1 Context

2 Quadcopter Model

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 [1].

The dimensions and chracteristics of the agent have been determined using both realistic averages [2] 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.

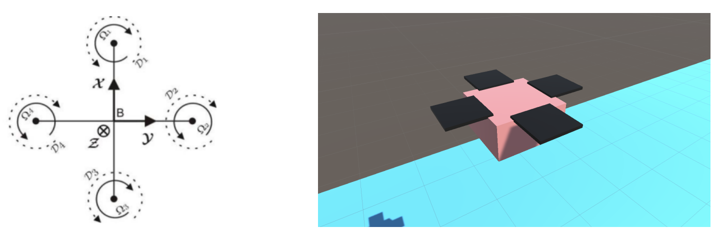


Figure 2: Propeller Diagram & Simulated Agent

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 [3].

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

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

Figure 3: Control Operations and Thrust Levels

It is noted that yaw is omitted from Fig. 3. 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 table below:

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_+

Figure 4: 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. 5.

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

Figure 5: Thrust Constants for a drone of mass m kg

3 Stabilisation: Proportional-Integral-Derivative Controller

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.

As such, the need for a control system becomes apparent. A PID (Proportional-Integral-Derivative) is a feedback control system, which uses the error between the current state and the desired state to calculate the control parameters.

Fig. 6 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.

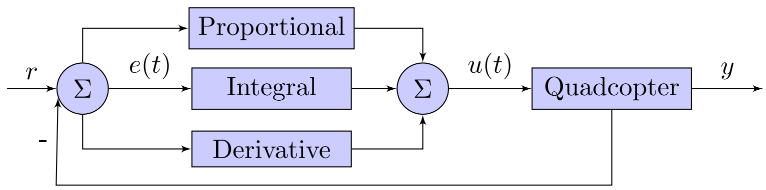


Figure 6: PID Controller

The PID controller is defined by three constant parameters, 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:

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

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

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

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. 7.

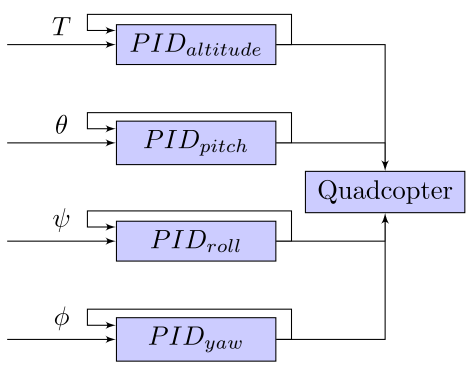


Figure 7: System Model

The constants are then tuned to generate expected stabilisation behaviour. As mentioned previously, each propeller has a base thrust level equal to the force required to hover, $T_b = \frac{mg}{4}$. The PID controller is then used to calculate the thrust level to be added to each propeller, T_{add}, in every frame of the simulation, to perform the desired operation.

Using manual tuning, the values were optimised as shown in Fig. 8

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

Figure 8: Tuning Constants

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

3.1 Faults - Oscillation

4 Methodology - what are the metrics, how do we define success?

Qual + Quant analysis!

5 Introducing Autonomy

The initial aim is to achieve position control for an agent, such that when given a current position $[x_0, y_0, z_0]$ and a goal position $[x_1, y_1, z_1]$, the agent will move to the goal 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.

This can be split up into two main goals:

1. Apply a force in the opposite direction when avoiding other agents.
2. Use trigonometry to apply relative ϕ and θ forces, reducing these forces when closer to the goal using a separate PID position controller.

5.1 Boids: Flocking Behaviour

Initially, multiple agents are now introduced into the environment and are programmed to hover. We will define two constants to assist in the autonomy of the agents. First, an avoidance radius; namely, the *square avoidance radius*. For the purposes of algorithmic efficiency, we compare the magnitude of a vector, squared, between agents to the defined square avoidance radius, as per *Unity*'s recommendation [4]. We also specify the number of agents to generate and the spawn radius in which to generate them. Finally, we define the *neighbour radius*, defined as the radius in which an agent will consider other agents as neighbours. This is used to reduce the number of calculations required for each agent, as it will only consider nearby agents.

Notably, we must also define the implementation of 'applying a force F to an agent' with respect to the agents' control parameters. Initially, we consider the difference between the position vector in the agent's local frame and the environment's global frame, namely, the rotation matrix that characterises the agent's orientation. The mathematics of such an operation is outside the scope of this report. In short, we can either calculate the rotation matrix, R, representing this change of frame and utilise a transformation matrix, such that $F_{global} = R \cdot F_{local}$, or use quarternions to represent the change of frame. *Unity* has a built in method to handle this, but the source code is obfuscated. We will utilise this method, `transform.InverseTransformVector()`, to calculate the force vector in the agent's local frame. This can then be split up into the constituent pitch and roll magnitudes by taking the x and z components respectively.

We then introduce Reynolds' principles [5] to simulate some flock behaviour. The first principle, *separation*[1]. In practice, this is done as follows:

1. For each agent, calculate the distance to each other agent within the neighbour radius.
2. For each distance, if the magnitude of the vector is less than the square avoidance radius, add the vector to the total force vector.
3. Divide the total force vector by the number of agents within the neighbour radius.
4. Apply the force vector to the agent.

To implement the second principle, *alignment*, we take the heading of each agent within the neighbour radius and average them. We then apply a force in the direction of the average heading to the agent. As with the separation principle, this force is separated into pitch and roll components.

Finally, we implement the third principle, *cohesion*. This is in principle, the inverse to the separation principle. Hence, this is implemented as follows:

1. For each agent, find the global position of each other agent within the neighbour radius.
2. Sum all these positions and divide them by the number of neighbours to find the average position between all neighbours.
3. Subtract the agent's current position from the average position to find the force vector between the two.
4. Apply this force vector to the agent, after transforming it into the agent's local frame.

Finding the balance between cohesion and separation is important. If the cohesion force is too strong, the agents will collide. 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 explore this in more detail in the next section.

In general, tuning the influence of all three components is challenging. Even considering the constants in the simulation, Reynolds 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. The method for this is empirical; we can observe the behaviour of the agents and adjust the constants accordingly. We introduce the method for this below.

Initially, we will set the constants as shown in Table 1. These may be adjusted later. Additionally, we will set the weights for each component as shown in Table 2.

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

Table 1: Spawn Constants

Behaviour	Weight
Separation	1
Alignment	1
Cohesion	1

Table 2: 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

[1] In Reynolds' original article, this is called *collision avoidance*

black circle. This is shown in Fig. 9. 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. 9 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. 10. 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. 11.

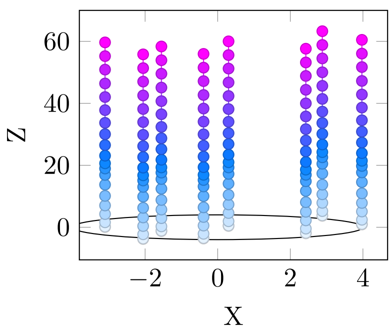


Figure 9: Initial

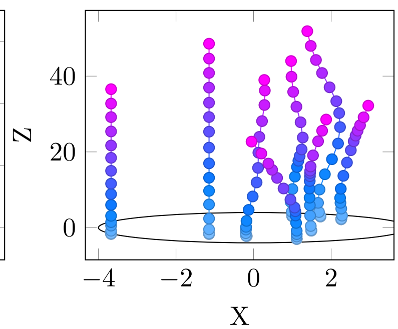


Figure 10: Avoidance Tuning

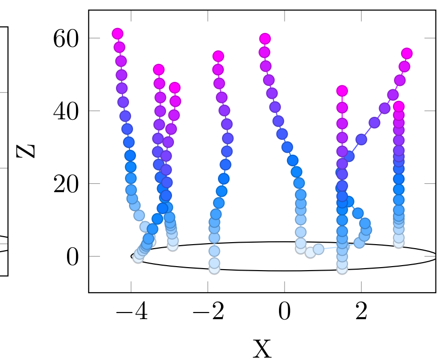
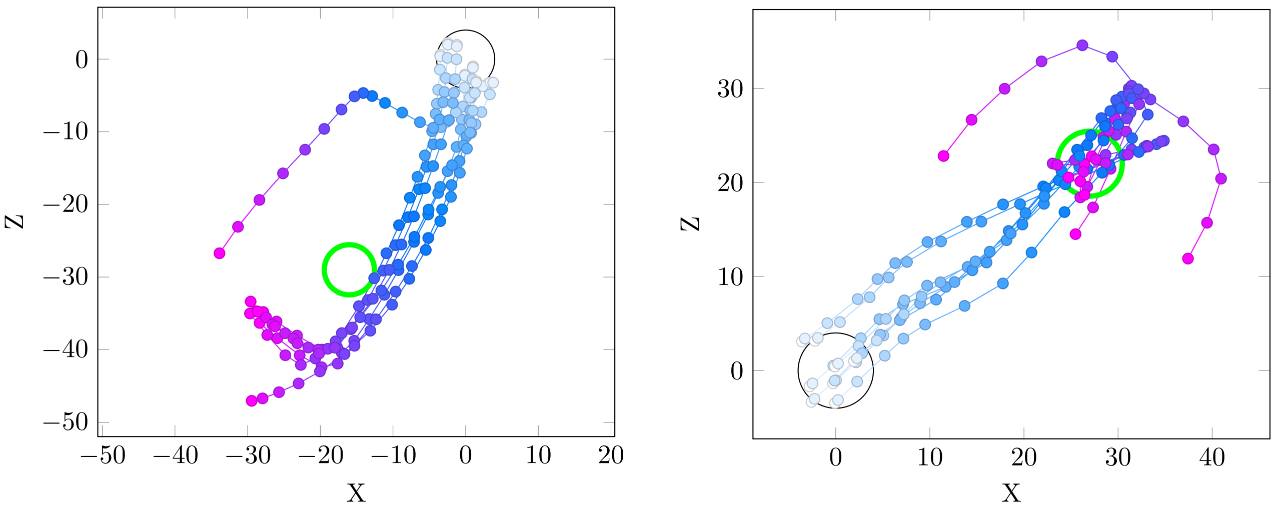


Figure 11: Final Tuning

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. However, this may change when we look at implementing our fourth behaviour, *goal seeking*.

5.2 Boids: 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. 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. 12a. 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. 12b.



(a) Normalised Force Vector

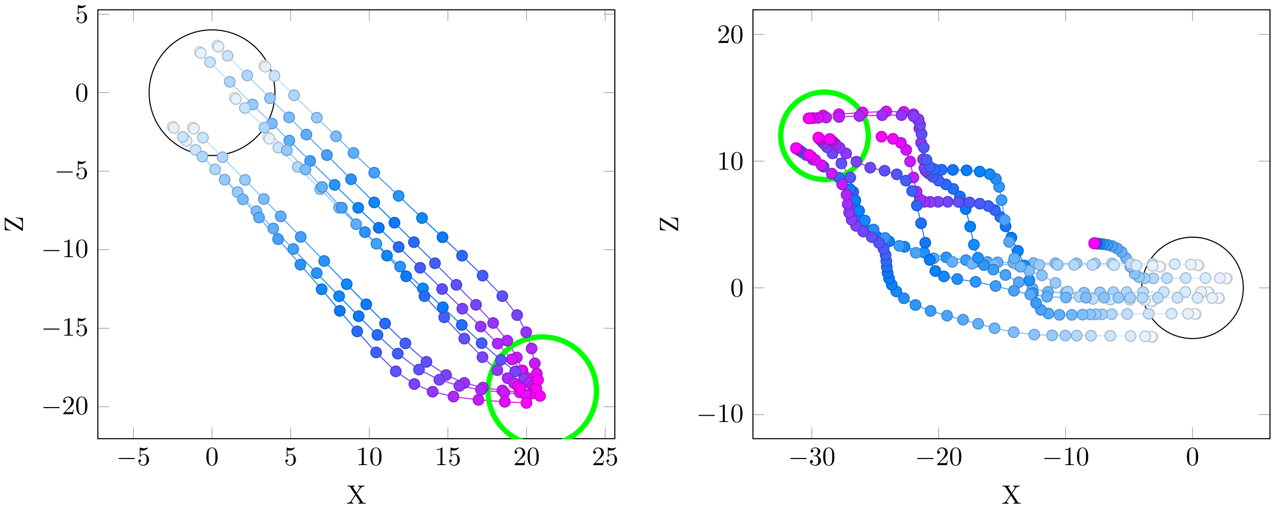
(b) Clamped Force Vector

Figure 12: Seeking Behaviour

The behaviour now seen in Fig. 12b 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 3. 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. 13a. 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 7.



(a) Final Seeking Behaviour

(b) Three Initial Behaviours + Seeking

Figure 13: Finalising Seeking Behaviour

6 Obstacles

6.1 Obstacle Avoidance

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.

6.2 Limitations of Boids

7 Complex Pathfinding for Hostile Environments

7.1 A* - The Hacks & Bodges?

7.2 Pair Potential Model

7.2.1 Obstacle Avoidance

8 Real World Limitations

8.1 Real World Represenation

8.2 Sensing & Vision

9 Extensions

9.1 Obstacle Detection

9.2 Semi-Autonomous Control

10 Evaluation

10.1 Speed-Success-Flock Size Tradeoffs

10.2 Use Cases

References

[1] 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

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

[3] 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.

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

[5] 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