Tractable Robot Simulation for Terrain Leveling

by

© **Daniel Cook**

Supervisor: Andrew Vardy

A thesis submitted to the School of Gradate Studies in partial fulfillment of the requirements for the degree of Master of Engineering.

Faculty of Engineering and Applied Science

Memorial University

October 2017

St. John's, Newfoundland and Labrador, Canada

Abstract

This thesis describes the problem of terrain leveling, in which one or more robots or vehicles are used to flatten a terrain. The leveling operation is carried out either in preparation for construction, or for terrain reparation. In order to develop and prototype such a system, the use of simulation is advantageous. Such a simulation requires high fidelity to accurately model earth moving robots, which navigate uneven terrain and potentially manipulate the terrain itself. It has been found that existing tools for robot simulation typically do not adequately model deformable and/or uneven terrain. Software which does exist for this purpose, based on a traditional physics engine, is difficult if not impossible to run in real-time while achieving the desired accuracy. A number of possible approaches are proposed for a terrain leveling system using autonomous mobile robots. In order to test these approaches in simulation, a 2D simulator called Alexi has been developed, which uses the predictions of a neural network rather than physics simulation, to predict the motion of a vehicle and changes to a terrain. The neural network is trained using data captured from a high-fidelity non-real-time 3D simulator called Sandbox. Using a trained neural network to drive the 2D simulation provides considerable speed-up over the high-fidelity 3D simulation, allowing behaviour to be simulated in real-time while still capturing the physics of the agents and the environment. Two methods of simulating terrain in Sandbox are explored with results related to performance given for each. Two variants of Alexi are also explored, with results related to neural network training and generalization provided.

Acknowledgements

I would like to thank my supervisor, Dr. Andrew Vardy, for encouraging me to continue my studies at the graduate level and providing me with guidance and flexibility during my research program. I would also like to thank Ron Lewis for his words of advice throughout my program.

The work presented in this thesis was supported in part through funding from the REsponsive AUV Localization and Mapping (REALM) project at Memorial University of Newfoundland.

Finally, I would like to offer special thanks to my girlfriend Keri Knox, and my parents Arlene and Michael Cook for their patience, encouragment, and support.

Table of contents

Title page **i**

Abstract **ii**

Acknowledgements **iii**

Table of contents **iv**

List of tables **vii**

List of figures **viii**

List of Algorithms **x**

List of abbreviations **xi**

1 Introduction **1**

1.1 Contributions 6

1.2 Scope of Work and Thesis Overview 7

2 Literature Review **9**

2.1 Robotic Construction and Site Preparation 9

2.2 General-Purpose Robot Simulation 12

2.3 Neural Network-Based Simulation 15

2.4 Summary 17

3 Background **19**

3.1 Artificial Neural Networks 19

3.2 Simulation of Robots and Vehicles 23

3.3 Summary 24

4 Terrain Leveling **25**
4.1 Defining the Leveling Task 25
4.2 Terrain Leveling Algorithms 27
4.2.1 Probabilistic Algorithm 27
4.2.2 Results: Probabilistic Algorithm 30
4.2.3 Memetic Algorithm 32
4.3 Prototype of a Leveling Robot 36
4.3.1 Hardware 37
4.3.2 Noted Issues 39
4.3.3 Alternate Platform 40
4.4 Discussion 41
4.5 Summary 42

5 High-Fidelity Simulation of a Leveling Robot **43**
5.1 Architecture 45
5.1.1 Assembly 47
5.1.2 Platform 48
5.1.3 Terrain 48
5.1.4 Experiment 50
5.2 Defining a Robot Model 52
5.3 Generating Training Data 54
5.4 Performance 58
5.4.1 Computational Performance 58
5.4.2 Qualitative Evaluation 63
5.5 Summary 65

6 Neural Network-Based Simulation **66**
6.1 Implementation Overview 67
6.2 Alexi: Version 1 69
6.2.1 Network Architecture 69
6.2.2 Data Preprocessing 70
6.2.3 Training 72
6.2.4 Results 72
6.3 Alexi: Version 2 76

6.3.1 Network Architecture 76
6.3.2 Data Preprocessing 78
6.3.3 Training 78
6.3.4 Results 79
6.4 Computational Performance 88
6.5 Discussion 89
6.6 Summary 90

7 Conclusions **92**
7.1 Future Work 93
7.1.1 New Approaches to Leveling 93
7.1.2 Enhancements to Neural Network-Based Simulation 96

Bibliography **98**

List of tables

4.1 Main hardware for prototype leveling robot 37

5.1 Variables recorded by Chrono simulation 55
5.2 Terrain parameters for performance evaluation 60

6.1 MSE and R^2 of each network on testing set (Alexi V1) 73
6.2 Spearman correlation coefficients 76
6.3 MSE and R^2 of each network on testing sets 82

List of figures

4.1 32x32 terrain before and after flattening by 8 agents 30
4.2 Comparison of various constant values for ρ_{pu} and ρ_{de}. 31
4.3 Comparison of various k values in pick-up and deposit probabilities. . . 31
4.4 Comparison of average units moved between the probabilistic algorithm and each variant of the memetic algorithm. 35
4.5 Comparison of average cycles between the probabilistic algorithm and each variant of the memetic algorithm. 35
4.6 Dagu Rover 5 with mounted hardware. 38
4.7 Pololu Zumo 32U4 40

5.1 Block diagram of Sandbox's architecture 46
5.2 Profile view of a particle terrain (left), and wireframe view of a SCM mesh terrain (right) 50
5.3 Example experiment configuration file 51
5.4 Example URDF file 53
5.5 Height-map used in performance evaluation (left) and the same height-map rendered in 3D (right) 59
5.6 Mean per-frame execution time using particle and SCM terrain models 60
5.7 Memory utilization of particle and SCM terrain models 61
5.8 Mean per-frame execution time on particle terrain in single-threaded and multi-threaded mode 61
5.9 Memory utilization of particle terrain in single-threaded and multi-threaded mode 62

6.1 Activation function used in Alexi V1 70
6.2 Comparison of real and predicted values on testing set for v (Alexi V1) 74
6.3 Comparison of real and predicted values on testing set for $\Delta\theta$ (Alexi V1) 74

6.4 Scatter plot of real and predicted values on testing set for v (Alexi V1) 75
6.5 Scatter plot of real and predicted values on testing set for $\Delta\theta$ (Alexi V1) 75
6.6 Terrain A (left) and Terrain B (right) 79
6.7 Comparison of real and predicted values on testing set A for Δx (Alexi V2) 80
6.8 Comparison of real and predicted values on testing set A for Δy (Alexi V2) 81
6.9 Linear regression plot of real and predicted values on testing set A for Δx (Alexi V2) 81
6.10 Linear regression plot of real and predicted values on testing set A for Δy (Alexi V2) 82
6.11 Comparison of real and predicted values on testing set A for v 83
6.12 Comparison of real and predicted values on testing set A for $\Delta\theta$ 83
6.13 Linear regression plot of real and predicted values on testing set A for v 84
6.14 Linear regression plot of real and predicted values on testing set A for $\Delta\theta$ 84
6.15 Comparison of real and predicted values on testing set B for v 85
6.16 Comparison of real and predicted values on testing set B for $\Delta\theta$ 85
6.17 Linear regression plot of real and predicted values on testing set B for v 86
6.18 Linear regression plot of real and predicted values on testing set B for $\Delta\theta$ 86
6.19 Screenshot of Alexi V2 demonstrating deformable terrain 87

List of Algorithms

1 Probabilistic leveling algorithm requiring actuated component 27
2 Algorithm governing the *move_agent*() function 28
3 Memetic algorithm, first variant 34
4 Memetic algorithm, second variant 34
5 Control algorithm used for data generation 56
6 Hill-sensing algorithm relying on magnetometer 95

List of abbreviations

ANN	Artificial Neural Network
CPR	Cycles Per Revolution
CPU	Central Processing Unit
FIFI	First-In First-Out
HAT	Hardware Attached on Top
IMU	Inertial Measurement Unit
RAM	Random Access Memory
RNN	Recurrent Neural Network
ROS	Robot Operating System
SGD	Stochastic Gradient Descent
URDF	Unified Robot Description Format
USART	Universal Synchronous/Asynchronous Receiver/Transmitter
XML	eXtensible Markup Language

Chapter 1

Introduction

Autonomous mobile robots which operate over uneven terrain are under active research for a variety of applications such as space exploration [27], construction, and search and rescue [43]. Due to the complex and often remote nature of these applications, it is advantageous to develop these systems using the aid of simulation. In the case of construction, it is often necessary that the robots in use are able to modify the terrain itself either actively or as a side-effect of their motion. Many software packages exist in both the open-source and commercial space for robotics simulation, however most do not support the simulation of outdoor environments and fewer offer a method to simulate deformable terrain. Those that do are either prohibitively expensive in the commercial space, or otherwise use approximations in the simulation which make them less ideal for evaluation of algorithms.

This thesis is concerned with simulating an environment in order to prototype a terrain leveling robot, as a precursor to a construction task. It is envisaged that such a system might eventually be designed using several robots in a swarm configuration. Such a system could also potentially be used to aid reparation efforts in outdoor areas affected by disaster (natural or otherwise). In the case of a natural event such

as flooding as the result of a hurricane, it is typical in rural areas to see roads and bridges quickly eroded away. As an example, in the case of Hurricane Igor which impacted the province of Newfoundland and Labrador in 2010, approximately 150 communities became isolated as a result of damaged roadway infrastructure [45]. A multi-robot system could be used in such a future situation to aid in the construction of temporary bypasses in the absence of human workers, and more specifically to level damaged terrain such that it is traversable by humans as well as other machines. This would mitigate risk to both sanctioned human responders as well as civilians who may attempt to carry out work themselves. Additionally, the concept has been briefly explored in relation to seabed preparation for underwater construction [19]. For underwater construction, current methods typically involve the use of large machinery or other tools which require multiple personnel to operate. Leveling is often carried out using a tool called a screed across the seabed, or by dredging [29]. Such a tool is usually suspended from one or more surface vessels, and care must be taken that the tool remains level at the seabed. In conditions creating high-wave activity, this may often prove difficult. Other tools have been proposed, such as the SILT Wing Excavator by Francingues et al [28]. The device is described by the authors as one which requires less manpower for operation, improving mobilization. However it still requires at least one surface vessel to operate, and is deployed via crane from a barge. The use of a multi-robot system instead allows the required personnel for operation to be reduced much further since the system operates with a higher degree of autonomy.

Prototyping a multi-robot system such as this hardware-first would not be ideal, as multiple robots must be built and debugged and a model environment must be created for them to operate in. It is also unclear what sort of control strategy might be optimal for such a task. Several algorithms must be developed and tested, each of which may have varying hardware requirements. Using simulation allows one to

focus primarily on algorithm development before focusing on hardware development and incurring the associated cost. Simulation also allows one to rapidly test different hardware platforms. For example, the current hypothesis is that the leveling task may be carried out using only the motion of the robots themselves (see Section 7.1.1 for further discussion). If this is not the case, an approach using some sort of actuated manipulator will need to be tested.

There is a plethora of existing work in robot simulation software [18], all of which generally argues that simulation-first development alleviates many of the common issues that arise in robot development and aides development productivity. Arguably, the most prominent of these issues tends to be expense and difficulty. Many of the simulators currently in use for robotics offer some level of support for physics simulation, usually made possible by way of open-source physics engines such as Bullet [2] or ODE [7]. These simulators are also typically coupled with 3D rendering engines for visualization, such as OGRE [6] or Irrlicht [3]. These engines offer a relatively high level of physical realism, while being fast enough to run simulations in real-time. Simulators based on these engines may support the simulation of *static* terrain, modeled as a convex mesh. Simulation tools in the closed-source and/or commercial space such as Vortex [11] provide earthmoving simulation (typically for the purpose of equipment training simulation). In this environment the simulated volumes employ a hybrid modeling approach, which combines models based on discrete particle simulation as well as mesh-based soil simulation, in order to provide a realistic approximation which maintains interactivity [34].

Rather than rely on approximations, a simulator designed for robotics prototyping should aim to be as accurate as possible in order to ensure that the resulting control algorithms translate to reality. For the simulation of a terrain leveling system such as the one described here, the simulator is required to support the interaction of a

potentially large volume of granular material, which is manipulated by one or more robots. Accuracy is important not only to the simulation of the robot movement about the environment, but also to the interaction of the robot with the environment during the leveling task. The simulation of granular material using particle-based methods such as one discussed by Bell et al. typically come with very high compute times per-frame, taking the simulation out of the real-time domain [14]. In their 300-frame simulation of a bulldozer through a granular volume, they report a compute time of 17.4 minutes per frame. To be considered real-time the frame-rate of the simulation should be *interactive*, which conventionally refers to a frame-rate of 15 frames-per-second or higher (approximately 66 milliseconds per frame) [12]. In a simulation such as this, true interactivity is not always necessary (as it is in a game, for instance). However, the rate should be high enough to allow an accelerated view of the systems performance versus an offline method.

In order to maintain both accuracy and interactivity of the simulation, this thesis proposes a simulator based on a system of neural networks, rather than a physics engine traditionally found in simulation software. A neural network trained on data generated from a high-fidelity non-real-time simulation allows the simulation to retain a realistic physical model, while providing real-time prediction of the robot's motion. The neural network model proposed allows simulating a vehicle or robot which has a 3D orientation inside a 2D simulation environment. Removal of a true 3D physics solver and 3D graphics removes the majority of the computational burden of typical simulations.

At first glance, it may appear to some that using neural networks to bridge high-fidelity simulation to low-fidelity is an unnecessary step. If computational power is limited, why not start with the low-fidelity simulation in the first place? If the application demands the highest level of accuracy, why not wait for the high-fidelity

simulation to execute?

To the first question, the case against low-fidelity in such a simulation is that often so much of the necessary detail is abstracted away. For the application described in this work, the movement of the robots about the environment and their physical interaction with the terrain is central to their operation. Taking any part of this physical interaction out of the model will negatively affect the utility of the simulation and potentially skew the design of the controller. As the simulation is simplified and high-fidelity components are abstracted away, so too are any components of the robot controller that may deal with them. This phenomenon has been noted at least as far back as 1991 in Rodney Brooks' seminal work [16]. There, Brooks argues that early robotics work carried out in a "blocks world" often gave way to special-purpose solutions that did not generalize to reality. The argument here against low-fidelity simulation follows the same principle.

The second question is admittedly one of patience. There is nothing stopping one from waiting the required time for a simulation to execute; however with limited computational resources the time to completion often reaches upwards of several days. Consider for example that the high-fidelity simulation used to generate training data (see Section 5.3) is reduced in scope from what would be required to allow an agent to fully execute a leveling algorithm over a terrain. For data generation, we require only a small number of frames to be executed. For a full-length run of a leveling algorithm, we expect that the execution time in terms of number of frames would be much higher. Furthermore, relying only on the high-fidelity simulation would require that the terrain be larger in area and use finer particle scale or denser mesh. Assuming optimistically the same run-time of approximately 8.98 seconds per frame as described in Section 5.4.1 (medium-size particle terrain), and predicting optimistically that a full-length simulation could require at least 256000 frames, the total execution time

reaches approximately 27 days. For the development of algorithms for operation on uneven terrain and for terrain modification, it is advantageous to iterate through the development process more quickly.

1.1 Contributions

The the primary contributions of this thesis are derived from simulators discussed in Chapter 5 and Chapter 6. Related work in the literature towards neural network-based simulation required the collection of data from the movement of a real-world robot. Using the high-fidelity simulator, this thesis demonstrates that a usable system can be trained using data collected in simulation, removing the need to develop a hardware platform for experimentation. This thesis also represents the first known step towards using a neural network-based method to predict the deformation of terrain. On the subject of the high-fidelity simulator itself, two methods of simulating a deformable terrain are evaluated in terms of both computational and practical performance. The high-fidelity simulator is novel in that it is the only known platform explicitly designed for the simulation of a robot over a deformable terrain. Chapter 6 provides network training and generalization results for several versions of the simulator. These results are focused primarily on the simulation of robot motion over an uneven terrain, however early results for a network simulated terrain deformation are also provided.

The author has several published or submitted works related to this thesis:

- *A Survey of AUV and Robot Simulators for Multi-Vehicle Operations* [22] [Published, Conference]
- *Terrain Leveling by a Swarm of Simple Agents* [19] [Published, Conference]

- *Towards Real-Time Robot Simulation on Uneven Terrain Using Neural Networks* [21] [Published, Conference]
- *Simulation of a Mobile Robot on Uneven Terrain Using Neural Networks* [20] [Submitted, Journal]

It should be noted however that *A Survey of AUV and Robot Simulators for Multi-Vehicle Operations* was written and published prior to beginning the work contained in this thesis.

1.2 Scope of Work and Thesis Overview

While the motivating application of the simulator described is a terrain leveling system using a swarm of robots, this thesis is focused on the problem of accurately simulating the movement of a single robot as a necessary first step. The simulation of a complete terrain leveling swarm is considered outside of the scope of Chapters 5 and 6, and is left as a future extension of the systems described here. An overview of an early hardware prototype is described in Section 4.3. A discussion of a practical real-world implementation of a terrain leveling robot is purely hypothetical from the perspective of this thesis, however the prototype which was designed provides a basis for the robot modeled in simulation.

Chapter 2 presents a review of previous works in the areas of robotic construction, robot simulation, and neural networks applied to simulation. Chapter 3 gives a brief overview of both artificial neural networks and robot simulation. This overview focuses on feed-forward neural networks as they are exclusively used within this work. Chapter 4 provides a formal definition for the terrain leveling task, and describes two leveling algorithms developed using low-fidelity simulation. These algorithms describe

the first attempts at developing a terrain leveling system. It is shown that while simulation was a highly useful tool in their development, the limitations of the low-fidelity simualation are reflected in the algorithms themselves. Chapter 4 also describes a robot prototype which was built following the development of the previous leveling algorithms. Continued use of simulation was preferred following its construction, due in part to issues in creating an environment for the prototype to operate in. Chapter 5 describes the development of a high-fidelity terrain leveling simulator, which sought to re-create the prototype robot and an environment in which to experiment with leveling algorithms. Several performance characteristics are compared, and practical considerations of the simulators use are explored. Chapter 6 provides a description of a 2D simulator based on neural-networks. Generalization results for several versions of this simulator are given, to demonstrate that the neural networks can be appropriately trained for such a task. Early results are also given for the prediction of terrain deformation using the same neural network concept. Chapter 7 provides final conclusions of the completed work, as well as discussion of possible future extensions to the concepts discussed within the scope of this thesis.

Chapter 2

Literature Review

A selection of work from the areas of swarm construction, robot simulation, and neural network-based simulation are reviewed in Sections 2.1, 2.2, and 2.3. These domains in particular are covered to provide context to the concepts discussed in this thesis. As previously discussed, intended application for the terrain leveling system is to aid in construction or terrain reparation. Works related to robotic construction are reviewed to provide background to this application. Similarly, simulation and neural network-based simulation works are reviewed to provide background to other works which made use of or developed simulators for robotics, as well as explored the use of neural networks for simulation.

2.1 Robotic Construction and Site Preparation

The potential for robotic swarms to be utilized for construction purposes has been recognized for several decades, beginning in the literature with Brooks et al.'s theoretical description of a swarm used in the construction of a lunar base [17]. Their work proposed that such a swarm modeled on the collective behaviour of ants or termites

would allow the construction system to be more robust against a changing remote environment. Their work also describes an algorithm which might be employed by the swarm for leveling terrain in preparation for construction, from which some inspiration is taken for the system described earlier in this chapter.

Theraulaz and Bonabeau later applied stigmergic behaviour to the problem of collective building, in which they simulated the construction of three-dimensional nests [51]. In their work they demonstrated that biologically-inspired behaviour algorithms could be used to guide the construction of relatively complex 3D structures, analogous to construction observed in nature by insects such as wasps. Each agent in the system responded to particular configurations of blocks, by then triggering a 'building behaviour' (the placement of a block). This work represents one of the earliest practical applications of collective robotics to construction.

Parker et al. described a collective robot system for off-Earth site preparation [46]. Their work proposed an algorithm based on a number of high-level and low-level behaviours, selected through a mechanism of *motivational behaviour*. While behaviour-based, the algorithm did not make use of stigmergic behaviour and instead assumed that some map of the environment was available globally to each agent in the system. This work was demonstrated by way of simulation, however they had begun to implement the system on a number of real-world robots. They stated that more extensive experimentation was possible in simulation as opposed to on physical robots and that simulation "can contribute to the variety of situations and robot control designs that can be explored".

Wawerla et al. applied collective construction robots to the task of building a barrier [55]. The robots constructed the barrier from coloured blocks within a 2D planar environment. Experiments involving multiple robots were carried out only in simulation, due to "limitations of hardware availability and reliability". They

did however demonstrate that the system could function in reality in a single-robot experiment. The simulated environment allowed experiments with up to 8 robots, repeated 10 times. Each simulation was limited to run for 4500 seconds. This is in contrast to the single-robot experiment which ran for only 3 trials, each trial running for approximately 30 minutes.

Napp and Nagpal demonstrated a method for building ramp structures over irregular terrain, with the goal of making the terrain traversable [43]. The robots they describe are capable of depositing a formation of amorphous material at a location on the terrain. These deposits may be made using an adaptive algorithm with multiple robots. The algorithm is adaptive (or reactive) in the sense that agents collectively react to the addition of new material, in order to ensure that the resulting structure is navigable according to a set a mathematical constraints.

Werfel et al. implemented another collective construction system for building 3D nest-like structures using a number of blocks [56]. Their system made use of a number of minimalistic robots with limited actuation and sensing abilities. The agents placed bricks by reacting to the current configuration, however their movement is restricted to a set of pre-compiled rules called a "structpath". The structpath is generated from a 3D representation of the desired structure, and is available to each agent in the system. Such a system used for building structures provides the inspiration for the terrain leveling algorithm described in this thesis, where the terrain leveling operation is carried out prior to construction using another robotic swarm.

Ardiny et al. provide a further review of construction using autonomous mobile robots [13]. Their review examines the field of autonomous construction across the axes of applications, materials, and robotic systems, and explores the challenges related to each.

2.2 General-Purpose Robot Simulation

Recent years have seen the emergence of computing hardware capable of high-fidelity graphics and physical simulation for games and multimedia. The ubiquity of such hardware and software has also led naturally to the utilization of the same technologies for simulation of many kinds, including that of mobile robots and their sensors and acutators.

One of the earlier and arguably most notable of the simulators to arise from this development is the Stage simulator, in development since 2001 as part of the Player Project (formerly Player/Stage) [54]. The Stage simulator provides simulation fidelity suitable mainly for indoor environments, as it does not allow movement in the "up" direction and is described as a 2.5D simulator. It also does not offer any dynamic physics simulation. Stage has, however, been reported to scale linearly with the number of robots present in an environment, and has been suggested as a candidate for swarm robotics simulation. Gazebo, a simulator which is developed in cooperation with the Player Project, aims to provide 3D dynamics and is intended to allow for higher-fidelity outdoor simulation [38]. Despite being compatible with the Player project, the authors state that Gazebo is not intended as a replacement for Stage, as the inclusion of 3D dynamics greately reduces the number of robots which may be simulated concurrently.

As part of the Swarmanoid project [9], the ARGoS simulator was developed to allow the simulation of large numbers of mobile robots in a dynamic 3D environment [47]. ARGoS differs from its contemporaries in that it supports simulation backed by multiple physics engines running simultaneously. When simulating several types of robots in a single environment for example, one may use different physics engines for each type of robot so that the most appropriate or efficient engine is used to model

the agent in question. This division of labour within the simulator comes with the caveat that agents modeled using differing physics engines may not interact with each other; they may however still communicate as the physics layer is not responsible for those mechanisms.

Coppelia Robotics develops the Virtual Robot Experimentation Platform (V-REP) as a commercial product (offering a free license for eductaional purposes) [50]. V-REP's greatest strengths lie in its well-developed user interface as well as embedded scripting language. Robot models may be developed with embedded Lua scripts, allowing models to be ported between users of V-REP without requiring additional plugins or installations. It allows one to create a multi-robot simulation nearly trivially, however like most other 3D simulators it may suffer from performance degredation as the number of agents increases. These performance issues may be more pronounced if one is also using built-in sensors such as vision sensors which are computationally demanding to simulate.

CM Labs develops the commercial software Vortex Studio [11]. Vortex is marketed as being suitable for equipment and operator training, as well as for prototyping mechatronic products. Vortex Studio offers an option for earth moving simulation, which provides an environment for simulated construction equipment such as bulldozers and excavators to interact with the terrain. The method used to simulate interaction with the terrain employes a hybrid approach of mesh deformation and particle simulation, in an effort to maintain interactivity while being relatively realistic. The marketing material places an emphasis on pile formation and pouring in the simulation (being necessary for simulating excavators), while Holz's work underpinning the simulation [33] describes experiments related to particle adhesion for pile forming. While this is useful for a terrain leveling simulation, we believe a more accurate and generalized method of simulation is required. Being commercial software,

it is also much more expensive than other available options for simulation software, imposing a potential barrier to access.

NetLogo is a programming language and modeling environment commonly used in swarm robotics research, though it is not designed specifically for this purpose [52]. It is described by Tisue and Wilensky as an environment for simulating "natural and social phenomena". It is intended to be simple to use by students and researchers who may not have a strong background in programming, and it differs from some other simulators in that it offers no physics simulation whatsoever. It instead focuses on algorithmic simulation, allowing for adjustable parameters through a user-defined GUI and a 2D visualisation for output.

Platforms intended for game development have also seen use in simulation for robotics. The Unity game engine has been used to develop a mobile robot simulator [32], as well as a robot operator training game and simulator [23]. While a platform such as this is typically not intended to be used for simulation, there is very little separating a game engine from a simulator using a physics engine such as ODE [7]. In each case the underlying rendering and physics systems are typically designed to acheive real-time performance and may sacrifice accuracy for this purpose.

Several surveys exist in the literature which can provide further review of available robot simulators. A survey by Castillo-Pizarro et al. provides a more in-depth study of some simulators described here, as well as several others [18]. A previous survey by the author surveyed simulators that may be suitable for underwater robotics, but many of the simulators discussed are also suitable for land-based robotics [22].

2.3 Neural Network-Based Simulation

Artificial neural networks have been in use in a variety of capacities for several decades, and have recently seen a resurgence owing to increases in computing power, improved methods of training and the adoption of "big data". So-called "deep learning" methods such as recurrent or convolutional neural networks have seen renewed use for tasks such as speech recognition and computer vision. Neural network-based learning methods have been shown to offer high capacity for generalization (accuracy of prediction given unknown input) and noise tolerance (ability to generalize when trained with noisy input). While neural networks typically require significant computing time in order to train (typically in proportion to the size of the network and number of training examples), trained networks used in production generally execute predictions in a fraction of that time. It is primarily for this reason that neural networks have been explored for use as emulators to replace more computationally expensive portions of software.

The earliest known use of neural networks for this purpose is by Grzeszczuk et al., in which neural networks were used in a system called *NeuroAnimator* to replace numerical calculations needed for physics-based animation [31]. Typically in a physics-based platform the motion defined for an animation would be computed based on a number of controller inputs and forces. These inputs are integrated over a number of relatively small time-steps using a numeric solver for a system of equations or constraints. For more complex systems, it is not hard to imagine that the required computational time balloons rather quickly. Grzeszczuk et al. proposed that a neural network could be trained to emulate the physical model with enough accuracy such that it could replace the physics layer in the animation application. The system they

describe has several attributes which carry over to the work described here. In particular, their system used single-layer feed-forward neural networks to predict changes to state variables in each time-step rather than predicting the final value. This significantly reduces the range of output variables, and when training data is properly normalized allows the network to be trained using much smaller hidden layer dimensions. They additionally used a heirarchy of smaller networks rather than one large network to predict the state variables. They found that large monolithic networks were harder to train, and that using a number of smaller networks allowed state variables to be separated according to their dependencies. Their system was trained by using a set of samples generated by recording pairs of input and output vectors from the physics-based numeric simulator. In particular, rather then generating training pairs uniformly for all possible input/output values, they found it suitable to instead sample only pairs which would typically be seen in practice. This allowed faster generation, and the neural network was still able to generalise without apparent loss of fidelity to these other "unseen" examples when used in production.

Appearing much later, Pretorius et al. in a successive series of papers proposed a system employing neural networks to replace physics-based simulation of robots, which they refer to as Simulator Neural Networks (SNNs) [49] [48]. Their system was designed for an Evolutionary Robotics (ER) application, in which robot controllers must be evolved and evaluated relatively rapidly as part of a genetic algorithm. Rather than relying on real-world evaluation of candidate controllers, ER researchers often rely on evaluation in simulation. Pretorius et al.'s work proposed that a simulator using a neural network to emulate physical simulation would greatly increase the performance for these ER simulators. The network architecture and training methodology they use follows similarly from Grzeszczuk et al. Training data is generated by providing random input to a prototype robot and the corresponding output is

recorded. However rather than relying on simulation to generate training data, they instead gathered data by tracking a real-world robot using an overhead camera. The networks employed as SNNs were single-layer feed-forward networks, and they (like Grzeszczuk et al.) found greater results in using several smaller networks to compose the underlying system. Their earlier explorations produced encouraging results, and more recently they demonstrated the SNN system for use in evolving controllers for a snake-like robot [57].

De at al. presented "Physics-driven Neural Networks-based simulation system" (PhyNNeSS), a neural network-based platform for simulating deformable models for medial applications [25]. Their system was devised to drive simulations which must support haptic feedback, which they state must run at a much higher update rate (1000Hz) than typical visual-only simulations which are typically rendered at 30Hz. Their work deviates from the previous literature in that they use Radial Basis Function networks (RBFN) rather than vanilla feed-forward networks, however data generation and network training proceed along similar lines. Their work demonstrated that such a neural network-based approach can be made suitable for systems much more complex than simple rigid bodies.

2.4 Summary

This chapter has presented a review of a selection of work from three areas; robotic construction, robot simulation, and neural network-based simulation. The reviewed works provide foundation and inspiration for the work described here. The works related to robotic construction in Section 2.1 provide a background to the larger problem we wish to solve in terrain leveling. Section 2.2 provides a review of existing 'traditional' simulation solutions, some of which we have attempted to use in past

experiments with mixed results. Finally, several works specific to neural network-based simulation were reviewed in Section 2.3 to provide an assesment of similar attempts towards utilizing neural networks to enhance simulation.

Chapter 3

Background

This chapter provides a condensed overview of concepts related to both neural networks, as well as the simulation of robots. Section 3.1 provides an overview of the basic back-propigation algorithm used during network training, and briefly describes some more advanced concepts used to aid generalization and/or prevent over-fitting of the network. Section 3.2 provides a short summary of the various components making up a typical robotic simulation.

3.1 Artificial Neural Networks

An Artificial Neural Network (ANN) is an approach for computation modeled on the interconnection of biological neurons in the brain. Artificial 'neurons' in the network are represented as a node with one or many input and output connections to other nodes. Each input to a neuron is given a weight which may strengthen or weaken the contribution of an input value. Each weighted input value is typically summed together and passed through an activation function $\sigma(x)$. The activation function is often a continuous differentiable 'squashing' function such as the logistic sigmoid or hyperbolic tangent. The selected activation function typically depends on

the particular problem the network is applied to, as well as the desired range of the function.

In a feed-forward network architecture (among others), neurons are organized into several layers. These include an input and output layer as well as one or more hidden layers, where each layer l receives input only from layer $l-1$. Mathematically, the activation value of a neuron j in layer l is given by Equation 3.1. w^l_{jk} denotes the weight value assigned to the connection from the kth neuron in the $(l-1)$th layer to the jth neuron in the lth layer.

$$a^l_j = \sigma(\sum_{k=1} w^l_{jk} a^{l-1}_k) \tag{3.1}$$

This equation may be re-written in a vectorized format as

$$a^l = \sigma(w^l a^{l-1}) \tag{3.2}$$

Where a^l now represents a vector of activation values at layer l, w a matrix of weights at layer l, and activation function σ is applied element-wise. The elements of w^l are weight values w^l_{jk} such that the value is placed at the jth row and the kth column of w.

At the input layer $l = 0$, a vector x may be presented to the network where each element maps to each input node in the layer. The values at each input node are propagated forward through the network, being weighted and summed by nodes in each subsequent layer until being emitted at the output layer. The weighting of input from preceding nodes is analogous to the reinforcement of some neural pathways over others in the brain, and allows the network to respond appropriately to a variety of input. For this process to work effectively however, the weight values of the network must be trained. Training is typically carried out via gradient descent by way of

the back-propagation algorithm. In this process, a number of possible input and corresponding output vectors are provided to the network as training examples. The input values are passed through the network, and the result is compared with the known output. The error computed by a loss function at the output layer may then be propagated backwards through the network, adjusting the weight values at each node correspondingly. The training examples are fed into the network repeatedly for a number of *epochs* and the weights adjusted such that the total error at the output layer is reduced on each iteration.

The loss function used for neural networks in this work is the mean squared error (MSE), which is given by

$$C = \frac{1}{2}\sum_{j}(y - a^{x,L})^2 \tag{3.3}$$

Where y_j is the output of the output layer, and $a^{x,L}$ is the activation value of the Lth layer, where L is the index of the final (output) layer in the network for a given training example x.

Using this loss function, the error at the output layer L for each training example x is first computed by

$$\delta^{x,L} = \nabla_a C \odot \sigma^{'}(z^{x,L}) \tag{3.4}$$

Where $\nabla_a C$ is the gradient of C with respect to $a^{x,L}$, $\sigma^{'}$ is the derivative of activation function σ, $z^{x,L}$ is the weighted activation value $z^{x,L} = w^L a^{x,L-1}$, and $\odot$ is the Hadamard product (element-wise product).

The error at each previous layer is then computed by

$$\delta^{x,l} = ((w^{l+1})^T \delta^{x,l+1}) \odot \sigma^{'}(z^{x,l}) \tag{3.5}$$

for each layer from $l = L - 1$ to $l = 0$. Once the error has been back-propagated through the network, weights for each layer may be updated by

$$w^l \leftarrow w^l - \frac{\eta}{m} \sum_x \delta^{x,l} (a^{x,l-1})^T \tag{3.6}$$

where η is the learning rate and m is the number of training examples.

L2 regularization is used in this work to help alleviate over-fitting, in which the neural network is trained such that it may only predict the training data it was shown correctly (*i.e.* unable to generalize to unseen data). The L2 regularization term is added to the loss function such that

$$C = \frac{1}{2} \sum_j (y_j - a_j^L)^2 + \frac{\lambda}{2n} \sum_w w^2 \tag{3.7}$$

where λ is the regularization parameter, n is the number of training examples, and $\sum_w w^2$ provides the sum of the squares of all weights in the network (*i.e.* summed for each weight matrix at each layer l).

Batch normalization is also used in this work to aid training. Training examples are typically fed into the network in batches (thought of as a matrix where each column represents a training example). Batch normalization adjusts the vectors in the batch such that they have a mean of zero and are of unit variance.

Several sets of data are typically used in the training and evaluation of a neural network. These are the training, validation, and testing sets. The training set is simply the set of input-output pairs fed into the network during the training process described above. The validation and testing sets are other data which are not shown to the network during training. The validation set is used during training to assess the performance of the network in its current state (often at the end of an epoch), by testing the network with the inputs of the validation set and comparing the network

output for each example with the real output found in the set. The testing set is used in a similar manner using the final trained network. Unseen data is used in both cases to provide a measure of how well the network generalizes to unseen data. A network with high training accuracy but low performance in validation and/or testing is said to suffer from overfitting. Guiding training using metrics obtained on the validation set can aide in improving generalization on the final testing set.

3.2 Simulation of Robots and Vehicles

Computer simulation appears in many forms across many different fields, generally referring to some method by which a real-world system or phenomenon may be modeled using a computer system. These may be simulations which execute without user interaction or visual output (referred to here as 'offline' simulation), or those which provide visualization or allow the user to interact with the simulated system in real-time. It is possible to make use of either approach in the field of robotics, however it is often advantageous to provide some real-time visual feedback to the user. This allows one to observe the system in an environment that mimics the real-world environment it is to be used in.

Real-time simulations using 3D visuals are not unlike modern games, which themselves may (to a degree) be considered simulation. The main components of such simulators are a *physics engine* and a *graphics engine*. As their names imply, the physics engine is responsible for driving the physical interactions of objects in the scene, such as motion and collision, and maintaining the physical state of the objects. The graphics engine drives the visual representation of the objects, and depending on the engine used, may allow the developer to implement high-fidelity visual effects such as those found in modern games. Craighead et al. assert that a simulator with

a high-level of visual (or physical) fidelity should use high-resolution textures and models with a high polygon count [24]. In modern graphics engines however, these textures and models are typically loaded as external assets and have very little to do with the engine or software. It is up to the end-user of the engine to provide assets of sufficient quality. It is then also the users' responsibility to ensure that the system that will run the simulation is of sufficient power to execute the software in real-time using high-resolution assets. For the simulation to appear real-time, the graphics engine needs to update the screen at a rate of 15Hz or higher. This definition of 'real-time' follows roughly the same meaning as found in the area of real-time rendering, where we are interested in the minimum frame-rate needed for interactivity [12]. Since the visuals are driven by the underlying physics engine, the physics must also be stepped at least as many times a second.

3.3 Summary

This chapter has provided a preliminary introduction to the topics of artificial neural networks, as well as robot simulation. Section 3.1 provided a description of the mathematics and design of a feed-forward neural network, as well as the backpropigation algorithm. Section 3.2 provided an introduction to robot and vehicle simulation, and described the common components which they are composed of. The concepts of offline and high-fidelity simulation were introducted. Finally, a formal definition of real-time simulation was provided.

Chapter 4

Terrain Leveling

Prior to discussing terrain leveling simulation in Chapters 5 and 6, it is helpful to explore the previous work which was carried out in terrain leveling system without the aid of more advanced simulators. Section 4.1 first provides a formal definition of the leveling task. The definition provided builds on on a previous description of the site preparation task given in the literature. Section 4.2 provides details of two terrain leveling algorithms which were designed and tested using low-fidelity simulation. Results for each are given, where in each case the leveling algorithm appears to be viable. Despite this, it is noted that the experiments were carried out using low-fidelity simulation, and that the algorithms make several assumptions which may not be satisfied in real-world deployments. Section 4.3 describes a hardware prototype of a leveling robot which was designed, which was later modeled in simulation.

4.1 Defining the Leveling Task

The terrain leveling task is a particular subtask of the more abstract collective site preparation task proposed by Huntsberger et al. [35]. Their work does not provide a formal specification of the task, however they describe the task as involving the

removal of rocks or other debris by a swarm of bulldozer-like robots. Here, terrain leveling as a task is given a more strict definition within the context of site preparation.

The following assumptions define the environmental constraints of terrain leveling task envisioned for this work:

1. The site acted upon is a 3D volume of granular material
2. No granule is too large to be moved by a single robot
3. The volume of material in the site remains constant
4. The terrain remains traversable

Assumption (1) dictates that the material acted upon by agents in the swarm is granular in nature, such as sand or gravel. Assumption (2) precludes configurations such as those described in [35] in which large rocks may have to be moved by multiple agents in cooperation. Assumption (3) dictates that material may not be brought in from outside the site, nor may material be removed from the site. The leveling task therefore involves re-configuring the existing granular medium such that it becomes flat. Finally, assumption (4) stipulates that modifications are not made to the terrain which would prevent the a robot from traversing it (*e.g.* there should be no large differences in height from one point of the terrain to the next). In reality these assumptions, such as (2), would need to be relaxed in order for a leveling agent to operate in a typical environment, but they have proven useful for the bounding the scope of our investigations.

It is also necessary to provide some specific definition of 'flatness' that defines the goal state of the task. Given a 2D function $f(x, y) \in \mathbb{R}$ where $x, y \in \mathbb{R}$ such that $f(x, y)$ defines the height of the terrain at location (x, y), a terrain which is sufficiently flat is one in which $\left|\nabla f(x, y)\right| \leq \epsilon$ for all values (x, y). Here ϵ defines a threshold,

where smaller values imply a more uniform terrain surface. This value is considered to be application specific.

4.2 Terrain Leveling Algorithms

4.2.1 Probabilistic Algorithm

Previous work by the author explored an algorithm using agents capable of explicitly manipulating the environment [19]. An algorithm was developed for leveling terrain which targeted underwater vehicles. It was assumed that each agent had the ability to pick up material from the sea floor, and deposit it in another location. Each agent could hold only a single unit of material, and the model was simplified such that material was always picked up from or deposited directly below the agent.

```
Loop
    for each agent ∈ environment do
        if agent sensing then
            if rand() < ρ_sense then
                h ← sensed_height
                enqueue_memory(h)
                α ← update_average()
            end
            state ← acting
        else if agent acting then
            if agent not carrying material and rand() < ρ_pu and h > α then
                pick up material
            end
            if agent carrying material and rand() < ρ_de and h < α then
                deposit material
            end
            move_agent()
            state ← sensing
Forever
```

Algorithm 1: Probabilistic leveling algorithm requiring actuated component

```
if rand() < 0.75 then
    move_random()  // Only if an agent is not occupying new position
else
    move_forward()  // Only if an agent is not occupying new position
```

Algorithm 2: Algorithm governing the *move_agent*() function

The model used in the development of this algorithm differs from the continuous environment discussed in Section 4.1. Instead a discrete grid is employed which the agents are able to move about. Agents may move in any direction on the grid to an adjacent cell, but are prevented from leaving the grid or moving into a cell occupied by another agent. The agents are prevented from moving in to occupied space by an outside oracle—there is no real communication between agents. As agents move about the terrain, they are always considered to be just above the seafloor. As such, an agent moving to an adjacent cell would complete the action regardless of the difference in height of the cells. Agents act in two stages, *sensing* and *acting*. In the sensing stage, agents may sample the height of the cell they occupy. This is followed by the acting stage, where an agent may execute one or more actions. The height sampled in the sensing stage is stored in a FIFO queue which may hold a maximum of eight heights. The acting stage allows the agent to move to an adjacent cell, and either pickup or deposit material at that location.

As each agent moves about the environment, it is given ρ_{sense} probability of storing a sampled height value from the current cell on the read step (*i.e.* upon arriving in a new cell, the agent has a ρ_{sense} chance to sample the height). By sampling heights sparsely about the grid, each agent is able to estimate the overall average height α of the terrain. Each agent is instructed to pickup material above this computed average, and deposit only at heights below the average. Each of these pick-up and deposit operations are carried out with probability ρ_{pu} and ρ_{de} respectively. Using

probabilistic decision-making for each of these actions offers the agents some freedom against incomplete information. Each agent has only an estimate of the average height—making a pick-up or deposit decision based strictly on this value causes the agents to act on a value that contains inherant error. Probabilities provide extra freedom of movement to counteract this. Additionally, the agents are meant to loosely mimic swarms found in reality. In the absense of exact information with respect to *why* an organism may make a particular decision, probabilities allow us to create an apporiximate model for our algorithm.

The algorithm was tested with two variations in the selection of probabilities ρ_{pu}, ρ_{de}, and ρ_{sense}. The first selected various constant values in the range (0.0, 1.0) for each variable, while the second allowed the probability to be computed by each agent based on the height of its current cell. This value was computed on a curve defined by the functions $\rho_{pu} = (\frac{height}{k+height})^2$ and $\rho_{de} = (\frac{k}{k+height})^2$, as described in work by Deneubourg et al. [26] and Vardy et al. [53] The value of ρ_{sense} is chosen experimentally; generally speaking it should be low enough that cells are sampled sparsely across the grid to reduce the possibilty that an agents sensed height α is derived from a singular location.

Experimentation with this algorithm was carried out in a low-fidelity voxel-based simulator. In this environment each cell of the terrain is represented by a stack of voxels shaded by height. Agents are each represented by an icosahedron. For the purpose of generating results over repeated executions of the algorithm, the simulation could be run in a headless mode which delivered output to a command-line interface rather than the 3D voxel view. Experiments with each probability function were conducted over a range of probabilities for k-values in an effort to determine which performed the best. For each value, the simulation was run 30 times (ending when the terrain reached a flat state), and the mean number of cycles to completion was recorded for the value.

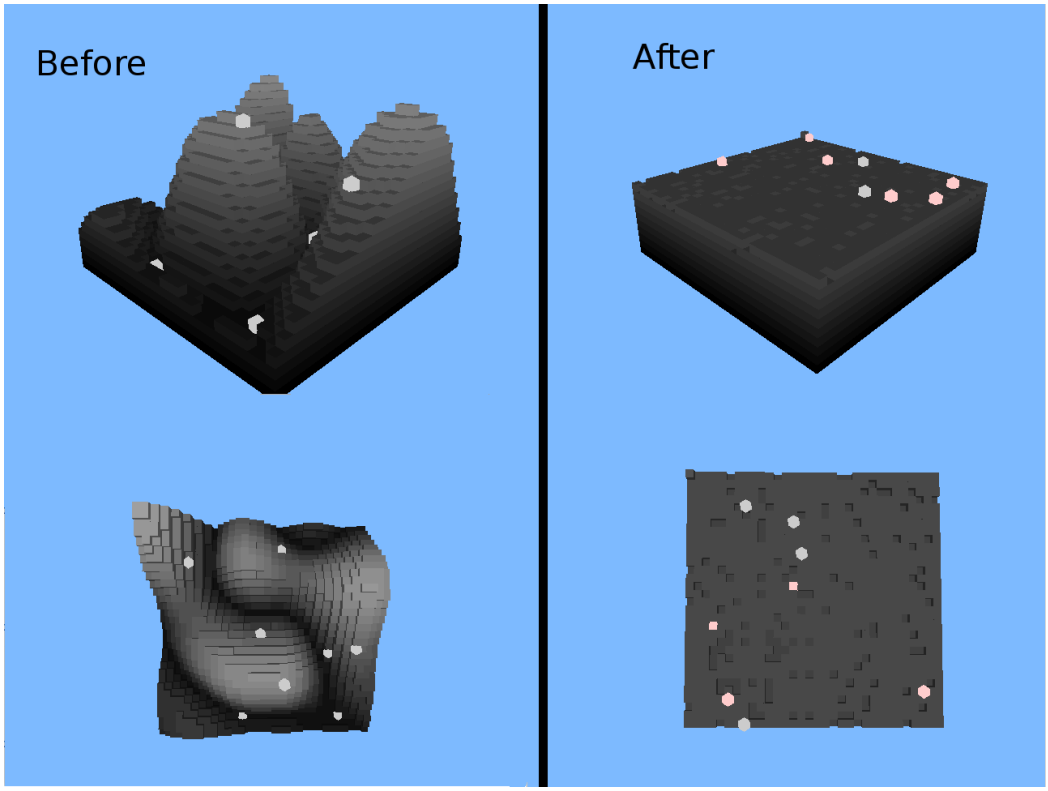

Figure 4.1: 32x32 terrain before and after flattening by 8 agents

4.2.2 Results: Probabilistic Algorithm

The performance of the algorithm was evaluated by the average time to completion, where time is measured as the number of cycles of the algorithm's outer-most loop as seen in Algorithm 1. Averages were taken over 30 runs for each probability or k-value shown in Figure 4.2 and Figure 4.3, where each execution was halted when the level condition of the terrain was reached.

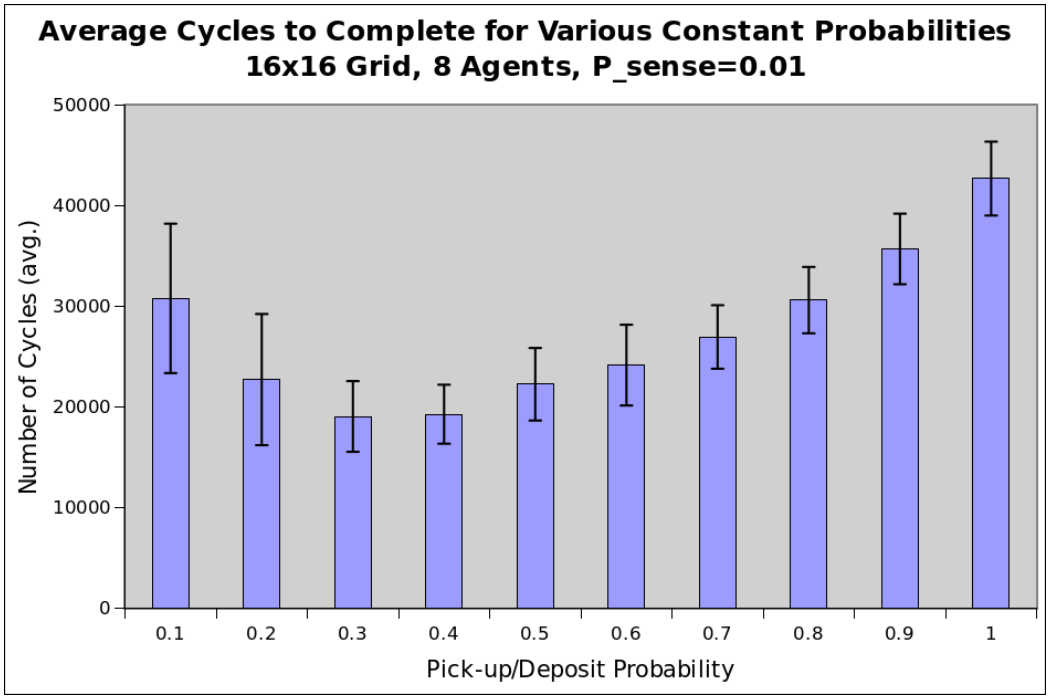

Figure 4.2: Comparison of various constant values for ρ_{pu} and ρ_{de}.

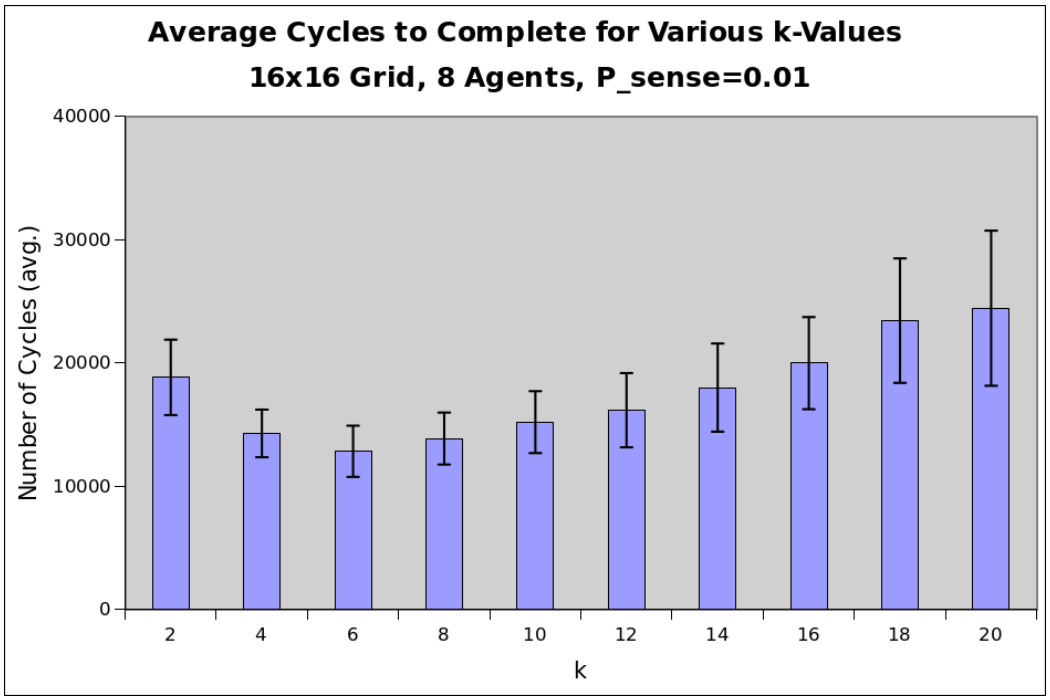

Figure 4.3: Comparison of various k values in pick-up and deposit probabilities.

A U-shaped curve can be seen in time to completion in both Figure 4.2 and Figure 4.3, with minimums at 0.3 and $k = 6$ respectively. This probability for ρ_{pu} effectively controls how far (or how long) an agent may explore after encountering a cell of height greater than α. Similarly, the probability for ρ_{de} controls how far an agent may move before depositing once the condition that its height is less than α is met. It was conjectured that, assuming it is most efficient to pick up material from peaks and deposit in valleys, that the optimal probability value may relate the the average radius of hills and valleys found on a terrain. By selecting a value appropriate for a given terrain, the probability may in most cases allow the agent to move forward enough that it is positioned closer to the maximum height on the grid before picking up material. The same reasoning holds for the deposit operation.

4.2.3 Memetic Algorithm

Two variations of the algorithm described in Section 4.2.1 were later developed which incorporated evolutionary algorithms. They are each considered a memetic algorithm as defined by Moscato [42], due to a combination of both a global population-based search as well as a local heuristic search (in this case, the heuristic is the requirement that pick-up and deposit still occur above and below height α).

In each variation, a chromosome is represented by a permutation of 20 (possibly repeating) actions. Each element of the permutation may be 1 of 10 possible actions—8 directions of movement, as well as the pick-up and deposit operation. Actions are indicated by the integers 0-9 inclusive. Not all permutations are valid—restrictions are placed on the ordering of actions. It is enforced that pick-up actions must always come before deposit actions, and there must be an equal number of pick-up and deposit actions in the permutation.

The same fitness function is also used in each variation, where the fitness of a

chromosome in the population is defined by the equation $f = \frac{n_{changes}}{pickups_{total}+deposits_{total}}$, where $n_{changes}$ is the number of *positive* changes made to the terrain. A positive change is defined as one which is likely to make the terrain flatter, the determination of which is part of the logic within the pick-up and deposit actions. A positive pick-up is one which removes material from above the agent's calculated average height α. A positive deposit is one where material is deposited below the agent's calculated average height, and $\left|height_{deposit} - height_{pickup}\right| > 1.0$ where $height_{deposit}$ and $height_{pickup}$ are the heights *after* the deposit and pick-up operations respectively.

Each variation of the algorithm is differentiated by the population model used. In the first variation, each agent in the swarm represented an individual in the evolutionary population. That is, each agent held a single chromosome. In the second variation, each agent held a list of 5 chromosomes. In each case it was assumed that an agent external to the swarm was responsible for managing the population of chromosomes—performing selection, transferring chromosomes between agents, etc.

In the first variant (1 chromosome per agent), each agent is initialized with a random (valid) chromosome. Each agent executes the sequence of actions stored in their chromosome five times. At the end of these five executions, the fitness of each agent is then evaluated. The agent with the highest fitness keeps its current chromosome, while the chromosomes of all other agents are replaced with new randomly generated chromosomes.

In the second variant (5 chromosomes per agent), each agent is initialized with a random pool of valid chromosomes. Each of these permutations are executed in turn, with the fitness of each evaluated at the end of the sequence. For each agent, those chromosomes with fitness below some threshold are replaced with new randomly generated chromosomes.

Input : $c_x \leftarrow$ random, valid chromosome for each agent x
Loop
  for each *agent* $x \in$ *environment* **do**
    for *5 iterations* **do**
      for each *action* $\in c_x$ **do**
        $execute(action)$
      $x_{fitness} \leftarrow f(c_x)$
    end
  $evaluate\_fitness()$ `// External oracle selects the fittest agent`

  for each *agent* $x \in$ *environment excluding the fittest* **do**
    $c_x \leftarrow$ new random chromosome
Forever

Algorithm 3: Memetic algorithm, first variant

Input : $\mathbb{C}_x \leftarrow$ a set of 5 random, valid chromosomes $c \in \mathbb{C}_x$ for each agent x
Loop
  for each *agent* $\in$ *environment* **do**
    for each $c \in \mathbb{C}_x$ **do**
      for each *action* $\in c$ **do**
        $execute(action)$
      $c_{fitness} \leftarrow f(c)$
    for each $c \in \mathbb{C}_x$ **do**
      if $c_{fitness} < 1.0$ **then**
        $c \leftarrow$ new random chromosome
Forever

Algorithm 4: Memetic algorithm, second variant

In either approach, while each action in a permutation is attempted during execution, these actions are allowed to fail silently. That is, for example, if an agent attempts to pick up material and conditions are not satisfied, the agent will do nothing until its next action is called. Similarly, if an agent attempts to move but is prohibited by the boundary or the presence of another agent, it will remain in place. This exploits domain knowledge within the evolution—we already know that for flattening we must remove material from above the calculated average, for example. Therefore, if the agent is not above the average the pickup action will fail. Similar logic follows for depositing below the average. This failure will be reflected in the fitness, since the

failed pickup will not record a positive change in the terrain.

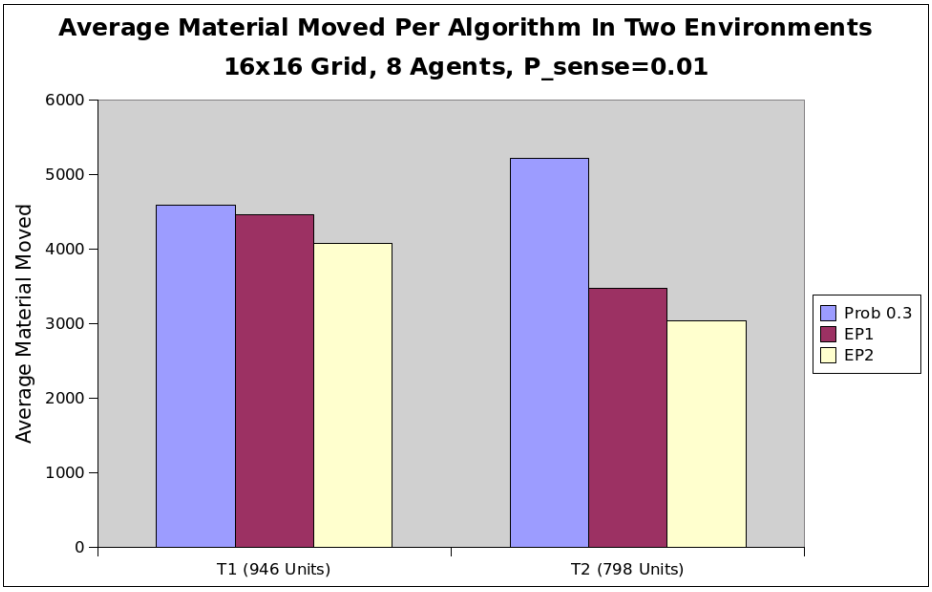

Figure 4.4: Comparison of average units moved between the probabilistic algorithm and each variant of the memetic algorithm.

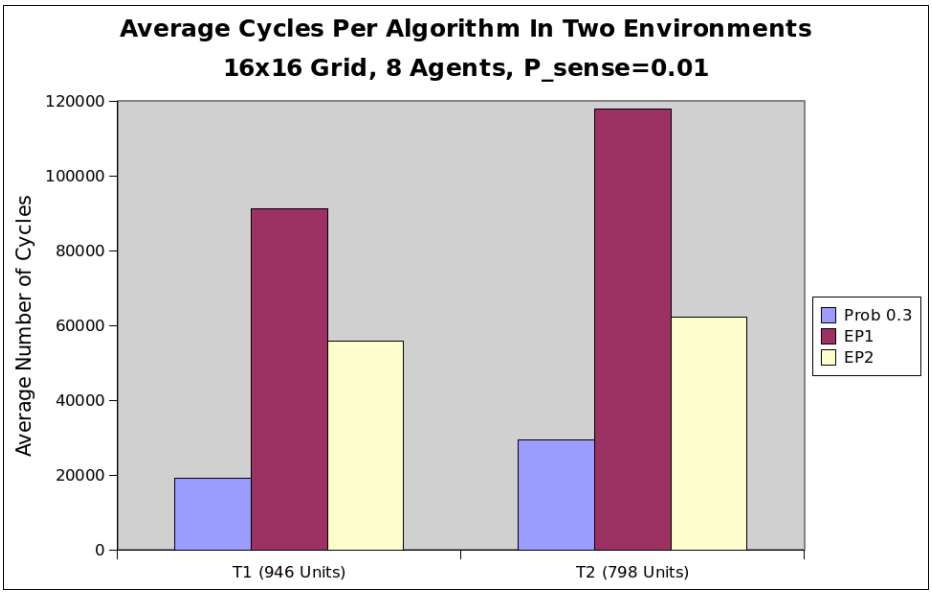

Figure 4.5: Comparison of average cycles between the probabilistic algorithm and each variant of the memetic algorithm.

Experiments were carried out to compare each memetic variant with the probabilistic algorithm, where in the original algorithm ρ_{pu} and ρ_{de} are set to the constant value 0.3. In the second memetic variant, a threshold of 1.0 is used as fitness to keep a chromosome in an agent. The results of these experiments are shown in Figure 4.4 and Figure 4.5. Averages were taken over 30 runs on each terrain, where each execution was halted when the level condition of the terrain was reached. From Figure 4.4 and Figure 4.5, it can be seen that each variant of the memetic algorithm out-performs the probablistic version in terms of units of material moved. Here a unit is defined as a single 'block' of a single grid cell, *i.e.* a cell of height 1 contains 1 unit. Each variant performs much worse in terms of time to complete the leveling operation. It appears from these results that while the memetic version appears to be slower, it may be useful in developing stategies which are more conservative with respect to effort taken in relocating material.

Experiments were run in simulation similar to the one used in the probabilistic version of the algorithm. However rather than using the 3D output, the algorithm was instead run using a 2D command-line view to allow the simulation to proceed faster.

4.3 Prototype of a Leveling Robot

Prior to settling on the full use of simulation to first develop the terrain leveling system, a hardware prototype for a leveliing robot was assembled. The goal was to be able to test algorithms developed in low-fidelity simulations such as those used in Sections 4.2.1 and 4.2.3, using the hardware to verify the algorithms in a mock environment. This environment was to be composed of aquarium gravel, intended to implement the task environment discussed in Section 4.1.

4.3.1 Hardware

Component Type	Model
Chassis	Dagu Rover 5 (with encoders)
Mainboard	Pololu A-Star 32U4 Robot Controller
IMU	Pololu AltIMU-10 v4
Wireless	X-Bee Pro

Table 4.1: Main hardware for prototype leveling robot

The prototype is designed around the Dagu Rover 5 chassis, a tracked vehicle with two front-wheel motors and encoders. A tracked skid-steer vehicle was selected as such a model reflects what is typically seen in real-world construction vehicles. The vehicle benefits from the added traction provided by tracks in the gravel environment used for testing. Unfortunately, the nature of skid-steering makes odometry inherently error-prone. This makes the odometry-aided algorithm described in Section 7.1.1 difficult to implement correctly on this particular prototype. The Dagu chassis allows reconfiguring the body height, by adjusting the inclination of the legs; the height may vary by a maximum of 3.8cm. The motors which ship with the chassis have a gear ratio of 86.8:1, with a maximum speed of 25cm/s. The included encoders are stated to provide 1000 counts per 3 revolutions, or 333.33 Counts Per Revolution (CPR).

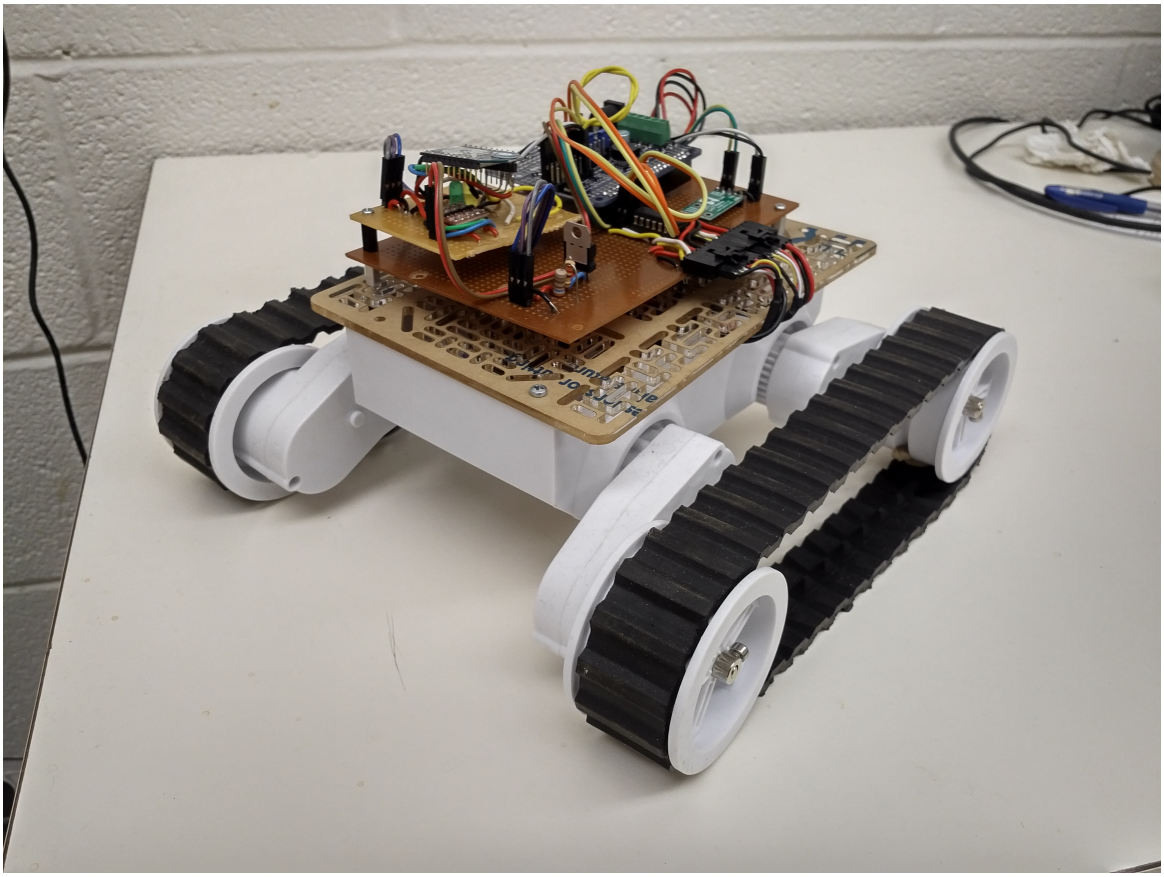

Figure 4.6: Dagu Rover 5 with mounted hardware.

The mainboard selected was the A-Star 32U4. Pololu offers this platform in several packages, one of which is a full robot controller. This variation includes a dual-motor driver, as well as compatibility with the Raspberry Pi (via the Pi HAT specification). The A-Star 32U4 itself is based on the Atmel ATmega 32U4 microcontroller, a 16MHz CPU with 32kB of flash memory and 2.5kB of SRAM. The board is compatible with the Arduino prototyping platform [1], and is hardware-equivalent to the Arduino Leonardo. Compatibility with the Aduino software ecosystem allows for more rapid development that what is generally possible when developing with a microcontroller on its own. A large number of libraries for interfacing with various kinds of hardware peripherals are available under open-source licenses, which decreases overall

development time. Pololu also provides several Arduino-compatible libraries for their hardware, including a library specifically for the A-Star 32U4. The A-Star library does not include methods for interacting with wheel encoders, however one included with the Zumo 32U4 (discussed in Section 4.3.3) was ported with relative ease. This assumes however that the encoders are wired the same as those included with the Zumo platform.

The 32U4-based boards include only a small number of interrupt pins, and several of those are multiplexed with other board features which may be needed. Each wheel encoder has two channels (A and B), each of which must be monitored via interrupt to count revolutions. To reduce the number of needed interrupt pins, the Zumo 32U4 XORs each channel together and connects the resulting output to a single interrupt pin. The second encoder channel (channel B) is connected to a non-interrupting input. The resulting XOR'd signal and the signal from channel B can be used to reconstruct the signal from channel A in software by XORing the inputs, since $(A \oplus B) \oplus B = A$.

The IMU contains an accelerometer, gyroscope, magnetometer and a barometric altimeter. These are the LSM303D containing both the accelerometer and magnetometer, the L3GD20H gyroscope and the LPS25H digital barometer.

The X-Bee radio fitted to the prototype is used only for debugging. It allows the device to report the state of the algorithm at each cycle, such as linear/angular acceleration or estimated position. A receiver module may be used via USB adapter on a PC, allowing the incoming data to be logged and/or plotted. This is particularly useful when calibrating devices such as the IMU.

4.3.2 Noted Issues

The prototype performed relatively well when its movement was tested in the aquarium gravel environment. It was discovered however that it is possible for gravel of

this size to become stuck in a space on the backside of the wheels. Gravel with larger granules would eliminate the issue, however if they are too large it may impact the robots ability to move or dislodge it. A shroud placed around the back of each wheel may also eliminate the issue, without needing to change the test environment.

The X-Bee radio used on the prototype operates at 3.3V, versus the 5V used by the A-Star mainboard. This necessitated the use of a logic level shifter to allow communication via USART between the devices. The A-Star provides pins for both a 5V and 3.3V rail, however the 3.3V pin did not appear to be powered when tested. This required the addition of a voltage divider circuit to be constructed on the perforated circuit board used also to mount the IMU and motor sockets.

4.3.3 Alternate Platform

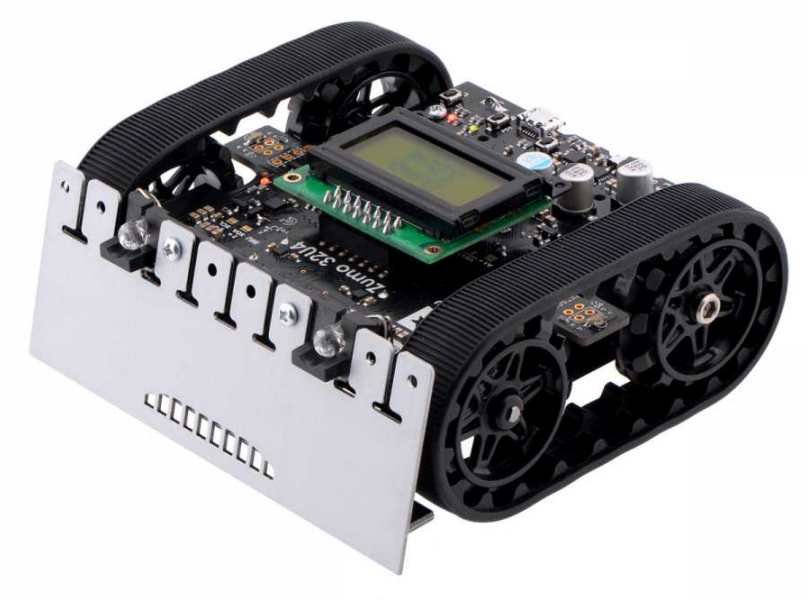

Figure 4.7: Pololu Zumo 32U4

Prior to the construction of the prototype depicted in Figure 4.6, a smaller version was built using the Pololu Zumo 32U4 platform (see Figure 4.7). This platform

is very similar electronically to the A-Star used in the existing prototype, and is also equivalent to the Arduino Leonardo platform. The Zumo also includes several infrared (IR) sensors, and integrates the same accelerometer and gyroscope found on the AltIMU-10 package.

It was initially selected as it is also a tracked vehicle, and includes a flat plate mounted at the front which could be used as a plow in a gravel environment. It was found however that the Zumo has very little clearance between the ground and the battery pack which is mounted at the bottom of the unit. It would frequently get stuck on top of relatively small mounds of gravel, and eventually dig itself in as the tracks spun in place. The front plate also did not fare well as a gravel plow, and often prevented the vehicle from moving forwards in many instances.

4.4 Discussion

Experimenting with both the probabilistic and memetic terrain leveling algorithms in simulation helped validate both the terrain leveling concept itself, as well as the individual algorithms. A simulator which allowed both 3D rendering and 'offline' execution was useful for first visually validating results and debugging issues, and later executing the system at full speed to gather performance metrics. The restrictions placed on each algorithm (and each simulator, by extension) such as discrete movement are helpful for early prototyping, however they are naturally unrealistic for further development. In order to carry the idea further, the algorithms and simulation must either be updated to operate in a continuous environment, or a real-world environment must be created. The Dagu-based prototype described in Section 4.3, while relatively small, still required a large gravel-filled space in which to operate when the need to eventually experiment with multiple agents is considered. Developing the

hardware also requires writing software for interfacing with sensors and peripherals, in addition to developing the behaviour algorihm itself. The desire to avoid the complications of hardware development, as well as the need for a high-fidelity simulation of deformable terrain, led to the development of the simulator discussed in Chapter 5.

4.5 Summary

Here we have provided a formal definiton for the leveling task, which provides context for discussion in subsequent sections of this thesis. This definition provides several assumptions the algorithms (as well as simulations) are bound to in their design. The algorithms described were each tested within a simulated environment, however the simulations utilized were of low fidelity. The low-fidelity simulations are helpful with respect to validating the algorithms, however one can see that they are simplistic in their design and ultimately restrict the types of experiments which may be carried out. Instead we wish to develop these algorithms in a more realistic environment, leading to the high-fidelity simulator described in Chapter 5. Finally, this chapter also described an early hardware prototype of a leveling robot. Development in hardware was eventually abandoned in order to persue a high-fidelity simulator, due to in part to issues in creating a suitable operating environment. However, the prototype provided inspiration for the design of the simulated agents and helped tie the simulations to reality.

Chapter 5

High-Fidelity Simulation of a Leveling Robot

This chapter describes a high-fidelity simulator used to simulate a single terrain leveling robot, called *Sandbox*. This simulator is used to generate training data for use in the neural network-based simulator described in Chapter 6. As discussed in Chapter 4, low-fidelity simulation of a terrain forces some assumptions on the algorithm itself. The voxel-based simulator used to develop the previous leveling algorithms was capable of providing only discrete movement to the agents. Furthermore, the dynamics of a granular terrain were not represented, where instead the terrain was represented by a number of blocks which could be turned 'on' or 'off' by an agent. The high-fidelity simulator is capable of simulating a volume of deformable terrain by one of two methods, as well as a more accurate representation of the leveling robots. This simulation however is non-real-time.

In order to accurately train the real-time neural network-based system described in Chapter 6, a rather large amount of training data capturing both the movement of a robot as well as the terrain must be generated. Rather than capturing data from

a real-robot, this work opts instead to generate the data using Sandbox. Generating data using a simulated environment allows more information to be captured in a more straight-forward manner than would be possible using a real-world system. Capturing movement of a robot would require some method of tracking via an overhead camera, or very accurate odometry from the robot itself. Recording information about the movement of the terrain would at the least require an overhead depth camera, and any more information beyond changes in height would be much harder to capture. Furthermore, one of the motivations for creating the simulator described in Chapter 6 is the absence of any real-world robot or testing environment to work with.

The fundamental purpose of Sandbox is to allow the simulation of leveling robot/vehicle prototypes, as well as a terrain which may be deformed by these robots. It allows simulating (in principle) a wide variety of vehicles, however it has been tested primarily with both wheeled and tracked skid-steer vehicles similar to the one described in Section 4.3. Simulation of deformable terrain is carried out using either using a volume of discrete spherical particles, or a deformable mesh model implemented by Chrono called *Soil Contact Model (SCM)* [39].

Section 5.1 provides further detail on the design and architecture of the simulation software, and describes the methods available for simulating deformable terrain. Section 5.2 provides a discussion of the tools used to design and represent a model robot within the simulator. Section 5.3 discusses the use of the simulator to generate training data for the neural network-based simulator covered in Chapter 6. Section 5.4 provides quantitative results comparing the computational performance of several aspects of the simulator, as well as qualitative discussion of these aspects in their practical use.

5.1 Architecture

Sandbox is based on Chrono Engine [41], a library which prioritizes faithfulness to physical realism over real-time performance. Chrono is not a stand-alone simulator, but may be thought of as a simulation *engine* which may be used to build simulation software. For the needs of Sandbox it provides a physics and collision system offering a higher degree of realism than is available in other freely-available physics engines. Chrono additionally provides Sandbox with several ways to visualize the systems being modeled, including real-time output via the Irrlicht rendering engine and offline output to be rendered by the POV-Ray ray-tracing engine.

While Chrono allows the simulation of soft-body dynamics and fluids, for the purposes of Sandbox only rigid-body simulation is considered. In this regard, objects composing a simulation in Chrono are made up of rigid-bodies with physical properties such as mass and inertia, defined by collision primitives (sphere, cube, mesh, etc.) and optionally *constraints* between other bodies. In Sandbox these constraints are often *joints*, such as fixed or revolute joints. Each of these bodies may optionally be visualized using *assets*. Assets provide a geometric description to the rendering system (Irrlicht or POV-Ray) for displaying the body, and the visual geometry does not necessarily have to be identical the collision geometry. Visual assets are not critical to the execution of the simulator, however they are useful for debugging simulations.

Using the low-level components for physics and rendering provided by Chrono, a framework for terrain leveling simulation was built which form Sandbox. These components may be thought of as the mid-level of the system. These components include a model importer, height-map importer, particle system, Unified Robot Description Format (URDF) [10] importer, and tracked vehicle. The model importer is used to load 3D models from file formats such as Wavefront .obj and convert them to the

Chrono mesh representation. The height-map importer is used to load a grayscale image, and optionally scale the image for use as a height-map terrain. The particle system generates a particle terrain from a supplied height-map, and generates a container in which to place the terrain. The URDF importer is used to convert a URDF representation of a robot to a Sandbox assembly (discussed further in Section 5.1.1). Finally, the tracked vehicle component adds facilities to a loaded assembly for the simulation of a tracked skid-steer robot. In particular, this component is responsible for generating track links about the assemblies wheels, and provides an interfacing for controlling motor speeds of the left and right tracks.

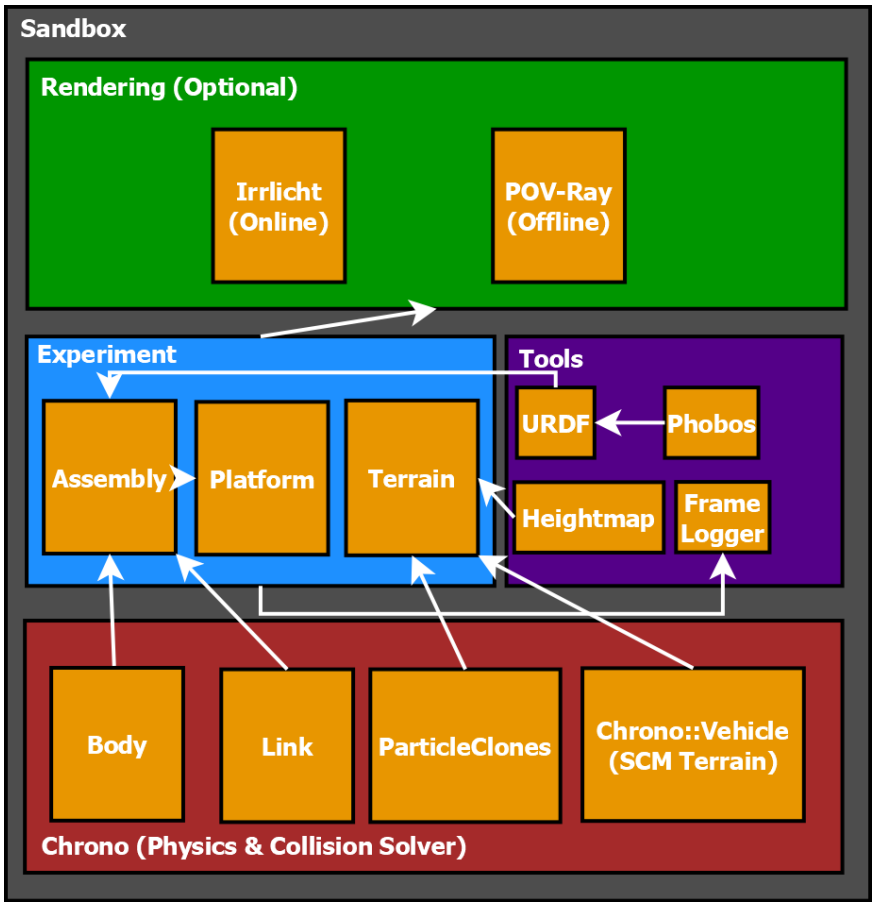

Figure 5.1: Block diagram of Sandbox's architecture

Sandbox uses these concepts as the basis of implementation for higher-level constructs used to implement the terrain leveling environment. The major components of the system are *experiments*, *assemblies*, *platforms*, and *terrain*. Each of these components are discussed further in the following sections. A block diagram depicting the high-level architecture of Sandbox may be seen in Figure 5.1.

5.1.1 Assembly

An assembly is a composition of geometry and connecting joints representing a system in the simulation. For the purposes of Sandbox, assemblies are the representation of a robot model loaded from a document in the XML-based Unified Robot Description Format (URDF). The URDF and the process of creating an assembly in this format is given in Section 5.2.

Sandbox assemblies support a subset of the representable geometries and joints available in the URDF. Components of the assembly may load geometry for both visual representation and collision—these geometries may be either boxes, cylinders or triangle meshes. The assembly will also load inertial objects defining component mass and inertia. Visual and collision geometries as well as inertial objects are represented internally by Chrono *bodies*.

Assemblies may contain either fixed, revolute, or engine joints. Each joint defines a constraint between two geometric components, as well as the relative transform of one component in relation to the other. Fixed joints maintain a fixed transform between bodies, while revolute joints allow one body to rotate about an axis relative to another. Engine joints are revolute joints which may be powered. Powered joints are not typically described by tools exporting URDF models—a method for specifying an engine joint to Sandbox is described in Section 5.2. Assembly joints are represented internally by Chrono as *links*.

5.1.2 Platform

A platform is a representation of a robot or vehicle simulated in the Sandbox environment. Platform objects provide a binding between a vehicle assembly and a control algorithm. Additionally, platforms provide emulation for hardware such as the IMU. Platforms assume that the vehicle being modeled is skid-steer, and algorithms implemented for a platform control the vehicle by varying the desired linear and angular velocities. The platform object converts these values into individual wheel speeds using specified constants for wheel base and wheel radius. This conversion is calculated using the kinematic model of a differential drive robot. Wheel speeds v_{left} and v_{right} are calculated in radians-per-second by

$$v_{left} = \frac{l - \frac{B\omega}{2}}{r}$$

$$v_{right} = \frac{l + \frac{B\omega}{2}}{r}$$

where B is the wheel base of the vehicle, r is the wheel radius, l is linear velocity, and ω is angular velocity.

5.1.3 Terrain

There are two supported methods for representing terrain in Sandbox, using either a volume of discrete particles or a method provided by the "Chrono::Vehicle" module based on SCM. Each terrain model provides a method to capture the current height-map as it is modified throughout the simulation.

The particle-based method generates a number of spheres to fill a volume defined by a grayscale height-map. For each pixel in the map, particles are generated vertically up to a maximum height defined in the experiment configuration file. Particles are

given a uniform radius and density. Particles are generated adjacent to each other such that each pixel in the height-map contains exactly one particle stack. The mass of each particle is calculated by $m = \frac{4\pi r^3 \rho}{3}$, and the moment inertia of each particle as $I_{XX} = r^2 m$ where r is the particle radius and ρ is the particle density.

The SCM-based method provided by Chrono simulates terrain using a deformable triangle mesh. Like the particle method, the mesh surface shape is defined by a grayscale height-map where each pixel in the height-map maps to a vertex in the resulting mesh. SCM is based on Bekker's terramechanics theory [40], and simulates the contact between a rigid body and a volume of plastically deformable soil.

For both terrain modeling methods, the height-map is typically scaled uniformly from its original size by the *xy_scale* parameter of the experiment configuration (See Figure 5.3). The resulting dimensions of a particle terrain are $(x, y) = (map_{width} \cdot particle_{radius} \cdot xy\_scale, map_{height} \cdot particle_{radius} \cdot xy\_scale)$. For SCM terrain, each vertex is placed in a grid orientation. Thus the resulting dimensions of an SCM terrain are $(x, y) = (map_{width} \cdot xy\_scale, map_{height} \cdot xy\_scale)$.

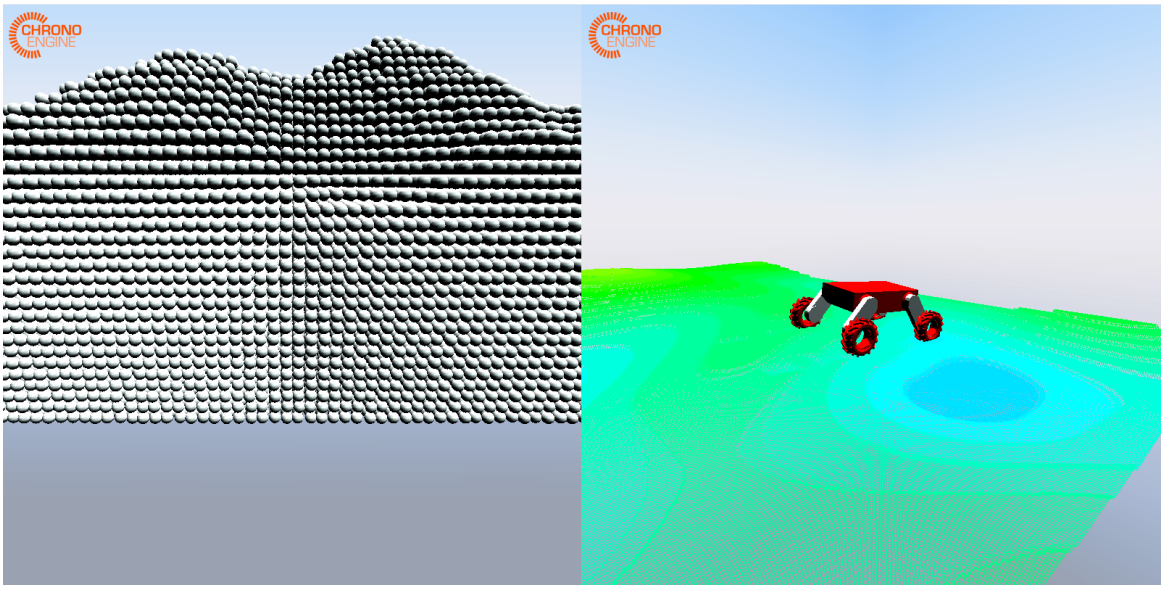

Figure 5.2: Profile view of a particle terrain (left), and wireframe view of a SCM mesh terrain (right)

5.1.4 Experiment

An experiment may be considered the 'root' object in the Sandbox system. Several experiments may be loaded, however they are separate entities and are required to be run in succession. Each experiment may define a single terrain as well as one or more platforms. The experiment object provides a method of initialization, as well as a method to step all child objects in a single frame.

Experiments are initialized using an externally-defined configuration file which is loaded at run-time. These files define the variables such as the initial position and orientation of a platform, the terrain height-map, terrain particle size, and the scaling values for the terrain (See Figure 5.3).

The experiment step method is used to step each platform in the system forward by one frame. This does not execute a step of the physics simulation (which is handled externally by Chrono), but instead executes a step of the platform algorithm.

Additionally, the step method deals with re-positioning agents at the outer bounds of the terrain. To allow data to be generated quickly, Sandbox implements a "wrap-around" feature for terrains such that a platform is teleported to the opposite side of the terrain when it reaches the boundary. Doing this allows terrain height-maps to be relatively small while still generating enough data for later use. The procedure necessitates that the terrain height-map loaded in the simulation is *tileable* (*i.e.* seamless on each side).

```
output_directory_prefix = ./tracked_noplow_static/
chrono_data_path = ../data/

[vehicle]
tracks = true
model = Dagu5
x = -20.0
y = 8.0
z = -20.0
r = 0.0
p = 0.0
h = 0.0

[map]
filename = terrain_large.bmp
scale = 5.0
particle_radius = 0.15
xy_scale = 0.4
model = particle

[raygrid]
resolution = 2.0

[experiment]
name = sparse_random
algorithm = random
```

Figure 5.3: Example experiment configuration file

5.2 Defining a Robot Model

Robot and vehicle models are defined in Sandbox using URDF. URDF is the standard XML format typically used by (and developed for) the Robot Operating System (ROS). URDF is used by several ROS-related simulation and visualization tools such as Gazebo and Rviz, and allow representing a robot model as a hierarchical scene-graph.

The components of a URDF file used primarily by Sandbox are *links* and *joints*, which differ from the naming conventions used by Sandbox and Chrono. URDF links define geometries and inertial objects which are stored as Chrono bodies. URDF joints define constraints between links—following the same naming used in Sandbox, but differing from the Chrono naming of constraint objects as links. An example URDF demonstrating links and joints is provided in Figure 5.4. URDF joint elements may contain the *limit* tag which typically defines the safety limits of joint. This tag may contain the *velocity* attribute, which when set to '1.0' forces Sandbox to interpret the joint as a powered engine joint if the type is also set to 'revolute'.

```
<?xml version="1.0"?>
<robot name="origins">
  <link name="base_link">
    <visual>
      <geometry>
        <cylinder length="0.6" radius="0.2"/>
      </geometry>
    </visual>
  </link>

  <link name="right_leg">
    <visual>
      <geometry>
        <box size="0.6 .2 .1"/>
      </geometry>
      <origin rpy="0 1.57075 0" xyz="0 0 -0.3"/>
    </visual>
  </link>

  <joint name="base_to_right_leg" type="fixed">
    <parent link="base_link"/>
    <child link="right_leg"/>
    <origin xyz="0.22 0 .25"/>
  </joint>

</robot>
```

Figure 5.4: Example URDF file

While one could construct robot and vehicle models manually using URDF, it is more practical to create the model visually in a modeling tool. This work has made use of *Phobos* [8], which is a plugin for the Blender modeling software. Phobos was originally created for use with MARS (Machina Arte Robotum Simulans) [5], however its use of standard URDF allows it to be useful for any environment which can use the format. Using Blender, Phobos allows detailed geometries to be created as part of the visual and collision representation of the robot and allows Wavefront OBJ meshes to be exported along-side the URDF document. Phobos also allows one to define parent-child relationships between objects using built-in Blender tools, and provides a means to define joints and constraints on these relations. Additionally, the tool also allows the automatic generation of appropriate inertial objects based on the body geometric properties and user-defined mass of each robot component (URDF link).

Whether defined manually or using Phobos, Sandbox makes use of a naming convention when constructing models from URDF files. Four wheel bodies are required to be present in the model, each named 'bl_Wheel', 'br_Wheel', 'fl_Wheel', and 'fr_Wheel'. The positions of these bodies are used to generate track links on tracked vehicles. The names are used when searching for bodies in the Chrono scene graph, both for track generation as well as when applying power to wheels attached using engine revolute joints. Exactly one body named 'Body' is required to be present in the URDF, which is assumed by Sandbox to be the chassis of the robot. This body is used to generate IMU data for the platform.

5.3 Generating Training Data

The primary purpose of the Sandbox simulation as it relates to this work is for the generation of training data to feed into a neural network. Data is generated during

the execution of an experiment (see Section 5.1.4) which is configured to run a single vehicle using a particular control algorithm designed for data generation. The experiment is run for a predefined number of simulation steps, with a number of parameters of the simulation being recorded at the end of each step. The collection of recorded data for an experiment is referred to as a *session*, and that data for each step is referred to as a *frame*. All of the available parameters which are recorded in each frame are listed in Table 5.1. Not all of these captured parameters are necessarily used by the neural network, but are available for future use or are used in data pre-processing. As described in Section 5.1.3, a corresponding top-down image of the current terrain height-field is also recorded with each frame.

Table 5.1: Variables recorded by Chrono simulation

Variable	Description	Input/Output
x	Absolute X position	N/A
y	Absolute Y position	N/A
Δx	Change in X position	Output
Δy	Change in Y position	Output
$\Delta\theta$	Change in yaw	Output
v_{left}	Left motor speed (rps)	Input
v_{left}	Right motor speed (rps)	Input
φ	Vehicle pitch (radians)	Input
ψ	Vehicle roll (radians)	Input
l	Desired linear velocity	N/A
ω	Desired angular velocity	N/A

The x, y, and θ values are captured from the absolute orientation of the robot chassis as computed internally by Chrono. Δx, Δy, and $\Delta\theta$ are captured as the difference between the x, y, and θ values of the current frame and the previous frame, such that each delta is relative to the absolute world position of the chassis. Values φ and ψ are also captured from the orientation of the robot chassis as reported by Chrono.

The vehicle control algorithm used to generate data attempts to move the vehicle

Input : Number of frames n, where $n \geq 8$
Output: Desired linear velocity l, desired angular velocity ω
$i \leftarrow 0$;
for $i < n$ **do**
  if $0 \equiv i \mod 250$ **then**
    $r \leftarrow uniform\_rand(0.25, 1.0)$;
  if $i \leq \frac{n}{8}$ **then**
    $l \leftarrow 3r$;
    $\omega \leftarrow 0$;
  else if $\frac{n}{8} \leq i < \frac{2n}{8}$ **then**
    $l \leftarrow -3r$;
    $\omega \leftarrow 0$;
  else if $\frac{2n}{8} \leq i < \frac{3n}{8}$ **then**
    $l \leftarrow 0$;
    $\omega \leftarrow \frac{r\pi}{2}$;
  else if $\frac{3n}{8} \leq i < \frac{4n}{8}$ **then**
    $l \leftarrow 0$;
    $\omega \leftarrow \frac{-r\pi}{2}$;
  else if $\frac{4n}{8} \leq i < \frac{5n}{8}$ **then**
    $l \leftarrow 3r$;
    $\omega \leftarrow \frac{r\pi}{2}$;
  else if $\frac{5n}{8} \leq i < \frac{6n}{8}$ **then**
    $l \leftarrow -3r$;
    $\omega \leftarrow \frac{-r\pi}{2}$;
  else if $\frac{6n}{8} \leq i < \frac{7n}{8}$ **then**
    $l \leftarrow -3r$;
    $\omega \leftarrow \frac{r\pi}{2}$;
  else if $\frac{7n}{8} \leq i < n$ **then**
    $l \leftarrow 3r$;
    $\omega \leftarrow \frac{-r\pi}{2}$;
  $i \leftarrow i + 1$
end

Algorithm 5: Control algorithm used for data generation

over a terrain using a variety of motor inputs. Since two of the inputs to each neural network are the left and right motor speeds, it is important to generate a variety of speeds which can be sampled over the total range. Rather than generating random motor speed values directly, the algorithm instead sets the desired linear and angular velocities. The motor speeds are then calculated according to the equations in Section 5.1.2. Each session is run for n frames, separated into $\frac{n}{8}$-frame blocks—each with varying combinations of the vehicles' desired linear and angular velocities. For example, the first block runs with forward linear velocity and zero angular, the second with negative forward velocity and zero angular, the third with forward linear velocity and right angular velocity, and so on. Each of these blocks are themselves split into 250-frame segments, each of which apply a random scaling coefficient to the maximum desired linear or angular velocity. This process is illustrated in Algorithm 5, in which l and ω refer to *desired* linear and angular velocity respectively.

The simulator also generates data to be used in the simulation of terrain deformation. The method used depends on the terrain model selected, however in both cases Sandbox generates a series of height-map images—one for each frame in the session. For height-map generation in the particle-based method, there are potentially a large number of discrete objects making up the terrain volume from which we wish to extract a surface. To capture a height-map from the particles, ray-casting is utilized. A number of rays are generated starting from a height above the terrain and ending at height $y = 0$. The rays are cast uniformly over the area of the terrain, with a density which may be defined in the experiment configuration. The ray-casts return a value representing a relative distance from the starting point of the ray. Each of these distance values are converted to a grayscale pixel value and written to an image representing the height-map of the frame. For SCM-based terrain, height-map generation is more straightforward since the terrain exists as a triangle mesh (already

a convex surface with known heights at each point). For each vertex in the terrain mesh, the height is recorded. Each height value is converted to a pixel intensity as a fraction of the maximum height found in the terrain, and each pixel is recorded as a height-map image for the frame.

5.4 Performance

Several approaches to vehicle and terrain modeling have been tested in Sandbox with regards to both computational performance and usability. Chrono supports both single-threaded and multi-threaded collision solvers, each of which have been experimented with in the design of Sandbox. As outlined in Section 5.1.3, Sandbox implements two methods of simulation for deformable terrain. Each of these methods have also been experimented with in terms of performance and usability.

The single-threaded and multi-threaded collision solvers are compared using the particle terrain model, as it is the most likely candidate to benefit from multi-threading due to the large number of bodies which must be simulated. However, it was not immediately obvious prior to experimentation whether the size of the problem warranted the extra overhead created by the introduction of multiple threads.

Each terrain model available in Sandbox is useful for simulating different kinds of deformable terrain (*e.g.* loose gravel or soil). Thus it is beneficial to compare and contrast each model in terms of computational performance, as well as qualitatively in terms of their implementation and use.

5.4.1 Computational Performance

Evaluation with respect to computational performance of the simulator is carried out by way of quantitative analysis. The metrics compared are mean frame time and total

memory utilization, where mean frame time is the time to complete one algorithmic step of the platform followed by one dynamics update by Chrono.

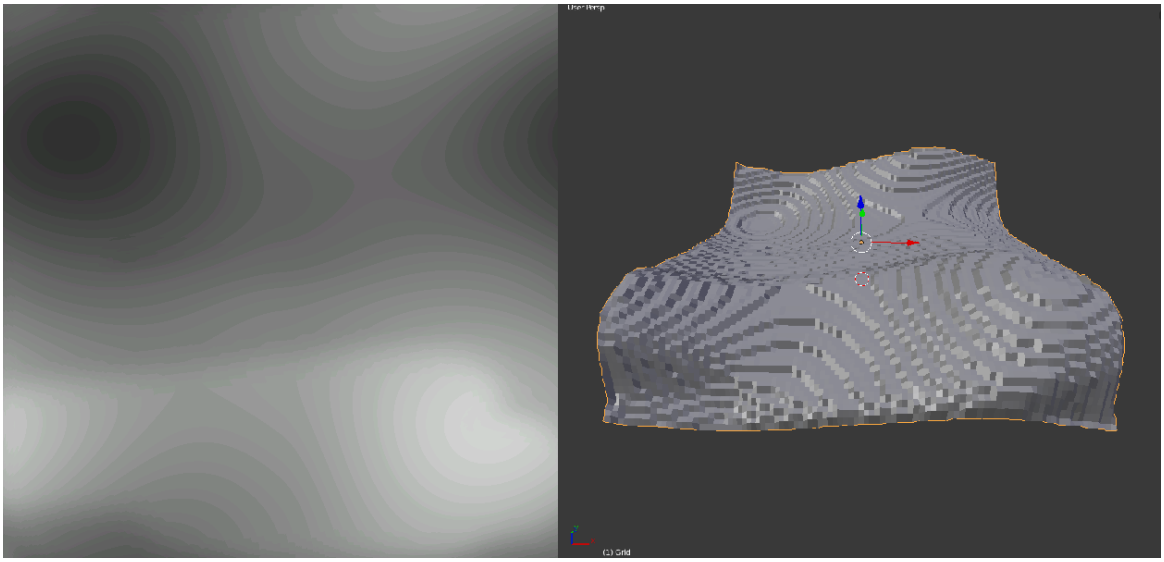

Figure 5.5: Height-map used in performance evaluation (left) and the same height-map rendered in 3D (right)

Comparisons were conducted using three different scales applied uniformly to the width and length of the terrain for each of the particle and SCM terrain models (see Table 5.2). Each test was run using the same 600x600 pixel height-map, scaled to the same in-simulation dimensions for each model. Due to the way each model interprets the height-map, different scaling values are used to reach the desired dimensions. The maximum height of each terrain for each model is 5 units. In the case of the particle model, the particle radius is 0.15.

Table 5.2: Terrain parameters for performance evaluation

	Particle			SCM		
	Small	Medium	Large	Small	Medium	Large
Dimensions	18x18	36x36	72x72	18x18	36x36	72x72
Scale	0.1	0.2	0.4	0.03	0.06	0.12
Num. Particles	30642	112570	490379	N/A	N/A	N/A

Evaluations were executed on a PC with a Core i7-4700HQ CPU (8 logical cores) and 16GB of RAM, where the simulator was compiled with version 17.0.1 of the Intel C++ compiler. Multi-threaded evaluation was carried out using a maximum of 8 threads, and uses the parallel collision solver provided by the 'Chrono::Parallel' module.

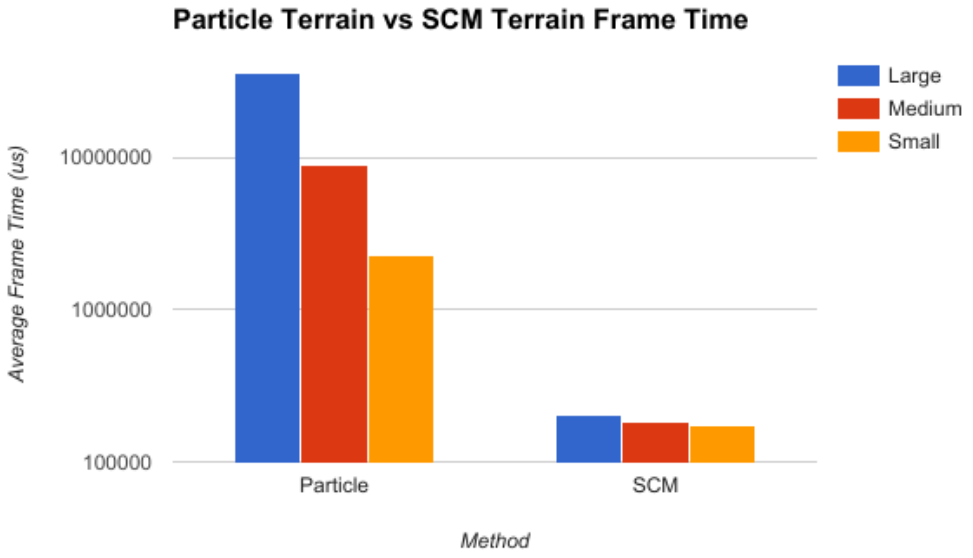

Figure 5.6: Mean per-frame execution time using particle and SCM terrain models

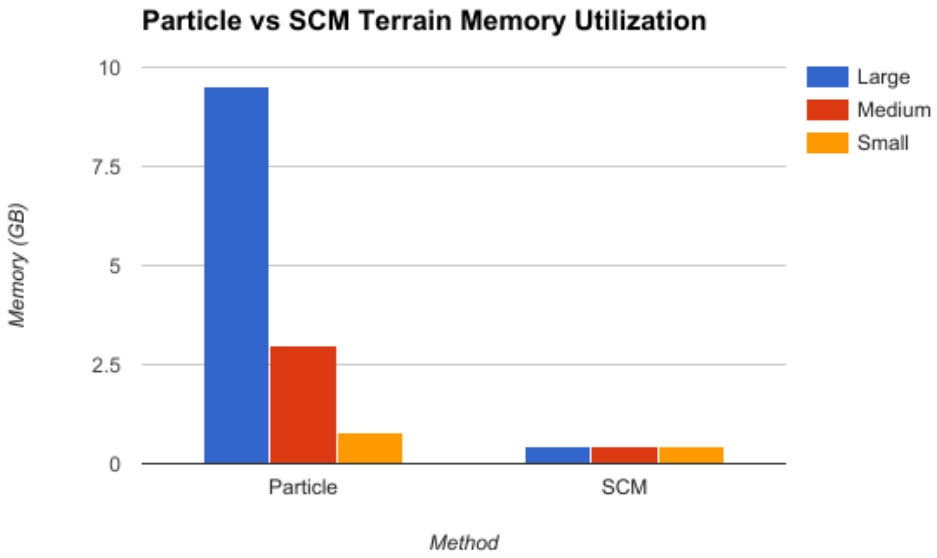

Figure 5.7: Memory utilization of particle and SCM terrain models

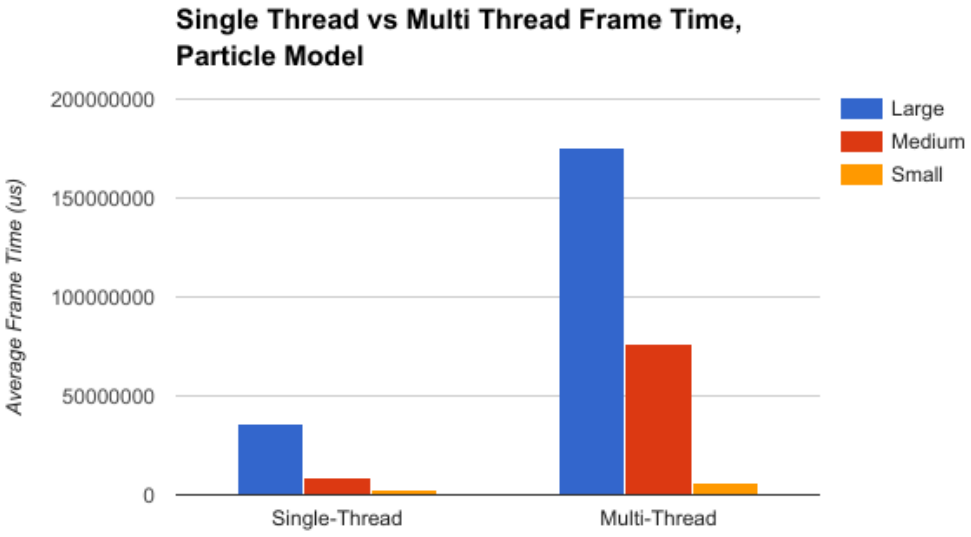

Figure 5.8: Mean per-frame execution time on particle terrain in single-threaded and multi-threaded mode

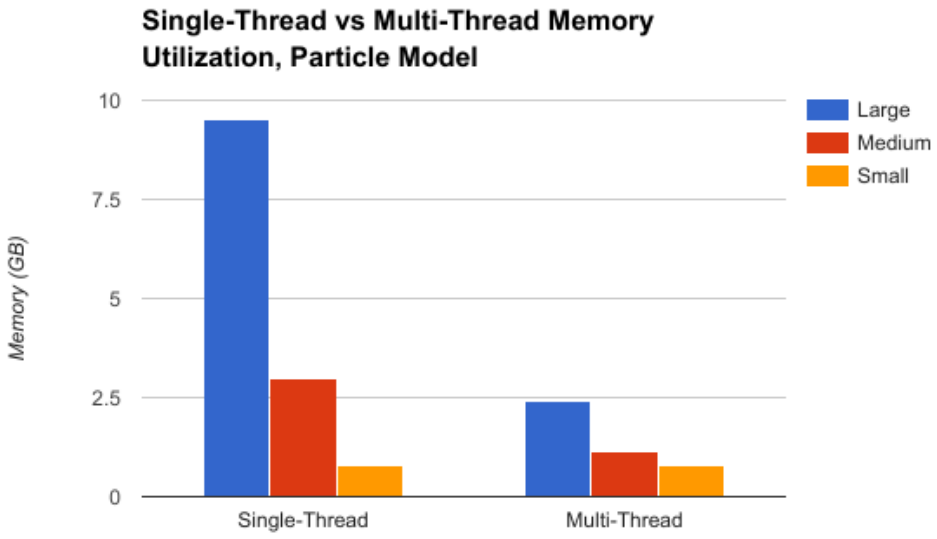

Figure 5.9: Memory utilization of particle terrain in single-threaded and multi-threaded mode

Figure 5.6 demonstrates the mean time per frame in microseconds for each terrain size using either particle or SCM terrain. The particle method displays much larger steps in execution time for each terrain size. This is due in large part to the higher number of bodies which must be simulated as the terrain grows—particularly the larger number of *contact points* which must be computed by the collision solver. The time taken to execute a frame also grows with terrain size in the SCM model, however much more slowly.

Figure 5.7 demonstrates the total memory utilization of the simulation using either particle or SCM terrain. Following a similar pattern to frame execution time, the particle terrain consumes substantially more memory than the SCM method and with a rather wide disparity between terrain sizes within the particle model. However unlike execution time, which appears to grow by a linear factor, memory use appears to grow exponentially. Memory utilization in the SCM model appears to remain flat

across terrain size.

Figure 5.8 demonstrates the mean frame execution time of the particle model for each terrain size using both the single-threaded and multi-threaded collision solver in Chrono. At all sizes, the multi-threaded solver is slower than the single-threaded version. This is likely due to the overhead involved in the parallel model which must manage multiple threads of execution and communicate information between them. It is probable that as the number of particles increases, the gap between single and multi-threaded performance would close as parallelism becomes more efficient relative to the size of the problem. Due to the constraints of the evaluation hardware, this has not been tested. It is worth noting in Figure 5.9 that the memory utilization decreases for each terrain size when using the parallel solver. This would support much longer terrain sizes, however the time to execute becomes prohibitive when used for the purpose data generation as it is in this work.

5.4.2 Qualitative Evaluation

The practicalities and limitations of the available terrain and vehicle models are evaluated qualitatively. Particular attention is paid to the terrain models, as that is where differences appear to be the most pronounced.

Referring to Figure 5.7 one can see that large terrains using the particle model quickly consume the majority of available RAM on the test system. It is very likely the majority of memory available on most commodity PCs available at the time of writing would also be exhausted. The effect is great enough that the ‘large’ terrain dimension in Section 5.4 was selected by trial-and-error to find the largest particle terrain that could reasonably fit in memory on the test machine alongside other necessary running software. If used strictly for generating training data however, the terrain size is not generally an issue since the wrap-around teleportation applied will give the illusion

of infinite terrain to the vehicle. An attempt was made to reduce the memory consumption of the particle model, by using the Chrono-provided *ParticleClones* object. Rather than creating a unique body for each particle, the ParticleClones object shares properties between a large number of bodies. Using this system however, the particles no longer appeared to collide with the container holding the volume in place.

The method of tracking height-map changes between frames in the particle model is particularly intensive (see Section 5.3). The ray-cast operation in the computational evaluations consumed a sizeable portion of the recorded average frame time in the 'large terrain' trial. This feature should therefore be disabled if the simulator is not being used to generate height-map data throughout the simulation.

Tracked vehicle models running on the particle terrain model often have several issues. Depending on the properties of the particles themselves (namely mass, density, and radius) vehicle tracks will often become dislodged and slide off of the wheels they are wrapped around. This has been observed both with the vehicle track model developed for Sandbox, as well as the tracked vehicle model distributed with the 'Chrono::Vehicle' module. Using smaller, heavier particles appears to alleviate the issue. However, using smaller particles has the effect of shrinking the dimensions of the terrain, requiring a greater scaling value to be used for the height-map. At large dimensions, this generates an intractable number of particles to be simulated on typical commodity hardware.

Simulations using the SCM terrain model are less computationally intensive than particle terrains by a fairly wide margin. This should, in theory, allow much larger terrains to be simulated versus the particle terrain model. In practice, the area of each polygon making up the terrain mesh is required to be limited relative to the size of the vehicle which will come in contact with it. Thus to create very large terrains, a high-resolution height-map must be used and then scaled down to the desired in-simulation

dimensions in order to increase the polygonal density. Low-density meshes result in what is best described as undefined behaviour. Typically however, the result is the instantaneous break-up of the vehicle tracks (if present) and/or a rapid 'flying-away' of the vehicle itself.

The speed of the SCM model allows generating a much larger number of frames for training data in a period versus what is possible using particle terrain. However, the SCM model appears best suited for simulating granular material which is relatively cohesive, such such as soil or clay. Simulating loose particles such as gravel is more difficult due to the nature of the model. The particle model by its own nature excels in the simulation of loosely packed granules.

5.5 Summary

This chapter has presented Sandbox, a simulator built on the Chrono engine designed for terrain leveling simulation. In addition to Chrono, Sandbox makes use of other tools such as URDF robot definitions as well as the Phobos add-on for the modeling software Blender. We have described how Sandbox can be used to generate training data for a neural network, using a specialized algorithm to generate a large number of data points as the simulated vehicle moves about the terrain. Several performance metrics are compared, in particular demonstrating the disparity between a particle-based terrain simulation method and the Soil Contact Model (SCM). Performance characteristics of both the single-threaded and multi-threaded collision solver provided by Chrono are also discussed. A qualitative evaluation of Sandbox (and by extension, Chrono) is given, in which the practicalities of various aspects of the simulator are discussed.

Chapter 6

Neural Network-Based Simulation

Using the training data obtained from the simulator described in Chapter 5, a much computationally cheaper simulation can be leveraged for prototyping robot control methods. The simulator developed for this purpose provides basic 2D visuals, and replaces a physics engine with a neural network. This chapter describes *Alexi*, which has been designed to be used in conjunction with training data obtained from Sandbox in order to allow a computationally efficient terrain leveling simulation. In addition to the simulator itself, several tools used to support the training and testing of neural network models have been developed. Additionally, tools for pre-processing training data have also been implemented.

Section 6.1 provides an overview of the implementation and design of the simulator, as well as the tools developed to support data preprocessing and network training. Section 6.2 describes the first implementation of the Alexi system, and provides results related to the training of each neural network used in the simulator for robot kinematics. Section 6.3 provides the implementation details for a later implementation of Alexi, again providing results for networks trained for robot kinematics, as well as a network trained for terrain deformation. Finally, Section 6.5 offers a discussion of the

concept as a whole, and offers a qualitative comparision of the results of each version of Alexi.

6.1 Implementation Overview

Alexi has been implemented in two different versions, here referred to as *V1* and *V2*. Each version is composed of three tools. In addition to the simulator itself, tools for data preprocessing and network training have been developed, where all three are intended to be used as a 'pipeline' following the generation of data using Sandbox. The data preprocessor is used in the first step, and is used to apply the various transformations discussed in Sections 6.2.2 and 6.3.2. The result of the transformations are saved to disk in a format specific to the neural network library used. The training tool used in the next step provides an interface to load the preprocessed data, and execute training of each network. The trained networks are saved to disk, again in a format specific to the neural network library used.

Both V1 and V2 of Alexi provide a 2D simulator to visualize the response of the robot, as well as to provide input to the neural networks for physics emulation. The simulator provides a view of a height-map, as well as a marker illustrating the position and orientation of the robot. The simulation is 2D in the sense that information is rendered in only the x and y dimensions. However, the height-map pixel information is used to construct a third dimension, providing pitch and roll information to the neural network input. This is achieved by finding the slope of the line between a point in front of and behind the vehicle for pitch, and between points on the left and right side for roll, where the height of each point is captured from the pixel intensity of the height-map. Mathematically, these pitch and roll angles are found by $\varphi = \arctan \frac{h_{front} - h_{back}}{2r}$ and $\psi = \arctan \frac{h_{left} - h_{right}}{2r}$ respectively, where h_* is the height

(represented by pixel intensity) at each point about the vehicle, and r is the radius of the area covered by the vehicle.

Each version of Alexi differ in their underlying implementation. Version 1 was developed in C++ and depends on the Fast Artificial Neural Network (FANN) library [44]. Version 2 was developed in Python and depends on the Keras neural network package [4]. The 2D simulators for each version are largely the same, however version 1 is developed with the Simple DirectMedia Layer (SDL) library directly, whereas version 2 is developed with PyGame (a thin Python interface to SDL). SDL (and by extension PyGame) are graphics libraries which may be used to implement 2D or 3D rendering.

In both versions of Alexi, the vehicle position is updated by

$$x \leftarrow x + v\cos(\theta)$$

$$y \leftarrow y + v\sin(\theta)$$

where v is vehicle speed. Vehicle yaw is simply updated each frame by $\theta \leftarrow \theta + \Delta\theta$. The input v may either be predicted directly by one of the neural networks, or computed as $v = \sqrt{\Delta x^2 + \Delta y^2}$ where Δx and Δy are predicted by a neural network. Because v must always be positive, it is multiplied by -1.0 when the desired linear velocity is also negative.

Alexi V2 allows predicting deformation of the terrain. To simulate changes to the terrain, values of the pixel intensity of the height-map are updated in a window centered around the robot position. The intensity values (representing height) are offset by a delta value Δh predicted by a neural network, such that $h \leftarrow h + \Delta h$ for each pixel in the window.

6.2 Alexi: Version 1

6.2.1 Network Architecture

Version 1 employs two separate feed-forward neural networks, each of which are used to predict robot speed v and change in robot yaw $\Delta\theta$ respectively. As was similarly found by Pretorius et al, using separate networks with single-node output vectors tended to preform better than a single network used to predict all values [48]. Each network is provided the same four input variables: v_{left}, v_{right}, φ, and ψ.

Each network is constructed with a single hidden, in addition to the input and output layers. The hidden layer contains 20 nodes. Nodes in the input layer use linear activation functions, while the hidden layer and output layer use the hyperbolic tangent function *tanh* (see Figure 6.1). The *tanh* function was chosen as it is symmetric about zero with a range of (-1, 1), appropriate for the min-max normalized data fed into each network. A coefficient for the steepness of the *tanh* function was chosen to be 0.5 such that

$$\tanh(x) = \frac{1 - e^{-0.5x}}{1 + e^{-0.5x}}$$

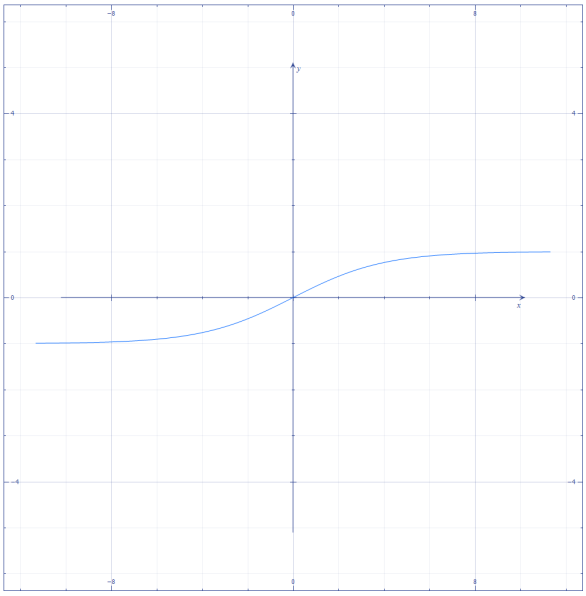

Figure 6.1: Activation function used in Alexi V1

Initial weights for each network were initialized using *normalized initialization* [30]. Using this method, weights are initialized as uniform random values in the range

$$\left(-\sqrt{\frac{6}{n_{in} + n_{out}}}, \sqrt{\frac{6}{n_{in} + n_{out}}} \right)$$

where n_{in} is the number of input connections to a node, and n_{out} is the number of output connections from a node.

6.2.2 Data Preprocessing

Several pre-processing stages are applied to the raw data captured from Sandbox. These include outlier removal, normalization, quantization, and optionally the application of an exponential smoothing filter. Each frame captured from Sandbox is treated as an individual sample when fed into the neural network; however in this step each variable is processed (independently from other variables) as a series across

the frames.

Frames passed in as training input are culled based on outliers found in several variables. Frames are removed where $\Delta\theta > 0.25$ or $\Delta\theta < 0.25$, $\varphi > 1.22$ or $\varphi < -1.22$, $\psi > 1.22$ or $\psi < -1.22$, and where $v - v_\mu > v_\sigma$ (where v_μ is the mean of v and v_σ is the standard deviation of v).

Each input and output value is min-max normalized to a value in the range $(-1, 1)$ by the equation

$$x'_i = \frac{x_i - \frac{X_{max} - X_{min}}{2}}{\frac{X_{max} - X_{min}}{2}}$$

where X_{min} and X_{max} are the minimum and maximum values found in the series. The quantization operation maps several ranges of value to a single value such that

$$x = \begin{cases} -0.8 & -1.0 < x \leq -0.8 \\ -0.6 & -0.8 < x \leq -0.6 \\ -0.4 & -0.6 < x \leq -0.4 \\ -0.2 & -0.4 < x \leq -0.2 \\ 0 & -0.2 < x < 0.2 \\ 0.2 & 0.2 \leq x < 0.4 \\ 0.4 & 0.4 \leq x < 0.6 \\ 0.6 & 0.6 \leq x < 0.8 \\ 0.8 & 0.8 \leq x < 1.0 \end{cases}$$

for each value x in the series. The operation assumes the data has previously been normalized to the range (-1,1).

The exponential smoothing filter is implemented by the equation

$$y_i = \alpha x_i - (1 - \alpha) y_{i-1}$$

where x_i is the i^{th} input in the series.

6.2.3 Training

Each network is trained using data from a single session containing 16000 frames, where each frame provides a training example to the network. The motion data was generated from a tracked robot model moving about a static particle terrain. The terrain was held static as version 1 of Alexi does not model changes in the terrain height-map. The networks are trained using the *iRPROP-* algorithm available in the FANN library [36]. iRPROP- is a variant of the RPROP (resilient backpropagation) algorithm [15], which is in turn a learning heuristic applied to the standard backpropagation algorithm. A learning rate of 1.25 was used to train the network predicting v, while a learning rate of 0.25 was used to train the network predicting $\Delta\theta$. Training of each network is run for 2000 of epochs. The epoch which yields the lowest error on the *testing* set is kept as the final network configuration, where the testing set is another session captured from Sandbox which is not shown to the network during training.

6.2.4 Results

The metrics used to evaluate training performance of each network are the mean squared error and the coefficient of determination (R^2). These metrics are calculated between the real output logged by Sandbox in the session used as the testing set, and the output predicted by the neural network when given the same input values. Table 6.1 provides the MSE and R^2 value for each variable predicted.

The mean squared error, as the name implies, provides a measure of the average difference between the expected and predicted output of a neural network (given identical inputs). A value of zero would imply that the neural network perfectly predicts the output, and so lower values are generally better. There is no threshold for the MSE which is universally regarded as acceptable—the acceptable error largely depends on the problem and the data in question. It is therefore more beneficial to compare results with others found for a similar (or identical) problem. The coefficient of determination represents the squared correlation between the predicted output and the expected output. More specifically, it is a measure of the extent to which the variance in the dependent variable (predicted output) is predictable from the independent variable (expected output). A value of 1 implies that the dependent variable can be predicted without error from the independent variable. Thus, higher values a generally better and imply a better predictive ability for a neural network.

Table 6.1: MSE and R^2 of each network on testing set (Alexi V1)

Variable	MSE	R^2
v	0.0133	0.398
$\Delta\theta$	0.0062	0.56

Figures 6.2 and 6.3 provide plots of the real output for each frame of the testing set *vs.* the values for the same data predicted by the neural network. Plots illustrating R^2 are shown in Figures 6.4 and 6.5.

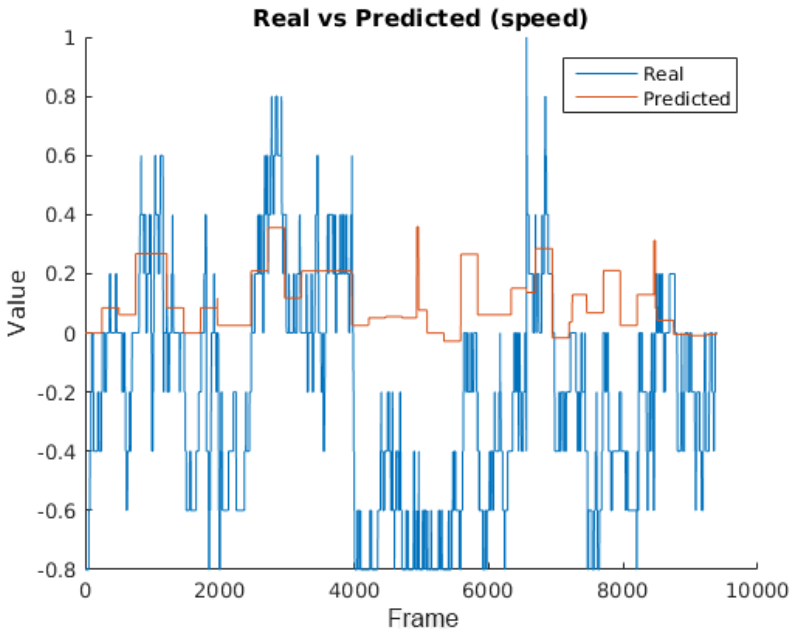

Figure 6.2: Comparison of real and predicted values on testing set for v (Alexi V1)

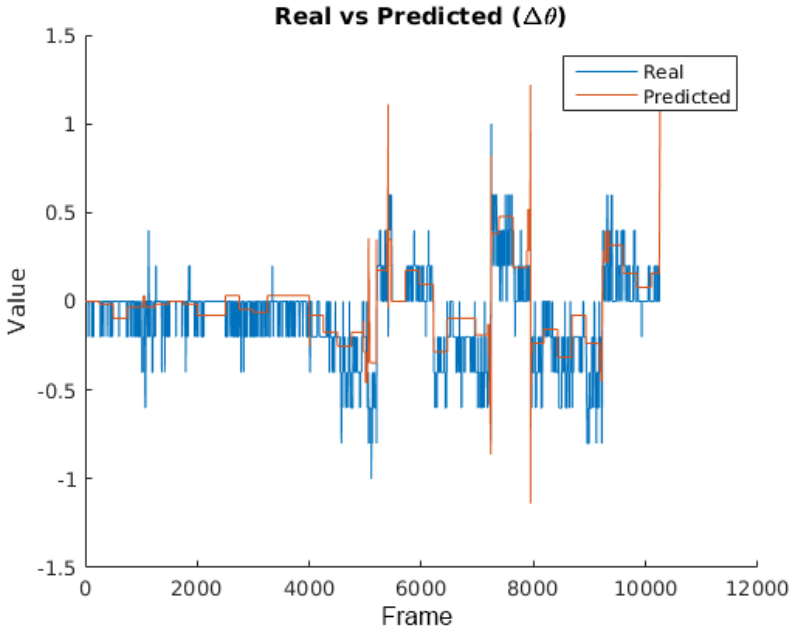

Figure 6.3: Comparison of real and predicted values on testing set for $\Delta\theta$ (Alexi V1)

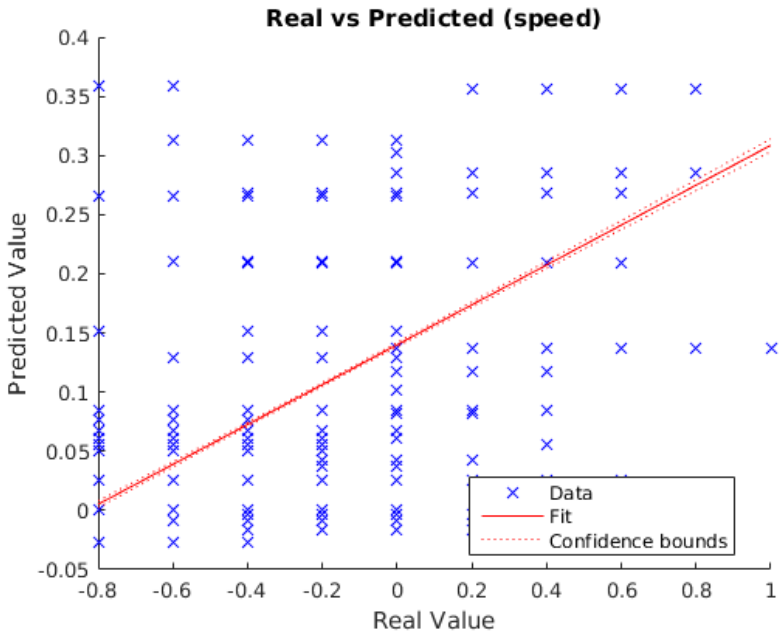

Figure 6.4: Scatter plot of real and predicted values on testing set for v (Alexi V1)

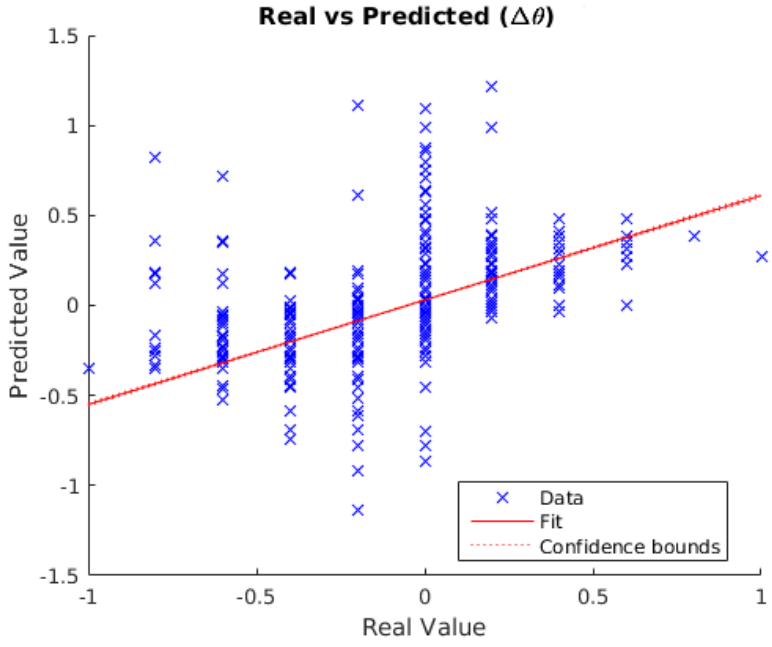

Figure 6.5: Scatter plot of real and predicted values on testing set for $\Delta\theta$ (Alexi V1)

In the Alexi V1 2D simulator the robot correctly responds to motor commands in

terms of speed and turn direction. We can show that the response of the robot in the 2D simulator is generally consistent with the response in Sandbox, by measuring the correlation between variables in the system. Table 6.2 gives correlation coefficients for two pairs of variables:

- wheel speed ($v_{left} - v_{right}$) and $\Delta\theta$—these should correlate as it is expected that positive or negative differences in left and right wheels speeds create a positive or negative change in yaw
- robot speed (v) and pitch (φ)—expected that speed should decrease as pitch increases

These coefficients are calculated using output values in the testing set, as well as the output predicted by the neural networks recorded from a session in the 2D simulation.

Table 6.2: Spearman correlation coefficients

Type	($v_{left} - v_{right}$) vs $\Delta\theta$	v vs φ
Simulated	0.9731	-0.0066
Testing	0.8371	0.1091

6.3 Alexi: Version 2

6.3.1 Network Architecture

Version 2 employs two separate feed-forward neural networks for robot kinematics, each of which are used to predict robot speed v, and change in robot yaw $\Delta\theta$ respectively. Like V1, each network is provided v_{left}, v_{right}, φ, and ψ as input. Our first experiments attempted to use three networks, for each of Δx, Δy, and $\Delta\theta$. This pattern is based on the networks used by Pretorius et al, where three networks were

also used for the same output variables. The results for Δx and Δy are given for comparison against the v prediction network that was eventually used in the Alexi simulator.

A feed-forward network is also used to train the terrain update model. The network is trained to predict changes in pixel intensity for the height-map terrain as the robot moves. The network is given v_{left}, v_{right}, φ, and ψ as input, as well as an array of values representing the change in pixel intensity of the previous frame, where each pixel maps to a single input node. The output of the network is the predicted changes in height (pixel intensity) at each point of the height-map, such that the number of output pixels is the same as the number of input pixels (*i.e.* a 1-to-1 map).

Each kinematic network has 13 hidden layers containing 7 nodes each, in addition to the input and output layers. All layers, including the input and output layer, make use of the hyperbolic tangent activation function *tanh*. Each layer adds an L2 regularization term (see Section 3.1) in the weight update step, where the regularization coefficient is 0.0001. Batch normalization is also applied for each layer. Network weights are initialized as uniform random values in the range $(-0.05, 0.05)$.

The terrain network has 20 hidden layers containing 500 nodes each, in addition to the input and output layers. This network is much larger than those used for kinematics, due the the much larger number of input and output nodes required. All layers use a linear activation function. The network layers apply the same regularization, batch normalization, and weight initialization parameters as those used in the kinematic networks.

6.3.2 Data Preprocessing

As in V1, Alexi version 2 applies several preprocessing stages to the raw kinematic data captured from Sandbox. These again include min-max normalization and quantization. A smoothing operation via convolution rather than exponential smoothing is also applied—an array of length 250 where each element is $\frac{1}{250}$ is convolved with each feature series. Frames containing outliers are discarded, where $\left|\Delta x - \Delta x_{\mu}\right| \geq 3\Delta x_{\sigma}$ or $\left|\Delta y - \Delta y_{\mu}\right| \geq 3\Delta y_{\sigma}$. Each operation is run successively, in the order of: outlier removal, min-max normalization, smoothing, and quantization.

Terrain data is generated from the height-map images captured by Sandbox for each frame in a session. Each image is first cropped in a 128×128 pixel window about the robots position on the height-map. This position is determined from the positioning data stored by Sandbox for the frame. The cropped images are then loaded sequentially and subtracted from the previous image, to store as a matrix of intensity differences. The image differences are flattened into 1D arrays when loaded again for training. Only those frames which were not excluded in any of the culling operations on kinematic data are processed.

6.3.3 Training

Each network (both kinematic and terrain) is trained using data from 3 sessions containing 64000 frames each, where each frame provides a training example to the network. The motion data was generated from a wheeled robot model about a deformable SCM terrain. A fourth session, also containing 64000 frames, is used as a testing set. Testing sets using two different terrains were generated, the height-maps of which are shown in Figure 6.6. Only terrain A is used to generate training data. For each session the same vehicle is used, however a different starting position is selected

in order to generate a different path from each other session.

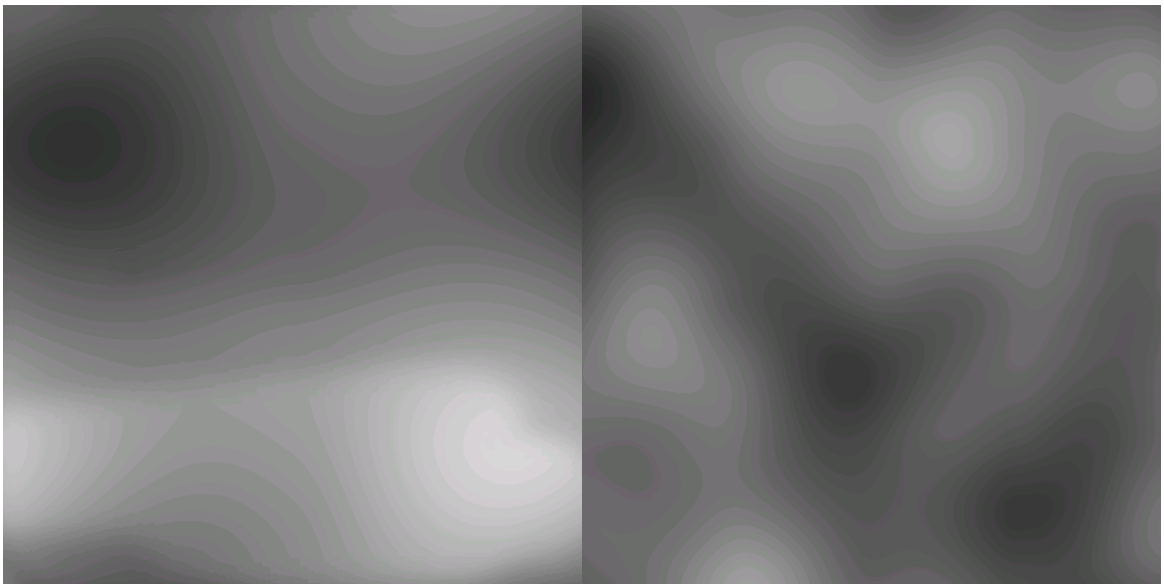

Figure 6.6: Terrain A (left) and Terrain B (right)

The kinematic networks are trained using the stochastic gradient descent (SGD) algorithm provided by Keras, with default parameters (learning rate of $\eta = 0.01$, momentum and decay of 0). Training examples are evaluated in batches of 32 frames. Training for each network is run for 25 epochs. Mean squared error is used as the loss function during training.

The terrain network is trained using the 'Adam' stochastic optimizer implemented by Keras [37], where the loss function is again the mean squared error. Training is run for 10 epochs, and training examples are passed to the network in batches of 32.

6.3.4 Results

Kinematic networks are evaluated quantitatively by the same metrics as used in V1: mean squared error and coefficient of determination.

Alexi V2 was tested using configurations of two networks as well as three. Figures 6.7, 6.8, 6.9, and 6.10 provide plots illustrating MSE and R^2 for the three-network

configuration, predicting Δx, Δy, and $\Delta \theta$. These networks were tested using only the testing set generated on terrain A.

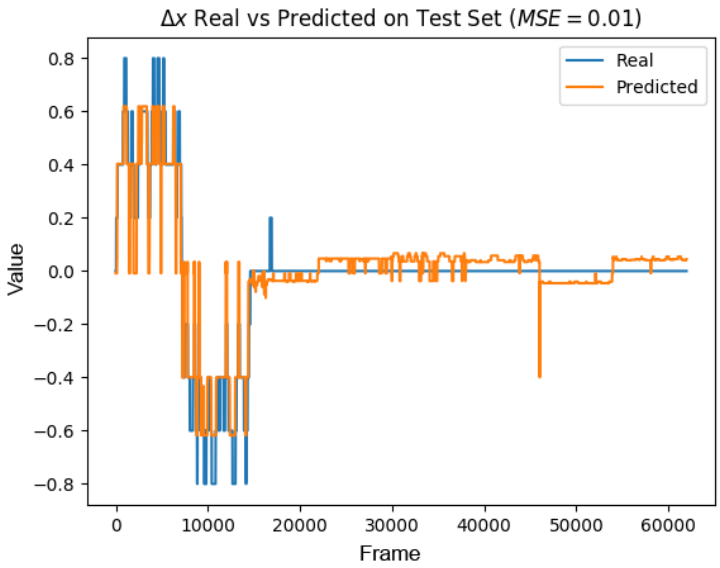

Figure 6.7: Comparison of real and predicted values on testing set A for Δx (Alexi V2)

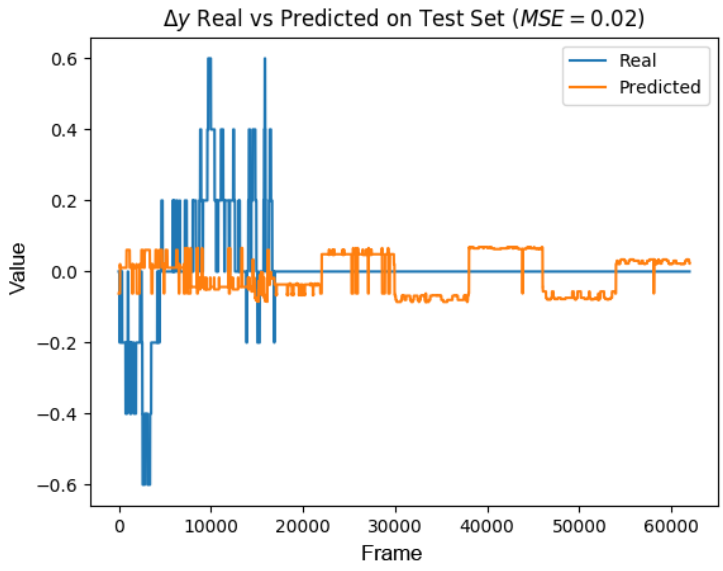

Figure 6.8: Comparison of real and predicted values on testing set A for Δy (Alexi V2)

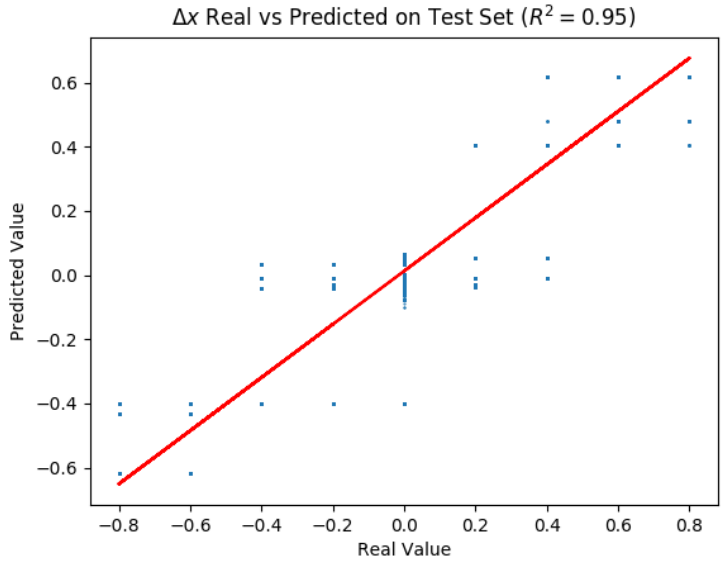

Figure 6.9: Linear regression plot of real and predicted values on testing set A for Δx (Alexi V2)

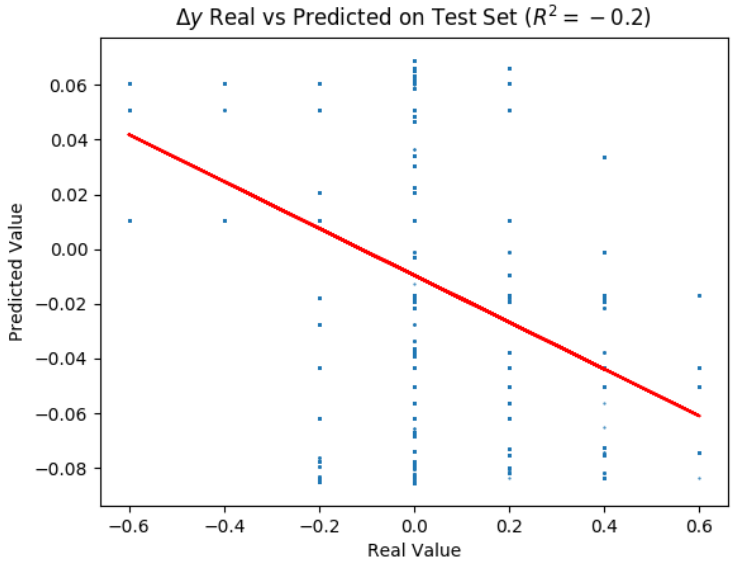

Figure 6.10: Linear regression plot of real and predicted values on testing set A for Δy (Alexi V2)

The two-network configuration was evaluated using data from both terrain A and terrain B. Terrain B is used to further demonstrate the ability of the networks to generalize, as the unseen data from terrain A still comes from the same terrain the networks were trained against. Figures 6.11, 6.12, 6.13, and 6.14 provide plots illustrating MSE and R^2 obtained on testing set A, using the two-network configuration predicting v and $\Delta\theta$. Figures 6.15, 6.16, 6.17, and 6.18 provide plots for testing set B, using the two-network configuration. MSE and R^2 values are recorded in Table 6.3.

Table 6.3: MSE and R^2 of each network on testing sets

	Terrain A		Terrain B	
Variable	MSE	R^2	MSE	R^2
v	0.0426	0.8828	0.0279	0.9254
$\Delta\theta$	0.0003	0.9657	0.0012	0.9842

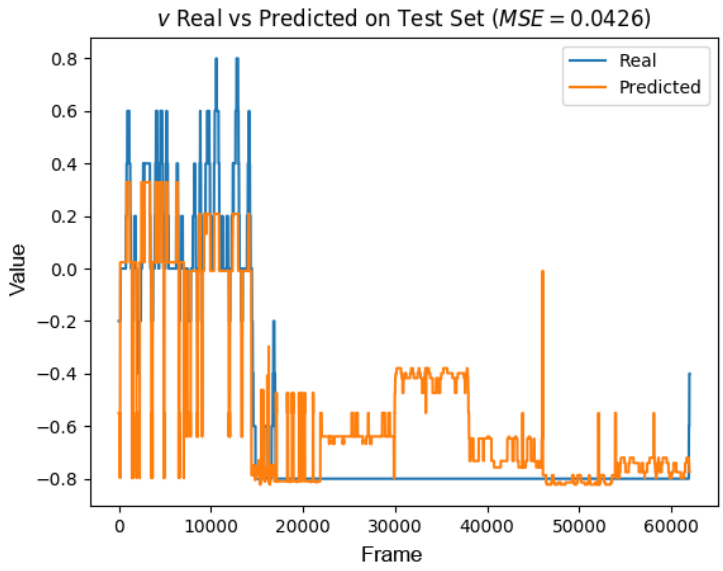

Figure 6.11: Comparison of real and predicted values on testing set A for v

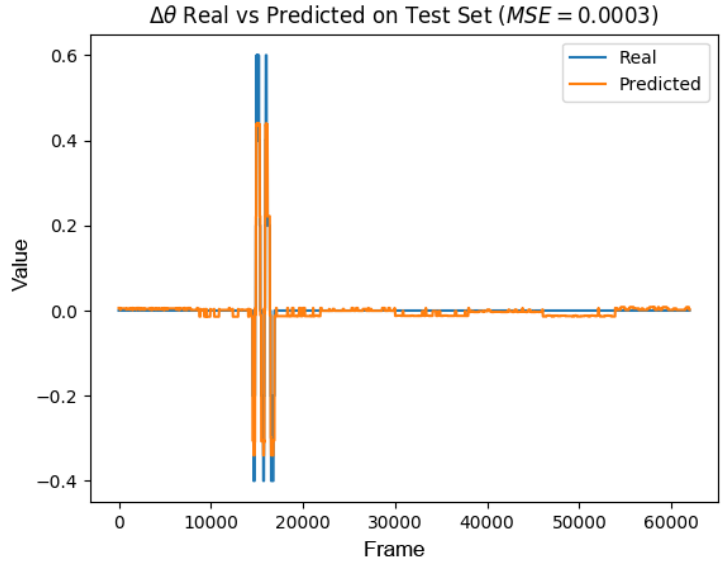

Figure 6.12: Comparison of real and predicted values on testing set A for $\Delta\theta$

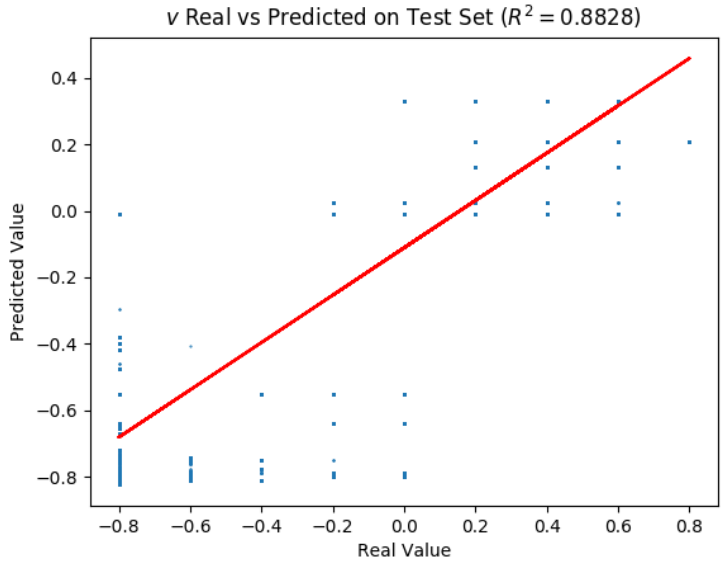

Figure 6.13: Linear regression plot of real and predicted values on testing set A for v

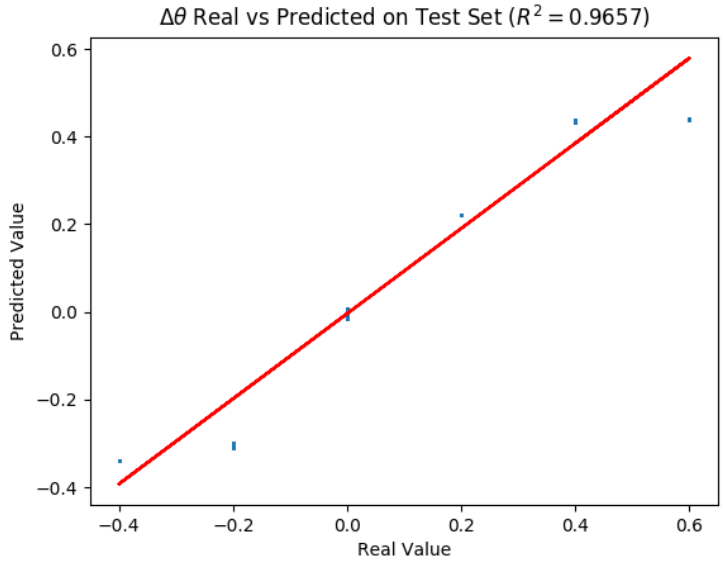

Figure 6.14: Linear regression plot of real and predicted values on testing set A for $\Delta\theta$

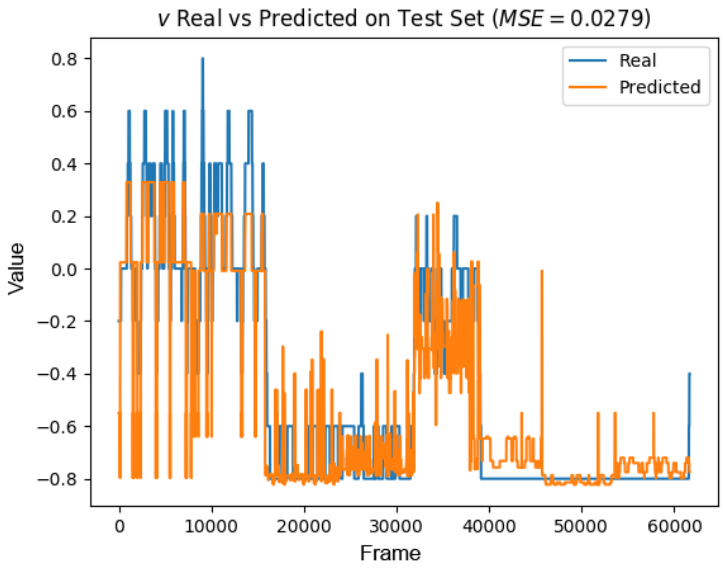

Figure 6.15: Comparison of real and predicted values on testing set B for v

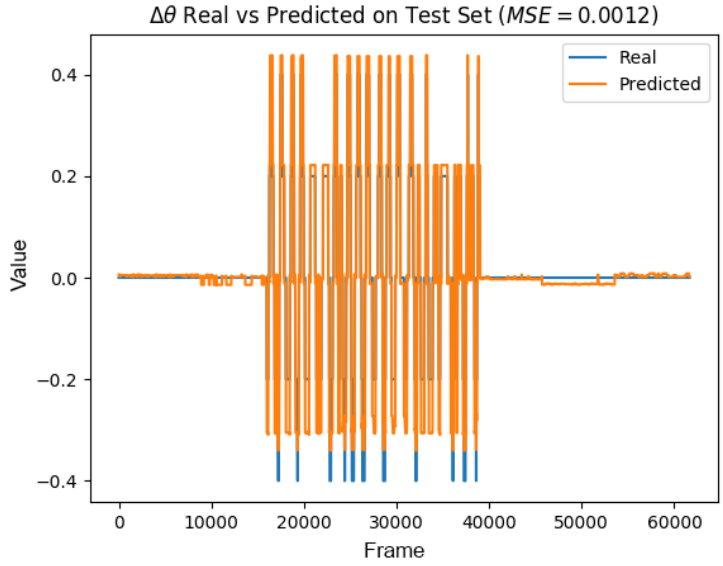

Figure 6.16: Comparison of real and predicted values on testing set B for $\Delta\theta$

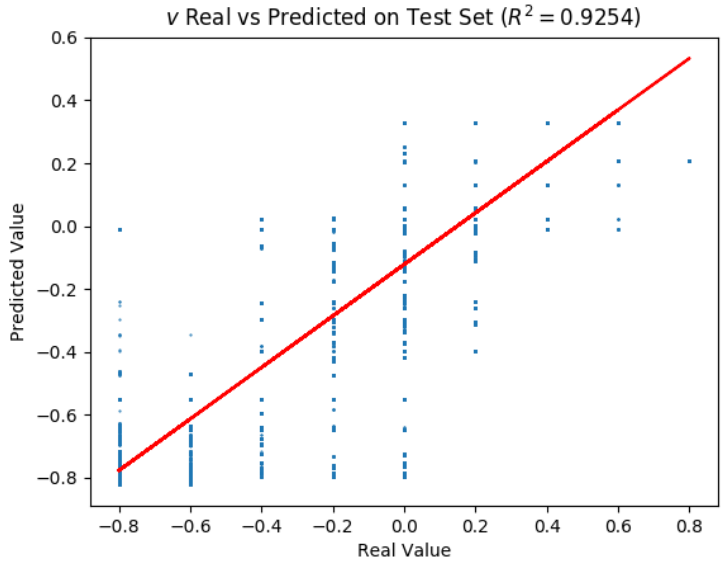

Figure 6.17: Linear regression plot of real and predicted values on testing set B for v

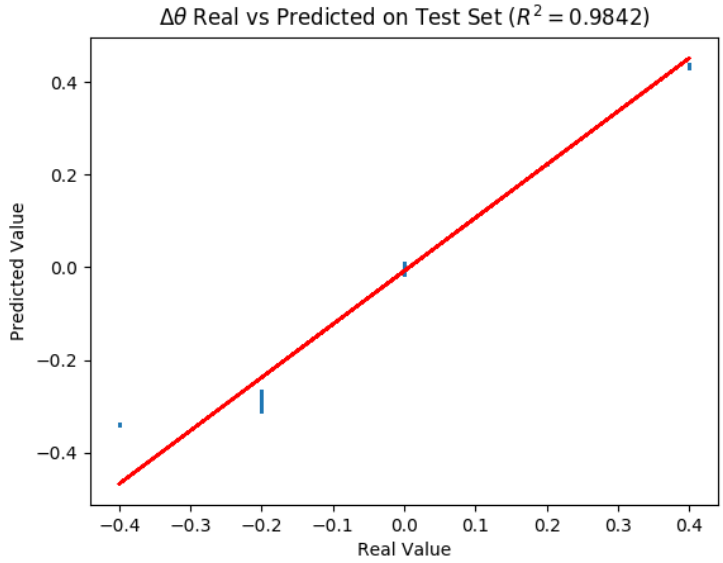

Figure 6.18: Linear regression plot of real and predicted values on testing set B for $\Delta\theta$

The terrain network, having 16384 output nodes, is somewhat more difficult to

assess in terms of the MSE and R^2 metrics. Given that the intent of the network is to provide an approximation of terrain deformation about the simulated robot, it is appropriate to observe the output of the network visually in the 2D simulator.

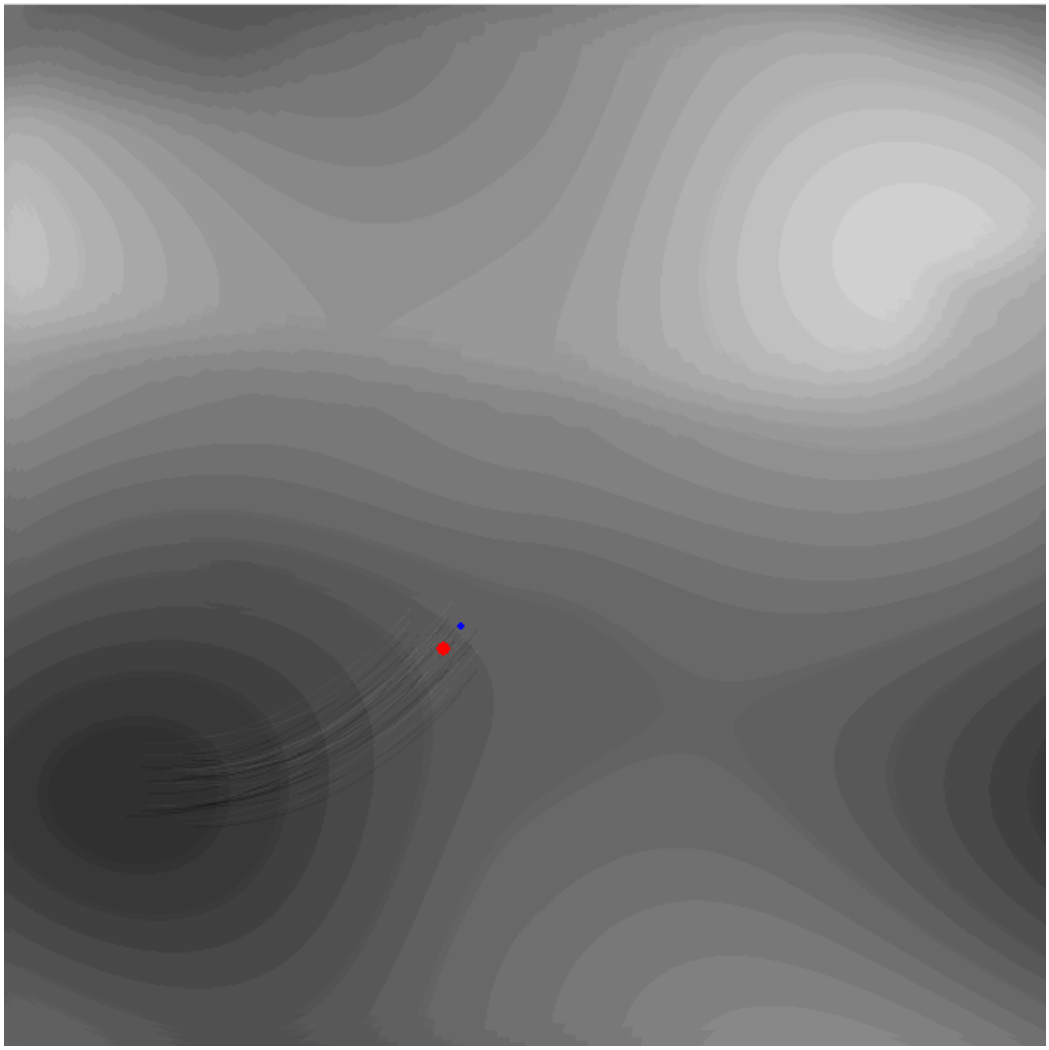

Figure 6.19: Screenshot of Alexi V2 demonstrating deformable terrain

As seen in Figure 6.19, the terrain appears to be deformed in along the same trajectory as the vehicle's movement. Several 'lines' across the width of the vehicle appear to form which appear somewhat consistent with the tracks seen in the training data generated by Sandbox. However, despite the the deformation having roughly the

right shape and orientation, there are clearly too many 'tracks' being predicted along the vehicle path. The terrain deformation shown may still be useful in in some level of algorithm prototyping, however more accurate training of the terrain prediction network is likely needed for it to be truly useful.

As in V1, the robot correctly responds to motor commands in terms of speed and turn direction and its trajectory also appears to be consistent with that which is observed in Sandbox.

6.4 Computational Performance

The 2D simulator of Alexi V1 performed considerably better than Sandbox in terms of computational speed. On average Sandbox took 620000μs per frame, versus an average of 285μs per frame in the 2D simulator, for an average speedup factor of approximately 2175. The performance was measured on a laptop PC with an Intel Core i7-4700HQ CPU, running Linux kernel 4.4.34. Each simulator was compiled using Intel C++ compiler 16.0.3.

The 2D simulator in V2 performs also performs better than Sandbox in terms of computational speed. On average Sandbox took approximately 150000μs per frame, versus an average of approximately 22500μs per frame in the 2D simulator, for an average speedup factor of approximately 9. For a terrain height-map doubled in size from that used in training, Sandbox took an average of approximately 470000μs per frame while the frame-time in the 2D simulator remained constant. This gives a speedup factor of approximately 20.9.

6.5 Discussion

This work represents one of the only known applications of neural networks to the problem of robot simulation. Because of this, performance of the trained networks are compared to the results obtained by Pretorius et al. All of the R^2 values for networks trained in the two-network of the V2 system are similar to those obtained by Pretorius et al, however similar results were not obtained using explicit networks for Δx and Δy in the three-network configuration. One possibility considered for this discrepancy is the lack of motion in the y-direction by the vehicle in the training data—when generating linear motion, the vehicle moves predominately along the x-axis due to the vehicle's initial orientation. This is due to programming issue within Sandbox that causes vehicle tracks to be initially generated axis-aligned rather than aligned to the vehicle frame. For wheeled vehicles this is not an issue, however for conistency all vehicles are restricted to the same starting orientation. An attempt was made to remedy this by swapping the x and y values for one of the sessions used in the training set, however it did not have any measurable effect on the training of the network.

Like Pretorius et al. it was found that the trained networks approximate the robot's motion with enough accuracy for the intended application. Despite the disparity between real and predicted values at many time-steps, the predicted values generally follow the overall trend seen in the real values. Absolute accuracy is not necessary to capture enough of the physical model represented by the original Sandbox simulation. Predicting only the speed of the robot rather than changes in position appears to work equally as well, while reducing the number of networks which need to be trained. Evaluation using testing data collected from multiple terrains shows that the network is able to generalize to unseen environments, in addition to unseen

data from the same environment.

The results obtained (in terms of the metrics given) are significantly better in V2 over V1. This is likely due to the added facilities provided by Keras to aid network training and prevent overfitting, such as batch normalization and weight regularization.

Visual results for terrain deformation suggest that while simulation via neural networks for this purpose is feasible, further work will be needed for better accuracy. The terrain predictor is currently modeled as a regression problem, along the same lines as the kinematic networks. However due to the extra complexity, a classification network which predicts discrete vehicle 'footprints' for various orientations may be more suitable.

It is worth noting that despite the promising results acheived in Alexi, the neural networks only capture the conditions of the terrain on which they were trained. While it has been shown that the networks can generalize across different terrain topologies, changes to the behaviour of the terrain itself (*i.e.* greater friction or looser soil) will require re-training the networks. While outside of the scope of this work, a more robust training set would include data from both many topologies and many terrain parameters.

6.6 Summary

We have presented in this chapter a novel robot simulator called Alexi, which is based on neural networks rather than a traditional physics engine such as the one used in Sandbox. Two versions of the simulator have been developed, with the second providing far superior results in terms of network generalization. In the second version, two network configurations were tested; one in which three networks were utilized, and

another in which two were utilized. Both versions demonstrate considerable speed-up over the Sandbox simulation with respect to computational performance, which was a primary objective of this research. An early experiment in simulating terrain deformation using a neural network was conducted; this will require further work to be useful in any meaningful way, however the work presented here represents a valuable first step.

Chapter 7

Conclusions

This thesis has presented a novel set of simulation software, for the purpose of simulating a robot on an uneven, deformable terrain. Taken together, the simulators discussed (Sandbox and Alexi) represent a software pipeline. This pipeline can be used to model a robot in a high-fidelity simulated environment (Sandbox), and capture the physics of both the robot's movement as well as the deformation of the terrain using a neural network, which in turn drives a much less computationally demanding simulation (Alexi). Once terrain prediction is improved further, the simulators should allow rapid prototyping of a terrain leveling system. In its current state, Alexi represents the first known implementation of a simulator which allows the modeling of an uneven terrain without relying on a 3D physics engine. The ability to model a vehicle with 3D orientation within an efficient 2D environment is novel, and offers advantages in terms of both performance and realism over other methods. Scripted movement for example would be much faster but lack physical modeling, while a traditional 2D physics solver lacks the ability to incorporate 3D orientation information. Computational performance results of the Alexi simulator show that the neural network-based

simulation is significantly faster than its more traditional counterpart, while still providing accurate robot motion. Results related to terrain deformation using a neural network represent an early step towards this goal, but show that the idea is viable.

7.1 Future Work

This section discusses several concepts that, while outside of the scope of this thesis, may be worthwhile to explore as an extension of the work described here. Section 7.1.1 discusses possible algorithms which may be applied to the terrain leveling problem, and tested using the Sandbox and Alexi simulation environments. Section 7.1.2 discusses possible enhancments which could be made with respect to neural network-based simulation.

7.1.1 New Approaches to Leveling

The algorithms described in Section 4.2 are dependent on the presence of a mechanism for gathering material. While the simulation carried out for evaluating those algorithms didn't consider it, the malfunction of such a mechanism would effectively disable the agent. Many of the rules that have been defined for the behaviour of these agents exist only to govern the pick-up/deposit states. Systems which assume that the agents have some form of actuation are typically more complex mechanically, which creates more opportunity for error. To reduce the system to a minimal set of hardware and software, a method which does not rely on actuated earth-moving components should be explored.

Assuming the site to be leveled is composed of granular material as described in Section 4.1, it should be possible to level the terrain using only the motion of the agents themselves. Without any method of explicitly gathering material, the effiency

of the leveling operation will depend on the pattern of movement taken by the agent. Like Algorithm 1, it is conjectured that the most efficient method should focus on moving material from peaks to valleys in the terrain. Rather than explicitly picking up material at a peak, a passive algorithm should instead drive each agent to seek out peaks and maneuver about them in such a way that material is disloged. This material will gradually be redistributed to the valleys as the agents continue moving.

A passive algorithm for leveling may still proceed by starting each agent on a random walk, as initially no agent should have any record of the terrain configuration. Since the system relies on the movement of the agent to dislodge the terrain, it is not possible to make use of the rule pick-up/deposit above and below the average terrain height. Futhermore, a land-based vehicle may not have the capacity to accurately measure it's altitude relative to the terrain. One possible approach simply makes use of a magnetometer to measure the 3D orientation of the agent. As the vehicle moves, it should check if its pitch angle crosses a particular threshold ϵ—if it does, we can then increase the speed of the vehicle and proceed in a straight line. The random walk may be resumed as the pitch crosses back below the threshold. This 'charging' action, if executed repeatedly, should allow material forming a peak in the terrain to

be eroded.

Loop

- **for** *agent* **do**
  - $pitch \leftarrow$ get_pitch()
  - **if** $pitch > \epsilon$ **then**
    - set_linear_velocity(v_{climb})
    - set_angular_velocity(0)
  - **else if** $pitch \leq \epsilon$ **then**
    - set_linear_velocity($rand()$)
    - set_angular_velocity($rand()$)
- **end**

Forever

Algorithm 6: Hill-sensing algorithm relying on magnetometer

Algorithm 6 has a minimal set of rules and requires very simple hardware—only a means for the robot to move and sense its orientation. However, given the limited amount of time the agent is likely to spend at a peak using this method, it may not be particularly efficient. A better approach may be to perform some maneuver at the peak, such as spinning the robot in place for some period of time. This is likely to remove more material than merely driving over the peak, however the algorithm then requires some method of detecting a peak rather than an incline. An approach using the existing hardware from Algorithm 6 could continue on a straight path when an incline is detected, but instead begin a 'spinning' maneuver once a decline is detected.

This method should improve on Algorithm 6 with respect to rate of material redistribution, however it is not without its own issues. Relying only on a magnetometer to detect agent orientation, which typically posess some level of internal error, means

that the agent may detect a peak where there is none. It may be that the detected decrease in pitch is due to error, or that the agent has instead sensed a local maximum rather than the global maximum. The algorithm also still relies on a random-walk in order to find peaks, and there is no notion of memory for peaks that have been encountered. The agents in [19] were assumed to have some means to sense the height of the terrain, presumably by sonar or another range-finding method given that they operated underwater. For a land-based robot, elevation could be detected in other ways. One possibility would be to use a altimeter, keeping track of changes in elevation from the starting position. It would also be possible to use odometry (assuming the agent posesses wheel encoders), since the agent is also aware of its orientation. Using either of these methods, combined with odometry to allow the agent to determine its position relative to its starting location, a rough map of peak locations could be recorded in the agents memory. This would then allow peaks to be sought out explicitly rather than randomly, as well as allow the agents to distinguish between local and global maxima. However, odometry over uneven granular material is not a trivial task and may not be possible using a minimal set of hardware and software.

7.1.2 Enhancements to Neural Network-Based Simulation

Our work towards neural network-based simulation has largely incorporated elements of work by Pretorious et al. as a starting point. We believe however that using this work as a basis, several enhancements could be made with respect to neural network-based simulation that may increase accuracy and generalization.

As mentioned in Section 6.5, the current system requires that the networks be re-trained if the terrain parameters (such as friction) are changed. It should be possible to remove this limitation by training the networks on a wide variety of terrain conditions, and providing terrain parameters as input to the network. It should be

noted however that this would likely require significantly more training data than is currently necessary.

The current system predicts only kinematic values v and $\Delta\theta$. It may also be possible to make explicit use of kinetic data such as inertia. However it is also possible that kinetics of the system are captured implicitly by the neural networks. Additional experiments should be carried out to determine whether making explicit use of kinetics has any measurable effect on accuracy.

Bibliography

[1] Arduino. http://www.arduino.cc. Accessed: 2017-04-30.

[2] Bullet physics. bulletphysics.org. Accessed: 2017-04-30.

[3] Irrlicht engine–a free open source 3d engine. irrlicht.sourceforge.net. Accessed: 2017-04-30.

[4] Keras: Deep learning library for theano and tensorflow. keras.io. Accessed: 2017-03-16.

[5] MARS: An open-source, flexible 3d physical simulation framework. http://rock-simulation.github.io/mars/. Accessed: 2017-03-16.

[6] OGRE—open source 3d graphics engine. ogre3d.org. Accessed: 2017-04-30.

[7] Open dynamics engine. ode.org. Accessed: 2017-04-30.

[8] Phobos. https://github.com/rock-simulation/phobos. Accessed: 2017-03-16.

[9] Swarmanoid: Towards humanoid robotic swarms. http://www.swarmanoid.org/. Accessed: 2017-04-30.

[10] URDF—ROS wiki. wiki.ros.org/urdf. Accessed: 2017-02-19.

[11] Vortex studio. https://www.cm-labs.com/vortex-studio/. Accessed: 2017-04-30.

[12] T. Akenine-Möller, E. Haines, and N. Hoffman. *Real-time rendering*. CRC Press, 2008.

[13] H. Ardiny, S. Witwicki, and F. Mondada. Construction automation with autonomous mobile robots: A review. In *Robotics and Mechatronics (ICROM), 2015 3rd RSI International Conference on*, pages 418–424. IEEE, 2015.

[14] N. Bell, Y. Yu, and P. J. Mucha. Particle-based simulation of granular materials. In *Proceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 77–86. ACM, 2005.

[15] H. Braun and M. Riedmiller. RPROP: a fast adaptive learning algorithm. In *Proceedings of the International Symposium on Computer and Information Science VII*, 1992.

[16] R. A. Brooks. Intelligence without representation. *Artificial intelligence*, 47(1):139–159, 1991.

[17] R. A. Brooks, P. Maes, M. J. Mataric, and G. More. Lunar base construction robots. In *Intelligent Robots and Systems' 90.'Towards a New Frontier of Applications', Proceedings. IROS'90. IEEE International Workshop on*, pages 389–392. IEEE, 1990.

[18] P. Castillo-Pizarro, T. V. Arredondo, and M. Torres-Torriti. Introductory survey to open-source mobile robot simulation software. In *Robotics Symposium and Intelligent Robotic Meeting (LARS), 2010 Latin American*, pages 150–155. IEEE, 2010.

[19] D. Cook and A. Vardy. Terrain leveling by a swarm of simple agents. In *Proceedings of the 8th International Conference on Bioinspired Information and Communications Technologies*, pages 55–58. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2014.

[20] D. Cook and A. Vardy. Simulation of a mobile robot on uneven terrain using neural networks. *Expert Systems: Journal of Knowledge Engineering*, 2017. Submitted.

[21] D. Cook and A. Vardy. Towards real-time robot simulation on uneven terrain using neural networks. In *Neural Networks (IJCNN), 2017 International Joint Conference on*, pages 1688–1695. IEEE, 2017.

[22] D. Cook, A. Vardy, and R. Lewis. A survey of AUV and robot simulators for multi-vehicle operations. In *Autonomous Underwater Vehicles (AUV), 2014 IEEE/OES*, pages 1–8. IEEE, 2014.

[23] J. Craighead, J. Burke, and R. Murphy. Using the unity game engine to develop sarge: a case study. *Computer*, 4552:366–372, 2007.

[24] J. Craighead, R. Murphy, J. Burke, and B. Goldiez. A survey of commercial & open source unmanned vehicle simulators. In *Proceedings 2007 IEEE International Conference on Robotics and Automation*, pages 852–857. IEEE, 2007.

[25] S. De, D. Deo, G. Sankaranarayanan, and V. S. Arikatla. A physics-driven neural networks-based simulation system (PhyNNeSS) for multimodal interactive virtual environments involving nonlinear deformable objects. *Presence*, 20(4):289–308, 2011.

[26] J. L. Deneubourg, S. Goss, N. Franks, A. Sendova-Franks, C. Detrain, and L. Chrétien. The dynamics of collective sorting robot-like ants and ant-like robots. In *First Int. Conf. on the Simulation of Adaptive Behaviour*, pages 356–363, Cambridge, MA, 1990. MIT Press.

[27] T. Fong, M. Allan, X. Bouyssounouse, M. G. Bualat, M. Deans, L. Edwards, L. Fluckiger, L. Keely, S. Lee, D. Lees, et al. Robotic site survey at haughton crater. In *Proceedings of the 9th International Symposium on Artificial Intelligence, Robotics and Automation in Space (iSAIRAS), Los Angeles, CA*, 2008.

[28] N. Francingues, D. Thompson, E. C. McNair, and L. Saenz. The silt wing excavator–an innovative sediment leveling and grading device. In *Dredging'02: Key Technologies for Global Prosperity*, pages 1–14. 2003.

[29] C. Gerwick. *Construction of marine and offshore structures*. CRC press, 2002.

[30] X. Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Aistats*, volume 9, pages 249–256, 2010.

[31] R. Grzeszczuk, D. Terzopoulos, and G. Hinton. Neuroanimator: Fast neural network emulation and control of physics-based models. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 9–20. ACM, 1998.

[32] U. H. Hernandez-Belmonte, V. Ayala-Ramirez, and R. E. Sanchez-Yanez. A mobile robot simulator using a game development engine. In *Proceedings of the ROboticS SUmmer Meeting ROSSUM 2011*, 2011.

[33] D. Holz. Parallel Particles (P2): A parallel position based approach for fast and stable simulation of granular materials. 2014.

[34] D. Holz, A. Azimi, and M. Teichmann. Advances in physically-based modeling of deformable soil for real-time operator training simulators. In *Virtual Reality and Visualization (ICVRV), 2015 International Conference on*, pages 166–172. IEEE, 2015.

[35] T. Huntsberger, H. Aghazarian, E. Baumgartner, and P. S. Schenker. Behavior-based control systems for planetary autonomous robot outposts. In *Aerospace Conference Proceedings, 2000 IEEE*, volume 7, pages 679–686. IEEE, 2000.

[36] C. Igel and M. Hüsken. Improving the RPROP learning algorithm. In *Proceedings of the second international ICSC symposium on neural computation (NC 2000)*, volume 2000, pages 115–121. Citeseer, 2000.

[37] D. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[38] N. Koenig and A. Howard. Design and use paradigms for gazebo, an open-source multi-robot simulator. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2149–2154, Sendai, Japan, Sep 2004.

[39] R. Krenn and G. Hirzinger. SCM–a soil contact model for multi-body system simulations. 2009.

[40] S. Laughery, G. Gerhart, and R. Goetz. Bekker's terramechanics model for off-road vehicle research. Technical report, DTIC Document, 1990.

[41] H. Mazhar, T. Heyn, A. Pazouki, D. Melanz, A. Seidl, A. Bartholomew, A. Tasora, and D. Negrut. Chrono: a parallel multi-physics library for rigid-body, flexible-body, and fluid dynamics. *Mech. Sci*, 4(1):49–64, 2013.

[42] P. Moscato et al. On evolution, search, optimization, genetic algorithms and martial arts: Towards memetic algorithms. *Caltech concurrent computation program, C3P Report*, 826:1989, 1989.

[43] N. Napp and R. Nagpal. Distributed amorphous ramp construction in unstructured environments. *Robotica*, 32(02):279–290, 2014.

[44] S. Nissen. Implementation of a fast artificial neural network library (fann). *Report, Department of Computer Science University of Copenhagen (DIKU)*, 31:29, 2003.

[45] G. of Canada. Canada: Reports of hurricanes, tropical storms, tropical disturbances and related flooding during 2010. Technical report, World Meteorological Organization, 2011.

[46] L. E. Parker, Y. Guo, and D. Jung. Cooperative robot teams applied to the site preparation task. In *Proceedings of 10th International Conference on Advanced Robotics*, pages 71–77, 2001.

[47] C. Pinciroli, V. Trianni, R. OGrady, G. Pini, A. Brutschy, M. Brambilla, N. Mathews, E. Ferrante, G. Di Caro, F. Ducatelle, et al. ARGoS: a modular, parallel, multi-engine simulator for multi-robot systems. *Swarm intelligence*, 6(4):271–295, 2012.

[48] C. J. Pretorius, M. C. du Plessis, and C. Cilliers. Simulating robots without conventional physics: A neural network approach. *Journal of Intelligent & Robotic Systems*, 71(3-4):319–348, 2013.

[49] C. J. Pretorius, M. C. du Plessis, and C. B. Cilliers. Towards an artificial neural network-based simulator for behavioural evolution in evolutionary robotics. In *Proceedings of the 2009 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists*, pages 170–178. ACM, 2009.

[50] E. Rohmer, S. P. Singh, and M. Freese. V-REP: A versatile and scalable robot simulation framework. In *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1321–1326. IEEE, 2013.

[51] G. Theraulaz and E. Bonabeau. Coordination in distributed building. *Science*, 269(5224):686, 1995.

[52] S. Tisue and U. Wilensky. Netlogo: A simple environment for modeling complexity. In *International conference on complex systems*, volume 21, pages 16–21. Boston, MA, 2004.

[53] A. Vardy, G. Vorobyev, and W. Banzhaf. Cache consensus: Rapid object sorting by a robotic swarm. *Swarm Intelligence*, 8(1):61–87, 2014.

[54] R. Vaughan. Massively multi-robot simulation in stage. *Swarm Intelligence*, 2(2-4):189–208, 2008.

[55] J. Wawerla, G. S. Sukhatme, and M. J. Mataric. Collective construction with multiple robots. In *Intelligent Robots and Systems, 2002. IEEE/RSJ International Conference on*, volume 3, pages 2696–2701. IEEE, 2002.

[56] J. Werfel, K. Petersen, and R. Nagpal. Designing collective behavior in a termite-inspired robot construction team. *Science*, 343(6172):754–758, 2014.

[57] G. W. Woodford, M. C. Du Plessis, and C. J. Pretorius. Evolving snake robot controllers using artificial neural networks as an alternative to a physics-based simulator. In *Computational Intelligence, 2015 IEEE Symposium Series on*, pages 267–274. IEEE, 2015.