Developing AntBot: A navigational system inspired by the insect brain

Robert E. F. Mitchell

Master of Informatics
Informatics
School of Informatics
The University of Edinburgh
March 23, 2019

Supervised by
Dr. Barbara Webb

Acknowledgements

Pending . . .

Declaration

I declare that this dissertation was composed by myself, the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

Robert Mitchell

Abstract

Pending . . .

Contents

List of Figures

List of Tables

1 Introduction

Navigation is a complex task. Determining a sequence of actions to reach a known location, based on a combination of sensory inputs requires a lot of computational power. Desert ants, are capable of performing such a task over comparitively huge distances with limited, low resolution sensory information and remarkable efficiency. While the exact method by which the ants perform this task is still unknown, a reasonably complete navigational model can be constructed from existing physiologically plausible components, which may mimic the insect behaviour.

In this paper we introduce a combined model, the Extended Central Complex (ECX) model for insect navigation. To be clear, there is no (known) physiological basis for such a model; however, is not biologically infeasible, and may provide insight into the operation of the real insect brain. The ECX model combines the tasks of Visual Navigation, Path Integration, and Collision Avoidance; using, the Mushroom Body Circuit (MB) [1], the Central Complex model (CX) [**?**], and Optical Flow Collision Avoidance (OFCA) [7] for each task at a low level, then combining their outputs to get a form of higher visual processing (similar to the weighted "base model" described in [**?**][1]). The ECX model is a modified Central Complex model, named simply to ensure distinction between the two models. The individual components are all biologically plausible and two of three are known to be physiologically plausible. [1, **?**, 7].

This project primarily extends the work done in [7]. As such we continue using the AntBot platform; a robot constructed for the express purpose of experimenting with the algorithms in the *Ant Navigational Toolkit* [3, **?**].

1.1 Motivation

Currently, a full base model for insect navigation does not exist [**?**]. We here aim to take the abstraction presented by *Webb* and create a biologically plausible implementation using our three-system approach. Both the MB and CX models have been implemented and tested on AntBot previously [10, 7, 3, **?**], and a model combining the two has also been attempted by *Zhang* in [**?**]. This is used as an inspiration and will be discussed further in Section 2.4.

The previous AntBot implementations have demonstrated good performance of the CX and MB models individually [10, 7]. Performance of a combined system has also been shown to be reasonable, however, it is less consistent than we would desire [**?**]. In the combined model tests from [**?**] we note two key methodological problems: A fixed outbound route, and fixed component weightings. We address the former by adding the OFCA component to our model; as in [7], the AntBot will follow a non-deterministic outbound route through a cluttered arena. The latter brings up the more complicated question of plausible synaptic plasticity which, while undoubtedly interesting, lies outside the scope of this project (though some modelling of plasticity will be included). It is worth noting that here may also have been unknown tecnical issues with the robot which affected the results of [**?**], making them less consistent than they should be in reality (see [7] and later in this work).

[1] [DRAFT] This is the review paper that you sent me in prepreint. So far as I can tell, it has not yet been published; will it be out by April? How should I refer to this if the paper has not been published?

While [7] provides a reasonable collision avoidance system based on optical flow, it does not fit so neatly into the ECX model. We therefore aim to explore an alternative, yet still biologically plausible collision avoidance system which will fit into the ECX model.

Our ultimate hope, is to provide some insight into the precise biological systems in play during a point-to-point navigational task.

1.2 Practical Goals

We aim to build upon the experimental scenario from [7]. The robot will be tested by allowing it to navigate through a cluttered environment using a collision avoidance system. The navigational systems will then be tasked with bringing the robot home through the cluttered environment using a combination of visual information, a path integration vector, and collision avoidance.

In order to achieve this experimental goal, we break the project down into four components:

1. The first stage will involve solving some technical issues picked up by [7]; making any hardware/software adjustments required to provide a solid foundation on which to develop.

2. We can then begin investigating existing systems and establishing experimental metrics. This stage will involve research and review of new topics (the main one being the Central Complex model for Path Integration), and their implementations on the robot (if they exist). This stage will look to establish appropriate metrics by testing the existing CX model in a non-deterministic navigational task (see Section 4).

3. This stage will involve the set up and testing of the individual components of the ECX model. Building the modified optical flow system, adapting the work from *Zhang* to combine the MB model with the CX, and finally, putting the three pieces together.

4. Finally, the collection and compilation of results from the combined system and the individual systems.

1.3 Results

This work is based on work done previously by Leonard Eberding, Luca Scimeca, Zhaoyu Zhang, and Robert Mitchell. [3, 10, **?**, 7].

Significant contributions of this project:

1. Research and installation of a new compass sensor for the AntBot control systems.[2]

2. A major code refactor has taken place to make AntBot more usable for this project, and make it more accessible for future students.

[2] [DRAFT] practically, its looking like this will be less and less useful, I've not had a chance to actually try re-writing the control code to use the new sensor. That said, time went into this early on, and it would be nice to include this.

3. Addition of a calibration system to allow the user to auto-detect the position of the camera lens attachment (see Section 3).

4. Results gathered for the Central Complex model using a non-deterministic outbound route.

5. *[FUTURE]* Construction of a modified optical flow collision avoidance system, and a modified Mushroom Body model based on [7] and [**?**] respectively.

6. *[FUTURE]* Construction of a biologically plausible "base model" for insect navigation, the Extended Central Complex (ECX) model.

7. *[FUTURE]* Results indicating the capability of the ECX model.

2 Background

This project builds directly upon [7]. We first provide a review of the relevant background topics from that paper, before developing the relevant ideas further for this project. This will be a very brief summary of the work that took place and any relevant results or new perspectives. For more detail, please consult [7].

2.1 Optical Flow for Collision Avoidance

Optical flow is a large and diverse area of study. As such, we will not provide a complete background on the fundamental principles. Relevant terms will be explained, however, a succinct background of the concepts necessary for this paper can be found in [7]. A comprehensive introduction is given by *O'Donovan* in [8].

The main driving point in this paper is the integration of multiple navigational systems into a single model, namely, the Central Complex. Optical flow flitering worked well for a standalone collision avoidance system, however, it does not fit so neatly into the CX model. We therefore require a different approach. The relevant background revisits the concept of the *focus of expansion* (FOE) from [7, 8]. The FOE is the point from which all optical flow vectors originate. The location of the FOE can tell us things about the motion, and depth of the image.

In [7], the FOE was used only to compute time-to-contact with an obstacle. In this work, we instead use it directly to determine the potential location of an obstacle. As stated in [2] [3], computing the FOE is not trivial. Following [7] we will be using a dense optic flow field (tracking motion for every pixel in the image) as the computation of sparse fields was shown to be unreliable on the AntBot. This makes the problem more difficult; the basic from-flow method for computing the FOE given by [8] is computationally complex for a dense flow field [7]. The time taken to compute may become prohibitive. We will therefore look at using a *subset*[4] (see Section 4) optical flow field for the method provided by *O'Donovan*.
We will also revisit the matched-filter collision avoidance system from [7], though this system remains wholly unchanged at its core. For details, please see [7].

2.1.1 The Least Squared-Error method

This is the method given by *O'Donovan* and discussed in [7]. The FOE is computed simply as:

$$FOE = (A^T A)^{-1} A^T \mathbf{b} \tag{1}$$

$$A = \begin{bmatrix} a_{00} & a_{01} \\ \dots & \dots \\ a_{n0} & a_{n1} \end{bmatrix} \qquad \mathbf{b} = \begin{bmatrix} b_0 \\ \dots \\ b_n \end{bmatrix}$$

[3] [DRAFT] This is the source for the 'fuzzy' FOE; not included in this submission but I may revisit it, time-allowing; I have most of the section written

[4] [DRAFT]This is a subset of a dense flow field, tracking every nth pixel instead of all pixels. This isn't a dense flow field and it isn't sparse in the classical sense so I provide this term; it will be explained in Methods.

Where, each pixel $p_i = (x, y)$ has associated flow vector $\mathbf{v} = (u, v)$. Finally, set $a_{i0} = u$, $a_{i1} = v$ and $bi = xv - yu$. Note that this computation and explanation have been more or less copied from [7] for the benefit of the reader. For more details, the reader should consult [7], or [8].

This method for estimating the focus of expansion was originally given by *Tistarelli et al.* in their paper *Dynamic Stereo in Visual Navigation*[**?**, 8], and it serves as an excellent example for the reason the FOE is so difficult to compute. In theory, we should be able to take any two vectors **u**, **v** from the flow field, and compute the point at which lines running along them intersect. This point of intersection would give us the FOE[8]. While wonderfully simple, this method only works for a perfect flow field. In reality, flow fields are imperfect; for example, visual noise can cause disruptions to areas of the field. Therefore picking two arbitrary vectors could lead to an erroneous FOE. Unravelling the matrix notation of Equation 1, shows this to be a least squares technique; essentially, we try to fit the best FOE to the flow field given.

2.2 The Mushroom Body for Visual Navigation

The Mushroom Body model is an artificial neural network which models the mushroom body structures present in the insect brain[1]. It consists of three layers: Projection Neurons (PNs), Kenyon Cells (KCs) and Mushroom Body Output Neurons (MBONs). These MBONs are also referred to as Extrinsic Neurons (ENs) by older works, we use MBON herein. The original MB model proposed by *Ardin et al.* for navigation in [1] contained 360 visual PNs (vPNs), 20,000 KCs, and a single MBON. Every KC connects to the MBON, and each KC also connects to 10 vPNs chosen uniform randomly. The KC-MBON connections all start with weight $w = 1$. The figure and caption from *Ardin et al.* is given here in Figure 1. In the implementation by *Ardin et al.* and previous AntBot implementations, captured images were converted to greyscale and downsampled [1, 3, **?**, 7]. As such, each pixel can be described by its brightness (greyscale value) and each KC has a brightness threshold.

Learning occurs by showing patterns (images) to the vPNs. Each KC sums the brightness of all connected vPNs and compares that sum to the activation threshold. If the total brightness is greater than the threshold then the KC is *activated*. As the connections between the vPN and KC layers are random, a given image will generate some sparse pattern of KC activation; to learn this pattern, the weight of the KC-MBON connection is lowered from 1 to 0 for the active KCs.

To determine image familiarity during the recapitulation process, a pattern is projected onto the vPNs (again giving a sparse pattern of KC activation). The MBON then sums the synaptic (KC-MBON) weights of all active KCs to obtain a *familiarity* measure; recall, these weights are decayed as part of the learing process; thus, the lower the MBON output, the more familiar the image (and vice versa). Different route following strategies have been implemented (scanning [1, 3, **?**], Klinokinesis [**?**], combination with the CX model [**?**](see below), and visual scanning [7]); but, most commonly, some form of scanning is used whereby the agent scans an arc for the most familiar direction.

This has been a brief explanation for context. Part 1 gives slightly deeper insight[7] (particularly in relation to AntBot projects), and of course the work of *Ardin et al.* can give the full details [1].

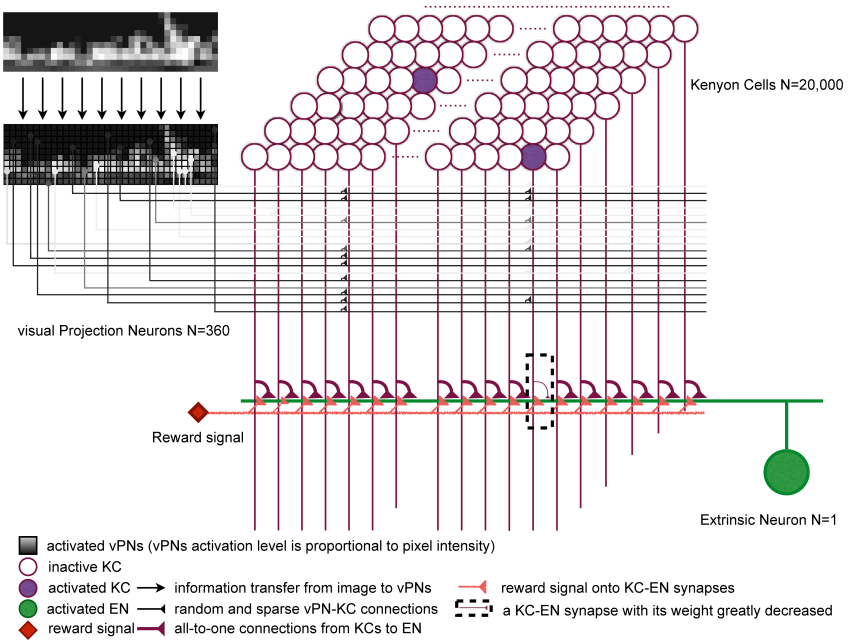

Figure 1: The Mushroom Body circuit: (Caption from *Ardin et al.*, Figure 2; note, their description and figure uses "EN" instead of "MBON"): Images (see Fig 1) activate the visual projection neurons (vPNs). Each Kenyon cell (KC) receives input from 10 (random) vPNs and exceeds firing threshold only for coincident activation from several vPNs, thus images are encoded as a sparse pattern of KC activation. All KCs converge on a single extrinsic neuron (EN) and if activation coincides with a reward signal, the connection strength is decreased. After training the EN output to previously rewarded (familiar) images is few or no spikes.

The MB model described above has been tested on the AntBot in three different works (albeit using different methods and metrics); the network as presented in [3, **?**, 7] differs only in the number of PNs present (900 vPNs are used in all three works), and the KC modelling (*Ardin et al.* use a spiking model[5] for the KCs [1]). More relevant to this work, is the proposed modification given by *Zhang*, whereby, 8 MBONs are present as opposed to 1 (see Section 2.4).

2.3 The Central Complex for Path Integration

The Central Complex (CX) is a highly conserved structure present in the insect brain[9, **?**]. Though the finer structural details and component position may vary, the basic composition is more or less the same across species [9]. Organization and function of the various parts of the CX are given by *Pfeiffer and Homberg* in [9][6]

We instead direct our attention to the CX model presented by *Stone et al.* which is the first neural model for path integration in the insect brain, with structure drawn purely from physiology. Interestingly, this model is a more advanced version of an earlier

[5]The AntBot projects do not explicitly model membrane potentials and realistic responses. They use a simple abstraction for the sake of complexity. The network function is essentially the same.

[6][DRAFT] I would like to include a short overview of the neurobiology but have left it until I can properly read through [9]. I think I will need to leave this out, however, as I'm not sure how to summarise the most relevant parts of their work.

model presented by *Haferlach et al.*, which was *evolved* using a Genetic Algorithm [6]. We present this also, purely for the sake of interest.

2.3.1 The Evolved Model for Path Integration

Prior to the CX model described below, there were several candidate neural networks proposed for PI in ants [6]. *Haferlach et al.* report the two best-known candidates being hand-designed, with more recent research into an evolutionary approach to network design [6]. The approach presented in [6] follows this trend and employs a Genetic Algorithm (GA) (see Appendix **??** for a brief overview) to evolve a neural model which is suitable for PI with biologically plausible inputs.

Haferlach et al. encode solutions as lists of integers. In these lists there are *start markers* and *end markers* which delimit the definitions of the actual neurons or sensors present in the model. The complete model (topography, connection weights, sensors, and effectors) is limited to a fixed-size encoding (genome limited to 500 parameters; 50 neurons maximum) [6]. An example encoding from [6] can be seen in Figure 2[7].

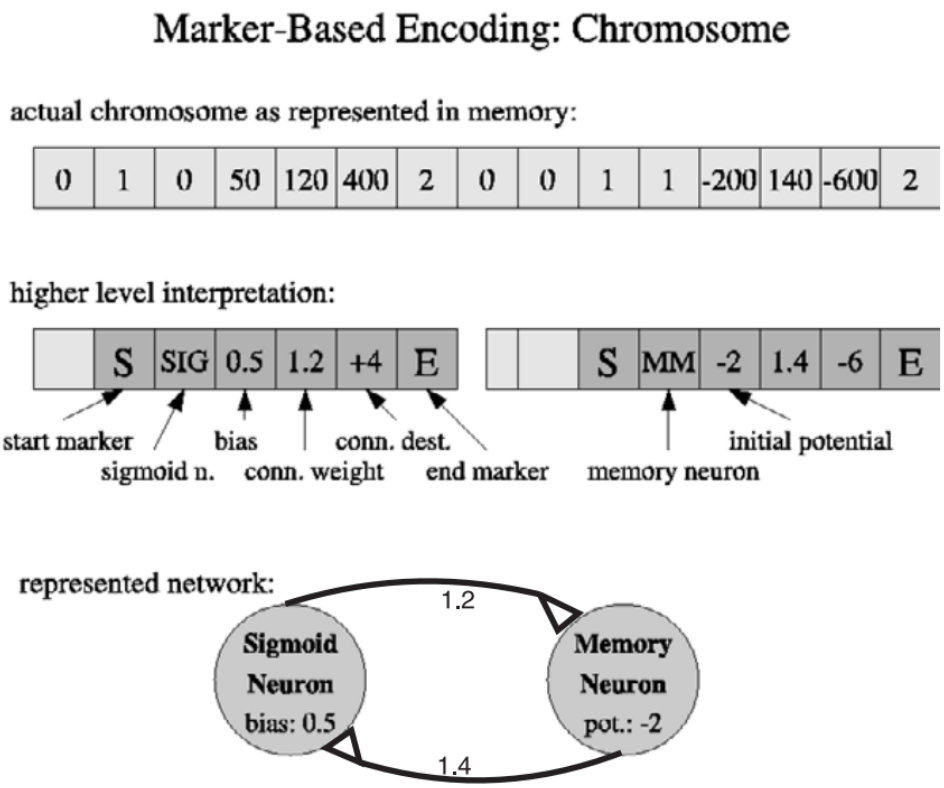

Figure 2: (Figure 2 from [6]) The Marker-Based encoding scheme used by *Haferlach et al.*, demonstrating how a simple neural model would be encoded as a sequence of integers.

[7] [DRAFT] This figure doesn't seem to explain how the connections are actually mapped. A connection destination is given for the Sigmoid Neuron as +4 but I have no idea how this actually converts to a connection. Along the same lines, the Memory Neuron gives -6 as its connection destination, but it connects back to the Sigmoid Neuron which only required +4 to reach the Memory Neuron. How were these mappings determined?

Glancing at the paper by *Fullmer and Miikkulainen, 1992* (cited as inspiration); they give a key to each neuron which identifies it uniquely, which doesn't seem toappear in the model in the Figure 2.

Localised *tournament selection*[8] is used to pick solution for transition into the next generation. Each solution is assigned a location in 1-dimensional space; a solution is picked randomly, then another solution within distance k of the first is picked for the tournament (k usually between 5 and 15) [6]. The solution with higher fitness wins the tournament and progresses to mutation and *crossover*[9] This localised selection allows the algorithm to find multiple *good* solutions which need not share the same genetic information.

The fitness of each solution is evaluated by having the agent navigate an outbound path through two randomly chosen points, then testing its ability to navigate back to the start point. During the outbound phase, information is input into the path integration network via the sensors; the network output is ignored. During the inbound route, the agent is steered by the network output. An individual is evaluated until some maximum time limit m is reached, and the distane to the start point $dist_t$ is computed at each time step. The fitness is then the inverse sum of squared distances [6]:

$$f = \frac{1}{\sqrt{\int_0^m dist_t^2 dt}} \tag{2}$$

The fitness is then augmented to penalise homeward routes which spiral (circling the start point, getting closer with each pass), and also to reward networks which are simple in structure; shown in Equations 3 and 4 respectively[10].

$$f = \frac{1}{\sqrt{\int_0^m dist_t^2 |\theta - \omega| dt}} \tag{3}$$

$$f = \frac{1}{(1+k_n)^{c_n}(1+k_c)^{c_c}\sqrt{\int_0^m dist_t^2 |\theta - \omega| dt}} \tag{4}$$

where ω is the nest heading, θ is the agent's heading, k_n is a neuron penalty constant, k_c is a connection penalty constant, c_n is the number of neurons, and c_c is the number of connections (i.e. the network is heavily penalised for the number of connections and neurons it has, forcing simpler networks) [6]. These are applied as a two-stage evolutionary process; evolve a solution first using Equation 3 to compute the fitness, then use this solution to seed a new population and evolve a solution from here using Equation 4. The two-stage evolutionary processalone did not find acceptable solutions. The search space was constrained by providing the following four constraints on network topology: Each direction cell had to be connected to at least one memory neuron; each direction cell had to be connected to at least one sigmoid neuron; sigmoid neurons had to be connected to turn effectors; a maximum of two turn effectors (left and right) were allowed [6].

Haferlach et al. present the model shown in Figure 3; notice the evolved structure bears a striking resemblance to the CX model presented by *Stone et al.* (Figure 4). Indeed, this network operates on the same principle; encoding a heading vector across multiple neurons with directional preference. This network provides a compact, elegant, robust

[8]Tournament Selection - Two solutions are picked randomly, then their fitness compared to determine if they will be selected

[9]Crossover - A particular method for breeding two solutions whereby their genomes are cut at a particular point and the ends swapped.

[10][DRAFT] I didn't have time to fix the equation formatting (equation number bumped out of position). This will be sorted in the final version.

model capable of performing path integration with reasonable errors. The network also demonstrated some capability for PI with obstacles present (though errors were generally much greater) [6].

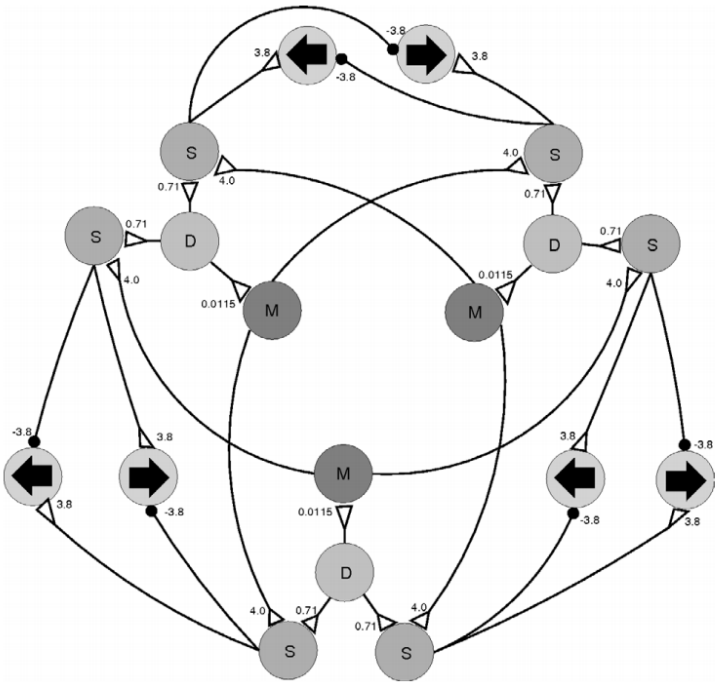

Figure 3: (Figure 3 from [6]) The high-fitness network, consistently evolved using the constrained two-stage evolution process illustrated by *Haferlach et al.*

2.3.2 The Central Complex Model

The Central Complex model is a six layer artificial neural network presented by *Stone et al.* which has been shown to provide a plausible neural substrate for Path Integration (PI) both in simulation and on the AntBot platform [10, **?**]. The model presented is shown in Figure 4. Splitting the model into its six layers, we get a breakdown of functionality:

- Layer 1: Heading preprocessing (TL), Speed (TN)
- Layer 2: Heading preprocessing (CL1)
- Layer 3: Heading (TB1)
- Layer 4: Memory (CPU4)
- Layer 5: Normalisation and Inhibition (Pontine Neurons)
- Layer 6: Steering/output (CPU1)

Figure 4 shows four types of neuron: TN (Tangential Neuron), TB1 (green), CPU4 (yellow and orange), and CPU1 (dark blue, light blue, and purple).

While Figure 4 (Left) shows a distinction between CPU1a (blue) and CPU1b (purple) neurons, we will ignore this distinction; for clarity, the physiological mapping between

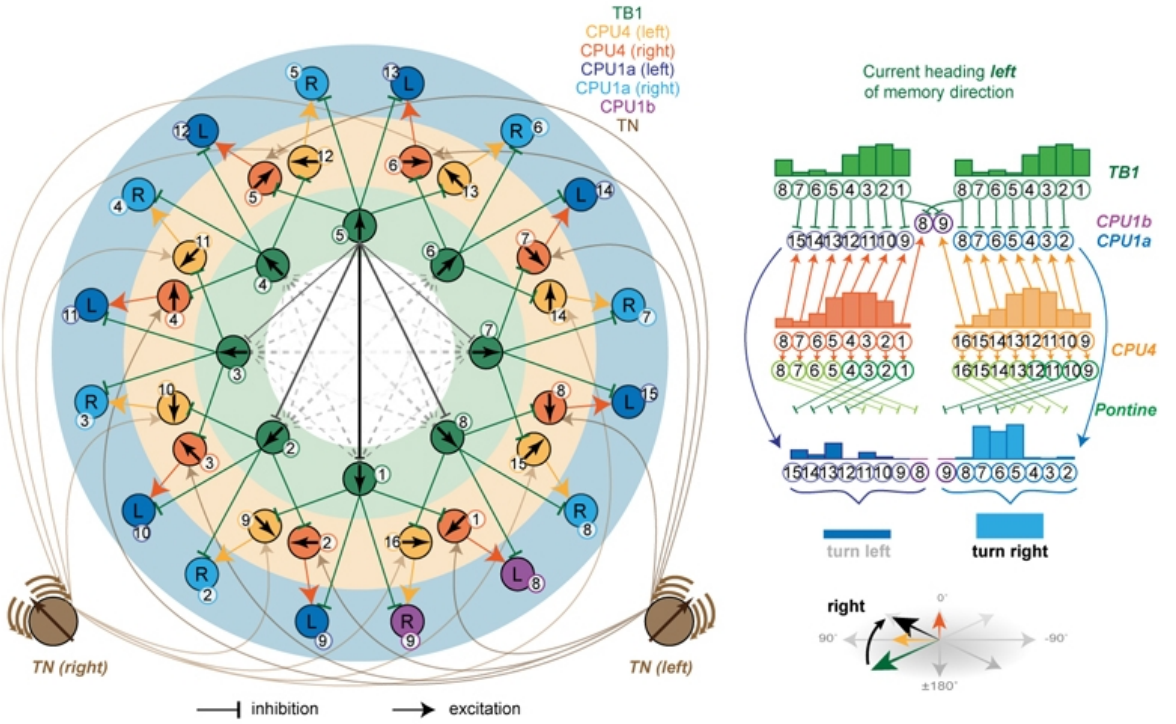

Figure 4: The Central Complex model presented by *Stone et al.*. (Left) This graph demonstrates the basic structure of the CX model (Figure 5G from [**?**]). Pontine neurons have been excluded for clarity. (Right) This graph shows how signals propogate through the network where the current heading lies to the left of the desired heading, i.e. a right turn should be generated (Figure 5I from [**?**]). The numbers given at each layer on the right correspond to the numbers given for each neuron in the graph on the left.

these CPU1 subtypes in the Upper Central Body (CBU) and the Protocerebral Bridge (PB) is different, but the function they serve in the Central Complex is the same [**?**]; thus, the distinction makes no difference in the model. Similarly the normalisation and inhibition function of the Pontine Neurons only has an effect when the agent experiences holonomic motion (motion where the view direction does not match the direction of travel); as AntBot is incapable of such motion, we can safely ignore the function of the Pontine Neurons also; in our case, the Pontine Neurons would have the same activity patterns as the CPU4 neurons [**?**]. Figure 4 (Right) shows how the Pontine inhibition is structured.

We now break down the different neuronal types and their proposed functions. Citation is given, but for the avoidance of any doubt, the following descriptions are adapted from *Stone et al.* (see STAR Methods) [**?**]. We also add implementation information for the AntBot where appropriate; occassionally the AntBot implementations are slightly different.

Simulated Neurons: Each neuron is described by its firing rate, where the firing rate r is a sigmoid function of the input I:

$$r = \frac{1}{1 + e^{-(aI-b)}} \tag{5}$$

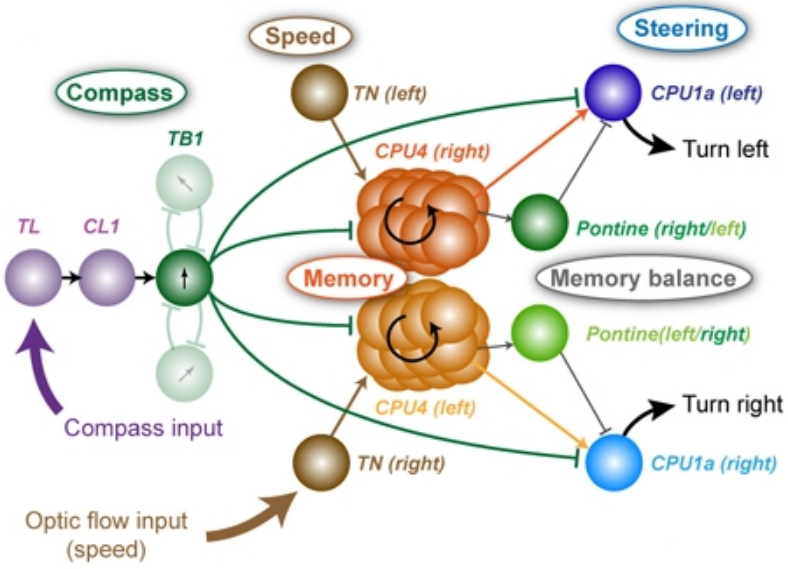

Figure 5: Here we can see the layers of the CX model and how they fit together. A heading signal is input to the TL neurons, propogating through the CL layer to TB1 (heading ring-attractor) and CPU4 (memory). TN neurons (speed sensitivity) input directly to CPU4. So, the combination of heading and speed inputs to CPU4 gives a measure of distance travelled in a particular direction; this facilitates generation of a steering command in CPU1 providing a mechanism for Path Integration.

where a and b are parameters which control the slope and offset of the sigmoid function [**?**]. Optionally, Gaussian noise may be added to the output. The input to each neuron is a weighted sum of the activity of each neuron that synapses onto it; say neuron j, the input is:

$$I_j = \sum_i w_{ij} \cdot r_i \tag{6}$$

The weights used by *Stone et al.* can only be 0, 1, or -1 for no-connection, excitatory, or inhibitory respectively [**?**].

TL & CL1: The TL neurons are the input point for heading information. In the ant brain this heading information comes from a *sky-compass*; the ant eye contains cells sensitive to polarized light which allows the ant to infer an accurate, allothetic direction from vision. Interestingly, there is also evidence that ants have the capability to infer a direction without a view of celestial cues, suggesting they may have access to some other signal, the candidate signal being the geomagnetic field [4, 5]. The TL neurons have been shown to be polarisation sensiive across multiple insect species [**?**] (see STAR Methods). Each TL neuron has a preferred direction (i.e. a specific direction of poliarisation sensitivity) θ_{TL}, and there are 16 such neurons representing the 8 directions around the agent (i.e. $\theta_{TL} \in \{0°, 45°, 90°, 135°, 180°, 225°, 270°, 315°\}$) [**?**]. Together, the 16 TL neurons encode the heading of the agent in a single timestep; each neuron receives input activation as:

$$I_{TL} = cos(\theta_{TL} - \theta_h) \tag{7}$$

where θ_{TL} is the preferred heading of the neuron as above, and θ_h is the current heading

of the agent. In the next heading layer, there are 16 CL1 neurons which use inhibition to invert the polarisation response [**?**]. *Stone et al.* comment that these neurons effectively make no difference to the model and are included for completeness. They are also included in previous AntBot projects which make use of the CX model [**?**, 10]. On AntBot, the heading is derived from the onboard compass on the mobile phone (see Section 3).

TN: There are 4 TN neurons which act as an input for speed information. The TN neurons are sensitive to optical flow, and can be split into two subtypes: TN1, and TN2. *Stone et al.* showed that TN1 neurons are inhibited by simulated forward flight and excited by simulated backward flight, while TN2 neurons are inhibited by simulated backward flight and excited by simulated forward flight [**?**]. Each of the four TN neurons has a tuning preference; these tuning preferences were measured in bees as approximately $+45°/-45°$ for TN2, and $+135°/-135°$ for TN1 (where $0°$ is straight ahead). In short, we have TN1_{left}, TN1_{right}, TN2_{left}, and TN2_{right}. It is thought that these neurons provide a (or part of a) mechanism for odometry by allowing the model to integrate speed with respect to time giving a distance measure [**?**]. *Stone et al.* give the speed calculation from the TN neurons as:

$$I_{TN_L} = [sin(\theta_h + \phi_{TN}) \quad cos(\theta_h + \phi_{TN})]\mathbf{v} \tag{8}$$

$$I_{TN_R} = [sin(\theta_h - \phi_{TN}) \quad cos(\theta_h - \phi_{TN})]\mathbf{v} \tag{9}$$

where $\mathbf{v}$ is the velocity of the agent in Cartesian coordinates, $\theta_h \in [0°, 360°)$ is the current heading of the agent, and ϕ_{TN} is the preferred angle of that TN neuron [**?**].

The TN neurons are not modelled in the *basic* CX model on AntBot. There is, however, a *holonomic* implementation which models both the TN1 and TN2 neurons is present on the robot. The formulae above are implemented on the robot but they are never used. Instead, the raw left and right speeds computed from the optical flow are passed in (see *Scimeca* [10] for the details of speed computation from optic flow). The TN2 neurons will simply clip these speeds to make sure they lie in $[0, 1]$; the TN1 neurons will perform $(1 - s)/2$ for both the left and right speeds s, then clip the output to lie in $[0, 1]$. The output is returned directly, rather than being returned as a sigmoid (as for all other neuron types).

TB1: There are eight TB1 neurons, each with a directional preference θ_{TB1}, which correspond to the eight cardinal directions in the model. Each TB1 neuron receives excitatory input from the pair of CL1 neurons that have the same directional preference. The TB1 layer contains inhibitory connections between peer neurons where each TB1 neuron strongly inhibits other TB1 neurons with opposite directional preferences (see Figure 4) forming a *ring attractor*[**?**]. The weighting for an arbitrary inhibitory connection from neuron i to neuron j is given by:

$$w_{ij} = \frac{cos(\theta_{TB1,i} - \theta_{TB1,j}) - 1}{2} \tag{10}$$

where $\theta_{TB1,i}$ is the direction preference of TB1 neuron i (similarly for $\theta_{TB1,j}$). The total input for each TB1 neuron from the CL1 layer at timestep t is:

$$I_{TB1_j^{(t)}} = (1-c) \cdot r^{(t)}_{CL1_j} + c \cdot \sum_{i=1}^{8} w_{ij} \cdot r^{(t-1)}_{CL1_j} \tag{11}$$

where $c = 0.33$ is a scaling factor which determines the relative strength of excitation from the CL1 layer and inhibition from other TB1 neurons; the AntBot implementation of the TB1 neurons is the same. This network layer produces a stable heading encoding which provides accurate input for the CPU4 layer, underpinning accurate path integration.

CPU4: The 16 CPU4 neurons receive input in the form of an accumulation of heading $\theta_h^{(t)}$ of the agent, along with a modulated speed response from the TN2 neurons [**?**]. The CPU4 neurons accumulate distance with direction. The input the CPU4 neurons is given by:

$$I^{(t)}_{CPU4} = I^{(t-1)}_{CPU4} + h \cdot (r^{(t)}_{TN2} - r^{(t)}_{TB1} - k) \tag{12}$$

where $h = 0.0025$ determines the rate of memory accumulation, and $k = 0.1$ is a uniform rate of memory decay [**?**]. All memory cells are initialised to $I^{(0)}_{CPU4} = 0.5$ and are clipped at each timestep to fall between 0 and 1 [**?**]. As shown in Figure 4, each TB1 provides input to two CPU4 neuron, each of which receives input from the TN2 cell in the opposite hemisphere. As these neurons accumulate distance with respect to a direction, they provide a population encoding of the *home vector* (the integrated path back to the nest) [**?**]. Interestingly, *Zhang* showed that the network can be initialised to an arbitrary state, allowing the agent to navigate along arbitrary vectors [**?**]. While this seems intuitive, the experimental evidence is valuable and demonstrates that the CPU4 layer could form a basis for the *vector memory* discussed by *Webb* in [**?**] (see Section 2.4).

The basic CX model on the AntBot does not model TN neurons, so the input is computed differently; furthermore, the gain and loss factors are different. The input can be expressed as:

$$I^{(t)}_{CPU4} = I^{(t-1)}_{CPU4} + s \cdot ((1 - r^{(t)}_{TB1}) \cdot g - l) \tag{13}$$

where $l = 0.0026$ is the uniform rate of memory loss, $g = 0.005$ is the uniform rate of memory gain, and s is the current speed. This value is then clipped to lie in $[0, 1]$. As an aside, in the holonomic model mentioned previously, the input is computed as in Equation 12 as the TN neurons are present in the model.

Pontine: The pontine neurons project contralaterally connecting opposite CBU columns (shown in Figure 4 (Right)). The 16 pontine neurons each receive input from one CPU4 column [**?**]; pontine input can be given simply as:

$$I^{(t)}_{Pontine} = r^{(t)}_{CPU4} \tag{14}$$

The pontine neurons are not implemented on AntBot for simplicity. AntBot is incapable of holonomic motion, so they would have no functional effect.

CPU1: There are 16 CPU1 neurons present in the model. Each of which receives inhibitory input (weight $= -1$) from a TB1 neuron; where, each TB1 neuron provides

inhibitory input to two CPU1 neurons (in the same pattern as the TB1-CPU4 connections - see Figure 4). Each CPU4 neuron also provides input to a CPU1 neuron (so we get input from vector memory and current heading). The CPU1 input can be expressed as:

$$I_{CPU1}^{(t)} = r_{CPU4}^{(t)} - r_{Pontine}^{(t)} - r_{TB1}^{(t)} \tag{15}$$

As can be seen the CPU1 neurons also receive input from the pontine neurons. The CPU1 neurons form two sets, those connecting to left motor units and those connecting to the right. On AntBot the pontine term is omitted (i.e. the input becomes $I_{CPU1}^{(t)} = r_{CPU4}^{(t)} - r_{TB1}^{(t)}$).

We choose a direction simply by summing the CPU1 outputs on both sides and the difference indicates the direction and angle of the required heading correction:

$$\theta_h^{(t)} = \theta_h^{(t-1)} + m \cdot (\sum_{i=1}^{8} r_{CPU1R_i} - \sum_{i=1}^{8} r_{CPU1L_i}) \tag{16}$$

where $m = 0.5$ is a constant [**?**]. The angle computation is the same on the AntBot.

There are some details which have been omitted for clarity. The stack presented should effectively describe how the network generates a turning signal from its current state and inputs to perform effective, accurate path integration. For further details, please consult [**?**] (the STAR Methods section contains the full technical details of the model).

2.4 The Eight MBON Model (CXMB)

In [**?**], the MB network is modified by adding 7 MBONs. Each MBON has its own KC-MBON connection array, each with their own unique weights. Each connection array corresponds to one of the eight cardinal directions represented by the TB1 layer of the CX model (namely, 0°, 45°, 90°, 135°, 180° ,225°, 270°, 315°) (see Section 2.3).

Image memory now has an associated direction, so, training is performed with respect to orientation. For example, if the agent has a heading of 45° when a image is stored, then only the corresponding connection array has its weights updated during learning. Practically, this is done by querying the TB1 layer of the CX model to find the current direction according to the model (rather than directly querying the robot's onboard compass). This process is visualised in Figure 6.

While, in theory, this eight MBON model could function independently, *Zhang* uses it to (rather neatly) augment the CX model; this allows navigation to be performed using a combination of visual memory and path integration information. The navigation process can be described by a sequence of equations. We modify the notation slightly for clarity, but the equations are the same as presented in [**?**].

The MB circuit is shown an image in the usual way; however, we now get eight responses, giving us a response distribution. This distribution can be interpreted as giving us the most likely direction of travel, when the image presented was first observed. This distribution requires some modification to integrate it into the CX response. Let M_i be the familiarity response of the ith MBON; the responses are normalised as:

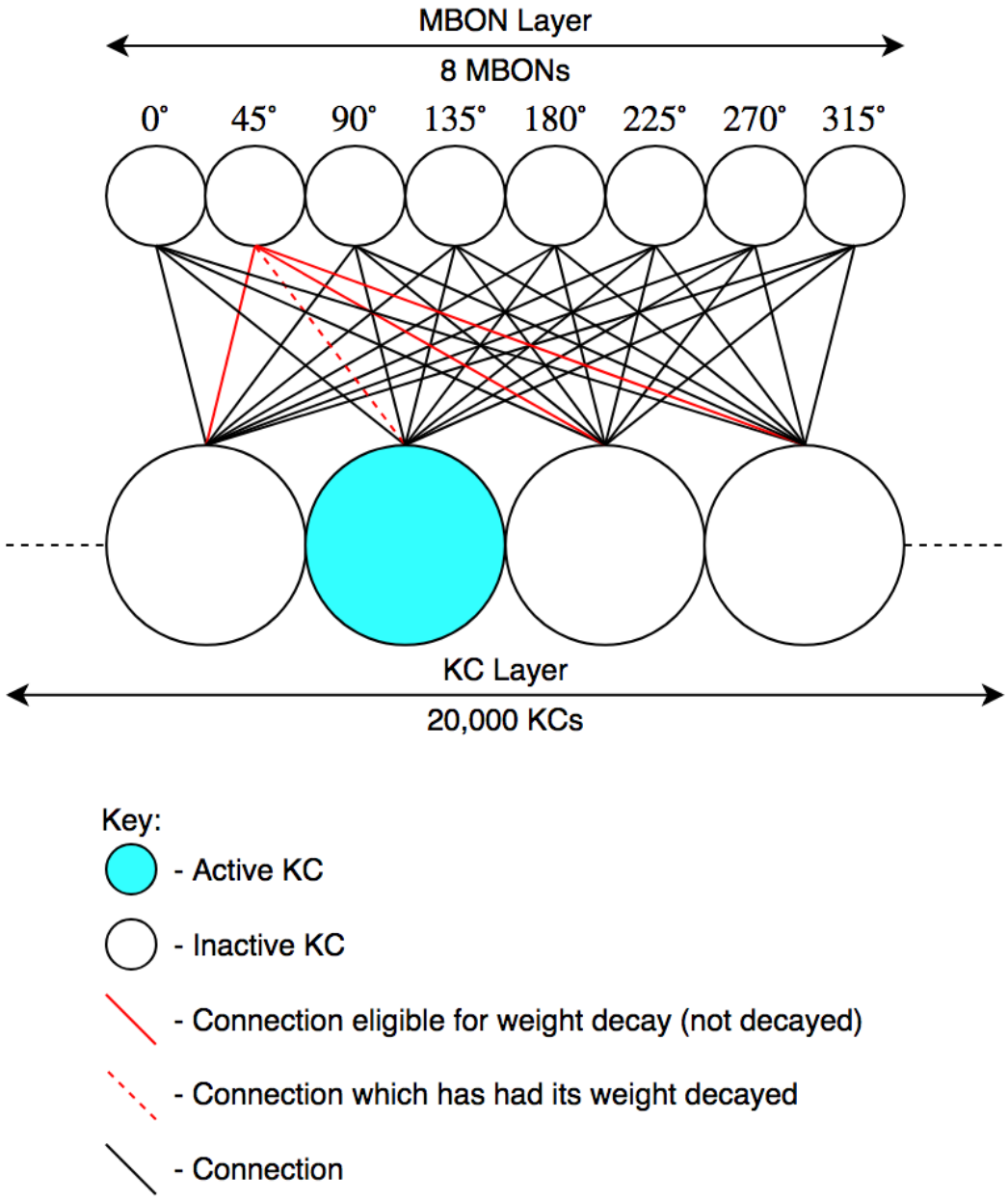

Figure 6: Our interpretation of the eight MBON model proposed by *Zhang*. Every KC connects to every MBON. All connection weights start out at $w = 1$. Following the example presented in the text, if an image being learned corresponds to facing a direction of 45°, then only the conncetions to that MBON (highlighted in red) are eligible to have their weights modified. Recall, however, that these weights will only be modified if the KC was activated (not shown in the figure).

$$\bar{M}_i = \frac{M_i}{\Sigma_{k=0}^{8} M_k} \tag{17}$$

Which gives us the normalised response $\bar{M}_i$ between 0 and 1. $\bar{M}_i$ must be inverted so that the most likely direction has the greatest response (in the MB model, the most familiar direction would give the lowest response):

$$\bar{M}_i^{-1} = 1 - \bar{M}_i \tag{18}$$

This visual response is then combined with the memory response from the CPU4 layer of the CX model to give output at the CPU1 layer:

$$CPU1_{output} = k \cdot W_{CPU4} \cdot CPU4 + (1 - k) \cdot W_{MBON} \cdot \bar{M}^{-1} \tag{19}$$

where k is a weighting factor that determines the relative stregths of the CX response and the MB response in the output ($k = 0.8$ in [**?**]), $\bar{M}^{-1}$ is the collection of all inverse normalised MBON responses, W_{CPU4} is a custom matrix[11], W_{MBON} is an identity matrix that will expand the MBON response array from 8x1 to 16x1 [**?**].

In oreder to test the CXMB model, *Zhang* first demonstrated the functionality of a *copied memory model*. The test of the copied memory model aimed to prove that the CPU4 state of the agent could be copied and stored, the CPU4 state modified, and then restored from the copy to allow the agent to navigate home. The copied memory model was tested by sending the agent on a pre-determined oubound route to a feeder (chosen at random, but consistent between trials), copying the CPU4 state, and allowing the robot to navigate home using the CX model; the CPU4 state will be modified by this homeward navigation. The agent is then replaced at the feeder, its CPU4 state restored from the copy, and tasked with navigating home a second time. The copied memory model was tested as a pre-requisite for testing the CXMB model, however, it demonstrates an important capability of the CX model; namely, the capability to directly load a state into its memory in order to navigate; a concept referred to as *vector memory* in [**?**]. Indeed, this concept of vector memory is shown to be quite a useful tool in insect navigation [**?**]. It is worth noting that, the precise biological mechanism employed by insects to perform this task, is unknown at time of writing. It would be neat to reuse the CX circuitry, but this is an open question.

The CXMB model is then tested in the same way. The agent follows its oubound route to the feeder, navigates back once using the CX model (the MB model is trained on this first inbound trip), is replaced at the feeder, and finally, tasked with navigating back again, this time using the CXMB model. To be clear, the CPU4 state of the CX model is stored after the outbound route, and loaded back into the network before the second inbound trip. *Zhang* reports that the second inbound trip (using both CXMB) showed more heading adjustments during its traversal, and, on average, performed slightly better than a pure CX implementation[**?**]. It should also be noted that the average performance of both models in *Zhang's* work was good [**?**].

We note two methodological problems with *Zhang's* work: The outbound route, while randomly chosen, was the same accross all trials, and, for the copied memory and CXMB experiments, the agent was placed facing the nest for the test run (second inbound). To add to the first problem, the outbound route ended such that the robot was facing back towards the nest. Both of these flaws could lead to apparent path integration behaviour where, in fact, the agent could have made it sufficiently close to the nest by simply following a straight line. While the agent was certainly employing the CX to perform path integration, we do not feel that the tests chosen provide strong evidence of functionality.

[11] This is the term used by *Zhang* to describe the W_{CPU4} matrix. More specifically, this matrix describes the connections between the CPU4 and CPU1 layers of the CX model.

2.5 Review of Part 1

The work in [7] specifically covers the Mushroom Body and Optical Flow Collision Avoidance. The Mushroom Body model used in [7] is the same as that used by *Ardin et al.* in [1], except for the small differences noted in Section 2.2. The model tested in [7] used a single MBON with a scan-based route following strategy. The scanning differed from previous works as it was implemented as an image manipulation algorithm (termed *visual scanning*) instead of a physical turn performed by the AntBot.

Multiple OFCA systems were tested, however, in final experiments an optical flow filtering system is used. In short, a pattern of expected motion is created, then the actual observed motion is projected on top of it. An absolute difference is computed between the two and this can tell us if part of the image is moving faster than we expect. This can be used to detect obstacles. This strategy proved simple, but effective. While we move away from it for the bulk of this project, it was used for initial tests of the CX model, as it provided a simple out-of-the-box method to generate a non-deterministic path for the model to integrate.

Both systems functioned well and provided a solid baseline from which we will work in this project. The MB model proved very capable at following routes learned in a non-deterministic fashion through a cluttered environment.

3 Platform

The AntBot is a small autonomous robot built to allow easy testing of the various navigational algorithms in the *Ant Navigational Toolkit* on a physical agent. AntBot is assembled using off-the-shelf parts with an Android phone at its heart. We give a high level description here; additional detail can be found in [7].

3.1 Hardware

The robot is composed of three main parts: a Dangu Rover 5 chassis, an Arduino, and a Google Nexus 5 Android smartphone. The smartphone provides a (theoretically) solid base for an autonomous agent, with sufficient processing power and memory to run the models, a convenient touchscreen interface to display information and an adaptable control interface, and a built-in camera and extensive software libraries. As such, the smartphone works as the computational centre, performing all higher sensory processing and acting on this analysis. Once the algorithm running on the phone has decided on a course of action, it sends a command via a serial connection to the Arduino board; the board, then parses the command and translates it into a sequence of motor commands which are sent to the built-in motor board in the chassis.
Additionally, the robot possesses a 360° camera attachment, giving the robot a panoramic view to match that of the desert ant[12]. The robot was also modified as part of [7] to allow it to be charged with an external charger without dismantling it.

3.2 Software

The robot requires two levels of software; higher level Java (Android) code to perform the high level functions; and, lower level code written in C/C++ (Arduino). The Arduino code can be thought of as the firmware.

3.2.1 Android

Android can provide a nice basis for working on robotics projects. The Operating System permts multiple applications to run asynchronously and pass information between them; for example, the user could develop an application to perform all visual processing then broadcast a processed image for other applications which require it. The original AntBot software was split into separate five applications: Ant Eye, Visual Navigation, Path Integration, Combiner, and Serial Communications, neatly compartmentalising each task. Unfortunately, in recent years, this structure was abandoned and almost all code is contained within the *AntEye* application. Only the Serial Communications application remains true-to-design.

Typical flow within AntEye will see an image captured from the front camera, processed such that we get a downsampled 90x10 image which represents the 360° view around the robot. This image is then used for optical flow analysis. The image and flow analysis are then made available to any thread which requires it. A thread will then use this data to run an experiment.

There are a lot of criticisms we can raise about the robot (in both a hardware and software sense); ultimately the key problem may be lack of developer familiarity with

[12] In reality, the desert ant has a visual field of only 280°[1], however an approxmiation of 360° was considered appropriate when the robot was built.

Figure 7: The AntBot, connected to the charging station. This figure also shows the position of the mobile phone, camera attachment, and retroflective motion capture markers on the top of the robot.

Figure 8: The Kogeto Dot 360° panoramic lens attachment.

Figure 9: A sample view of the lens from the front facing camera, before any processing.

the Android platform. While Android seems to work well at first glance, it can ultimately be quite restrictive; we leave this to the later discussion.

3.2.2 Arduino

The Arduino code acts as a bridge between the Android platform and the Dangu motor board, as there are no libraries which will directly connect the two. The Android code contains a Command class which allows the user to insert robot commands into the code; these are then transformed into serial messages by a broadcast library [3] and sent to the Arduino. The Arduino code contains a parser which will decode the serial messages and then call the correct method to execute the desired motor commands.

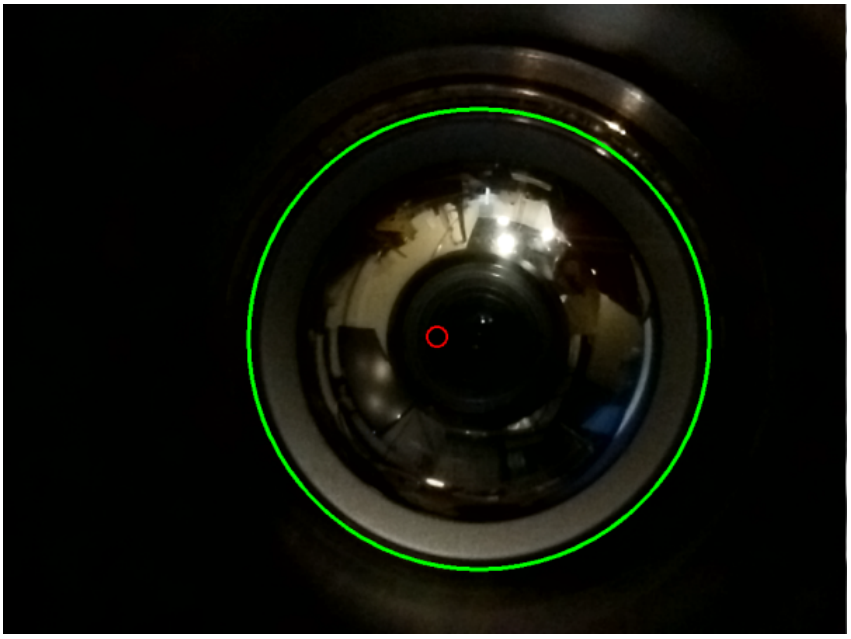

Figure 10: The hard-coded centre used for the polar transform (red), against the circle detected using the Hough transform (green).

3.3 Modifications and Development

The modifications or additions listed below are a small part of the dissertation, however, for one reason or another (see Discussion) these ended up taking most of the project time.

Compass Sensor: While the Android phone has a built-in compass (used for other parts of this project) this is impractical to use as part of a control system. The communications delay is too great to allow accurate feedback control. As a step to introducing proportional control to the robot, we undertook an investigation into available compass sensors. Multiple sensors and libraries were tested and ultimately we settled on a Grove 6-Axis Compass & Accelerometer. This sensor remained reasonably accurate in when tested alongside motors (a source of magnetic interference) and the library code was straightforward to include. The sensor was installed by the Informatics Workshop. While we have not used it for this project, we would encourage a future student to make use of this.

Ocular Calibration: The 360° camera attachment is held in place by a pressure sensitive adhesive which allows the position to shift if the attachment is knocked or removed. The lens was removed for use in another project, and on re-attaching, it was noticed that the image was warped, indicating that the pre-processing was not working as required. The image pre-processing algorithm used a hard-coded pixel value for the centre of the camera attachment which was no longer correct (see Figure 10). We implemented a simple detection algorithm using the Hough transform available in OpenCV to allow the user to detect the new position of the camera attachment and set the centre accordingly. The calibration is not performed automatically as the Hough transform can be slow, and is highly dependent on lighting; instead, it need only be done if the attachment is known to have moved.

Code Refactor: As mentioned above, all behavioural code for the robot has moved

into the AntEye application. In fact, most code had been integrated into a single Android Activity (Java Class); this file contained multiple long threads dictating behaviour as well as a number of utility methods and robot commands. There were also a ludicrous number of global variables. The resulting file was over 4000 lines long and almost unusable. Particuarly aggregious was the fact that roughly half of the code was irrelevant to this project, but did need archived for reference purposes. As such, it was decided that it was worth refactoring the codebase in order to make it easier to work. All utility functions were split into their own static Util class, similarly, the robot control commands were moved to the Command class. Runnable threads were moved into an archive package and split into classes according to their use: Mushroom Body, Central Complex, Optical Flow, or Old Navigation; that final group consists of the oldest code which is not used explicitly but is useful for reference purposes. A single thread was left in the main Activity file to act as a test thread. A lot of unused code was removed entirely.

There are still many problems with the existing codebase. This simple refactor ended up taking up a lot of time due to complications from working within the Android framework; furthermore, many concessions had to be made for the same reason.

Video Recording: In order to analyse the visual processing in more detail, a video recording utility was added to AntBot. Again platform constraints mean that the utility is perhaps not as well-made as it could be. A straightforward method using FFMPEG had to be abandoned due to library conflicts; this method would have allowed a video file to be recorded directly rather than the frame-stitching solution described below. The real reason this proved difficult was that the recording had to take place after preprocessing (image unwrapping and downsampling to the 90x10 360° image). In order to achieve this frames are passed off to a recording buffer after processing, the buffer is then processed asynchronously by a separate thread which saves the frames to a set directory. Pleasantly, this does not impact the frame rate. The video frames then have to be retrieved from the robot and stitched together using a custom python utility (a simple wrapper around an FFMPEG command). The recorded video can then be analysed offline. This utility was created to offline analysis of optical flow, however, it also highlighted a major problem with the robot which was not previously known.

Video Pre-processing Pipeline: When stitching the video frames from the recorder, it was noticed that the framerate was not quite right. The framerate on the robot was found to be an unacceptably low 2fps. In an effort to improve the framerate we isolated different parts of the processing pipeline. We found the best possible framerate to be approximately 14fps with no processing (this is computed simply as the numebr of times the *onCameraFrame()* callback function is called per-second). Adding processing steps back to the pipeline, two steps were found which caused the framerate to crash. The first step was an image convolution algorithm; the unwrapped image does not line up with the direction of travel of the agent, so the resulting frame must be shifted (azimuth) so that the centre of the frame is in line with the forward direction. This step was reducing the framerate by approximately 5fps. Replacing the implemented algorithm with an OpenCV library implementation reduced this to approximately 1fps cost. Similarly, an image downsampling algorithm had been manually implemented which cost around 5fps; again, replacing this with an OpenCV library implementation reduced the cost to approximately 1fps, if that. The full pipeline can now run at 10-12fps with the recording utility running on top. When simulating a model like the

ECX, the framerate does drop to 8-10fps. We also attempted to move the frame processing into its own high-priority thread, however, this resulted in numerous bugs and did not improve the framerate in any noticable way.

This raises an interesting problem. We do not know when this code was added to the robot but in any project following its insertion, the framerate surely affected the performance of the agent. We can say for sure that the algorithms used and tested in [7] were affected. Indeed, in [7] it is noted that few images were stored during Visual Navigation experiments. The low storage rate was thought to be a thread synchronisation issue, but it is now clear that the frames were simply not finished processing in time to be stored. In testing the matched-filter collision avoidance system from [7] we note a significant change in behaviour after the framerate improvement. It is suspected that this is due to an increase in flow information, and therefore an increase in noise which can cause erroneous and erratic behaviour, not witnessed in [7].

4 Methods

4.1 Collision Avoidance

The AntBot cannot be fitted with sensors which feedback to the higher models (see Discussion). We therefore continue to look for a method of collision avoidance which can be integrated into the Central Complex which uses the available visual information. The matched-filter model from [7] performed reasonably consistently but was initially considered too coarse-grained for integration into a modified CX. Our first approach therefore looked for expansion patterns in certain areas of the flow field.

4.1.1 Shifting Expansion Fields

We noted in research that the Focus of Expansion is used in many visual collision avoidance systems. Usually it is used as a stepping stone to compute the time-to-contact with an obstacle. Some papers describe using the FOE to determine a turn direction in the event a collision is determined [**?**, **?**]. We also note that the often the FOE seems to shift to the deepest part of the image [**?**, **?**] (though we should also note that in [**?**] an avoidance saccade is triggered *away from* the FOE).

If the FOE *is* drawn to the deepest part of the image then its location could be reliably used to choose a steering direction in a collision avoidance system. Furthermore, this has an intuitive integration into the CX. If we split the 360° image seen by AntBot into eight equal sections reflecting the eight cardinal directions of the CX model, we can use the horizontal position of the FOE as an input signal. The position would mark a "bump" of activity in a set of eight neurons (similar to the output layer of the eight MBON MB model). Ultimately we wish to see if the FOE will behave predictably in the presence of a looming obstacle, which relies on being able to compute the FOE reliably. When working with the FOE, [7] notes difficulty in computing it consistently. Often nonsensical values would be output by the computation. Here however, we managed to compute reasonably sensible values, though they were not consistent or predictable. To compute the FOE we use the OpenCV *calcOpticalFlowFarneback()* function to produce a dense optical flow field then use the computation from [8] (see Section **??**).

While fixing the problems raised by [7] made the computed FOE more sensible in terms of general location, the lack of consistency was cause for further investigation. For this purpose, the video recording tool described in Section 3.3 was developed with the goal of performing offline video analysis of the agent's point of view. Thus, we can visualise the computed optical flow field observed by the robot under controlled circumstances (see Figure 11).

Figure 11: A subset of points from the dense optical flow field observed by the agent. The agent is experiencing forward translational motion. These frames were captured after the framerate improvement from Section 3.3. The FOE is also shown in green.

This analysis produced the following observations:

1. The optical flow field is incredibly noisy: The majority of the flow produced is erratic and unpredictable.

2. The amount of perceived motion is tiny: In simply looking at the video it is

incredibly difficult to determine how the motion observed relates to the movement experienced by the camera.

3. The FOE jumps randomly between frames: The FOE is not consistent between frames though it does appear to stay in the same region (though we are not sure this indicates anything significant). We cannot rely on its location when presented with a looming object in the frame.

4. Looming objects can be identified from the flow field: This observation explains the good performance seen by [7] when examining a matched-filter system. Looming objects can appear as a large (often single frame) disturbance of the low-level noise produced by the flow field.

We can therefore conclude that optical flow cannot be used to compute any fine-grain properties. The flow field at large is just noise, and only drastic disturbances result in a large disturbance in a specific region. While this effectively closes the lid on any further investigation into the FOE/depth methods which utilise optical flow, it does give us one semi-predictable property of the flow field which can potentially be used.

4.1.2 Matched Filters

The matched-filter model of collision avoidance as seen in [7] was used without modification for the experimental results in this paper. However, in an effort to establish the ECX model, we present here an adaption of matched-filter collision avoidance which can have its outputs integrated into the CX. We will use the language of artifiial neural networks, but the principle is essentially the same as [7].

The neural model for collision avoidance features two types of neuron, ACC (ACCumulation) and MRSP (Movement ReSPonse). ACC neurons are key in a working system, so we spend most of the time discussing them.

ACC Neurons: ACC neurons take on the role of the leaky accumulators from [7, **?**]. The system contains two ACC neurons, ACC_{left} and ACC_{right} for the left and right sides respectively. We investigate three different representations of these neurons: Rate, Leaky Integrate & Fire (LI&F), and Reset Integrate & Fire (RI&F). The two I&F variants are only subtly different and behaved largely the same but we include them for completeness. We will go through them in chronological order.

The matched-filter system produces a sum for the left and right hand sides of the robot. This sum denotes the difference between the expected motion and the observed motion in the image frame. If we take the difference between the two sums we can see if one side has experienced a significant deviation from the expected motion (i.e. a looming obstacle on that side). This difference acts as the input to the ACC neurons.

ACC Neurons (LI&F): For both I&F models, we need to set a lower bound on what we consider an input. Each input must be greater than (absolute value) a lower threshold in an attempt to reduce noisy inputs to the ACC neurons. For more details as to the filter matching, please see [7]. Once the input has been thresholded, an ACC neuron at time t can be described as:

$$V(t) = \gamma \cdot V(t-1) + I_{ext} \tag{20}$$

where $V(t)$ is the membrane potential, t is the current timestep, γ is the leak current and I_{ext} is the external current flowing in - the thresholded difference of flow sums. We use terms like membrane potential and external current because these best describe the function in the language of simulated neurons. It should be noted by the reader that we do not explicitly model voltages and currents in the model though this is just a case of semantics; these properties could be modelled more explicitly.

We also define V_{fire}, the value at which the ACC neurons fire. If $V(t) > V_{fire}$ for some ACC neuron, then the neuron that exceeded V_{fire} produces some output signal and both have their membrane potential reset to zero (the resting potential). The neuron that fires then governs the signal produced by the MRSP layer.

ACC Neurons (RI&F): The RI&F representation is almost identical. The difference arises from the leak model. In RI&F we reset the neuron periodically (e.g. every third timestep). While this is functionally and I&F neuron, it does not function in any biologically plausible manner. It is essentially the same as the leaky accumulators from [7]. We can describe the RI&F neuron at time t by:

$$V(t) = \begin{cases} V(t-1) + I_{ext} & \text{if } t \not\equiv 0 \bmod p \\ 0 & \text{otherwise} \end{cases} \tag{21}$$

where $V(t)$, I_{ext}, and t are as above and p is the reset period (for example, if $p = 3$ the membrane potential is reset every 3 timesteps). Firing behavriour is the same as the LI&F model.

ACC Neurons (Rate): While I&F neurons provide an intuitive neural metaphor for the leaky accumulators, the CX model (into which this CA system is being built) uses a rate model for all other neurons (including our own MRSP neurons). As such we felt it appropriate to investigate such a representation for the ACC layer. In fact, using a rate representation requires only a single ACC neuron and allows for the generation of a steering response which is proportional to the size of the optical flow disturbance (in theory, the bigger the obstacle or loom effect, the bigger the response); neither of which are possible with the I&F representation. Recall, a firing rate given some input I is represented as a sigmoid:

$$r = \frac{1}{1 + e^{-(aI-b)}} \tag{22}$$

where a and b determine the *slope* and *bias* of the function respectively. If we set $a = 5$, $b = 0$ we get the rate function shown in Figure 12. Importantly, we must scale down the input (difference of flow sums) such that it lies in (-1,1).
If the flow disturbance observed on both sides is equal, then the rate response should lie around 0.5. A looming object detected on the right-hand side results in a negative input drawing the output closer to 0. Conversly, a looming object detected on the left-hand side wil result in a positive input, driving the output closer to 1. We have, in a roundabout way, ended up with an optical-flow balance strategy [11].

4.2 Path Integration

4.3 The Complete System

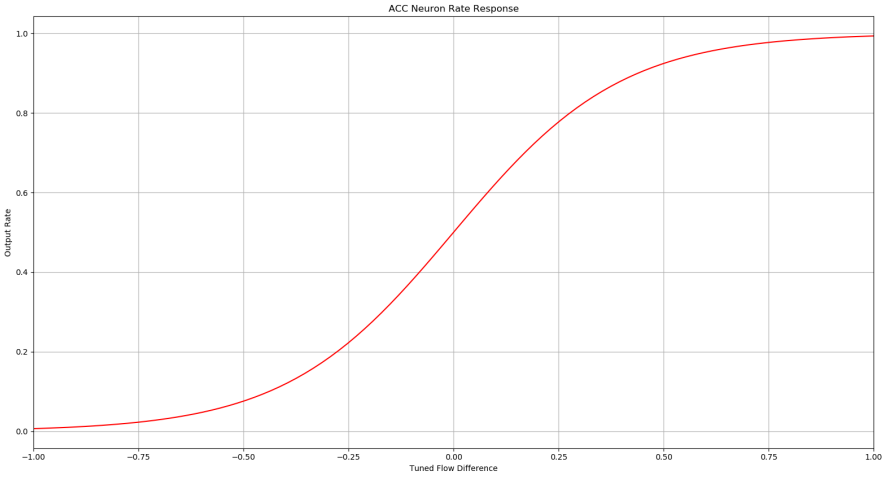

Figure 12: Function of firing rate response of the single ACC neuron against difference input; sigmoid with $a = 5$, $b = 0$. The input to this neuron is scaled down to lie between -1 and 1.

5 Experimentation

Experimentation methods not final, anything in this section illustrates a rough plan only.

5.1 General

5.2 Collision Avoidance

5.3 Visual Navigation

5.4 Path Integration

6 Results and Evaluation

Included for skeleton purposes.

7 Discussion

Included for skeleton purposes.