AntBot:
A biologically inspired approach to Path Integration

Luca Scimeca

Project supervisor: Barbara Webb

A paper presented for the Bachelor of Engineering in
Artificial Intelligence and Software Engineering

School of Informatics
University of Edinburgh
United Kingdom
April 17, 2017

Acknowledgements

I want to give my thanks to my supervisor, Barbara Webb, for giving me the opportunity to work on this project, and for supporting me throughout its course, giving me invaluable suggestions without which this would have been impossible. I want to thank Aleksandar Kodzhabashev for helping me get familiar with the AntBot and the Android applications running on it when the project was still at its dawn, as well as, Leonard M. Eberding and Canicius Mwitta, which have worked on the AntBot in previous years, and provided the base system on which I integrated the devised algorithms for this project. A final special thank you goes to Tom Stone, which has devised many of the algorithms here presented, and whose code was an invaluable reference throughout the building of the models.

Declaration

This is to certify that the work contained within has been composed by me and is entirely my own work. No part of this thesis, unless otherwise specified, has been submitted for any other degree or professional qualification.

Signed:

Luca Scimeca

Contents

Abstract

We introduce the problem of *homing* for animals with a fixed nest and present *Path Integration* as a solution to the homing problem. Path Integration (PI) is a mechanism by which an animal (or robot) is capable of maintaining a reference to its starting position (*home*) as it moves arbitrary outwards, and later use said reference to return home. Any application of PI involves a continuous sum of distance and orientation information to be able to maintain an *home vector* reference. After describing a simple PI implementation, we present a neural model for Path Integration in *bees* and *ants*. The model, recently proposed in *Stone et al* [1], encodes path integration into a multilayer network, which resembles the neural connectivity and behavior of observed selected neurons in the *bee* brain (Central Complex model). We build two versions of the model: one, taking orientation information as well as the *action* speed of the robot; the other, using the same orientation information but retrieving speeds from Optic Flow. We implement the models on the AntBot, a smart-phone driven robotic platform built to test models of the insect brain. We build the models in Java, using as a starting point the python simulator build in *Stone at al*. We use a combination of methods in the *ejml* Java library, *OpenCv* and custom made matrix operation to model the cells in the network. We add/modify *Arduino* modules to allow the AntBot to move in a *continuous* fashion, as well as modify the higher level *Serial* Android applications for the smart-phone to use the new type of motion. We build a compass from the smart-phone's *GAME_ROTATION_VECTOR*. We process consecutive 360° (low resolution) frames to find flow patterns as the robot moves, and use the *matched filter* model by *Franz & Krapp* [2] to transform the retrieved flow into left/right speed estimates of the robot. After incorporating the speed and orientation information in the models, we test them by running the AntBot outwards for $\approx 45sec$, reaching a variable distance of $600cm$ to $1200cm$ from its starting position, and then driving *home* for an equal time. After testing, we find the CX models performing similarly to the standard PI implementation, always driving the robot within *approx 60cm* from its starting position, thus providing direct evidence of the ability of the model to work in a real-world robot. Moreover, although noisy, we find Optic Flow unaffected by hardware bias (like *action* information normally would). Finally, we discuss some limitations of the platform to fully test the proposed mechanism and the possibility of future work in the project by merging the CX PI implementation with a visual based homing mechanism [3, 4].

1 Introduction

Robots imitating animal behavior have been built since the dawn of Robotics itself [5, 6], biologically inspired Robotics is therefore far from being a new concept. The relation of Biology to Robotics is usually evident, as it is quite simple to see how many of the problems that scientists try to solve in Robotics, from legged motion to gripping mechanisms and morphology exploitation, have already been solved by Nature in beautiful and efficient ways. The relation of Robotics to Biology, on the other hand, is somewhat harder to appreciate, as it is not clear what Biology could gain from Robotics models. Although harder to perceive, the relation is indeed there, and Robotics has been and can be used to test biological hypothesis. The assumptions to be made when following this approach are many, and the extent to which the testing of a robot implementing a theorized biological model can be used as empirical evidence for said hypothesis is limited. Nonetheless, the approach can give invaluable insights into the understanding of biological systems just as it does into the understanding of the engineering side of the problem [7].

In this paper we try to combine the two-way relation between Robotics and Biology, testing biological models on a real-world robot to gather evidence in support or against said theoretical models, and to gain insights on the Robotics side on the picture. The models implemented are

largely related to *Path Integration* (PI) and *homing* for insects, more specifically to that of *ants* and *bees*.

The work is carried out on AntBot, a biologically inspired, smart-phone driven robot platform which can be used to implement and test biological models of the insect brain.

We structure the paper as follows: in Section 2 (*Background*) we give an overview of the origin of the research problems we try to tackle in this and related research, and an introduction to the biological side of the picture. The focus of this research is on Path Integration (PI). We therefore briefly introduce navigation for animals, emphasizing the importance of Path Integration in the context of navigation, after which we introduce PI more in depth (Section 2.3), and present a biologically inspired model for Path Integration (Section 4.3). In Section 3 (*AntBot*) we move on to present the platform in use for the project, both from an hardware and software perspective, and we describe the relevant and novel modification needed for the completion of the project. In Section 4 (*Methods*) we describe the implementation of the models and the retrieval of fundamental sensory inputs to the models from the robot's surroundings. We pass on to testing the implementation of the models and report the results in Section 5 (*Testing*) and Section 6 (*Results*). We summarize, and discuss the research and the results in the framework of the project in Section 7 (*Discussion and Conclusion*); after which we discuss interesting forks of the current project into future research (Section 8: *Future work*).

1.1 Contributions

The work done here builds on previous projects, more specifically previous work done by Leonard Matthias Eberding [8] and Canicius Mwitta [9]. The following is a list of the most significant personal contributions to the project:

1. Created custom (anew from basic operations) Matrix libraries in Java (e.g. Matrix multiplication, dot products, Fourier Transform of vectors etc.)
2. Implemented custom Java testing environment for existing neural models.
3. Implemented new action system in Arduino to allow the AntBot to move in a continuous fashion (new Arduino parser, executioner and communication functions – Section 3.2.1)
4. Implemented Android protocols for new platform motion (*SerialCommunicationApp*, broadcasting mechanism functions etc.)
5. Implemented magnetic field independent compass from smart-phone sensors (Section 4.1).
6. Implemented neural models in Android and Java (Basic version & Holonomic), integrated models in the AntBot Android existing system, and implemented methods and protocols to interact with the models (update states, decay memory, wrap cells etc. – Section 4.3, Section 4.4).
7. Modified flow pre-processing steps and implemented *dense* and *sparse* Optic Flow algorithms anew (Section 4.5).
8. Implemented *matched filter model* in Java and applied it to the Optic Flow algorithms for speed retrieval. Furthermore, devised and implemented new type of filter for flow matching.

Additionally to the implementation, a considerable amount of time was spent doing literature review on the relevant papers, the understanding of the biology behind the proposed models, the study of the models themselves together with the understanding of the problem both from a biological and robotics perspective, and finally, the testing of the models and the results reported.

2 Background

2.1 Overview

Some insects, such as ants or bees, have been shown to be able to learn and follow long foraging routes, after which they are capable of returning to their nest. Such insects have been the subject of interest of many navigational models as they are concrete proof of the possibility of achieving complex navigational abilities with limited brain power[10, 11, 12].

Amidst the different species that display such homing capabilities, the desert ant belonging to the genus *Cataglyphis* presents some of the most impressive characteristics. Foraging in an environment lacking visual diversity and unable to use pheromone trails [13], desert ants have been shown to rely on other strategies for homing, most preeminently: Path Integration, and vision based landmark guidance.

To put together these strategies, and incorporate the biological and behavioral evidence of the sensory information available to the insect, we briefly present a particular model: *The architecture of the desert ant's navigational toolkit*, by Rüdiger Wehner [12]. We present the model for two reasons: one, to put Path Integration in the context of *Navigation* for animals, thus clarifying why we are interested in PI to begin with; two, to present concepts from related research, relevant to navigation and PI.

2.2 The Ant Navigational Toolkit

The Ant Navigational Toolkit model by Wehner presents the theory by which ants perform homing through a combination of Path Integration and visual based landmark navigation.

Path integration is a mechanism by which animals are capable of maintaining a sense of the distance and direction traveled since leaving the nest (see Section 2.3 for more details). At any given point in time, then, an ant such a *Cataglyphis* would possess some kind of vector equivalent representation of the home position. By comparing their current heading with that of the stored vector, they can perform homing by simply adjusting their heading to align with that of the stored vector, and follow it back to their starting position. To be able to perform such a feat, ants need to retrieve two fundamental pieces of information from their runs: displacement (distance traveled) and orientation (compass information).

The displacement information can be retrieved by a combination of a pedometer (i.e. stride integrator) and possibly the integration of some Optic Flow cues [14, 15]. In bees, distances have been shown to depend mainly on visual cues and are therefore dependent on the environment in which the animal moves, however, these have also been shown to be robust against many optic variations [16]. The orientation is mainly believed to be retrieved from a polarization compass, i.e. a direction is extrapolated from the polarization pattern observed in the sky [17], with the possibility of using the sun position for 180° disambiguations.

Landmark guidance refers to the ability of the animals to store landmark based memories and successively retrieve and use these memories to perform navigational tasks. In the past, several attempts have been made to try and model the landmark-based navigation ability of ants and bees [18, 19, 20]. After decades of experiments, to a greater or lesser extent, all seem to agree

that the landmark based guidance has so far only been shown to give local views, i.e. they do not give global metric information (a *map*) to the animal, but rather local, possibly action associative, information [21].

The two described mechanisms can indeed account for much of the observed behavior in ants and bees. A possible integration of the two can see the path integrator active during all foraging trips, to allow the animal to return back to the nest even when out discovering new routes. The landmark based navigation, instead, would be more or less active depending on the familiarity of the route to the animal. Here, for example, known/pre-traced routes would give more opportunities to the ant to save landmark views and use them in future runs to improve navigation [12].

The independent *routines* the animal is known to possess together with the animal's ability to actuate them concurrently or intermittently constitute the essence of the *Ant Navigational Toolkit Model*. Jointly, these two factors can account for the observed (complex) animal's behavior, without the need of feeding all the information retrieved by the single routines, to a central processing unit (see Figure 1).

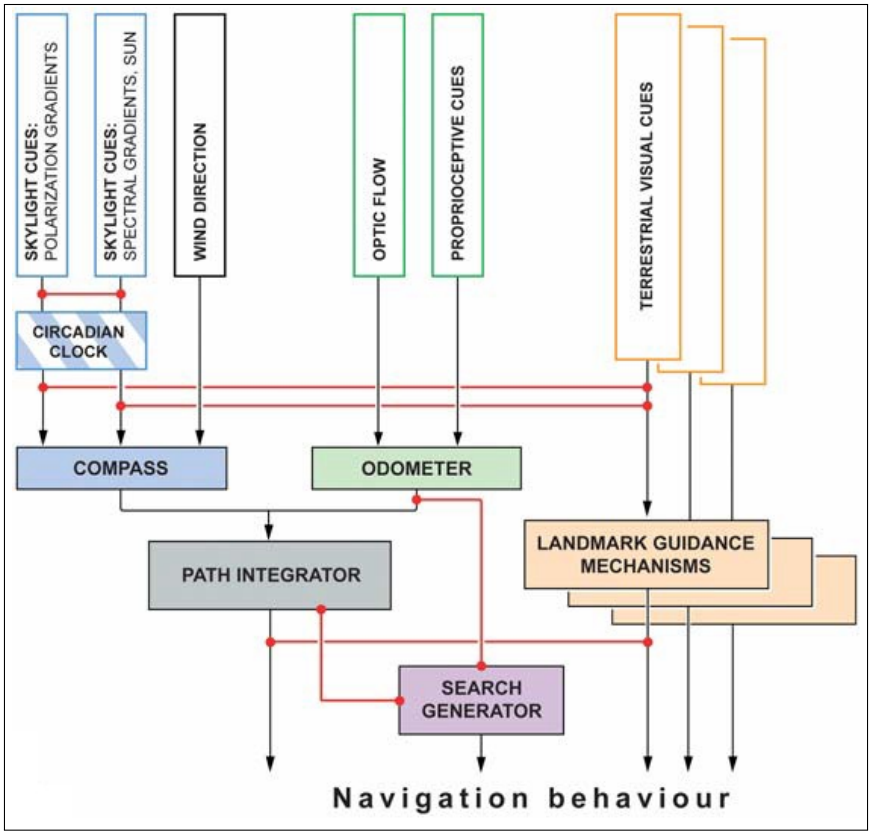


Figure 1: The Figure shows the Ant Navigational Toolkit model as designed by Wehner [12]. The Path integrator and Landmark guidance mechanism models are integrated in a system where each is independent and can communicate simultaneously or successively depending on external stimuli and internal state. The red links show interacting mechanisms supported by empirical evidence.

Projects proceeding this, have implemented various navigational routines inspired by the Ant Navigational Toolkit model and leading to the current thesis. More specifically, models for visual-based navigation have been implemented and tested in the AntBot platform (see Section 3) [8, 9]. This project builds on the previous platform routines, to implement and test models emulating the ability of animals such as ants and bees to perform *Path Integration*. The models here implemented are novel and have only recently been conceived [1]. As such, we implement, and compare the models with the dual Robotics-Biology relationship in mind, i.e.: we build the models on a physical robot to concretely test and observe them in the real world; and we compare said models to known previous successful robotics solutions to the homing problem, thus gaining insights in both frameworks.

2.3 Path Integration

As previously mentioned, some animals, in particular those with a fixed nest, possess the ability to return to their lair after outwards runs for foraging or other necessities. One of the most accepted mechanisms conferring the animal such an ability is Path Integration. Any employment of Path Integration involves a continuous sum of distance and direction information while the animal (or *bot*) is moving outwards, and the use of said cumulative information to return home when necessary.

Over the years, Path Integration has been widely studied and versions of it implemented in robotic prototypes [22, 23]. The attractiveness of Path Integration as a model for homing is interesting for robotics, but a PI model both robust for homing purposes and biologically plausible can serve both engineering and biology as it can give insights on how nature has solved the homing problem.

Previous studies suggest the most robust PI models in animals are the ones including a geocentric Cartesian coordinate system [24].

2.4 Simple PI implementation

In its most simplistic form, Path Integration can be achieved by an animal (or robot) from the knowledge of its orientation and distance traveled as follows:

Let t be the time step in any algorithm and θ_t be the orientation of the robot at time t, then:

$$\theta_t = \theta_{t-1} + \Delta\theta \tag{1}$$

where $\Delta\theta$ is the change in orientation from time step $t-1$ to t.

If the robot's position is defined in an x-y Cartesian coordinate system, we can find the new coordinates x_t and y_t as:

$$x_t = x_{t-1} + d\cos\theta_t \tag{2}$$

$$y_t = y_{t-1} + d\sin\theta_t \tag{3}$$

where d the distance traveled from time step $t-1$ to t.

A continuous employment of equations (1), (2) and (3) can allow a robot to keep track of its pose in 2d (position + orientation), and later use said information to return to its starting point (2d origin in this case). To be able to use the above equations, however, it is necessary for the robot to compute or *sense* its change in orientation (i.e. $\Delta\theta$) and its distance traveled (i.e. d) on each iteration step.

2.5 Central Complex model

The Central Complex (CX), a cluster of modular neuropils in the insect brain, was previously shown play a fundamental role in spacial representations and locomotor control [25]. Not until recently, however, has there been a fully integrated model linking the Central Complex to Path Integration; more specifically the type of PI thought possible and plausible for insects, with their behavior and limited brain capabilities.

The model described below is based on a recent paper [1], where after analyzing the response of certain neurons in the brain of the bee in different conditions, it was possible to show that both neurons susceptible to polarized light (functioning as a compass) and Optic Flow sensitive neurons (giving self motion information to the animal), interact with identified neurons in the Central Complex. The connectivity of said neurons was further analyzed and from empirical evidence a model was proposed, capable of driving the animal back to its nest after arbitrary outward routes.

The model relevant for performing Path Integration is composed of three main layers: a *direction*, a *memory* and a *steering* layer. Each layer is composed of 8 to 16 *cells* and is partially connected to one to two other layers through either excitatory or inhibitory connections. Each of the cells in the network is a *spiking* neuron, which *increases*/*decreases* its potential in accordance to connected cells or outside stimuli, and possibly *decays* over time (see Figure 2).

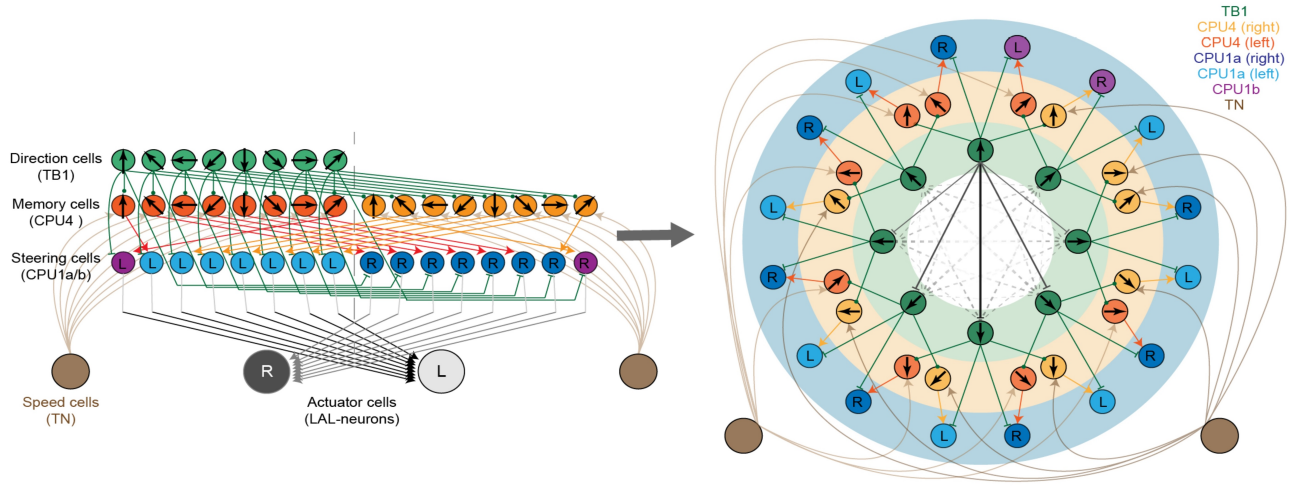


Figure 2: The Figure, taken from *Stone et al* [1], gives a graphic overview of the CX model. The Figure on the left shows the three sets of layers as described in Section 2.5, and the connections between cells of each layer. In the Figure on the right, the cells are disposed as a ring, giving visual cues of which cell encodes which direction or direction related information (arrows within the circles).

The *direction* layer (*TB1*), possesses 8 cells, each possessing a different *preferred* direction around the animal. The first cell's preferred direction is 0° and the following encode increasing directions clockwise from 0° and distant 45° from each other, covering the whole 360° range. The *TB1* layer retains the *orientation* of the robot, each cell being more or less active depending on its preferred direction being more or less correspondent to the current orientation of the robot.

The *memory* layer (*CPU4*) is composed of 16 cells, divided in two sets of 8 cells corresponding to the directional cells (TB1) for the left and right steering respectively. Each memory cell is charged proportionally to the distance traveled in its direction, but simultaneously reduced by a continuous *loss* of memory as the robot is running (*memory decay*).

The *steering* layer (*CPU1a/b*) is composed of 16 cells, 8 of which represent the amount the robot needs to turn *left* to return to its nest, and likewise the remaining 8, the amount the robot needs to turn *right*.

In addition to the described main layers, we possess further middle layers through which information is processed. Connected to the TB1 layer we have a *CL1* layer, and connected to the *CL1* a *TL1* layer; the roles of the *TL1* and *CL1* layers are explained in layer sections.

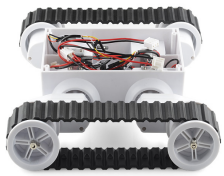
As the robot starts its outward run, the circuit runs in the background and *charges* its memory and steering cells accordingly to the robot behavior. The speed of the AntBot is given in input to the *memory* cells, which will all be excited proportionally to it. The *direction* layer regulates the excitement of the *memory* cells by inhibiting the cells proportionally to how concordant their direction is with the direction of the robot, leaving the most excited cells to be the ones modeling directions opposite its current one. When prompted to return to the nest, the network takes over the driving of the AntBot, which steers accordingly to *steering* layer, whose cell values are summed to retrieve (*left/right*) steering information. A constant, tuned, *memory decay* further regulates the *memory* cells behavior and allows the AntBot to detect when it is home during its inbound route (see Section 4.3 for implementation details of the CX model).

3 AntBot

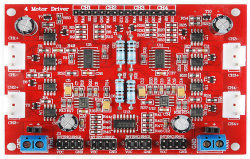
3.1 Hardware

The AntBot platform is a low cost robotics platform built to simulate biologically inspired algorithms for insect behavior. The platform, initially designed by Aleksandar Kodzhabashev [26] has four main components: a Dangu Rover 5 chassis, a motor driver board, an Arduino Mega 2560 and a Nexus 5 Android smart-phone (see Figure 3).

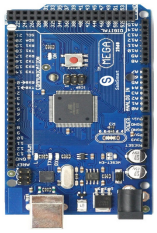
The majority of the processing takes place on custom Android applications running on the Nexus 5 smart-phone, which communicate to the Rover chassis through the Arduino board and motor driver board. The chassis features 4 independently drivable motors, powered by an on-board power supply; the Rover 5 motor driver board allows the Arduino to communicate with the Rover board, while a USB connection between the Nexus 5 and the Arduino allows the smart-phone to send commands to the electric board through its serial port.



(a) Dangu Rover 5 chassis

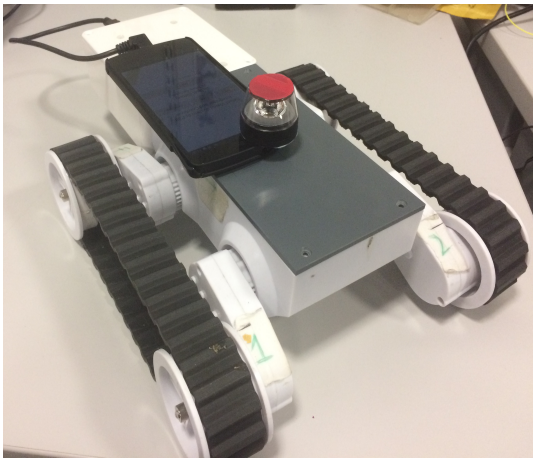


(b) Rover 5 motor board



(c) Arduino Mega 2560

(d) Nexus 5



(e) AntBot

Figure 3: The Figure shows the hardware components in the AntBot platform.

3.2 Software

The majority of software used to run the AntBot is custom made. This paper builds on code previously developed by Leonard Eberding [8] and Canicius Mwitta [9] (see Section 1.1 for an overview of the contributions in the project).

3.2.1 Arduino

The Arduino side of the AntBot Software is composed of two main components, a *parser* and an *executioner*. The *parser*, when called, reads and buffers the commands sent to the Arduino

through the serial port. The commands are thus transformed into action commands which are sent to the *executioner* for completion. The *executioner* receives action commands and executes them by duly driving the Rover 5 motors. After each command dispatch, the motor encoders are queried and speed/displacement information is retrieved and sent back through the serial port. As the AntBot is powered, the Arduino code runs continuously in the background. The *Main* application repeatedly employs the *parser* component which listens to and queues messages sent through the serial port, while actions are executed through the *executioner* should any be found.

For the purpose of this project, we add extensions to the parser and executioner to be able to *listen* for new commands, and execute these accordingly. Previous versions of the code would allow the AntBot to *move* forward or backwards a predetermined distance, *turn* right or left a given degree or *halt* the motors. As the experiments in the project required the robot to be running in a continuous smooth fashion, new commands were devised to allow the board to be run by sending left-right wheel *speed* commands, which endow the AntBot with the ability to move in arcs and generally to follow more realistic motion patterns.

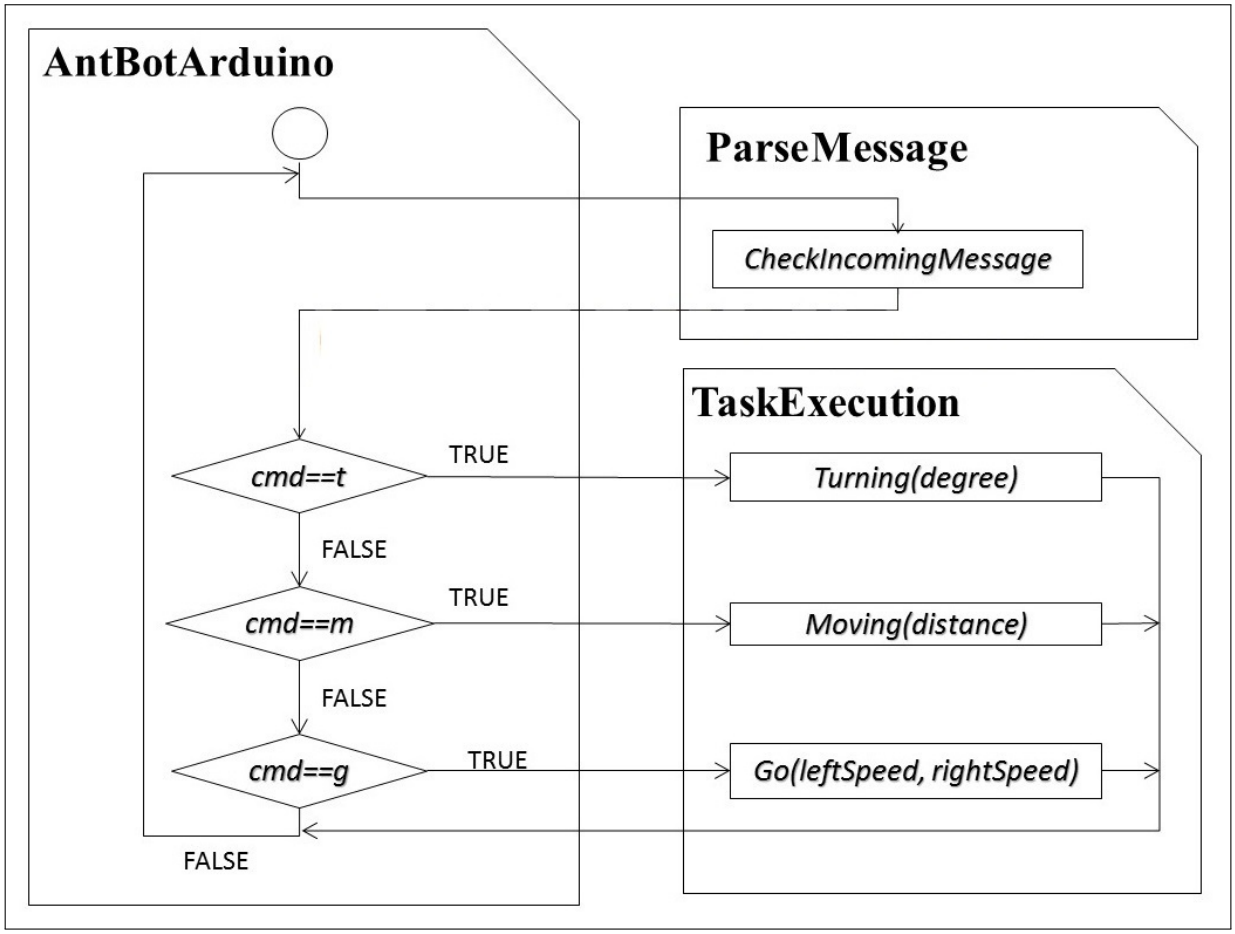


Figure 4: The Figure gives an overview of a typical run of the Arduino side of the software in this project. The *AntBotArduino* functions as a *main*, effectively controlling the execution of the various components. The *ParseMessage* module parses the incoming messages trough the serial port, while the *TaskExecution* module provides functions to instruct the Rover to perform actions, through the motor board.

Table 1: The tables shows the commands used to communicate to the Rover chassis through the Arduino board. A continuous loop listens to the serial port waiting for commands. The commands are sent in the form of messages which are duly parsed and thus executed through the Arduino executioner module (see Figure 4). The *Move* and *Turn* commands were largely based on a previous implementation by Eberding [8], the *Go* command was added from scratch to allow continuous motion during the runs.
Note: The syntax must be the one specified in the *message* column.

Command	**Message**	**Action**
Move	t 0 m *distance* n	Move forward of the specified *distance* (in meters).
Turn	t *angle* m 0 n	Turn of the specified *angle* (in degrees).
Turn and Move	t *angle* m *distance* n	Turn of the specified *angle*, after move forward of the specified *distance*.
Turn left	l	Constantly turn left.
Turn right	r	Constantly turn right
Go	**g *leftSpeed rightSpeed* n**	**Set left motor to *leftSpeed* and right motor to *rightSpeed*.**
Halt	h	Stop motors and bring robot to a halt.

3.2.2 Android

On the Android side of the AntBot software, five applications run concurrently to drive the robot accordingly to models devised. The applications communicate via a network of broadcasts which allows each thread to register and listen to events from other relevant threads.

The *SerialCommunicationApp* is the only application to communicate to the Arduino side of the AntBot software. The application uses *serial* libraries to send commands to the Arduino board, which parses and executes them through the components previously described.

The *PathIntegrationApp* and *VisualNavigationApp* are the *behavioral* applications, and are meant to implement the Path Integration and vision-based navigation algorithms on the AntBot. The applications are designed to be autonomous and be able to be run independently from other components should this be necessary.

The *CombinerApp* is the application which fuses the actions of the *behavioral* applications. This component communicates with the SerialCommunicationApp to drive the AntBot.

The *AntEye* application is the *main* component in the architecture. As the name may suggest, the application communicates with the smart-phone camera and therefore needs to run in the foreground. The application starts the necessary threads and runs all other applications in the system at its launch.

As the Android applications are running, a Java server can be connected to the smart-phone

through wi-fi hotspot. The server-side application can be used to send manual commands to the AntBot, through the Android and Arduino libraries. Moreover, the application allows for actions and commands to be monitored as the system is running the Rover chassis. Figure 5 gives an overview of the AntBot Software architecture.

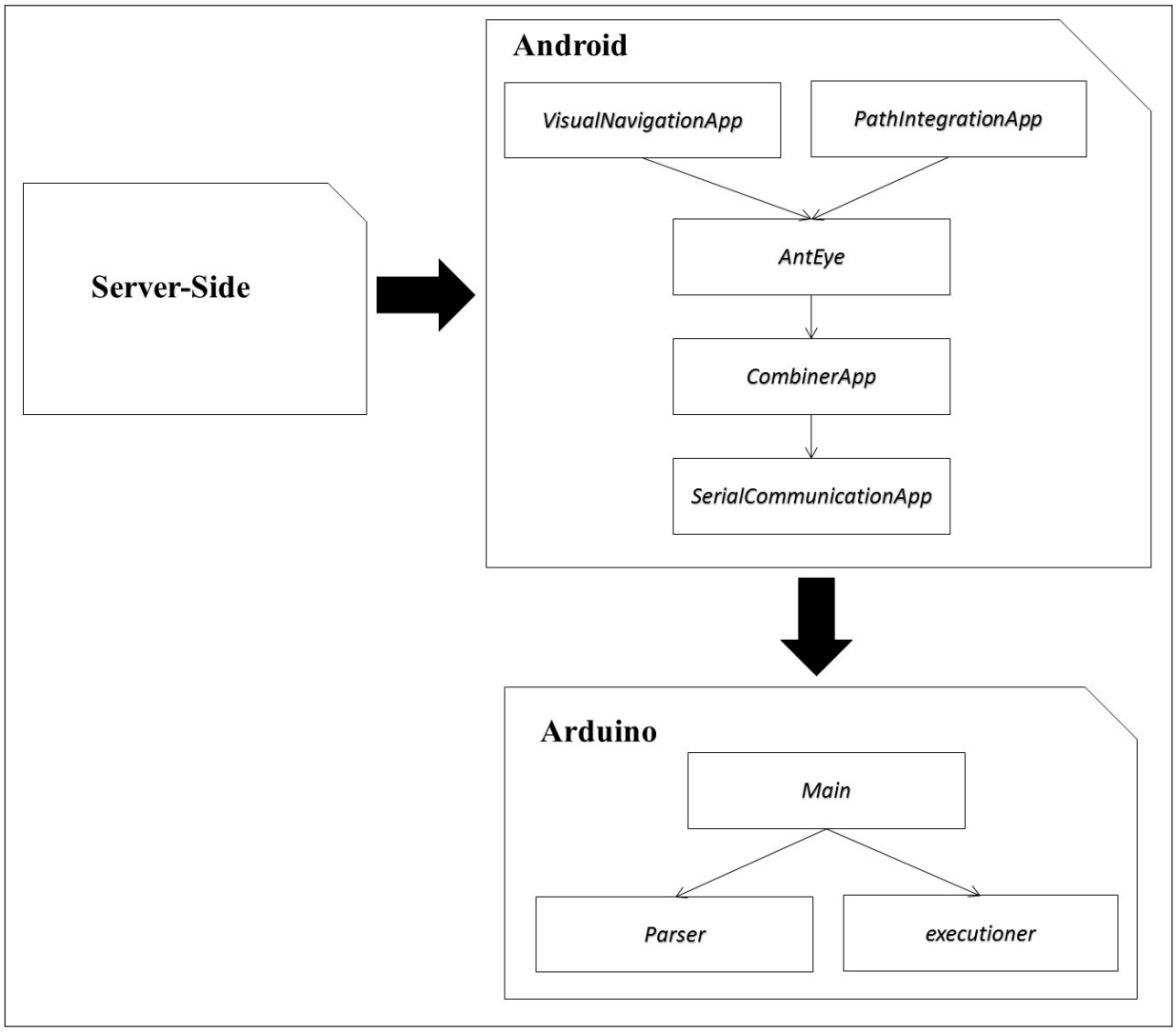


Figure 5: The Figure gives an overview of the AntBot software architecture. The majority of the software runs on the Android applications, which communicate to the Arduino side of the system to drive the Rover chassis. Moreover, a server-side application allows the monitoring of the actions in execution and for custom commands to be sent to the robot. An arrow from a component A to a component B is drawn if component A depends on or calls/listens to component B during its run.

4 Methods

4.1 Compass

One of the most fundamental pieces of information needed for the models described in Section 2.4 and Section 2.5 is the direction, or heading, of the animal throughout the run.

As a compass is required to implement the Simple PI and CX model, we need to retrieve orientation information from the platform. The Android OS and related devices provide several ways to retrieve such information, the simplest of which is by querying the smart-phone through the *TYPE_ORIENTATION* sensor. The logical sensor uses a combination of gravity and geomagnetic information (from the respective physical sensors) and combines them to retrieve an (x, y, z) vector significant of the yaw, pitch and roll of the device.

Although appealing in many ways (i.e. points north, noise is not cumulative etc.), the logical sensor is subject to magnetic field disturbances, which are almost guaranteed, given the position of the mobile phone on the platform. Moreover, after implementing a custom compass from the magnetic sensor, we observe the returned values to be unstable and oscillate between $-40°$ to $+40°$ from the real orientation of the smart-phone.

To devise a magnetic field interference-free compass we instead retrieve the yaw, pitch and roll from what is known as a *GAME_ROTATION_VECTOR* . The rotational vector is retrieved from the combination of gyroscope and accelerometer and therefore will not point north, but is otherwise unaffected by magnetic fields.

The logical *GAME_ROTATION_VECTOR* sensor, when queried in the system, returns four values, which we define as x, y, z and w. The four values, together, form a unit *quaternion*. We retrieve the yaw, pitch and roll from the quaternion as follows:

We first perform 5 middle computations, calculating values a, b, c, d, and e as

$$\begin{aligned} a &= -2(y^2z^2) + 1 \\ b &= 2(xy + wz) \\ c &= \begin{cases} -2(xz + wy) & if\ -2(xz + wy) \leq 1 \\ 1 & otherwise \end{cases} \\ d &= 2(yz - wx) \\ e &= -2(x^2 + y^2) + 1 \end{aligned} \tag{4}$$

Once the 5 middle computation values are retrieved, we can finally compute the *yaw*, *pitch* and *roll* as

$$\begin{aligned} yaw &= \arctan 2(a, b) \\ pitch &= \arcsin(c) \\ roll &= \arctan 2(d, e) \end{aligned} \tag{5}$$

The *arctan2* function accounts for the $\arctan 180°$ ambiguity and returns angles in the full 2π range.

At last, when the smart-phone is laid flat on the AntBot platform (see Figure 3e) we can obtain the 2D orientation degree θ by

$$\theta = \frac{180 * yaw}{\pi} \tag{6}$$

Bees and ants retrieve orientation information by polarization patterns in the sky. Similarly to the above implementation, the animal's *compass* will not necessarily point north, but serve as a relative heading reference throughout a run. The orientation retrieval here described, however, may suffer from cumulative error over time. Although that may be the case, with respect to the magnetic sensor, the *GAME_ROTATION_VECTOR* sensor's perceived "relative rotations are more accurate, and not impacted by magnetic field change" [27]. In addition, the drift is highly negligible and, with no precision instruments, empirically observed to be undetectable over runs $> 24hours$ (see Section 7 for more on compass alternatives).

4.2 Simple PI implementation

We build a simple Path Integration mechanism on the AntBot and later use it for comparison with the biologically inspired Central Complex model implementations. Here, we build on an existing PI mechanism in the AntBot [9], and follow the simple PI implementation in Section 2.4. We use equations (1), (2) and (3), to retrieve an estimate of the x-y Cartesian coordinate position of the AntBot at every action update iteration (i.e. $\approx \frac{1}{3}sec$).

We substitute θ_t in equation (1) with the orientation retrieved through the compass implementation of the smart-phone, and compute d (the distance traveled in equations (2) and (3)) as:

$$d = \frac{s_l + s_r}{2}\Delta t \tag{7}$$

where $\Delta t \approx \frac{1}{3}sec$, and s_l and s_r are respectively the left and right *action* speeds in input to the AntBot.

When the AntBot is made to return, we use the computed x_t, y_t and θ_t to compute a *direction* to turn (ρ) and *distance* to travel (r), for the robot to return *home*. We compute ρ and r as:

$$\rho = 180 + arctan\frac{y_t}{x_t} - \theta_t \tag{8}$$

$$r = \sqrt{x_t^2 + y_t^2} \tag{9}$$

4.3 Central Complex model

Like previously mentioned, the Central Complex has been shown to play a fundamental role in the animal's ability to perform homing after arbitrary outwards runs. After observing the neural connectivity in specific parts of the bee brain, and their excitation patters in various conditions, the Central Complex model was developed, putting together said evidence into a network. The network incorporates the sensory inputs of the animal (i.e. Optic Flow sensitive neurons for

speed, and polarized light sensing neurons for orientation) into a biologically inspired model, shown, in simulation, to be able to drive an agent back home after an arbitrary outward run.

In the CX model previously described (see Section 2.5), each of the information encoded in the different layers can be thought of in terms of addition of sinusoids at the fundamental frequency. In can be shown that any point in 2d can be expressed as a sinusoid. Here the sinusoid would represent a point in polar coordinates, where the *phase* shift of the sinusoid is representative of the angle and its amplitude is equivalent to the radial distance from the origin. It is in this framework that we encode the vector information in the various layers of the Central Complex model. We can describe each cell in a CX layer encoding positions in 2d as:

$$A\cos(\gamma+\theta) \tag{10}$$

where γ represents the preferred angle of the cell (see Figure 2), and θ the orientation of the animal.

The *memory* layer (*CPU4*), for example, can encode the *home vector* as a sinusoid where each set of 8 cells in the 16 composing the layer encodes a sinusoid with period 2π (redundant memory). An important property of this representation is that any linear combination of two sinusoids with the same period results in a sinusoid with the same period (although possibly a different phase shift and amplitude). If follows that by summing two sinusoids representative of two consecutive movement vectors, we obtain a sinusoid corresponding, in radial distance (i.e. amplitude) and angular coordinate (i.e. phase shift), to the sum of the consecutive movement vectors i.e.:

$$\begin{aligned} A\cos(\gamma+\theta_1)+B\cos(\gamma+\theta_2) &= A[\cos(\gamma)cos(\theta_1)-sin(\gamma)sin(\theta_1)] \\ &\quad + B[\cos(\gamma)cos(\theta_2)-sin(\gamma)sin(\theta_2)] \\ &= x\cos(\gamma)-y\sin(\gamma) \end{aligned}$$

where

$$\begin{aligned} x &= A\cos(\theta_1)+B\cos(\theta_2) \\ and \quad y &= A\sin(\theta_1)+B\sin(\theta_2) \end{aligned}$$

It is this property that allows the *compass* layer's inhibiting response to update the CPU4 layer by cell subtraction. The *compass* layer (*TB1*), in fact, encodes the orientation of the animal as a sinusoid at each given step and subtracts its cell values to the corresponding connected cells in the *memory* layer (inhibitory behavior) on each iteration step. The TB1 layer therefore charges the CPU4 layer by leaving more active those cells encoding the *opposite* direction to the one the robot is currently facing, effectively allowing the network to maintain a consistent *home vector* throughout [1].

We implement each layer in the Central Complex model following the mathematical description described in the *Simulation Methods* of *Stone et al* (see Appendix I for a *print* of the section, reporting in detail how each layer was implemented). As a starting point for the Java implementation of the model, we use the python simulation developed for the paper [1]. We implement the model through the use of the *ejml* Java library. The library possesses useful Java classes for manipulation of relatively small matrices besides methods for some basic matrix operations and manipulations. The majority of these, however, needed to be adapted and or created anew (see

Section 1.1).

We implement each layer in the Central Complex model described (see Appendix I) as an *ejml SimpleMatrix* matrix, each element of which represents a spiking neuron in the layer. We create and use 7 main operations to interact with the model cells during each *update iteration* in a typical run. Each of the 7 operations is fundamental in one of three primary activities:

1. Compass update
2. Displacement update
3. Turning Generation

Compass Update

In the *compass update* activity we create and use a *tl2Output* function to update the state of the tl2 layer. The tl2 *SimpleMatrix* layer is updated with the current orientation of the AntBot (retrieved from a custom made compass, see Section 4.1). We use this layer to compute the state of the *cl1* layer through the use of a *cl1Output* function. Finally we update the state of the *tb1* layer, encoding the compass information in the system (ring attractor state on the *protocerebral bridge* [1]), through the *tb1Output* function, which computes the new *tb1* state by combining its current state with the newly computed *cl1* layer (see complementary *code* for more information on the implementation of each function).

Displacement Update

At this point we possess an updated *tb1* layer encoding the orientation of the robot. In the *displacement update* activity we first update the state of the *memory* layer, by implementing a function *cpu4Update*, which modifies the *memory* state, by adjusting its former cells' values with both the compass information (now encoded in the *tb1* layer), and a given speed. For the purpose of these experiments we feed the system a speed proportional to the one sent in input to the platform motors during each run (i.e. the input speed is the *action* speed). At last, we encode the *memory* status to the *cpu4* layer, by passing each memory cell's value through a *sigmoid*, and setting it in the corresponding *cpu4* cell through the *cpu4Output* function.

Turning Generation

In the final activity we compute and set the steering layer's cells value through a *cpu1Output* function, after which the layer will encode the steering direction to return to the nest. We decode the layer through a *motorOutput* function, which returns the direction, in radians, needed to perform homing.

During a normal run of the system, each of the activities is executed sequentially on every iteration, allowing the memory cells (*cpu4*) to charge through repeated compass and memory updates, and lead the AntBot back to the *nest* after the end of the outbound route. As the AntBot is notified of the beginning of the inbound run, in fact, the computed direction from the *motorOutput* function is used, together with the current direction of the robot, to perform gradual orientation adjustments and finally head home. Figure 6 gives an overview of the algorithm.

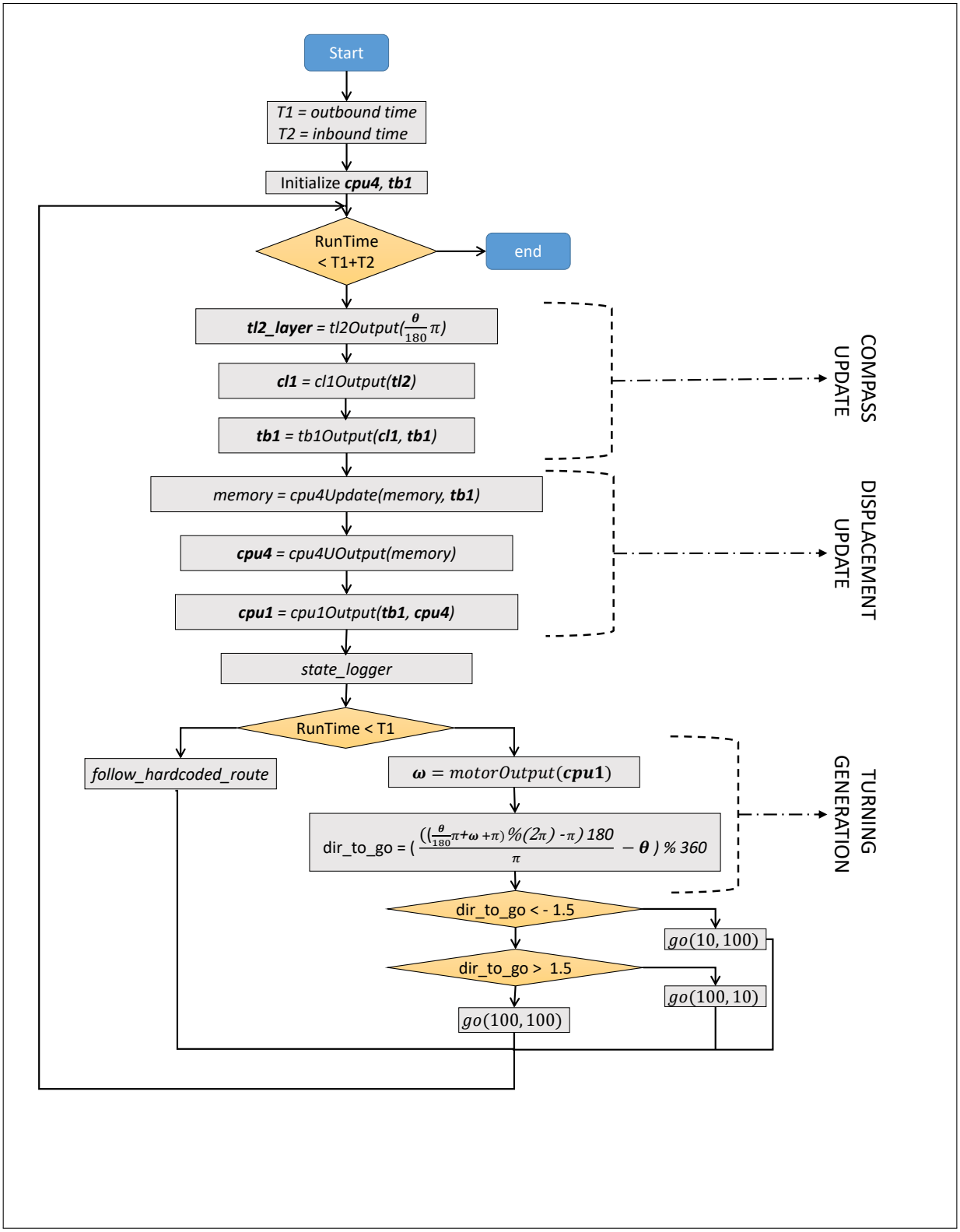


Figure 6: The Figures shows a flowchart of the high level algorithm used to run the AntBot through the CX model.

In this project, we achieve a *decoupling* of the robot action adjustments (motor commands) and model's updates (CX sequential compass and memory updates), by performing each of the two operations independently, through separate timers. After empirical observations and unit testing, we find the best network response when setting the model updates to a rate of $\approx 3.2Hz$, and the robot or action adjustments to a rate of $\approx 2.1Hz$. The decoupling was introduced to allow the model to reach a significant charged state after each run (see Section 5 for more information on run set ups). However, we observe cumulative error and the introduction of irregular behavior as the rate difference between the two decoupled systems increases. We therefore limit the decoupling to the above set rates.

Ultimately, the implementation of the models, needs *tuning* to account for the excitation, inhibition and decay rates and generally to allow the different layers to interact harmoniously and prevent any one to bias the network towards certain forms. Most preeminently we adjust a *memory_gain* and *memory_loss* coefficients to control the rate at which the cells *excite* given the compass layer and speed information, and the rate at which the memory cells *decay* over time. For the rest of the experiments we set $memory\_gain = 0.005$ and $memory\_loss = 0.0026$.

NOTE: the memory *gain* and *loss* parameters regulate the minimum number of memory update iterations required for the network to behave coherently. With the current settings (together with the speed range in input to the network), we perform at least ≈ 150 to 200 iterations on each run.

4.4 Holonomic motion and model enhancement

In recent updates of the paper by *Stone et al* [1], by studying the neural activity of Optic Flow related neurons in the Central Complex of the tropical *Megalopta* (a species of bee), a set of neurons was found to selectively provide input to two CX compartments; the neurons were called tangential neurons (TN). The TN's relation to the CX was probed in depth in the paper. The tangential neurons were found to react to specific movement directions (left vs right hemisphere) and to be connected to what were previously identified as CPU4 cells. The TN neurons are essentially speed sensing neurons providing information on the left/right speed of the animal. The combined left/right speed information, together with the orientation, possibly retrieved by the polarization patterns in the sky, can account for the previously implemented CX model to be robust against holonomic motion (e.g. the kind of motion a bee can achieve).

We modify the previous implementation of the CX model by attempting to: one, avoid depending on the action speed, previously given in input, and subject to hardware bias; two, account for holonomic motion, i.e. we make the network robust to the AntBot's orientation being different than its direction of motion. The model's changes follow the CX model in *Stone et al* [1]. To build the updated CX model we leave the core implementation previously achieved untouched and only modify/add parts of the architecture where needed.

First and foremost, we need two speed inputs in the network. The two speed *detectors* need be at specific left-right positions, more precisely at -45° and 45° with respect to the robot's frontal orientation. Each of the speed inputs needs to behave like the TN cells previously described, i.e. it should be most *excited* when the direction of the AntBot is in line with the orientation of the sensor, and react less as its direction shifts away for the same. Moreover, the sensor should

detect translatory speeds, and be otherwise invariant to rotations. The left and right speed retrieval is achieved through the use of Optic Flow (see Section 4.5 for a detailed description of the Optic Flow algorithms).

Once we posses left/right speeds, we need to modify the network to account for the new information. We change the connectivity between the TB1 and CPU4 (*compass* to *memory* layer) and CPU4 and CPU1 layers (*memory* to *steering* layer). The majority of the functions previously implemented now need differentiating between the right and left hemisphere and compute information differently (or with different rates) depending on which hemisphere they belong to. Moreover, we add two functions *tn1Output* and *tn2Output* which duly convert the left/right speed estimates to cells (*SimpleMatrix* matrices), in input to the *cpu4Update* function, which updates the memory according to their values and the compass layer (*tb1*).

4.5 Optic Flow

Optic Flow can be described as the pattern of visual motion perceived when an observer moves relative to an environment [28]. Strong evidence suggests that animals, when moving, perceive distinctive patterns of Optic Flow, which can be fundamental in retrieving information concerning the direction and velocity of motion, relative self distance to objects or even object's curvatures [29, 30, 31]. Although highly informative, the Optic Flow patterns are not so simple to use, in fact, one must first account for noise, distance variability of objects in the visual field and the rotatory-translatory dependencies of single vector velocities in the flow pattern, which make it impossible for any one single pixel to be informative, and thus require us to retrieve a larger set to be able to faithfully decompose the vectors into their rotational and translatory components (at least seven flow vectors have been shown to be needed [32]).

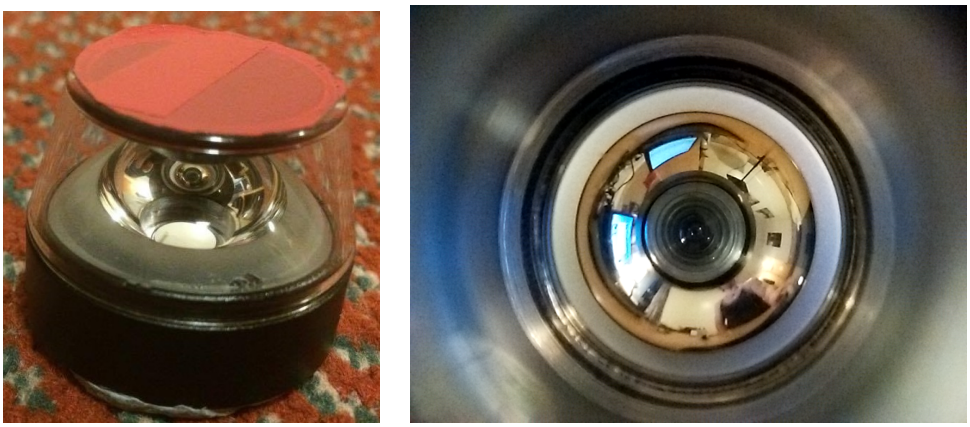
We retrieve speed cues and attempt to account for the above mentioned issues by using the extended matched filter model proposed by *Franz & Krapp* [2]. The idea behind a matched filter model is to use a *template* to filter retrieved flow vectors. Here, the filtering serves two main purposes: one, to extract speed information from the flow vectors; two, to minimize the variability of the information retrieved from the flow vectors (due to noise and due to the distance variability of objects from the robot in different scenes). It is important to note that, however reduced, the variability of objects in the environment is not accounted for, thus the model needs extending if more faithful measurements are required.

The process to retrieve the left and right speed information follows largely 3 steps, i.e.:

1. Frames retrieval
2. Flow computation
3. Filtering and speed retrieval

4.5.1 Frames Retrieval

To be able to use Optic Flow for speed computations it is first necessary to capture consecutive frames picturing the surrounding environment. For this purpose, we augment the front camera of the Android smart-phone with a 360 attachment, which allows each of the captured pictures to have a 360-ringed view of the surrounding environment (see Figure 7).



(a) Smart-phone attachment (b) Sample front camera image

Figure 7: The figures show an up-close view of the 360 attachment used to augment the smart-phone (Figure (a)) and a sample image taken from the front camera of the smart-phone, after augmenting it with the 360 attachment (Figure (b)).

To be able to process the incoming frames, the image processing needs to be done on the *Ant-Eye* Android application, as the foreground processes are the only ones capable of accessing the camera.

For each incoming frame we perform the following steps:

1. Retrieve the RGB (Red, Green and Blue) frame from front camera.
2. Extract the Blue channel (validated to be the most discriminatory channel in previous projects).
3. Crop the frame to extract the ring containing the 360° view of the surroundings.
4. Reshape the cropped frame into a rectangular 360x40px image.
5. Down sample the reshaped image into a 10x90px image corresponding to a 360° view of the surroundings.
6. Shift the frame around the x-axis to center the 360° view (i.e. pixels in column 45 correspond to 0° – front –, pixels in columns 0 and 90 correspond to −180° and 179° respectively – back –)

The first frame pre-processing steps are largely based on implementations from previous projects [8, 9], with changes to de-noising and blurring techniques for the final frame retrieval and the additional final shift to center frames w.r.t. the direction and view of the AntBot. Figure 8 gives an overview of the frame pre-processing steps.

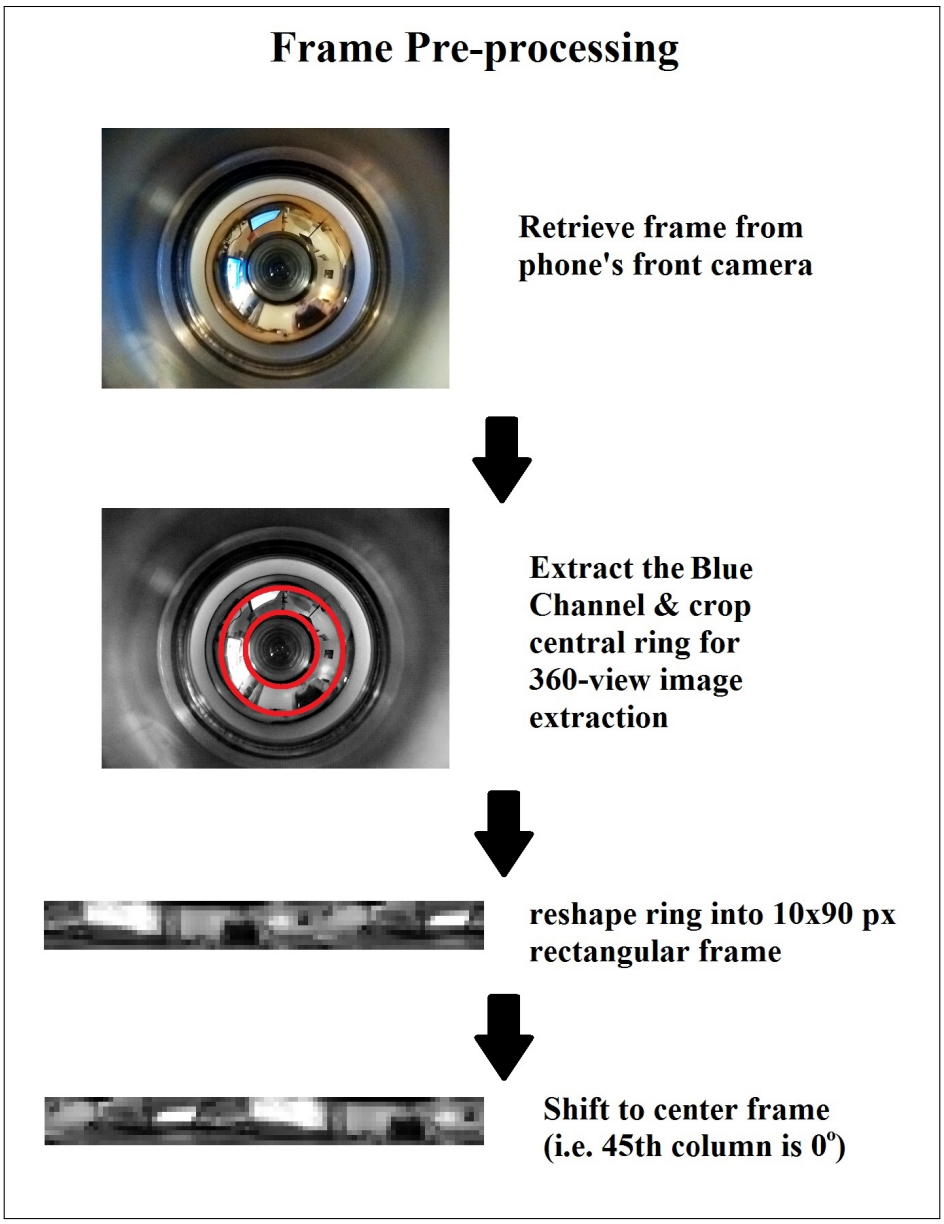
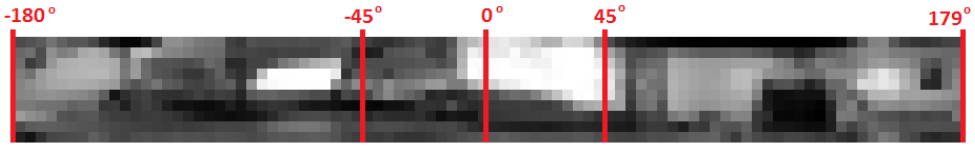


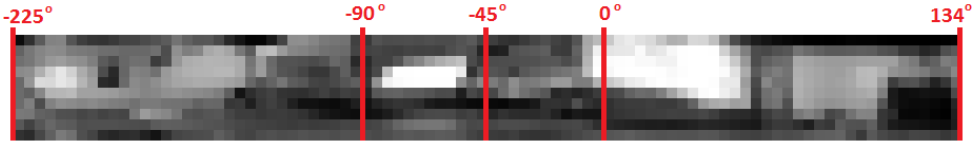
Figure 8: The Figure shows examples of a frame in the various pre-processing stages preceding Optic Flow computation.

4.5.2 Flow Computation

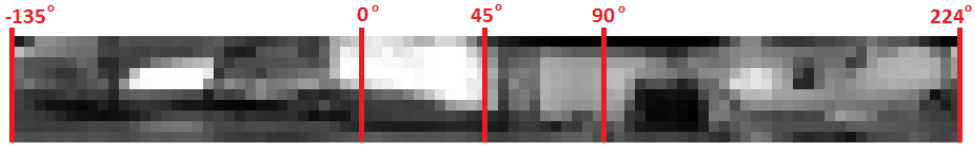
Once a 10x90px frame containing a 360° view of the robot surroundings is retrieved, we compute two further images each shifting the current frame left/right to match the preferred direction of the flow detectors in the bee (see Section 4.4). Therefor, one will be a 360° image centered at −45° and another will be centered at +45° w.r.t. the current robot heading (see Figure 9).



(a) Pre-processed frame



(b) Left flow frame



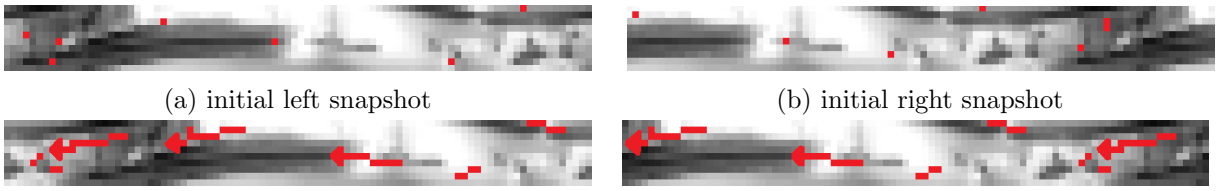
(c) Right Flow frame

Figure 9: The figures show the *shift* step in the flow pre-processing stage. The rectangular, centered, 360-view retrieved in the previous stages (Figure (a)) is shifted right/left to retrieve two additional 360 images: one, a left view centered at $-45°$ (Figure (b)); and two, a right view, centered at $+45°$ (Figure (c)).

After computing the two left-right images, we can finally retrieve flow vectors from each of the left and right frames in turn. We base the speed computation on two Optic Flow approaches: one, using the implementation of the sparse *Lucas-Kanade* algorithm [33]; and another, using the dense Optic Flow *Farneback* algorithm [34].

An iterative version of the *Lucas-Kanade* sparse Optic Flow algorithm is implemented through the OpenCv method *calcOpticalFlowPyrLK*, which given two consecutive frames and a set of points in the first frame, computes the position of said points in the second frame. To retrieve the set of points to track in the first frame we use the OpenCv $goodFeaturesToTrack$ function, which given an image, it retrieves the x-y coordinate of the most prominent corners in it. Figure 10 shows an example flow output when applying the above specified process to a sample frame.

The dense Optic Flow following *Farneback*'s algorithm is instead here implemented through the OpenCV *calcOpticalFlowFarneback* function, which given two consecutive frames computes a matrix of vectors (in x-y coordinates) corresponding to the displacements of each pixel from the previous to the current frame (i.e. each element of the matrix will be an array of double containing how much the pixel in the respective position has moved in its x and y component). This algorithm will compute a vector for every pixel in the image, and it will therefore not rely on the results from the *goodFeaturesToTrack* function.



(a) initial left snapshot (b) initial right snapshot

(c) left flow from subsequent snapshot (d) right flow from subsequent snapshot

Figure 10: The figures report a typical output example of the *Lucas-Kanade* sparse Optic Flow algorithm on two left and right consecutive frames. The top left and right images are taken at time t (Figure (a)-(b)) and show in red the points in the frames which are meant to be tracked in the next iteration (i.e. the points resulting from the *goodFeaturesToTrack* OpenCV function). The bottom two figures (Figure (c)-(d)) show the sparse flow output resulting from the tracking of the previously found points in the frames taken a $t+1$. The red arrows start from the position of the found points in the frame at time $t+1$ to the position of the same in the previous time step.

4.5.3 Filtering and Speed Retrieval (filter 1)

This far in the algorithm, we have a left and right OpenCv matrix containing flow vectors for the left and right frames respectively. To be able to apply the matched filter model it is now necessary to retrieve a template to filter the found vectors with. As the motion of the AntBot is limited to two dimensions (altitude can be discarded), we create a template vector containing the preferred flow direction for each of the pixel columns in the retrieved image (i.e. 90).

The template is constructed as follows:

We create a Nx3 matrix D where

$$D_i = [\cos(-\pi + i\frac{2\pi}{N}) \quad \sin(-\pi + i\frac{2\pi}{N}) \quad 0] \quad for \quad i \in \{0, N-1\} \tag{11}$$

are rows in D and $N = 90$ (we ignore 3D flow by setting the last value of each row vector to 0).

We create a left and right flow axes to use for the retrieval of the prefer flow vectors for the respective directions as:

$$axis_l = [\sin(\theta - \frac{\pi}{4}), \quad \cos(\theta - \frac{\pi}{4})] \quad and \quad axis_r = [\sin(\theta + \frac{\pi}{4}), \quad \cos(\theta + \frac{\pi}{4})] \tag{12}$$

where θ is the current orientation of the robot in radians.

Finally, we can compute the preferred flow vectors as:

$$F_{l,i} = D_i \times axis_l \times D_i \quad and \quad F_{r,i} = D_i \times axis_r \times D_i \quad for \quad i \in \{0, N-1\} \tag{13}$$

where $\times$ denotes the vector cross product and $F_{l,i}$ and $F_{r,i}$ are the i^{th} row vectors in the, respectively left and right, preferred flow matrix F. This is equivalent to equation (4) in *Franz & Krapp* [2].

Once the preferred flow vectors for each of the pixel columns in the frame are retrieved, it is time to apply the filtering through the computed template. We retrieve the left and right speeds

through the *linear range model* in *Franz & Krapp* [2]. For the *sparse* Optic Flow implementation, it is necessary to center all found vectors to 0, i.e. we subtract to each of the point positions found after applying the *Lucas-Kanade* algorithm, the positions of the same points in the previous frame. For the *dense* Optic Flow implementation, instead, each pixel contains a vector significant of the x-y motion displacement over two consecutive frames, and therefore does not need centering.

After the vectors are centered, we compute the left and right speeds as:

$$\begin{aligned} s_l &= \sum_{i=0}^{K-1} wF_{l,v_i[0]} \cdot v_i + n \\ s_r &= \sum_{i=0}^{K-1} wF_{r,v_i[0]} \cdot v_i + n \end{aligned} \tag{14}$$

where K is the number of vectors found by the sparse Optic Flow algorithm, v_i is the i^{th} found flow vector in the frame, n is noise (set to 0 in the current experiments) and w is a weight, here manually tuned and optimally found to be:

$$w = \frac{1000}{K\Delta t} \tag{15}$$

where Δt is the time elapsed between the two consecutive frames used for the Optic Flow computation. Note, here, the choice of $F_{r,v_i[0]}$, i.e. we filter each computed flow vector, with the preferred vector whose origin is the index corresponding to the x (or column) coordinate of the origin of the flow in the image. Figure 11 gives an overview of the filtering process.

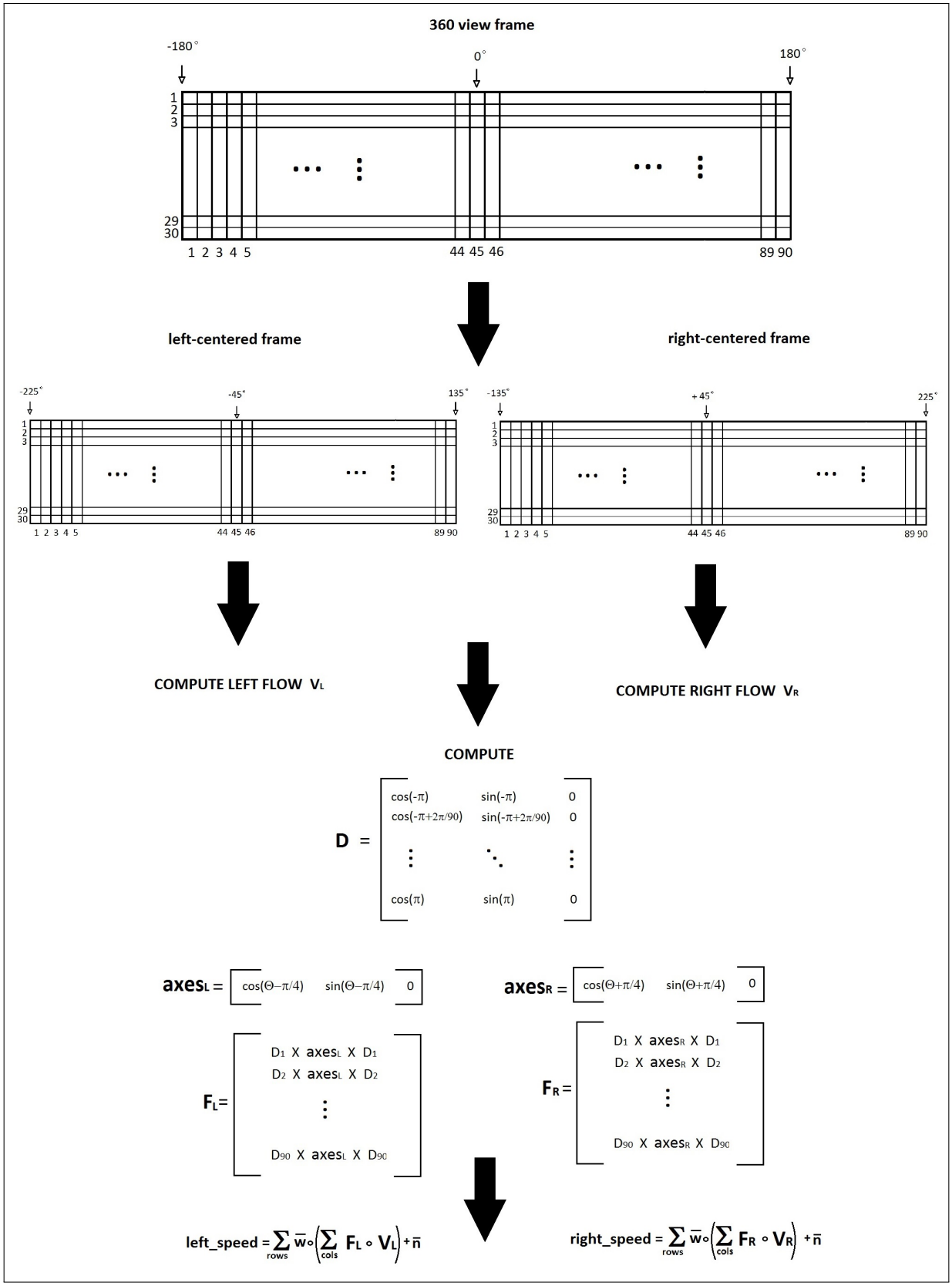
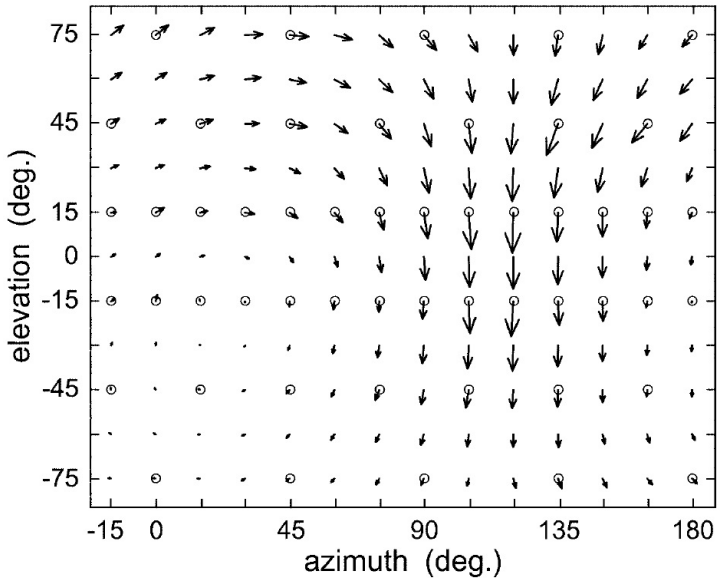


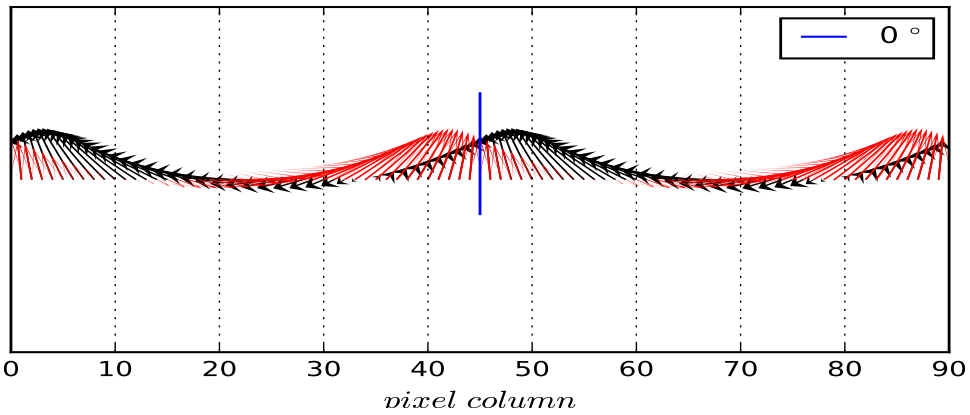
Figure 11: The Figure gives an overview of the implemented Optic flow left and right speed retrieval process, through match filters.

4.5.4 Filtering Variant (filter 2)

The type of filtering introduced in the previous section is optimal when dealing with cluttered environment and without any additional information on the distance of objects around the robot. Although that might be the case, the filter built in *Franz & Krapp* [2] and *Stone et al* [1] was done so expecting a flow pattern somewhat different than the type of flow detected from the smart-phone front camera when the 360° picture is extracted. Figure 12 shows the expected flow pattern found for the neuron VS7 in *Franz & Krapp*, and the response of our implemented filter for the *left* and *right* preferred direction over the 90 column pixels like previously described.



(a) VS7 neural response (preferred flow)



(b) Implemented filter(s)' response

Figure 12:
Figure (a) shows the response of the neuron VS7. The flow pattern is the one expected to be seen for the neuron to be maximally active. Similar patters are expected to filter the found flow vectors in the AntBot (Figure taken from *Franz & Krapp* [2]).
Figure (b) shows a plot of the left (black plot) and right (red plot) computed preferred flow vectors for each column in the 90 columned pixel frame used for Optic Flow computation.

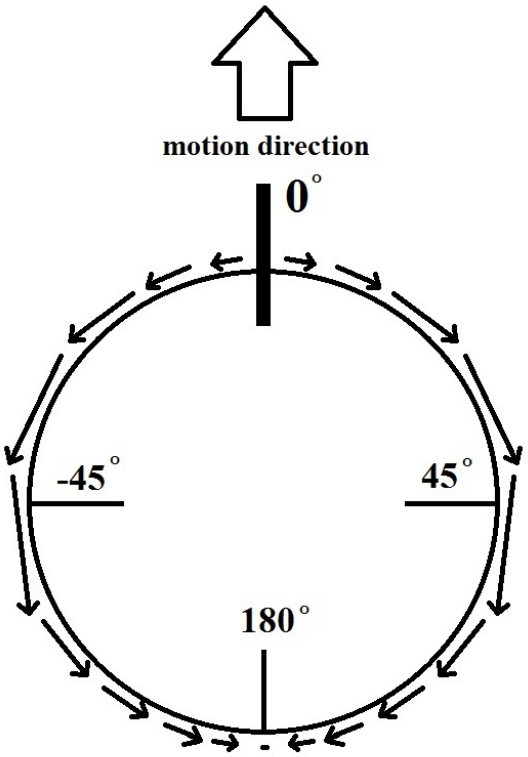
As the *bee* visual field is not limited to a narrow eye-height view of the world, but rather takes visual cues from below while it's moving, the flow pattern expected by some neurons in the brain is similar to the one in Figure 12a when turning. A turn, for the AntBot would translate in flow patterns detected being ≈ 0 on their elevation (or y) axis, and positive or negative (depending on the direction of motion) in their *azimuth* (or x) axis (see Figure 10).

We modify the filtering process to match the type of flow expected during an AntBot run. In such a run, the y axis will retain nearly no information and therefore can be discarded. Moreover, instead of projecting the found preferred vectors on the x axis for matching, we can find a more simplistic and useful description by using the projection of a moving vector in a unit circle, i.e. we substitute the filters F_l and F_r previously used for matching by a new filter F where:

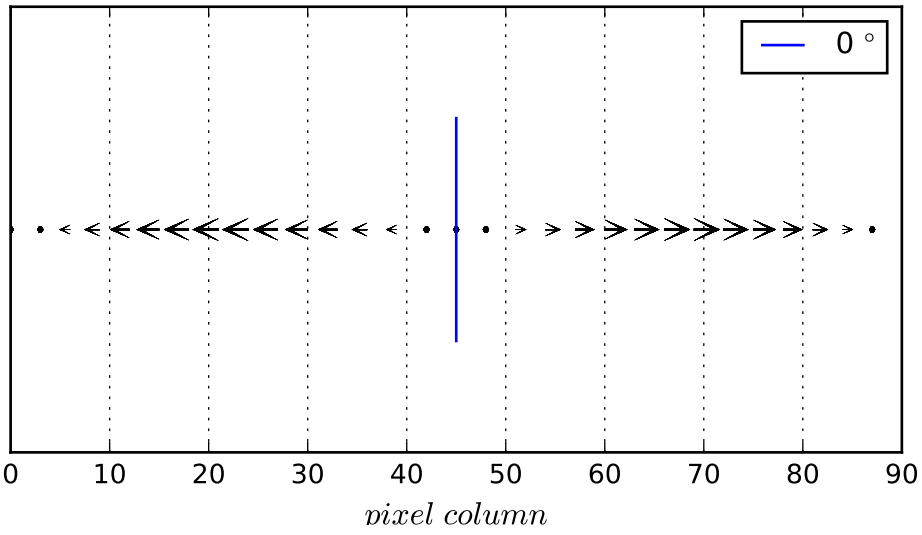
$$F_i = [\sin(-\pi + i\frac{2\pi}{N}) \quad 0 \quad 0] \quad for \quad i \in \{0, N-1\} \tag{16}$$

And N is the number of column pixels used (i.e. 90).

The filter used will be the same for the left and right shifted frames, allowing each preferred direction to be most active when the described pattern matches that of Figure 13 in the corresponding retrieved frame in their direction. The remaining filtering steps will otherwise be untouched.



(a) Motion pattern



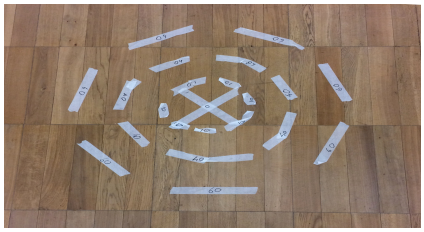
(b) Implemented filter(s)' response

Figure 13:
Figure (a) shows the idea behind the modification of the filter for speed retrieval. When moving forward, for example, we expect the rectangular 360 image to have vector flows different in length throughout the frame, in the pattern shown in the Figure.
Figure (b) shows the filter response of the modified filtering process for this project. During matching, the found flow vectors will be projected onto the corresponding filter vectors, nullifying the information on the y axis and scaling the vectors differently depending on the area of the image they are found in.

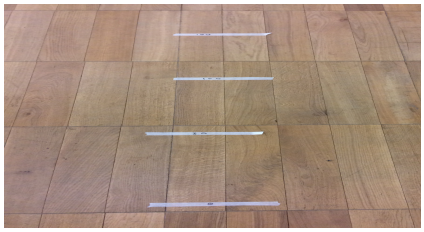
5 Testing

After implementing the Optic Flow speed retrieval algorithms, and the CX models as previously described in Java, we embed them in the Android *AntEye* existing application to allow them to drive the AntBot platform. We can now pass on to testing the models in the real world. We break down the testing in two blocks: in one, we test the Optic Flow based left-right speed retrieval (dense and sparse algorithm); in the other, we test the ability of the different models to perform homing after random outwards routes.

In all the following experiments we test the AntBot in a $1550x1240cm$ rectangular area, mainly bare of objects accounting for a non-cluttered environment (see Figure 14). Moreover, we gather information on each run through a custom made logger, which creates a file and later appends the state of the various internal variables (i.e. direction, speeds, memory cell state, frame rate, memory update iteration number etc.) throughout the run. The logger, was tuned to log information at $\approx 3Hz$, (not unintentionally) a similar rate to that of the memory updates in the system.



(a) Start



(b) Calibration section

(c) Pitch

Figure 14: The figures give an overview of the pitch used for testing. Figure (a) shows concentric rings, each 10 cm in radius larger than their inner ring, used as a starting position (*nest*) for the AntBot during *homing* tests. Figure (b) shows lines, each 10cm from each other, used to calibrate the speed of the AntBot during the runs. Figure (c) gives a general view of the pitch environment.

5.1 Optic Flow based speed retrieval

We test the Optic Flow based speed retrieval algorithm previously implemented on the AntBot, by running the system in the pitch in different conditions. We perform several sets of runs to: compare the first and second implemented matched filter; compare the *sparse* and *dense* Optic Flow implementations; assess the interpretability of the best performing filter responses, and fi-

nally test those against basic scenarios, to ensure the correctness of the implemented algorithms.

In the first set of experiments we perform 4 runs, each of which sees the AntBot running in a straight motion at a constant distance of $\approx 620cm$ from both the left and right walls. In *run 1* and *run 2* we set the matched filter to use the first filter implementation from *Franz & Krapp* [2] (*filter 1*), and run the AntBot once with the *sparse* Optic Flow implementation, and once with its *dense* counterpart. In *run 3* and *run 4*, we set up the experiments like before, except we use the second, custom, flow filter implementation for the filtering process (*filter 2* – Section 4.5.4).

In the second set of experiments, we set the matched filter to use *filter 2* and the *dense* Optic Flow algorithm, and test the ability of the matched filter response to react to runs with variable speeds. We therefore place the AntBot at a constant distance of $\approx 620cm$ from both the left and right walls, and run it in a straight motion at a speed of $\approx 6cm/sec$, $4.5cm/sec$ and $2cm/sec$. We thus compare the responses and assess the algorithm's suitability for speed estimation in the CX model.

In the third set of runs, we place the AntBot at the center of the described pitch, trying to minimize the bias induced by proximity to objects/walls, and we run 4 simple scenarios, i.e.: straight-forward motion (like previously), straight-backward motion, right curve and left curve. We log and compare the left-right filter responses in each case separately to assess their correctness with respect to the different types of motion.

Finally we assess the bias of the closeness to objects by placing the AntBot: first, at $\approx 190cm$ from the right wall and $\approx 1050cm$ from the left wall, testing the influence of objects' distances to the speed inputs; and after, at $\approx 1050cm$ from the right wall and $\approx 190cm$ from the left wall, effectively inversing the visual bias in the previous set.

5.2 Path Integration & Homing

We test the Path Integration models previously implemented, by running the AntBot on random outwards routes and later attempting to home with different strategies. We test the AntBot in the pitch previously described, where we run the algorithms for 30 trials, divided in 3 sets of 10 runs each. We create 10 random outward routes, each driven by the AntBot in $\approx 45sec$ and driving the robot at a variable distance between $600cm$ and $1200cm$ from *home*, after which we perform homing for a maximum of 45 additional seconds. We do testing by hard-coding the routes in the system and, for each route, carry out 3 different runs:

- In the first test, we run a thread in the Android smart-phone implementing the first Central Complex model. For each update iteration in the CX network ($\approx \frac{1}{3}sec$) we feed the compass information retrieved by the magnetic field independent orientation computation, and the speed at which we run the AntBot (*action* speed).
- In the second test, we run a thread implementing the second Central Complex model, where we feed the compass information, like previously, and a left-right *flow* speed retrieved by the *dense* Optic Flow algorithm described in previous sections (see Section 4.5), with the second implemented matched filter template (*filter 2*).

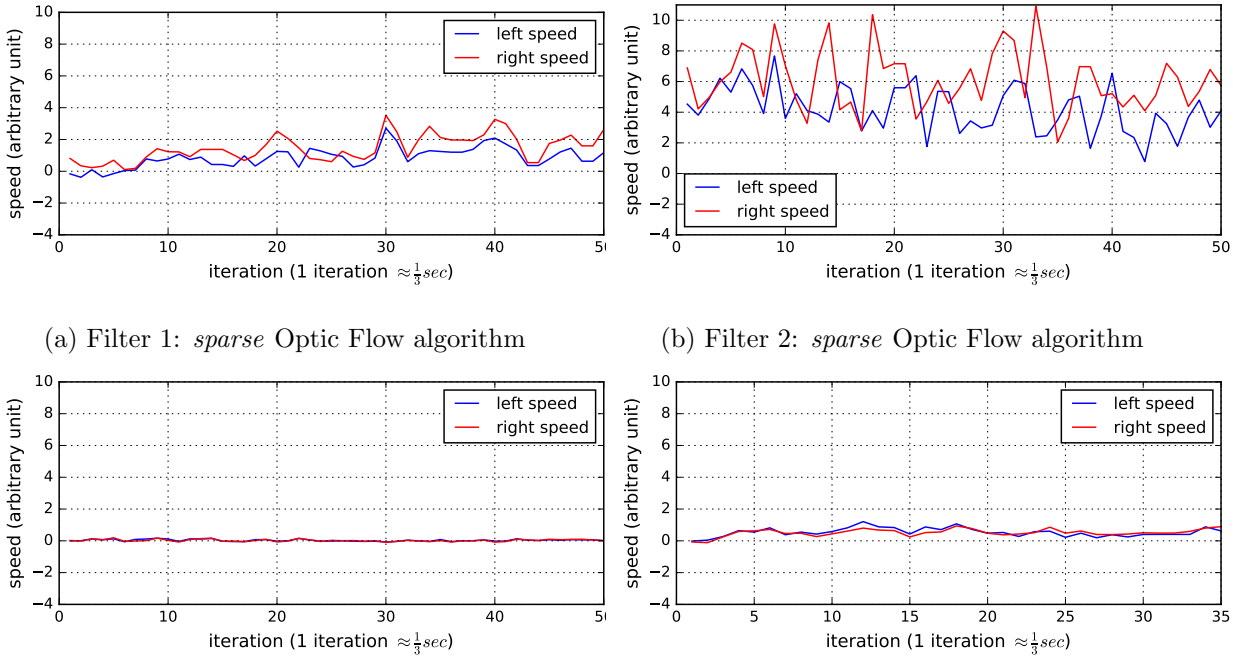
- In the final test, we run a thread implementing a classic Path Integration algorithm. We obtain a *home vector* by a simple sum of vectors over the run. Each vector in the sum is retrieved by a combination of the speed input to the board (speed/time for vector distance), and compass information like previously described (see Section 4.2).

After performing 10 runs as described above (i.e. 30 total runs), we compare the obtained results and assess the ability of the new models to perform homing.

6 Results

6.1 Optic Flow based speed retrieval

After running the AntBot for 4 runs, with different matched filters and Optic Flow algorithms, we assess the usefulness of each approach by analyzing the speed response for each run. Figure 15 gives an overview of the matched filter Optic Flow speed response in the runs. Given the settings for this set of tests (i.e. straight motion, constant speed and AntBot equidistant from left/right walls), we expect constant positive values as the filter's response to the Optic Flow readings, i.e. we expect the speed detector to read similar positive left/right speeds from the flow patters computed over consecutive frames.



(a) Filter 1: *sparse* Optic Flow algorithm

(b) Filter 2: *sparse* Optic Flow algorithm

(c) Filter 1: *dense* Optic Flow algorithm

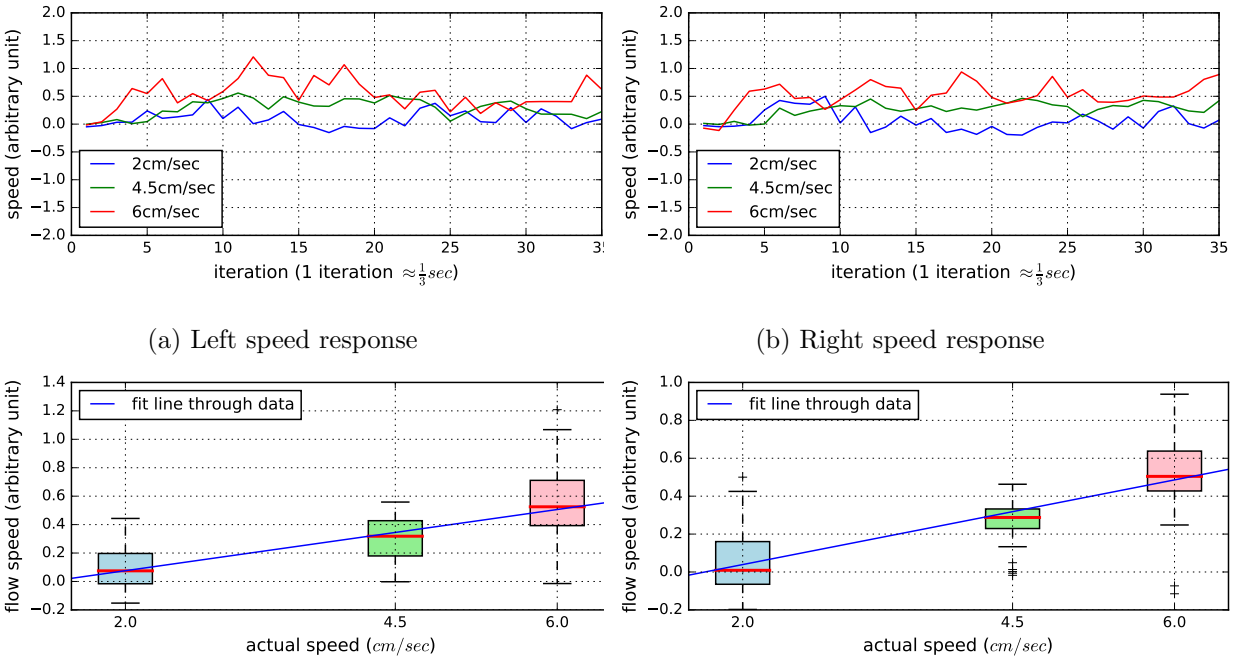
(d) Filter 2: *dense* Optic Flow algorithm

Figure 15: The figures report the matched filter's Optic Flow speed responses when filtered through the original implementation (Figures (a)-(c)) and the newly devised matched filter for the AntBot (Figures (b)-(d)). In all runs, the AntBot is running at a constant speed of $6cm/sec$. As clear from the graph, the sparse Optic Flow implementations (Figures (a)-(b)) result in largely variable values, making it hard for the system to charge its memory cells smoothly throughout a run. The dense Optic Flow implementations are more promising, giving a smoother response over the runs. The first filter (Figure (c)), however, does not capture the forward motion of the robot as the response oscillates from negative to positive values and averages to a 0 throughout the run. The combination of the dense Optic Flow algorithm with the second matched filter returns the expected response, reading a positive and, comparatively, much more stable left/right speed through the run.

As clear from Figure 15, the only reliable speed estimate in this very simple setting results with the combination of the *dense* Optic Flow algorithm with the *second* implemented matched filter (Figure 15d). The other responses are either too variable (charging the cells of the network in

undesirable and unpredictable ways), or quasi null (Figure 15c shows an example of the left and right flow patterns reporting values of ≈ 0 for the left and right speeds). The variability of the *sparse* flow implementation is due to the inability of the algorithm to guarantee homogeneous sparsity in the image. The flow vectors found on each frame, in fact, are heavily dependent on the objects and visible edges of the surrounding areas (less sharp edges translate to less points to track being detected by the *goodFeaturesToTrack* OpenCv algorithm). The non-balanced number of vectors in the left-right side of the frames retrieved accounts for unstable and generally unusable speed estimates (see Discussion for a more in depth analysis).

We set the AntBot to use the best performing algorithms (i.e. *filter 2* and *dense* Optic Flow), and test the matched flow algorithm's sensitivity to speed changes by running the AntBot, like previously, in a straight motion and equidistant from left/right walls, but at speeds of respectively $6cm/sec$, $4.5cm/sec$ and $2cm/sec$ in each run. Figure 16 shows the left and right filter's response for each trial in turn. As clear from the figures, the matched filter Optic Flow left and right speed response, if noisy, are as expected. The difference between the readings at $4.5cm/sec$ and $2cm/sec$ is somewhat less discernible but also largely correct. As the speed decreases, the image difference between two consecutive frames is increasingly less, and thus the Optic Flow algorithm will not be able to compute reliable flow vectors. When that is the case, the noisy readings of the flow might obscure the true computed vectors between consecutive frames and the filter's response might loose in accuracy (e.g. blue plot in Figure 16b between iteration 5 and 10).



(a) Left speed response

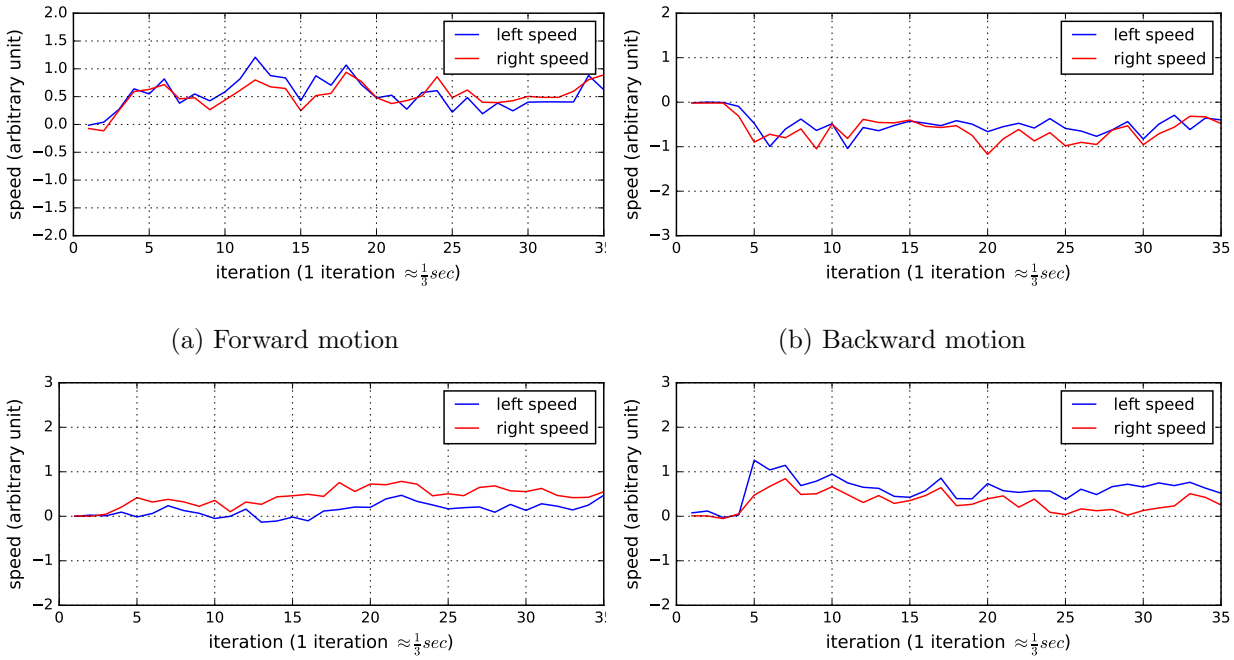
(b) Right speed response

(c) Left speed response - boxplot

(d) Right speed response - boxplot

Figure 16: The figures report the matched filter's speed responses when the flow pattern from the *dense* Optic Flow algorithm are filtered through *filter 2*, at different speeds. As clear from the figures the responses are largely correct, i.e. the filter detects increasingly left-right higher speeds as the true robot speed increases.

We test the left-right speed estimation of the *dense* Optic Flow algorithm with *filter 2* on simple scenarios to assess its correctness with respect to the type of motion performed. We run the AntBot at full speed ($\approx 6cm/sec$) forward and backward and log the matched filter responses on each iteration ($\approx \frac{1}{3}sec$). After, we test responses to left curves by running the right motors at full speed and the left at approximately 10% power, and vice-versa for right curves. Figure 17 shows the filter's response in each scenario. In Figures 17a and 17b is quite evident how the filter's response correctly discerns forward from backward motion. When moving forward, the left-right speeds are positive numbers, while when moving backwards at the same speed the response is similar in modulus but negative in sign. Both left and right responses are also approximately the same. In Figures 17c and 17d it is instead possible to see the responses of the filters when the robot performs curved motion. In both cases, the filter correctly detects the faster flow pattern on the right (when curving left) or left (when curving right), consistently throughout the runs.



(a) Forward motion

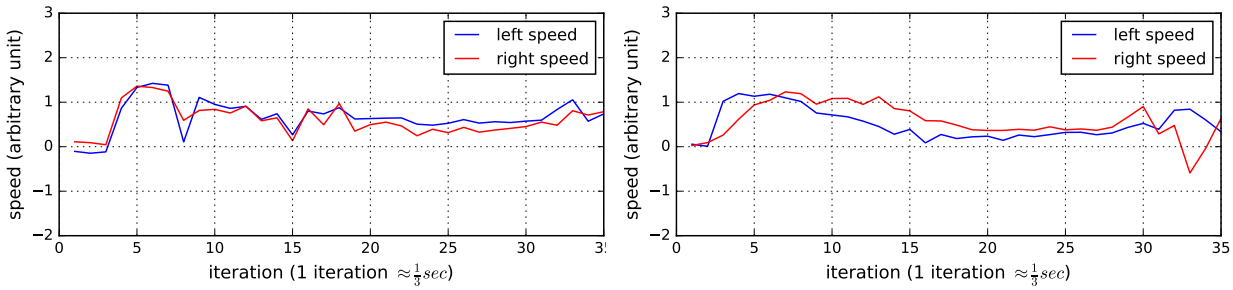
(b) Backward motion

(c) Left curve

(d) Right curve

Figure 17: The figures report the matched filter's Optic Flow speed responses when the AntBot is run through simple motion scenarios i.e.: moving forward (Figure (a)), moving backward (Figure (b)), curving left (Figure (c)) and curving right (Figure (d)).

Ultimately we test the AntBot on two further runs by biasing the left and right closeness to walls and measuring the matched filter response, expecting faster detected speeds for the closest walls.



(a) Left-biased run

(b) Right-biased run

Figure 18: The figures report the matched filter Optic Flow speed responses when persistently biased on the right or left by closeness to walls. In both cases it can be observed how the closer wall induces the algorithm to read faster speeds on the corresponding side of the AntBot.

For each of the *biased* runs, given the continuous unbalanced closeness to left or right walls, we expect the matched filter to detect faster speeds on the left or right side respectively. Figure 18 gives an overview of the filter responses for each of the different settings. Figure 18b clearly shows how closeness to the right wall induces the matched filter's algorithm to read faster right speeds. In Figure 18a, although the left speed is, as expected, faster than the right detected speed, it is not so by the same margin. The response is possibly due to a large number of factors, more eminently the type of left wall texture (effecting the Optic Flow's ability to detect flow vectors reliably), and the presence of edged corners (windows etc) accounting at times for distances being larger than the wall distances.

For the remaining experiments we set up the second CX model with its best performing variants, i.e. we perform Optic Flow computation through the *dense* Optic Flow algorithm and we filter the found flow patterns through *filter 2*.

6.2 Path Integration & Homing

After running the AntBot for 30 runs, effectively performing homing over 10 routes with 3 different Path Integration algorithms, we compare the performance of each algorithm in turn. The results of the runs are summarized in Table 2.

As expected, the standard Path Integration implementation establishes a good base performance for the robot. The AntBot using a simple sum of vectors, in fact, manages to return to within $30cm$ from the *nest* 9 times out of 10, setting a good comparative bound for the CX algorithms. The first CX model results somewhat less performing, managing to return to within $30cm$ from its starting position 7 times out of 10, but only missing its target of a distance grater than $50cm$ in one occasion (run No 8). The second CX model implemented performs 9 successful runs, effectively reaching a similar performance to the standard PI model in the test set. Figure 19 summarizes the results for each set of runs.

Table 2: The table reports the statistics gathered on the AntBot runs during testing. The tests are divided in three sets, each utilizing a different PI model and consisting of 10 runs where the AntBot is made to follow a random outward route for $\approx 45sec$ and later made to return to its starting position.

Model	Run No	Distance to Nest (cm)	Time to Home (sec)	Total memory update iterations	Average Frame Rate (fps)	Distance to Next (Mean)	Distance to Nest (Std)
First CX Model (1 speed input)	**1**	12	35	175	N/A	21.7	14.44
	2	3	36	169	N/A		
	3	9	32	172	N/A		
	4	32	35	177	N/A		
	5	8	24	143	N/A		
	6	34	40	162	N/A		
	7	29	25	150	N/A		
	8	52	41	174	N/A		
	9	24	49	189	N/A		
	10	14	28	148	N/A		
Second (enhanced) CX Model (left-right flow)	**1**	11	32	183	3.0592	16.2	11.06
	2	5	33	141	2.9034		
	3	7	29	175	2.9801		
	4	21	34	168	2.2462		
	5	10	24	153	3.1801		
	6	19	32	162	2.7462		
	7	41	24	155	1.6924		
	8	30	32	163	2.5021		
	9	11	43	187	3.1210		
	10	7	25	144	2.9141		
Standard (simple) Path Integration	**1**	12	29	N/A	N/A	18.2	8.61
	2	16	31	N/A	N/A		
	3	8	26	N/A	N/A		
	4	15	27	N/A	N/A		
	5	30	19	N/A	N/A		
	6	25	31	N/A	N/A		
	7	34	26	N/A	N/A		
	8	22	34	N/A	N/A		
	9	10	41	N/A	N/A		
	10	10	28	N/A	N/A		

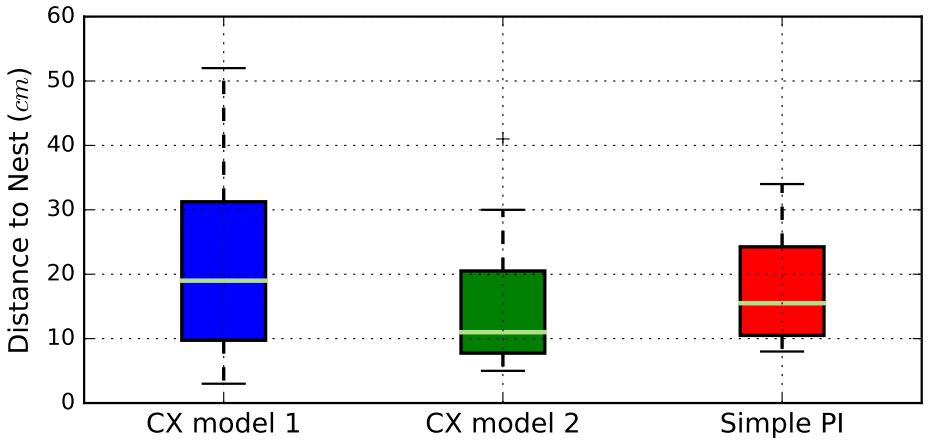
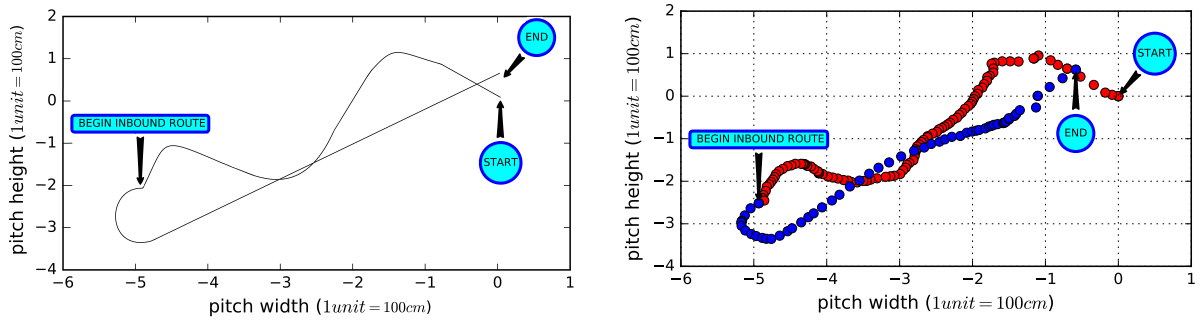


Figure 19: The Figure summarizes the results obtained for each implemented model. All implementation reach a similar performance. The simple PI implementation sets a good performance comparison for the CX model. Most evident amongst the results are the first CX model's variability from its median value and the second PI implementation's lowest *median* distance from *home*.

Both the standard PI model (sum of vectors), and the first implemented CX model have the drawback of relying on action information to compute the direction to follow when homing. The action information is known to be noisy, as the platform will not uphold any symmetry or uniformity constraint when executing actions, thus inducing the information to be subject to hardware bias (e.g. we observe different turning speeds when curving right or left with the same proportion of torque). Hence, we believe the main reason for the lower performance of the first CX model to lie the run time of the state update. The model, in fact, charges the cells during its outward routes (much like the standard PI computes the home vector on its way out). However, differently than the standard approach, it performs cell's updates on each iteration, thus recomputing cells values on its way back and possibly introducing further bias in the network, effectively inducing the AntBot to, at times, *miss* its *nest*. The standard PI model, on the other hand, after computing the home vector in the first 45 seconds of the run, will follow the computed vector to the starting position, without further adjusting its estimate on the way back (i.e. cumulating less noise).

The second CX model relies solely on the computed speeds from the flow algorithm and therefore is independent from hardware bias. The network's speed estimates, however, depend on object's closeness. If throughout the majority of the run, closer objects are present in the pitch on one side of robot, the network will behave as if the AntBot were going faster on the corresponding side and mis-charge the CX cells accordingly. This, can be seen in run No 7 (see Table 2), where the AntBot was observed to follow a wall on the right for $\approx 9sec$.



(a) Traced AntBot route

(b) *Cpu4* state map

Figure 20: The figures compare the *mental* state of the robot against its actual tracked route during a sample run. Figure (a) shows the reconstructed route after video-tracking the AntBot throughout the sample run. Figure (b) shows the positions of the AntBot at each iteration step ($\approx \frac{1}{3} sec$) as decoded from the logged memory cells (*cpu4* layer) during the run. The red circles represent the states from the start of the run to the end of the outbound route. The blue circles represent the states from the start of the inbound run to its end.

Interestingly, we notice a correlation between the AntBot's frame rate and its homing performance (see Table 2), observing better performance when the smart-phone frame rate increases. We plot the frame rate against the AntBot performance in Figure 21. The Figure shows a clear inverse relation between the final distance to the *nest* (or the homing performance of the AntBot) and the average frame rate registered during the runs. Assessing whether these results are statistically significant is beyond the scope of this paper, and given the limited number of tests it is much too early to determine whether that is truly the case. However, is it quite evident how the best performing runs were achieved with an average frame rate of $\approx 3 fps$ (bottom right cluster in Figure 21 contains the 6 best runs). It is therefore quite safe to put an empirical lower bound to the minimal frame rate needed for the AntBot to perform reliably, i.e. 3fps.

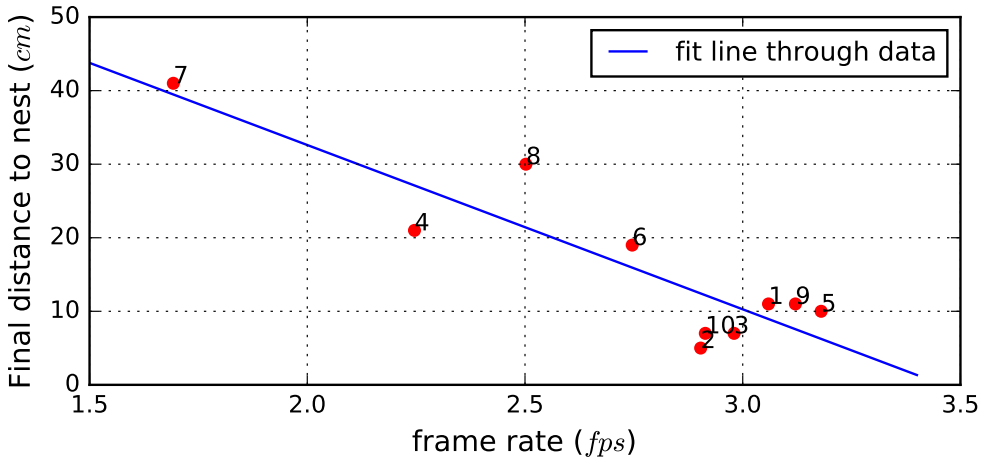


Figure 21: The Figure reports the final distance from the nest achieved at the end of each of the second CX model's 10 runs, against their average detected frame rate. Each point in the plot is tagged with the corresponding run number. The plot shows a clear inverse relation between the two dimensions.

Note: The network update iteration rate was previously decoupled from the motor action update and set at 3.2Hz (see Section 4.3) (here we find a best performance around 3fps!). The relationship between the network update iteration and the frame rate might indeed be important to the performance of the robot as intuitively, a too high frame rate would induce the network to barely detect any motion from one frame too another, while if the frame rate were to be too low, each network update iteration would pull the same speed states several times before the smart-phone updates them, thus accounting for larger and larger approximations to the real speeds at each given time.

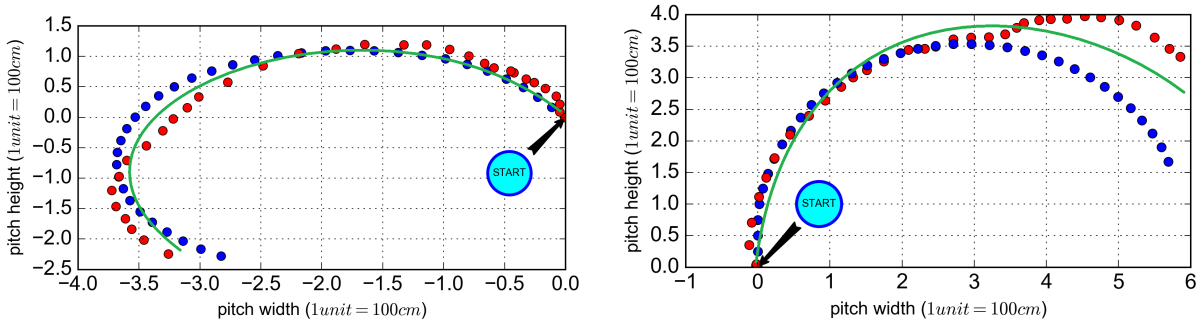
7 Discussion and Conclusion

After introducing and implementing the Central Complex model in the AntBot platform we test the models by performing several runs, each with different settings and/or combination of algorithms, and compare the homing behavior of the CX models to that of a simple Path Integration mechanisms.

The *sparse* Optic Flow algorithm was incapable of returning enough flow information for the matched filter to read off left-right speeds from the captured consecutive frames. The failure of the algorithm is due to its very nature and that of the matched filter. In the sparse Optic Flow algorithm, in fact, we find only a few trackable points on each frame and base the Optic Flow retrieval off of the same. The matched filter used to detect translations, is based on the underlying assumption of a flow *balance* in the two half of a frame to be processed, as clearly visible from the filter plot in Figure 13. If, for example, the AntBot was to rotate clockwise on the spot, the sum of the flow vectors in the right half of the image, after their projection to the preferred flow, would add up to a positive number which, on average, would equal in modulus the negative number resulting from the sum of the projected vectors on the right side of the same frame, ultimately reading a speed of ≈ 0. In the case of the sparse algorithm, even in this simple rotational case, it is clear how the balance of the flow vectors in the right/left side of the image would be hindered by the simple unpredictability of where, and how many vectors, the algorithm would find at any given point, which would depend on the environment, position, speed and many other factors throughout the run. As expected, then, the response of the matched filter on the retrieved *sparse* flow patterns, is non-uniform and otherwise unusable in the *CX* network.

The *dense* Optic Flow algorithm returns, in average, a good speed estimate during runs, effectively allowing the network to perform *homing* successfully. An Optic Flow based speed, with no additional objects' distance information, however, may perform differently in cluttered environments, as closer or further objects will account for the platform reading higher or lower speeds than usual. More specifically, one of two outcomes can be expected: one, random objects positions at various distances from the AntBot biases the network differently during each run and accounts for the platform to finally *miss* its target nest; two, over relatively long runs, the clutter observed will generally be balanced left and right, thus, on average, the noise will cancel itself out and allow the AntBot not to be influenced by it. It is worth exploring which is the case in future tests and assess the robustness of the approach against clutter.

The CX models are shown to be able to perform homing. The first CX model implemented reaches a larger median distance from the nest than the simple PI implementation, possibly due to the effect of cumulative *biased* error (due to hardware) over longer update times in each run (see Figure 22). This however might not be true if we do not feed the network *action* information. It would be interesting, in fact, to test the behavior of the first CX model against a simple PI mechanism when the speed in input to both is the one retrieved from wheel encoders in the platform, or better a combination of encoder's value and action information, effectively reducing noise and possibly changing the results gathered in this paper.



(a) Left curve state comparison

(b) Right curve state comparison

Figure 22: The figures compare the *position* belief of the AntBot on each iteration step ($\approx \frac{1}{3}sec$) when the robot is is made to curve left and right with the torque ratios specified in Section 6.1. The AntBot curves are *biased*, i.e. with proportionally opposite left-right torques the robot moves at different speeds. The blue circles show the positional *belief* of the AntBot when using the simple PI algorithm, and the *action* speed of the AntBot for distance estimates. The red circles show the positional *belief* of the AntBot when using the second CX model (decoding the *cpu4* layer), and retrieving left/right speed estimates from the Optic Flow algorithm. The green line shows the reconstructed curve after video-tracking the robot throughout the run. Clearly, after long curves, the robot position retrieved through the Optic Flow estimation is closer to the real position of the AntBot. The mismatch, however, is negligible for short curves since the direction, retrieved from the implemented compass, is independent from hardware bias, and will allow either algorithm to retrieve vectors pointing in the same direction. Nonetheless, the first PI implementation does not take into account the slower curving speed of the robot when performing left curves. As a result, the left curve positional estimate after $\approx 800/900cm$ is increasingly distant from the real position of the AntBot.

The second CX model performs closely to the simple PI mechanism, managing to return to within $30cm$ from *home* in 9 out of 10 runs. Amazingly, the model only uses visual inputs, i.e. besides the compass, it solely relies on the observed Optic Flow in the environment to charge the network and perform homing, giving testing evidence of the ability of the developed neural model to be able to drive back an agent after moving outwards. Moreover, the type of inputs in the network are noisy (see Section 6.1 for speed testing results), reflecting a noisy encoding of self-movement in the world, and providing evidence of the model's robustness against such noise (and flow computations on extremely low resolution images). We further observe possible correlations between the frame rate and the homing performance, which may also relate to the internal memory update rate previously set.

From the results gathered, the lower median distance to *home*, achieved by the second CX model could be due to: one, the type of *home vector* computed through the CX (spiking neural) network and sinusoidal encoding of vectors; two, to the Optic Flow computation which, unaffected by hardware bias accounts for a better performance. To assess which is which, it may be worthwhile running tests with additional scenario combinations. For example, the first CX model and the simple PI implementation could be adjusted to use the average left/right speed retrieved from Optic Flow, instead of the *action* speed given in input in the previous tests. Thereafter, their performance could be compared to the results here gathered.

Finally, the testing of the model on the AntBot, while extremely insightful, is limited by the type of motion the platform is capable of. More specifically, due to the inability of the Rover chassis to move in a specific direction without facing it first, we did not test the models against holonomic motion patterns and trails. Should the platform be augmented with holonomic wheels or the models re-used on a different, holonomic motion enabled, platform, it would be interesting to test the second implemented Central Complex model against such motion, and assess its robustness against it during simulation, in a real-world scenario.

8 Future Work

Throughout the project, design choices had to be made for the system to work at its best with the resources at hand. We discuss possible improvements and directions for future research.

The current compass implementation relies on the *GAME_ROTATION_VECTOR* sensor in the Nexus 5, which retrieves a quaternion through a combination of information from the gyroscope and accelerometer in the smart-phone. We process and retrieve the orientation of the robot by extracting rotation information from the quaternion. Although this methods allows more precise and reactive direction readings, it is bound to fail over time. The time required for any one reading to be off of $> 1°$ was empirically observed to be at least longer than $\approx 24h$. In the future, however, it might be worth retrieving compass information unaffected from cumulative noise. Previous experiments have shown how insects use the polarization pattern in the sky to orientate themselves [17]. For this reason, it could be worthwhile extending the platform with a (physical or logical) polarized light compass, which would be independent from both cumulative noise and motors' magnetic fields, and could give yet more insights on how insects perform such tasks.

A second important future update in the platform would be the adding of a simple obstacle avoider. In its current state, in fact, never during a run the AntBot is made to analyze its surroundings in search of potential obstacles to avoid or circumvent. A simple obstacle avoider, whether implemented through the use of the retrieved *dense* Optic Flow, or additional distance sensors, could endow the platform with the possibility to run in cluttered environments, thus giving additional testing prospects in the future. Moreover, the CX models, in simulation, are shown to be able drive an agent back to its *nest* in scenarios where multiple long obstacles are placed between the nest and the food source [1]. It would be worthwhile to test the ability of the implemented models to perform such tasks in a real-world scenario.

As the Optic Flow implementation goes, the 360° frames only take a *narrow* eye-height view of the surroundings. This view, besides returning different flow patterns than the ones hypothesized in animals (see Section 4.5.4), partly accounts for the retrieved speeds to be heavily dependent on the closeness of surrounding objects. When running the AntBot on environments with walls distant over $1200cm$ on any side, the computed flow vectors on the corresponding sides of the retrieved frames are insignificant and noise takes over effectively disrupting any possible accurate speed reading. Two solutions are possible: one, calibrating the flow detection beforehand; two, changing the angle of the camera for the retrieved frames. In the first case, the *calibration* stage could be set up so that the frame rate chosen for the robot's run would change based on distance readings of its surrounding. For example, a simple 360° distance reading of the pitch (achieved through a sonar or IR sensor) prior to a run, could allow us to decrease the frame rate proportionally to the median objects' distance in the room. The change between two consecutive frames then, would be considerable independently from how far walls are, and the speeds retrieved with the matched filter model described in this paper would be sensible values. The second solution would see the 360° camera attachment modified to retrieve views of the ground below the robot, beside the eye-height view previously shown. Looking at flow patterns on the ground would not only allow for the surrounding objects to affect less the retrieved speeds, but possibly to reintroduce the first filter model (see Section 4.5.3) as the flow patterns would be much similar to the ones expected from *bees* or *ants*.

Ultimately, the implemented models are within but only a subset of the ant's abilities to perform

navigation. Previous projects test the possibility of performing homing solely through visual cues [8, 9], and implement other proposed navigation models compatible to the *Ant Navigational Toolkit*'s view of the ant's navigational abilities. Future projects can merge the current models to visual based homing implementations [3, 11] and investigate ways of navigating through the joint or selective use of these methods. For example, to perform homing, at any given point in time a heuristic could be proposed to quantize the certainty of which direction to follow for each model, and a selector could use one or the other method based on their believed reliance (e.g. select the most certain, or set up a vote system for each model and pick a direction based on it). Another simple approach could see a combination of the models to retrieve a home direction, by for example averaging the home vectors of the first N most certain models.

Implementations such as the ones above proposed, could more interestingly allow us to set up a network whereby at the beginning of the run, the AntBot almost completely relies on the CX PI model to move outwards in search of food and perform homing. After a route has been found and followed several times, a visual based homing system will have gathered enough *snapshots* of the trail to and from the food source and will gradually increase in its *certainty* estimation of the home vector, effectively contributing to the decision of the direction to follow more and more, and allowing the runs to be more accurate (the contribution could for example be implemented in one of the ways above described) [4]. This behavior, if achieved, would be of great importance as evidence of the possibility of achieving ant like homing behavior in accordance to the *Ant Navigational Toolkit*'s model of navigation.

References

[1] Thomas Stone, Nicolai B. Wedding, et al. A neural substrate for path integration in the bee brain. 2017.

[2] Matthias O. Franz and Holger G. Krapp. Wide-field, motion-sensitive neurons and matched filters for optic flow fields. *Biological Cybernetics*, 83:185–197, 2000.

[3] Paul Ardin, Fei Peng, Michael Mangan, Konstantinos Lagogiannis, and Barbara Webb. Using an insect mushroom body circuit to encode route memory in complex natural environments. *PLoS Comput Biol*, 12, 2016.

[4] Martin Müller and Rüdiger Wehner. Path integration provides a scaffold for landmark learning in desert ants. *Current Biology*, 20:1368–1371, 2010.

[5] W. Grey Walter. An imitation of life. *Scientific American*, pages 42–45, 1950.

[6] W. Grey Walter. A machine that learns. *Scientific American*, pages 60–63, 1951.

[7] Barbara Webb. What does robotics offer animal behaviour? *Animal Behaviour*, 60:545–558, 2000.

[8] Leonard M. Eberding. Development and testing of an android-application-network based on the navigational toolkit of desert ants to control a rover using navigation and route following. Master's thesis, The University of Edinburgh, 6 2016.

[9] Canicius Mwitta. Developing antbot: A mobile-phone powered autonomous robot based on the insect brain. Master's thesis, The University of Edinburgh, 8 2016.

[10] Wehner and R. J Comp Physiol A. Desert ant navigation: how miniature brains solve complex tasks. *Journal of Comparative Physiology A*, 189:579588, 2003.

[11] Bart Baddeley, Paul Graham, Philip Husbands, and Andrew Philippides. A model of ant route navigation driven by scene familiarity. *Plos Computational Biology*, 8:1–16, 2012.

[12] Rüdiger Wehner. The architecture of the desert ant's navigational toolkit (hymenoptera: Formicidae). *Myrmecological News*, 12:85–96, 2012.

[13] Kathrin Steck, Bill S. Hansson, and Markus Knaden. Smells like home: Desert ants, cataglyphis fortis, use olfactory landmarks to pinpoint the nest. *Frontiers in Zoology*, 2009.

[14] Matthias Wittlinger, Rüdiger Wehner, and Harald Wolf. The ant odometer: Stepping on stilts and stumps. *Science*, 312:1965–1967, 2006.

[15] R. Wehner and B. Ronacher. Desert ants cataglyphis lotus use self-induced optic flow to measure distances travelled. *J Comp Physiol A*, 177:21–27, 1995.

[16] Lars Chittka and Jrgen Tautz. The spectral input to honeybee visual odometry. *The Journal of Experimental Biology*, 206:2393–2397, 2003.

[17] Rüdiger Wehner and Martin Mller. The significance of direct sunlight and polarized skylight in the ants celestial system of navigation. *Myrmecological News*, 103(33), 2006.

[18] S. P. D. Judd and T. S. Collett. Multiple stored views and landmark guidance in ants. *Nature*, 392:710–714, 1998.

[19] Lambrinosa et al. A mobile robot employing insect strategies for navigation. *Robotics and Autonomous Systems*, 30:39–64, 2000.

[20] Matthew Collett and Thomas S. Collett. The learning and maintenance of local vectors in desert ant navigation. *The Journal of Experimental Biology*, 212:895–900, 2009.

[21] Rüdiger Wehner et al. Ant navigation: One-way routes rather than maps. *Current Biology*, 16:7579, 2006.

[22] Dimitrios Lambrinosa, Ralf Möllera, Thomas Labhartb, Rolf Pfeifera, and Rüdiger Wehnerb. A mobile robot employing insect strategies for navigation. *Robotics and Autonomous Systems*, 30:3964, 2000.

[23] Thomas Haferlach, Jan Wessnitzer, Michael Mangan, and Barbara Webb. Evolving a neural model of insect path integration. *Adaptive Behavior*, 15:273–287, 2007.

[24] RJ. Vickerstaff and A. Cheung. Which coordinate system for modelling path integration? *Journal of theoretical biology*, 263:242–261.

[25] K. Pfeiffer and U. Homberg. Organization and functional roles of the central complex in the insect brain. *Annual review of Entomology*, 59:165–184.

[26] Aleksandar Kodzhabashev. Roboant: build your own android robot.
url: `http://blog.inf.ed.ac.uk/insectrobotics/roboant/`
last checked: April 17, 2017.

[27] Android. Api guides: Motion sensors.
url: `https://developer.android.com/guide/topics/sensors/sensors_motion.html#sensors-motion-rotate`
last checked: April 17, 2017.

[28] Andrew Burton and John Radford. *Thinking in Perspective: Critical Essays in the Study of Thought Processes*. Methuen.

[29] B. Ronacher and R. J Comp Wehner. Desert ants cataglyphis fortis use self-induced optic flow to measure distances travelled. *Journal of Comparative Physiology A*, 177:2127, 1995.

[30] Mandyam V. Srinivasan and R. L. Gregory. How bees exploit optic flow: Behavioural experiments and neural models. *Philosophical Transactions: Biological Science*, 337:253–259, 1992.

[31] Perotti VJ1, Todd JT, Lappin JS, and Phillips F. The perception of surface curvature from optical motion. *Percept Psychophys*, 60:377–388, 1998.

[32] H. C. Longuet-Higgins and K. Prazdny. Interpretation of a moving retinal image. volume 208, pages 385–397.

[33] B. D. Lucas and T. Kanade. An iterative image registration technique with an application to stereo vision. *Proceedings of Imaging Understanding Workshop*, pages 121–130, 1981.

[34] Gunnar Farnebäck. Two-frame motion estimation based on polynomial expansion. *Proceedings of the 13th Scandinavian Conference on Image Analysis*, pages 895–900, 2003.

APPENDIX I

THE NEXT PAGES CONTAIN A SECTION COPIED AND PRINTED WITH PERMISSION FROM *Stone et al* [1].

Simulation Methods

The proposed path integration network was implemented in Python 2.7, using a simple firing rate model for each neuron [48], in which the output firing rate r is a sigmoid function of the input I:

$$r = \frac{1}{(1+e^{-(aI-b)})}$$

where parameters a and b control the slope and offset of the sigmoid (see Extended Data Fig. 6 for parameter values and curve shapes). Optional Gaussian noise $\epsilon_r \sim \mathcal{N}(0, \sigma_r^2)$ can be added to the output, which is then clipped to fall between 0 and 1. The input I is given by the weighted sum of activity of neurons that synapse onto neuron j:

$$I_j = \sum_i w_{ij} r_i$$

In the current simulation these weights only take values of 0 (no connection), 1 (excitation) or -1 (inhibition) with optional added Gaussian noise $\epsilon_w \sim \mathcal{N}(0, \sigma_w^2)$. In the case of added noise, the sign of weights is preserved by clipping any values that fall outside this range, i.e. excitatory connections cannot become inhibitory and vice versa. Rather than tuning the weights, we tune the sigmoid function parameters for each neuron type to balance the number and scale of the inputs to each layer (see Extended Data Fig. 6a). Tuning was carried out visually, by attempting to ensure that each layer would cover a full range of firing rates during a batch of typical runs. Due to the robust nature of this network, many combinations of parameters work and there was no need to carry out extensive parameter tuning, e.g. through a grid search.

The model consists of five layers of neurons, some of which have additional properties to those above (described in more detail below): the TN and TL (input) layers receive direct sensory input from the agent; the TB1 (compass) layer has self-connection weights with values that

can fall between -1 and 0; the CPU4 (proposed memory) layer has additional synaptic accumulation; and the CPU1 (output) layer connects to the agent's motor system. In the following description of the individual layers of our model, we use θ for allocentric and φ for egocentric angles. A superscript in parentheses is used to represent values at a particular time step, and subscripts to differentiate parameters by layer and cell index.

Speed Layer 1 - TN neurons: In our simulation the speed estimate, in terms of forward-to-backward optic flow originating from the diagonally offset preference angles of TN cells on each hemisphere, is calculated by

$$I_{\mathrm{TN_L}} = \begin{bmatrix} \sin(\theta_\mathrm{h} - \phi_\mathrm{TN}) & \cos(\theta_\mathrm{h} - \phi_\mathrm{TN}) \end{bmatrix} \mathbf{v}$$

$$I_{\mathrm{TN_R}} = \begin{bmatrix} \sin(\theta_\mathrm{h} + \phi_\mathrm{TN}) & \cos(\theta_\mathrm{h} + \phi_\mathrm{TN}) \end{bmatrix} \mathbf{v}$$

where v is the velocity of the agent in Cartesian coordinates, $\theta_\mathrm{h} \in [0,\ 2\pi)$ is the current heading of the agent and φ_{TN} is the preference angle of a TN neuron, i.e. the point of expansion of optic flow that evokes the biggest response. For our model, a default preference angle of $\phi_{\mathrm{TN}} = \frac{\pi}{4}$ was used. TN2 neurons act as a rectified linear function, meaning they respond in a positive linearly proportional manner to I_{TN}, but have no response to negative flow (backwards motion) (Extended Data Fig. 6a).

$$r_{\mathrm{TN2}} = \max(0, 2I_{\mathrm{TN}} - 1)$$

Optional Gaussian noise ε_r can be added to the output, after which activity is clipped to fall between 0 and 1, as above.

Heading Layer 1 - TL neurons: The first direction-related layer consists of 16 inhibitory TL neurons, which have been shown to be polarization sensitive across a range of insect species [13,20,22] and to encode visual landmarks used to compute heading direction in flies (ring neurons

[11]). Each TL neuron has a preferred direction θ_{TL}, with the 16 neurons representing 8 cardinal directions $\theta_{TL} \in \{0, 45, 90, 135, 180, 225, 270, 315\}$ twice over. Collectively they encode the heading of the agent at every time step, by each receiving input activation corresponding to the cosine of the angular difference between the current and their preferred heading:

$$I_{\mathrm{TL}} = \cos(\theta_{\mathrm{TL}} - \theta_{\mathrm{h}})$$

Heading Layer 2 - CL1 neurons: The 16 CL1 neurons have a response as described in [49] i.e., they are inhibited by TL neuron activity, effectively inverting the polarization response. This is included for completeness but makes no functional difference in our current model.

Heading Layer 3 – TB1 neurons: The 8 TB1 neurons receive excitatory input from each pair of CL1 neurons that share same directional preference, θ_{TB1}. The TB1 layer also contains mutually inhibitory connections (see Extended Data Fig. 6b), with a weighting that reflects stronger inhibition for greater difference in their preferred directions [9,10,14]:

$$w_{ij} = \frac{\cos(\theta_{\mathrm{TB1},i} - \theta_{\mathrm{TB1},j}) - 1}{2}$$

Where $\theta_{TB1,i}$ and $\theta_{TB1,j}$ are the preferred directions of their respective CL1 inputs. The total input for each TB1 neuron is:

$$I^{(t)}_{\mathrm{TB1},j} = (1-c)r^{(t)}_{\mathrm{CL1},j} - c\sum_{i=1}^{8} w_{ij} r^{(t-1)}_{\mathrm{TB1},i}$$

where c = 0.33 is a scaling factor for the relative effect of lateral TB1 inhibition compared to the direct CL1 excitation. This layer thus acts as a ring attractor [50], which creates a stable sinusoidal encoding of the heading direction, reducing noise from the previous layers, and forming the underpinning for the accurate memory and steering functions in subsequent layers.

Layer 4 - CPU4 neurons: The CPU4 layer consists of 16 neurons. The input for these neurons is an accumulation of heading $\theta_h^{(t)}$ of the agent represented by the sinusoidal TB1 response, modulating the speed signal from the TN2 neurons in the noduli, as reported in the current paper. In addition, there is a constant memory decay to all CPU4 cells:

$$I_{\mathrm{CPU4}}^{(t)} = I_{\mathrm{CPU4}}^{(t-1)} + h(r_{\mathrm{TN2}}^{(t)} - r_{\mathrm{TB1}}^{(t)} - k)$$

where h = 0.0025 determines the rate of memory accumulation and k = 0.1 the uniform rate of memory loss. All memory cells are initialized with a charge of $I^{(0)} = 0.5$ and as they accumulate are clipped on each time step to fall between 0 and 1. The eight TB1 neurons each provide input to two CPU4 neurons, each of which also receives input from a single TN2 cell, coming from either the left or right hemisphere. As these neurons integrate the velocity (speed and direction) of the agent, activity across this layer at any point in time provides a population encoding of the home vector.

Layer 5 - Pontine neurons: 16 pontine neurons project contralaterally and connect two CBU columns eight columns apart from one another [31,34] (see Extended Data Fig. 5e,f and Extended Data Fig. 6b). Each cell receives input from one CPU4 column:

$$I_{\mathrm{Pontin}}^{(t)} = r_{\mathrm{CPU4}}^{(t)}$$

Layer 6 - CPU1 neurons: The CPU1 layer has 16 neurons. It consists of two subtypes of neurons, CPU1a and CPU1b that exhibit distinct projection patterns between the PB and the CBU and are conserved across insect species [31,34]. Each TB1 neuron provides inhibitory inputs (weight = -1) to two CPU1 neurons, in the same pattern as TB1-CPU4 connections. Additionally, each CPU4 neuron provides input to a CPU1 neuron, but with the offset connectivity pattern shown in figure 4g & h, which produces the connectivity matrix shown in Extended Data Fig. 6b. As TB1 input is inhibitory and CPU4 input excitatory, the effective input to CPU1 is the dif-

ference of the activity in these units, representing the difference between the integrated path and the current heading direction. Finally, CPU1 cells also receive inhibitory input from contralateral pontine neurons so their total input is:

$$I_{\mathrm{CPU1}}^{(t)} = r_{\mathrm{CPU4}}^{(t)} - r_{\mathrm{Pontin}}^{(t)} - r_{\mathrm{TB1}}^{(t)}$$

The CPU1 neurons form two sets, connecting to either the right or left motor units (postulated to be located in the lateral accessory lobes, the anatomical convergence site of CPU1 neurons). The activation of each set is summed, and the difference determines the turning direction and angle of the agent. Currently this is done by multiplying the difference in summed activity by a constant m=0.5, which is used to change the heading of the agent by that amount of radians:

$$\theta_{\mathrm{h}}^{(t)} = \theta_{\mathrm{h}}^{(t-1)} + m\left(\sum_{i=1}^{8} r_{\mathrm{CPU1L},i}^{(t)} - r_{\mathrm{CPU1R},i}^{(t)}\right)$$

All connection weight matrices and other model parameters can be seen in Extended Data Fig. 6.

In our model, the unit of distance is arbitrary, so we describe everything here in terms of steps in x and y and time steps t, which provides a meaningful measure of accuracy on a homing task by examining the tortuosity of a homing route, the angular errors, and errors relative to the distance of the outbound path. Outbound routes were generated by a filtered noise process, approximating a second order stochastic differential equation (SDE):

$$\omega^{(t)} = \lambda\omega^{(t-1)} + \epsilon_{\omega}$$

$$\theta_{\mathrm{h}}^{(t)} = \theta_{\mathrm{h}}^{(t-1)} + \omega^{(t)}$$

where for each time step the change in angular velocity ε_{ω} was generated by drawing from a von Mises distribution with zero mean:

$$\epsilon_{\omega} \sim VonMises(\epsilon_{\omega}|\mu = 0, \kappa)$$

where μ measures the location and κ is the concentration. For our simulations κ = 100, with smaller values increasing the tortuosity of the outbound route. We used λ = 0.4, to minimize excessive spiraling motion. Acceleration a for outbound routes is generated by drawing $\frac{T}{50}$ evenly spaced values from a uniform distribution:

$$a \sim U(a_{\min}, a_{\max})$$

and setting the acceleration between those points using third order spline interpolation, causing the agent to speed up and slow down in a smooth manner, thus imitating natural flight behavior. Velocity of the agent is determined at each time step by a linear drag model:

$$\mathbf{v}^{(t)} = \mathbf{v}^{(t-1)} + \begin{bmatrix} \sin(\theta_{\mathrm{h}}^{(t)}) \\ \cos(\theta_{\mathrm{h}}^{(t)}) \end{bmatrix} \cdot a^{(t)}(1 - F_D)$$

where F_D=0.15 is the default drag. For regular trials a_{min} = 0 and a_{max} = 0.15 were tuned to cause v to mostly fall below 0 and 1, allowing the TN cells to capture all speeds without their activity saturating, whereas for inbound paths a constant a = 0.1 was used. The agent's starting position on each simulation is $x^{(0)}$ = 0, $y^{(0)}$ = 0. These are updated iteratively depending on the velocity.

$$x^{(t)} = x^{(t-1)} + v_0^{(t)}$$

$$y^{(t)} = y^{(t-1)} + v_1^{(t)}$$

All code used for the described simulation is available at https://github.com/InsectRobotics/path-integration.