Enabling Multirotors to Perform Construction Tasks Using Swarm Algorithms

Jens-Christian Finnerup

Kongens Lyngby 2016

Summary (English)

The goal of the thesis is to ...

Summary (Danish)

Målet for denne afhandling er at ...

Preface

This thesis was prepared at DTU Compute in fulfilment of the requirements for acquiring an M.Sc. in Engineering.

The thesis deals with ...

The thesis consists of ...

Lyngby, 01-August-2016

Not Real

Jens-Christian Finnerup

Acknowledgements

I would like to thank my....

Contents

Chapter 1

Introduction

In today's world, we increasingly rely on robotics to perform a wide range of tasks, with use cases ranging from complimenting, or even substituting, human labour in factories to more complex tasks such as automated aerial photography. Though technology continues to improve, the use of robotics is often confined to within predictable environments, where machines are told what to do and when to do it. More recently though, as control algorithms improve, autonomous robots are starting to become a practical reality, with increasing ability self-plan and work unsupervised. As such we experience a paradigm shift, where robots are no longer limited to predictable and confined environments, but can act freely and adapt to changing circumstances.

As a consequence of this, new use cases and areas within robotics are starting to gain interest. Specifically the area of autonomous flying vehicles is gaining interest with companies such as amazon, preparing to use flying drones to deliver packages. This requires the drone to be able to plan according to changes in the environment as well as being able to coordinate its movements in relation to other drones around it. At a currently more theoretical scale, using flying drones to perform construction tasks has been subject for testing by universities such as *ETH Zurich*, where micro drones were used to construct a 6 meter tall tower [ALH^{+}14].

The task of using flying vehicles to perform construction is especially interesting because it combines many of the challenges that face robotics today, as agents become more autonomous. A construction task work as a great test bed, for testing flying drones as it involves multiple agents working on the same problem as once and as such they must have the ability to plan according to changes in their individual environment.

One solution to the challenges of enabling drones to collaborate, is to create individual sectors of space in which only a single agent is allowed to move at a certain time. This is a simple way to prevent collisions between the drones, but it also poses some optimization challenges, as the path of the drones cannot intersect. This means that even if a path for a drone towards a goal is optimal (shortest), it cannot be used if it intersects with the path of another agent. Another drawback of this simple solution is that it limits the amount of agents able to act with in a certain space, to a small number, as reach has to have its own designated area. As such there is great motivation to design control algorithms, which can enable drones to collaborate on tasks, while allowing them to avoid collisions without limiting their flight space.

This thesis focuses on how create and use algorithms from the field of swarm robotics as control for flying drones. The different algorithms will be implemented on a simulation test bed, to compare their complexity and their ability to give optimal movements to the drones within the simulation. For the comparison, different metrics will be used measure effectiveness in allowing drones to solve a collaborative problem. As a basis for the comparison, this thesis will document the implementation of the simulation platform and the control algorithms.

1.1 Problem Description

The following will outline the detailed goal of this thesis, as well as the subgoals necessary to achieve it.

1.1.1 Main Problem

To create and use swarm algorithms to enable quadcopters to coordinate their movements in a multi-agent environment.

1.1.2 Sub Problems

- To design and implement a simulation platform, based on a physical model, within which the algorithms can be used to control simulated agents
- To design a problem for the agents to collaborate on, to use as a standardized test for performance of the algorithms
- To determine the correct metrics for optimizing and comparing the swarm algorithms
- To implement swarm algorithms based on existing concepts, and test them using the metrics and the simulation platform
- To optimize the control algorithms based on pre mentioned tests, and analyze potential areas of improvement

1.2 Project plan

We note that the contents of the project plan is also something we would like to see in the introductory chapter of your thesis. In fact, you can reuse your final project plan (possibly extended) as the introduction. If you prefer to write an introduction from scratch, it is, of course, important that it is consistent with the final project plan.

Chapter 2

Methodology

In order to evaluate the different control algorithms and their performance, the approach towards designing and testing the algorithms must be systematic and concise. The different subproblems specified in 1.1.2, can be said to either relate to either the simulation design or the algorithm design. As such, for the purpose of this thesis, a different approach will be used when dealing with constructing the simulation and when designing and testing the control algorithms. In this chapter, I will outline these approaches and the methodology used. The first three sections of this chapter will deal with the simulation methodology, whereas the latter three will deal with the implementation and testing of the control algorithms.

2.1 Control Problem

The main research problem os this thesis involves building and evaluating different control algorithms, and as such each algorithm must be subject to the same evaluation technique. This evaluation technique will throughout the thesis be referenced to as the *Control Problem*, and the purpose of the agents will be so solve this problem collectively. As such the problem chosen must be the same problem throughout every simulation and independent of the control algorithms themselves. The problem must also be able to be solved collectively, by multiple agents at the same time, while still posing enough challenges to properly test the control algorithms. Posing dangers to the agents, such as the risk of collision.

This control problem will exist within the simulation as a physical task that the agents have to carry out. Chapter 3.3.1 will outline the concrete control problem used, as well a discussion on the specific control problem's advantages and limitations.

2.2 Simulation Implementation

The control problem and the simulation are mutually dependent, as the problem has to able to exist and be solvable within the simulation. As such the simulation has to be designed with the control problem in mind, but the control problem also has to be made with the limitations of the simulation taken into account. The simulation aspect is crucial to the validity of the results expressed in this thesis. For the purpose of solving the problems outlined in this thesis, the simulation will be a virtual environment, in which drones can be deployed. The drones will have to be subject to the same physical laws, as they would in real life such that the findings and test results can by applied to issues and control techniques in the real world.

In order to reduce complexity of the simulation, some physical laws are not taken into account when designing the simulation. This is done based on assumptions on which laws influence the results. An outline of the simulation design, can be found in Chapter 3.

2.3 Validation

Validation is a method to ensure that the simulation is mimicking real life physics and not simply animating the desired behavior of the drones. The validation will be carried out iteratively along with the construction of the simulation, to monitor the progress of the simulation construction, and act as a sort of checklist to the accuracy and capabilities of the simulation. The validation phase consists of multiple validation tests, meant to test if the simulation actually simulates the physical mechanics required. These mechanics can range from gravitational pull to accurate impulse and mass on the bodies in the simulation. The validation tests will when range from checking the accurate acceleration of drones in free fall etc. These validation tests are also important because the simulator must have the predictable behavior across many scenarios, such that they can be compatible. The validation tests will be discussed and outlined in Section 3.5

2.4 Test Metrics

Up until now this chapter on methodology has been focused on systematically creating and validating the simulation platform. The following part will be covering my approach to creating and testing the control algorithms.

The comparison of the algorithms tested within this thesis, will be done based on their ability to solve the Control Problem. However the control problem must be designed in such a way, that it can be solved to various degrees. To measure how well the control problem is solved, some metrics must be chosen so that they represent the challenges of the control problem. I.e. if the control problem challenges the agents in collision avoidance, a test metric could be each agent's distance the all other agents over time. Time to solve the problem will also act as a simple test metric. The metrics chosen will be covered in Chapter 3 and will in part be based on the challenges of swarm robotics covered in `Multi-robot navigation in formation via sequential convex programming` [AMBR15].

2.5 Control Implementation

The algorithms tested within this thesis and presented as *swarm algorithms* will be covered in depth in Chapter 4. When implementing and testing the algorithms in this paper, the algorithms will be partly implemented through experimentation. This is due to the fact that swarm algorithms such as the *Artificial Bee Colony* (or ABC) [BRG$^+$11], cannot directly be mapped to the issue of controlling quadcopters to perform construction tasks. The ABC has a different number of agents and multiple categories. Because of this, mapping some of the algorithms will involve some experimentation and for the sake of methodology it must be stated that algorithms will undergo a level of personalization when implemented. This will be documented along with how much the algorithms ends up differing from its theoretical counterpart.

2.6 Optimization

In contrast to the Simulation Implementation, the implementation of the control algorithm will not be followed by a validation step in the same way. This is due to the fact the control algorithms cannot be validated with the same fixed criteria, as their desired behavior deviates for each algorithm. Instead, they will undergo optimization. This refers to the activity of not only mapping the control algorithms to control quadcopters, but also tuning them in an attempt to maximize their performance.This will be in benchmarked with the test metrics and the control problem within the simulation. The optimization will only be done on the control algorithms themselves, and not by tuning simulation variables, as all algorithms bust be tested on the same simulation. The simulation variables will be predetermined in the *Validation* phase (see Section 3.5) and hereafter remain static.

CHAPTER 3

Simulation

As mentioned previously in this paper, there are many advantages to performing simulation vs. carryings

The simulation environment, hereafter referred to as simple *the simulation*, will be the 3D platform on which the quadcopters and their control algorithm can be tested. The simulation is carried out in continues time, and in this chapter i will outline the design and implementation of the simulation platform. Furthermore also the validation step used to ensure that the simulation follows the required physical mechanics.

3.1 Simulation vs. Animation

3.2 Software

The software used in this paper to create the simulation platform is Matlab Version 8.7 (R2016a) with Simulink [?]. Simulink was chosen based on its ease of use and ability to perform simulations continuous time. Simulink also has a 3D graphics engine called `VR3D`[?], which can be used to model 3D animations from any signal output from the simulation model. I.e. if the simulation models a flying quadcopter, the position values of the quadcopter can be sent as signals to the 3D model. The ability to 3D animate the movements of the quadcopters is important, as *seeing* their actual flight paths can be a simple way to discover errors in either the simulation model or the control algorithm.

3.3 Design

This section will outline the specifications of the simulation platform, after which it is designed. Note that some specifications were not intended from the beginning, such as a limitation on global velocity, but are the result of an iterative design process and tuning the simulation.

3.3.1 Control Problem

The goal of the control problem is to challenge the control algorithms such that they can be compared. The control problem chosen for the purpose of this thesis, will be for the agents to collaborate on assembling a structure. The structure will be made of boxes or bricks (hereafter referred to only as boxes), which the agents are able to lift and relocate. The starting position of the boxes will be in one end of the rectangular environment, where the assembled structure will be in the other end. This type of setup requires movement by the agents in close proximity to each other, which introduces the risk of collision. This type of problem must also be solved in a specific order, such that the bottom of the structure is created first etc. The control algorithm will be given the position of boxes as well as the position for the proposed bricks in the completed structure. The measurements used to test the performance of the control algorithm on this problem are outlined in 3.6. Figure 3.1 shows the design of the simulation environment, with the control problem and the agents viewed from the top.

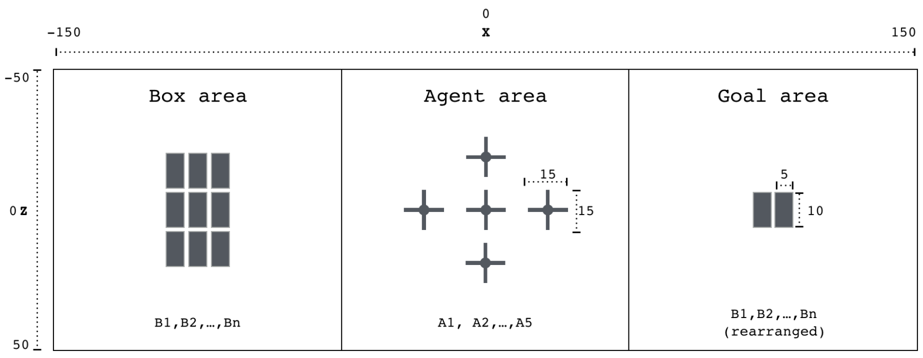

Figure 3.1: Design of simulation environment, as viewed from above

3.3.2 Assumptions on physics

The goal of the simulation is to simulate multiple flying quadcopters, and as such physical mechanics play an important role of the simulation model. This section will list the specifications of the simulation environment as well as some physical parameters. Due to the desire to limit complexity some physical parameters, such as wind, will be assumed to be non-existent in the simulation model. Others such as gravity and impulse must be included in the model, as they play a more vital role in producing accurate test results. The specifications of the physical parameters can be found in Table 3.3, followed by a brief reasoning behind the parameter.

Parameter	**Role in simulation**
Gravity	Simulated
Mass/Impulse	Simulated
Collisions	Detected, but not simulated
Friction	Not simulated
Wind resistance	Not simulated

Table 3.1: Physical parameters

- **Gravity** is an essential part of the simulation, as the control algorithm has to be tested in an environment where objects behave with close real-life resemblance. Please see 3.4.3 for the implementation of this mechanic
- **Mass** as well as the impulse that an object has when in movement, is an important aspect since it relates to how much force is required to affect

the movement of an object

- **Collisions** are important as well as they are used as a test metric for how well the control algorithms are performing. They are however not important to simulate or animate in 3D, as long as they can be detected. As such they are simply detected, to avoid adding complexity
- **Friction** to objects such as the friction between boxes, is assumed to be non-existent in the simulation. This also applies for the quadcopter's ability to carry the boxes, as it is assumed to always to functioning

3.3.3 Agents

The *Agents* within the simulation refer to the drones. The word agent is used as the problem of this thesis, can be mapped as a multi-agent problem with each drone acting as an individual agent. Table 3.2 shows the specifications of the agents in the simulation and is followed by description of, and a reasoning behind each specification.

Parameter	**Description**	**Value**
Agent-type	What type of agent to be simulated	*Quadcopter*
Size	The space occupied by an agent	$15x15x2cm$
Local Control	The low-level control system which balances and controls the agent position	*SimulinkPD*
Mass of agent	The number of quadcopters acting to solve the control problem	$150g$

Table 3.2: Agent Specifications

- The **agent-type** is important to not be changed throughout the simulation, as this determines which control surfaces and abilities an agent has. The quadcopter type is chosen because it is most common test platform for micro aerial vehicle (or MAV) control algorithms, and has for many become synonymous with the word *drone* [ALH⁺14]
- **Size** refers to the volume of space taken up by an agent. To reduce complexity an agent is represented by four arms and a center sphere in the simulation. A collision is defined by two or more agent's overlapping each others current space
- **Local Control** does not refer to the control algorithm that the quadcopters use to navigate, but rather the algorithm which stabilizes each

machine individually. This is a fairly standard and well tested part of modern quadcopters, and as such a standard library from Simulink is used for the desired effect. Please see 3.4.2 for the implementation of this

- **Mass of the agent** is implemented through a combination of both the gravitational mechanic and the PD controller library provided by Simulink. 150g is chosen as this is a common weight for MAVs

3.3.4 Simulation Environment

This subsection outlines the specifications for the simulation environment. This refers to the specific values of the physical mechanics used to simulate a real world environment, as well as the number of objects in, and the size of the simulated environment. These can be found in Table3.3

Parameter	**Description**	**Value**
Gravity	Mechanic to simulate Gravity	9.8m/s^2
Dimensionality	The number of space dimensions	3D
Floor size	The floor size relating to the area of the simulation	300x100cm
Number of agents	The number of quadcopters acting to solve the control problem	5
Number of boxes	The number of boxes to be moved to their respective goal position	9
Mass of box	The assumed mass of one box throughout the simulation	*massless*
Max. velocity	The limit of the velocity of any object on every axis	15cm/s
Max. acceleration	The limit of the acceleration of any object on every axis	9.8m/s^2
Attach threshold	The maximum distance from an agent to a box, before the box can be attached	2cm

Table 3.3: Environmental Specifications

- **Gravity** as mentioned in Table 3.1 the and is simulated as a force pulling on the Y-axis of the simulation with the normal gravitational acceleration of 9.8m/s^2
- **Dimensionality** is set to 3 dimensions. Having a 2-dimensional simulation could have worked for some applications, however one of the challenges

of mapping swarm algorithms to controlling drones, is the 3rd dimension. As such, having the simulation be in 3 dimensions is necessary and cannot be scaled down to reduce complexity without affecting the validity of the test results

- The **floor size** of the simulation platform is chosen to be $300x100cm$, where the height is infinite. This is relatively small, but sufficient as the size of the drones are $15x15x2cm$, giving them plenty of space to move in

- **Number of agents** and are chosen to be 5. Initially the reasoning behind this was for testing purposes, but when experimentation with the swarm algorithms started, the 5 agents turned out to be posing the same challenges as a higher number due to the size of simulation environment. Therefore a higher number isn't needed

- **Number of boxes** ALRIGHT THIS NEEDS TO BE ADJUSTED

- The **mass of a box** is central to the performance of the control algorithms as it changes the total mass of the drone when carried. To reduce the complexity of the simulation however, these have been assumed to be massless and will as such not affect the drones movement

- **Max. velocity** is a parameter that was decided upon after some trial and error with getting the physical objects in the simulation environment, to behave in a real-life manner

- **Max. acceleration** was as with the limited velocity, inserted to force objects into behaving in a real life manner. This was needed as the Simulink PD component (see Section 3.4.2) had a tendency to overcompensate when accelerating drones to a desired velocity

- **Attach Threshold** is an interesting and quite important parameter. It builds on the assumption that an agent at any given velocity, can grab and carry a box as long as the grabbing action is performed when the distance between the agent and the box is smaller than or equal to this value

3.4 Implementation

After assessing the needs of the simulation model, the actual construction in simulink is done by connecting different modules in a circuit like manner. The model consist of a mixture of predefined Simulink blocks and custom Matlab script blocks, for more advanced functionality, such as the control algorithms.

The finished model, which can be seen in Figure 3.2. The modules and submodules of the simulation model have been labeled with a title, describing their functionality.

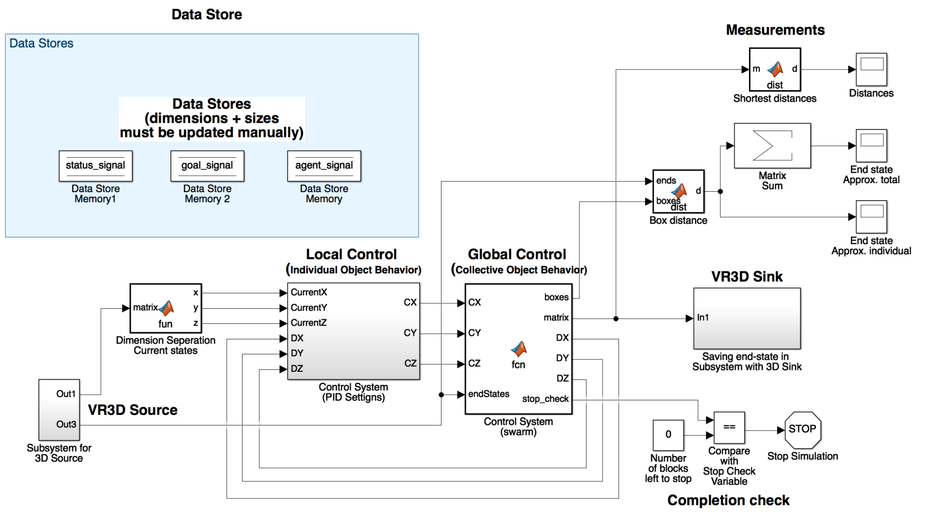
Data Store
Data Stores
Data Stores (dimensions + sizes must be updated manually)
status_signal
Data Store Memory1
goal_signal
Data Store Memory 2
agent_signal
Data Store Memory
Measurements
m dist d
Shortest distances
Distances
Matrix Sum
End state Approx. total
ends boxes dist d
Box distance
End state Approx. individual
Local Control (Individual Object Behavior)
Global Control (Collective Object Behavior)
matrix fun x y z
Dimension Seperation Current states
CurrentX CurrentY CurrentZ DX DY DZ CX CY CZ
Control System (PID Settigns)
CX CY CZ endStates fcn boxes matrix DX DY DZ stop_check
Control System (swarm)
VR3D Sink
In1
Saving end-state in Subsystem with 3D Sink
Out1 Out3
VR3D Source
Subsystem for 3D Source
0
Number of blocks left to stop
==
Compare with Stop Check Variable
STOP
Stop Simulation
Completion check

Figure 3.2: Overview of finished model in Simulink

The following subsections will outline how the different model requirements from Section 3.3, were implemented in a step wise manner.

3.4.1 VR3D, Sources and Sinks

As shown in Figure 3.1, the simulation environment has certain starting points for the boxes and agents, and a desired endpoint for the boxes. The 3D animation is created with the Simulink plugin VR3D [INSERT REFERENCE], which acts both as a source and a sink. The VR3D editor is used to map the starting points of the objects, as well as the shape, size and all other visual aspects of the 3D environment. The actual 3D visuals can be found in Figure 3.3 where the transparent boxes are not physical objects but simply show the desired structure of the grey boxes. The data from the 3D environment, is then sent as a signal to the simulink model and to the control algorithm, which alters the signal and sends it back into the VR3D block to create an animation. The subsystems of the model which hold the source and the sink can be seen in Figure 3.2 whereas the contents of the sink, with the signal parsing, is shown in Figure 3.4

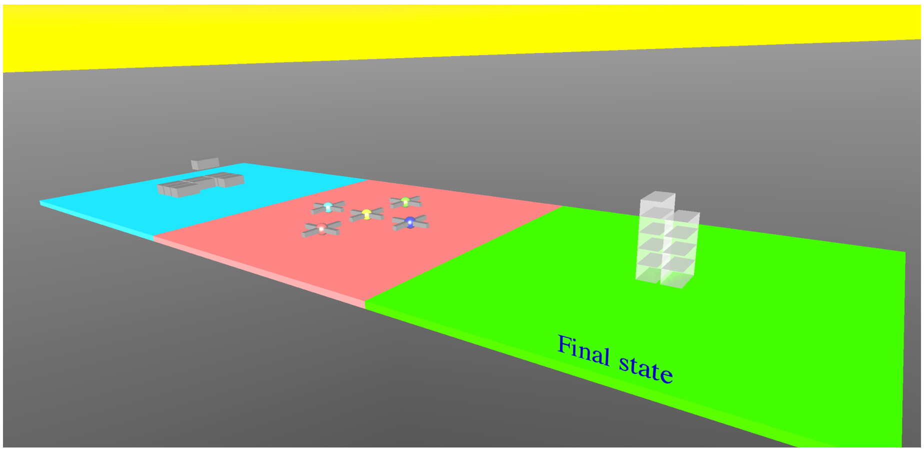

Figure 3.3: Screenshot from the VR3D environment used in the simulation

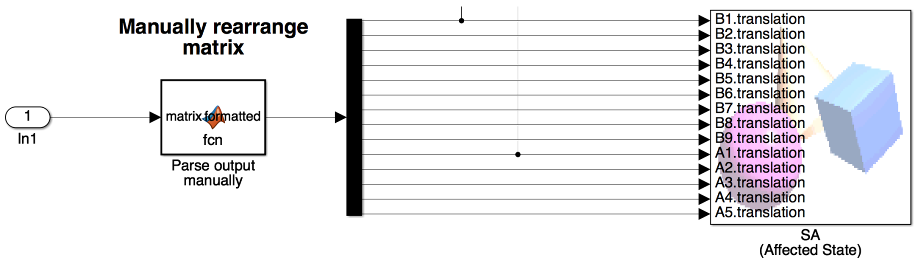

Figure 3.4: Signal from model being parsed to VR3D Sink

3.4.2 PD Controller

As mentioned in Section 3.1, it is important the swarm control algorithm cannot alter the position of the agents directly, but instead give instructions to the quadcopters to change their thrust etc. Distribution of control is implemented by having two separate control systems as briefly mentioned in 3.3.3. The two systems are shown in Figure 3.3, as blocks labeled `Local Control` and `Global`.

The `Global` Control System, controls and navigates the agents using the desired swarm algorithm. This Control block os what is referenced to throughout this thesis, as the *Control System*.

The `Local Control` is a subsystem (collection of Simulink blocks) responsible

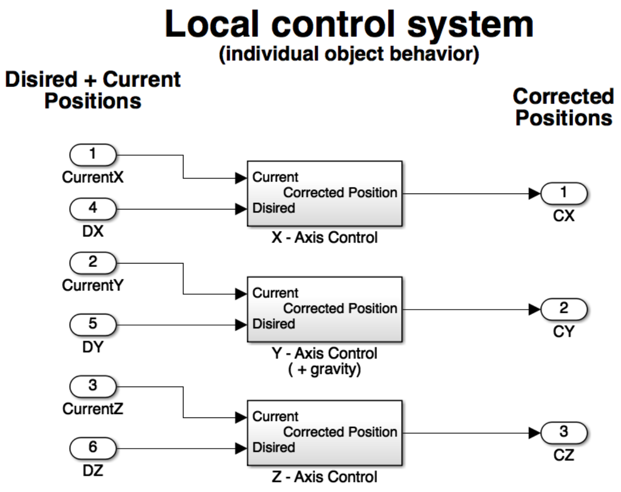

Figure 3.5: Local Control subsystem

for handling the physical mechanics of the individual objects within the simulation. It uses a Simulink PD plugin, to enable the quadcopters to balance and move around in a real life manner. In practice, the global control system maps the different agents' desired position at a given time, sends this as a signal to the local control system. The local control system then attempts to achieve the results within the physical boundaries of the simulation [INSERT REFERENCE]. Figure 3.5 shows the local control system receiving the current and desired locations from the global control system, and for making the appropriate alterations. For the process of adjusting the PD to mimic real world mechanics, please see Section 3.5.

3.4.3 Gravity & Mass

The impulse of moving objects and their mass affected by gravity, is implemented within the Local Control System. This is done with a collection of Simulink blocks, which have been adjusted to enable real life object behavior based on newton mechanics [INSERT REFERENCE]. Figure 3.6 shows the model used to control behavior on the Y-Axis of the simulation (vertical), which includes the gravitational acceleration constant of $9.8m/s^2$. The models for the X+Z Axis are similar, with the difference of not having a constant acceleration.

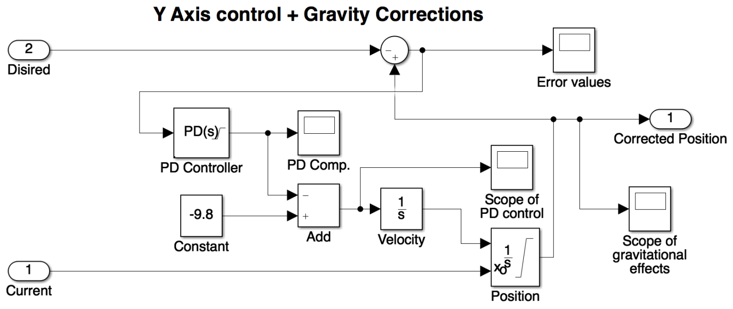

Figure 3.6: Simulink model to implement physics mechanics on vertical axis

3.4.4 Control Loop

The Control Loop is the part of the model labeled as `Global Control` in Figure 3.2. This Simulink block consists of a Matlab function and within this Matlab function, the control loop is implemented, giving actions to each agent based on their sensory inputs. This is where the swarm algorithms, to be outlined in Chapter 4, are implemented based on the agent's sensory inputs. As mentioned in Section 3.4.2 this iteratively communicates with the Local Control block. This block also computes the remaining part of the Control Problem, and sends a signal to the model to perform a simulation stop when the control problem is completed.

3.4.5 Data Stores

The Data Stores in Figure 3.2 work in connection with the Global Control Loop of the model, to store the data in between the iterations of the control loop. This is used to save the agent's sensory inputs over time, as well as the positions and other simulation data if required. This is needed due to the fact that Simulink Matlab function blocks, overwrites local variables for each computation of the control loop. As such these data stores are a way to use global variables in simulink.

3.4.6 Measurements

The measurements blocks in simulation model, are connected to the Global Control Loop, and perform different calculations on the progress of the simulation. The measurements are outlined in Section 3.6, and are exported to graphs such that different control algorithms can be compared. This is done with the Simulink Scope block.

3.4.7 Completion Check

Figure 3.7 shows the part of the simulation model which is responsible for stopping the simulation time when the control problem is solved completely or when the time limit of 500 seconds is reached. The control problem is solved once the blocks left to be moved correctly are equal to zero.

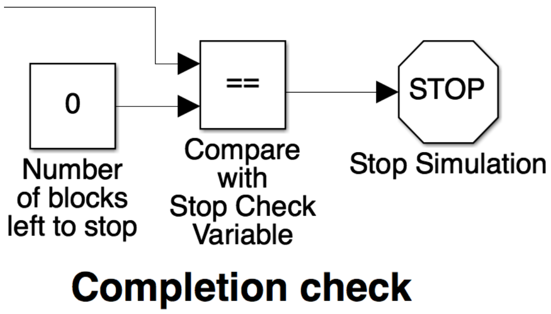
0
Number of blocks left to stop
==
Compare with Stop Check Variable
STOP
Stop Simulation
Completion check

Figure 3.7: Completion check of the model

3.5 Validation

The validation of the simulation is done to check the it can mimic desired real world behavior. For the purpose of this simulation, the desired behavior is outlined in the physical assumptions of Section 3.3.2. The simulation however cannot be run without a control algorithm to move the agents, and for the purpose of validation, I use a **Shortest Path** algorithm. This simply directs the agents to move in a straight line from the boxes to their desired location, until all boxes have been placed correctly. This can be predicted to cause collisions, as the straight lines will be closer to each other, than the agent's width. For validation purposes this is desired, as it will test the simulation's ability to detect collisions. The collisions detection will be tested with 5 agents and 9 boxes to rearrange, whereas the behavior of the physical mechanics will be tested only on a single agent. This is done as the properties of one agent are similar to multiple.

3.5.1 Adjusting for desired behavior

One of the most challenging aspects to adjust for real world like behavior, turnout out to be the Local Control system which handles the balancing of the agents and physics of the agents and boxes. In this subsection i will go through three iterations on the PD adjustments to make the quadcopters accelerate and brake in a realistic fashion. The word *realistic* is used loosely, as it simply refers to a behavior where the quadcopters can neither accelerate or break too quickly, but still maintain control. As mentioned in Section 3.4.2, this is how the effect of impulse in moving objects, is implemented. The figures of this subsection show a single agent's movement on the X-axis, as it solves the setup from Figure 3.3, which requires a lot of movement back and forth with breaking and acceleration.

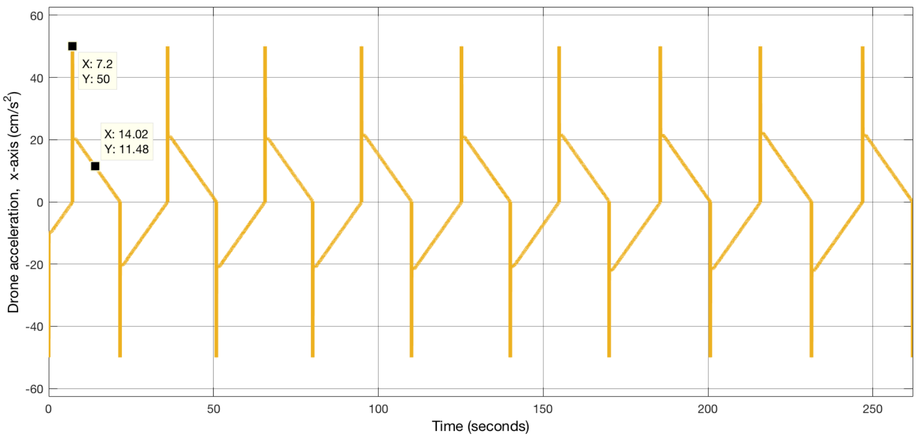

Figure 3.8: Agent acceleration before adjustment

Figure 3.8 shows the initial (or unadjusted) PD provided by Simulink, where the agent 'jolts' to an acceleration of 50 and then almost immediately, going back to linear decrease in acceleration. The desired behavior here is the symmetry of the graph as the reverse happens when moving in the other direction. The undesired part, is the large peak (or jolt) which resulted in a unstable movement in the simulation environment. To counter act this, the gain values of the controller were adjusted.

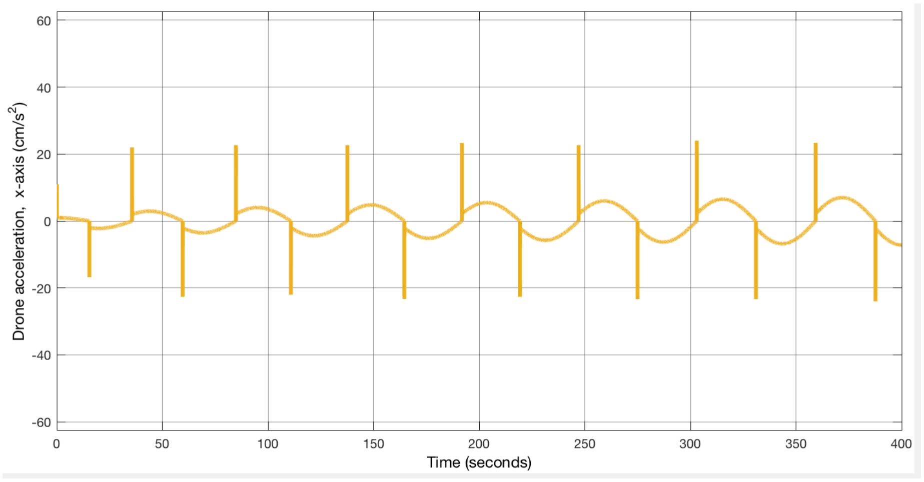

Figure 3.9: Agent acceleration after adjustment of PD gains

After some adjustment, the acceleration on the X axis where as shown in Figure 3.9. This can be seen to produce a peak acceleration of under half of the previous setting which resulted in a more stable movement. The curves show that the corrections where not instantaneous. The curves however grow in their peak values, which suggest that the controller is now too weak to remove energy from the system fast enough, and instead adds energy by over correcting.

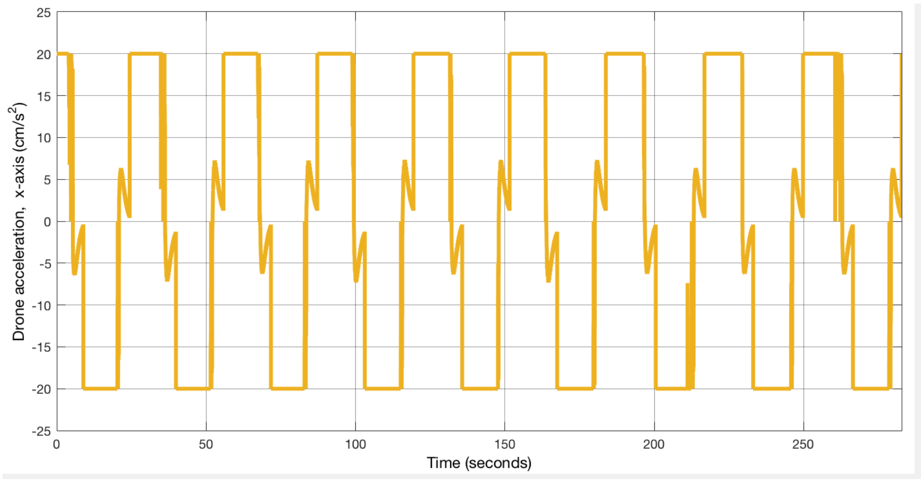

Figure 3.10: Agent acceleration after final adjustments

To combat this, it was effective to decrease the reaction time of the controller. The peaks are still relatively steep, however due to the faster reaction time, a much wider more stable acceleration curve is achieved. With this setup the peaks represent the movement from picking up a box to the placement point, where the small correction peaks represent breaking to pick up or drop off a box.

The final adjustments to the Local Control system PD, are as shown in Table ??.

Parameter	**Value**
P-Gain	1.6
D-Gain	1.8
Filter Coefficient	100
Acceleration Limit	+-20
Velocity Limit	50

Table 3.4: Local Control PD Settings

3.5.2 Gravity

To test the gravitational mechanics, a simulation is run with a single agent, moving 9 boxes to a desired location (identical box setup previously showed in Figure 3.3, only with a single drone). What can be seen in Figure 3.11, is the vertical position of the agent (yellow line), first falling unaffected and then being corrected by its thrusters to match a desired altitude. The blue line is the vertical position of a box, starting at a height of 10 cm and then falling to the floor height of 0. The simulation is not run to completion but only for approximately 7 seconds to show the gravitational acceleration. From the figure a slightly curvy correction can also be observed until the agent reaches a height of around 5, which is the PD controller working to stabilize the agent by adding thrust.

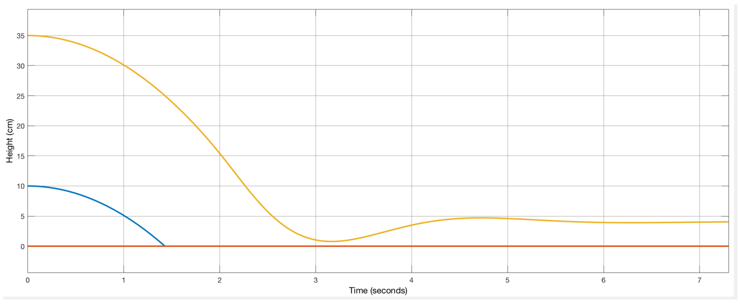

Figure 3.11: Box and agent movement on vertical axis over time

3.5.3 Collision Detections

One of the measurements taken by the simulation model, is the distance from each agent to its closest neighboring agent, from midpoint to midpoint. The way that the model detects collisions is when this distance is lower than the diameter of a single agent. This is tested on a shortest path algorithm and the agent distances over time as the simulation is completed is shown in Figure 3.12. As figure shows, collisions are not being avoided as the shortest paths for one agent, is almost directly next to the shortest path for another. The dotted black line represents the minimum distance before a collision and not surprisingly the agents are in almost constant collisions. The agent represented by the green line has a closer starting point to pick up a box and due to this head start, has fewer collision points. The Shortest Path is as such not a desirably control algorithm, however works nicely as a proof of concept for the collision detection and other mechanics, as it is simple to implement.

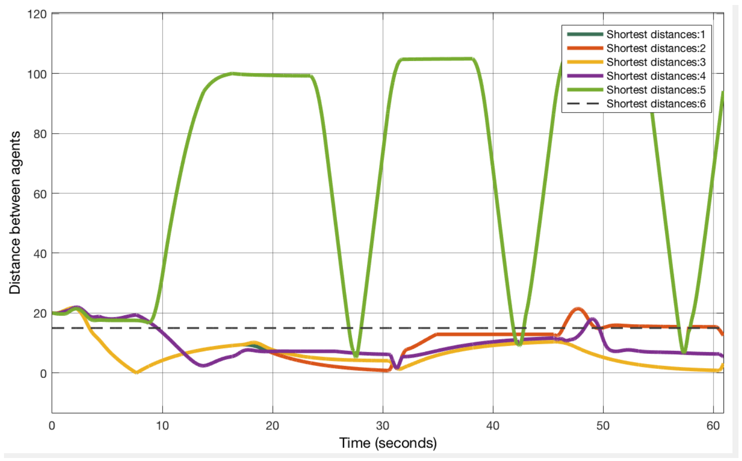

Figure 3.12: Control Algorithm: Shortest Path - Agent distances

3.6 Test Metrics

Some of the test metrics used to evaluate the performance of the Control Algorithms, have previously been mentioned. This section will outline all test metrics, used in this thesis, and provide a justification behind each metric. Table 3.5 shows the specific metrics.

Metric	Preference	Kind
Number of collisions	Lowest	*Quantitative*
Time spent completing control problem	Lowest	*Quantitative*
Utility of space	Lowest	*Quantitative*
Shape of flight pattern	N/A	*Qualitative*

Table 3.5: Algorithm Test Metrics

- **Number of collisions** will be the number of times an agent moves below the collision line, as showed in Figure 3.12
- **Time spend completing the control problem** refers to the time from the simulation starting, to the last block being placed in the control problem structure

- **Utility of space**, means how efficiently the agents move in proximity to each other. The lowest utility of space, is a level in which the agents move as close to each other as possible without triggering a collision
- **Shape of flight pattern** is a metric used to evaluate the performance in a qualitative way. Overviewing the flight pattern of the simulation on specific control algorithms, can sometimes give insight to issues, not easily visible on graphs or from measurements

Chapter 4

Swarm Logic & Swarm Algorithms

This Chapter will briefly outline some of the concepts used in this thesis to differentiate different types of swarm algorithms. It will also introduce the algorithms implemented as part of this thesis as well as the reasoning behind.

4.1 Multi-body vs. Multi-agent path planning

The distinction swarm and non-swarm intelligence in an algorithm is similar to the distinction multi-body and multi-agent planning [BLK10]. Multi-body planning refers to a centralized planning or control system in which the agents in the system are all controlled by a singular unit and as such all share the same information. Multi-agent planning however refers to a system in which the agents acting do not share the same information and acts as individuals, potentially with regards to the other agents. As such there is no master and no slaves and all agents are making decisions on their own.

Before a control algorithm can be categorized as a *Swarm Algorithm* [BLK10], it must function within a multi-agent environment and not a multi-body. The implementation of the control loop in the simulation used in this thesis, enables completely decentralized decision making by the agents, in order to house algorithms that fall into the swarm algorithm category.

4.1.1 Simulated decentralization

The decentralization of the agents is implemented in the global control loop of the simulation model. It is achieved by not letting the agents act on information which is gathered by other agents, other than what a particular algorithm allows. The decentralized aspect of the agents can be said to be simulated, because the agents do have access to the information of other agents, but is simply *told* not to use it, in order to simulate complete decentralization.

4.2 3-Dimensional Stigmergy

The first algorithm that will be tested will be s pheromone based swarm algorithm. Pheromone based control of agents is known as *Stigmergy* [Mas03] and has in previous works [Mason, Zachar], been used for 2-dimensional construction using multi-agents. With inspiration from this work, i will attempt to map a pheromone based algorithm onto the 3-dimensional simulation environment.

In practice, pheromones represent a biological trace of scent left by an agent, which can be detected by other agents. In the simulated environment, the pheromones will be digitally represented and simply act as the only information which can be shared between agents.

The aim of implementing a 3-dimensional stigmergic algorithm as a control algorithm, is to have a swarm algorithm which functions in space, controlling an n-number of agents, while still keeping the algorithmic complexity lower, than i.e. a self improving algorithm.

4.3 Layered Stigmergy

After implementing a 3-dimensional pheromone algorithm (See Chapter 5), the second algorithm i planned to implement is a revised version of the 3 dimensional stigmergy control algorithm. This choice was made based on the tests of the

3-dimensional stigmergy algorithm (see Section 6.1) and the potential areas of improvement identified.

The *Layered* aspect, comes from the amount of pheromones

Chapter 5

3-Dimensional Stigmergy

This chapter will outline the implementation attempt of the a stigmergy control algorithm, as well as some of the theory behind pheromone controlled agents.

5.1 Digital Markers (Pheromones)

In the world of biology, stigmergy refers to using pheromones as a means of communication between agents. Originally described by (Grasse 1959)[HM99] as the way that termites or ant colonies maintain efficiency and overall planning when building hives. Pheromones are put in place by the agents as a marker to reinforce the behavior of other agent. Pheromones can be weakened over time or amplified by other agents passing over them, and as such the influence of the pheromones on the environment are increased.

Several implementations of algorithms using stigmergy have been attempted in a 2-dimensional space, however my literature research found limited evidence of attempts of using stigmergy in 3 dimensions to control flying vehicles.

5.2 Implementation

5.2.1 Assumptions

The implementation of this stigmergy based algorithm, operates under the assumption that an agent can identify an object of interest when the object is within a certain distance from the agent. This is necessary for the stigmergy algorithm to work as a complete solution to navigating the agents, as the pheromones only serve to guide the agents in specific directions. As such, without the agents' ability to identify objects of interest, they would simply move in a more efficient manner, but without performing the desired actions when arriving at their desired location. As such, it is assumed that the agents know to pick up a block or put it down, when they find themselves at the location where this is supposed to happen.

This removes an element of complexity from the swarm algorithm, but is considered a fair assumption. It is therefore also assumed that this can be physically implemented with current sensor technology and object recognition, without negatively affecting the behavior of the stigmergy algorithm.

5.2.2 Specifications

There are multiple variables to consider when implementing a pheromone based control algorithm. As the pheromones only represent a common set of information shared by all agents, in an otherwise decentralized environment, it is important to consider how this information is interpreted by the agents. The first implementation attempt was done with the parameters identified in Table ??

Variables	**Value**
Pheromone grid size	$30 \times 10 \times 10$
Pheromone fatigue	$0/s$
Number of pheromone grids	$1/s$
Constant pheromone	0.2
Triggered pheromone	1
View radius of agent	$2 \times agent - radius$
Agent states	4

Table 5.1: Specifications of 3D Stigmergy implementation

- The **Pheromone grid** is an empty $30x10x10$ matrix, containing only zeroes as the initial value. This is how the pheromones will be represented in 3 dimensions. The size and shape of this array is the same as the simulation environment, and when a pheromone is placed in space, the value of the pheromone will change from zero to the desired pheromone value

- **Pheromone fatigue** refers to the decrease in all pheromones over time. This is used to determine how long a pheromone can remain *unused* before it fades and no longer has any effect on the behavior of the agents. The initial implementation was done with zero fatigue to observe the effect of having permanent pheromones

- **Constant pheromone** refers to the value that will be placed in the pheromone grid, just by an agent passing over it. **Triggered pheromone** will be the higher of a pheromone that the agent places in the grid, after having found an object of interest. Going back to the control problem, this can either be a block to be placed in the construction, or the area in which the block is supposed to be placed

- **View radius of agent** works on the assumptions , that any agent is able to identify an object of interest, as long as it is within a given radius

- **Agent states** is the number of states that an agent can be in, which in the case of the first implementation attempt is chosen to be four

- **Number of pheromone grids** translates into the number of different kinds of pheromones

5.2.3 State machine diagram

Figure 5.1 shows the state machine diagram of the agents as implemented in this stigmergy algorithm. The goal of the agent is (as described in the control problem section) to find and collect blocks, and place them in the desired (yet simple) construction. As the figure shows, the agent has 4 states, where the most important part of the figure, is the transition from the first to the second, and the third to the fourth state. In these transition the agent uses the pheromone grid to choose an area to search for either a block or a goal in. Goals refer to the desired position of a block.

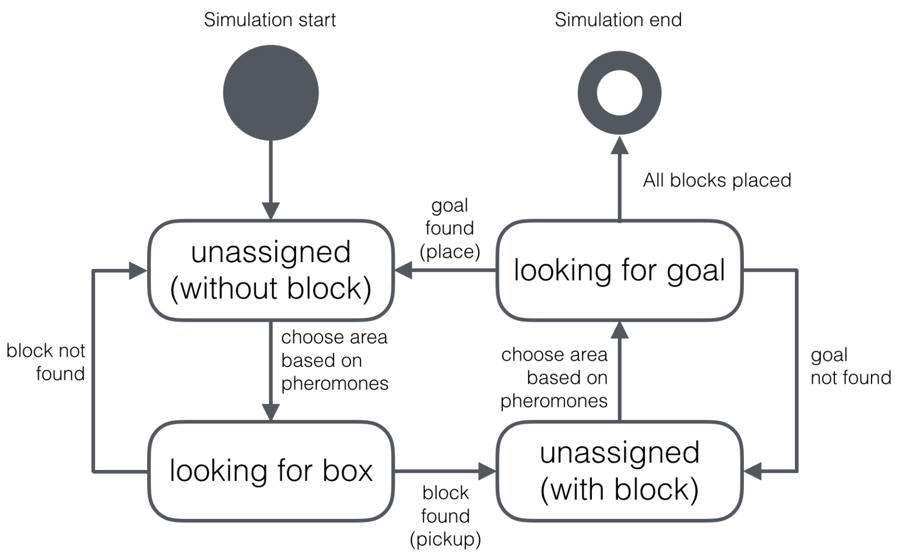

Figure 5.1: State machine diagram for an agent (including simulation start/finish)

5.2.4 Development of pheromones

The pheromones in the pheromone grid, are initially all given the value of zero. What this practically means is that there are not pheromones present in the simulation environment initially. In order for the pheromones to develop over time and affect the behavior of the agents, the agents themselves need to place them. This requires a set of general conditions under which the pheromones should be placed, which are outlined in Table **??**.

Situation	**Action**	**Condition**
Block found	place pheromone with value 1	none
Goal found	place pheromone with value -1	none
At any time	place pheromone with value 0.2	pheromone < 0.2

Table 5.2: Rules for pheromone placement

The negative pheromone value of the goal, represents a practical way of distinguishing pheromones that lead to the goals, from the pheromones that lead to blocks. This builds on an assumption of

5.2.5 Reacting to Pheromones

When the pheromones have started developing in the environment, the agents need to react to them for the pheromones to have any influence. As previously mentioned, this happens after state 1 and 3, which is shown in Figure 5.1.

The pheromones will influence the search of the agents and different strategies can be used to determine in what way. In the bullet points below i have listed some considerations that arose, from implementing effect of pheromones on the agents.

- **Limit search space -** One way that the agents can react to pheromones, is to only move along paths that have already been marked with a pheromone value. This is essentially still a random search in space, but within a more limited scope, which suggests that it could be faster than without pheromones
- **Strongest pheromone -** Instead of performing a random search throughout the environment, another simply way to use pheromones as guides is to go for the strongest pheromone (highest or lowest value depending on state). If the strongest pheromones are placed at the blocks and the goals over time, then these would be efficient to find if the agents are looking for the strongest pheromone
- **Distance effects -** A slightly more complex way of using the pheromones to navigate, is to perform a distance calculation to the pheromones in the grid, and make the pheromones closest appear stronger than the ones further away

Before any pheromone is placed in the grid, the search strategy for all agents is **random search** through out the level. The concrete implementation of the stigmergy algorithm, uses the a mix of all three strategies mentioned above, when a pheromone is present.

The actual Matlab code of the implementation can be found in Appendix A.

5.3 Testing

The following section includes the testing of the algorithm described above, and an evaluation of the test metrics outlined previously in Section 3.6

5.3.1 Pheromone grid

After running the algorithm on the agents, the agents where able to complete the control problem successfully. The pheromone grid continuously evolved throughout the simulation, and after execution, the pheromone grid had the values which are shown in Figure 5.2.

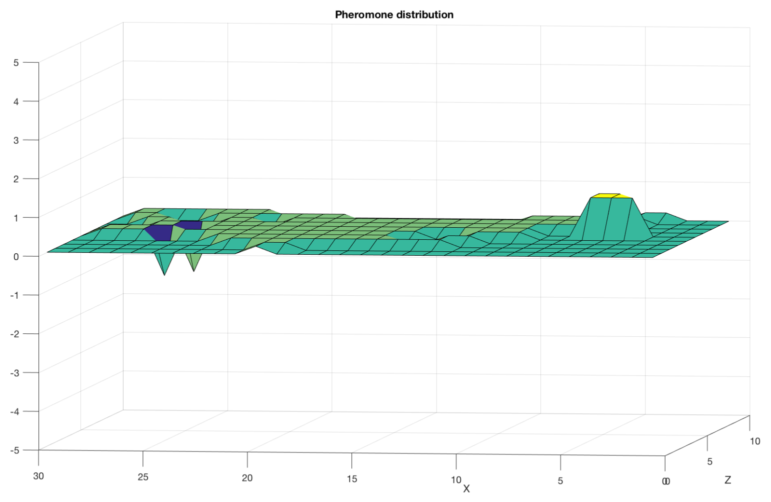

Figure 5.2: Pheromone grid after simulation has finished (mapped onto the horizontal dimensions)

5.3.2 Performance

Metric	Preference	Kind
Number of collisions	Lowest	18
Time spent completing control problem	Lowest	120.1*seconds*
Utility of space	Lowest	74%
Shape of flight pattern	N/A	See comments below

Table 5.3: 3D Stigmergy performance

- **Number of collisions** - The number of collisions of 18 is acceptable, for a solution spanning 120 seconds. However as Figure 6.3 shows, the time spent in collisions is quite high

- **Time spend completing the control problem** is twice as high as the shortest path algorithm tested in Section 3.6, however with a lot less collisions.

- **Utility of space -** This number is ideally balanced with the number of collisions. The utility of space is desired low (which indicates that more agents can be added without causing issues in obstacle avoidance), as long as it does not create collisions. As the collision number is already 18 over a long period of time, theres no desire to have a lower utility of space in this stigmergy algorithm

- **Shape of flight pattern -** For this 3D stigmergy algorithm, the flight pattern looks as expected. First the agents search in random locations, before quickly becoming more effective because of the pheromones. The randomness in the flight patterns can be seen in Figure 6.4, where the distances are not constantly minimized, but at multiple times seem to be increasing away from the goal. This happens when the agent does not have a strong enough pheromone follow, in the correct direction. This happens in the beginning, but as the pheromones are developed, the curve becomes steeper downwards, as desired

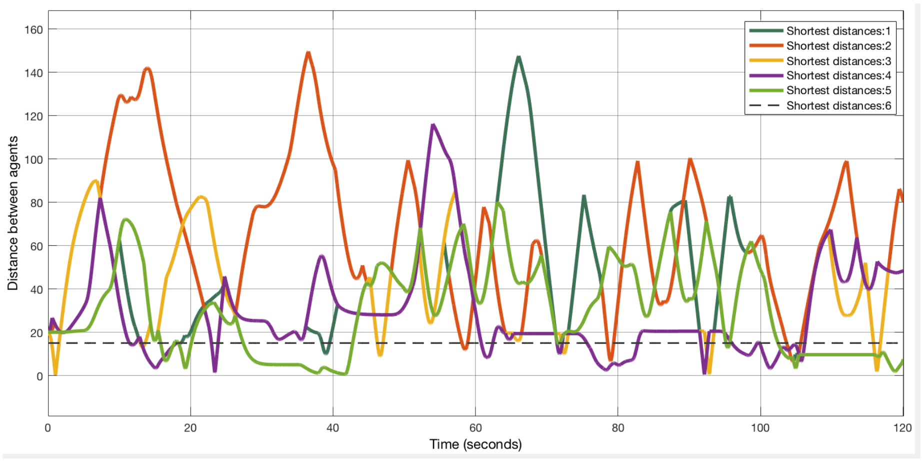

Figure 5.3: Collisions of agents throughout simulation

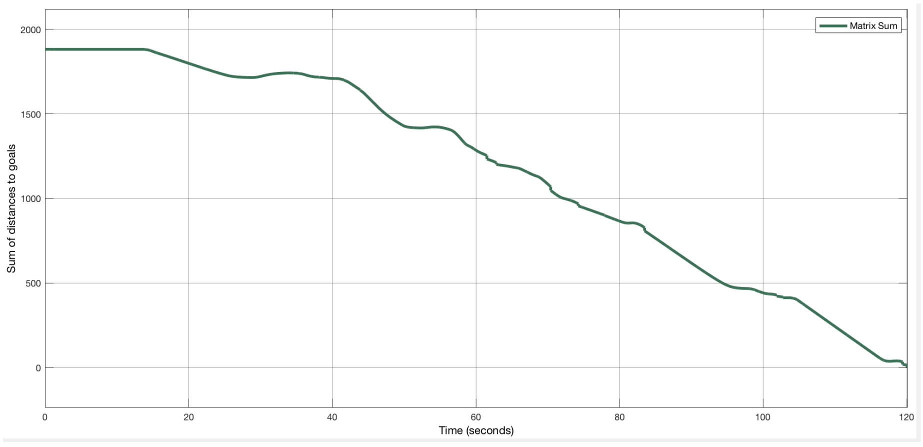

Figure 5.4: Total distances for blocks to goals over time

5.3.3 Improvement

The digital pheromones show to be having the desired effects as the curve in Figure 6.4, starts out relatively flat, and then steepens. This shows that the agents are going from moving randomly, to moving according to the pheromones, solving the level faster.

An area of improvement however is the number of collisions. For this i attempted to implement a layered version of the Stigmergy algorithm of this chapter, which is covered in Chapter 6.

Chapter 6

Layered Stigmergy

This chapter will outline the implementation of a multi layered stigmergy algorithm, as an attempt to create a swarm algorithm which better avoids collision than simple 1 layered stigmergy. The algorithm implemented in this chapter is identical to the Stigmergy algorithm from Chapter 5, in the way that it used the same pheromone grid to navigate its agents. It also uses the same reaction to the pheromones, and the same rules for placement.

The difference will be that this algorithm contains 2 pheromone grids instead of a single one.

6.0.1 Assumptions

Having two pheromone grids implies the assumption that the agents are able to tell two different kind of pheromones apart. Digitally this is represented as two different grids, but the effect is assumed also be achievable with a multitude of different concentration levels in a single pheromone grid, as long as the agents can differentiate between the concentrations.

6.0.2 Specifications

Table 6.1 shows the specifications of the second pheromone grid. Note that the specifications of the other pheromone grid is identical to one from previous chapter, along with the rest of the algorithm as well.

Variables	**Value**
Pheromone grid size	$30 \times 10 \times 10$
Pheromone fatigue	$2/s$
Number of pheromone grids	2
Constant pheromone	2
Triggered pheromone	0

Table 6.1: Specifications of 2nd pheromone grid

- The **Pheromone grid** is similar in size to the first pheromone grid
- **Pheromone fatigue** is high in the second grid, meaning that a pheromone will disappear approximately 1 second after it has been placed
- This second grid only operates with **Constant pheromones** in contrast to the first. This means that the agents always trail pheromones after them on this grid, but to not change the strength of their pheromone based on outside events
- **Number of pheromone grids** is now 2, hence the name *layered stigmergy* as there are more layers/kinds of pheromones

6.0.3 Development of pheromones in 2nd layer

The aim of the 2nd layer of pheromones, is to enable the agents to tell each other apart and evaluate a distance or direction to other agents. As such the pheromones disappear quickly after being placed, but have very simple rules. The only condition for placement of a pheromone in the second layer, is that an agent is at the same location as the pheromone being placed.

6.0.4 Reacting to Pheromones

The agents have been given the added condition, that any area they move into, must not have a pheromone in the second layer. As such, if there is a digital marker in the second grid at a position, an agent cannot be in that position. If the agent has itself placed the marker, the check happens before the placement, to avoid agents moving away from themselves.

When a marker near is detected in the 2nd, a search is recalculated based on the pheromones in the 1st grid.

6.1 Testing

6.1.1 Pheromone grids

The resulting pheromone grids from running the simulation are updated throughout the simulation. Figure 6.1 shows the 2nd layer of pheromones, which approximately denotes the agent positions. The faster the pheromones decay the more accurately the pheromones display the positions of the agents. From the position of the pheromones in the grid, it can be seen that they are not colliding.

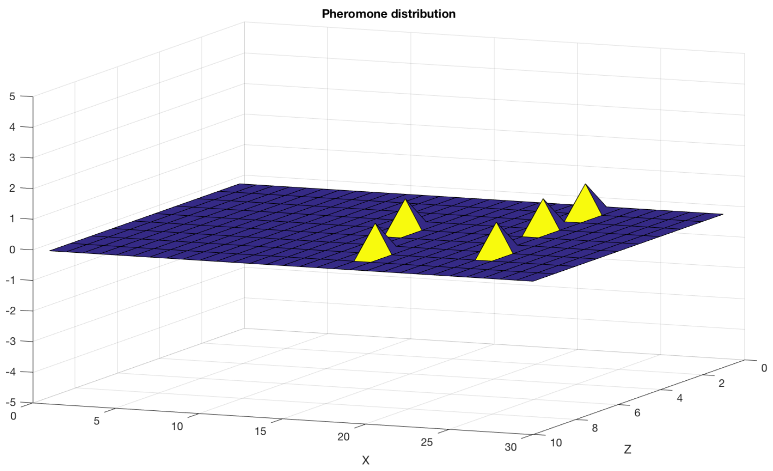

Figure 6.1: 2nd Pheromone layer (Pheromone sources) snapshot from simulation

The 1st pheromone grid, is roughly similar to grid from previous implementation, as it can be seen in Figure **??**.

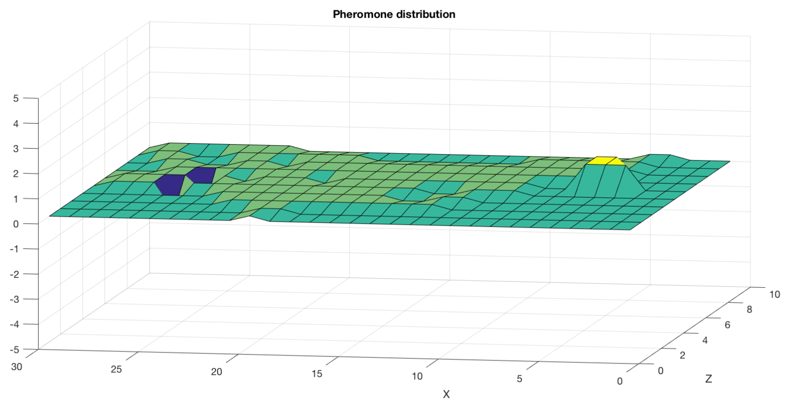

Figure 6.2: 1st pheromone grid after simulation has finished (mapped onto the horizontal dimensions)

6.1.2 Performance

The overall performance results of the layered stigmergy algorithm can be found in Figure 6.2.

Metric	**Preference**	**Kind**
Number of collisions	Lowest	23
Time spent completing control problem	Lowest	242.8*seconds*
Utility of space	Lowest	78%
Shape of flight pattern	N/A	See comments below

Table 6.2: Layered Stigmergy performance

- **Number of collisions -** The aim of implementing a second pheromone layer, was to decrease the number of collision. This was unsuccessful, however the time spent in a collision is a lot lower than in the non-layered stigmergy algorithm. As such, the layered stigmergy can be seen to have a positive impact in the fact that the agent move away from each other when colliding.
- **Time spend completing the control problem -** With over a doubling in the simulation time, it is possible to argue that the benefit of lower collision time, is worth the efficiency drawback

- **Utility of space -** 4% higher than the non-layered stigmergy algorithm, can be considered a fine result, since the collision avoidance requires more space to maneuver within

- **Shape of flight pattern -** The flight pattern is very similar to the flight pattern on the non-layered stigmergy algorithm. The difference is when agents attempt to avoid a collision, and spend a long time moving in the opposite direction of their desired location. This result is due to the fact that there is no optimization to the collision avoidance. As such, when avoiding a collision, an agent is simple reattempting a search in a different direction, as it normally would as a part of the swarm

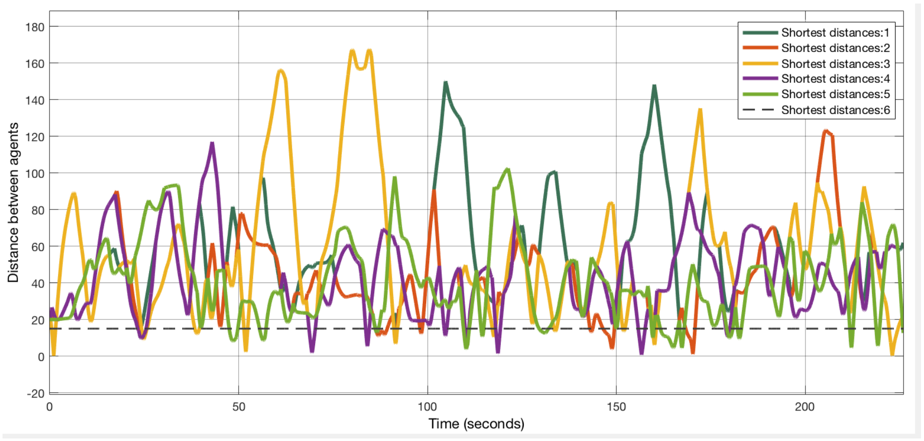

Figure 6.3: Collisions of agents throughout simulation, with layered stigmergy

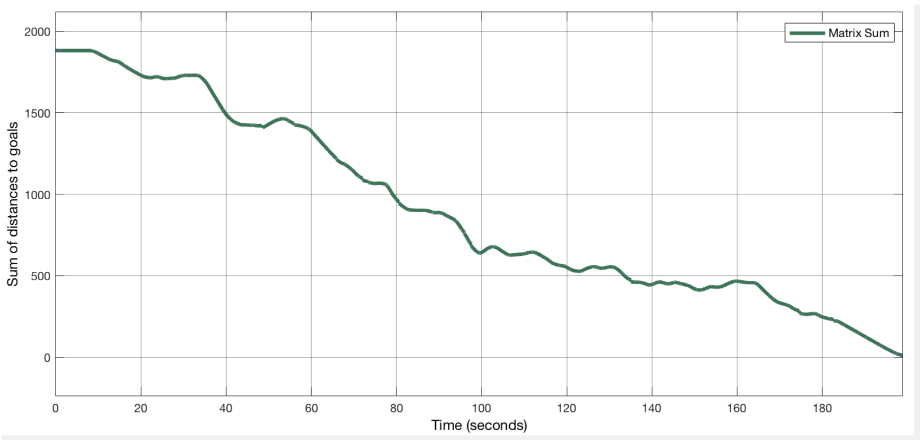

Figure 6.4: Total distances for blocks to goals over time

6.1.3 Improvement

The digital pheromones show to be having the desired effects as the curve in Figure 6.4, starts out relatively flat, and then steepens. This shows that the agents are going from moving randomly, to moving according to the pheromones, solving the level faster.

An area of improvement however is the number of collisions. For this i attempted to implement a layered version of the Stigmergy algorithm of this chapter, which is covered in Chapter 6.

Chapter 7

Appendix A

The following appendix contains the control loop for the non-layered 3-Dimensional Stigmergy implementation.

```
for i = 1:length(agents_at)

        % Decrease pheromones

        index_of_agent = agents_at(i);

        % Agent Positions
        agent_x = CX(index_of_agent);
        agent_y = CY(index_of_agent);
        agent_z = CZ(index_of_agent);
        agent_pos = [agent_x agent_y agent_z];

        % Pheromone Index
        p_index_x = floor((agent_x/10)+16);
        p_index_y = floor((agent_y/10)+6);
        p_index_z = floor((agent_z/10)+6);

        % Radius limiters when close to envirornment edge
        z_neg = 1;
        z_pos = 1;
        x_neg = 1;
        x_pos = 1;
        if p_index_z <= 1
            p_index_z = 1;
            z_neg = 0;
        elseif p_index_z >= 10
            p_index_z = 10 ;
            z_pos = 0;
        end

        if p_index_y <= 1
            p_index_y = 1;
        elseif p_index_y >= 10
            p_index_y = 10 ;
        end

        if p_index_x <= 1
            p_index_x = 1;
            x_neg = 0;
        elseif p_index_x >= 30
            p_index_x = 30;
            x_pos = 0;
        end

        curr_p_index = [p_index_x, p_index_y, p_index_z];

        % Value of current pheromone
        % 0: Empty, 1:
        cur_p_value = pheromones(p_index_x, search_height , p_index_z);

       % pheromones(:) = 0;
        % Setting current value to something
        % pheromones(p_index_x, p_index_y, p_index_z) = 0;


        % Assign movement towards box
        if agents_moving_to(i,1) == 0

            % Value, Index of highest element
            [maxval, maxind] = max(pheromones(:));
            [maxxidx, maxyidx, maxzidx] = ind2sub(size(pheromones),maxind);


            % Agent data
            % 1: State
            % 2,3,4: Coordinates of desired movement
            % 5: Box to carry
            % 6: Pheromoned or random (0:random, 1:p, 2:new p)

            if maxval == 0 || agents_moving_to(i, 6) == 2
                agents_moving_to(i, 2) = randi([-149 149],1,1);
                agents_moving_to(i, 3) = search_height;
                agents_moving_to(i, 4) = randi([-49 49],1,1);
                agents_moving_to(i, 6) = 0;

            elseif agents_moving_to(i, 6) == 0;
            disp('Global␣Pheromone')
            agents_moving_to(i, 6) = 1;
            agents_moving_to(i, 2) = maxxidx*10-150;
            agents_moving_to(i, 3) = search_height;
            agents_moving_to(i, 4) = maxzidx*10-50;
        else
             disp('Local␣random')
             agents_moving_to(i, 6) = 2;
             agents_moving_to(i, 2) = randi([agent_x-20 agent_x+20],1,1);
             agents_moving_to(i, 3) = search_height;
             agents_moving_to(i, 4) = randi([agent_z-20 agent_z+20],1,1);
        end

        agents_moving_to(i, 1) = 1;

        % Figure
        Y=squeeze(pheromones(:,search_height,:));

        x_offset = Y;
        x_offset2 = Y2;

        figure(1), surf(x_offset);
        xlabel('Z'), ylabel('X'), title('Pheromone␣distribution');
        axis([0 10 0 30 -5 5]);

    elseif agents_moving_to(i,1) == 1

        DX(index_of_agent) = agents_moving_to(i, 2);
        DY(index_of_agent) = agents_moving_to(i, 3);
        DZ(index_of_agent) = agents_moving_to(i, 4);

        C = [CX(index_of_agent) CY(index_of_agent) CZ(index_of_agent)];

        % Boxes
        box_dist = agent_view_radius;
        closest_box_index = 0;
        for b = 1:last_box_at
            dist = max(abs([CX(b) CY(b) CZ(b)] - C));
            if blocks_moving_to(b) == 0 && dist <= box_dist
                box_dist = dist;
                closest_box_index = b;
            end
        end

        if closest_box_index > 0
            DX(index_of_agent) = CX(closest_box_index);
            DY(index_of_agent) = CY(closest_box_index) + block_height;
            DZ(index_of_agent) = CZ(closest_box_index);
        end

        D = [DX(index_of_agent) DY(index_of_agent) DZ(index_of_agent)];

        % Check if they are standing still, reset
        if max(abs(C-D)) < stand_still_threshold
            if closest_box_index > 0
```

```
                agents_moving_to(i, 1) = 2;
                blocks_moving_to(closest_box_index) = 1;
                agents_moving_to(i, 5) = closest_box_index;
                pheromones(p_index_x, search_height, p_index_z) = 1;
            elseif agents_moving_to(i, 6) == 0; % if not pheromoned
                agents_moving_to(i, 1) = 0;
                disp('Back␣to␣search␣1');
            else
                agents_moving_to(i, 1) = 0;
                disp('Back␣to␣search␣0');
            end
        elseif cur_p_value == 0;
            pheromones(p_index_x, search_height, p_index_z)  = 0.2;
        end


    % Assign movement towards goal
    elseif agents_moving_to(i,1) == 2

        % Value, Index of highest element
        [maxval, maxind] = min(pheromones(:));
        [maxxidx, maxyidx, maxzidx] = ind2sub(size(pheromones),maxind);

        if maxval == 0 || agents_moving_to(i, 6) == 2
            agents_moving_to(i, 2) = randi([-149 149],1,1);
            agents_moving_to(i, 3) = search_height;
            agents_moving_to(i, 4) = randi([-49 49],1,1);
            agents_moving_to(i, 6) = 0;

        elseif agents_moving_to(i, 6) == 0;
            disp('Global␣Pheromone')
            agents_moving_to(i, 6) = 1;
            agents_moving_to(i, 2) = maxxidx*10-150;
            agents_moving_to(i, 3) = search_height;
            agents_moving_to(i, 4) = maxzidx*10-50;
        else
             disp('Local␣random')
             agents_moving_to(i, 6) = 2;
             agents_moving_to(i, 2) = randi([agent_x-20 agent_x+20],1,1);
             agents_moving_to(i, 3) = search_height;
             agents_moving_to(i, 4) = randi([agent_z-20 agent_z+20],1,1);
        end

        agents_moving_to(i, 1) = 3;

        % Figure
        Y=squeeze(pheromones(:,search_height,:));
        x_offset = Y;
        figure(1), surf(x_offset);
        xlabel('Z'), ylabel('X'), title('Pheromone␣distribution');
        axis([0 10 0 30 -5 5])


    elseif agents_moving_to(i,1) == 3
```

```
            index_of_carried = agents_moving_to(i,5);

            CX(index_of_carried) = CX(agents_at(i));
            CY(index_of_carried) = CY(agents_at(i))-block_height*.5;
            CZ(index_of_carried) = CZ(agents_at(i));

            C = [CX(index_of_agent) CY(index_of_agent) CZ(index_of_agent)];

            % Goals
            goal_dist = agent_view_radius;
            closest_goal_index = 0;
            for b = 1:last_box_at
                dist = max(abs([EX(b) EY(b) EZ(b)] - C));
                if dist <= goal_dist
                    goal_dist = dist;
                    closest_goal_index = b;
                end
            end

            % Check if they are standing still, reset
            if closest_goal_index > 0
                DX(index_of_agent) = EX(closest_goal_index);
                DY(index_of_agent) = EY(closest_goal_index)+block_height;
                DZ(index_of_agent) = EZ(closest_goal_index);
            else
                DX(index_of_agent) = agents_moving_to(i, 2);
                DY(index_of_agent) = agents_moving_to(i, 3);
                DZ(index_of_agent) = agents_moving_to(i, 4);
            end

            D = [DX(index_of_agent) DY(index_of_agent) DZ(index_of_agent)];

            % Check if they are standing still, reset
            if max(abs(C-D)) < stand_still_threshold
                if closest_goal_index > 0
                    agents_moving_to(i, 1) = 0;
                    agents_moving_to(i, 5) = 0;
                    blocks_moving_to(index_of_carried) = 2;
                    pheromones(p_index_x, search_height, p_index_z)
= - 1;
                elseif agents_moving_to(i, 6) == 0; % if not pheromoned
                    agents_moving_to(i, 1) = 2;
                    disp('Back␣to␣search␣3');
                else
                    agents_moving_to(i, 1) = 2;
                    disp('Back␣to␣search␣4');
                end
            end



        end
    end
```

Chapter 8

Appendix B

The following appendix contains the control loop for the layered 3-Dimensional Stigmergy implementation.

```
% Assign Desired Positions
 for i = 1:length(agents_at)

     % Decrease pheromones

     index_of_agent = agents_at(i);

     % Agent Positions
     agent_x = CX(index_of_agent);
     agent_y = CY(index_of_agent);
     agent_z = CZ(index_of_agent);
     agent_pos = [agent_x agent_y agent_z];

     % Pheromone Index
     p_index_x = floor((agent_x/10)+16);
     p_index_y = floor((agent_y/10)+6);
     p_index_z = floor((agent_z/10)+6);

     % Radius limiters when close to envirornment edge
     z_neg = 1;
     z_pos = 1;
```

```
        x_neg = 1;
        x_pos = 1;

        if p_index_z <= 1
            p_index_z = 1;
            z_neg = 0;
        elseif p_index_z >= 10
            p_index_z = 10 ;
            z_pos = 0;
        end

        if p_index_y <= 1
            p_index_y = 1;
        elseif p_index_y >= 10
            p_index_y = 10 ;
        end

        if p_index_x <= 1
            p_index_x = 1;
            x_neg = 0;
        elseif p_index_x >= 30
            p_index_x = 30;
            x_pos = 0;
        end

        curr_p_index = [p_index_x, p_index_y, p_index_z];

        % Value of current pheromone
        % 0: Empty, 1:
        cur_p_value = pheromones(p_index_x, search_height, p_index_z);

      % pheromones(:) = 0;
        % Setting current value to something
        % pheromones(p_index_x, p_index_y, p_index_z) = 0;


        % Assign movement towards box
        if agents_moving_to(i,1) == 0

            % Value, Index of highest element
            [maxval, maxind] = max(pheromones(:));
            [maxxidx, maxyidx, maxzidx] = ind2sub(size(pheromones),maxind);


            % Agent data
            % 1: State
            % 2,3,4: Coordinates of desired movement
            % 5: Box to carry
            % 6: Pheromoned or random (0:random, 1:p, 2:new p)

            if maxval == 0 || agents_moving_to(i, 6) == 2
                agents_moving_to(i, 2) = randi([-149 149],1,1);
                agents_moving_to(i, 3) = search_height;
                agents_moving_to(i, 4) = randi([-49 49],1,1);
                agents_moving_to(i, 6) = 0;
            elseif agents_moving_to(i, 6) == 0;
                disp('Global Pheromone')
                agents_moving_to(i, 6) = 1;
                agents_moving_to(i, 2) = maxxidx*10-150;
                agents_moving_to(i, 3) = search_height;
                agents_moving_to(i, 4) = maxzidx*10-50;
            else
                 disp('Local random')
                 agents_moving_to(i, 6) = 2;
                 agents_moving_to(i, 2) = randi([agent_x-20 agent_x+20],1,1);
                 agents_moving_to(i, 3) = search_height;
                 agents_moving_to(i, 4) = randi([agent_z-20 agent_z+20],1,1);
            end

            agents_moving_to(i, 1) = 1;

            % Figure
            Y=squeeze(pheromones(:,search_height,:));
            Y2=squeeze(pheromone_sources(:,search_height,:));

            x_offset = Y;
            x_offset2 = Y2;

            figure(1), surf(x_offset);
            xlabel('Z'), ylabel('X'), title('Pheromone distribution');
            axis([0 10 0 30 -5 5]);

            figure(2), surf(x_offset2);
            xlabel('Z'), ylabel('X'), title('Pheromone distribution');
            axis([0 10 0 30 -5 5]);

        elseif agents_moving_to(i,1) == 1

            DX(index_of_agent) = agents_moving_to(i, 2);
            DY(index_of_agent) = agents_moving_to(i, 3);
            DZ(index_of_agent) = agents_moving_to(i, 4);

            C = [CX(index_of_agent) CY(index_of_agent) CZ(index_of_agent)];

            % Boxes
            box_dist = agent_view_radius;
            closest_box_index = 0;
            for b = 1:last_box_at
                dist = max(abs([CX(b) CY(b) CZ(b)] - C));
                if blocks_moving_to(b) == 0 && dist <= box_dist
                    box_dist = dist;
                    closest_box_index = b;
                end
            end

            if closest_box_index > 0
                DX(index_of_agent) = CX(closest_box_index);
                DY(index_of_agent) = CY(closest_box_index) + block_height;
                DZ(index_of_agent) = CZ(closest_box_index);
            end

            D = [DX(index_of_agent) DY(index_of_agent) DZ(index_of_agent)];

            % Check if they are standing still , reset
            if max(abs(C-D)) < stand_still_threshold
                if closest_box_index > 0
                    agents_moving_to(i, 1) = 2;
                    blocks_moving_to(closest_box_index) = 1;
                    agents_moving_to(i, 5) = closest_box_index;
                    pheromones(p_index_x, search_height, p_index_z) = 1;
                elseif agents_moving_to(i, 6) == 0; % if not pheromoned
                    agents_moving_to(i, 1) = 0;
                    disp('Back␣to␣search␣1');
                else
                    agents_moving_to(i, 1) = 0;
                    disp('Back␣to␣search␣0');
                end
            elseif cur_p_value == 0;
                pheromones(p_index_x, search_height, p_index_z)  = 0.2;
            end

            % Pheromone sources
            pheromone_sources(p_index_x, search_height, p_index_z) = 0;
            taken = find(pheromone_sources(p_index_x-x_neg:p_index_x+x_pos,
                        search_height, p_index_z-z_neg:p_index_z+z_pos),1);
            pheromone_sources(p_index_x, search_height, p_index_z) = 1;



            pheromone_sources(p_index_x-x_neg:p_index_x+x_pos,
                              search_height, p_index_z-z_neg:p_index_z+z_pos) = 0;
            pheromone_sources(p_index_x, search_height, p_index_z) = 1;





        % Assign movement towards goal
        elseif agents_moving_to(i,1) == 2

            % Value, Index of highest element
            [maxval, maxind] = min(pheromones(:));
            [maxxidx, maxyidx, maxzidx] = ind2sub(size(pheromones),maxind);

            if maxval == 0 || agents_moving_to(i, 6) == 2
                agents_moving_to(i, 2) = randi([-149 149],1,1);
                agents_moving_to(i, 3) = search_height;
                agents_moving_to(i, 4) = randi([-49 49],1,1);
                agents_moving_to(i, 6) = 0;

            elseif agents_moving_to(i, 6) == 0;
                disp('Global␣Pheromone')
                agents_moving_to(i, 6) = 1;
                agents_moving_to(i, 2) = maxxidx*10-150;
                agents_moving_to(i, 3) = search_height;
                agents_moving_to(i, 4) = maxzidx*10-50;
            else
                 disp('Local␣random')
                 agents_moving_to(i, 6) = 2;
                 agents_moving_to(i, 2) = randi([agent_x-20 agent_x+20],1,1);
                 agents_moving_to(i, 3) = search_height;
                 agents_moving_to(i, 4) = randi([agent_z-20 agent_z+20],1,1);
            end

            agents_moving_to(i, 1) = 3;

            % Figure
            Y=squeeze(pheromones(:,search_height,:));
            x_offset = Y;
            figure(1), surf(x_offset);
            xlabel('Z'), ylabel('X'), title('Pheromone␣distribution');
            axis([0 10 0 30 -5 5])


        elseif agents_moving_to(i,1) == 3

            index_of_carried = agents_moving_to(i,5);

            CX(index_of_carried) = CX(agents_at(i));
            CY(index_of_carried) = CY(agents_at(i))-block_height*.5;
            CZ(index_of_carried) = CZ(agents_at(i));

            C = [CX(index_of_agent) CY(index_of_agent) CZ(index_of_agent)];

            % Goals
            goal_dist = agent_view_radius;
            closest_goal_index = 0;
            for b = 1:last_box_at
                dist = max(abs([EX(b) EY(b) EZ(b)] - C));
                if dist <= goal_dist
                    goal_dist = dist;
                    closest_goal_index = b;
                end
            end

            % Check if they are standing still, reset
            if closest_goal_index > 0
                DX(index_of_agent) = EX(closest_goal_index);
                DY(index_of_agent) = EY(closest_goal_index)+block_height;
                DZ(index_of_agent) = EZ(closest_goal_index);
            else
                DX(index_of_agent) = agents_moving_to(i, 2);
                DY(index_of_agent) = agents_moving_to(i, 3);
                DZ(index_of_agent) = agents_moving_to(i, 4);
            end

            D = [DX(index_of_agent) DY(index_of_agent) DZ(index_of_agent)];

            % Check if they are standing still, reset
            if max(abs(C-D)) < stand_still_threshold
                if closest_goal_index > 0
                    agents_moving_to(i, 1) = 0;
                    agents_moving_to(i, 5) = 0;
                    blocks_moving_to(index_of_carried) = 2;
                    pheromones(p_index_x, search_height, p_index_z)
= - 1;
                elseif agents_moving_to(i, 6) == 0; % if not pheromoned
                    agents_moving_to(i, 1) = 2;
                    disp('Back␣to␣search␣3');
                else
                    agents_moving_to(i, 1) = 2;
                    disp('Back␣to␣search␣4');
                end
            end


            % Sources of pheromones
            pheromone_sources(p_index_x-x_neg:p_index_x+x_pos,
                search_height, p_index_z-z_neg:p_index_z+z_pos) = 0;
            pheromone_sources(p_index_x, search_height, p_index_z) = 1;

        end
    end
```

Bibliography

[ALH$^+$14] F. Augugliaro, S. Lupashin, M. Hamer, C. Male, M. Hehn, M. W. Mueller, J. S. Willmann, F. Gramazio, M. Kohler, and R. D'Andrea. The Flight Assembled Architecture installation: Cooperative construction with flying machines. *IEEE Control Systems*, 34(4):46–64, August 2014.

[AMBR15] J. Alonso-Mora, S. Baker, and D. Rus. Multi-robot navigation in formation via sequential convex programming. In *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 4634–4641, September 2015.

[BLK10] Subhrajit Bhattacharya, Maxim Likhachev, and Vijay Kumar. Multi-agent path planning with multiple tasks and distance constraints. In *ICRA*, pages 953–959, 2010.

[BRG$^+$11] Preetha Bhattacharjee, Pratyusha Rakshit, Indrani Goswami, Amit Konar, and Atulya K. Nagar. Multi-robot path-planning using artificial bee colony optimization algorithm. In *Nature and Biologically Inspired Computing (NaBIC), 2011 Third World Congress on*, pages 219–224. IEEE, 2011.

[CPD$^+$14] Ruaridh Clark, Giuliano Punzo, Gordon Dobie, Rahul Summan, Charles Norman MacLeod, Gareth Pierce, and Malcolm Macdonald. Autonomous swarm testbed with multiple quadcopters. In *1st World Congress on Unmanned Systems Enginenering, 2014-WCUSEng*, 2014.

[DHB15] Shreyansh Daftry, Christof Hoppe, and Horst Bischof. Building with drones: Accurate 3d facade reconstruction using mavs.

In *Robotics and Automation (ICRA), 2015 IEEE International Conference on*, pages 3487–3494. IEEE, 2015.

[HM99] Owen Holland and Chris Melhuish. Stigmergy, self-organization, and sorting in collective robotics. *Artificial life*, 5(2):173–202, 1999.

[LGB+16] Pierre LATTEUR, Sébastien GOESSENS, Jean-Sébastien BRETON, Justin LEPLAT, Zhao MAb, and Caitlin MUELLER. Drone-Based Additive Manufacturing of Architectural Structures. 2016.

[Mas03] Zachary Mason. Programming with stigmergy: using swarms for construction. In *ICAL 2003: Proceedings of the eighth international conference on Artificial life*, pages 371–374, 2003.

[mat16] matlab matlab. MATLAB and Simulink for Student Use - Math software for engineering and science students - MathWorks Nordic, 2016.

[PKG+02] Giovanni Cosimo Pettinaro, I. Kwee, Luca Maria Gambardella, Francesco Mondada, Dario Floreano, Stefano Nolfi, J.-L. Deneubourg, and Marco Dorigo. Swarm robotics: A different approach to service robotics. In *33rd International Symposium on Robotics*, pages 71–76, 2002.

[SMVDPB05] John A. Sauter, Robert Matthews, H. Van Dyke Parunak, and Sven A. Brueckner. Performance of digital pheromones for swarming vehicle control. In *Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems*, pages 903–910. ACM, 2005.