SIMULATING THE EMERGENT AUTONOMOUS BEHAVIOUR OF UNMANNED AERIAL VEHICLE SWARMS

by

ANTONIO BRITO

DRAFT

Supervised by Nathan Griffiths

Department of Computer Science
University of Warwick
2024

Contents

1 Context **4**
- 1.1 Introduction . . . 4
- 1.2 Background . . . 5

2 Physics Model **6**
- 2.1 Quadcopter . . . 6
- 2.2 Stabilisation . . . 9
  - 2.2.1 Proportional Controller . . . 10
  - 2.2.2 Derivative Controller . . . 11
  - 2.2.3 Integral Controller . . . 12
  - 2.2.4 PID Controller . . . 14
  - 2.2.5 PID Parameter Tuning . . . 15
- 2.3 Environment Model . . . 18

3 Introducing Autonomy **20**
- 3.1 Movement to Position . . . 20
- 3.2 Boids (1987) . . . 23
  - 3.2.1 Avoidance . . . 24
  - 3.2.2 Alignment . . . 25
  - 3.2.3 Cohesion . . . 26
  - 3.2.4 Combining Behaviours . . . 27
- 3.3 Extended Boids . . . 30
  - 3.3.1 Goal Seeking . . . 30
  - 3.3.2 Terrain Generation . . . 32
  - 3.3.3 Terrain Avoidance . . . 34

4 Methodology **37**
- 4.1 Scenario 1: Shortest First Arrival Time . . . 37
- 4.2 Scenario 2: Reduced Spread (Cohesivity) . . . 38
- 4.3 Scenario 3: Reducing Collision Numbers . . . 38

5 Optimisation **39**
- 5.1 Simulated Annealing . . . 40
- 5.2 Simulations . . . 42
- 5.3 Results . . . 42

6 Improving Autonomy **42**
- 6.1 Boids (1999): Introducing Obstacle Avoidance . . . 42
- 6.2 A Simplified Model . . . 42
- 6.3 Local Potential Field Emergence . . . 43
- 6.4 Final Results . . . 43

7 Future Work **43**
7.1 LPFE Implementation 43
7.2 Online Optimisation 43
7.3 Hostility 43
7.3.1 Hostile Environments 43
7.3.2 Hostile Agents 43

8 Evaluation **43**
8.1 Project Management 43
8.2 Comparing Models 43
8.3 Speed-Success-Flock Size Tradeoffs 43
8.4 Use Cases 43
8.5 Author's Assessment of the Project 43

SIMULATING THE EMERGENT AUTONOMOUS BEHAVIOUR OF UNMANNED AERIAL VEHICLE SWARMS

ANTONIO BRITO

Abstract

Path planning and control algorithms for mobile robots, specifically unmanned aerial vehicles (UAVs), which typically fall under the A* family of algorithms, have been well explored[1, 2]. Likewise, multi-agent pathfinding within the context of UAV systems have been explored, most notably by Burwell [3].

Notably, early developments in the field of swarm robotics on ground vehicles had been made[4], with the focus shifting from ground vehicles to aerial vehicles in recent years. With this has come descriptions of the technical frameworks[5] and operational challenges of UAV swarms[6].

This project will explore the feasibility of using a simpler, rule based system to simulate emergent behaviour for control of UAVs, based on an extension of the principles of Reynolds' Boids[7].

We also explore the limitations of this approach, specifically in 3D, and explore an extension to our developed algorithm which focuses on local search and a potential field approach to path planning (Local Potential Field Emergence), with the aim of reducing complexity in the parameter space of the Boids algorithm.

Through this project, we aim to provide a proof of concept for the feasibility of emergent behaviour in UAV swarms, and to provide a basis for further research in the field.

1 Context

1.1 Introduction

The use of unmanned aerial vehicles (UAVs) has been growing rapidly, with both civil and military applications. Namely, their ease of deployment, low maintenance cost, high-mobility and ability to hover mean they are well suited to a variety of tasks, such as surveillance, reconnaisance, search and rescue, and logistics[8].

Utilising multiple agents for these applications has several advantages. Firstly, it introduces redundancy into the system, allowing for effective fault tolerance, which is extremely vital in safety-critical applications[9]. A recent example of this is the use of UAV swarms in forest fire monitoring[10].

Additionally, the ability to scale the number of agents in the swarm allows for greater efficiency in surveillance and reconnaisance tasks, as well as the ability to cover a larger area in search and rescue operations over a given time period.

In Section 4, we will explore tailoring the behaviour of our agents to simulate some scenarios, based on these expected applications of UAVs. To this end, we will explore the flexibility of UAV swarms in the context of adapting to different scenarios.

Implementations for the control and path planning of mobile robotics is well studied. Namely, A* and its variants are well studied and widely used for path planning in mobile robotics[2]. Additionally, Particle Swarm Optimisation (PSO) has been used for path planning in UAV swarms[11].

This project's focus will be on a simulated implementation of Reynolds' Boids[7] algorithm, which is a rule-based system for simulating emergent behaviour in a flock of birds. It has been demonstrated that this algorithm can be used for effective autonomous control of swarms[12], but lacks in the ability for pathfinding and obstacle avoidance[11]. Hence, we will look to extend this autonomus control in a world-like environment, by implementing goal seeking, terrain avoidance and obstacle avoidance as added behaviours.

The implementation of this algorithm will be in the context of a 3D environment, with the aim of providing a proof of concept for the feasibility of emergent behaviour in UAV swarms, based on optimising the influence of the behaviours in our algorithm to align to our expected applications of UAV swarms. We will explore issues with this approach, and propose an extension to our developed algorithm which modifies the behaviours of the agents to simulate an artificial potential field, which has been shown

to be effective as a mobility model for UAV swarms[13].

1.2 Background

This proof of concept will be based in the Unity game engine, which provides a physics engine and a 3D environment for our simulation. As such, it handles collisions, rigid body dynamics and raycasting, which will be vital for our simulation. Unity has been demonstrated to handle complex agent-based simulations, even extending to the ability to perform more complex artificial intelligence tasks including reinforcement learning[14], making it a sufficient platform for this project.

This project will build upon the work of two-dimensional Boids simulations for control of UAV swarms, mainly by Madey and Madey [15] by extending the simulation to three-dimensions and adjusting the model to account for quadcopter physics. We also consider a method of combining individual behavioural rules[16].

We will optimise our combined behaviour set by using an optimisation technique called simulated annealing. This is a probabilistic technique for approximating the global optimum of a given cost function. We will use this to optimise the weights of the behaviours in our algorithm to align to our expected applications of UAV swarms.

Whilst the optimisation of individual behaviours of Boids flocking has been explored, this has typically been achieved through the use of genetic algorithms or particle swarm optimisation[1], as documented by Alaliyat et al. [17] in their comparative study of such optimisation methods.

We will note limitations with this approach, mainly within the difficulty in optimising the behaviour space.

[1]PSO is a function optimisation method, so can be used for both parameter optimisation (here) and path planning (earlier).

2 Physics Model

Here, we will discuss the physics model of our simulation, which is comprised of agents and the agent's interactions with the environment. We will also discuss the control system for the agents.

2.1 Quadcopter

For the physics model, the coordinate space is referred to using *local tangent plane coordinates*, namely that z is east, x is north and y is up. The principal axes of movement are *pitch*, along the transverse (z) axis, *roll*, along the longitudinal (x) axis, and *yaw*, along the vertical (y) axis.

A UAV can be modelled as a rigid body with mass m, on Earth. As such, gravity acts on the agent at $9.81\frac{m}{s^2}$. The drag coefficient is estimated at 0.975 considering the mass of the body [18].

The dimensions and chracteristics of the agent have been determined using both realistic averages [19] and estimations. These can be seen in Fig. 1.

Parameter	Value
Mass, m	10 kg
Length, l	0.7 m
Width, w	0.7 m
Height, h	0.25 m
Propeller Area, A	0.16 m^2
Distance To Propeller Centre, d	0.5 m
Drag Coefficient, C_d	0.975

Figure 1: Dimensions of the Agent

The agent is modelled as a quadcopter, with four propellers. These can be independently controlled to affect the movement of the agent. The body diagram of the quadcopter is shown in Fig. 2 alongside the local axes of movement. The agent as modelled in Unity is shown in Fig. 3 with the same axes.

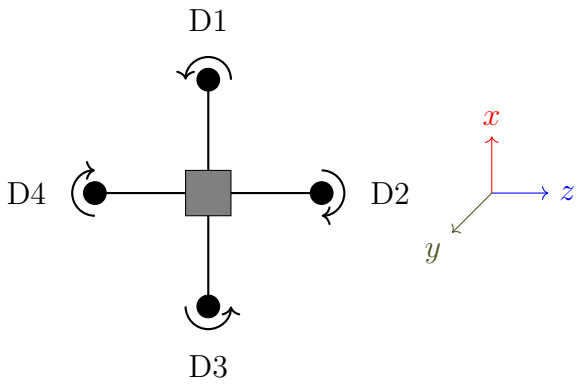


Figure 2: Quadcopter Body Diagram in the x-z plane

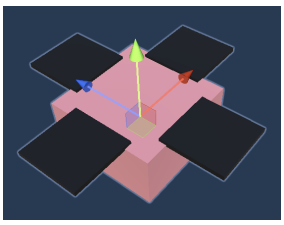


Figure 3: Agent in Unity

For this investigation, a quadcopter was chosen as the type of agent, rather than a fixed-wing aircraft, due to increased maneouverability, simplicity when considering autonomous flight and simplicity in takeoff and landing procedures [20].

There are four elements of control for an agent:

- Roll angle (in radians), ψ
- Pitch angle (in radians), θ
- Yaw rate (in $\frac{rad}{s}$), ϕ
- Vertical thrust (in Newtons), T

We can define roll (ψ) as the rotation of the agent about the x axis, pitch (θ) as the rotation about the z axis, and yaw (ϕ) as the rotation about the y axis.

For simplicity, these can be solely controlled by combinations of propeller thrust levels.

A base thrust level, T_b, can be defined, which is the minimum thrust level required to maintain a stable hover. This is the thrust level required to counteract gravity, so

$T_b = mg$. For each control operation, T_b can be augmented by a thrust level, T_{add}, such that $|T_{add}| < T_b$, which is the additional thrust required to perform the operation.

As such, the control parameters can be operated by the following eight combinations of thrust levels for each propeller, with a high thrust level $T_{add} > 0$ and a low thrust level $T_{add} < 0$:

Operation	High Thrust	Low Thrust
Positive Roll, ψ_+	D_4	D_2
Negative Roll, ψ_-	D_2	D_4
Positive Pitch, θ_+	D_3	D_1
Negative Pitch, θ_-	D_1	D_3
Positive Thrust, T_+	D_1, D_2, D_3, D_4	none
Negative Thrust, T_-	none	D_1, D_2, D_3, D_4

Table 1: Control Operations and Thrust Levels

It is noted that yaw is omitted from Fig. 1. For the purposes of the simulation, the propellers do not rotate. This lends itself to needing a workaround for simulating yaw. In *Unity*'s physics engine, the rotation of opposite propellers can be simulated by applying forces along the x and z axes. As such, yaw is simulated according to the values in 2.

Operation	Propeller	Direction of Force (respectively)
Positive Yaw, ϕ_+	D_1, D_2	Z_-, X_-
Positive Yaw, ϕ_+	D_3, D_4	Z_+, X_+
Negative Yaw, ϕ_-	D_1, D_2	Z_+, X_+
Negative Yaw, ϕ_-	D_3, D_4	Z_-, X_+

Table 2: Yaw Directions and Respective Forces

In real world physics, a couple of propellers could not provide thrust in both extremes of the same axes, as they could not switch rotation direction. Likewise, forces in adjacent propellers would not cause rotation in the same direction. However, for the purposes of simplicity within the simulation, this is ignored.

It is then possible to set some thrust constants for each control operation. An example assigment for the pitch and roll operations is seen below in Fig. 3.

Thrust Level	Thrust (N)
High	$\frac{2m \cdot g}{4}$
Normal	$\frac{m \cdot g}{4}$
Low	$\frac{0.5m \cdot g}{4}$

Table 3: Thrust Constants for a drone of mass m kg

At this stage, we have a model for the control and movement of the agent. An input is given to the propeller controller, which outputs the thrust level as required in Fig. 3 to the individual propellers, causing a rotation of the agent in the desired direction.

2.2 Stabilisation

An issue arises with this implementation; the simulation becomes unstable as thrust cannot be provided in an accurate enough manner to counteract excessive rotation, notably in the roll and pitch axes.

This can lead to the agent's rotation becoming overly large, causing it to roll over. If we consider an example input, represented by a single keypress to generate thrust to cause a pitch rotation, we can model the deviation from the desired pitch angle $\theta_d = 0.26$ over time, as shown in Fig. 4.

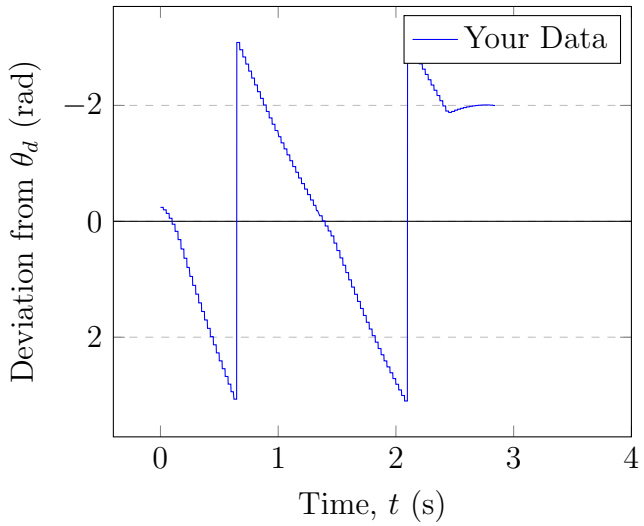


Figure 4: Pitch angle due to input over time

Fig. 4 shows that when given an input, the agent exceeds the desired pitch angle, crossing $\theta_d = 0$, then continues to rotate until it collides with the ground at $t = 3$

seconds.

As such, the need for a control system becomes apparent.

2.2.1 Proportional Controller

A first intuition here is to control the rotation by reducing the thrust level as the agent approaches the desired angle, such that the thrust level at some time, t, is inversely proportional to the deviation from the desired angle as shown in Eq. 1.

$$T_t \propto \frac{1}{\theta_d} \tag{1}$$

We can then define a proportionality constant, K_p according to the inverse relationship in Eq. 1. Using different values for K_p, we can observe some behaviours in the deviation from the desired angle over time, as shown in Fig. 5, which is split into two subfigures for clarity. The left-hand graph shows $K_p = 1$ and $K_p = 5$, and the right-hand graph shows $K_p = 5$ and $K_p = 10$.

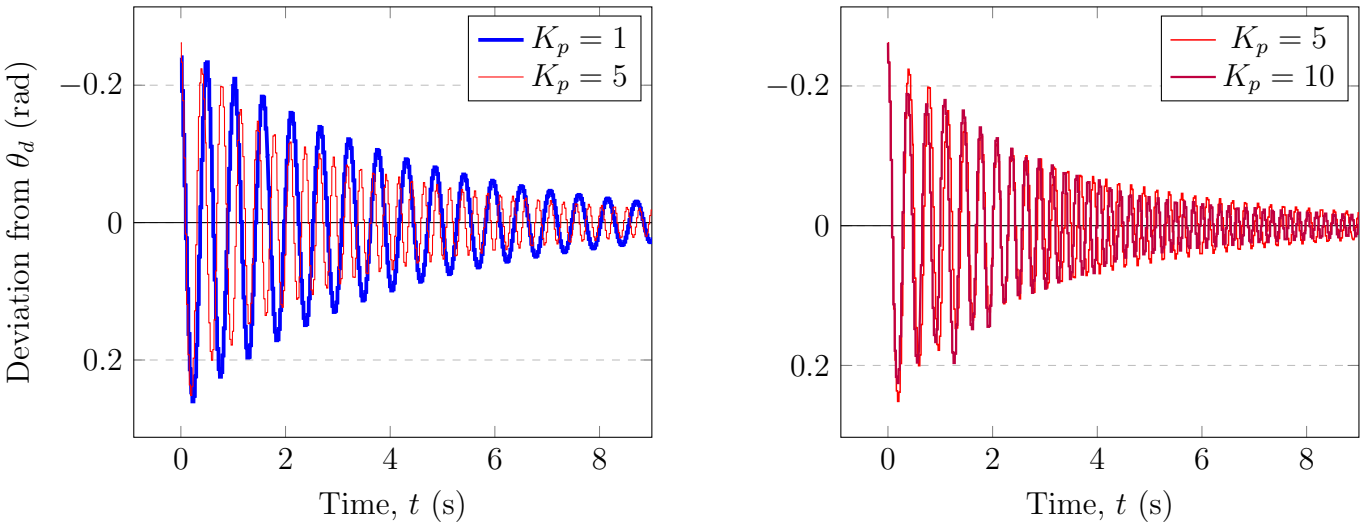


Figure 5: Pitch angle due to input over time with proportional error correction

As shown in Fig. 5, we can see that adding a proportional controller to the system creates a steady state, as we can see that the limit of θ_d as $t \to \infty$ is ≈ 0 in both graphs.

In the left-hand graph, we can see that increasing K_p decreases the amount of deviation from the desired angle. However, in the right-hand graph, we can see that this increase becomes negligible at a point, and does not reach zero. Instead, we can

observe some oscillations in this state, as the agent continually overshoots the desired angle, before correcting itself, and overshooting again, to a lesser extent each time.

In some sense, we require the agent's controller to predict overshooting the desired angle.

2.2.2 Derivative Controller

In order to predict overshooting the desired angle, we need to modify the controller to consider the rate of change of the deviation. A deviation that is changing rapidly (oscillating) will also have a large (but decreasing) rate of change of deviation.

Hence, we can aim to reduce this rate of change of deviation by adding a term to the controller that is proportional to the rate of change of deviation, namely, a derivative term. This is shown in Eq. 2.

$$T_t \propto \frac{1}{\theta_d} - K_d \frac{d\theta}{dt} \tag{2}$$

As before, we can introduce a proportionality constant, K_d, to the derivative term. We can then observe the effect of this term on the deviation from the desired angle over time, as shown in Fig. 6. We note that we keep $K_p = 5$ constant.

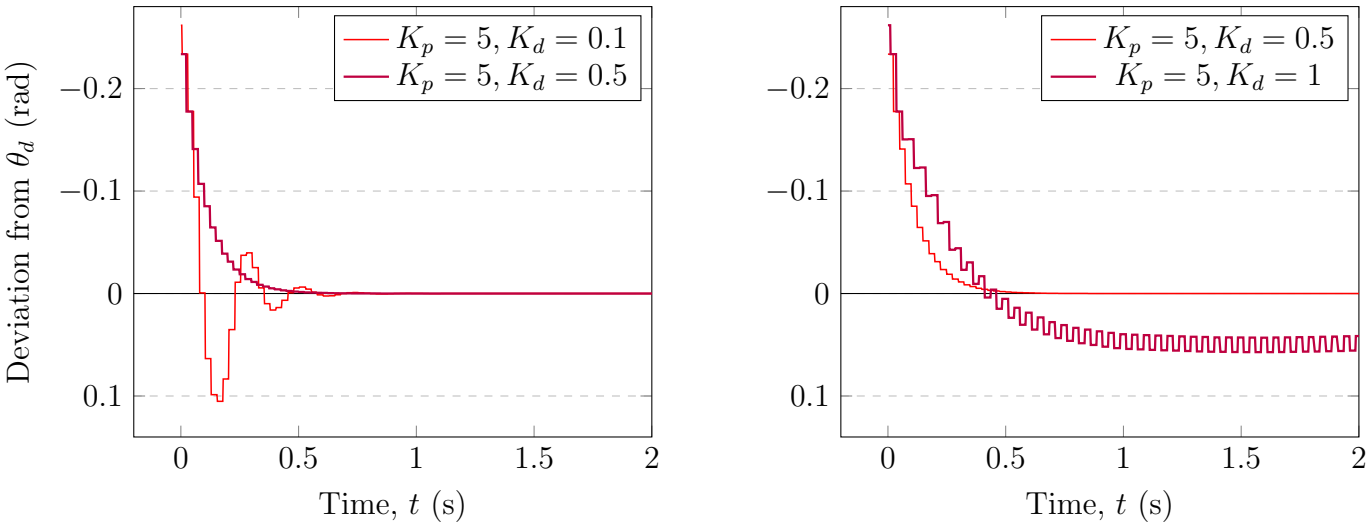


Figure 6: Pitch angle due to input over time with additional derivative error correction

Fig. 6 shows that adding a derivative controller with a small K_d reduces the length of time of oscillation before reaching the desired angle. This behaviour can be seen at $K_d = 0.1$. At $K_d = 0.5$, this behaviour is improved and the oscillations stop entirely. However,

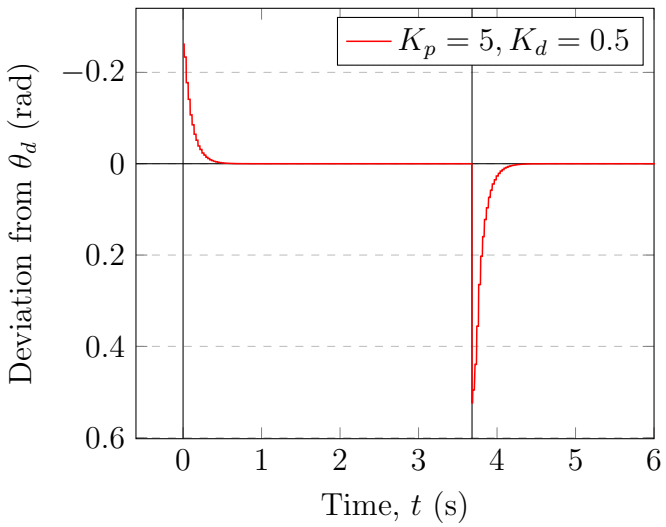


Figure 7: Pitch angle due to inputs at $t = 0$ and $t = 3.7$ seconds

at $K_d = 1$, whilst the controller achieves a steady state with minimal oscillation, it has overshot the desired angle, introducing steady state error into the system.

Furthermore, a second issue with the system has arisen; the agent's rotation is not quick enough to respond to an input. For example, we can see the responsiveness of the agent to a new input in Fig. 7, where a forward pitch command is given, as before, at $t = 0$ seconds, then a backward pitch command is given at $t = 3.7$ seconds, both of which are shown as vertical lines on the graph.

We can see that the time to settle is ≈ 0.5 seconds, which may cause issues in our simulation. When introducing multiple agents into the system, responsiveness is key, to ensure last minute direction changes can be made, for example, to avoid collisions.

2.2.3 Integral Controller

We are able to reduce this reaction time and the possibility of steady state error by attempting to reduce the area between the θ_d and the actual angle (the curve in the graphs).

Namely, we can introduce an integral term, which is proportional to the area between the θ_d and the actual angle. This is shown in Eq. 3.

$$T_t \propto \int_0^t \theta_d - \theta(\tau) d\tau \tag{3}$$

We can then observe the effect of this term on the deviation from the desired angle

over time, as shown in Fig. 8. We note that we keep $K_p = 5$ and $K_d = 0.5$ constant, and test the values $K_i = 2$ and $K_i = 5$.

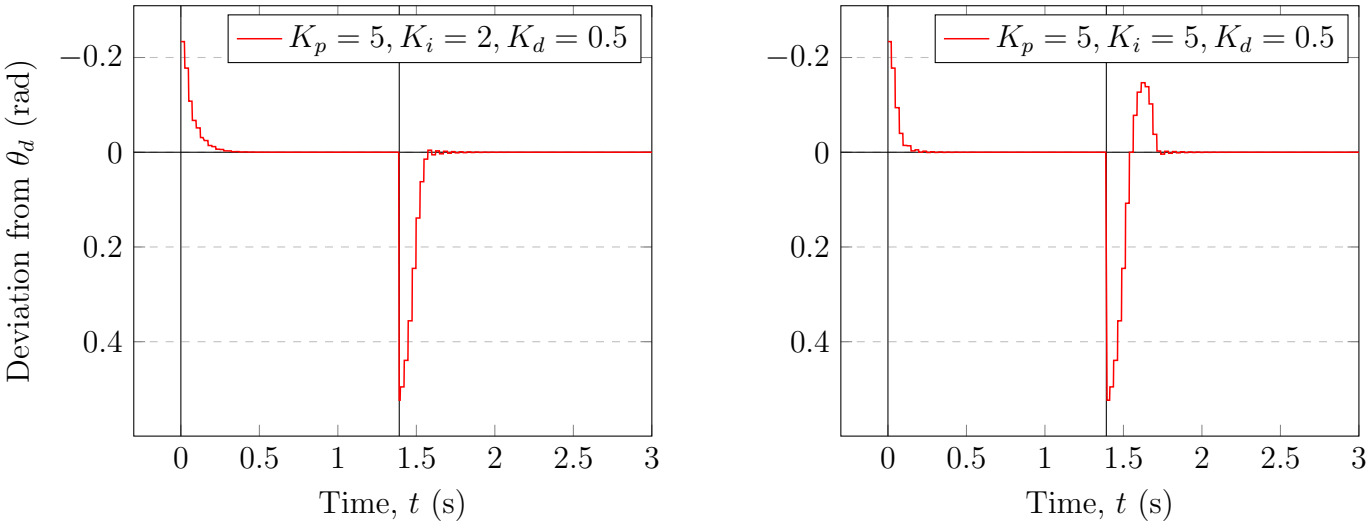


Figure 8: Pitch angle due to inputs at $t = 0$ and $t = 1.4$ seconds

We note that the time to achieve a steady state is reduced, meaning the responsiveness of the system has improved. However, we note now that with the constant $K_p = 5, K_i = 5, K_d = 0.5$, an overshoot is present. To mitigate this, we can increase K_d. This is shown in Fig. 9.

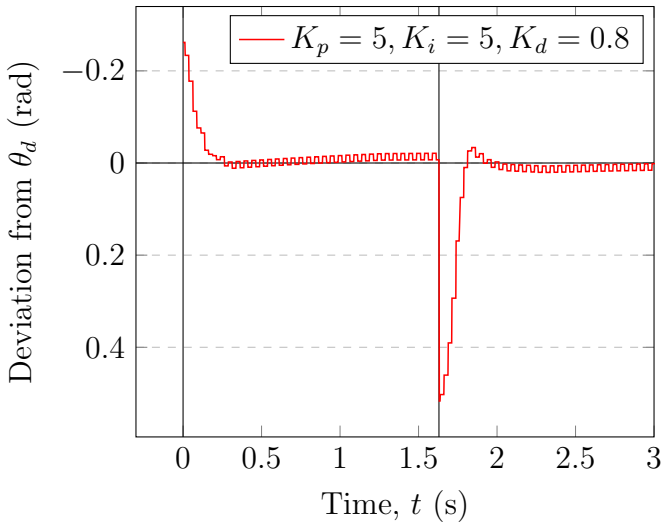


Figure 9: Pitch angle due to inputs at $t = 0$ and $t = 1.6$ seconds

Note that in Fig. 9, the overshoot is reduced, but we now have steady state and

small angle oscillation errors. We can see that tuning the PID controller is a complex task akin to a chicken-and-egg problem, which we will explore further in Section 2.2.5.

2.2.4 PID Controller

We have constructed a PID (Proportional-Integral-Derivative) system; a feedback control system which uses the error between the current state and the desired state to calculate the control parameters. We now generalise this for all possible movement operations and call the desired state r and the current state y.

Fig. 10 shows the feedback loop of the PID controller. The error, $e(t)$, is calculated as the difference between the desired state, r, and the current state, y. The error is then fed into the three controllers. The output of each controller is then summed to produce the control variable, $u(t)$, which is then fed into the system. The system then produces the output, y, which is fed back into the error calculation.

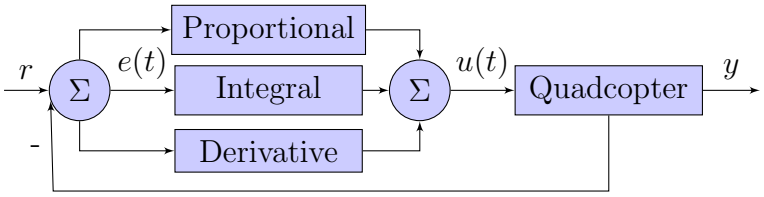


Figure 10: PID Controller

The PID controller is defined by three constant parameters derived earlier, K_p, K_i and K_d, which are the proportional, integral and derivative gains respectively. The control variable, $u(t)$, is then defined as:

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau)d\tau + K_d \frac{de(t)}{dt} \tag{4}$$

where $e(t)$ is the error at time t. Considering our simulation operates in discrete time intervals (frames), this can be approximated, using the *Euler method* as:

$$u(t) = K_p e(t) + K_i \frac{(e_t + e_{t-1})t}{2} + K_d \frac{e_t - e_{t-1}}{t} \tag{5}$$

Hence, a feedback loop of the entire system can be produced, taking into account different PID controllers for each control operation. This is shown in Fig. 11.

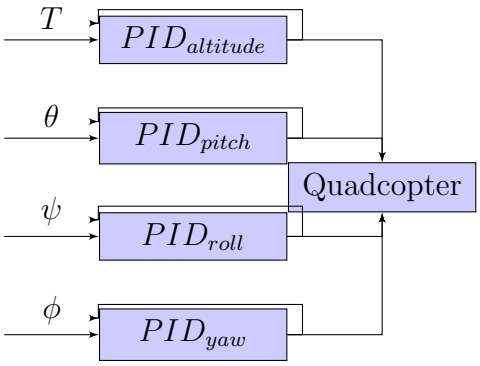


Figure 11: System Model

2.2.5 PID Parameter Tuning

As noted previously in Section 2.2.3, tuning the PID controller is a complex task and often suboptimal[21]. We require the PID controller to be stable, responsive and have minimal steady state error to ensure the agents are responsive and accurate in their movements.

Empirical Method Using a manual tuning method, we can observe the behaviour of the system and adjust the constants accordingly. A set of sufficient values were determined as shown in Fig. 4.

Control Operation	K_p	K_i	K_d
Thrust	6	5	2
Pitch	10	10	2
Roll	10	10	2
Yaw	10	10	2

Table 4: Tuning Constants

It is noted that the agent is slightly unstable at very small angles. This is negligible.

Ziegler-Nichols Method The Ziegler-Nichols method is a heuristic method for tuning PID controllers. It is based on the response of the system to a step input.

McCormack and Godfrey [22] propose a set of tuning rules for a classic control system, which result in the proportional gain K_p and two values representing the integral and derivative time constants, T_i, T_d, respectively. An extract from *Table 1* of the paper is shown in Table 5.

Tuning Rule	Required	Controller Parameters
ZN	K_u, T_u	$K_p = 0.6K_u, T_i = 0.5T_u, T_d = 0.125T_u$

Table 5: Ziegler-Nichols Method

The term K_u refers to the optimal gain; the gain at which the system oscillates at a constant amplitude and frequency. The term T_u refers to the time period of this oscillation.

Fig. 12 shows the oscillation of the system following a positive pitch command with K_p increasing from 1 at a rate of 1 every second. We can then determine the opimal gain to be the point at which the oscillations become stable. This is shown to be circa $K_p = 7$. A zoomed in version of the plot shows the time period of this oscillation to be $\approx T_u = 0.6$ seconds.

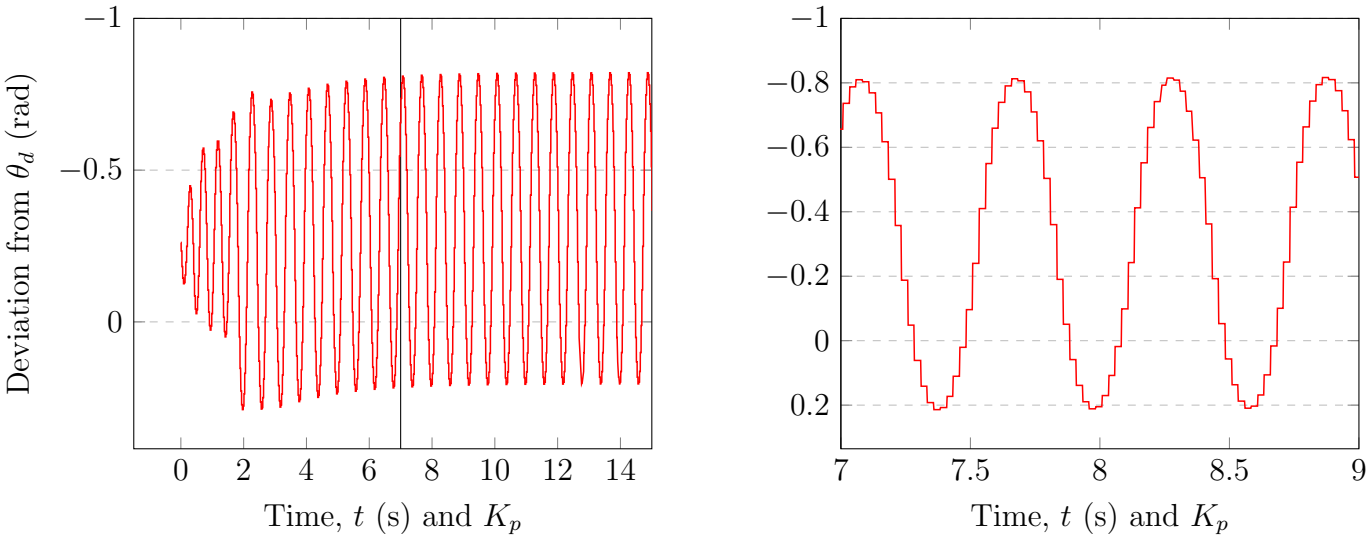


Figure 12: Oscillation of the system with varying K_p

We can then use Table 5 to determine the controller parameters. We find that $K_p = 4.2, T_i = 0.3, T_d = 0.075$.

At this stage, we can determine the values for the integral and derivative gains from the method, which states the relationships shown in Equation 6.

$$K_i = \frac{K_p}{T_i} \quad \text{and} \quad K_d = K_p T_d \tag{6}$$

Hence, we find that $K_i = 14, K_d = 0.315$.

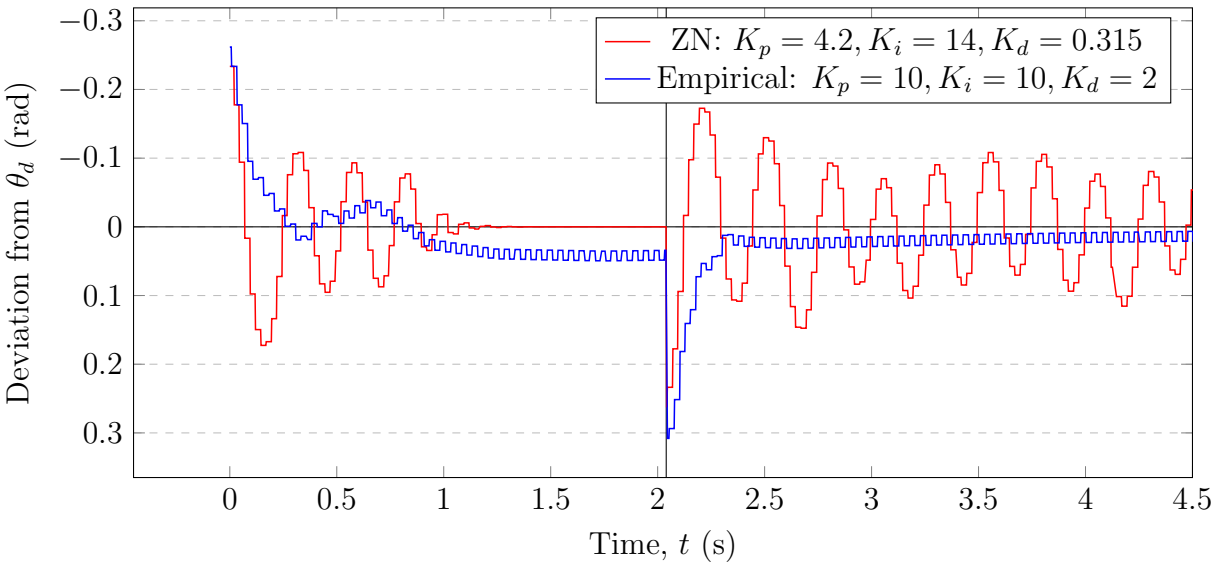


Figure 13: Pitch angle due to inputs at $t = 0$ and $t = 2.03$ seconds

We can compare this behaviour with the values obtained from the empirical method described earlier. When given a positive pitch command to obtain the desired angle $\theta_d = 0.26$ radians, then a release of this command to reset the desired angle to $\theta_d = 0$, we can observe and compare the behaviours for both sets of constants. This is shown in Fig. 13.

We can see that the empirical method results in a faster settling time, but has a small steady state error, which is not present in the results of the Ziegler-Nichols method. However, the Ziegler-Nichols method results in a small oscillation centered at the desired angle.

As the Ziegler-Nichols method is a heuristic method, it is not guaranteed to be optimal. However, it is a good starting point for tuning the PID controller. We can see from our comparison that an ideal set of constants lies somewhere between the results from both methods. We can improve on our current results by observing the behaviour of the system and adjusting the constants accordingly.

Namely, we can see that the small oscillation present in the Ziegler-Nichols method can be reduced by increasing the derivative gain, such that $K_d = 0.9$. This should also mitigate the small steady-state error present from the empirical method. To reduce this further, we will adjust the proportional gain such that $K_p = 8$ and the integral gain such that $K_i = 12$.

This is shown in Fig. 14.

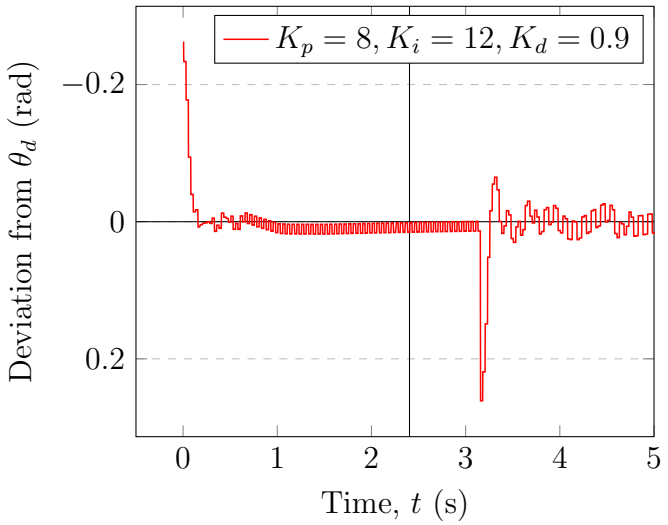


Figure 14: Pitch angle due to inputs at $t = 0$ and $t = 1.6$ seconds

Conclusion Fig. 14 shows a tuned PID pitch controller, derived from a combination of empirical and heuristic methods. We will then extend these values to our other control operations, roll and yaw, and keep the thrust constants that we obtained through the empirical method for simplicity.

The final PID constants are shown in Table 6.

Control Operation	K_p	K_i	K_d
Thrust	6	5	2
Pitch	8	12	0.9
Roll	8	12	0.9
Yaw	8	12	0.9

Table 6: Final PID Constants

However, steady-state error is still present in the system. We can continue to use advanced methods to attempt to tune this, however, for the purposes of this project, we will consider this to be sufficient.

As such, we have completed our design of the agent physics model.

2.3 Environment Model

We will now discuss how the agents interact with each other and objects in the environment according to the Unity physics engine.

Agents may collide with any object in the environment defined to have a ‘collider’. This is a built-in Unity component that defines the shape of the object and is used to detect collisions. The agent’s collider is comprised of the colliders of its parts, namely the agent’s body and its rotors.

Likewise, all objects we instanitate from this point onwards, such as the terrain, obstacles and platforms, will have colliders.

We will specify that the agents are resistant to collisions with other agents.

3 Introducing Autonomy

The initial aim is to achieve position control for an agent, such that when given a current position $p_0 = [x_0, y_0, z_0]$ and a goal position $p_{goal} = [x_1, y_1, z_1]$, the agent will move to the goal position. This sets the foundation for the behaviour of the agents; we will model these as a 'desire' to move to a specific position.

There are several methods for this. An agent may yaw to face the goal, then pitch towards it. In the case of this project, time-sensitive position decisions will need to be made when considering swarm dynamics and collision avoidance, so a more efficient method is required. As such, the aim will be to pitch and roll towards the position.

3.1 Movement to Position

We can determine the input commands required for an agent to reach p_{goal} from p_0 by performing vector calculations.

Firstly, we can model these two points on a coordinate grid. For representational clarity, we will assume the model to be in two dimensions for now. We can then calculate the vector between the two points, $v_g = p_{goal} - p_0$. We can see that the vector subtraction represents the transformation vector required to reach the goal position.

This is shown in Fig. 15.

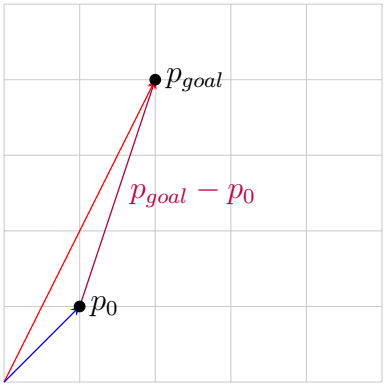


Figure 15: 2D Coordinate Grid with p_{goal} and p_0

We then require the agent to pitch and roll towards the goal position. This requires a change of basis between the global frame (as represented in 15) and the agent's local frame. This is because the agent's yaw component means that the agent's pitch and roll directions will not cause a movement along the global axes, except the trivial case, where the agent is lined up with the global axes.

We can determine the transformation required to convert the global vector v_g into the local frame, resulting in the vector v_l by calculating the rotation matrix between the two frames of reference.

We can initially do this in two dimensions by determining a transformation matrix M_2 such that:

$$\begin{bmatrix} x_l \\ y_l \end{bmatrix} = M_2 \cdot \begin{bmatrix} x_g \\ y_g \end{bmatrix} \tag{7}$$

We note that in the two-dimensional case, M_2 need only be a roation matrix representing the yaw of the agent. We can derive M_2 as follows. We first consider a vector in the global frame, v_g and the local frame v_l. We note that their magnitudes are equal, with $|v_g| = |v_l|$. We can then determine the angle between the two vectors, β, and the angle between the global x-axis and the vector v_g, α.

This is shown in Fig. 16.

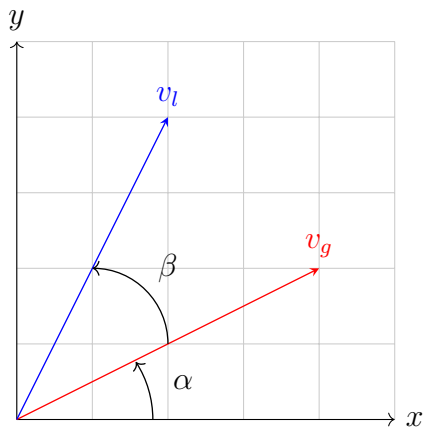


Figure 16: 2D Coordinate Grid with v_l and v_g

Hence, we can determine that the x and y components of v_g are:

$$\begin{aligned} x_g &= |v_g| \cos(\alpha) \\ y_g &= |v_g| \sin(\alpha) \end{aligned} \tag{8}$$

It must then hold that:

$$
\begin{aligned}
x_l &= |v_l| \cos(\alpha + \beta) \\
y_l &= |v_l| \sin(\alpha + \beta) \\
&\implies \\
x_l &= |v_g| \cos(\alpha)\cos(\beta) - |v_g| \sin(\alpha)\sin(\beta) \\
y_l &= |v_g| \sin(\alpha)\cos(\beta) + |v_g| \cos(\alpha)\sin(\beta) \\
&\implies \\
x_l &= x_g \cos(\beta) - y_g \sin(\beta) \\
y_l &= x_g \sin(\beta) + y_g \cos(\beta)
\end{aligned}
\tag{9}
$$

We can represent this as a matrix equation:

$$
\begin{bmatrix} x_l \\ y_l \end{bmatrix} = \begin{bmatrix} \cos(\beta) & -\sin(\beta) \\ \sin(\beta) & \cos(\beta) \end{bmatrix} \begin{bmatrix} x_g \\ y_g \end{bmatrix}, \text{ where } M_2 = \begin{bmatrix} \cos(\beta) & -\sin(\beta) \\ \sin(\beta) & \cos(\beta) \end{bmatrix} \tag{10}
$$

Returning to three dimensions, we can represent our problem as determining a transformation matrix M_3, such that:

$$
\begin{bmatrix} x_l \\ y_l \\ z_l \end{bmatrix} = M_3 \cdot \begin{bmatrix} x_g \\ y_g \\ z_g \end{bmatrix} \tag{11}
$$

In the three-dimensional case, M_3 will be a rotation matrix representing the yaw, pitch and roll of the agent. As such, it will be comprised of three individual rotation matrices for each axis of rotation. Hence:

$$
M_3 = R_y \cdot R_x \cdot R_z \tag{12}
$$

We can then derive the rotation matrix R_x in a similar method to M_2.

We note that R_x will represent a rotation about the x-axis, namely, the roll of the agent. This means that the y and z components of the vector v_g will be rotated through an angle of ψ, whilst the x component will remain constant. We can then determine the rotation matrix from earlier intuition as:

$$
R_x = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\psi) & -\sin(\psi) \\ 0 & \sin(\psi) & \cos(\psi) \end{bmatrix} \tag{13}
$$

Considering the other two rotation matrices, R_y and R_z, where the xz and xy planes are rotated respectively, we can see:

$$\begin{aligned} R_y &= \begin{bmatrix} \cos(\theta) & 0 & -\sin(\theta) \\ 0 & 1 & 0 \\ \sin(\theta) & 0 & \cos(\theta) \end{bmatrix} \\ R_z &= \begin{bmatrix} \cos(\phi) & \sin(\phi) & 0 \\ -\sin(\phi) & \cos(\phi) & 0 \\ 0 & 0 & 1 \end{bmatrix} \end{aligned} \tag{14}$$

At this stage, we can split the local vector v_l into its constituent pitch and roll components, which can be applied to the agent as input commands.

We see that we may obtain the pitch and roll components of the vector v_l by taking the x and z components respectively. This is shown in Fig. 17.

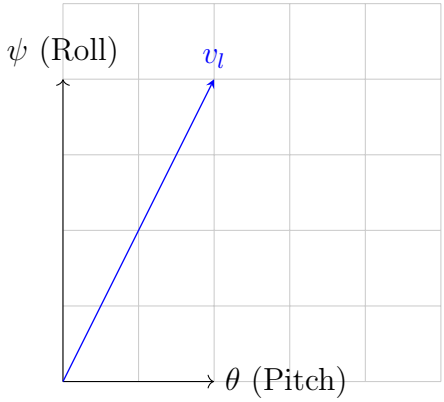


Figure 17: Constituent Pitch and Roll Components of v_l

Usefully, this conversion step can be performed in the Unity physics engine using the built-in method `transform.InverseTransformVector()`, which will convert the vector from the global frame to the local frame of the agent.

3.2 Boids (1987)

At this stage, we introduce multiple agents into the environment. We will initially consider the agents to be autonomous, and have the ability to hover by default.

We will make some assumptions about the awareness of the agents at this stage and provide some important definitions. These are:

- Agents are aware of the positions of all other agents within a certain radius, r. This is defined as the *neighbourhood radius*.
- Agents are aware of their own position with respect to the global reference frame.
- Agents in a flock of size n are defined as $a_1, ..., a_n \in A$.

The motivation behind the use of the neighbourhood radius is to reduce computational complexity. Generally, an increase in the number of adjacent agents will result in an increase in the number of calculations required. We can reduce this by not including the agents that have no effect on the behaviour of the agent in question.

We will then introduce the concept of *Boids* [7]. This is a method of simulating flock behaviour, where agents are programmed to exhibit a set of behaviours that result in emergent flock behaviour. We will explore the implementation of these behaviours in the next section.

For all behaviours, we calculate a force vector that is applied to the agent, with respect to the agent's local frame.

3.2.1 Avoidance

The avoidance behaviour is used as a method for agents to avoid collisions with each other. To this end, the behaviour causes agents to move away from each other when they are within a certain distance of each other.

We will call this distance d, and define it as the *square avoidance radius*. This is the distance at which agents will begin to move away from each other. We will define this as the square of the distance between two agents, $a_1, a_2 \in A$, such that $d = (|a_1| - |a_2|)^2$, for the purposes of algorithmic efficiency. This is a common method in the *Unity* physics engine [23].

We can then determine the avoidance vector for an agent, a, by calculating the mean distance vector between a and all other agents within the neighbourhood, a_i, such that $|a_i| - |a| < d$.

Specifically, we populate the set R_a with all agents $a \in A$ that are within the neighbourhood of a.

$$\vec{A} = \begin{cases} 0 & \text{if } R_a = \emptyset \\ \frac{1}{|R_a|} \sum_{a_i \in R_a} \vec{a_i} - \vec{a} & \text{otherwise} \end{cases} \tag{15}$$

We can model an example with two agents in two-dimensional space. This can be seen in Fig. 18. We see the two agents a_1 and a_2 have position vectors $[2, 2]$ and $[3, 3]$ respectively.

Hence, the avoidance vectors for the two agents are:

$$\begin{aligned} \vec{a^1_{avoid}} &= \frac{1}{1}\left(\begin{bmatrix}3\\3\end{bmatrix} - \begin{bmatrix}2\\2\end{bmatrix}\right) = \begin{bmatrix}1\\1\end{bmatrix} \\ \vec{a^2_{avoid}} &= \frac{1}{1}\left(\begin{bmatrix}2\\2\end{bmatrix} - \begin{bmatrix}3\\3\end{bmatrix}\right) = \begin{bmatrix}-1\\-1\end{bmatrix} \end{aligned} \tag{16}$$

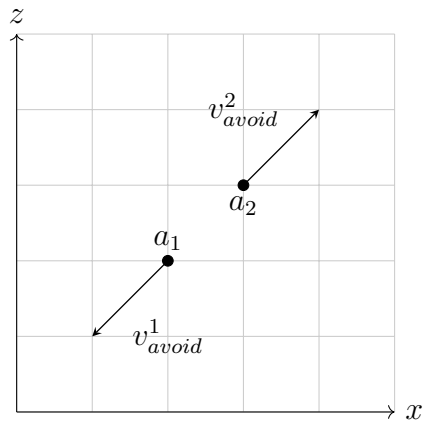


Figure 18: Avoidance vectors for two nearby agents

3.2.2 Alignment

The alignment behaviour causes agents to align their headings with those of their neighbours. This may, in turn, reduce collisions. We can determine the alignment vector for an agent, a, by calculating the mean heading vector between a and all other agents within the neighbourhood, a_i, such that $|a_i| - |a| < r$.

Specifically, we have:

$$\vec{B} = \begin{cases} 0 & \text{if } R_b = \emptyset \\ \frac{1}{|R_b|}\sum_{a_i \in R_b} h(\vec{a_i}) & \text{otherwise} \end{cases} \tag{17}$$

We can model three agents in two-dimensional space. This can be seen in Fig. 19. We see the three agents a_1, a_2 and a_3 have heading vectors $[1, 1]$, $[1, 0]$ and $[0, 1]$ respectively.

We introduce three agents as the simplest stable state of the system. Using only two agents would result in a system where the heading vectors would oscillate between the two agents without any external forces.

We can then determine the alignment vectors for the three agents as follows:

$$\begin{aligned} \vec{a^1_{align}} &= \frac{1}{2}\left(\begin{bmatrix}1\\0\end{bmatrix} + \begin{bmatrix}0\\1\end{bmatrix}\right) = \begin{bmatrix}0.5\\0.5\end{bmatrix} \\ \vec{a^2_{align}} &= \frac{1}{2}\left(\begin{bmatrix}1\\1\end{bmatrix} + \begin{bmatrix}0\\1\end{bmatrix}\right) = \begin{bmatrix}0.5\\1\end{bmatrix} \\ \vec{a^3_{align}} &= \frac{1}{2}\left(\begin{bmatrix}1\\1\end{bmatrix} + \begin{bmatrix}1\\0\end{bmatrix}\right) = \begin{bmatrix}1\\0.5\end{bmatrix} \end{aligned} \tag{18}$$

We can visualise the current heading vectors at time t and the determined heading vectors for time $t+1$ (indicated in red) in Fig. 19[2], where we see the new heading vectors have less variance in their angles.

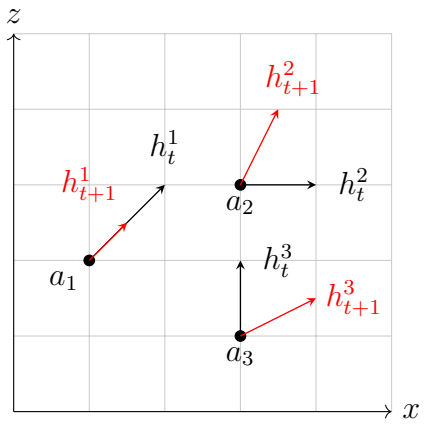


Figure 19: Alignment vectors for three nearby agents

3.2.3 Cohesion

Finally, we implement the third principle, *cohesion*. This is in principle, the inverse to the avoidance principle. It ensure members of the flock stay together. We can determine the cohesion vector for an agent, a, by calculating the mean position vector between a and all other agents within the neighbourhood, a_i, such that $|a_i| - |a| < r$.

Specifically, we have:

[2]N.B. The agents will likely not be in the same position after one timestep. We consider this as negligible in the visualisation.

$$\vec{C} = \begin{cases} 0 & \text{if } R_c = \emptyset \\ \left(\frac{1}{|R_c|} \sum_{a_i \in R_c} \vec{a_i}\right) - \vec{a} & \text{otherwise} \end{cases} \tag{19}$$

We consider three agents a_1, a_2 and a_3 in two-dimensional space with position vectors $[1, 2]$, $[3, 3]$ and $[3, 1]$ respectively. We can then determine the cohesion vectors for the three agents as follows:

$$\begin{aligned} \vec{a^1_{cohere}} &= \frac{1}{2}\left(\begin{bmatrix}3\\3\end{bmatrix} + \begin{bmatrix}3\\1\end{bmatrix}\right) - \begin{bmatrix}1\\2\end{bmatrix} = \begin{bmatrix}2\\0\end{bmatrix} \\ \vec{a^2_{cohere}} &= \frac{1}{2}\left(\begin{bmatrix}1\\2\end{bmatrix} + \begin{bmatrix}3\\1\end{bmatrix}\right) - \begin{bmatrix}3\\3\end{bmatrix} = \begin{bmatrix}-1\\-1.5\end{bmatrix} \\ \vec{a^3_{cohere}} &= \frac{1}{2}\left(\begin{bmatrix}1\\2\end{bmatrix} + \begin{bmatrix}3\\3\end{bmatrix}\right) - \begin{bmatrix}3\\1\end{bmatrix} = \begin{bmatrix}-1\\1.5\end{bmatrix} \end{aligned} \tag{20}$$

Modelling the two agents, we can see the effect of cohesion on the determined cohesion vectors in Fig. 20.

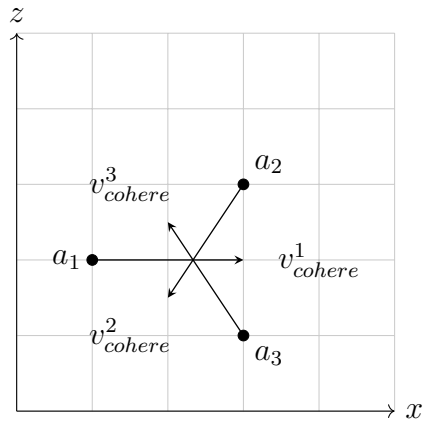


Figure 20: Alignment vectors for three nearby agents

3.2.4 Combining Behaviours

Fig. 20 demonstrates that the balance between cohesion and separation is important. If the cohesion force is too strong, the agents will collide, as shown. If the separation force is too strong, the agents will become disconnected and fly apart. This is similar in principle to the *strong nuclear force* acting within the nucleus of an atom. The force is repulsive at close ranges (the avoidance radius), but attractive at larger ranges (the

neighbourhood). We will explore this further in Section 6.3.

In general, tuning the influence of all three components is challenging. Even considering the constants in the simulation, Reynolds [7] notes that the relative weights that influence the strength of each component '*is a precarious interrelationship that is difficult to adjust*'. Despite this, we can combine these three independent behaviours and attempt to tune them.

We can define a *composite behaviour*, which is defined as a linear combination of weights and behaviour vectors. In three dimensions, for the three behaviours, we have:

$$\vec{V} = \alpha \begin{bmatrix} x_{avoid} \\ y_{avoid} \\ z_{avoid} \end{bmatrix} + \beta \begin{bmatrix} x_{align} \\ y_{align} \\ z_{align} \end{bmatrix} + \gamma \begin{bmatrix} x_{cohere} \\ y_{cohere} \\ z_{cohere} \end{bmatrix} \tag{21}$$

The method for tuning the constants α, β, γ is empirical; we can observe the behaviour of the agents and adjust the values accordingly. We introduce the method for this below.

Initially, we will set the constants as shown in Table 7. These may be adjusted later. Additionally, we will set the weights for each component as shown in Table 8.

Constant	Value
Number of Agents	8
Spawn Radius	4
Neighbour Radius	1
Square Avoidance Radius	16

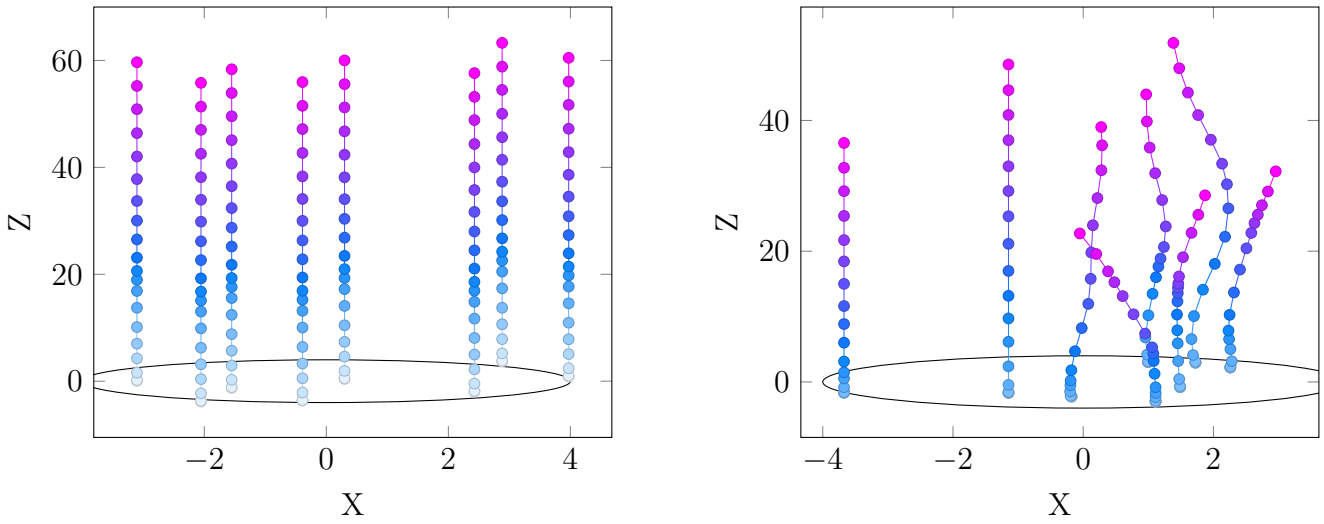
Table 7: Spawn Constants

Behaviour	Weight
Separation	1
Alignment	1
Cohesion	1

Table 8: Boids Weights

We can then simulate a run with these constants, culimating in the eight agents' X and Z positions within the space of 10 seconds, updated every 0.5 seconds. The spawn radius is also shown by the black circle. This is shown in Fig. 21a. When discussing the behaviour of individual agents, we will refer to them by their X-coordinate upon despawning at $t = 10$, left to right. We see in Fig. 21a that the agents do move together, in a straight line, signifying that the alignment weight is initially well-tuned. However, we notice the behviour of Agents 5 and 6. These agents spawn outside of each other's neighbour radius, so will never make an effort to move towards each other. Additionally, the avoidance weight is not strong enough to pull them towards each other (as a result of being replused away from Agents 4 and 7, respectively). We will

modify the neighbourhood radius to **3** and increase the weight of avoidance to **1.5**. This simulation results in the positions in Fig. 21b.



(a) Initial

(b) Avoidance Tuning

Figure 21: Simulations

Note that the agents intially move away from each other - this is due to the increased effects of avoidance. We note, however, that Agents 1 and 2 do not appear to be pulled together, despite being within each other's neighbourhood. To mitigate this, we can increase the weight of cohesion to **1.4** and reduce the square avoidance radius to **12**. Finally, we can increase the weight of alignment slightly, to **1.2**, to reduce the effects of agents veering at the sides of the flock. This simulation results in the positions in Fig. 22.

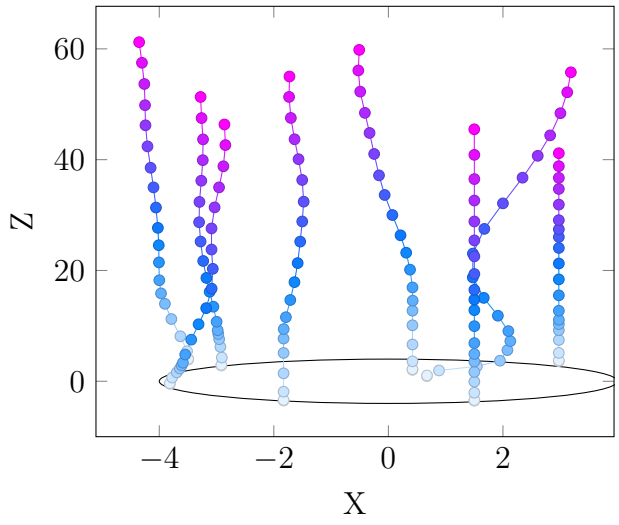


Figure 22: Final Tuning

The results from each stage of the tuning process have their strengths; for our purposes, the final tuning is the most appropriate due to their responsiveness within the flock.

3.3 Extended Boids

We have implemented the three behaviours of the Boids algorithm. We can now extend this to include a fourth behaviour, *goal seeking*. This will allow us to simulate the agents' ability to seek a goal position within the world. We will discuss the implementation of this below.

3.3.1 Goal Seeking

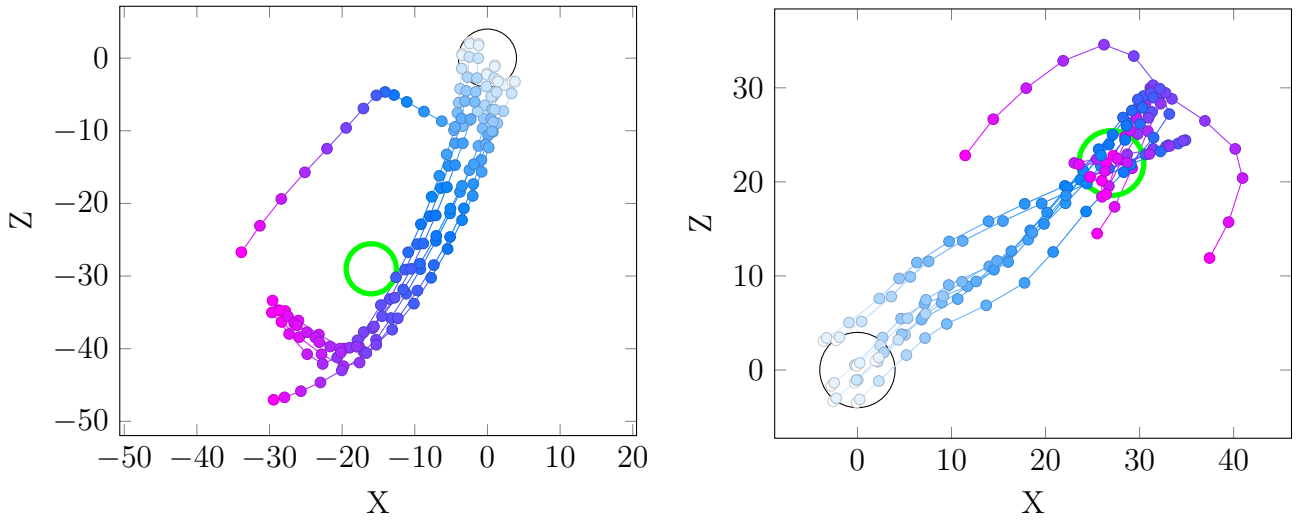
We can implement the agents' ability to seek for a goal position in a similar method to the three behaviours Reynolds mentions. We can define a *goal sphere*, with a given radius, centred around a random position in the world. We can then implement the *seeking* behaviour to calculate the distance vector between the agent and the goal.

Specifically, we have:

$$\vec{D} = \vec{g} - \vec{a} \tag{22}$$

We can then apply a force vector in the direction of the goal. At present, we choose the weight of the seeking behaviour to be 2, so that it is most prominent. This results

in some interesting 'orbiting' behaviour. The location of the goal is shown by the green circle in Fig. 23a. We note that the agents do not actually enter the goal radius. This behaviour was likely down to the method being used before to implement the agent's moves. The agent would receive the given force vector, then normalise it - acting as a maximum clamping value for the force. This turned out to not be best practice, as whilst in close spaces, the direction of the desired goal *(e.g. avoidance of nearby neighbours)* would be more important than the magnitude of the force, at higher distances between the current and goal position of the agent, the magnitudes, specifically of the pitch and roll components, become more significant in path planning. As an alternative, we simply use Unity's built in `Math.Clamp` function to set the minimum and maximum foce values to ones we deem safe. An example simulation is seen below in Fig. 23b.



(a) Normalised Force Vector

(b) Clamped Force Vector

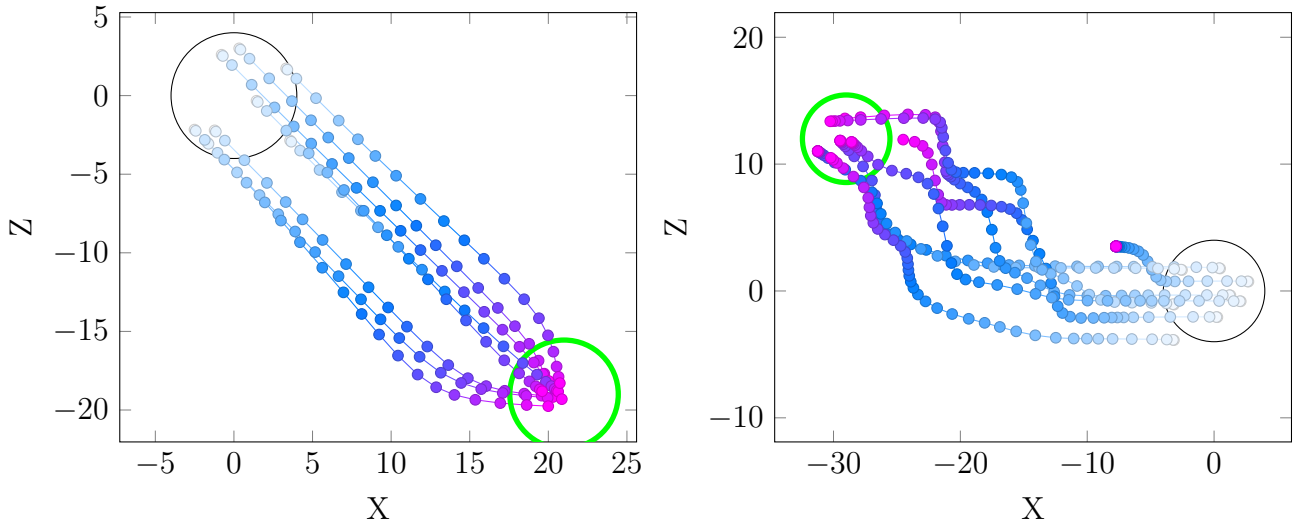
Figure 23: Seeking Behaviour

The behaviour now seen in Fig. 23b is accurate, but not ideal, as the agents overshoot the goal and find themselves backtracking during pathfinding. This can be mitigated by implementing an Error Controller, using the fundamentals of control theory discussed in Section 2.2. This is done by modifying the force vector such that it reduces with the distance between the agent and the goal position.

We can then instruct the agents to stop once they have detected that they are within the goal space. An example simulation of this is shown in Fig. 24a. We note that the agents gravitate towards the centre of the goal space. This is unwanted, as it does not make enough considerations for adjacent agents and their respective avoidance radii. At

this point, we can then reintroduce our three initial behaviours and tune the influence of *seeking* accordingly. an example simulation of this can be seen in

At this stage, we note an important flaw with our current model. The weights of the individual behaviours, ideally, should not stay constant throughout the duration of flight. For example, when the agents are far away from the goal, the seeking behaviour should be more prominent. However, when the agents are close to the goal, the agents should be more keen to avoid each other. We introduce alternative models in Section.



(a) Final Seeking Behaviour

(b) Three Initial Behaviours + Seeking

Figure 24: Finalising Seeking Behaviour

3.3.2 Terrain Generation

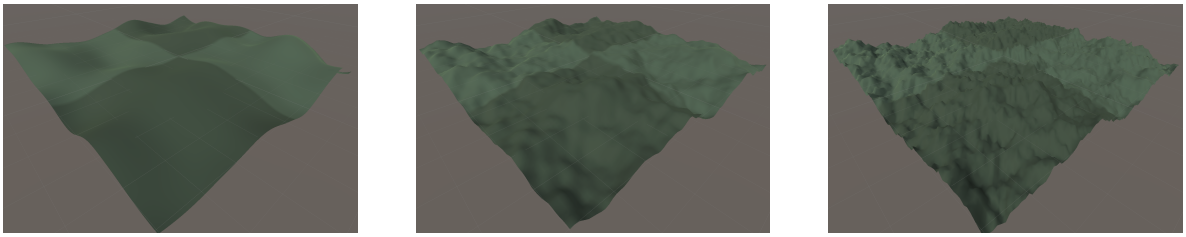
In previous sections, we have focused on the pathfinding behaviour of the agents in an empty space. However, in the real world, there are many obstacles that the agents must avoid. In this section, we introduce a terrain model to the simulation to parallel real world deviations in terrain heights. This is done using Perlin Noise[24, 289-293]. Specifically, for our implementation, we will look at generating a waveform with frequency f and amplitude A to model the terrain. We can then use this waveform to generate a heightmap for our terrain mesh.

We will consider four parameters which characterise how our terrain is generated. The number of octaves, O_n, lacunarity, l, persistence, p, and scale, s. An octave is a measure of the ratio bewteen frequencies in a waveform. Accordingly, the lacunarity, l, controls the increase in frequency between octaves; an increase in lacunarity results in

an increase in frequency between octaves. This is characterised by the realtionship:

$$\forall x \in \{O_1, O_2, ..., O_n\}, f_x = l^{O_x}$$

Lacunarity, then, allows us to tune the increase in detail between octaves. We can see in Fig. 25, with four octaves, the effect of lacunarity on the generated terrain. We can see that as the lacunarity increases, the terrain becomes more 'spiky'.



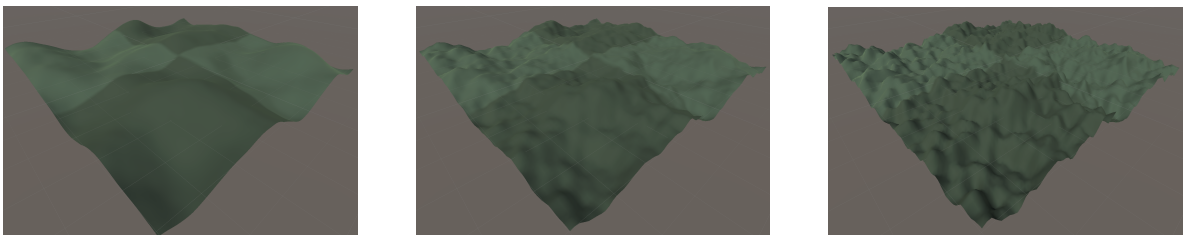
(a) $l = 1$ (b) $l = 2$ (c) $l = 4$

Figure 25: The effect of lacunarity on the generated terrain

The persistence, p, controls the decrease in amplitude between octaves. This allows us to tune how much each subsequent octave decreases in contribution to the overall waveform. This is characterised by the relationship:

$$\forall x \in \{O_1, O_2, ..., O_n\}, A_x = p^{O_x}$$

We can see below, in Fig. 26, the effect of persistence on the generated terrain. We can see that as the persistence increases, the height of the smaller details in the terrain increases. As we have kept the lacunarity constant, the spikes are not as evident as with Fig. 25c.

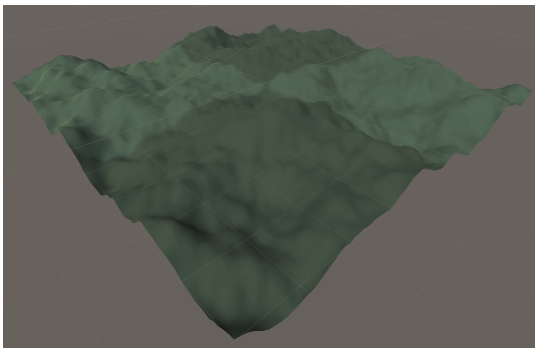


(a) $p = 0.2$ (b) $p = 0.5$ (c) $p = 0.8$

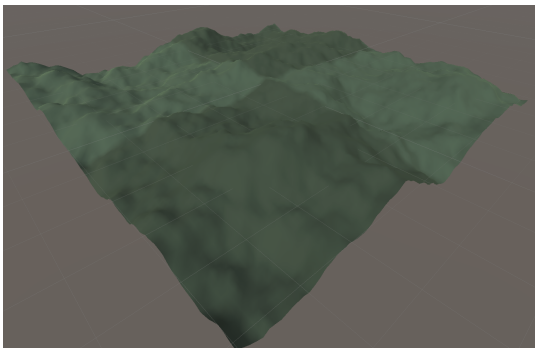
Figure 26: The effect of persistence on the generated terrain

Finally, the scale, s, contols the level of detail in the waveform that is applied to the terrain. We see in Fig. 27, the effect of scale on the generated terrain. We can see that

as the scale increases, the terrain becomes more 'zoomed out', and the details become less prominent. Notably, we can see a mountain ridge in the quadrant closest to the camera increase in size and distance away from the origin[3] between Fig. 27a and Fig. 27b.



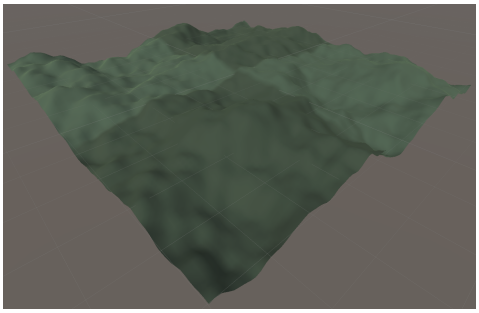
(a) $s = 25$



(b) $s = 28$

Figure 27: The effect of scale on the generated terrain

After tuning the parameters based on a combination of empirical methods and noted examples [25, pg.3-4], the following parameters were chosen, with the final terrain mesh shown alongside:



(a) Final Terrain Mesh

Parameter	Value
Number of Octaves	4
Lacunarity	2
Persistence	0.5
Scale	25

(b) Perlin Noise Parameter Values

Figure 28

3.3.3 Terrain Avoidance

Our final additional behaviour, then, will be to implement terrain avoidance. This will allow the agents to avoid the terrain mesh, and will be a crucial addition to the agents' autonomy. We will discuss the implementation of this below.

[3]The origin is the top left (the quadrant furthest away from the camera) of the terrain mesh.

At present, we do not have a method to 'sense' terrain. We can implement a simple method to detect terrain by casting a ray downwards from the agent's position, similar to a ranging sensor such as a LiDAR scanner.

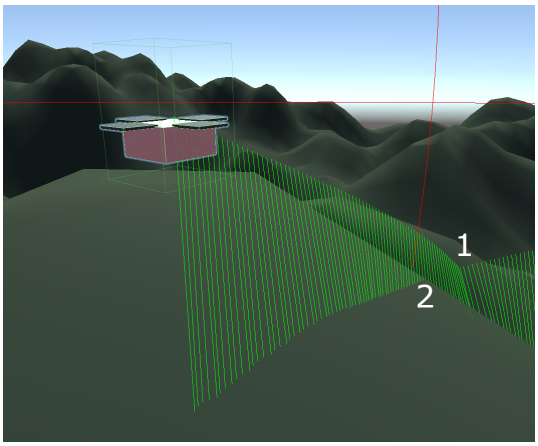
We can then perform terrain avoidance by generating upward thrust, u to cause an increase in the agent's local y position, a_y if the distance between the agent and the terrain is less than a given threshold, t.

We can define this behaviour as follows:

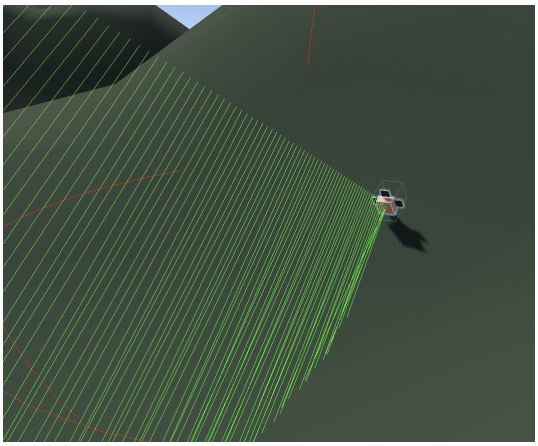
$$\vec{E} = \begin{bmatrix} 0 \\ U \\ 0 \end{bmatrix}, U = \begin{cases} \vec{0} & \text{if } a_y > t \\ u & \text{if } a_y \leq t \end{cases} \tag{23}$$

We will initially set $t = 5, u = 10$.

We can see this behaviour in operation in Fig. 29a, where an agent approaches a steep hill. We see that before point 1, the agent does not increase its altitude. When a ray is cast at point 1, the agent detects that the distance between itself and the terrain is less than the threshold, and increases its altitude. Point 2 is the crest of the hill. At this point, the agent detects that there is still not enough distance between itself and the terrain, so the altitude continues to increase. In Fig. 29b, we see that the agent has detected the terrain, but appears to carry too much forward momentum and is locked in a tilt, so cannot avoid the hill.



(a) Agent approaching a steep hill



(b) Agent failing to avoid terrain

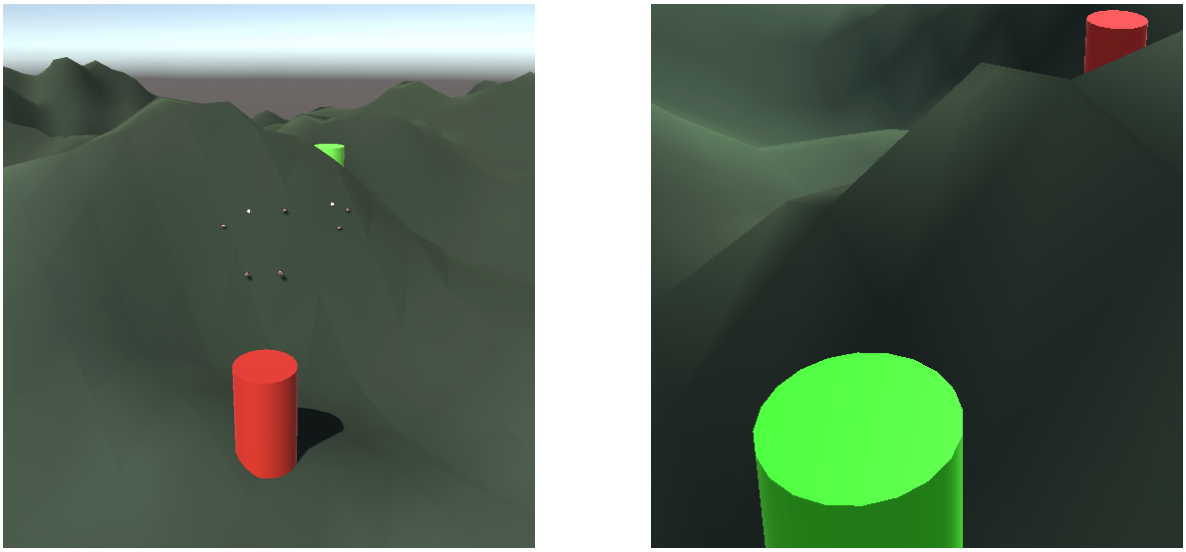
Figure 29: Naive Terrain Avoidance

These observations indicate that the current implementation of terrain avoidance is not sufficient. We will modify our ranging sensor by tilting it slightly in the direction

of the agent's velocity. This will allow the agent to detect the terrain in front of it, as well as below it, increasing the agent's ability to avoid terrain.

Unfortunately, the scale of the terrain proved a large challenge in optimising the avoidance behaviour. A limitation of the agents in the current implementation is that steep and large hills were hard to counter - mainly due to the behaviour only being able to avoid hills by increasing altitude. This is not ideal, as the agents should be able to avoid terrain by moving around it, rather than over it. We will discuss alternative methods in Section 6.3.

In line with the terrain changes, we will make some changes to how the simulation begins and ends. We define two platforms - a (red) start platform and a (green) end platform. The start platform is a flat surface, with a radius of 4, centred at the origin. The end platform is a flat surface, with a radius of 4, centred at a random point within a radius of 100m of the origin. The agents will begin on the start platform, and will be instructed to seek the end platform. These platforms are seen below in Fig. 30.



(a) Start Platform (b) Goal Platform

Figure 30: Start and Goal Platforms in a single simulation

At this stage we have a set of behaviours that allow the agents to navigate a complex terrain mesh. We can now consider the optimisation of these behaviours to allow the agents to operate in a more autonomous manner.

4 Methodology

In Section 1.1, we discussed the desire to tailor the agent's behaviour to operate in particular scenarios, based on some expected applications of UAV swarms.

As such, we define three scenarios that we will aim to optimise the agent's behaviour for. Alongisde these scenarios, we will define some metrics that we will use to evaluate the agent's performance, which will be used to inform the cost function we aim to optimise.

4.1 Scenario 1: Shortest First Arrival Time

Firstly, we will consider the scenario where reaching the goal in the shortest time is the most important factor. This is a present requirement for time-sensitive situations, where the redundancy of multiple agents is used to ensure that the goal is reached as quickly as possible. In essence, it means agents can take more risks in their behaviour in order to achieve the objective.

Some example situations where this scenario would be of use include military operations in hostile environments, gaining situational awareness in disaster zones, and surveying hazardrous environments[26].

For the simulations of this scenario, we will consider the following metrics:

First Arrival Time We must consider the time taken for the first agent to reach the goal; it is the primary objective of the agents in this scenario. Specifically, this is the time taken for the first agent to enter the goal radius as defined in Section 3.3.1. We define a simulation run time, which will be the length a simulation is allowed to run for.

For the purposes of normalisation, we will use the time taken for the first agent to reach the goal, divided by the simulation run time. This will give us a value between 0 and 1, where a lower number indicates a better performance.

This is the primary objective of the agents in this scenario, so we will give it a larger weighting in our cost function.

Proportion of Agents Arriving A simulation run where more agents arrive at the goal is more successful than one where fewer agents arrive. Furthermore, a simulation run where no agents arrive would be considered a failure.

We will then, also consider the proportion of agents that do not arrive at the goal by the simulation run time. This will give us a value between 0 and 1, where a lower number indicates a better performance.

This is a secondary objective of the agents in this scenario.

4.2 Scenario 2: Reduced Spread (Cohesivity)

This scenario aims to reduce the spread of the agents in the flock. This is important in situations where the agents must maintain a close formation, such as in search and rescue operations, or in the delivery of goods. A simulation run's success in this scenario will be dependent on two metrics:

Standard Deviation of Arrival Times We will consider the times at which the agents arrive at the goal, for all agents that arrive before the simulation run time. We will then calculate the standard deviation of these times. A lower standard deviation indicates that the agents are arriving at the goal in a more uniform manner. We will give this metric a larger weighting in our cost function.

A key issue with this metric is that it does not consider the method in which the agents arrive at the goal. For example, if all agents arrive at the goal at the same time, the standard deviation will be 0. This does not necessarily imply that the agents are in a close formation.

First Arrival Time As with the first scenario, we still consider the time taken for the first agent to reach the goal. A run where all agents reach the goal with a low standard deviation between them, but taking a long time to do so is suboptimal. We will give this metric a smaller weighting in our cost function.

Proportion of Agents Arriving As with the first scenario, we will consider the proportion of agents that do not arrive at the goal by the simulation run time. We do this to ensure that the cost function accounts for the number of agents within an arriving group.

4.3 Scenario 3: Reducing Collision Numbers

This scenario aims to reduce the number of collisions between agents and between agents and other objects in the environment. This is important in situations where the

agents must operate in close proximity to each other, such as in the delivery of goods, or in the area monitoring. We will consider both the arrival time of the first agent and the proportion of agents that arrive at the goal, as in the previous scenarios, to a small extent, for similar reasons as justified previously.

We will also consider the following metrics:

Number of Collisions We will consider the number of collisions that occur between agents and between agents and objects in the environment. A simulation run where fewer collisions occur is more successful than one where more collisions occur. We will give this metric a larger weighting in our cost function.

5 Optimisation

Earlier, we utilised an empirical method to optimise our initial three behaviours. However, we note that for each extra parameter we introduce, the complexity of the parameter space increases exponentially. Consider an example in which each parameter is bounded by integers within the inclusive range $(0, 10)$. With 3 parameters, this is a three dimensional parameter space with $11^3 = 1331$ possible combinations. With 5 parameters, this is a five dimensional parameter space with $11^5 = 161051$ possible combinations. This is a combinatorial explosion, and it is not feasible to optimise the parameters in this way.

Looking at other probabilistic methods for optimisation, two methods stand out. Firstly, genetic algorithms, which are inspired by the process of natural selection. This method is well researched within the field of behavioural optimisation in swarm robotics. One notable example is Alaliyat et al. [17], who use a genetic algorithm for optimising the parameters of a Boids model. This is a computationally expensive method.

Alternatively, we may utilise a hill climbing[4] algorithm. This is a simple optimisation algorithm that iteratively improves a solution by incrementally changing a single parameter. This method is less computationally expensive than a genetic algorithm, but it is also less likely to find the global minimum of a function, due to the likely presence of local minima, which the hill climbing algorithm cannot overcome.

Hence, we will look at an alternative method to find optimal values in the parameter space.

[4]In this specific case, where we aim to minimise the cost function, this is more specifically a gradient descent algorithm.

5.1 Simulated Annealing

The method we will use to optimise the parameters is simulated annealing. This is a probabilistic method that is used to find the global minimum of a function within a given search space. It is based on the physical process of annealing, where a material is heated and then cooled slowly to remove defects and reduce the material's energy. It is, in principle, a more advanced version of the hill climbing algorithm which can overcome local minima.

The algorithm works by starting at a random point in the search space. It then selects a random neighbour of the current point and evaluates the cost function at this point, t. If the cost function at the new point is lower than the cost function at the current point, the algorithm moves to the new point. If the cost function at the new point is higher than the cost function at the current point, the algorithm moves to the new point with a probability that is dependent on the difference between the cost function at the new point and the cost function at the current point,$|c(t)-c(t+1)|$, and a temperature parameter, T. The algorithm then reduces the temperature parameter and repeats the process until the temperature parameter reaches a minimum value.

The method of which the temperature is reduced is crucial to the success of the algorithm. If the temperature is reduced too quickly, the algorithm may become stuck in a local minimum. If the temperature is reduced too slowly, the algorithm may take too long to converge to a solution. We will use a linear cooling schedule. That is, T decreases linearly with each iteration of the algorithm.

We will define our acceptance function, $P \in (0, 1)$ as follows:

$$P = \begin{cases} 1 & \text{if } c(t+1) < c(t) \\ e^{-\frac{c(t+1)-c(t)}{T}} & \text{if } c(t+1) \geq c(t) \end{cases} \tag{24}$$

We note the exponential term in the acceptance function. This is based on the *Boltzmann distribution*, which is used in statistical mechanics to describe the probability of a system being in a certain state. The Boltzmann distribution is characterised by the equation:

$$P(i) = e^{-\frac{E_i}{kT}} \tag{25}$$

Given the Boltzmann constant, k, and the temperature, T, the probability of a system being in a state with energy E_i is given by the above equation.

We can then define our parameters and our cost function. As noted previously in Section 3.2.4, the parameter space can be defined as a linear combination of the behaviours and their weights. For clarity, we now redifine this in terms of our extended parameter space, which includes the goal seeking and terrain avoidance behaviours.

$$\vec{V} = \alpha \begin{bmatrix} x_{avoid} \\ y_{avoid} \\ z_{avoid} \end{bmatrix} + \beta \begin{bmatrix} x_{align} \\ y_{align} \\ z_{align} \end{bmatrix} + \gamma \begin{bmatrix} x_{cohere} \\ y_{cohere} \\ z_{cohere} \end{bmatrix} + \delta \begin{bmatrix} x_{seek} \\ y_{seek} \\ z_{seek} \end{bmatrix} + \epsilon \begin{bmatrix} x_{terrain} \\ y_{terrain} \\ z_{terrain} \end{bmatrix} \tag{26}$$

We can then define our cost function, which is a linear combination of the metrics defined in Section 4. We will define a singular, modular cost function to allow for the easy addition of new metrics and the ability to tweak the weighting of the existing metrics.

The cost function will normalise the given weights, and the given metrics will also be normalised to ensure the output of the cost function lies in the range $(0, 1)$.

Specifically, we define the normalisation of the four weights as:

$$\forall w_i \in W, w_i = \frac{w_i}{\sum_{i=0}^{|W|} w_i} \tag{27}$$

We also define the normalised metrics as follows:

First Arrival Time We can normalise this value by dividing it by the simulation time (the length the simulation is allowed to run for).

Hence,

$$m_1 = \frac{\text{firstArrivalTime}}{\text{simulationRuntime}} \tag{28}$$

Proportion Reached In a similar nature to the normalisation method for the *first arrival time* metric, we can divide the number of agents that have not reached the goal by the simulation time by the total number of agents in the simulation, namely:

5.2 Simulations

5.3 Results

6 Improving Autonomy

6.1 Boids (1999): Introducing Obstacle Avoidance

6.2 A Simplified Model

We now aim to reduce the complexity of the parameter space in the hopes of strengthening the link between the parameters and the emergent behaviour.

6.3 Local Potential Field Emergence

6.4 Final Results

7 Future Work

7.1 LPFE Implementation

7.2 Online Optimisation

7.3 Hostility

7.3.1 Hostile Environments

7.3.2 Hostile Agents

8 Evaluation

8.1 Project Management

8.2 Comparing Models

8.3 Speed-Success-Flock Size Tradeoffs

8.4 Use Cases

8.5 Author's Assessment of the Project

References

[1] D. Mandloi, R. Arya, and A. K. Verma, "Unmanned aerial vehicle path planning based on a* algorithm and its variants in 3d environment," *International Journal of Systems Assurance Engineering and Management*, vol. 12, no. 5, pp. 990–1000, Jul 2021. [Online]. Available: https://link.springer.com/article/10.1007/s13198-021-01186-9

[2] T. Chen, G. Zhang, X. Hu, and J. Xiao, "Unmanned aerial vehicle route planning method based on a star algorithm," in *2018 13th IEEE Conference on Industrial Electronics and Applications (ICIEA)*, 2018, pp. 1510–1514.

[3] K. Burwell, "Multi-agent pathfinding for unmanned aerial vehicles," *Upc.edu*, Oct 2019. [Online]. Available: https://upcommons.upc.edu/handle/2117/176279

[4] F. Mondada, G. Pettinaro, A. Guignard, I. Kwee, D. Floreano, J.-L. Deneubourg, S. Nolfi, L. M. Gambardella, and M. Dorigo, "Swarm-bot: A new distributed robotic concept: Swarm robotics (guest editors: Marco dorigo and erol Şahin)," *Autonomous Robots*, vol. 17, 01 2004.

[5] Y. Zhou, B. Rao, and W. Wang, "Uav swarm intelligence: Recent advances and future trends," *IEEE Access*, vol. 8, pp. 183 856–183 878, 2020.

[6] M. Verdoucq, G. Theraulaz, R. Escobedo, C. Sire, and G. Hattenberger, "Bio-inspired control for collective motion in swarms of drones," in *2022 International Conference on Unmanned Aircraft Systems (ICUAS)*, 2022, pp. 1626–1631.

[7] C. W. Reynolds, *Flocks, Herds, and Schools: A Distributed Behavioral Model*. New York, NY, USA: Association for Computing Machinery, 1998, pp. 273–282. [Online]. Available: https://doi.org/10.1145/280811.281008

[8] S. Hayat, E. Yanmaz, and R. Muzaffar, "Survey on unmanned aerial vehicle networks for civil applications: A communications viewpoint," *IEEE Communications Surveys & Tutorials*, vol. 18, pp. 2624–2661, 2016. [Online]. Available: https://api.semanticscholar.org/CorpusID:424353

[9] I. Perez, A. Goodloe, and W. Edmonson, "Fault-tolerant swarms," in *2019 IEEE International Conference on Space Mission Challenges for Information Technology (SMC-IT)*, 2019, pp. 47–54.

[10] J. Hu, H. Niu, J. Carrasco, B. Lennox, and F. Arvin, "Fault-tolerant cooperative navigation of networked uav swarms for forest fire monitoring," *Aerospace Science and Technology*, vol. 123, pp. 107 494–107 494, Apr 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1270963822001687

[11] L. M. Pyke and C. R. Stark, "Dynamic pathfinding for a swarm intelligence based uav control model using particle swarm optimisation," *Frontiers in Applied Mathematics and Statistics*, vol. 7, Nov 2021. [Online]. Available: https://www.frontiersin.org/articles/10.3389/fams.2021.744955/full#B18

[12] P. Basu, J. Redi, and V. Shurbanov, "Coordinated flocking of uavs for improved connectivity of mobile ground nodes," in *IEEE MILCOM 2004. Military Communications Conference, 2004.*, vol. 3, 2004, pp. 1628–1634 Vol. 3.

[13] E. Falomir, S. Chaumette, and G. Guerrini, "A mobility model based on improved artificial potential fields for swarms of uavs," in *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2018, pp. 8499–8504.

[14] A. Juliani, V.-P. Berges, E. Teng, A. Cohen, J. Harper, C. Elion, C. Goy, Y. Gao, H. Henry, M. Mattar, and D. Lange, "Unity: A general platform for intelligent agents," 2020.

[15] A. G. Madey and G. R. Madey, "Design and evaluation of uav swarm command and control strategies," in *Proceedings of the Agent-Directed Simulation Symposium*, 2013, pp. 1–8.

[16] N. Watson, N. John, and W. Crowther, "Simulation of unmanned air vehicle flocking," in *Proceedings of Theory and Practice of Computer Graphics, 2003.*, 2003, pp. 130–137.

[17] S. A.-A. Alaliyat, H. Yndestad, and F. Sanfilippo, "Optimisation of boids swarm model based on genetic algorithm and particle swarm optimisation algorithm (comparative study)," in *European Conference on Modelling and Simulation*, 2014. [Online]. Available: https://api.semanticscholar.org/CorpusID:16528976

[18] G. Hattenberger, M. Bronz, and J.-P. Condomines, "Evaluation of drag coefficient for a quadrotor model," *International Journal of Micro Air Vehicles*, vol. 15, p. 4, 2023. [Online]. Available: https://doi.org/10.1177/17568293221148378

[19] M. Figliozzi, "Multicopter drone mass distribution impacts on viability, performance, and sustainability," *Transportation Research Part D: Transport and Environment*, vol. 121, p. 3, 2023.

[20] F. P. Thamm, N. Brieger, K. P. Neitzke, M. Meyer, R. Jansen, and M. Mönninghof, "Songbird - AN Innovative Uas Combining the Advantages of Fixed Wing and Multi Rotor Uas," *ISPRS - International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, vol. XL1, p. 345, Aug. 2015.

[21] S. Skogestad, "Probably the best simple pid tuning rules in the world," in *AIChE Annual Meeting, Reno, Nevada*, vol. 77. Citeseer, 2001, p. 276h.

[22] A. McCormack and K. Godfrey, "Rule-based autotuning based on frequency domain identification," *IEEE Transactions on Control Systems Technology*, vol. 6, no. 1, pp. 43–61, 1998.

[23] U. Technologies, "Unity - scripting api: Vector3.sqrmagnitude," 2022. [Online]. Available: https://docs.unity3d.com/ScriptReference/Vector3-sqrMagnitude.html

[24] K. Perlin, "An image synthesizer," in *Proceedings of the 12th Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '85. New York, NY, USA: Association for Computing Machinery, 1985, pp. 287–296. [Online]. Available: https://doi.org/10.1145/325334.325247

[25] T. R. Etherington, "Perlin noise as a hierarchical neutral landscape model," *Web Ecology*, vol. 22, no. 1, pp. 1–6, 2022. [Online]. Available: https://we.copernicus.org/articles/22/1/2022/

[26] T. Srinath, P. Can, and P. Mitch, "Minimum time trajectory generation for surveying using uavs," 2022.