technically an unsafe operation as down casting happens at run time as there is no way for the compiler to anticipate whether the down cast is legitimate or not. This has an effect of making it possible for the code to throw an exception as well as a performance overhead as the compiler is now forced to check for every down cast whether the down cast is legitimate or not.

After the type is resolved the particle array that corresponds to the type is fetched. Then the array is iterated through in parallel, with the parallelization being done by the .net thread scheduler which attempts to schedule the threads in the most optimal manner. The update itself is governed by the ISystem type which has the function signature of $Particle <' props > - > Grid- > (msg- > unit)- > Particle <' props >$, which is mapped to every particle in the update.

### 4.3.5 Systems

```
type Systems = {
    Systems : Dictionary<Type, obj>
}
member this.AddSystem<'a> : ISystem -> unit
member this.GetSystem<'a> : ISystem<'a>
member this.Run<'props..> :
    Systems<'props..>
    -> ParticleManager
    -> ParticleManager
```

The systems component is represented as a .net class and contains all the functions that govern the advection behaviour of the particles stored in the particle manager. The functions themselves are stored in a dictionary, with the particle property type they are supposed to apply to at every update, as the key. The value is then the corresponding function cast to an .net object.

#### Add System

When adding a system the type annotation provided will decide the key type for the dictionary and the function itself is cast to an object before being added to the dictionary.

#### Get System

When getting the system the dictionary is read for the provided type annotation to fetch the corresponding value. The value is then down cast from an .net object type back to the correct ISystem type.

### Run

The run function takes a invariant amount of type annotations which is provided by the library user. This is accomplished by having the systems be a class and the run function be a static method which has several overrides with varying amounts of type annotations available. This creates a sort of polymorphism replacement with type safety. The draw back is that the code must be repeated for how ever many type annotations you wish to support. (Particular supports up to 8). Then for however many type annotations provided. The update function from the particle manager is called for every type annotation which then in effect updates all the particles with the type annotations provided in the run function.

## 4.3.6 Implementing the Queue

The queue component had some differing approaches. Given the queue receives messages in parallel, one requirement of the queue was that it had to be thread safe. Another requirement was performance which a queue should be able to comply with as queues tend to have constant time complexity for both en-queuing and de-queuing. However when running in parallel with mutexes there is some overhead involved due to contention of resources.

The chosen approach was to use the built-in .NET concurrent queue object developed by Microsoft. This object was originally developed in C# and is thus mutable by default. The internal logic uses mutex locks instead of immutability to achieve thread safety. The main reasoning for choosing this approach was due to the performance which was significantly better than other functional approaches, thus it was chosen. A direct comparison between the differing approaches can be viewed in the evaluation chapter section 6.6.

### Reducers and Message types

As mentioned in the design chapter, the queues main function is to receive emitted messages from particles when they update and apply them to the simulation after the particle advection is finished for a specific time step.

As the messages emitted and the reaction to these messages had to be completely customizable by the user. A solution heavily inspired from Elmish architecture was chosen where the user passes in a reducer function.

The reducer function takes in a message type with an optional value bound to it and a model value which gets transforms based on the message passed in. The transformable model in this case is the particle manager object nd the message type is defined by the user as a type parameter when declaring the simulation.

#### Enqueue

The en-queue behaviour is the behaviour which allows particles to emit messages to the queue. By passing in a partially applied en-queue function into the particle update function the particles may pass in a message to finish the en-queue function when an event occurs which will place the message inside the queue.

**Listing 4.5:** Currying example with F#

```
let twoParamFunc a b =
    a + b
let twoParamCompilerView a =
    let innerFn b =
        a + b
    innerFn
```

Partial application is something innate to functional programming as all functions within F# are actually single parameter functions. When creating a multi-parameter function what occurs is the function is curried into various steps, returning partially applied functions based on the amount of parameters passed in to keep the constraint of single parameter functions. An example of this can be viewed in listing 4.5.

#### Empty Queue

When emptying the queue, the built in dequeue function is applied to empty the concurrent .NET queue and each message dequeued in passed into the given reducer function with the particle manager, transforming the particle manager object based upon the message passed in. An example of this can be seen in the ocean drift implementation section further down the chapter.

### 4.3.7 Implementing the Writer

The writer components main function is after every time step is done the to receive particle data contained within the particle manager that gets flushed to the writer. The writer is then tasked of writing said data to disk.

### Setting up a buffer

As writing to disk takes a large amount of the total execution as can be observed in the evaluation chapter 6.2. Efforts to minimize this was developed. An idea of creating a buffer was attempted as it would mean less, but larger, writes to disk which is expected to give a boost to execution time.

In addition to this asynchronous workflows were applied which would let the disk writes happen in parallel with the next particle advection updates. However to take advantage of this, a double buffering scheme was developed which would let one buffer write to disk concurrently while another buffer was reading incoming trajectory data. This would in theory let buffers write continuously without a need to stop as buffers are full. There is some danger here though as the writes have to happen in order as the trajectory data is dependant on order. The evalation of the effect of this can be seen in chapter 6.4.

## 4.3.8 Implementing Simulation

The simulation component pictured in figure 4.1 is the top layered component which is also illustrated with figure 3.2, which means it is composed of all the other components of the lower layers. The simulation components main responsibility is initializing every component as it is responsible for linking lambda functions defining behavior of the simulation to the underlying components. In addition to creating the run loop where the simulation is actually executed.

### Intializing Simulation

When initializing simulation, the class takes in three arguments, a config type. An lambda function that describes which variables in the netcdf file will be read. This lambda is described in the next function in section 4.3.4 which is where it gets passed into. In addition there is a reducer function that defines user created events which are dispatched when advecting particles. There is also a writer lambda which describes the particle types that will be written to disk.

**Listing 4.6:** Sample of user adding two environmental variables

```
type ParticleA = int
type ParticleB = float
type U = U
type V = V

let envData (grid : Grid) =
    Grid.AddComponent<U> "u" grid
    |> Grid.AddComponent<V> "v"

let config =
    Config.New 300 3600

Simulation(config, envData, Writer<ParticleA, ParticleB>())
```

A sample of how a user would set up a simulation object can be seen in listing 4.6.

### Run

**Listing 4.7:** Simulation run loop psuedocode

```
foreach file in simFiles:
    Grid <- ReadEnvironmentData file
    for ts in 0..TimeStep:
        for advectTs in 0..(AdvectTS/TimeStep):
            Systems.Run ParticleManager Grid Queue.Enqueue
            Particles <- Queue.Empty Particles
        Writer.Write Particles
        Grid.CurrentTs += 1
```

The run function is responsible for running the simulation. Psuedocode of the run function is provided in listing 4.7. Where each file path of simulation files provided is iterated over. Each iteration of this will iterate through the given time steps where particles are advected for that period of time. E.g having a time step of 1 hour and a advection time step of 10 minutes will mean (600/3600) = 6 iterations. After these advection iterations the particle trajectory is appended to the netcdf file to disk and the time step of the grid is incremented.

## 4.4 Ocean drift implementation

In this section a overview of how Particular could be applied to model a simulation of particles drifting in an ocean will be described.

### 4.4.1 Creating environment readers

```
type U = U
type V = V
type UW = UW
type VW = VW
type WW = WW
let reader (grid : Grid) =
    |> AddComponent<U> Element "u" true
    |> AddComponent<V> Element "v" true
    |> AddComponent<UW> Element "uw" true
    |> AddComponent<VW> Element "vw" true
    |> AddComponent<WW> Element "ww" true
```

First step to model is to get the environmental data from a provided netcdf formatted simulation file. In this case of ocean drift we care about two main factors; ocean currents and wind speed. Thus we can apply the add and get component API described previously this chapter as can be seen in listing 3.10. First defining the types for the environment data then creating a function with a (Grid -> Grid) signature that gets passed into the simulation model and is invoked every time a new simulation file is opened.

### 4.4.2 Particle Lambdas

Creating the particle lambda is the next step after defining the environment data. First we must create some functions which can be chained together to simulate particles drifting in the ocean. The theory behind these functions are described in section 2.7.4, this will be a description the implementation of these concepts in a programming sense using Particular and F#.

#### Advecting ocean horizonally and vertically

Psuedocode of the advect ocean function can be seen in listing 3.7, the reader fetches the ocean currents from the environment and indexes into the current time step as well as the depth of the current particle and the element the particle is currently on. The particle is then advected by a euler method algorithm. Advecting the particle vertically follows the same procedure with different advect velocity values.

### Advecting Wind

```
advectWind (particle : Particle) grid dispatch =
    windVelX = grid.GetComponent<WV>(particle)
    windVelY = grid.GetComponent<VV>(particle)
    currentVelX = grid.GetComponent<U> particle
    currentVelY = grid.GetComponent<V> particle
    particle.Pos.X += (windVelX − currentVelX)
    particle.Pos.Y += (windVelY − currentVelY)
```

When advecting the wind each particle over a given depth in the ocean gets advected by the differential of the wind drift and ocean current. Psuedocode for implementing such a function can be seen in listing 3.11.

### Stokes Drift

```
calculateStokesDrift (stokesVel, waveHeight, wavePeriod, Z) =
    surfaceSpeed = sqrt ( stokesVel.X^2 + stokesVel.Y^2)
    fm02 = 1.0 / wavePeriod
    totalTransport = 2 PI /16 fm02waveHeight ^
    2k = surfaceSpeed / 2 totalTransport
    stokesSpeed = surfaceSpeed ( k z )^2
    stokes.X = stokesSpeedstokes .X/ surfaceSpeed
    stokes.Y = stokesSpeedstokes .Y/ surfaceSpeed
    return stokes, stokesSpeed

stokes_drift ( particle ) =
    waveHeight = getWaveHeight ()
    wavePeriod = getWavePeriod ()
    stokesVel = getStokesVelocity ()
    stokesVel, stokesSpeed = calculateStokesDrift
(stokesVel ,waveHeight ,wavePeriod, particle.Z)
    return stokesVel
```

When implementing the stokes drift, which is not mathematically trivial, the math itself was ported directly from OpenDrift to Particular. Psuedocode of the stokes drift calculation can be viewed in listing 3.12.

#### Advecting Buoyancy

```
calculateBuoyancy particle =
    if particle.Z < 0 then
        particle.Z += particle.TerminalVelocity
        if particle.Z = 0 then
            dispatch (BottomHit particle)
        return particle
    else
        return particle
```

When calculating the buoyancy if the particle is not at the bottom of the sea the particle is advected with its given terminal velocity as well as a side effect is emitted if the particle hits the bottom after advecting emitting a message to the queue component to handle the particle hitting the bottom.

### 4.4.3 Setting the queue behaviour

```
bottomHit position particles =
    (*create particles*)
    newParticles = createParticles position 20
    Array.append newParticles particles

coastHit particle oldPos particles =
    particles
    |> Array.find particle
    |> { particle with Position = oldPos }

reducer (msg : Msg) (particles : Particles) =
    match msg with
    | BottomHit position ->
        bottomHit position particles
    | CoastHit (particles, oldPos) ->
        coastHit particle oldPos particles

type Msg =
    | BottomHit of Position
    | CoastHit of Particle*OldPosition
```

Finally, the queue is set by ocean drift to handle two specific scenarios; when particles hit the bottom of the ocean and the coast. When particles hit the bottom of the ocean floor it may spawn a multitude of smaller particles around the area of the impact. When particles hit the coast the particle is "thrown back" in the water of to its previous position and tries to advect again with different environmental characteristics as the time step has increased.

To set this up a message enum type had to be defined, with two respective cases. In listing 3.15, an example of a reducer function and message type has been implemented. The program flow in this case after each particle has updated all the messages received in the queue will be empties and piped through the reducer function provided which updates the particles object.

To dispatch messages the user can introduce them as side effects in the particle update lambda, as a dispatch function will be provided which is capable of emitting these events to the queue as they happen. In the case of ocean drift a natural place to introduce these side effects would be in the buoyancy function when hitting the sea floor and when advecting the ocean currents to dispatch a coast hit if no elements are found.

### 4.4.4 Putting it all together

**Listing 4.8:** Oceandrift example

```
(*Set amount of particles seeded *)
let n = 10000
(* getting reader data from disk *)
type U = U
type V = V
type UW = UW
type VW = VW
type WW = WW
let reader (grid : Grid) =
    |> AddComponent<U> Element "u" true
    |> AddComponent<V> Element "v" true
    |> AddComponent<UW> Element "uw" true
    |> AddComponent<VW> Element "vw" true
    |> AddComponent<WW> Element "ww" true

(*Defining the messages*)
type Msg =
    | BottomHit of Position
    | CoastHit of Particle*OldPosition

(*defining the reducer as seen in previous listing*)
let reducer = reducer

//setting up the unique particle property
```

```
type ParticleProp = TerminalVelocity of float

//setting up the config with a time step of 300 seconds
//and a 1 hour time delta.
let config = Config.New Seconds 300 Hours 1

(*defining the system *)
let system =
    advectOceanCurrents
    |> advectWind
    |> advectStokesDrift
    |> advectBuoyancy
    |> advectVertically
    |> ISystem

SimulationModel<Msg>(
    config,
    reader,
    Writer<Prop, B>()
)
|> fun x -> //adding the system
    x.AddSystem<ParticleProp> (system)
|> fun x -> //seeding N particles
    x.SeedParticles<ParticleProp> { TerminalVelocity = 0.2 } n
|> fun x -> //starting the simulation
    x.Start (Systems.Run<ParticleProp>)
```

Taking all the mentioned implementations the code to define a simulation of particles drifting in the ocean would look like the psuedo code in listing 4.8. This illustrates the conciseness and flexibility of the library where the code the library user has to provide is strictly their own mathematical functions and data/behaviour that only pertains to the specifics of their simulation. All the general boiler plate is abstracted away and the user will simply get trajectory data written to disk in a performant manner without really needing to know how the system is designed.

# 5 Alternate Implementations

This chapter will detail some alternate implementations of various components of Particular and their perceived pros and cons. This will serve as preparation and background for the evaluation chapter which will evaluate these mentioned approaches with the aforementioned implementation.
Section 5.1 will go through alternate approaches regarding how to iterate and store particles
Section 5.2 will go through alternate queue approaches
Section 5.3 will go through alternate ways to index and map neighbouring elements to each other
Section 5.4 will go through alternate ways to store trajectory data in buffers for writing

## 5.1 Alternate Approaches: Particle Update

### 5.1.1 Individual Lambda Approach

The current implementation groups particle types into various groups of arrays with specific functions mapped onto these arrays for each time step. The individual lambda approach can be thought of as a simpler approach to this problem where the update lambda is put into the particle update itself thus there is only one array with all the particle data.

The main advantage of this approach is that it is possible to make this completely type safe, as the lambda will have a specific function signature defined by a F# record type which is called for each individual particle when updated. With a custom defined union property type that contains the custom data for a given lambda.

### 5.1.2 Grid-Based Particle Approach

Unlike the aforementioned other approaches, the grid-based approach lets each element in the grid keep a list of particles currently inhabiting it in addition to the usual data tied to an element in the grid. Particular would then be structured as a large 2D array with the elements populating one dimension and the particles populating the second dimension. The expected result of this is a slightly better locality of reference as all the data for updating the particle trajectory would be contained within 1 index of the array. With the expected drawback being updates not as easily ran in parallel in an efficient manner as most elements would probably have no particles stored within them.

### 5.1.3 Using the Garnet framework

The final alternate approach was to use a ready made Entity Component System library named Garnet. As described in the background chapter, garnet is a Entity Component System architecture framework designed for game development.

The main advantage of this is that Garnet promises a specific set of functionality which fits the needs of Particular quite well and would execute the simulation in a relatively performant manner. While also providing a very maintainable API for the library user.

An innate issue with the garnet approach is for the most part its generality. While generality is usually a strength it can also be a weakness. In this case as game development usually requires more data types and the data itself changes types repeatedly Garnet is more tailored to entities adding and removing components constantly. Particular on the other hand would usually not have these behavioural properties ranked as highly, as particles are not expected to add/remove components at run-time that often. Thus a sparse-set implementation, (which garnet uses), has a trade-off of expected worse iteration performance vs better performance when changing entity types compared to a non sparse-set approach. Which is not ideal, garnet also does not support multi-threading out of the box which is also a potential issue.

## 5.2 Alternate Approach: Queues

### 5.2.1 Functional Queue

The chosen queue implementation as mentioned in the previous chapter was the built-in Microsoft Concurrent Queue. This is a mutable data structure which promises thread safety with the use of mutexes, not immutability. As such this locking nature naturally incurs a degree of overhead when ran in parallel as threads compete for a specific resource, a lock in this case which may cause stalling. However in the case of this system and the context the queue is used, there is no heavy operations tied to the mutex as the queue simply adds a message and exits as such the resource contention should not be too heavy of a penalty.

However a lock free approach was also designed and implemented from the bottom up in F#. The reasoning for making a functional queue type with a tuple of lists was to achieve amortized O(1) access time. When en-queuing append the message to the front list which is a O(1) time complexity operation as we are simply appending to the head of a linked list. When de-queuing if the back list is empty. Take the front list, reverse it and place it in the back list and return the head. if not empty simply fetch the next item in the back list until the back list is empty and repeat the process. The main reasoning for going with this approach is the functional nature of it, which promises type safety and thread safety innately. However a issue which became apparent was one of performance which will be further described in section 6.7.

## 5.3 Alternate Approach: Indexing neighbouring elements

### 5.3.1 Breadth first approach

**Listing 5.1:** Breadth first search psuedocode

```
let rec innerFn (current : int) (currentElem : CurrentElem) (n : int) =
    let nodeIdxs = GetSurrounding Nodes currentElem
    nodeIdxs
    |> Array.collect (fun nodeIdx ->
        let ix = GetSurroundingElements nodeIdx
        if (current + 1) = n then
            ix
        else
            ix
            |> Array.collect (fun index ->
                innerFn (current + 1) index
            )
    )
innerFn 0  elemIdx
```

A breadth first approach was another approach attempted when indexing element neighbours to corresponding elements in the element array. Psuedocode of the algorithm can be viewed in listing 5.1. Where each node surrounding a given element was fetched and for each of these nodes the elements surrounding them would be fetched and collected. This would be recursively repeated n times depending on how many "rings" of neighbours one wanted to store as neighbours for each element.

## 5.4 Alternate Approach: Write Buffers

### 5.4.1 Single Buffer

A single buffer approach was attempted as well as the current implementations double buffer approach described in chapter 4.3.7. The single buffer was a array of trajectory data which was continuously added to until it was registered as full, then it would write the entire buffer in bulk to disk. This was faster than writing every time step, the comparison to the double buffer approach may be seen at section 6.4.2.

# 6 Evaluation

This chapter will detail the evaluation of Particular and its various components compared to alternate approaches, as well as direct comparisons to the Opendrift framework.
Section 6.1 will detail the experimental setup in regards to hardware and software used to conduct the evaluations. In addition further explanation on how the benchmarking library for .net functions and how to interpret its result will be explained in section 6.1.1.
Section 6.2 will show the profiling of particular as a whole to display which parts of the system is the most expensive performance wise. Section 6.3 will detail the evaluate indexing neighbours, the cost of indexing and a straight comparison between Kd-trees vs breadth first searches when mapping particles to the correct area in the vector field.
Section 6.4 will evaluate buffers when writing particle trajectory to disk to ascertain the optimal buffer size and compare it with alternate approaches.
Section 6.5 will evaluate the chosen Particular particle update implementation described in chapter 4 with other alternate approaches described in chapter 5.
Section 6.6 will evaluate queues
Section 6.7 will evaluate the cost of interpolation and the real effects of it. (Is it worth the performance hit?).
Section 6.7 will compare opendrift and particular with a profile of opendrift and a execution time comparison.

| | |
|---|---|
| Cpu | Intel Core i7-4770K CPU 3.50GHz (Haswell), 1 CPU, 8 logical and 4 physical cores |
| RAM | 32GB 1600MHZ DDR3 RAM |
| HD | Samsung 256GB SSD with 500 MB/s read and 200MB/s write speed |
| GPU | Geforce 780GTX GPU |
| OS | Windows 10.0.19041.985 |

**Table 6.1:** Hardware specifications for evaluations

## 6.1 Experimental Setup

Hardware specifications used for all evaluations are listed in table 6.1. Furthermore all examples are evaluated with the .Net 5.0 compiler with the F# programming language with regards to Particular or Python 3.9 if evaluating the Opendrift framework. The benchmarkdotnet[1] library was used for all evaluations with the exception of the direct opendrift vs particular evaluation.

### 6.1.1 Benchmarkdotnet

The benchmarkdotnet library is, as the name implies, created for profiling .net code. Benchmarkdotnet is capable of measuring various low level characteristics such as cache hits, branch mispredictions, memory allocated etc.

Benchmarkdotnet when ran will generate an isolated project for each runtime setting and run it in the dotnet release mode. Each method and parameter combination provided for benchmarking will be launched as a benchmarking process multiple times. An invocation of the workload method is defined as an operation. An operation will be important for interpreting the results later on. A collection of operations is known as an iteration, an each benchmark has multiple forms of iterations.

These iteration types include; overhead warmup and overhead workload where the benchmarking library overhead is evaluated. Actual warmup; warmup of the workload method. Actual workload; the actual measurements. This means that the final result is the actual workload - the median of the overhead. The actual number of each iteration is set by default by the library for maximum perceived accuracy, this may be changed manually however for these evaluations they were not.

1. https://benchmarkdotnet.org/articles/overview.html

| Method | Name of method profiled |
|---|---|
| N | The number parameter for the specific run |
| Mean | The mean run time of the profiling |
| Gen (X) | The amount of GC gen X collections per 1000 operations. |
| Allocated | Size of allocated managed memory, per single invocation. |
| CacheMisses/Op | Cache misses (L1, L2 and L3) per operation in an invocation |
| BranchMispredictions/Op | branch mis-predictions per operation in an invocation. |

**Table 6.2:** Benchmarkdotnet legend

#### Benchmarkdotnet Profiling Interpretation

As stated previously, benchmark dotnet collects numerous characteristics of the program benchmarked, however it is not always clear what the numbers mean in the context of the characteristic. Thus a legend of the benchmarking results which will be used numerous times this chapter was created and may be viewed at table 6.2.

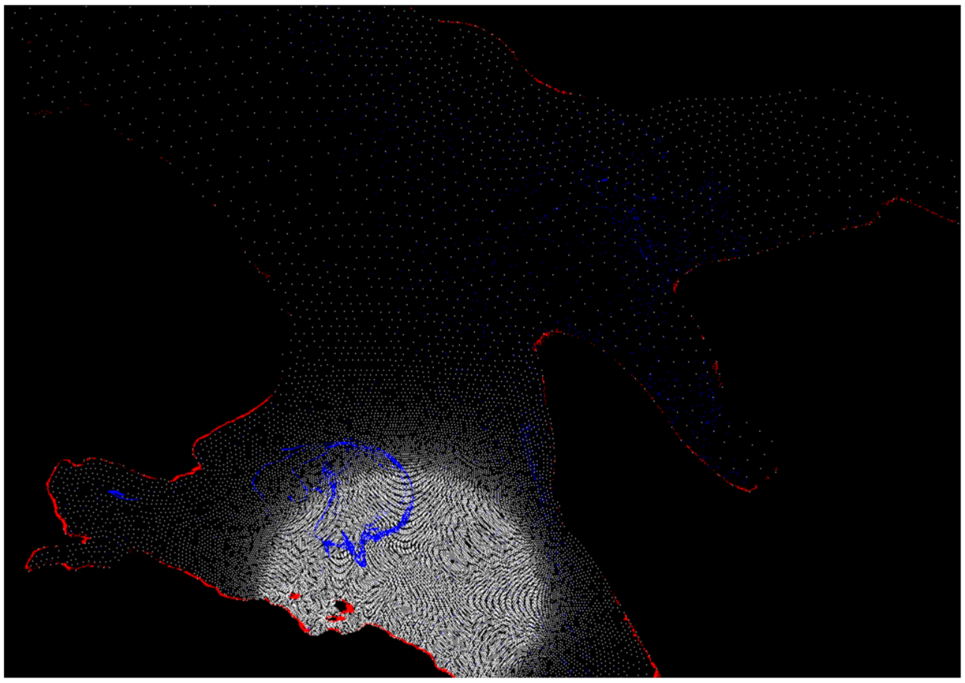
For the gen(X) profiling data as stated it counts GC collections per 1000 operations. An example being; if the value of Gen(x) is equal to 1 then that means the GC collects memory once per one thousand of benchmark invocations in generation X. The .Net GC has been explained in detail in section 2.6 in this thesis.

For the allocated profiling data, it only collects the amount of managed memory allocated. Thus any stack allocations or native heap allocations are not included.

### 6.1.2 Simulation Data

All evaluations will be done with a 3 dimensional vector field that represents the sea surrounding storholmen island. This simulation will be done over 24 hours with changing currents every hour. The vector field consists of 92073 elements composed of 40762 unique nodes. A picture of the vector field representing storholmen mid simulation can be viewed in figure 6.1. When the particles are seeded they will be dispersed uniformly across the vector field. All particles are also advected with the oceanographic forces previously described in the background and implementation chapters.

**Figure 6.1:** Partial view of storholmen in vector field form during particle simulation. Blue pixels are active particles. Red pixes signify particles that are stuck. White pixels representing nodes forming triangular elements.



## 6.2 Profiling of Particular

When profiling particular as a system of components. A simulation was ran as normal with two defined stages of the simulation which is preparing main loop and main loop. The reasoning for this is to keep it as close to the opendrift profiling tool as possible, which outputs the data in this format, as a basis of comparison.

The simulation used the aforementioned storholmen vector field with 120000 particles initially seeded over a 24 hour simulation time. With a time delta of 1 hour and a time step of 10 minutes. Which equals to a total of 144 updates for the simulation. The time was taken using the .net stopwatch[2] at various key points in the simulations execution path. The stopwatch class uses a high resolution performance counter if the operating system supports it, which in this case it does.

2. https://docs.microsoft.com/en-us/dotnet/api/system.diagnostics.stopwatch?view=net-5.0

| Time (sec) | Name |
|---|---|
| 19.7 | Preparing Main Loop |
| 0.26 | Reading Grid files from Disk |
| 19.3 | Indexing Neighbouring Elements |
| 0.1 | Seeding Initial Particles |
| 48.55 | Main Loop |
| 0.8 | Reading Environmental Data |
| 15.0 | Updating Particles |
| 6.0 | Advecting Particles |
| 2.7 | Interpolation |
| 9.0 | Mapping Particles |
| 1.0 | Emptying Queue |
| 25.7 | Writing Particles to File |
| 68.25 | Total Time |

**Table 6.3:** Opendrift Profiling Results

With this we can see there are three main steps during the simulation which takes most of the execution time, namely indexing neighbouring elements, updating particles and writing particles to file. These three steps taking 93% of the total execution time (64 seconds out of 68.25).

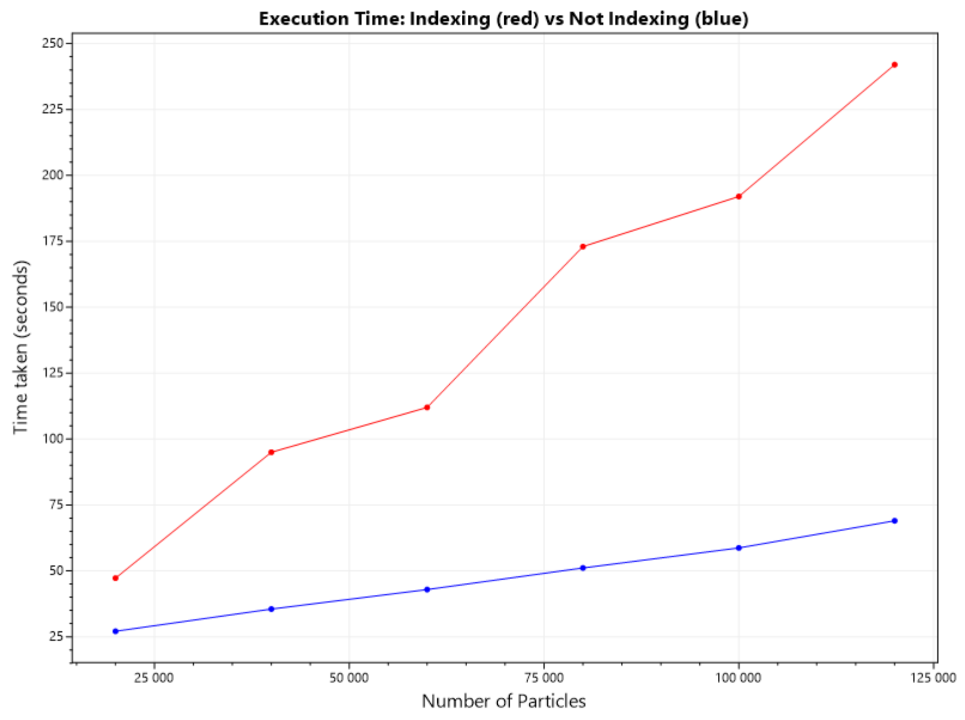
## 6.3 Indexing Neighbouring Elements

### 6.3.1 Performance cost of not indexing

Intially the implementation did not index data and simply queried the kd-tree every time a list of neighbours for an arbitrary element was needed. Which would happen frequently and would scale linearly with the amount of particles simulated.

Thus an evaluation of the execution time comparing a non indexed kd-tree solution with an indexed kd-tree solution was carried out. The evaluation itself ran a simulation with the aforementioned provided vector field over a 24 hour period. The execution time was recorded with the .net stopwatch. The result can be viewed at figure 6.2 where indexing will give a 320.29% execution time speed up (221 seconds vs 69 seconds). This is within expectation as queries to a Kd-tree is O(n), indexed into arrays like the current implementation is a O(1) operation.

**Figure 6.2:** Indexed (red) vs Not indexed (blue).



| Name | Time (Seconds) |
|---|---|
| Kd-Tree | 19.1 |
| Breadth First | 26.2 |

**Table 6.4:** Kd Tree index time comparison of 20 neighbours vs indeing 3 rings ( 24 neighbours)

### 6.3.2 Kd-Tree vs Breadth first indexing

The goal of this evaluation is to uncover the performance differences between a breadth first search indexing or kd-tree indexing to find the list of nearest neighbours for a specific element. As both are indexing they are ran before the main simulation loop thus do not scale with the number of particles, only the number of elements in the vector field.

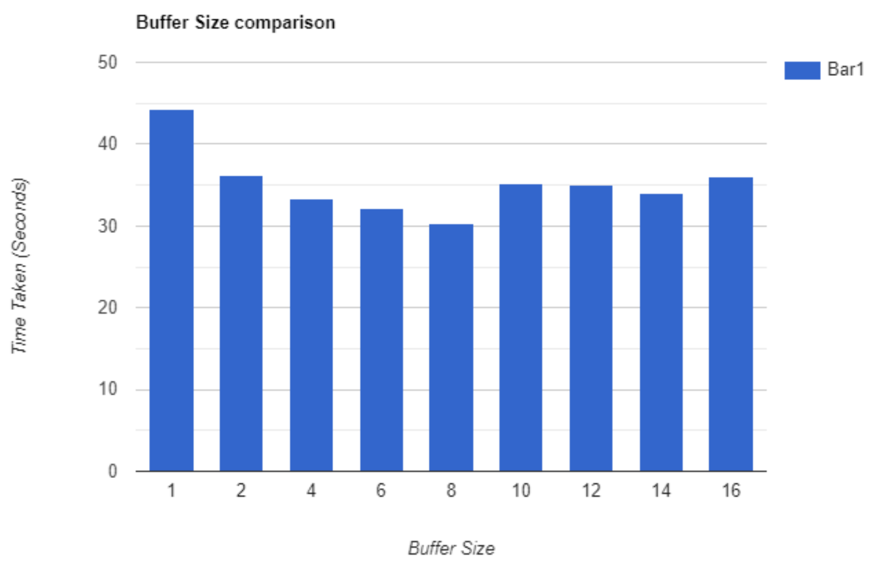
However breadth first may come out ahead for the nearest 2 rings. It will scale very poorly once you start going above 3 rings. Ring 4 was recorded to take 30 minutes which would give roughly 32 neighbours for each element, (each ring contains 8 neighbours). A result overview can be seen with the graph 6.4, where the kd-tree outperforms breadth first at 20 and will scale better as the neighbours increase.

## 6.4 Writing Particle Trajectories to Disk

This section will detail the evaluation of the process of writing particle trajectories to disk. Current implementation uses a double buffering scheme as described in the implementation chapter. A comparison will be run for the execution time for the double buffering scheme with varying buffer sizes to ascertain the optimal buffer size for a set amount of particles. A comparison will also be made with other implementations such as single buffer and no buffer.

### 6.4.1 Buffer size evaluation

**Figure 6.3:** Buffer Size Execution Time evaluation.

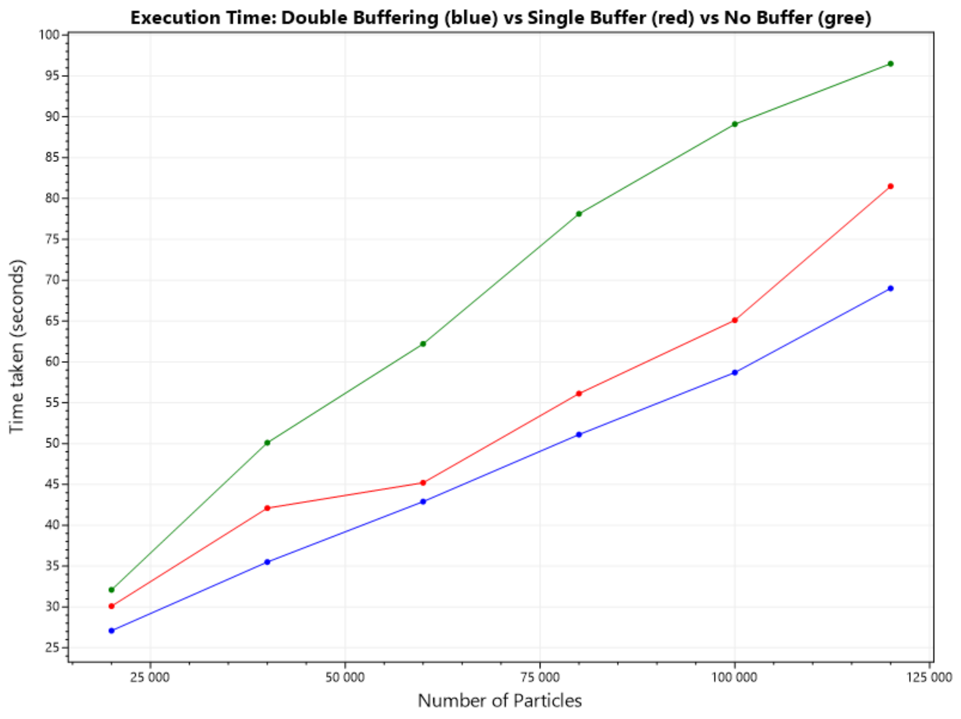


The current evaluation runs the particular simulation on the aforementioned vector field with 120000 particles initially seeded uniformly across the grid. If the vector field provided has 24 time steps it means the maximum amount of writes which can happen is 24.

It can be observed from 6.3 that a buffer size of 8 for environmental data with 24 total time steps seems to be optimal. This would likely change if the amount of time steps changes, however this result is not surprising as this means a clean 3 writes. It is a bit surprising that a buffer size of 12 is not superior as that is 2 clean writes. It is possible that SDS lite has problems handling large writes to that extent or the amount of memory needed that may hit performance negatively.

### 6.4.2 Comparisons with alternate buffer approaches

**Figure 6.4:** Buffer execution time comparison. Blue = Double Buffering. Red = Single Buffer. Green = No Buffer



The comparison evaluation was done by running a particular simulation with all buffers on the aforementioned vector field with a varying number particles initially seeded across the grid. Each buffer had a size of 8 where available which was found to be the best for both single and double buffering size wise.

Figure6.4 displays the total execution time for these particle numbers. It can be observed that double buffering wins out consistently with single buffers being a constant number two and no buffering lagging behind. All have close to linear scaling from particle array sizes of 20000 - 120000. Which is within expectation.

## 6.5 Updating Particles: Alternate approaches

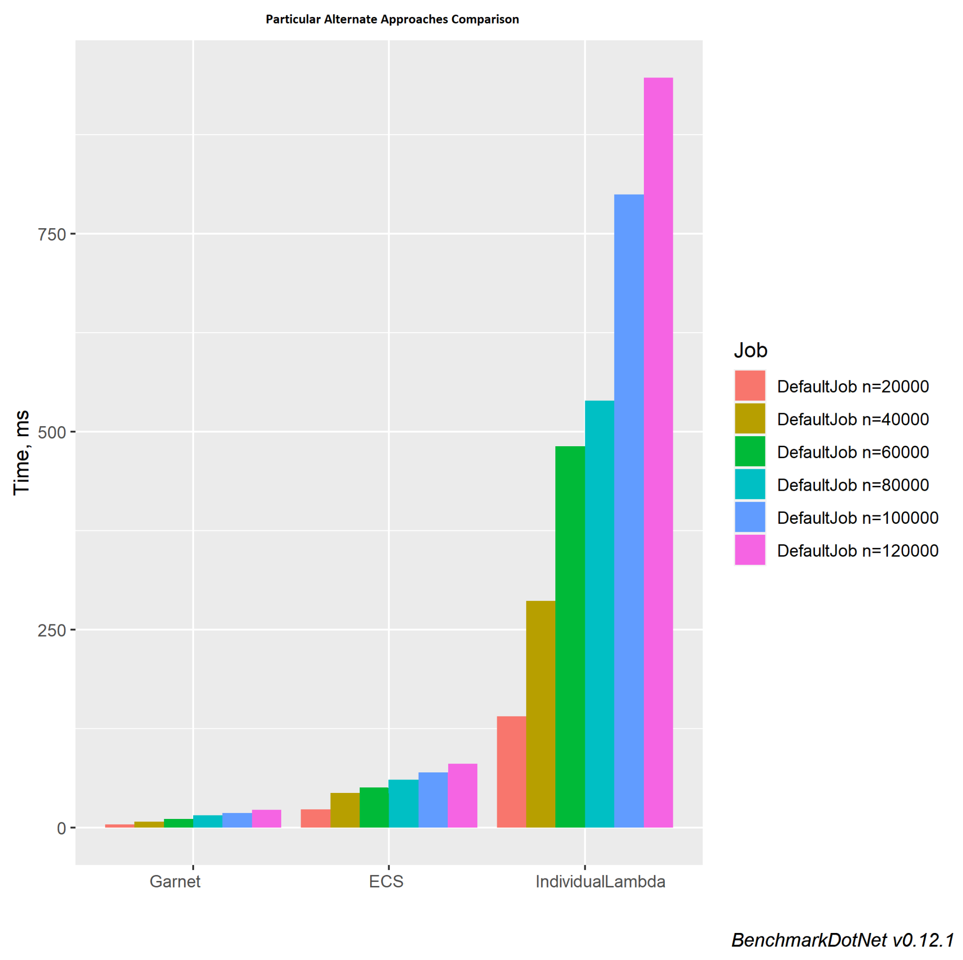
This section will detail the evaluation of the alternate approaches mentioned in section 5.1 (individual lambda) and 5.2 (garnet) compared to the current implementation (ECS) discussed in greater detail in the entirety of chapter 4. The experiment was conducted by isolating the particle iteration process within particular to only focus on how particles were stored and how they were iterated through when advected.

The experiment itself tested the capability of each approach to seed a set of N amounts of particles from 20000 - 120000. Then proceed to iterate through 24 hours of simulation data. A timestep of 10 was used which adds up to 144 iterations in total,(there are 144 10 minute chunks in 24 hours), of N particles. The benchmarkdotnet library was used to get run time performance as well as other data such as memory overhead, cache misses etc.

As can be viewed in figure 6.5 the three approaches are grouped with garnet being the fastest implementation followed by the current implementation with the individual lambda approach trailing significantly behind. With garnet at 120000 particles being 70.66% faster than the current implementation ($22.217ms$ vs $75.421ms$), and a sizeable 4200% speedup compared to the individual lambda approach $22.217ms$ vs $917.24ms$.

A profile of memory usage and other characteristics was also done with the benchmarkdotnet library and the result can be viewed in figure 6.6. This figure gives results within expectation and again illustrates the cost of immutability when it comes to performance. Since garnet does a lot of optimizations to avoid garbage collection and forgoes immutability as a result. Without immutability you get less memory allocated, less cache misses and less garbage collection. This makes intuitive sense as when you have an immutable collection every mapping will create an entirely new collection, rendering the previous references stale which will trigger the garbage collector. This means you constantly have new references this also means the cache will miss quite a bit as well and more memory is allocated as you are constantly creating new objects for every iteration. Further details on how Garnet avoids the GC and what costs it may come with will be discussed further in section 7.2.

**Figure 6.5:** Alternate approaches and current implementation run time performances

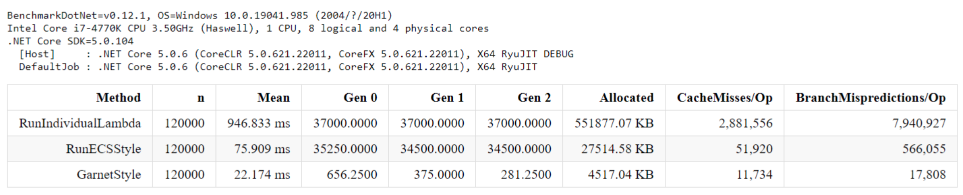


## 6.6 Queues: Functional Queue vs .Net Concurrent Queue

The goal of this evaluation is to discern the differences in performance and attempting to discover the underlying reasoning for the differences between a purely functional queue created in F# described in section 5.5 and the built-in Microsoft developed concurrent queue.

The experiment would be accomplished with each respective queue filled with N amount of messages, before being emptied and passed into a reducer. The reducer would take the message emptied from the queue and do a simple increment operation on a dummy model object.

**Figure 6.6:** Alternate approach profiling



```
BenchmarkDotNet=v0.12.1, OS=Windows 10.0.19041.985 (2004/?/20H1)
Intel Core i7-4770K CPU 3.50GHz (Haswell), 1 CPU, 8 logical and 4 physical cores
.NET Core SDK=5.0.104
  [Host]     : .NET Core 5.0.6 (CoreCLR 5.0.621.22011, CoreFX 5.0.621.22011), X64 RyuJIT DEBUG
  DefaultJob : .NET Core 5.0.6 (CoreCLR 5.0.621.22011, CoreFX 5.0.621.22011), X64 RyuJIT
```

| Method | n | Mean | Gen 0 | Gen 1 | Gen 2 | Allocated | CacheMisses/Op | BranchMispredictions/Op |
|---:|---:|---:|---:|---:|---:|---:|---:|---:|
| RunIndividualLambda | 120000 | 946.833 ms | 37000.0000 | 37000.0000 | 37000.0000 | 551877.07 KB | 2,881,556 | 7,940,927 |
| RunECSStyle | 120000 | 75.909 ms | 35250.0000 | 34500.0000 | 34500.0000 | 27514.58 KB | 51,920 | 566,055 |
| GarnetStyle | 120000 | 22.174 ms | 656.2500 | 375.0000 | 281.2500 | 4517.04 KB | 11,734 | 17,808 |

| Method | N | Mean | Gen 0 | Allocated | CacheMisses |
|---|---|---|---|---|---|
| Concurrent | 1000 | 24.97$\mu$ s | 5.73 | 23.48KB | 8 |
| Functional | 1000 | 24.97$\mu$ s | 23.01 | 94.02KB | 12 |
| Concurrent | 10000 | 302.67$\mu$ s | 58.12 | 234.48KB | 91 |
| Functional | 10000 | 368.34$\mu$ s | 188.96 | 937.02KB | 423 |
| Concurrent | 50000 | 1498.97$\mu$ s | 285.15 | 1171.48KB | 561 |
| Functional | 50000 | 3121.34$\mu$ s | 800.78 | 4687.02KB | 3,742 |
| Concurrent | 100000 | 2981.12$\mu$ s | 570.73 | 2343.48KB | 1,700 |
| Functional | 100000 | 9788.81$\mu$ s | 1562.50 | 9375.02KB | 17,224 |
| Concurrent | 500000 | 15220$\mu$ s | 2859.35 | 11718.48KB | 16,892 |
| Functional | 500000 | 98172$\mu$ s | 8200.01 | 46875.02KB | 239,702 |
| Concurrent | 1000000 | 32082$\mu$ s | 5687.5 | 23437.48KB | 29,039 |
| Functional | 1000000 | 241302$\mu$ s | 16000 | 93756.02KB | 769,602 |

**Table 6.5:** Table of queue benchmark results

This would aim to test the throughput of the queue both when adding and emptying, as well as other characteristics such as memory overhead.

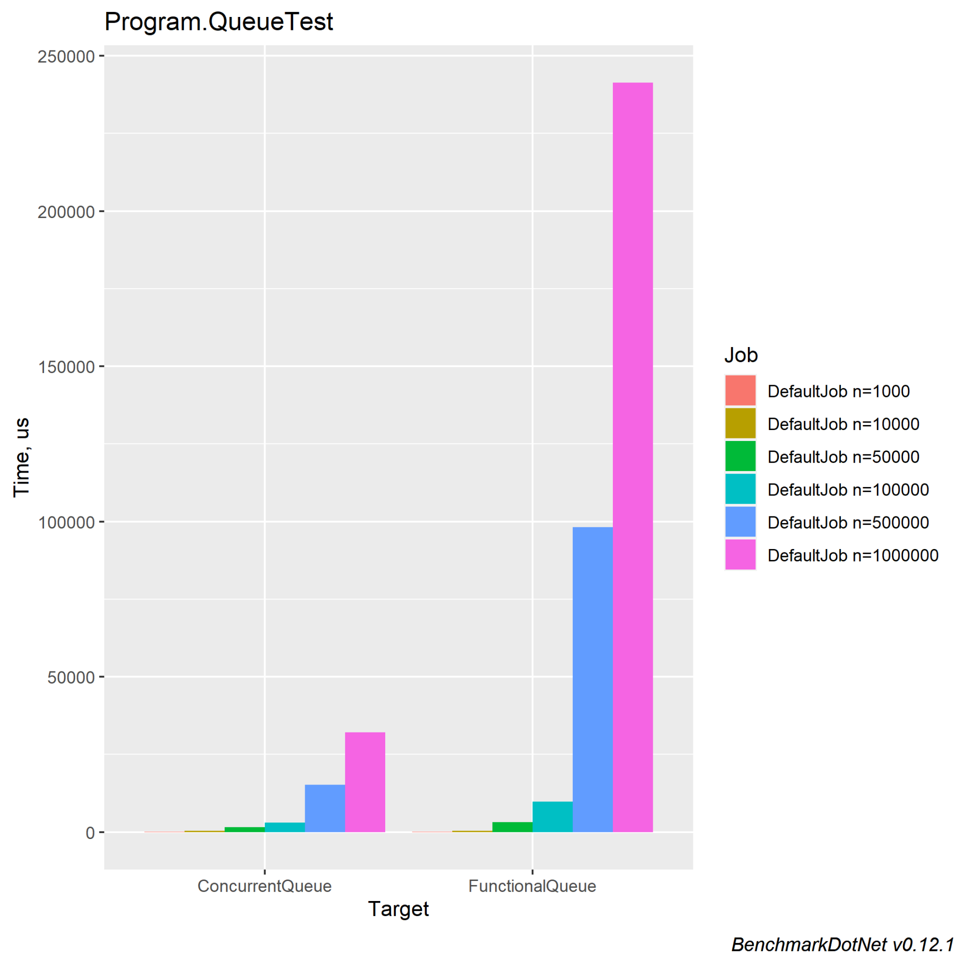
To evaluate this experiment, the benchmark dotnet library was used to benchmark the experiments.

Figure 6.7 and table 6.5 displays the results of the experiment. It can be observed at first glance that the built-in Microsoft queue vastly outperforms the purely functional queue in these specific tests.

Table 6.5 displays the amount of memory allocated and amount of cache misses performed by each test, with the Concurrent queue handily outperforming in all metrics.

This is within expectation as mutability, while mutability may introduce erroneous behaviours, it does also provide more efficient computing as a natural trade-off. This is especially true when dealing with the .net garbage collector

**Figure 6.7:** Bar plot of queue test. Y-axis shows time taken. Each bar represent a specific number of messages and test type, i.e functional or concurrent queue test.

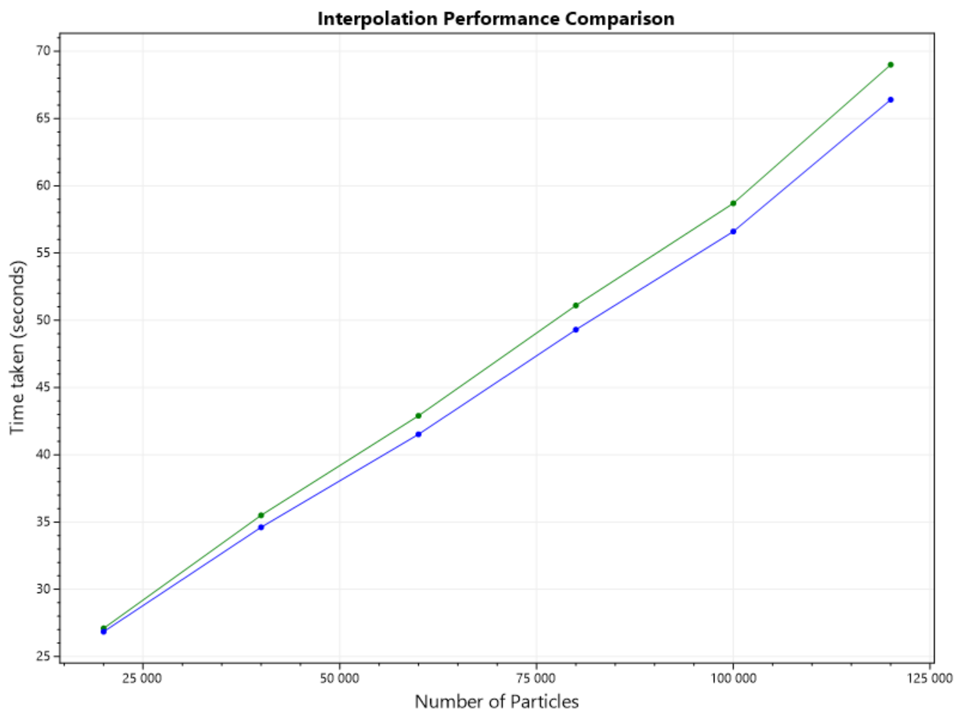


as each update of the queue incurs the garbage collector innately as the previous queue reference is abandoned as the structure is immutable, which is a significant performance hit over time.

## 6.7 Interpolation

The goal of this evaluation is to discern the costs of interpolation, both in run-time performance and potential memory overhead. The actual experiment was conducted with running particular with the provided simulation data with both interpolation and no interpolation. Which yielded the result which

**Figure 6.8:** Interpolation performance evaluation. X-Axis: Amount of particles. Y-Axis: Time Taken in seconds. Blue line: No interpolation. Green line: Interpolation.



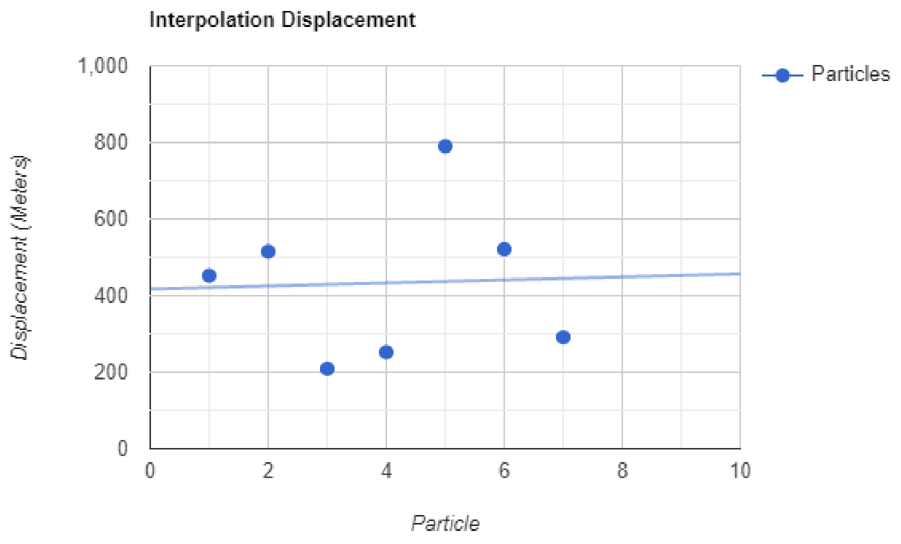
can be observed in figure 6.8. Interpolation scales in a similar manner of no interpolation but has a hit on the execution time, with up to a 3.8% slow down with 120000 particles initially seeded.

## 6.7.1 Effects of interpolation

This subsection will detail an experiment to ascertain what the real effect of interpolation is in terms of the final trajectory result. This experiment was conducted by setting a specific initial positional seed of ten particles and let it simulate a complete 24 hour trajectory with and without interpolation. This would yield 10 vectors of the particles final position with and without interpolation which would illustrate the potential change in results by applying the Pythagoran ($\sqrt{(}a^2 + b^2)$) formula to calculate the distance between these two vectors.

As the vector field provided for testing did not have the strongest currents (average of 0.09 m/s currents in the vector field provided), or other forces which enact upon the particles trajectory. However despite this there was an

**Figure 6.9:** Displacement of particles with interpolation vs no interpolation



average displacement of  450 meters.

## 6.8 Particular vs Opendrift

### 6.8.1 Execution time comparison

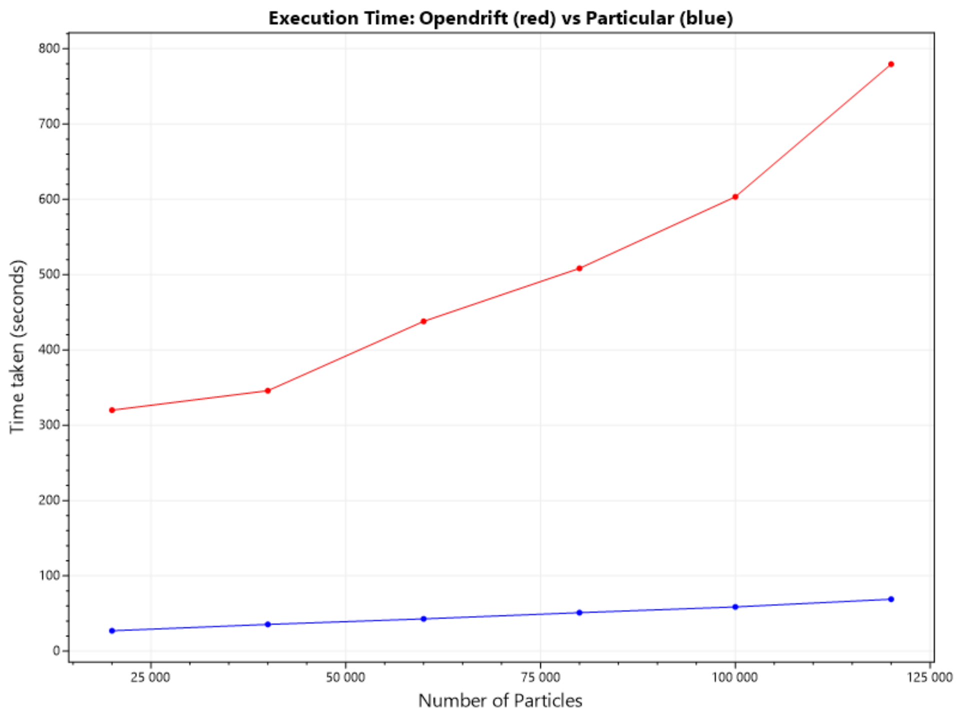
The goal of this evaluation is to detail the difference in performance at runtime between Particular and Opendrift. The experiment was conducted by having both opendrift and particular seed N amount of particles into a specific vector field of data. The simulation data lasted for 24 hours with a time step of 10 minutes for each particle advection. A total of 10 runs were done for both opendrift and particular and the mean of these 10 runs were taken as the final result to avoid potential random hiccups with the hardware while benchmarking.

The time taking for particular was done with the .net stopwatch[3] class which uses a high resolution performance counter if the operating system supports it, which in this case it does. Opendrift uses the timeit[4] python library to time its experiments, like the .net stopwatch uses a high resolution performance

3. https://docs.microsoft.com/en-us/dotnet/api/system.diagnostics.stopwatch?view=net-5.0
4. https://docs.python.org/3/library/timeit.html

counter if available.

**Figure 6.10:** Particular vs Opendrift, Particular represented by blue, opendrift by green.



As can be viewed in figure 6.10, particular outperforms opendrift in this specific experiment under the conditions described above. With the x-axis representing the amount of particles initially seeded and the y-axis the time taken for the simulation to complete in seconds. Opendrift at 120000 particles has 1128.99% less execution time (69 seconds vs 779 seconds), and will in all likelihood increase as opendrift scales worse than particular as the number of initial particles increases.

## 6.8.2 Profiling of opendrift

As can be seen in the profiling in table 6.6, particular outperforms opendrift at most stages even if the profiling stages are not exactly the same due to architecture differences. With the biggest difference happening when reading data from the environment which can be seen in the "storholmen2.nc" entry which is the file name of the simulation data. As opendrift does not really cache its data in memory, rather fetch it lazily from disk which has the pro of less memory overhead, but the con of a major performance hit when needing the

| Time | Name |
|---|---|
| 1.1 | Configuration |
| 8.6 | Preparing Main Loop |
| 3.8 | Making Dynamic Landmask |
| 4.7 | Moving Elements to Ocean |
| 9.28 | Readers |
| 14.3 | Global Landmask |
| 55.5 | Post Processing |
| 11:37.8 | Main Loop |
| 7:20.9 | storholmen2.nc |
| 1:33.4 | Updating Elements |
| 1.0 | Clean up |
| 11:48.6 | Total Time |

**Table 6.6:** Opendrift Profiling Results

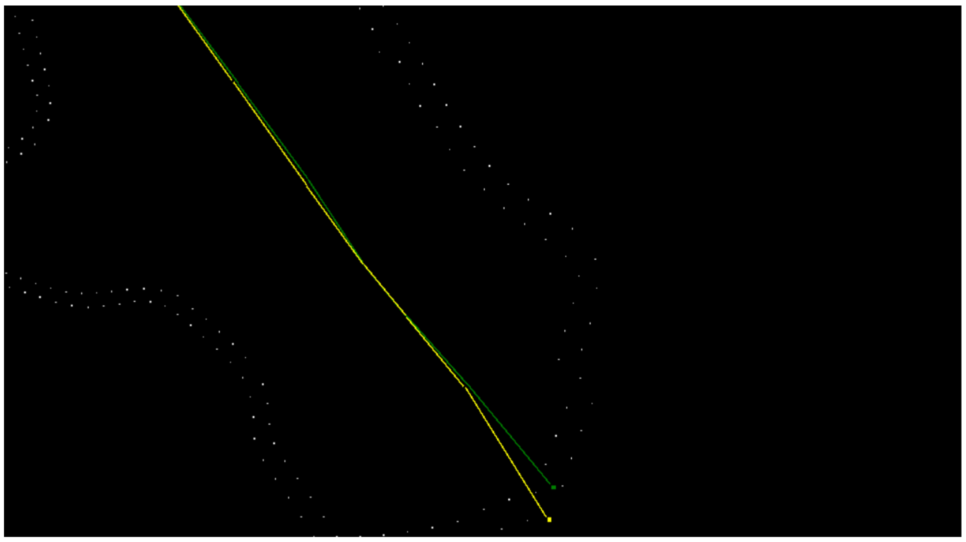
data.

### 6.8.3 Sanity check

A sanity check was also ran to see what the difference in the final result would be if two identical vector fields were provided to both particular and opendrift with a particle set at an identical starting location. The difference would be measured in the final trajectory position for each respective library. Important to note that this is not intended as a form of verification, simply a sanity check to ascertain whether the results for both frameworks are approximately the same.

The evaluation results for this was a displacement of 233.5 meters over a total distance travelled of 6.904$km$ for opendrift and 6.737$km$ for particular. A possible explanation for this discrepancy can be explained by opendrift using the 4th degree runga kutta advection whereas particular only supports euler's method of advection which is slightly less accurate approximation and will sometimes give different advection results. A visualization of these two particle vectors can be viewed at figure 6.11, which shows both vectors and the difference at the end of their simulated trajectories.

**Figure 6.11:** Particular vs Opendrift result comparison. Yellow = Particular, Green = Opendrift



# 7 Discussion

This chapter will contain discussions around Particular and what potential pros and cons there are to the prototype and why certain design choices were made.
Section 7.1 will discuss the key differences between Opendrift and Particular, with the perceived pros and cons of both being discussed.
Section 7.2 will discuss various possible optimizations that could be done to the current implementation of Particular and what costs it would potentially have. In addition a look at how Garnet was implemented and what pros and cons it may have will also b done.
Section 7.3 will discuss interpolation techniques in detail.

## 7.1 Opendrift vs Particular

As opendrift is implemented in python, an object oriented language compared to Particular which was implemented in F#, a functional programming language. It follows that there are some key differences in the implementation of both frameworks. This section will discuss the most significant of these and their design choices and what trade-offs these decisions result in.

### 7.1.1 Architecture of Opendrift

As previously mentioned opendrift is implemented with the OOP paradigm primarily and uses cornerstone OOP concepts such as inheritance and polymorphism as a way to introduce extendibility to their framework. This is in directly opposed to Particular which relies on purely on functional composition.

Opendrift has some base abstract objects which through inheritance and polymorphism allows a user to extend this framework with their own defined behaviour. Theses abstract classes include, the particle array which keeps track of the particle trajectory of the simulation. The model of the simulation which has methods such as update which must be overridden to define which functions are applied to the particles for each time step update. A reader class which is responsible for reading environmental data from disk for use in the simulation. Particular on the other uses lambda functions as described in the implementation chapter to define behaviours in events, particle updates and how to write data to disk. Which requires significantly less code and is just as extendable or even more so in comparison to opendrift. Even if opendrift has more readily made models and mathematical tools available, adding these models and tools is much easier in particular and requires almost none of the boiler plate code which OOP requires.

### 7.1.2 Particle Tracking

Opendrift uses a generic array class which is referred to as a lagrangian array which is tasked with keeping track of particles. The particles themselves are a collection of variables stored as an ordered python dictionary. So in the context of oceandrift the lagrangian array would have 1 ordered dictionary with Z, latitude and longitude values set in arrays with the keys being a simple "z", "lon" or "lat" string. In addition to the base variables such as id, status and age. This approach does allow for an easy and maintainable way for the user of the framework to customize the particles that are ran in the simulation.

Particular on the other hand uses the ECS inspired architecture. The main advantage is the ease of having separate particle types within a single simulation and being able to iterate through them in a CPU friendly manner. As it is essentially a dictionary with a type as a key and a corresponding array as value. Which means no matter how many particle types there are iterations are guaranteed to be contiguous sequential memory of particle data when iterating. While it is possible to have separate particle types in opendrift this leads to having branched data in the python dictionary which means for some entries in the dictionary behaviour for particle A will be executed and for other entries particle B behaviour will be executed. This is both not ideal in a maintainable sense nor in a performance sense as this sort of data handling leads to branching code which in turn is not CPU friendly. Which is backed up by the evaluation and profiling in the previous chapter where updating particles took particular 15 seconds whereas opendrift spent 40 seconds updating its particles.

As opendrift uses numpy[9] arrays for particle data it allows for very fast iterations. As numpy arrays in python are thin bindings for optimized pre-compiled C code. In addition to this Numpy arrays are also vectorized[10]. This means when faced with SIMD situations such as an array of particles having a specific vector mapped onto it it can provide great speed up as the CPU may process multiple operands on a single operation instruction making for more efficient computing. This as opposed to non-vectorized code where each instruction only operates on one operand at a time. This counteracts the drawback of opendrift not providing multiprocessing or multi-threading.[1].

Particular on the other hand does not have access to the numpy library as it is built in the .net environment. .Net recently has created intrinsic and vectorization hardware instructions within the .net environment for their RyuJit[2] compiler to allow for vectorization with the use of a built in vector type [3]. While Particular as of right now does not have this implemented and would require some redesign it is something to consider implementing in the future for even faster iterations when mapping data onto a particle array.

The redesign would have to change the way particles are stored, as they cannot be set in algebraic data types and must be purely primitive value types. So particles would be a collection of primitive value arrays such as floats and integers. However as seen in the particular profiling results, the particle updating was not the bottle neck of the simulation execution time thus was not prioritized.

1. https://wiki.python.org/moin/GlobalInterpreterLock
2. https://devblogs.microsoft.com/dotnet/ryujit-the-next-generation-jit-compiler-for-net/
3. https://docs.microsoft.com/en-us/dotnet/api/system.numerics.vector?view=net-5.0

### 7.1.3 Reader

Opendrift have several reader types derived from a base reader type which allows for extendability for reading from different sources of files, not just netCDF which Particular at his point only supports. The readers main job is to make environmental data available as well as manage the vector field mesh that particles are mapped to. A focus will be on a reader of unstructured grids which is what has been evaluated and is what Particular supports.

#### Setting Up Boundary For the Vector Field

Opendrift uses landmasking to set up the boundaries of its unstructured vector field. It does this by building a boundary polygon of a provided mesh to find the boundaries of the given field. It does this by running an algorithm which checks for boundary edges, which are defined as edges that are referenced by only a single triangle. These triangles are then given a special flag to indicate that the triangle is an edge triangle in the mesh.

The particular implementation does not currently have such a robust solution and relies on a specific format of the data grid data. The grid data will provide the amount of elements surrounding a node, and if an element's surrounding node has less than 4 neighbouring elements it intuitively means that the current element is a edge element and is flagged as such. The differences between the two approaches are not all that significant performance wise as can be seen in the evaluation.

A major difference however is the time spent reading from simulation data at the disk, opendrift spends almost 80% of its execution time reading from disk, whereas particular spends a fraction of the time reading from disk. Opting instead reading the data into memory, whereas opendrift lazily fetches the data when needed. This is good for memory efficency however hurts opendrift when it comes to performance massively having to keep fetching data from the hardrive for every particle update.

#### Mapping Particles to Vector Field

When mapping particles to the mesh after advection Opendrift uses a KdTree[4] from the scipy library to find the closest neighbours. Initially this tree is indexed by triangle positions and may be queried by particle positions to receive N amount of neighbours.

4. https://docs.scipy.org/doc/scipy/reference/generated/scipy.spatial.cKDTree.html

Particular on the other hand uses a KdTree to index all elements of the grid and their nearest neighbours. However, unlike opendrift it puts this neighbour data into a nested array for ease when looking up the neighbours of a specific element by the index. As opposed to keeping it in the tree, this allows for a O(1) + O(m) look up time with the M being the number of neighbours recorded for each element.

#### Interpolation

Opendrift uses linear interpolation which the scipy library provides interpolation[5]. Opendrift, unlike particular, currently supports 2d interpolation in addition to 1d interpolation. The main difference between the dimensions is that in 2d interpolation the depth of is also interpolated, so a particle at depth 5 will also interpolate with the values on depth 6 and 4 in addition to the usual immediate neighbours.

The trade-offs here are a more accurate approximation with 2d interpolation for performance which 1d provides as there is less look ups and operations required.

### 7.1.4 Advection

Opendrift lets the user choose between to advection methods, 4th order runga kutta and eulers method. Both of these methods are described in the background chapter earlier in this thesis. Particular on the other hand currently only supports eulers method, runga kutta will in theory give a more accurate advection at the cost of performance as there are simply more computations necessary.

### 7.1.5 Simulation Model

The opendrift simulation model is designed as a base model which contains a lagrangian array of particles. An update method is provided with the functions called for particle advection which may be overridden and is called every time step. Other helper methods are also available which may be overridden by the user such as behaviour when particles hit the bottom or when particles hit the edge of the vector field etc.

Particular rather than specific overridden methods for events such as hitting

5. https://docs.scipy.org/doc/scipy/reference/generated/scipy.interpolate.interp2d.html

the bottom of the ocean of the coast uses messages and queues which are described in both the design and implementation chapters. This provides a more generic way for the user to define the events and the result of these events themselves.

## 7.2 Optimizations

### 7.2.1 Garnet vs Particular

As can be seen in the benchmarking in figure 6.6, managed heap allocations, garbage collection and cache hits are increased by magnitudes between the garnet implementation and the current particular implementation.

The way garnet manages this is by using various techniques to minimize the GC impact. These techniques are; allocating on the stack, object pooling and avoiding closures.

Garnet allocates most of its objects on the stack by the use of the .net struct type, which turns objects to value types as opposed to regular reference type which a default F# record type is. When on the stack all variables declared are managed automatically by being destroyed when returning from a function. As a bonus all memory fragmentation is also avoided. The drawback is if the value type is large the copy may be expensive as a copy is performed whenever a value type is passed into a function.

Particular also lets the user define its types and it is up to them to make it a value or reference type, with value types usually outperforming reference types unless the value types become very large in size.

Garnet also avoids all sort of closures in F#, with closures being defined as a block of code which may be executed at a later time, but maintain the environment it was first created. So it can use local variables of the method it was created within, even after said method has returned. Garnet avoids this, particular however does not. The reasoning is that closures are not evil by themselves, while there is a performance penalty in using them sometimes closures are the correct answer for maintainable code. An example of closure usage is the partially applicated interpolation function in the particular reader, which keeps the reference of local variables when the interpolation function is declared for later use.

The biggest advantage garnet has over particular however is object pooling and overall allocation of objects. Garnet handles this by having a lot of pre-allocated buffers which are continuously re-used and never de-referenced so the GC never gets triggered. These buffers are also mutable so garnet, while implemented in F#, uses mostly mutable structures. These buffers take the shape of a sparse set data structure with entity indexes being the sparse set and components being in the corresponding dense set. This works as entities are simply integer indexes that points to an index in a corresponding dense set with components. This is also very performant as sparse sets has a O(1) for retrieving, inserting, removing and clearing data. On top of this components are also stored in a 64 element segments which provides a CPU-friendly iteration for caching purposes.

Particular on the other hand only has a collection of basic arrays, moreover these arrays are mapped and replaced each iteration as the values within them are immutable. This is by design as one of the key goals of particular was to create a 3d particle simulator using the functional paradigm. Thus foregoing immutability for performance would directly counter this goal, thus was not chosen. Immutability does have positive aspects such as being thread safe, (garnet does not have multithreading functionality), which particular takes advantage of. The cost of this is as can be seen in the memory evaluation, and somewhat intuitively, with increased allocations, increased use of the GC and cache misses.

### 7.2.2 Reading environment data

The current particular implementation currently fetches the environment data from a buffer in memory which has initially been read from disk. This is a valid approach, however for every fetch a copy has to be created, this strains the GC. To avoid this instead of using basic arrays to keep the data one could use the .net span type [6] type and .net readonly memory[7]. This would set up a virtual view of the data allowing particles to read the array data without copying, leading to a larger speedup and absolutely no GC involvement.

Another potential optimization is to create sub groups of environmental data and particles based upon their geographic positions. This would allow for more cache friendly iteration in theory, with spans getting a specific range of data within a larger array would also come at no extra cost performance wise.

6. https://docs.microsoft.com/en-us/dotnet/api/system.span-1?view=net-5.0
7. https://docs.microsoft.com/en-us/dotnet/api/system.readonlymemory-1?view=net-5.0

### 7.2.3 Improving type safety with F# crates

Across the implementation there was a recurring issue with a limitation of the F# type system. Namely as soon as a value is passed as an argument into a generic function, it will no longer be recognized as customisable to any generic type and is restricted to one application of type parameter.

The lone exception to this is that one may preserve polymorphism if locked within the local scope. This is due to the difference between a lambda bound and let bound polymorphism, with let bound polymorphism being predicative. The reasoning for it being this way is that predicativity and other limitations makes the type system simple enough that full type inference is always possible. Which is the trade off being made for limited polymorphism which is a feature of all ML dialect languages which includes f#.

**Listing 7.1:** function example

```
    forall a. [a] x [a] -> [a]
```

An example of this can be viewed in listing7.1, in order to apply the *forall* function to a pair of lists the *a* type must be substituted for the value a in the function typing such that the resulting function type and the type of the parameters matches up. For an impredicative system on the other hand the type substituted may be of any type including a type that is recursively polymorphic which means the *forall* function may be applied to any type of list containing elements of any generic type. As mentioned impredicative systems trade stronger polymorphism for not quite as robust type systems and type inference as it is no way for the compiler to 100% infer types when types can be anything (including recursive) as previously mentioned.

**Listing 7.2:** Optimal system function signature

```
    System : Particle<'a> -> Grid -> (msg -> unit) -> Particle<'a>
```

**Listing 7.3:** Systems list with different lambda types

```
[
    System (particle<int>  -> ..  ->particle<int>)
    System (particle<float> -> .. -> particle<float>)
]
```

These limitations were approached when implementing particular especially when implementing the systems portion of the particle iteration who's function signature can be seen at listing 7.2. The ideal implementation was to have the system lambda be completely generic and stored in a collection, then simply iterated through all systems at advection through that one collection. This however is not allowed by the F# type system due to the predicativity and meant that a circumvention had to be made casting the *particle* type to the .net *obj* type, as was described in the implementation chapter. As well as storing the system functions in a dictionary with a corresponding particle type it would transform. This then meant that the library user had to provide type parameters for every particle type in the simulation when defining particular, which is not ideal. This also meant that n amount of static overrides for running particular had to be coded with n signifying the amount of unique particle types there were in a given simulation. Particular currently supports 8 unique types.

Another way to get around this limitation of the f# type system is to use a pattern unqiue to f# known as crates. The crates job is to simulate existensials. The exisistensial concept may be found in languages such as haskell[8] innately. What existential types essentially does is infer a value as a type which is unknown statically either because it was intentionally hidden in a way which was known, or because the type was chosen at run time. At runtime it is possible to inspect the existential to find the value and type within. The problem still remains on how to extract this value when the type is unknown in a predicative polymorphic language. The solution is to provide a universally quantified function, which is a function that can handle values of any type. Thus existentials are values whereupon universals are the only construct able to operate on them.

**Listing 7.4:** Base crate example in F#

```
type Crate =
    abstract member Apply : CrateEvaluator<'ret> -> 'ret
and CrateEvaluator<'ret> =
    abtract member Eval<'a> : 'a -> 'ret
```

In f# an example implementation of the base crate may be viewed in 7.4. The method *Eval* takes a type parameter of *'a* and CrateEvaluator takes a type parameter *'ret*. This brings two different sources into scope at the same time to build our existential, this is also known as rank-2 polymorphism[11].

**Listing 7.5:** Crate sample for system in particular

```
type Particle<'a> = {
    Data : 'a
} with
    static member New d = {
        Data = d
}
```

8. *https : //wiki.haskell.org/Existential$_t$ype*

```
type System<'a> =
    System of
    (Particle<'a> -> Particle<'a>)*
    Particle<'a> []
with
    member this.Add (p) =
        let (System (q,pa)) = this
        System (q, Array.append [|p|] pa)
type SystemsCrate =
    abstract member Apply : SystemsCrateEvaluator<'ret> -> 'ret

and SystemsCrateEvaluator<'ret> =
    abstract member Eval<'a> : System<'a>  -> 'ret

[<AutoOpen>]
module SystemCrateOps =
    let make (s : System<'a>)  : SystemsCrate =
        { new SystemsCrate with
            member __.Apply e = e.Eval s
        }

    let inline run (s : SystemsCrate)  =
        s.Apply { new SystemsCrateEvaluator<_> with
            member __.Eval (l : System<'a>)  =
                let (System (f, pa)) = l
                pa
                |> Array.map (fun x -> f x)
                |> fun x -> make (System(f, x))
        }

    let print (s : SystemsCrate) =
        s.Apply { new SystemsCrateEvaluator<_> with
            member __.Eval (l : System<'a>)  =
                let (System (q,p)) =  l
                p
                |> Array.iter (fun s -> printfn "%A" s)
                make l
        }
let system = (
    fun p ->
        { p with Data = p.Data + 1 }),
        [|Particle<int>.New  6 ; Particle<int>.New 9|]
```

```
let system2 = (
    fun p ->
    { p with Data = p.Data + 1.0 }),
    [|Particle<int>.New  6.2 ; Particle<int>.New 9.2|]

let m = [
    make (System system)
    make (System system2)
]
m
|> List.map (fun c -> run c)
```

We can thus apply this to our systems example by creating a systems type with a function and a list of particles which the function is ran on every update. By observing the code listing at 7.5, we can see how we could apply these existensial concepts in f# to get type safety and removing the much maligned *obj* use. With the systems crate having three operations: *make* which creates a crate, *run* which updates the particle within a crate with the provided lambda function and *print* which prints the content of a crate. The end result of this is that we have allowed a list systems with different type parameters to exist. Thus when advecting in particular, we could simply just iterate through the one list, instead of using dictionaries and objects and variadic type parameters.

## 7.3 Interpolation

Particular currently uses barycentric coordinates which is an linear interpolation technique. The barycentric approach is also purely local which means it only takes weight contributions from the local points of the triangle and no outside sources. This approach is among the fastest ways to perform interpolation, however it comes at the cost of potential accuracy.

The inaccuracy will most likely appear on the border of the triangles. If triangle A borders triangle B, and both triangles have opposite mappings you will see a non-continuous value function between the triangles which is not derivable. This is an inaccuracy, which while not massive in scope does exist.

A possible solution to this if one needed more perceived accuracy would be to adopt a clough-tocher approach described in section 2.8.3. Clough-tocher derives conditions to ensure triangle continuity across edges of two triangles. These conditions are similar to bernstein polynomials[12] across the edge. Clough-tocher is also a cubic interpolation technique so there is also a higher fidelity of values available as opposed to the linear solution as an added bonus, however at the cost of extra computation.

This approach while more accurate would both be a performance hit as there are more computations with a more complex algorithm handling clough-tocher, as well as having to access triangle neighbours. This extra accuracy was not perceived worth it for the drop in performance as most triangles and their neighbours in the provided vector fields are likely to have similar mapping making the potential inaccuracies small to nonexistent. It is important to note that opendrift also uses a purely local interpolation scheme, thus the developers of opendrift also thought it good enough running a purely local interpolation.

# 8 Conclusion

This thesis has detailed the design and implementation Particular, a 3D trajectory simulator implemented in F#. The goal of Particular was to provide an alternative solution to other state of the art 3d trajectory simulators available today like opendrift by using the functional programming paradigm and the .net environment.

Compared to opendrift, particular allows for more performance and customizability of simulations with different types of particles without introducing complexity for the user of the library, by using an ECS-like architecture usually found in game development. Opendrift on the other hand is more mature framework and has more available models set up and a more generic implementation of reading environmental data. As opposed to particular which can only handle one specific format of data in its current implementation

An evaluation was also done on all particular components and potential alternate approaches with both profiling memory overhead and other program characteristics and benchmarking execution time. A direct comparison of opendrift vs particular was also done with particular having up to 1128.99% less execution time in the evaluation in equivalent simulation environments.

## 8.1 Future Work

While Particular currently functions for particle trajectory simulations with a high degree of customizability, there is a lack of generality when reading environmental data and vector field properties to particular. As of right now there is no ability for a reader to read from multiple files and there is only two readers available for the simulation corresponding to element and node points in the vector field. A way to implement a system that promises this generality would probably be a worthwhile endeavour.

In addition support for runga kutta advection and vertical interpolation is also a needed implementation to achieve the same accuracy as opendrift currently has. This will be at the cost of some performance, however is unlikely to be as significant as making particular on par execution time wise with opendrift.

In addition it is also possible to improve the implementation performance wise by implementing the optimizations discussed in section 7.2. Such as adding support for vectorization, lessening the load on the GC by using .net spans or a more CPU friendly iteration method. The question then becomes if the optimizations will lead to the code no longer being recognizable as functional. Thus limiting or removing the main benefits of going functional such as thread safety. For type safety implementing crates for all objects may be a worthwhile endavour for more type safety and less boiler plate code as well keeping the one of the corner stones of F#.

# Bibliography

[1] Linda R. Ascher Uri M.; Petzold. “Computer Methods for Ordinary Differential Equations and Differential-Algebraic Equations.” In: *Philadelphia: Society for Industrial and Applied Mathematics* (1998).

[2] P. J. Dormand J. R.; Prince. “New Runge–Kutta Algorithms for Numerical Simulation in Dynamical Astronomy.” In: *Celestial Mechanics* 18 (Oct. 1978).

[3] *Fundamentals of garbage collection*. Retrieved 17.May 2021. 2016. URL: `https://docs.microsoft.com/en-us/dotnet/standard/garbage-collection/fundamentals`.

[4] *Common Language Runtime (CLR) overview*. Retrieved 17.May 2021. 2020. URL: `https://docs.microsoft.com/en-us/dotnet/standard/clr`.

[5] Scott. Bilas. *ECS: A Data-Driven Game Object System*. [Retrieved 01.May 2021.

[6] Tocher Clough. *Clough Tocher, Triangular 2d Interpolation*. 1965.

[7] Micheal E Mortenson. “Mathematics for Computer Graphics Applications.” In: *Industrial Press Inc.* (1999), p. 264.

[8] Jean-Raymond Bidlot Øyvind Breivik Peter A. E. M. Janssen. “Approximate Stokes Drift Profiles in Deep Water.” In: (2014), pp. 2433–2445. DOI: `https://doi.org/10.1175/JPO-D-14-0020.1`.

[9] *What is NumPy?* Retrieved 17.May 2021. 2017. URL: `https://numpy.org/doc/stable/user/whatisnumpy.html`.

[10] J. Shin. “Introducing Control Flow into Vectorized Code.” In: *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques* (2017). DOI: `10.1109/PACT.2007.41`.

[11] Benjamin C. Pierce. “Types and Programming Languages.” In: *MIT press* (2002).

[12] Richard R. Goldberg. “Methods of real analysis.” In: *John Wiley Sons* (1964), pp. 263–265.