ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Lecture with Computer Exercises:
Modelling and Simulating Social Systems with MATLAB

Project Report

Desert Ant Navigation

Agata Kazmierczak

Zürich
14.12.2012

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of Originality

This sheet must be signed and enclosed with every piece of written work submitted at ETH.

I hereby declare that the written work I have submitted entitled

____

is original work which I alone have authored and which is written in my own words.*

Author(s)

Last name	First name
Agata	Kazmierczak

Supervising lecturer

Last name	First name

With the signature I declare that I have been informed regarding normal academic citation rules and that I have read and understood the information on 'Citation etiquette' (http://www.ethz.ch/students/exams/plagiarism_s_en.pdf). The citation conventions usual to the discipline in question here have been respected.

The above written work may be tested electronically for plagiarism.

Zurich 16.12.2012	Agata Kaźmierczak
Place and date	Signature

*Co-authored work: The signatures of all authors are required. Each signature attests to the originality of the entire piece of written work in its final form.

Print form

Agreement for free-download

I hereby agree to make my source code of this project freely available for download from the web pages of the SOMS chair. Furthermore, I assure that all source code is written by myself and is not violating any copyright restrictions.

Agata Kazmierczak

Table of Contents

1. Abstract

In this project I would like to investigate the complex processes behind ant’s orientation ability in varied terrain and navigation. I will try to expand previous research by introducing additional navigation mechanism that can be found in the literature. One example could be navigation by polarized sunlight. Furthermore I will try to introduce a mechanism that would allow ants to combine/prioritize navigation methods according to varying environmental conditions. Additionally, accounting for variation in single ant sensing capabilities will be introduced to the model.

2 Individual contributions

The group consisted of only one member, hence there was no division of work within it. All the parts necessary to create this project were done by the author. However some elements were modelled on previous year's projects and had been marked respectively.

3 Introduction and Motivations

Ants are truly fascinating little creatures – they can be found at almost every latitude in the world and in almost all possible habitats, even the most unwelcoming and extreme ones (i.e. deserts). What is more, ants are able to lift twenty times their own body mass and excurse spreading over a distance of up to 200 meters (which, keeping size-distance proportion corresponds to a walk of 35 km for a human being)(Megaro and Rudel 2011).

Keeping in mind all the aforementioned facts, one of the most intriguing questions that are coming to one's mind while observing desert ants is - how such a tiny (and being of a simple brain structure) insect is capable of searching for food and then returning to the nest in almost straight way after relatively long foraging tour? In vast group of ants species the desert ants seem to be existing in the most hostile and demanding environment. On the first sight it seems that there are not many clues that could help ant while searching for food and safely returning home and yet desert ant species developed some amazing mechanisms overcoming those possible difficulties. Therefore, there should be no doubt that they gathered attention of the researchers already more than one hounded years ago (see e.g. Darwin 1873; Murphy 1873). However it were the last 40 years that brought more detailed studies, especially in terms of focusing on the return path to the nest after foraging excursions (Wehner and Wehner 1986; Müller and Wehner 1988; Collett el al. 1999; Merkle 2007).

This work aims to gain more detailed and in-depth insight into specific orientation mechanisms that allows ant to cover such a long distance without getting lost and answer the questions of how does those mechanism are combined together in order to obtain best performance result and finally which of those mechanisms can be regarded as dominant and which the auxiliary.

4 Description of the Model

Years of observations revealed that deserts ants return to their nest after foraging on a rather straight way, often called home vector. The ability to make such a straight line, even without reference to the external cues is said to be based on endogenous mechanism of ‘dead reckoning’, further renamed to path integration or vector navigation (Merkle 2007). The aforementioned mechanism bases its action on constant integration of walking speed and angular variation during foraging route thus creating a variable known as ‘global vector’. Global vector stores the information needed to accurately determine the distance and direction of the nest that can be reached by the ant at any position or at any time.

In addition to path integration numerous studies indicates that desert ants, among many other species, are using ‘landmarks’ (often reffered to as ‘local vectors’) in order to improve their navigational abilities (Collett et al. 1998; Bisch-Knaden and Wehner 2003, Wehner 2003). However, the orientation based on local vectors is undoubtedly very error-prone – landmark can easily disappear or simply change their location during ant’s excursion. Interestingly - if the navigational system based on stored landmark positions fails, desert ants switch to their global vector one and thus are still able to return home safely. This type of behaviour may imply that global vectors might be used as the main tool in ant’s navigational toolkit.

To have its toolkit complete and to be sure that it is as reliable as possible desert ants are also using the so-called sky compass, consisting of the information derived from spectral skylight gradient, position of the sun and the pattern of the polarized light of the sky (Wehner 1997, 2001, Wehner and Srinivasan 2003, Wehner and Müller 2006).

4.1 Path Integration

There are certain types of information that are indispensable for successful path integration performance – accurate measure of the speed (or walking distance) and rotation during the entire excursion. In order to correctly measure those inputs ant must be capable of receiving and processing both external and internal signals.

Among external ones all possible visual signs can be distinguished, that is landmarks (in the immediate neighbourhood of the nest, as well as along the route or near the feeder) or objects situated in a relative distance, like the spectral skylight gradient, position of the sun or the pattern of the polarized light. Even though they do not help to determine speed, rotation movement or provide the information about the actual position of the ant, they indicate explicit direction (the local vector) to follow to get to the nest, feeder or next landmark on the way (Collett et al. 1998, 2003). This fairly auxiliary position of local vector navigation can be easily explained from the evolutionary point of view – in natural conditions landmarks are disappearance-prone, hence seem to be inappropriate as base element of path integration positioning system. Especially as path integration should work as 'standby system' enabling ant to determine its position and homing route when all other navigational systems would fail. When it comes to internal signals (proprioceptive signals), they seem to have little relevance in determining directions, however they significantly help ants to measure their walked distances (Merkle 2007).

As described above ants perform path integration by measuring and summing up all angular changes and distances walked during excursion, this complex process is however laden with errors referring most often either to accurate determination of homing direction, or misestimating the correct nest distance (Wehner and Wehner 1986; Müller and Wehner 1988). Those inaccuracies appear not only during the measurements performed by the ant but also during algorithmic integration, thus should have impact on overall length of the route, this hypothesis however have still not been investigated. What is more, the factors responsible for ants' path integration deviations during foraging excursions performed in natural conditions are not yet revealed (Merkle 2007).

4.1.1 Cartesian model for egocentric path integration – system of linear differential equations for the global vector

This part of the report will focus on the process of path integration itself, answering the fundamental question of how an ant integrates received information to create a home vector available all the time. The model discussed below presents a new approach developed on the basis of previously used models, namely geocentric (using Cartesian and polar coordinate) and egocentric ones (Merkle 2007).

Ant's two main sources of information (physiological sensing and locomotion apparatus) are highly dependent of its specific body architecture, simultaneously being related to its two symmetry axes – the posterior-anterior axis and vertical right-left axis. Therefore, when identifying these symmetry axes with the X and Y axes of a Cartesian coordinate frame (X,Y) and taking ant's body centre as the origin (0,0), the result is a planar moving coordinate frame where the relative position of any object in the ant's neighbourhood can be added. Hence, in resultant egocentric-cartesian model the global vector relative to ant's body axis orientation is G = (X,Y) (see Figure 2) . Keeping the value of Y=0 works as 'internal beacon' for nest orientation. Internal variable Y acts here as elementary negative-feedback control system of the turning rate, until the second internal variable X reaches desired zero value. Path integration values (X,Y) not only provide important information about ant's positional movement but can also serve as information input for orientation. In contrast, the similar model based on polar coordinate uses two variables *r,* distance to the nest, and *δ,* angle between head orientation and nest direction thus making it much more complicated (Merkle 2007) (compare Figure 1).

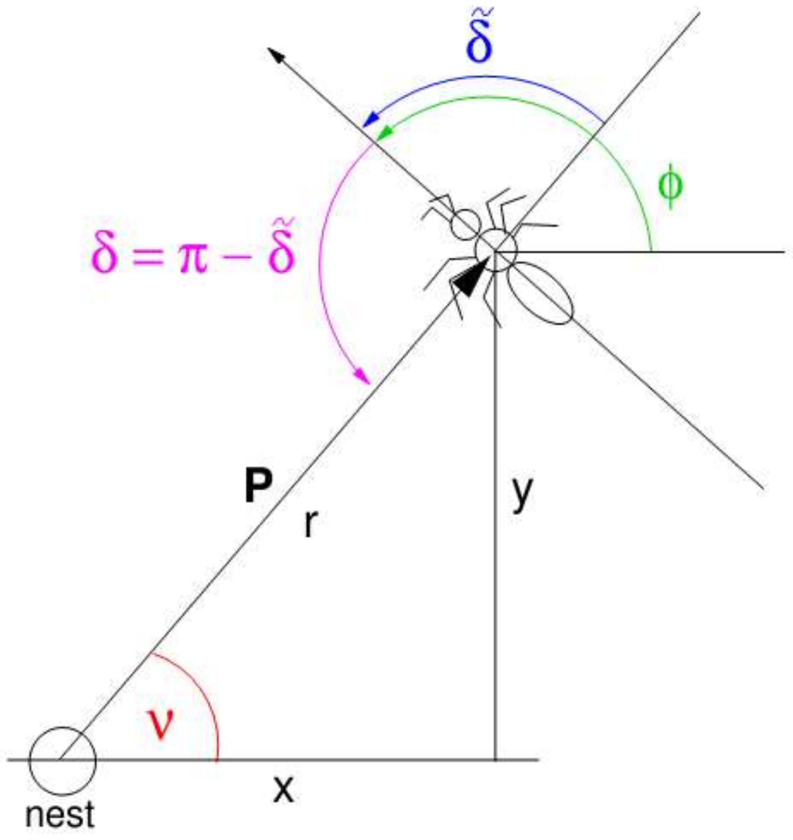


Figure 1.: Theoretical scheme of geocentric path integration model. *Φ* - orientation angle with respect to an external reference direction, represented by the x-axis. Both the cartesian coordinates (*x*, *y*) and the polar coordinates (*r, v*) indicate ant's position in relation to the nest. *P* - positional vector. Source: Merkle 2007.

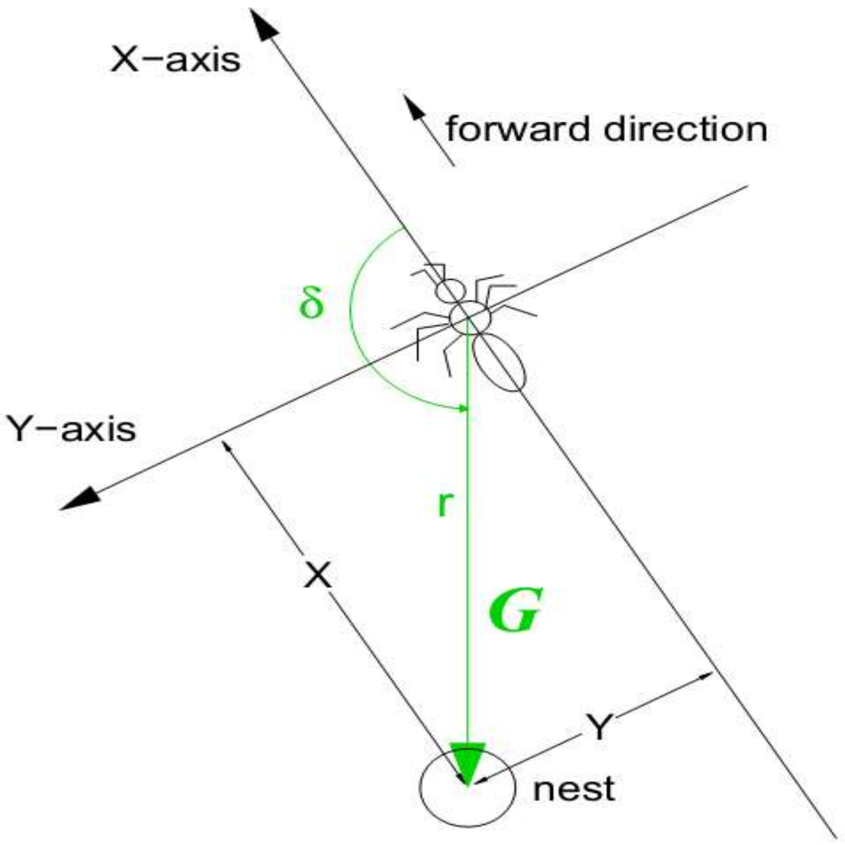


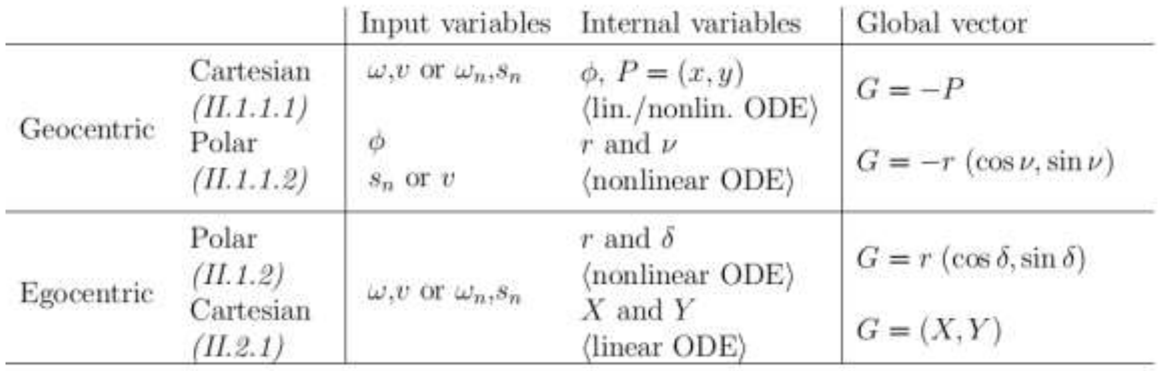
Figure 2.: Theoretical scheme of carthesian egocentric path integration model. Cartesian coordinates X,Y indicate the position of the nest in relation to ant's body axes and determine the global vector G = (X,Y). Source: Merkle 2007.

Given the information about ant's forward speed *v* and angular turning rate *ω*, the following model equations might be used to accurately update the global vector (X, Y) during motion:

$$\frac{dX}{dt} = -v + \omega \cdot Y$$

$$\frac{dY}{dt} = -\omega \cdot X$$

The two-dimensional differential equation system shown above is very simple – it is linear in the two variable quantities X and Y, what is more it only uses two speed input parameters, v – the rate of shifting the X coordinate backwards, ω - the rate of rotating the (X, Y) frame clockwise, as additive or multiplicative terms. In contrast to models used previously (compare Figure 3), the ant does not have to perform complicated calculations using trigonometric or other non-linear functions, thus it is highly probable that this type of operations can be easily accomplished by relatively simply structured brains (Merkle 2007) (see Figure 4).



		Input variables	Internal variables	Global vector
Geocentric	Cartesian *(II.1.1.1)*	ω,v or ω_n,s_n	ϕ, $P = (x, y)$ ⟨lin./nonlin. ODE⟩	$G = -P$
	Polar *(II.1.1.2)*	ϕ s_n or v	r and ν ⟨nonlinear ODE⟩	$G = -r\ (\cos\nu, \sin\nu)$
Egocentric	Polar *(II.1.2)*	ω,v or ω_n,s_n	r and δ ⟨nonlinear ODE⟩	$G = r\ (\cos\delta, \sin\delta)$
	Cartesian *(II.2.1)*		X and Y ⟨linear ODE⟩	$G = (X, Y)$

Figure 3.: Path integration models with the parameters and variables used for input, internal calculation and output as global vector. Source: Merkle 2007.

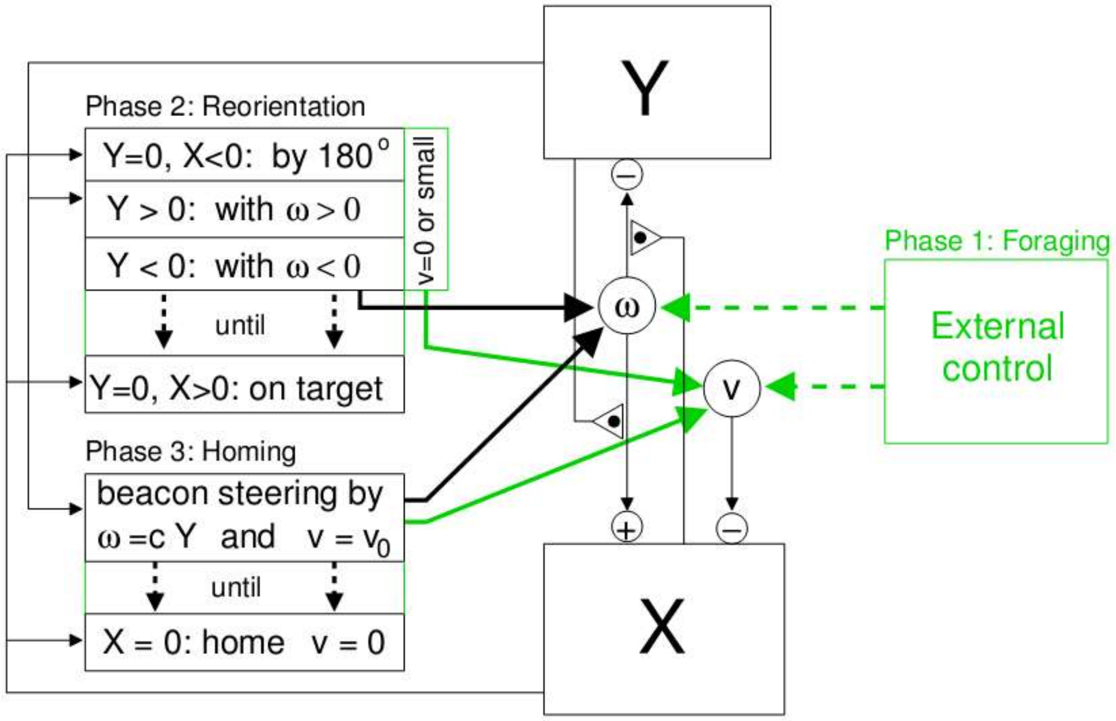


Figure 4.: Analogue circuit scheme of the egocentric cartesian path integration model. Dynamics of the two variables X and Y according to the differential equations presented above and their link to the physiological control parameters represented by the two speed parameters for turning - *ω*, and forward locomotion -*v*. In Phase 1, ω and v are externally controlled (random search, trained path towards feeder, etc.). During Phase 2 and Phase 3 the X and Y values feedback into the speed control conditions by counter–steering with respect to the 'internal beacon' Y = 0 in the latter case. Equations create fixed nonlinear control system along the homing path. Source: Merkle 2007.

4.2 Landmark Orientation

While returning to the nest from foraging excursion, desert ants use several navigational tools at one time (i.e. path integration and visual landmarks). Path integration mechanism constantly computes their distance and direction to the nest throughout the whole journey, allowing the same direct return from any current location. However on a familiar route, when ants can easily refer to visual landmarks they perform fixed and often indirect path consisting of several separate segments that can point in different directions. Such multi-segment journey may result from the fact that stored local vectors are associated with landmarks and are recalled when ant reaches appropriate place on its route. Studies indicates that local vector navigation dominates over global vector one while travelling across familiar, cluttered territory. Nevertheless it regains the lead once ant reaches clutter-free terrain or other navigational systems fail (Collett et al. 1998).

4.3 Sky Compass Orientation

It has been observed that whenever ant uses 'spectral cues' to determine its body axis' orientation with respect to the azimuthal position of the sun, it refers to the fact that light waves have different wavelengths and are not equally distributed over the sky (Wehner 1997). What is more, the sun and spectral cues are considered to constitute kind of additional navigational tools that are most likely implemented in case of polarization compass failure, which is believed to be ants' standard navigational tool (Wehner and Srinivasan 2003).

As studies reveal skylight polarization pattern ('polarization compass') gives the most effective and reliable results when it comes to resolving the orientation problems in desert ants (Wehner 2003; Wehner and Srinivasan 2003; Wehner and Müller 2006). This type of compass is based on ant's ability to recognize 'e-vector pattern' (produced by scattering of the sunlight at air molecules in the atmosphere, which is done by using specialized UV-receptors placed in the dorsal part of ant's compound eye (Merkle 2007) (see Figure 5). The mechanism working behind polarized compass seems to be fairly simple – desert ant measure the difference between their orientation and the e-vector one in the polarized light pattern. Even though this pattern changes with sun elevation ants store stereotypical template of it in their memory (Wehner 1997, 2001). The strategy requires matching stored template with the actual e-vector pattern in the sky and measuring rotations using the degree of compliance with the current observed e-vector pattern. Each rotation of ant's body axis entails increase or decrease of this compliance, hence giving the ant some clues not only about the direction of the performed rotation but also about its value (Merkle 2007).

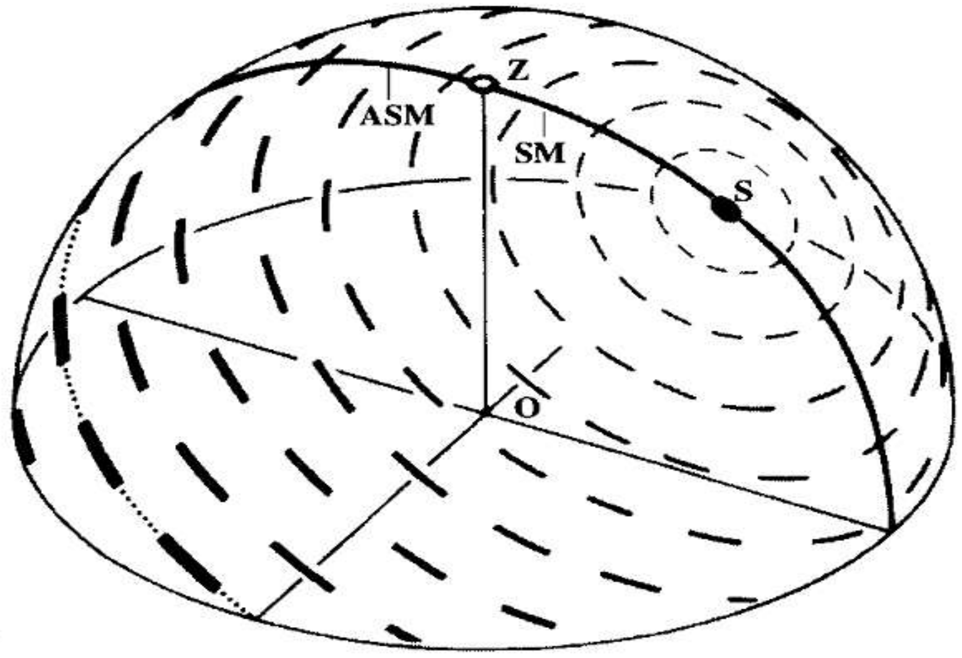


Figure 5.: Three-dimensional representation of pattern of polarization in the sky as experienced by and observer (*O*) in the centre of the celestial hemisphere. The e-vector directions and the degree of polarization are indicated by the orientation and the width of the black bars. Owing to the daily westward movement of the sun across the sky (by some 15 degrees per hour), the symmetry plane, and with it the entire e-vector pattern, rotates about the zenith. The e-vector distribution changes as the sun changes elevation but retains the mirror symmetry. In this figure, the solar elevation is 53 degrees. Source: Lambrinos et al. 1997.

Additionally, the problem of compensating the westward movement of the sky throughout the day has to be shortly discussed here. In the case of ant heading towards it nest after successful foraging, it is not necessary to possess any specific mechanism designed to compensate such movement during an excursion. Normally foraging routes last no longer than few minutes, hence the error resulting from aforementioned movement is insignificant. However, in case of ant heading towards known feeder it was proven that it has to have some kind of general inherent knowledge about the movement and the speed of the wandering sun which is further improved during repeated foraging excursions throughout lifetime (Wehner and Srinivasan 2003; Merkle 2007).

5 Implementation

5.1 Implementation – General Information

The simulations are made using two user-defined classes: world and ant. Class world defines the universe of simulation in which objects of class ant can wander freely in search for food. In turn, the universe of simulation is a N by N object, flat and composed of discrete fields of size 1x1, where 1 is an arbitrary unit of length.

Properties of object belonging to class world contain information about location of all the items that can exist in the simulation universe, that is:

- food (also called feeders) – objects of highest interest of ants, which they are in search of;
- nest, from and to which ants forage, “bringing back” food that they manage to find during their excursions;
- landmarks – objects used by ants for orientation, containing information about their relative position to nest;
- obstacles – objects that cannot be passed through by an ant and must be avoided.

Respective locations of aforementioned objects that can be found in the world class are stored into assigned cell arrays. There is also one additional, N by N matrix, where N describes size of the world, storing information about universe objects in terms of numbers as follows:

- 0 represent empty fields,
- 2 represent nest,
- 3 represent landmarks,
- 4 represent obstacles,
- 5 represent food/feeders, which is further used for plotting purposes.

Objects belonging to the class world can be randomly distributed across generated board, i.e. location of any item can be either defined randomly or by user, according to his desired position. A mixed solution is also possible, that is where some items are created randomly while others are predetermined by the user. In any case, nest is <u>always</u> located in the center of the map.

What is more, simulation universe is additionally populated with objects of class ant. It should be noted here that this project does not assume ants' ability to communicate which is due to lack of implementation of pheromone communication, therefore there is usually only one ant performing foraging excursion. However there do not exist any restrictions determining the arbitrary number of ants, nevertheless if they are unable to influence each other, this seem to be of minor impact to the model as a whole.

Object belonging to the class ant contains various types of information about itself and existing e-vector. Yet, not all information stored in this object (such as absolute position with respect to global coordinate system) is available for the ant to decide on its movement.

Once ant is created, main function of the code calls method walk, belonging to class ant, and repeatedly with each update step simulating single movement of an ant. In each step, ant is allowed to travel within its Moore neighbourhood (compare Figure 6) which allows eight possible directions to move, unless an obstacle is obstructing the way.

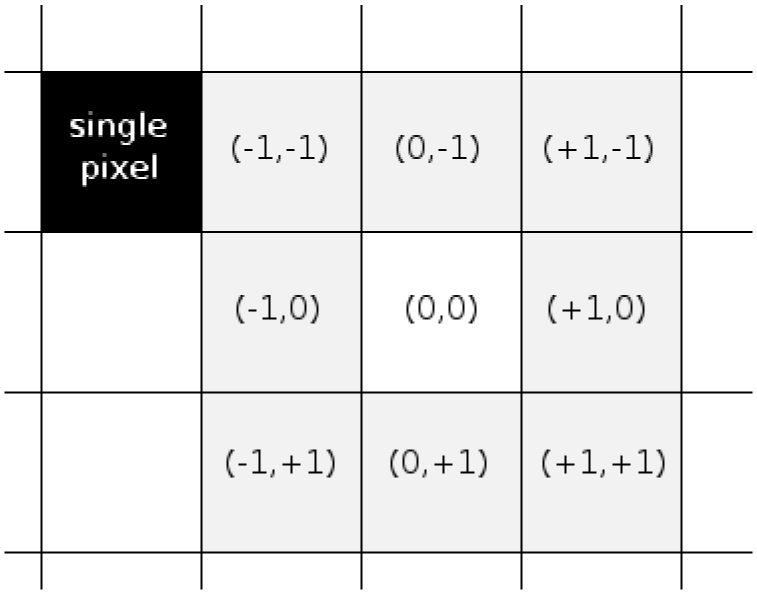


Figure 6.: Example of Moore neighbourhood. Source: http://zvold.blogspot.ch/2010/01/conways-life-and-brians-brain-cellular.html

Each time step lasts a single unit of time, therefore ant’s velocity can vary depending on which direction is chosen. If ant travels to one of the corners of Moore neighborhood, its velocity equals $\sqrt{2}$ [*unit of length / unit of time*]. In remaining cases its velocity equals 1 [*unit of length / unit of time*]. The choice of the direction is of a great interest in this work, and its implementation is therefore discussed in more details in the section below.

5.2 Ant's movement and path integration

During each time step a few actions are “undertaken” by the modeled ant. To begin with, ant is allowed to look around and “see and recognize” its surrounding. The objects that are “seen” by the ant may further be used while deciding which direction of movement will be chosen. The subsequent action is to devise the direction of ant's movement, based on its actual intention (looking for food / returning to nest) and objects “seen” (obstacle that requires alteration of path, landmarks, containing directional information, etc.). Consequently, the last action is to perform the step i.e. update ant's position and perform path integration. Each action is to be described in detail in the sections below.

5.2.1 Ant's sight

Each time the **antSight** function is called, it enables ant to “look around”. All the objects that lie within predefined sight radius are then stored in ant's **fieldOfVision** separate variable. Sight radius is arbitrary defined while ant is created and can be modified by the user at that moment. This project assumes that the concept of virtual ant's sight envelops all sensory information that would be available to an ant, including sight and smell. Therefore, ant is allowed to “see” behind obstacles that lie directly in its sight range, which is motivated by the fact that in real life ant would probably not be able to see an object behind a fallen branch, but could be able to smell it instead.

5.2.2 Ant's choice of direction

For this project ant can find itself in one of three main modes numbered 1, 2 and 3 respectively.

5.2.2.1 Mode 1 – Foraging

Mode 1 is set to be the foraging one, which define the goal of an ant as to find the food. In this particular state ant begins each of the simulations. The first step out of the nest is chosen at random among 8 possible directions. The next step direction is chosen based on normal random distribution with mean equal to the previous step and standard deviation increasing with the increase in the distance to the nest. These restrictions ensure that ant will follow more or less a straight path at the beginning of the simulation similarly to (Merkle 2007). However, the further ant gets the higher the probability of turning and eventually returning to the nest becomes.

This random stepping is performed as long as a food source can be observed. Once ant captures the food source in its sight rage it heads straight towards it. As soon as ant’s location equals the location of the food source, three things happen:

a) mode is changed to 2,
b) flag **feederFound** is set to true, indicating that a feeder was found and path integration allowing to return to its position is to be performed at each next step,
c) a global vector called **feeder_globalVector** is created with the value [0, 0].

In each next step, ant will update the value of the aforementioned vector using path integration process which after returning to the nest, will allow finding way to the feeder again, basing on PI.

5.2.2.2 Mode 2 – Homing

Mode 2 is set to be the homing one – once ant found a feeder it is then returning to the nest with a food portion. A basis for devising direction is a **globalVector** vector that was updated during each step of simulation and points from ant to the nest (according to ant’s coordinate system). Ant follows its best estimation until one of two things happen:

a) nest appears in the sight radius, in which case the indications of **globalVector** are overruled by sensory information, and ant heads straight towards to the nest,
b) (X,Y) values of the **globalVector** reach zero, which means ant arrived in assumed nest position, but did not find it.

If the scenario a) happens, then the mode is switched back to 1, foraging, and ant is off for another food search. The **globalVector** becomes reset in order not to transfer the error from previous run into the following one. This time however, an approximate location of the feeder is known, so while in mode 1, ant follows its **feeder_globalVecto**r indications to revisit previously found feeding site. Of course, if during its trip to feeder, another feeder is found, ant turns in its direction and then returns to nest, with **feeder_globalVector** being stored with respect to this new feeder.

In case of scenario b), the mode is changed to 3.

5.2.2.3 Mode 3 – Cannot Find the Nest – Spiraling

Whenever the ant is unable to find the nest, according to (Wehner and Srinivasan 1981; Hoffmann 1983; Wehner and Wehner 1986; Merkle 2007), it begins a systematic search for the nest. While this specific mechanism is engaged, it performs loops of increasing radius around the assumed nest position. In the model adopted in this project, this type of search was implemented in a way that ant begins a spiraling motion, with assumed nest position as its center until the nest is found. If the nest is found, everything continues as if nest was found while in mode 2.

5.2.3 Seen objects influence on directions choice

Each walking mode is additionally influenced by objects that can be encountered by the ant on the way. As it was mentioned before, in mode 1, seeing food overrules other indications, while in mode 2 or 3 it is seeing the nest. Other than that, two other objects can significantly influence ant's decision – landmarks and obstacles.

To begin with obstacles, their influence also depends on if ant possess a driving vector (sees food / nest or navigates with **globalVector** to the feeder / nest) or is randomly foraging. In the latter case, ant is simply forbidden to walk in the obstacle and another field, which belongs to ant's Moore neighborhood and lies at the border of the obstacle, has to be chosen (see Figure 7). Then ant follows with its random motion, which means it can walk along the obstacle or depart from it at any time.

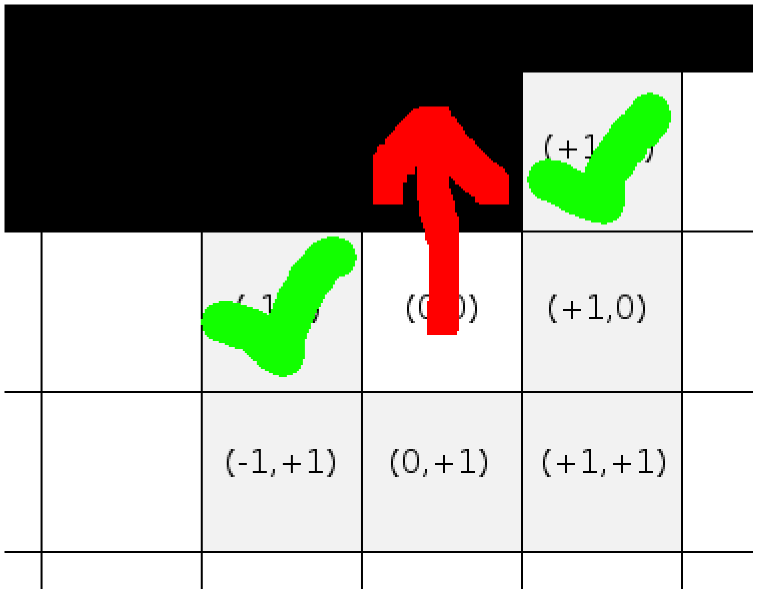


Figure 7.: Obstacle avoiding method scheme. The red arrow indicates intended direction, while green marks indicate possible directions of movement.

If ant possesses a driving vector, what happens is that the intended direction is stored in a separate variable, and then a field to step on is chosen as in previous case. In the next step, code checks if the intended direction is now free and if yes, performs step in that direction. Otherwise, it continues with the direction along the obstacle. After a step in intended direction is made, a normal walk mode is restored, and ant uses its driving vector to navigate again.

Of course during each step **antSight** and global vectors are updated, so that after obstacle is avoided they do not provide false indications.

In turn, landmarks are used for navigation only while ant is homing. For the purpose of this project it is assumed that ant was trained, i.e. seeing a landmark invokes a memory of a local vector pointing to the nest. In the code, each landmark stores such a local vector, and when it becomes visible, its vector is added to the vector pointing from ant to the landmark, which results in vector pointing from ant to the nest. If more than one landmark is seen, an average of two local vectors is used.

Such local vector is then averaged with the global vector that points to the nest, resulting in more accurate global vector, which is then updated. This obviously helps to compensate for the errors and provides better navigation.

Landmarks and local vectors are used only if they are seen, so if their layout of landmarks in the world would change – e.g. some would be removed, ant would just use other navigation means. This behaviour was observed in experiments (Collet et al. 1998).

5.2.4 Polarization compass

When ant is created, an e-vector of a random direction is also created. It can take one of 8 possible angles, corresponding to 8 possible directions of ant movement. As it was explained before, it seems that ants have a mechanism allowing them to compensate for the rotation of this vector during the day, therefore for simplicity, in this project the e-vector direction is assumed constant in time. Also, the concept of e-vector used here accounts for two sources of information – polarization patter of the sky and sun position, without explicitly showing mutual influences of each of these phenomenons separately.

5.2.5 Path integration

Path integration of the ant is based on the coordinate system attached to the ant, as it was described before. Ant stores a vector pointing from it to the nest, in term of that coordinate system. While ant turns, it checks its rotation with respect to the e-vector. Comparing current and previous rotation with respect to e-vector, ant devises an angular change that was performed and then, rotates stored global vector into its new coordinate system. After the rotation, step is performed, thus X variable (along ant's body) is updated by the length of step (either 1, or $\sqrt{2}$). This mechanism is applied for composing both vectors – one pointing to nest and one pointing to the previously found feeder.

Global vectors are altered each step. Also, presence of landmarks applies a correction to the global vector – a mean of global vector and local vector indications which reduces the error.

The error in path integration in this model is introduced by the use of variable **rotationerror**, which can be defined by the user. This was mentioned in details in (Merkle 2007) as linear underestimation of rate of change ω and is performed as follows:

$$\omega proc = rotationerror * \omega real\ ,\ 0 < rotationerror < 1$$

Presence of this error leads to underestimation of rotation angle.

6 Simulation Results and Discussion

6.1 PI Error

Introducing **rotationerror**, results in miscalculation of real nest position by ant, which has been previously also observed by (Merkle 2007). Comparison of errors can be seen in the Figure shown below:

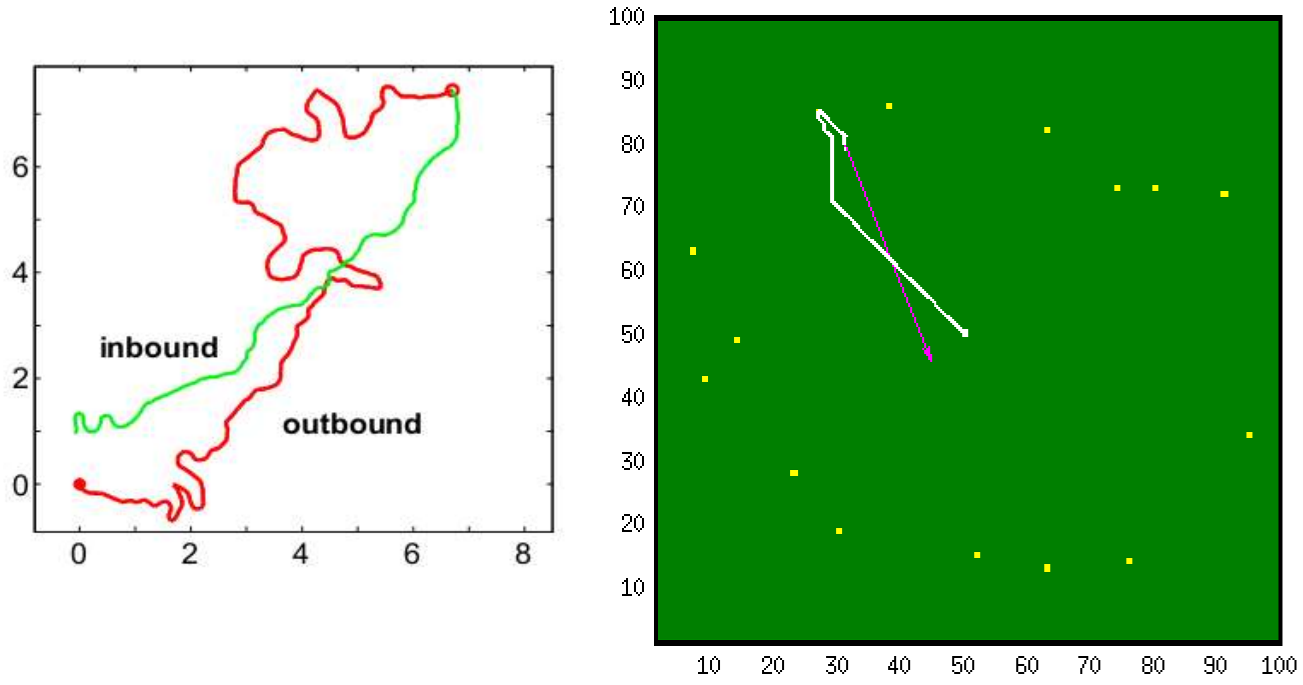


Figure 8.: Comparison of rotational errors.

This project's implementation of **rotationerror** seems to well approximate the linear underestimation of rotation.

Comparison of X Y coordinates of **globalVector** with results obtained by (Merkle 2007) reveal, that due to long straight sections of movement, project's implementation seems rather artificial. Greater variation of ant direction should be present to better replicate the behaviour of a real ant.

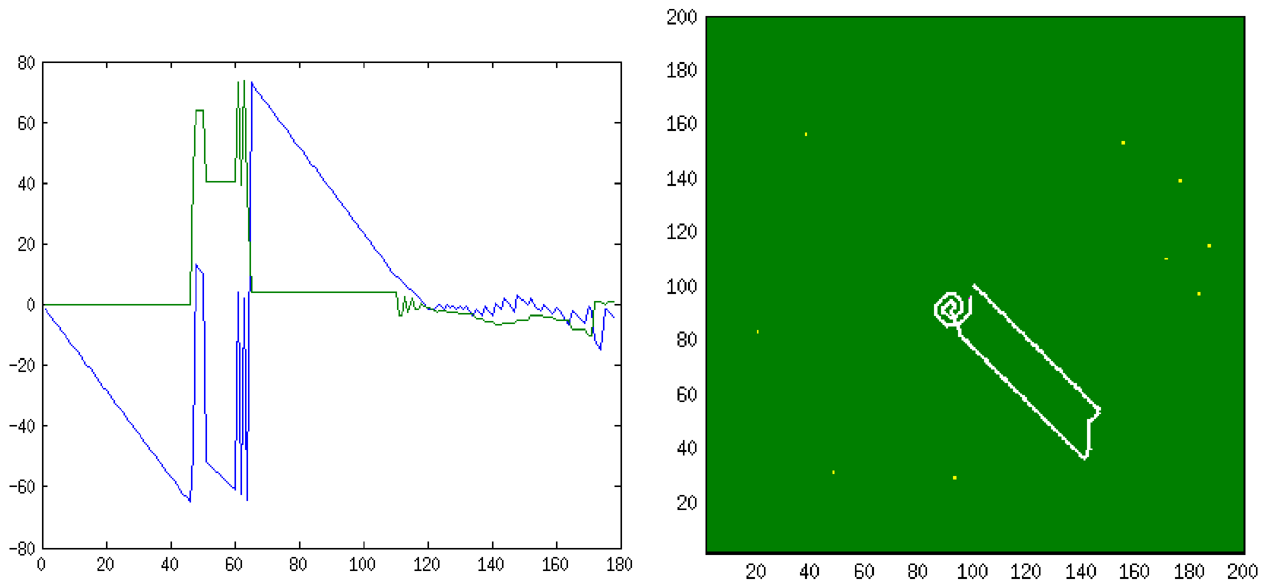


Figure 9.: Comparison of X Y coordinate graph

6.2 Pi error compensation by other means of navigation

6.2.1 Direct sensory input

Small error can be compensated by the fact that arriving to wrong location, which is reasonably close to the nest, allows ant to "see" the nest, and use this direct sensory input to overcome appearing error.

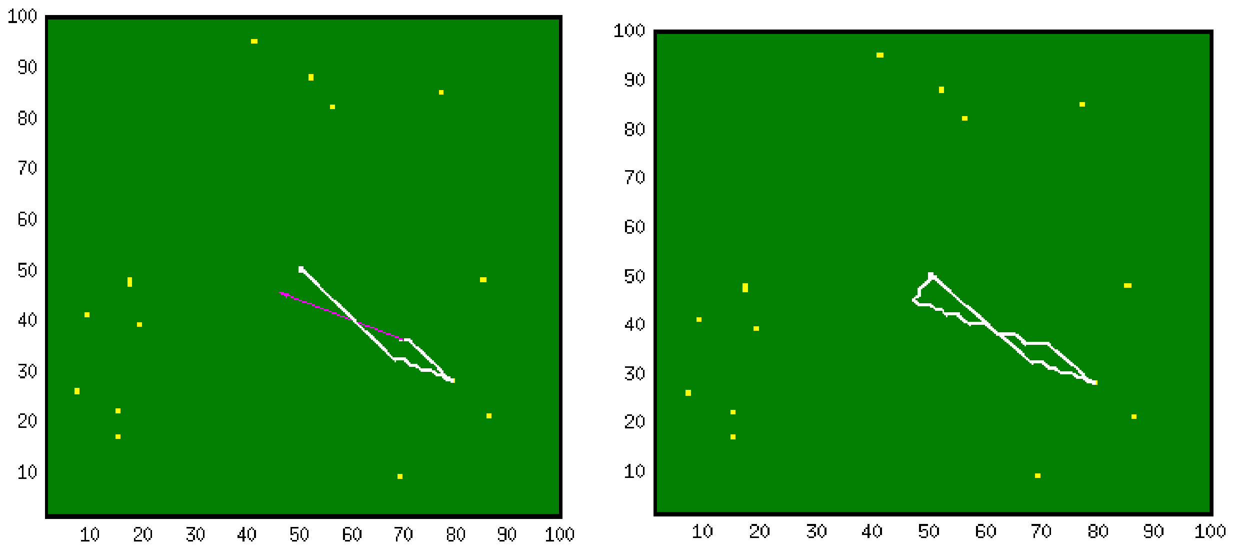


Figure 10.: First picture shows the PI error, with nest being in the centre, feeder in the rightmost point of the white ant trail and violet vector showing ant's **globalVector.** Second picture shows that after arriving to location indicated by **globalVector**, ant can see the nest at easily navigate to it by sensory information.

6.2.2 Landmarks

Having landmarks placed on the way allows ant to gradually compensate for PI error.

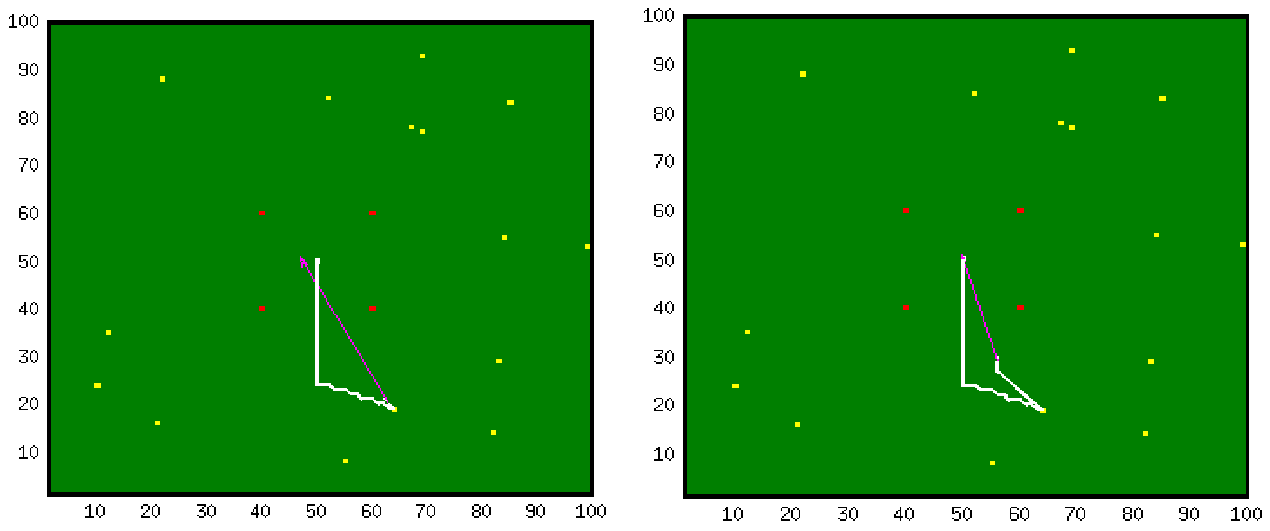


Figure 11.: Picture one shows globalVector in violet before correction (nest in center, feeder in rightmost point of the trail). Picture two shows the corrected globalVector after landmark was seen by the ant.

Due to the way it was introduced in the model, this change is gradual, and the longer the ant sees the landmark, the greater the correction.

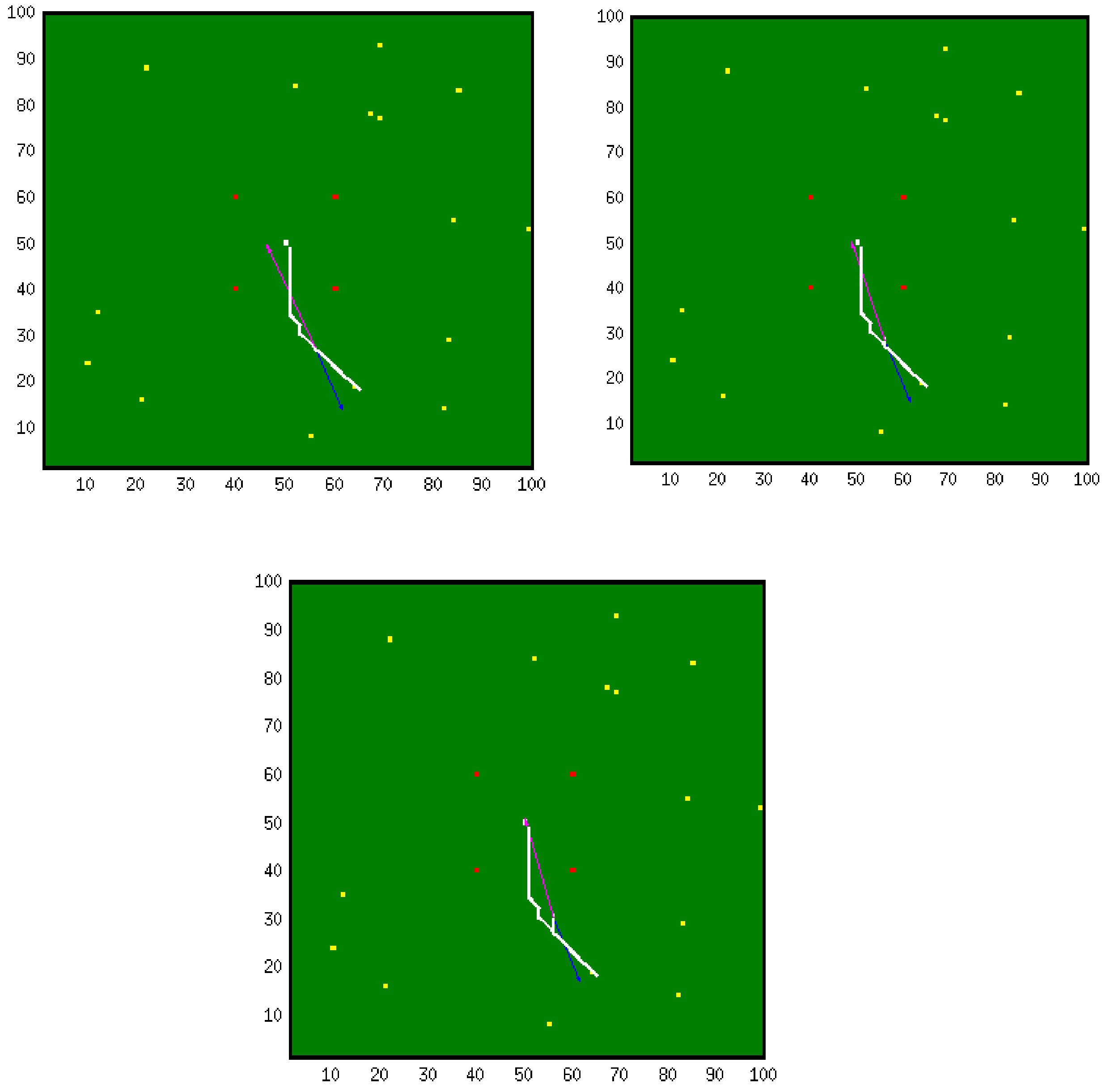


Figure 12.: Pictures one to three present progressing correction of **globalVector** (violet) while walking in the proximity of the landmark.

6.2.3 Specific search behavior

In case PI error was large enough that after arriving to estimated nest position, nest is nowhere to be seen, an efficient and thorough search strategy can allow to come into nest proximity, allowing the sensory information to take over, and bring ant back to the nest. However, performing spiralling motion means a lot of steps made, and thus, the error of **feeder_globalVector** increases dramatically, which can be seen.in the Figures presented below:

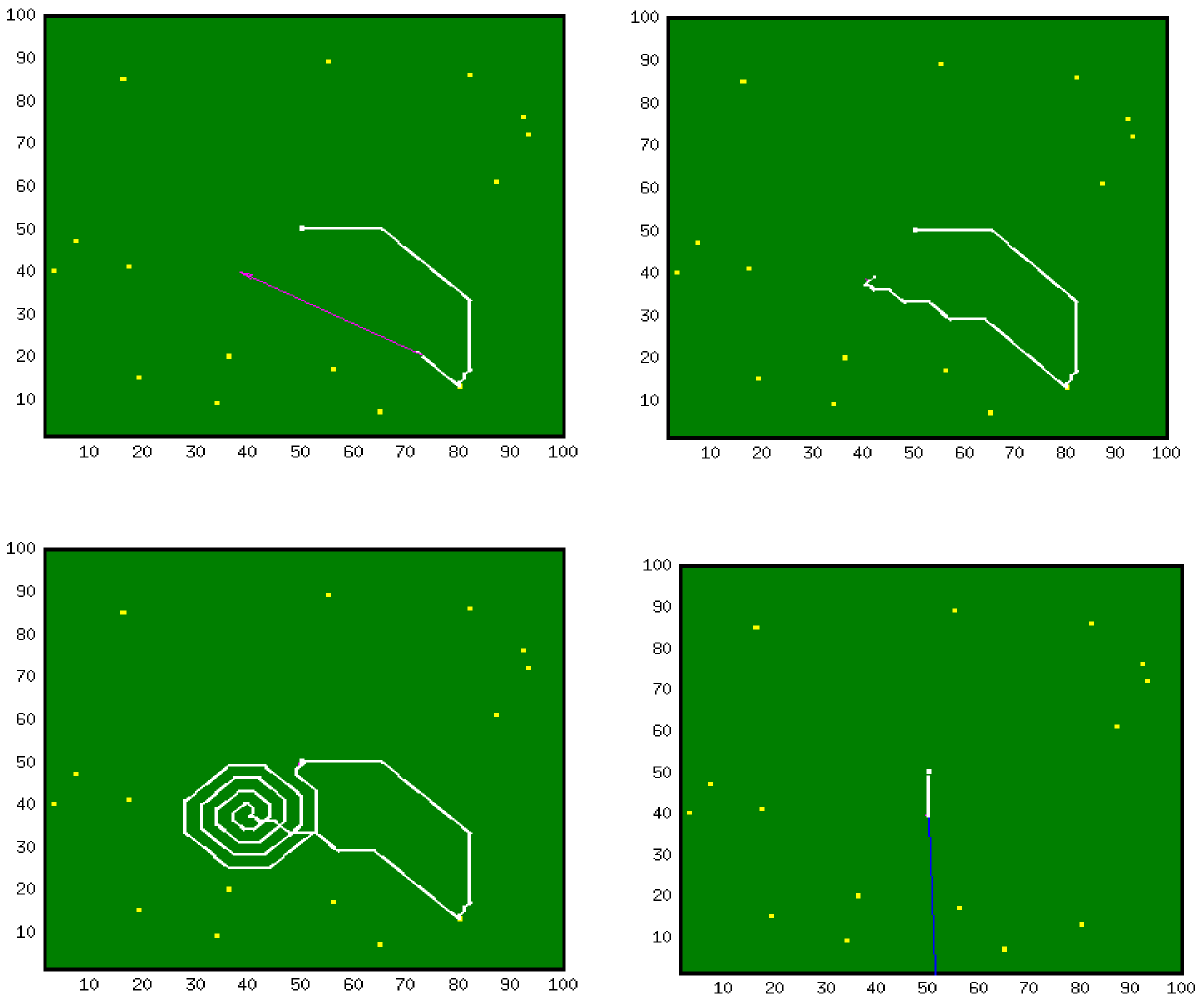


Figure 13.: Picture one portraits **globalVector** in violet, resembling big error (nest in the centre, feeder in the right-down corner of the trail). Picture two shows ant arriving two assumed nest position, not finding or seeing it and initiating nest search behaviour. Picture three shows spiral performed by the ant as long as the nest has been seen. Picture four shows that due to long walk, error of **feeder_globalVector** (blue) is very large (should be pointing to the same feeder as was found during excursion).

6.3 Obstacle Avoiding Mechanism

The set of figures below presents an obstacle avoiding in action. The implemented mechanism gives reasonable results and allows ant to navigate efficiently.

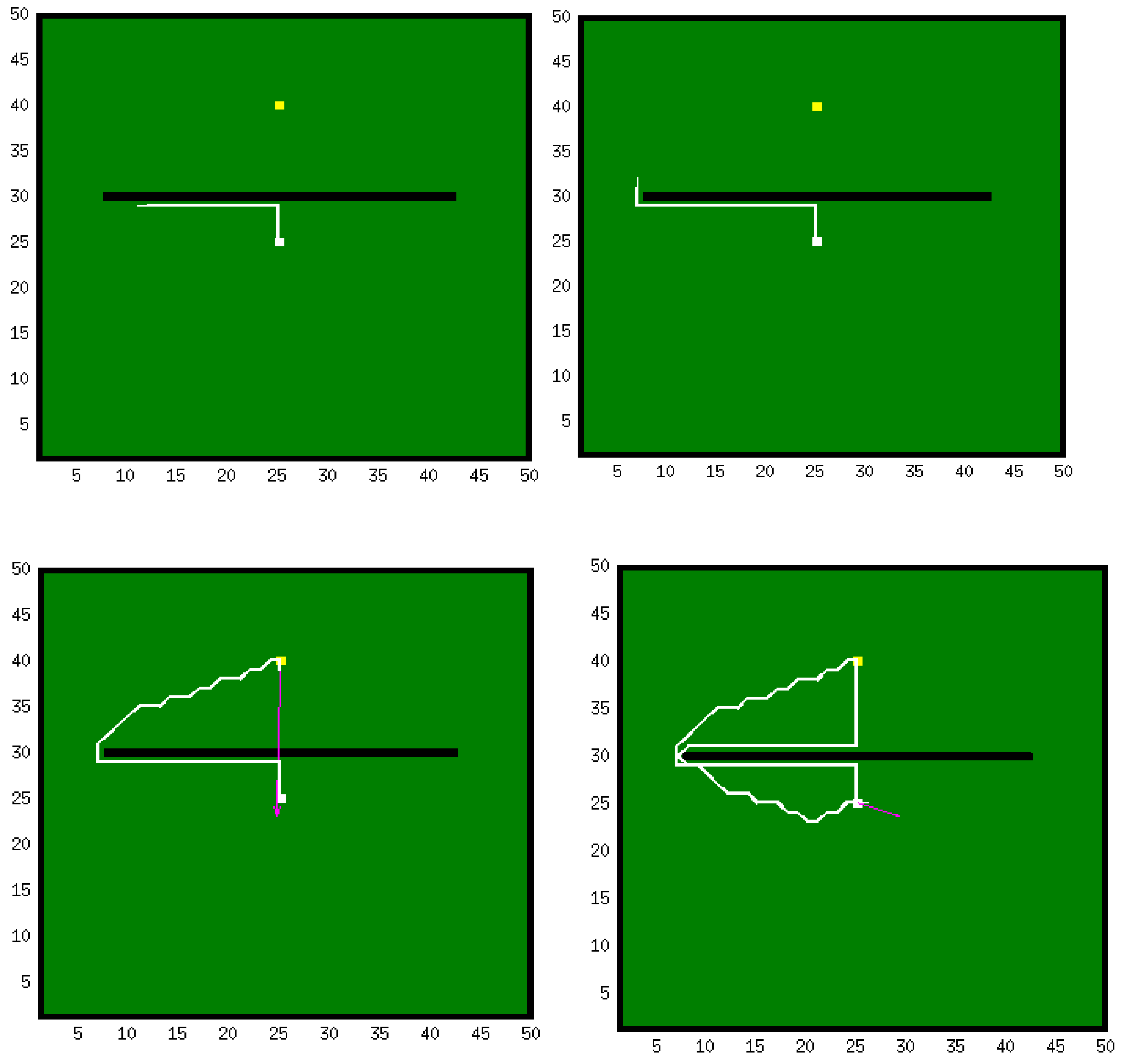


Figure 14.: Picture one shows ant encountering obstacle and engining avoiding technique. Picture two shows using stored intended direction to proceed after it is possible – edge of obstacle was encountered. Picture three shows **globalVector** in violet, pointing to nest after feeder was found. Picture four shows completed run, obstacle was avoided the second time on the run to the nest.

7 Summary and Outlook

To sum up, the produced model seems to prove that using multiple navigational means results in more efficient and accurate movement of modeled ants.

The possibility to compensate the path integration errors in one of three ways – seeing goal of search, using local landmarks or manifest appropriate search behavior – allows overcoming imperfections of the navigational system.

What is more, an orientation model, simple enough to be accommodated by limited amount of neurons in ants nervous system proved efficient and accurate enough to allow navigation in virtual universe. However, sometimes unexpected behaviours could be observed, which mean that a certain amount of work is required to debug and improve the current implementation.

The model implemented in this project uses simple means of deciding which navigational tools have priority – it's either overruling the rest (e.g. while seeing goal or avoiding obstacle) or calculating arithmetical mean of two vectors (global / local vectors). It would be interesting to introduce more sophisticated methods (e.g. weighted average with adjustable weights) and incorporate a learning mechanism, allowing ants to modify rules by which they utilize different tools.

Future work should also include extending the amount of navigational tools available to the ant. This could, among others, include separating the influence of sun position and sky polarization. Yet another extremely interesting alternation would be to separate sensory information available to ant, such as smell and sight, rather than have them as a combined tool. Moreover, greater insight into interactions between different senses and their influence power should also be of interest.

Additionally, pheromone deposition should be introduced, to allow multiple ant-to-ant influences and interactions to be examined. Varying decay of pheromones could be used to simulate different environments – ranging from African deserts to European forests and beyond.

Another task that should be undertaken is incorporating landmarks while ants are navigating back to the feeder, as well as mechanism by which ants learn which local vector should be assigned to a landmark, rather than have them arbitrarily defined with no error, as it is done in the present implementation.

Other errors than rotational one should also be introduced along with their influence studies. Possibility of ant to learn and modify its governing equations in order to compensate for those errors would be another interesting topic to work on.

8 References

Bisch-Knaden S, Wehner R. 2003. Landmark memories are more robust when acquired at the nest site than en route: experiments in desert ants. *Naturwiss* 90: 127–130.

Bisch-Knaden S, Wehner R. 2003. Local vectors in desert ants: context-dependent landmark learning during outbound and homebound runs. *J Comp Physiol* A 189: 181–187.

Collett M, et al. 1998. Local and global vectors in desert ant navigation. *Nature* 394: 269–272.

Collett M, et al.. 1999. Calibration of vector navigation in desert ants. *Curr Biol* 9: 1031–1034.
Collett M, et al. 2003. Do familiar landmarks reset the global path integration of desert ants? *J Exp Biol* 206: 877–882.

Collett TS, et al. 2003. Route learning by insects. *Curr Opin Neurobiol* 13: 718–725.

Darwin C. 1873. Origin of certain instincts. *Nature* 7: 417–418.

Hoffmann G. 1983. The random elements in the systematic search behaviour of the desert isopod *Hemilepistus reaumuri*. *Behav Ecol Sociobiol* 13: 81–92.

Hoffmann G. 1983. The search behaviour of the desert isopod *Hemilepistus reaumuri* as compared with a systematic search. *Behav Ecol Sociobiol* 13: 93–106.

Lambrinos D, et al. 1997. An autonomous agent navigating with a polarized light compass. *Adaptive Behavior* 6:131–161.

MATLAB On-line support – www. Mathworks.com

Megaro V, Rudel E. 2011. Desert Ant Behaviour. Lecture with Computer Exercises: Modelling and Simulating Social Systems with MATLAB. Project Report.

Merkle T, 2007. Orientation and Search Strategies of Desert Arthropods: Path Integration Models and Experiments with Desert Ants, Cataglyphis fortis (Forel 1902). PhD thesis, University of Bonn.

Murphy JJ. 1873. Instinct: a mechanical analogy. *Nature* 7: 483.

Müller M, Wehner R. 1988. Path integration in desert ants, Cataglyphis fortis. *Proc Natl Acad Sci USA* 85: 5287–5290.

Wehner R, Wehner S. 1986. Path integration in desert ants: approaching a long-standing puzzle in insect navigation. *Monit Zool Ital* 20: 309–331.

Wehner R. 1997. The ant’s celestial compass system: spectral and polarization channels. In: Pasteels JM, Deneubourg JL (editors). Orientation and Communication in Arthropods, pp. 145–185. Birkhäuser; Basel.

Wehner R. 1997. Insect navigation: low-level solutions to high-level tasks. In: Srinivasan MV, Venkatesh S (editors). From Living Eyes to Seeing Machines, pp. 158–173. Oxford University Press; New York.

Wehner R. 2001. Polarization vision - a uniform sensory capacity? *J Exp Biol* 204: 2589–2596.

Wehner R. 2003. Desert ant navigation: how miniature brains solve complex tasks. Karl von Frisch Lecture. *J Comp Physiol* A 189: 579–588.

Wehner R, Srinivasan MV. 1981. Searching Behaviour of Desert Ants, *Genus Cataglyphis* (Formicidae, Hymenoptera). *J Comp Physiol* 142: 315–338.

Wehner R, Srinivasan MV. 2003. Path integration in insects. In: Jeffrey KJ (editor). The neurobiology of spatial behaviour, pp. 9–30. Oxford University Press; Oxford.

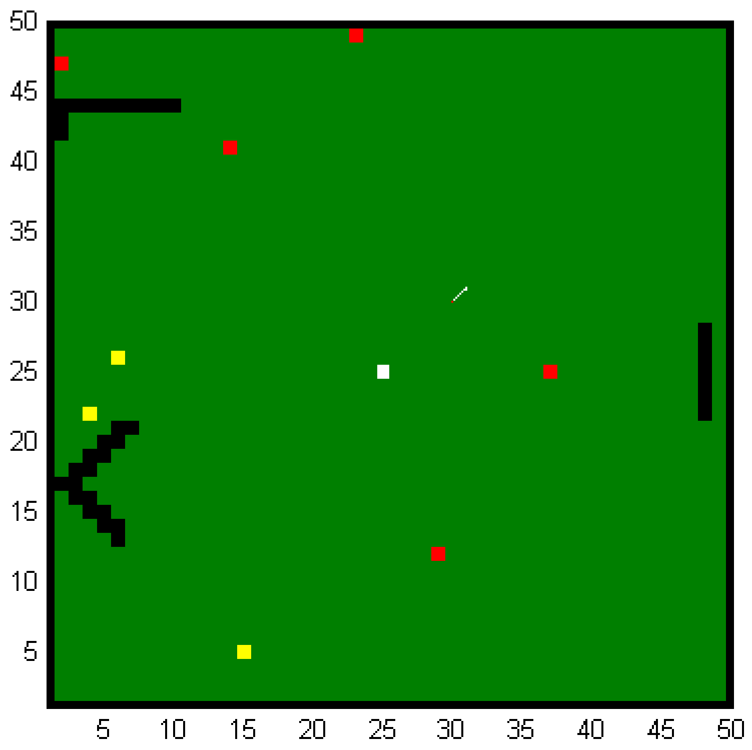
Wehner R, Müller M. 2006. The significance of direct sunlight and polarized skylight in the ant’s celestial system of navigation. *Proc Natl Acad Sci USA* 103: 12575–12579.

9 Matlab Code

```
clear all

cd('C:\Users\Agata\Desktop\Mrowki')

% define length of simulation
end_time=5;

% reply=input('Do you want to generate or load the map? 1 - generate map, 2
% - load map: ', 's'); if reply=='1'

% create universe
size=50;
universename='universe';
universe=World(size);
loc_obs=[];%{[8 30] [9 30] [10 30] [11 30] [12 30] [13 30] [14 30] [15 30] [16 30] [17
30] [18 30] [19 30] [20 30] [20 30] [21 30] [22 30] [23 30] [24 30] [25 30] [26 30] [27
30] [28 30] [29 30] [30 30] [31 30] [32 30] [33 30] [34 30] [35 30] [36 30] [37 30] [38
30] [39 30] [40 30] [41 30] [42 30]};
loc_land=[];%{[30 30] [20 20]};
loc_food=[];%{[40 45] [10 10]};
food_amount=3;
obs_amount=3;
land_amount=5;
ant_sight_range=5;
rotationerror=0.99; % from 0 to 1 is multiplied by good rotation
min_nest_dist=1/3;

% populate universe
universe=placeObstacles(universe,obs_amount,loc_obs);             % place obstacles in
the world
universe=placeLandmarks(universe,land_amount,loc_land);           % place landmarks in
the world
universe=placeFood(universe,food_amount,min_nest_dist,loc_food);   % place food in the
world

%     reply=input('Do you want to save generated map? Y/N [Y]: ', 's'); if
%     reply=='Y'
%         name=input('Give the name of the file:' , 's');
%        save (name, universename);
%     elseif reply~='Y' && reply~='N'
%         disp('Wrong input. Not saving.')
%     end

% elseif reply=='2'
%     filename=uigetfile; load(filename)
%
% end
```

```
%create ant
b=Ant(1,universe,ant_sight_range,universe.nest{1}, rotationerror);
%c=Ant(1,universe,ant_sight_range,universe.nest{1},1);
b=lightCompass(b);
%c=lightCompass(c);
plotWorld(b,universe)

clear_counter=1;
for n=1:end_time

    b=walk(b,universe);
    %c=walk(c,universe);
    clear_counter=clear_counter+1;
    if clear_counter==30
        close gcf
        clear_counter=1;
    end

    plotWorld(b,universe);
    hold on


    quiver(b.position(1),b.position(2),b.dir_lookup{b.direction}(1),
b.dir_lookup{b.direction}(2),0,'color',[1,1,1])
    if ~isempty(b.meanVector)
        quiver(b.position(1),b.position(2),b.meanVector(1),
b.meanVector(2),0,'color',[1,0,1])
    end
    if ~isempty(b.feeder_meanVector)
        quiver(b.position(1),b.position(2),b.feeder_meanVector(1),
b.feeder_meanVector(2),0,'color',[0,0,1])
    end
    plotAnt(b)
    %plotAnt(c) plotTrail(b) plotTrail(c) drawCircle(b)
    hold off
    pause(0.01);


    %k = waitforbuttonpress;
end
```

```
random first walk
random walk
random walk
random walk
random walk
```



Published with MATLAB® R2012b

```
classdef World
    properties
        size                % world is composed of size x size fields
        nest                % location of the nest in the world
        food                % matrix containing location of the food sources in the world
        ants                % matrix containing location of all ants in the world and
their velocity vectors
        landmarks           % matrix containing location of landmarks
        obstacles           % matrix containing location of fields that ants cannot pass
through
        taken               % matrix with locations of all taken fields (free are 0, rest
of numbers denote what type of object is in that spot)
    end
    methods (Access = public)
```

9.1 Initialize world

```
        function obj=World(n)
            obj.size = n;                                       % assign size to the world
            obj.nest{1}=[round(n/2),round(n/2)];                % place nest in the middle
            obj.taken=zeros(n,n);

            obj.taken(1,:)=ones(1,n)*4;                         % fill world boundaries with
obstacles
            obj.taken(:,1)=ones(n,1)*4;                         % fill world boundaries with
obstacles
            obj.taken(n,:)=ones(1,n)*4;                         % fill world boundaries with
obstacles
            obj.taken(:,n)=ones(n,1)*4;                         % fill world boundaries with
obstacles
            obj=fillTaken(obj,obj.nest,2);                      % call function that will
mark spot of the nest as "taken" and unavailable for other items (landmarks, food, etc)
        end
```

```
Error using World (line 15)
Not enough input arguments.
```

9.2 Fill taken

```
        function obj=fillTaken(obj,array,color)                       % function that marks
locations stored in "array" as ones in obj.taken array (ones = taken and unavailable)
            size_array=numel(array);                            % check what size is the
array
            for n=1:size_array
                obj.taken(array{n}(1),array{n}(2))=color;       % for each position
stored in "array" mark corresponding point in obj.take as one
            end
        end
```

9.3 Place landmarks

```
        function obj=placeLandmarks(obj,landmark_amount,loc_land)

            if isempty(loc_land)

                for n=1:landmark_amount
                    max_pos=obj.size;
                    istaken=1;
                    while istaken~=0                            % randomly generate
location of a landmark and check if it is not taken. If it is, repeat
                        x_landmark=randi([1,max_pos]);
                        y_landmark=randi([1,max_pos]);
                        istaken=obj.taken(x_landmark,y_landmark);
                    end
                    obj.landmarks{1,n}=[x_landmark,y_landmark];
                end

            else
                for m=1:numel(loc_land)
                obj.landmarks{1,m}=loc_land{m};
                end
                landmark_amount=numel(loc_land);                % store the amount of
landmarks
            end

            % assign a local vector to each landmark
            landmark_temp=[];
            for n=1:landmark_amount
                obj.landmarks{2,n}=obj.nest{1}-obj.landmarks{1,n};
                landmark_temp{n}=obj.landmarks{1,n};
            end
            obj=fillTaken(obj,landmark_temp,3);
        end
```

9.4 Place obstacles

```
    function obj=placeObstacles(obj,obs_amount,loc_obs)


        obj.obstacles=[];

        if isempty(loc_obs)

            for n=1:obs_amount
                max_pos=obj.size;
                istaken=1;
                while istaken~=0                            % randomly generate
location of a obstacle and check if it is not taken. If it is, repeat
                    x_obs=1+round((max_pos-2)*rand(1));
                    y_obs=1+round((max_pos-2)*rand(1));
                    istaken=obj.taken(x_obs,y_obs);
                end
                obj.obstacles{n}=[x_obs,y_obs];
            end

            %now grow the obstacles into random direction

            obs_amount=numel(obj.obstacles);
            dir_lookup={[1,1],[0,1],[-1,1],[-1,0],[-1,-1],[0,-1],[1,-1],[1,0]};
            for m=1:obs_amount
                current_obs=obj.obstacles{m};
                growth_dir=randi([1,8]);                % direction 1 - right, 2-
right up, 3 up an so on counter clockwise
                growth_amount=randi([round(obj.size/8),round(obj.size/4)]);
```

```
                    for l=1:growth_amount


growth_field=[current_obs(1)+dir_lookup{growth_dir}(1),current_obs(2)+dir_lookup{growth_d
ir}(2)];
                        while growth_field(1)>obj.size || growth_field(2)>obj.size ||
growth_field(1)<=0 || growth_field(2)<=0
                            %disp('poza')
                            growth_dir=randi([1,8]);

growth_field=[current_obs(1)+dir_lookup{growth_dir}(1),current_obs(2)+dir_lookup{growth_d
ir}(2)];
                        end

                        istaken=obj.taken(growth_field(1),growth_field(2));

                        while istaken~=0
                            %disp('zajete')
                            growth_dir=randi([1,8]);

growth_field=[current_obs(1)+dir_lookup{growth_dir}(1),current_obs(2)+dir_lookup{growth_d
ir}(2)];
                            istaken=obj.taken(growth_field(1),growth_field(2));
                        end

                        size_obs_matrix=numel(obj.obstacles);
                        obj.obstacles{size_obs_matrix+1}=growth_field;

                        % if it's growing at an angle (not vertical or
                        % horizontal) add another field, one to the right of
                        % the chosen one, to make it ant-tight
                        if norm(dir_lookup{growth_dir})>1
                            growth_field_bis=growth_field+[1 0];
                            size_obs_matrix=numel(obj.obstacles);
                            obj.obstacles{size_obs_matrix+1}=growth_field_bis;
                        end

                        current_obs=growth_field;
                    end
                end

            else
                for m=1:numel(loc_obs)
                obj.obstacles{m}=loc_obs{m};
                end
            end

            obj=fillTaken(obj,obj.obstacles,4);

            %include edges as obstacles
            xx=(1:1:obj.size);
            yy=ones(1,obj.size);
            zz=yy*obj.size;
            top=[xx;yy];                            % coordinates of all points belonging
to top boundary of the map
            bottom=[xx;zz];                         % coordinates of all points belonging
to bottom boundary of the map
            left=[yy;xx];          % coordinates of all points belonging to left boundary
of the map, without corners
            right=[zz;xx];         % coordinates of all points belonging to right boundary
of the map, without corners
            for n=1:obj.size
                obj.obstacles{numel(obj.obstacles)+1}=[top(1,n),top(2,n)];
                obj.obstacles{numel(obj.obstacles)+1}=[bottom(1,n),bottom(2,n)];
                obj.obstacles{numel(obj.obstacles)+1}=[left(1,n),left(2,n)];
                obj.obstacles{numel(obj.obstacles)+1}=[right(1,n),right(2,n)];
            end
         end
```

9.5 Place food

```
        function obj=placeFood(obj,food_amount,min_nest_dist,loc_food)

            if isempty(loc_food)
                max_nest_dist=floor(obj.size/2);    % define max distance so the food is
not placed outside the world

                for n=1:food_amount                 % loop to randomly place food in the
world but at a minimum distance from the nest

                    istaken=1;
                    while istaken~=0 || food_x_world<=0 || food_y_world<=0
                        r=randi([round(min_nest_dist*obj.size), max_nest_dist]);   %
generate random r polar coordinate in between max/min distance
                        angle=degtorad(round(rand(1)*360));                  % gen random
angle from 0 to 360 and convert to radians
                        [food_x,food_y]=pol2cart(angle,r);                   % convert
polar coordinates to cartesian, with respect to nest location
                        food_x_world=round(food_x+round(obj.size/2));        % store food
location, but with respect to [0,0] point of the world
                        food_y_world=round(food_y+round(obj.size/2));        % store food
location, but with respect to [0,0] point of the world
                        if food_x_world==0
                            food_x_world=1;
                        end
                        if food_y_world==0
                            food_y_world=1;
                        end
                        istaken=obj.taken(food_x_world,food_y_world);        % check if
the field is taken (have value of 1 in table obj.taken)
                    end

                    obj.food{n}=[food_x_world,food_y_world];

                end
            else
```

```
                for m=1:numel(loc_food)
                obj.food{m}=loc_food{m};
                end
            end

            obj=fillTaken(obj,obj.food,5);
        end

    end
end
```

Published with MATLAB® R2012b

```
classdef Ant

    properties
        localVector         % stores local vectors visible to ant
        globalVector        % stores ants global vector (in ant-tied coordinate system)
        lightVector         % stores current e-vector
        feederFound         % stores location of found feeders
        feeder_globalVector % stores global vector pointing to feeder in ant coordinate
system
        feeder_meanVector   % stores mean vector pointing to feeder in global coordinate
system
        meanVector          % contains mean vector, calculated based on global, local and
e-vector
        mode                % defines if ant is foraging or homing (1 - foraging, 2 -
homing)
        direction           % stores direction in a form 1 - 9 (dir_lookup) of a previous
movement
        sightRange          % defines how good is ant sight
        fieldOfVision       % stores objects in area that the ant can see
        trail               % stores current trail of ant
        pastTrails          % stores history of runs
        nestDist            % stores the value of distance at which ant is in relatino to
nest
        position            % stores ant position on map, not used for navigation
        dir_lookup={[1,1],[0,1],[-1,1],[-1,0],[-1,-1],[0,-1],[1,-1],[1,0]} % vector
```

```
containing all possible directions of movement (starting from right, then right up and so
on in counter clockwise manner)
        obstacleEncountered % flag, marking if ant tried to run into obstacle
        intended_dir        % in case ant encounteres obstacle, stores value of intended
direction, for the whole time obstacle is blocking the way
        walkType            % informas what type of movement ant is engaded in
        e_ant_angle         % ant_angle of ant body with respect to lightVector (e-
vector)
        rot_error           % arbitrary assumed rotation error, which is multiplied by
rotation in global vector calculation
        spiral              % used for ant looking for nest when can't find it when
homing
        spiral_turn         % as above
        spiral_step         % as above
    end

    methods
```

9.6 Initialize/create ant

```
        function obj=Ant(mode,world,sightRange,position,rotationerror)


            obj.mode=mode;                % 1=foraging, 0=returning to nest
            obj.sightRange=sightRange;
            obj.feederFound=false;        % feeder is not known at the beginning
            obj.globalVector=[0 0];
            obj.trail{1}=position;
            obj.position=position;
            obj.nestDist=norm(obj.position-world.nest{1});
            obj.obstacleEncountered=false;
            obj.rot_error=rotationerror;
            obj.spiral=1;
            obj.spiral_turn=1;
            obj.spiral_step=1;

        end
```

```
Error using Ant (line 35)
Not enough input arguments.
```

9.7 Ant sight- define what objects ant see in its field of view

```
        function obj=antSight(obj,world)

            obj.fieldOfVision=[];
            counter=numel(world.food);                      % check how many of
object type X exists in the world

            m=1;                                            % reset counter
            obj.fieldOfVision{1}=[];                        % preallocate (in case
none of these is in vision, still there's an empty cell)
```

```
            for n=1:counter                                     % for each existing
object of type X
                distance=world.food{n}-obj.position;            % check how far it is
from ant
                if norm(distance)<=obj.sightRange               % and if it is within
sight range
                    obj.fieldOfVision{1}{m}=distance;               % store in repsective
cell in obj.fieldOfVision variable
                    m=m+1;                                          % increase counter
                end
            end

            counter=numel(world.nest);                          % REPETITION OF THE ABOVE

            m=1;
            obj.fieldOfVision{2}=[];
            for n=1:counter
                distance=world.nest{n}-obj.position;
                if norm(distance)<=1/3*obj.sightRange           % HARDER TO SEE NEST than
anything else
                    obj.fieldOfVision{2}{m}=distance;
                    m=m+1;
                end
            end

            counter=numel(world.landmarks)/2;                   % REPETITION OF THE ABOVE
BUT!!!!

            m=1;
            obj.fieldOfVision{3}=[];
            obj.localVector=[];                                 % ALSO STORES LOCAL
VECTORS ACCOMPANYING EACH LANDMARK
            for n=1:counter
                distance=world.landmarks{1,n}-obj.position;
                temp_localVector=world.landmarks{2,n};
                if norm(distance)<=obj.sightRange               % only if it lies within
ant's sight range ofcourse
                    obj.fieldOfVision{3}{m}=distance;
                    obj.localVector{m}=temp_localVector;
                    m=m+1;
                end
            end

            counter=numel(world.obstacles);

            m=1;
            obj.fieldOfVision{4}=[];
            for n=1:counter
                distance=world.obstacles{n}-obj.position;
                if norm(distance)<=obj.sightRange
                    obj.fieldOfVision{4}{m}=distance;
                    m=m+1;
                end
            end

        end
```

9.8 Store and work with local vector (memory of landmarks)

```
        function obj=landmark(obj)
            loc_vec_x=0;
            loc_vec_y=0;
            loc_vec_counter=numel(obj.localVector);

            for n=1:loc_vec_counter
                obj.localVector{n}=obj.localVector{n}+obj.fieldOfVision{3}{n};      % sum
local vector and vector pointing from ant to landmark to get vector pointing from ant to
nest
                loc_vec_x=obj.localVector{n}(1)+loc_vec_x;
                loc_vec_y=obj.localVector{n}(2)+loc_vec_y;
            end

            % average all seen local vectors
            loc_vec_x=loc_vec_x/loc_vec_counter;
            loc_vec_y=loc_vec_y/loc_vec_counter;
            obj.localVector=[loc_vec_x, loc_vec_y];

        end
```

9.9 Path integration

```
        function obj=pathIntegration(obj)

            ant_orientation=obj.dir_lookup{obj.direction};
            previous_e_ant_angle=obj.e_ant_angle;
            obj.e_ant_angle=radtodeg(atan2(ant_orientation(1)*obj.lightVector{1}(2)-
ant_orientation(2)*obj.lightVector{1}(1),ant_orientation(1)*ant_orientation(2)+obj.lightV
ector{1}(1)*obj.lightVector{1}(2)));  % get ant_angle between two vectors 0-2pi
http://www.mathworks.com/matlabcentral/newsreader/view_thread/151925

            if obj.e_ant_angle<0                                 % THIS FUNCTION GIVES
WRONG RESULTS FOR NEGATIVE ANGLES, this compensates
                obj.e_ant_angle=-(180+obj.e_ant_angle);
            elseif obj.lightVector{1}+ant_orientation==[0 0]     % for some reason method
cannot see 180 degrees - this compensates for that!
                obj.e_ant_angle=180;
            end

            rotation=-obj.e_ant_angle+previous_e_ant_angle;     % rotation = rotation
rate, because each rotation is done in one unit time
            rotation=rotation*obj.rot_error;                     % include rotational
error  !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
            velocity=norm(ant_orientation);                      % ant velocity is equal
to it's previous step divided by one time unit, as each step is done in one time unit
```

```
            obj.globalVector=obj.globalVector*[cosd(rotation), -sind(rotation);
sind(rotation), cosd(rotation)];    % rotate global vector to a new frame, to which ant
rotated
            obj.globalVector(2)=obj.globalVector(2)-velocity;   % add ant translation in
y direction (ant always moves in Y direction in its frame of reference)

            if obj.feederFound                                 % in case location of
feeder was found, perform path integration on the way back to nest, and store global
vector pointing to feeder
                obj.feeder_globalVector=obj.feeder_globalVector*[cosd(rotation), -
sind(rotation); sind(rotation), cosd(rotation)];
                obj.feeder_globalVector(2)=obj.feeder_globalVector(2)-velocity;
            end

        end
```

9.10 Obtain vector from position of sun / light polarization

```
        function obj=lightCompass(obj)
            % define simplified polarizationCompass in terms of one of the
            % 8 directions present in ant's world
            obj.lightVector{1}=obj.dir_lookup{randi([1,8])};
            ant_initial_vec=[0 1];
            obj.e_ant_angle=radtodeg(atan2(ant_initial_vec(1)*obj.lightVector{1}(2)-
ant_initial_vec(2)*obj.lightVector{1}(1),ant_initial_vec(1)*ant_initial_vec(2)+obj.lightV
ector{1}(1)*obj.lightVector{1}(2)));    % at first ant is oriented as the global refence
system
            if obj.e_ant_angle<0                                  % THIS FUNCTION GIVES
WRONG RESULTS FOR NEGATIVE ANGLES, this compensates
                obj.e_ant_angle=-(180+obj.e_ant_angle);
            elseif obj.lightVector{1}+ant_initial_vec==[0 0]       % for some reason
method cannot see 180 degrees - this compensates for that!
                obj.e_ant_angle=180;
            end
        end
```

9.11 Navigate to feeder

```
        function obj=feederVec(obj)
            ant_dir=obj.dir_lookup{obj.direction};
            global_vec=[0 1];
            theta=radtodeg(atan2(ant_dir(1)*global_vec(2)-
ant_dir(2)*global_vec(1),ant_dir(1)*ant_dir(2)+global_vec(1)*global_vec(2)));  % get
ant_angle between two vectors
http://www.mathworks.com/matlabcentral/newsreader/view_thread/151925
            if theta<0                              % THIS FUNCTION GIVES WRONG RESULTS
FOR NEGATIVE ANGLES, this compensates
                theta=-(180+theta);
            elseif ant_dir+global_vec==[0 0]        % for some reason method cannot see
180 degrees - this compensates for that! (sum of vectors = 0 0 means vectors have 180
degrees difference)
```

```
                theta=180;
            end

            vector_feederCoord=obj.feeder_globalVector*[cosd(theta), -sind(theta);
sind(theta), cosd(theta)];               % transform ant's global vector to global
coordinates
            obj.feeder_meanVector=vector_feederCoord;
        end
```

9.12 Use all available tracking mechanisms to devise a walking direction to nest

```
        function obj=meanVec(obj)
            ant_dir=obj.dir_lookup{obj.direction};
            global_vec=[0 1];
            theta=radtodeg(atan2(ant_dir(1)*global_vec(2)-
ant_dir(2)*global_vec(1),ant_dir(1)*ant_dir(2)+global_vec(1)*global_vec(2)));  % get
ant_angle between two vectors
http://www.mathworks.com/matlabcentral/newsreader/view_thread/151925
            if theta<0                                  % THIS FUNCTION GIVES WRONG RESULTS
FOR NEGATIVE ANGLES, this compensates
                theta=-(180+theta);
            elseif ant_dir+global_vec==[0 0]       % for some reason method cannot see
180 degrees - this compensates for that! (sum of vectors = 0 0 means vectors have 180
degrees difference)
                theta=180;
            end

            vector_globCoord=obj.globalVector*[cosd(theta), -sind(theta); sind(theta),
cosd(theta)];               % transform ant's global vector to global coordinates

            if ~isempty(obj.localVector) && obj.walkType==6 % if there's a local vector
in range, calculate mean of it and global vector as a direction


obj.meanVector=[(vector_globCoord(1)+obj.localVector(1))/2,(vector_globCoord(2)+obj.local
Vector(2))/2];
                obj.globalVector=obj.meanVector*[cosd(-theta), -sind(-theta); sind(-
theta), cosd(-theta)]; % correct obj.globalVector based on landmark

            else
                obj.meanVector=vector_globCoord;
            end

            if norm(vector_globCoord)<1 && obj.mode==2
                obj.mode=3;
            end
        end
```

9.13 Walk

```
        function obj=walk(obj,world) %define walking direction, depending on ant goal
(homing / foraging) and world (seeing objects, walking into objects)

            % invoke antSight function to let the ant "look around"
            obj=antSight(obj,world);


            % FORAGING MODE
            % ---------------------------------------------------------------------------
-------------------------------------------------------
            if obj.mode==1

                move_vector=[];
                % If this is the first step, pick direction at random
                if isempty(obj.direction)  && ~obj.obstacleEncountered &&
~obj.feederFound
                    disp('random first walk')
                    %obj.direction=randi([1,8]);
                    obj.direction=1;
                    obj.walkType=1;                                             %
walkType equal to 1 means ant is foraging and performing first random stemp

                    % If no food is seen, pick a direction at random,
                    % chosen from a normal distribution based on previous
                    % direction, provided there's no obstacle in the way
                elseif ~obj.obstacleEncountered && ~obj.feederFound &&
isempty(obj.fieldOfVision{1})
                    disp('random walk')
                    obj.walkType=3;                                              %
walkType equal to 3 means ant is foraging and not seeing any food

randDirection=round(normrnd(obj.direction,0.1+obj.nestDist/world.size/1.5));
                    while randDirection>8 || randDirection<1

randDirection=round(normrnd(obj.direction,0.1+obj.nestDist/world.size/1.5));
                    end
                    obj.direction=randDirection;

                    % If the ant can "see" any food all calcualations are
                    % overwriten and the move direction points directly
                    % towards the feeder
                elseif ~isempty(obj.fieldOfVision{1}) && ~obj.obstacleEncountered
                    disp('food in range')
                    obj.walkType=2;                                             %
walkType equal to 2 means ant is foraging and seeing food

                    food_in_sightAmount=numel(obj.fieldOfVision{1});            % how
many food items ant can see
                    for n=1:food_in_sightAmount                                 % for
every seen food item
                        dist_to_seen_food(n)=norm(obj.fieldOfVision{1}{n});     %
calculate distance to ant
```

```
                    end

                    [~, closest_food]=min(dist_to_seen_food);                    % find
indices of closest food in fieldOfVision matrix
                    move_vector=obj.fieldOfVision{1}{closest_food};              %
define vector pointing to closesd food item seen by ant

                    % in case ant knowns position of any feeder from a
                    % previous run, navigate to that feeder using
                    % food_globalVector
                elseif obj.feederFound && ~obj.obstacleEncountered &&
isempty(obj.fieldOfVision{1})
                    disp('going back to feeder')
                    obj.walkType=2;                                              % is
equal to 2 because, behaves similarly to when food is seen
                    obj=feederVec(obj);
                    move_vector=obj.feeder_meanVector;

                end

                % evaluate move_vector, if exists (if ant sees food, it
                % WILL exist for sure, as walkType=2 will be invoked
                if ~isempty(move_vector)
                    if norm(move_vector)<1                                      % if ant
arrived at FOOD location
                        obj.mode=2;                                              %
change mode to homing (NOT ASSIGN DIRECION, HOMIN WILL)
                        obj.feederFound=true;
                        obj.feeder_globalVector=[0 0];                             %
reset global vector pointing to feeder
                    else
                        [ant_angle,~] = cart2pol(move_vector(1),move_vector(2));   %
based on vector, get polar coordinates pointing to closest food item direction
                        ant_angle=rad2deg(ant_angle);                            %
convert radians to degrees
                        if ant_angle<0
                            ant_angle=360+ant_angle;                             %
convert negative ant_angles to positive ones (0-360 instead of 0-180 and 0- -180)
                        elseif ant_angle==0                                      %
prevent confusion when ant_angle = 0
                            ant_angle=1;
                        end
                        dir=round(ant_angle/45);
                        if dir==0                                                 % we
have 8 directions only so dir 0 equals dir 8
                            dir=8;
                        end

                        obj.direction=dir;                                        %
store the new direction
                    end
                end
```

```
            end

            % HOMING MODE
            % --------------------------------------------------------------------------------
-----------------------------------------------------
            if obj.mode==2

                % If the ant can "see" the nest all calcualations are
                % overwriten and the move direction
                % poinobj=antSight(obj,world)ts directly towards the
                % feeder.

                if ~obj.obstacleEncountered && ~isempty(obj.fieldOfVision{2})
                    disp('homin seeing nest')
                    obj.walkType=4;                                              %
walkType equal to 4 means ant is homing with nest in range of sight
                    move_vector=world.nest{1}-obj.position;                      %
define vector pointing to closesd food item seen by ant

                elseif ~obj.obstacleEncountered && isempty(obj.localVector) &&
isempty(obj.fieldOfVision{2}) % use navigational tools for homing
                    disp('homin')
                    obj=meanVec(obj);
                    move_vector=obj.meanVector;
                    obj.walkType=5;                                              %
walkType equal to 5 means ant is homing

                elseif ~obj.obstacleEncountered && ~isempty(obj.localVector) &&
isempty(obj.fieldOfVision{2})
                    disp('homin seeing landmark')
                    obj.walkType=6;                                              %
homing and seeing local vector
                    obj=landmark(obj);
                    obj=meanVec(obj);
                    move_vector=obj.meanVector;
                end

                % calculate direction movement based on defined move_vector
                if ~obj.obstacleEncountered
                    if norm(move_vector)==0           % if arrived to the NEST
                        obj.globalVector=[0 0];        % clear error in PI
                        obj.mode=1;                    % change mode to foraging

                        if isempty(obj.pastTrails)
                            obj.pastTrails{1}=obj.trail;     % store trail of this run in
history of trails, for the first run
                        else
                            obj.pastTrails{end}=obj.trail;    % store trail of this run
in history of trails
                        end

                        obj.trail=[];                  % clear current trail
                        obj.spiral=1;                  % reset spiral controls
                        obj.spiral_turn=1;
                        obj.spiral_step=1;

                        % start going back to feeder (same code as in mode
                        % 1, when ant is going back to feeder)
                        disp('going back to feeder')
                        obj.walkType=2;                % is equal to 2 because, behaves
similarly to when food is seen
                        obj=feederVec(obj);
                        move_vector=obj.feeder_meanVector;


                    end

                    [ant_angle,~] = cart2pol(move_vector(1),move_vector(2));      % based
on vector, get polar coordinates pointing to closest food item direction
                    ant_angle=rad2deg(ant_angle);                                 %
convert radians to degrees

                    if ant_angle<0
                        ant_angle=360+ant_angle;                                  %
convert negative ant_angles to positive ones (0-360 instead of 0-180 and 0- -180)
                    elseif ant_angle==0                                           %
prevent confusion when ant_angle = 0
                        ant_angle=1;
                    end

                    dir=round(ant_angle/45);
                    if dir==0                                                 % we have 8
directions only so dir 0 equals dir 8
                        dir=8;
                    end
                    obj.direction=dir;

                end


            end

            % CANT FIND NEST MODE - SPIRALING
            % ---------------------------------------------------------------------------
-----------------------------
            if obj.mode==3 && ~obj.obstacleEncountered
                disp('cant find nest')
                obj.walkType=7;
                obj.direction=obj.spiral_turn;                  % go in the direction
indicated by spiral_turn (1 for first move)

                if obj.spiral_step<obj.spiral                   % as long as you walked
less then spiral moves (1 for 1st, 2 for 2nd and so on) in spiral_turn direction,
continue in this one
                    obj.spiral_step=obj.spiral_step+1;
                else
                    obj.spiral_turn=obj.spiral_turn+1;          % if you walked amount
```

```
of steps in spiral_turn direction equal to number of spiral, change direction
                    obj.spiral_step=1;                          % reset step counter, as
new direction was chosen
                    if obj.spiral_turn==5                       % when half of turns are
done, increase spiral counter - PROVIDES UNIFORM SPIRAL for some reason
                        obj.spiral=obj.spiral+1;
                    end
                end

                if obj.spiral_turn==9                           % only 8 directions
available, so if you reached last one
                    obj.spiral=obj.spiral+1;                     % all of turns are
done, increase spiral counter - PROVIDES UNIFORM SPIRAL for some reason
                    obj.spiral_turn=1;                           % and reset directions
                end

                if ~isempty(obj.fieldOfVision{2})               % if nest is found,
revert to mode 2
                    obj.mode=2;
                end
            end

            % BASED ON CHOSEN DIRECTION, ASSIGN NEW FIELD TO WHICH ANT
            % SHALL GO
            chosenNew_position=obj.dir_lookup{obj.direction};


            % OBSTACLE AVOIDING CHECK
            % ---------------------------------------------------------------------------
---------------------------------------------
            if ~isempty(obj.fieldOfVision{4})
% if ant can see any obstacles in it's field of vision
                obstacle_in_the_way=find(cellfun(@(x) isequal(x,chosenNew_position),
obj.fieldOfVision{4}), 1);     % check if there's an obstacle in the chosen field

                while ~isempty(obstacle_in_the_way)
% while there is, choose a new direction (to the left or to the right of the chosen one)

                    disp('active avoiding obstacles');

                    if obj.walkType==2 || obj.walkType==4  || obj.walkType==5 ||
obj.walkType==6 || obj.walkType==7 % engage obstacle avoiding in case ant has a
motiviation to go a certain way (to food / to nest)
                        disp('stored dir')
                        obj.obstacleEncountered=true;           % MARK THE FLAG
                        obj.intended_dir=obj.direction;         % STORE INTENDED
DIRECTION
                    end


                    neighbouringObs=obj.fieldOfVision{4}(find(cellfun(@(x) norm(x),
obj.fieldOfVision{4})<2));  % check if neighbouring cells contain obstacle

                    for n=1:numel(neighbouringObs)
```

```
                        forbidden_dir(n)=(find(cellfun(@(x)
isequal(x,neighbouringObs{n}), obj.dir_lookup)));   % list forbidden directions (i.e.
those having obstacles)
                    end

                    forbidden_dir=sort(unique(forbidden_dir));
% sort forbidden directions and remove duplplicates if present

                    if ismember(8,forbidden_dir) && ismember(1,forbidden_dir)
% if dir 1 and dir 8 are both forbidden, then, ant must used allowed dir instead
                        counter=1;
                        for n=1:8
                            if ~ismember(n,forbidden_dir)           % check if n belongs
to forbidden_dir
                                allowed_dir(counter)=n;             % store are allowed
directinos (empty fields around ant)
                                counter=counter+1;                  % raise counter
                            end
                        end
                        way(1)=allowed_dir(1);                      % first possible way
                        way(2)=allowed_dir(end);                    % second possible way
                    else
                        way(1)=forbidden_dir(1)-1;                  % first possible way
is one lower than lowest value (e.g. if there's an obstacle at directions from 7 to 8
then one can go to dir 6)
                        way(2)=forbidden_dir(end)+1;                % second possible way
is one higher than highest value (e.g. if there's an obstacle at directions from 7 to 8
then one can go to dir 9)
                    end


                    if way(1)==0                                    % only 8 possible
directions so 0 = 8
                        way(1)=8;
                    end
                    if way(2)==9                                    % only 8 possible
directions so 9 = 1
                        way(2)=1;
                    end

                    choice=randi([1,2]);                            % choose between two
directions closest to the obstacle - track it's boundary
                    chosenNew_position=obj.dir_lookup{way(choice)};
                    obj.direction=way(choice);                      % update ant
direction (body orientation)
                    obstacle_in_the_way=find(cellfun(@(x) isequal(x,chosenNew_position),
obj.fieldOfVision{4}), 1); % check again for obstacle in the new chosen field
                end
                % if the obstacle is not blocking the way, this does
                % nothing
            end


            % IF AN OBSTACLE WAS HIT, obj.obstacleEncountered FLAG IS
            % CHECKED, THEN ONLY THE PART OF CODE BELOW IS EXECUTED which
            % results in ant keeping the direction along obstacle UNLESS
            % the direction that was intended when at first found obstacle
            % is clear in that case, go the intended direction and CLEAR
            % THE obj.obstacleEncountered FLAG
            if obj.obstacleEncountered                                      % if ant hit
an obstacle before
                disp('keeping obs avoid direction')                         % check, if
there's still an obstacle in direction ant inteded to go
                obstacle_in_inteded_pos=find(cellfun(@(x)
isequal(x,obj.dir_lookup{obj.intended_dir}), obj.fieldOfVision{4}), 1);

                if isempty(obstacle_in_inteded_pos)                         % if there's
no obs in that direction
                    disp('keeping intended direction')
                    obj.obstacleEncountered=false;                          % change
obj.obstacleEncountered flag
                    obj.direction=obj.intended_dir;                         % go in the
intended direction
                    chosenNew_position=obj.dir_lookup{obj.direction};       % if dir that
was intended when ant encountered obstacle is free, go this way
                end

            end

            % UPDATE POSITION
            % ----------------------------------------------------------------------------
------------------------------------------------

            obj.position=chosenNew_position+obj.position;   % update position of ant
            obj.nestDist=norm(obj.position-world.nest{1});  % update ant's distance to
nest - THIS IS NOT AVAILABLE FOR PI, just for calculating normal random distr in random
movements
            obj.trail{end+1}=obj.position;                  % update trail variable
(append at the end)
            obj=pathIntegration(obj);                       % update global vector
        end

    end
end
```

Published with MATLAB® R2012b

```
function plotAnt(array)

%max=length(array.position);
%for n=1:max
    x=array.position(1);
    y=array.position(2);
%end
```

```
plot(x,y,'Color','red')
```

```
Undefined variable array.

Error in plotAnt (line 5)
    x=array.position(1);
```

Published with MATLAB® R2012b

```
function plotTrail(array)

max=length(array.trail);
for n=1:max
    x(n)=array.trail{n}(1);
    y(n)=array.trail{n}(2);
end


plot(x,y,'color','white','lineWidth',2)
```

```
Error using plotTrail (line 3)
Not enough input arguments.
```

Published with MATLAB® R2012b

```
function plotWorld(b,universe)
%plot
hold on
xlim([1 universe.size])
ylim([1 universe.size])
colormap ([0 0.5 0; 1 1 1; 1 0 0; 0 0 0; 1 1 0])
image(universe.taken')
axis ('square')
hold off
end
```

```
Error using plotWorld (line 4)
Not enough input arguments.
```

Published with MATLAB® R2012b

```
function drawCircle(obj)
%x and y are the coordinates of the center of the circle
%r is the radius of the circle
%0.01 is the angle step, bigger values will draw the circle faster but
%you might notice imperfections (not very smooth)
x=obj.position(1);
```

```
y=obj.position(2);
r=obj.sightRange;

ang=0:0.01:2*pi;
xp=r*cos(ang);
yp=r*sin(ang);
plot(x+xp,y+yp);
end
```

```
Undefined variable obj.

Error in drawCircle (line 6)
x=obj.position(1);
```

Published with MATLAB® R2012b

10