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 Behaviour: Simulating Movement and Navigation

Georg Wiedebach & Wolf Vollprecht

Zurich
December 2011

Agreement for free-download

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

Georg Wiedebach
georgwi@student.ethz.ch

Wolf Vollprecht
wolfv@student.ethz.ch

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

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.

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

Abstract

This paper is the final result of the course MODELING SOCIAL SYSTEMS WITH MATLAB™ which aimed to offer an insight into the MATLAB™programming language and to use said language to model social systems with various approaches. The timeframe of the course is one semester.
In this paper we will try to show how to replicate the behaviour of desert ants in a MATLAB™simulation. Furthermore we will discuss our results and compare them to experimental results obtained by biologists.

Contents

1 Individual contributions

The whole project was done in a cooperative manner.

2 Introduction and Motivations

We think ants are exciting animals because – despite their small body mass and therefore small brain – they form complex social structures. Great numbers of them work together efficiently. This requires a high level of coordination. We have already seen some videos which highlight the great achievements of ant colonies in building and hunting. When we found out about their navigation abilities we were curious to study how ants are able to cover extreme distances compared to their bodylength. The human being would definitely get lost when trying to journey this far in the desert without GPS or any other form of modern help, so one of our main goals will be to find out how ants can master this difficult task[1].

Ants have been subject of modern research since 1848, the reason often was interest in their instincts, society and of course the hope to learn from them. Studies in ant movement became even more challenging when scientists started to search for algorithms which solve the fundamental task of finding the shortest way in a graph (Graph Theory[7]). The class of ant colony optimization algorithms was introduced 1992 and has since been a field of active study[6].

However, these algorithms are modeling the behaviour of forest ants of the western hemisphere, which is not similar to the behaviour of desert in terms of choosing a good path and searching for food. Since we are studying desert ants we had to take a different approach. Desert ants rely much more heavily on the few landmarks they find in their environment and less on pheromone tracks other ants have left before them, which is what forest ants do. They make use of a path integraton which enables them to track their position in reference to where they started the journey, most likely the nest.

Results of interest are:

- How optimized and efficent is navigation by vectors
- How well do the ants learn in the course of repeated journey towards the food and back
- Of course we were as well motivated to improve our knowledge of MATLAB™

3 Description of the Model

Our aim was to create a model of desert ant behaviour, that reproduces the real ant behaviour as acurate as possible. This will include their search for food, their returning to the nest and their orientation with global and local vectors. Also we are going to test how close our algorithms are to real ant movement. Therefore we want to simulate the experiments described in the papers[5, 2]. Our model should be able to deal with different numbers of landmarks, obstacles and starting points. We would like to model our ant with the ability to learn and improve its efficiency when searching and finding food.

Because of the nature of the problem we chose to design our simulation around a time-discrete step-based model of an ant. We chose to simulate only one ant at a time, because we dont think that an higher number of ants would make much of a difference considering the vast space in the deserts. Therefore we can leave out influences of near ants like separation and cohesion (compare Agent Based Modeling[3]).

The simulation should be capable of finding a good path between the nest and feeder and by utilizing a simple learning process. We wanted to create a model, that can autonomously avoid obstacles and that does not get stuck in a corner. In order to meet this requirements we have split our simulation in two parts:

Landscape

Our landscape should contain all the information about the

- Position of the nest
- Position of the feeder
- Obstacles (stones, etc.), from which some can be used as landmarks

We chose to limit our landscape: We implemented fixed boundaries, which hinder the ant from escaping out of our experiment area. This is important to limit the time the ant needs to find food and thus making our simulation very less time-consuming. A boolean $n \times m$ - matrix stores all the information about taken and free points, where false represents a position the ant can not pass. Nest, feeder, landmarks and local vectors are saved separately as vectors, to make them easy to access (figure 1).

Ant

Our ant should follow certain, simple rules to move according to the studies we have received as part of the project description[4, 2, 5]. Such are basic rules like avoiding obstacles or a little more specific rules like following the global vector when

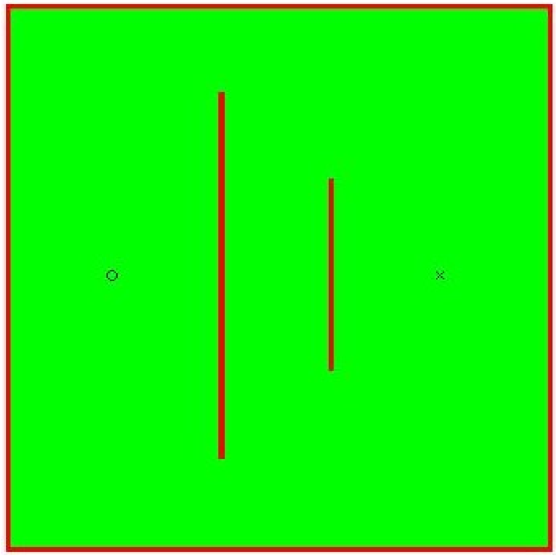


Figure 1: Example of a simple landscape: obstacles red, feeder and nest are indicated with x and o

returning to the nest and using the local vectors of the landmarks when searching for food again. During the simulation and after the ant has had success in finding food, the local vectors should as well change accordingly to the new found and possibly better path. If there are no local vectors yet, the model searches randomly until food is found.

3.1 Simplifications

There are some simplifications and assumptions we have made, the most important ones are:

- We decided to create fixed boundaries on our Landscape.
- For our model we strictly separate navigation by global vector (feeder to nest) and by local vectors (nest to feeder).
- The model will have a detection-radius in which landmarks, nest and feeder are recognized for moving and navigating (like a view radius). The detection radius does not put into consideration, that a real ant is not able to see through obstacles.

4 Implementation

As described above our simulation consists of two main parts: The landscape and the ant. We implemented both of them as separate classes. A third class, the simulation-class, that handles rendering, initialising and iterations. We also used a main-file in which we declared variables which often needed to be changed in order to influence the outcome of the simulation like the detection-radius of the ant or information about the map, that should be loaded into the simulation (figure 2).

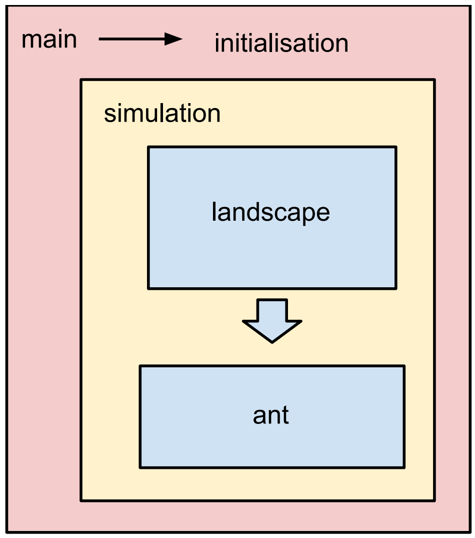


Figure 2: Flow chart of class-structure

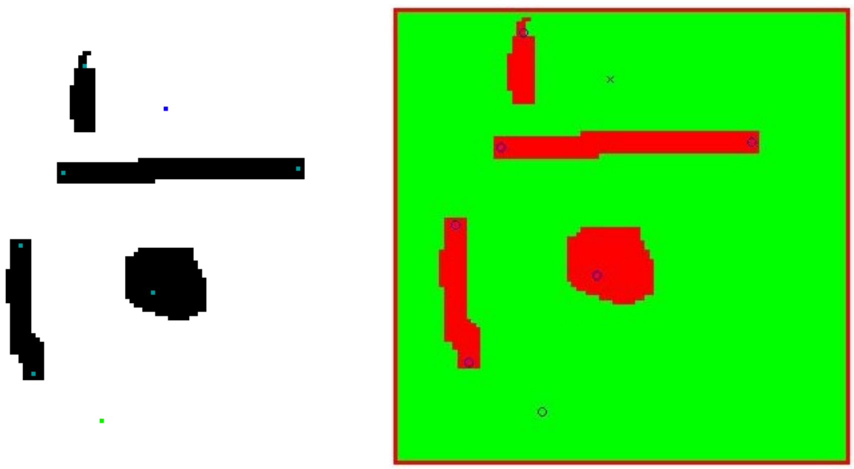
4.1 Landscape

The landscape class only contains information about the map, the nest and the feeder as well as some specified spots which are landmarks, used by the ant as anchor points for its local vectors.

We implemented different methods of loading landscapes into our simulation. Besides the possibility of creating the landscape-matrix in a separate m-file and the random-map generator we frequently used a simple but elegant method for generating maps out of arbitrary made generic Portable Network Graphics. The method detects specific color values and translates them into their meaning in the context of the landscape (figure 3 and table 1).

	Color in png-file	Color in MATLAB™
Obstacle	black	red
Nest	green	black circle
Feeder	blue	black cross
Landmark	turquoise	blue circle

Table 1: Color values and their meaning



(a) Original image (b) Generated landscape

Figure 3: Generated map from image

4.2 Ant

The class ant mainly contains the current position of the ant, the local vectors on landmarks and the path integrated global vector which always points back to the nest (as long as the ants moves are coherent). We built our ant around the method: move. The move function is invoked out of two different methods the find_food and the return_to_nest. In the following passage the most important methods of the class ant are described:

4.2.1 Find food

This loop iterates the move-method until the ant reaches the Feeder. Depending on how often the ant has already been on the track, it uses the aggregated local vectors to calculate a direction which the ant follows to reach the food faster. As soon as the feeder is in range of the detection radius the ant begins to run straight towards it.

4.2.2 Calculate the direction from landmarks

$$\vec{v}_{direction} = \sum_{i=0}^{n} \vec{l}_i \quad \forall ||\vec{l}_{i,pos} - \vec{a}_{pos}||_2 < r_{detection} \tag{1}$$

The method sums up all local vectors in detection radius of the ant and returns the resulting vetor. $\vec{l}_i$ are the local vectors, i ranging from the first to the last landmark and $r_{detection}$ is the detection radius of the ant.

4.2.3 Return to nest

When returning to the nest, the model uses the same move method as when searching, but instead of calculating a general direction out of the occurring local vectors the ant uses the global vector, which always leads straight back to the nest. While returning to the nest the model updates all local vectors while passing the related landmarks. In our implementation a local vector always points to the last landmark the ant passed by or is adjusted toward this position (see section below). Thereby the ant develops a steady route that is a close to the optimal route. This implementation must be tested for reliability later on.

4.2.4 Updating the local vectors on all landmarks

For the implementation we decided, as a simplification, that our model of the ant, would be able to remember only the last global vector where a landmark was spotted. For this reason the model, when spotting a new landmark always calculates a vector pointing to the latest landmark and thus developing a path that leads from the first landmark, the nest, to the latest, which should be quite close to the feeder. This implementation however will result in non-changing local vectors after the first run. So we included a asymptotic *grow-factor* (a), to prevent the local vectors from growing out of the boundaries. That only allows a small adjusting every time the ant passes the landmark. As a result the learning curve of the ant became much more interesting and realistic, as described below in the experimental results.

$$a = 0.5 * \exp\left(-\frac{||\vec{l}_i||_2}{10}\right) \tag{2}$$

$$\vec{l}_i = a * \mathrm{round}\left(\vec{l}_i + (\vec{g}_i - \vec{g}_{i-1})\right) \tag{3}$$

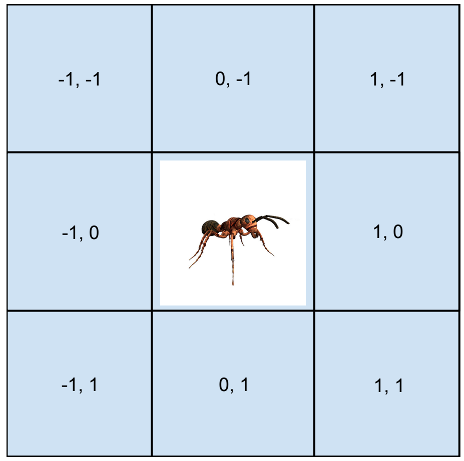


Figure 4: First order Moore neighbours

4.2.5 Move

The move method is the heart of the ant class: it accepts any general direction vector as input and sets the new position of the ant as a result. A general direction input can be calculated in the method find food or return to nest. It also handles all the checking for obstacles. Move is invoked in every time-step. Because the ant has only a choice of 8 possible next positions the method has to calculate a new direction vector to one of the first order Moore neighbours (figure 4).

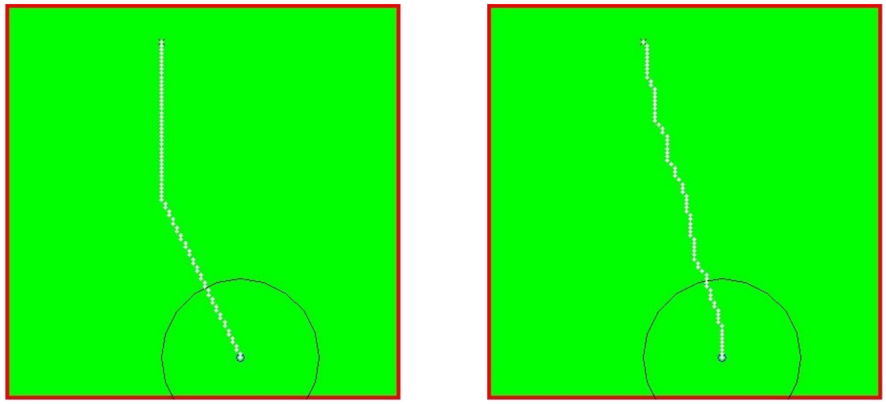
To calculate the matching Moore neighbour from the general direction vector we use the following formula and call the result the main-direction.

$$\vec{m} = \text{round}\left(\vec{dir} * \frac{1}{\max|dir_i|}\right) \tag{4}$$

In case the general direction is not exactly a multiple of a vector given by the Moore neighbourhood this calculation will result in a non-natural path. So we calculated a second-direction which is chosen as move direction with a certain probability depending on the angle between the general direction and the main direction. This allows the ant to walk directly towards a target (figure 5). We had to handle few special cases separately.

$$\vec{s} = \vec{m} - \vec{dir} * \min\left(|dir_i|\right) \tag{5}$$

$$p = \frac{\min|dir_i|}{\max|dir_i|} \tag{6}$$



(a) Without second direction (b) With second direction

Figure 5: Calculating move direction with and without second direction

If there is no general direction given to the method move, or if the general direction is zero a vector is generated based on the previous move direction. This vector then is turned around +/- 45 degree with a certain probability. This probability defines how twisted the ants path is. The figure 6 was taken with a low turning-probability (10 percent).

In the figure 6 it is easily seen, that the ant can not move trough obstacles. This is also part of the method move. Therefore the method checks the desired position on the map. If the position is not available the move-vector is turned around 45 degree clockwise or counter-clockwise, and the desired position is checked again until a possible step is found.

4.3 Simulation

The main purpose of the simulation class is to serve as a connector between the landscape class and the ant class. It also handles everything that has to do with output. The most important functions are the run-method and the render-method, these two are described below.

4.3.1 Run

In this method there are basically two while-loops checking whether the ant is searching for food or trying to return to the nest. This is indicated by two Boolean values in the class ant. In each cases the corresponding ant-methods are invoked. Here is a simple example of using parts of the run-method (pseudo code):

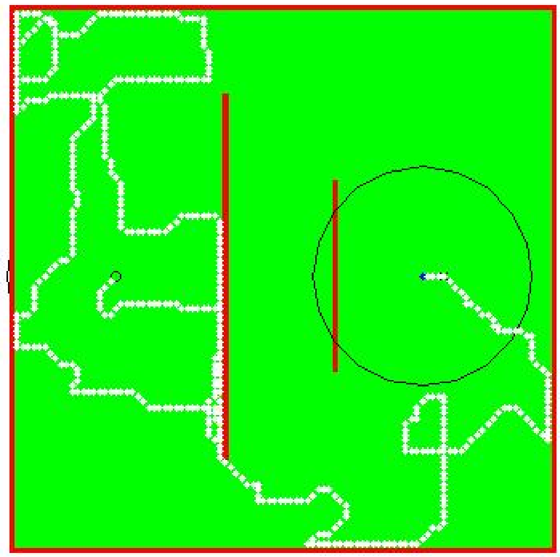


Figure 6: First search for food. No local vectors are set.

```
while the ant has no food
   search food and move one step
   if the simulation needs to be rendered
       render the actual position of the ant on the map
   end
end
```

4.3.2 Render

We soon realised that rendering the output while simulating is the main bottleneck in terms of time consumption, especially for the first random run. That is why we made it optional, so that it can be turned on or off for every simulation instance in the main-file. Another speed-improvement made here is achieved by not plotting the whole map but only the ant and the detection-radius. Local vectors are rendered in a different method after each successful returning to the nest. Finaly we introduced the option path_ render, which can be set true or false.

5 Simulation Results and Discussion

5.1 Expected Results

We expect our simulation to be able to find short paths from Point A to desired position B, in this case nest to feeder and feeder to nest. We tried to be as close to nature as possible and hope to be able to recreate some of the experimental results given. Some results we consider particularly interesting are avoiding obstacles on returning home and the use and improving of local vectors. Of course there sure are models of desert ant behaviour already, because these animals have been topic of research for a long time, but our simulation is mostly based on our own consideration so we are especially tense whether our results are accurate.

5.2 Experimental Results

5.2.1 Path Integration Experiment

The first run of our simulations completely reproduced a real experiment, given in the paper by R. Wehner[5]. The main purpose is to learn how effective and realistic the implemented model behaviour is.

In the experiment the ant is allowed to find food and then has to return to the nest with a correct global vector. To increase the difficulty two obstacles are placed between the nest and the feeder to study the return-behaviour of the ants. This interference is made after the ant reaches the nest without obstacles.

To reproduce this situation we set our ant at the feeder and adjusted the correct global vector to the nest. Our results (figure 7) are mostly the same as the results seen in the experiment. Only one thing differs in our model: The real ant always chooses the same direction to turn at the obstacle. This might be the result of a simple payoff-learning process, once effective there was no need for the real ant to change the way. Unfortunately the experiment is not repeated with a higher number of different ants.

In the two pictures in figure 7 two of our result-paths are shown. Obviously the simulated ant tries to move directly to the nest. Once there is a obstacle in the way the path turns randomly left or right until the obstacle is passed. Then the direct way to the nest is chosen again. During the time of wandering along the obstacle the global vector is adapted for every step the ant takes.

In this specific case the model is reproducing the real experiment very accurate. However there are still some open questions: whether the real ant at an obstacle indeed turns left or right based on a random decision is not clear. But as the experiment given only uses one single ant this question can not be answered.

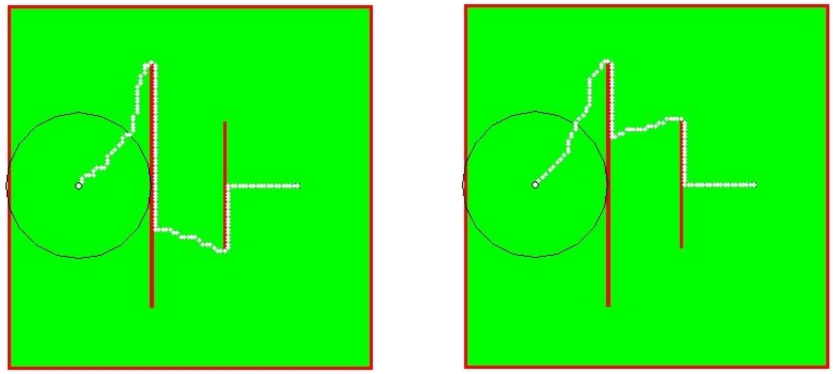


Figure 7: Returning to nest with global vector

5.2.2 Food searching by local vectors

In the implementation the local vectors are updated every time the ant reaches the nest. The figure 8 shows the path taken by the ant in her fourth run to the feeder. On each of the red obstacles there is a landmark with the associated local vector (yellow). If there is a landmark in the detection-radius of the ant, visualized by the black circle, the ant considers the local vector for the path. Of course the detection-radius is crucial in this experiment so we tried the same run with different values on a slightly different map.

In the figure three runs of the same simulation with different sizes of the detection-radius (black circle around the ant) are shown. It is obvious that the path becomes more direct, when the ant can see more landmarks at one time step, because the local vectors are summed up. In the first picture the smallest view radius is simulated and in the beginning it is easy to see, at which points the landmarks come into view. Between the second and third mark the ant looses the track for a short time.

5.2.3 Path Improvement

In the following section we have tested several maps and landmark-arrangements and graphed the number of steps the ant needed to find food and return to the nest. Of course once the local vectors were set and updated the step-number decreased. In figure 9 you can see the map we used for the results discussed below. An improvement can clearly be seen until the fourth or fifth time the ant has to reach the feeder. Then the local vectors are not improved anymore. Another noteworthy fact in this graph is, that the navigation by local vectors never reaches the same level of effectiveness as navigating back to the nest by the global vector. The exact results

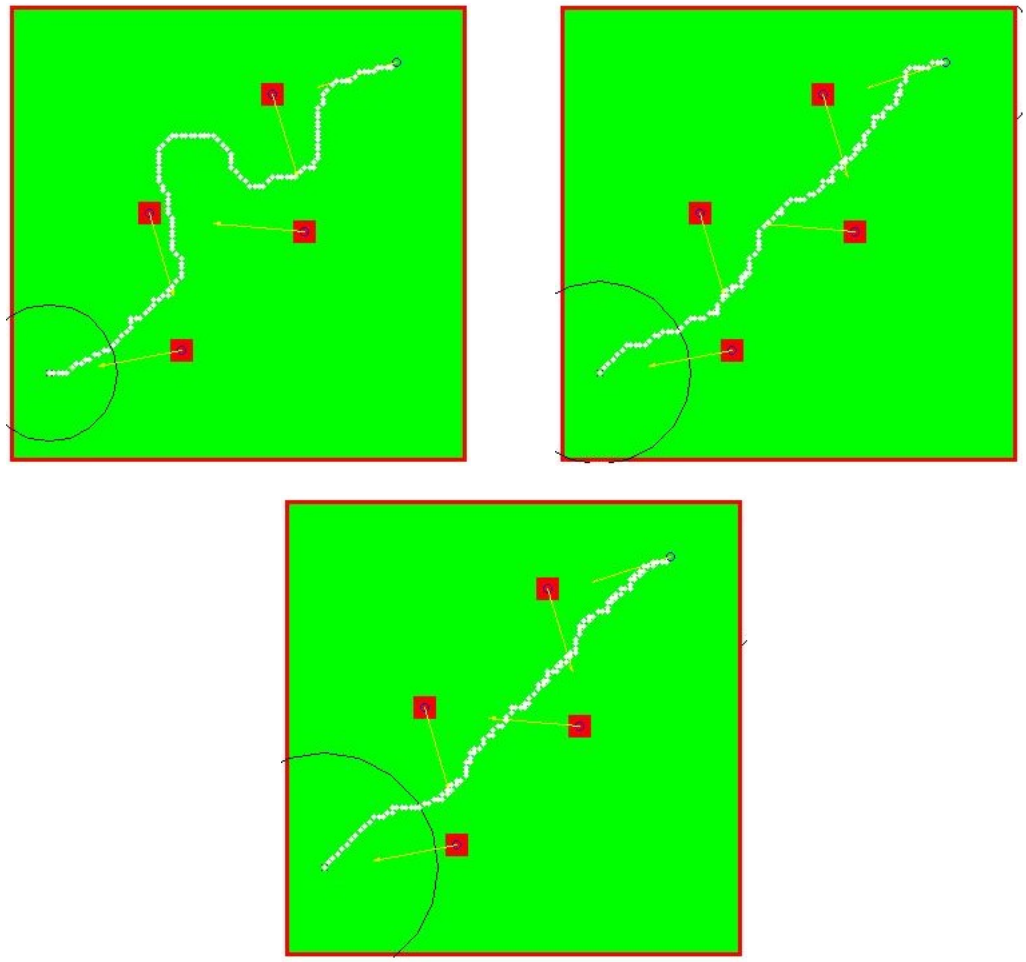
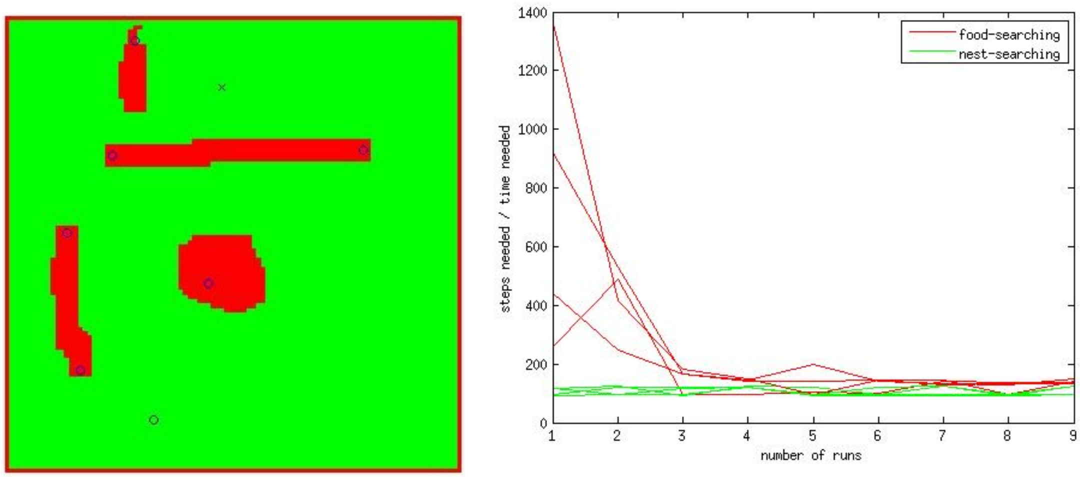


Figure 8: Comparison of different sizes of the detection radius

can differ in other maps with more or less landmarks, but some aspects are always the same:

- Global vector navigation is in most cases more efficient than local vector navigation.
- In the first few runs the drop of needed steps is the biggest.
- On some map arrangements it is not possible for our model to improve. This happens if there are to few local vectors or the detection-radius is to small.



(a) Original Map

(b) Step improvement

Figure 9: Path improvement with local vectors

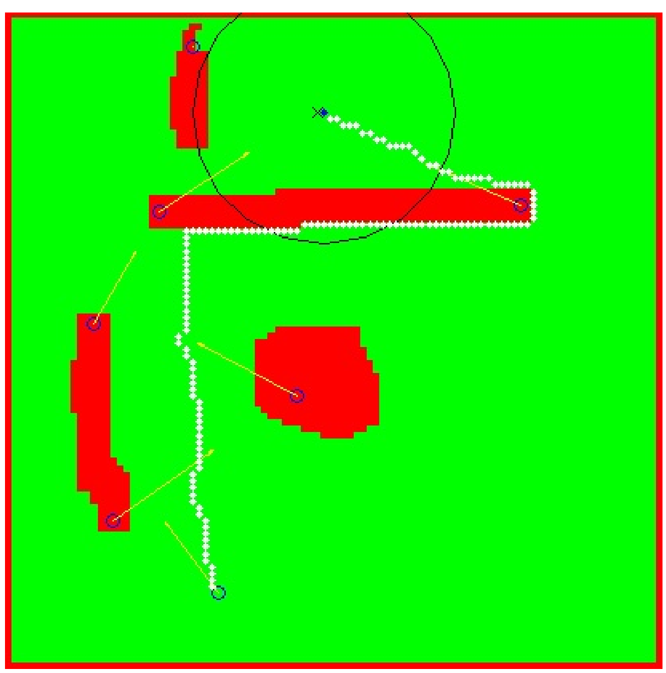


Figure 10: Map with local vectors and path

6 Summary and Outlook

6.1 Results

In this paper we showed that our model was able to replicate the path pattern of ants, when searching randomly for food, when returning to the nest by global vector and when using local vectors to search for food through a simple agent-based model. Most of the experiments described by Wehner[5, 2] were recreated and our modeled ants behaved indeed similar to the given results.

6.2 Further Improvments

One of the most obvious improvements to our model must be the inclusion of local vectors when returning home. Desert ants navigate with a certain chance by landmarks instead of the global vector on their way back to the nest as is showed in Wehner[2]. This is a complication, but can clearly improve the steps needed when returning home. Especially with a defective or even wrong global vector this can be very helpful when searching for the nest. This situation was not part of our model experiments and is therefore listed in our simplifications.

Also, we noticed that our model sometimes got stuck on random generated maps, when caves, small holes and certain patterns occurred. Our artificial made experimental maps where free of those trouble generating conditions, so that our model still produces valid results, but the improvement of the move method in the class ant definitely should be considered when further improvements are made.

Another question of interest is, how our model would behave if we gave it a higher degree of foresight in terms of moving. For now we have limited the move radius to the first order Moore Neighbours. One could also think of a ant, that can watch several steps ahead in order to find out which positions are desirable, creating a better path and realising obstacles earlier.

While running, our model uses the local vectors only as long as they stay in the detection radius of the ant, therefore forgetting about them as soon as they are out of sight. The same is true for the detection of the food. This leads in some situations to the ant loosing track, even after the food was in the detection radius. A more realistic approach would be to make the model slowly forget about the landmark or the position of the feeder.

A Research Plan

Group Name: The Anteaters Are Back

Group participants names: Wolf Vollprecht, Georg Wiedebach

A.1 General Introduction

We think ants are exciting animals because despite their small body mass and therefore small brain they form very huge and complex social structures. Very large numbers of them work together efficently like one body. This requires a high level of coordination. We have already seen some videos which show the great achievngs of ant colonies in building and hunting. Now we found out about their naviagion abilities and are curious to learn how ants are able to cover extreme (in comparison to their own body size) distances. The human being would definitely get lost when trying to jurney this far in the desert wthout GPS or any other form of modern help, so one of our main goals will be to find out how ants can master this difficult task.

Ants have been subject of modern research since 1848, the motivations were often interest in their instincts, society and of couse the hope to learn from them. Studies in ant movement became even more compelling when scientists started to look for algorithms that solve such fundamental tasks like finding the shortest way in a graph (Graph Theorie[7]). The class of ant colony optimization algorithms was introduced 1992 and has since been a field of active study[6].

A.2 Fundamental Questions

- How does ant movement and navigation work in challenging environments such as the desert?
  - Is ant communication connected to ant navigation?
  - Which mechanisms and factors influence ant movement?
  - Are there different strategies to find the shortest/safest etc. path?
- How can we describe ant paths in mathematical terms?
  - How efficient are our mathematical models of the real ant behaviour?
- How does our finding apply to the real world?

We would like to create a model of desert ant behaviour. This will include their search for food, their returning to the nest and their orientation with global and local vectors. Also we will see how close our algorithms are to real ant movement. Therefore we want to simulate the experiments described in the papers. Our model should be able to deal with different numbers of landmarks, obstacles and starting points. We would like to give our ants the ability to learn and improve their efficency when searching and finding food. Of course there will be some simplifications we eventually will have to deal with: Such as ... will be updated during work on the simulation.

A.3 Expected Results

We expect our simulation to be able to find short paths from Point A to desired positon B (i.e. feeder, nest) and back. We will try to be as cose to nature as possible and hope to be able to recreate some of the experimantal results given. Some results we consider particulary interesting are avoiding obstacles on returning home or finding a way to the nest after being deffered to a place where our ant has a non-fitting global vector. Of course we hope that there are already some good mathematical models available on ant movement, because these animals are topic of research for a long time already.

The evolution may have taught ants a lot of useful tricks and methods to survive in environmets like the desert. Probably there are more ways to orientate than only by landmarks. Ants may have similar ways to find out their geographic orientation like pigeons, or use the sun as a fix-point. We are curious to find out more about that.

B MATLAB™ – Code

B.1 main.m

```
%% Mainfile
% for common configurations of the simulation (mostly testing
% purposes and initiating)

clc;
clear all;
clf;
close all;
addpath('Maps');


%% Variables

runduration = 5;     % Duration of simulation
render = true;
path_render = true;



%% Options for different map-loading methods
% only one option should be enabled

% 1. Map from m-file
% ----------------------------------------------------------------
%map1


% 2. Random map from generator
% Some values need to be set by the user:
% ----------------------------------------------------------------
% mapsize = 100;
% s = simulation(mapsize, render, path_render);
% s.l.generateLandscape(mapsize, 30, 55, 0.8);
% s.l.nest = [5 5];
% s.a.position = s.l.nest;
% s.l.feeder = [95 95];


% 3. Map from image.png
% ----------------------------------------------------------------
s = simulation(100, render, path_render);
s.l.load_image('map2', 'png')
s.a.position = s.l.feeder;
s.l.landmarks = [s.l.landmarks; s.l.nest];
```

```



%% Run the simulation

s.a.createGlobalVector(s.l);
s.a.createLocalVectors(s.l.landmarks);

for i = 1:runduration
    s.run();
    i
end

%aviobj = close(s.aviobj);
% enable to create a movie (3/3)


%% Plotting the results on steps

figure(2)
plot(s.a.results_food_finding,'r')
hold on
plot(s.a.results_nest_finding,'g')
legend('food-searching','nest-searching')
xlabel('number of runs')
ylabel('steps needed / time needed')
```

B.2 simulation.m

```
%% Simulation Class
% Handles everything simulationwise e.g. run the simulation, define ...
    simulation wide parameters


classdef simulation < handle
    properties (SetAccess = private)
        l                % landscape
        a                % ant

        r                % true or false
        r_path           % true or false
        r_init           % true or false
        r_ant            % rendering
        r_ant_view       % rendering

%        aviobj = avifile('antmovie.avi','compression','None');
% enable to create a movie (1/3)
```

```
    end

    methods (Access = public)
        %% Initialization
        % Initalizes a simulation with landscape size N
        function S = simulation(N,r,r_path)
            S.l = landscape(N);
            S.a = ant();
            S.r = r;
            S.r_path = r_path;
            S.r_init = true;
        end

        %% Initiates the rendering
        function init_render(S)
            S.r_init = false;

            figure(1)
            imagesc(S.l.plant)
            axis off, axis equal
            colormap ([0 1 0; 1 0 0; 1 0 0])
            hold on
            plot(S.l.nest(1), S.l.nest(2),'o','Color','k')
            plot(S.l.feeder(1), S.l.feeder(2), 'x', 'Color', 'k');

            % If landmarks exits they are plotted
            if ¬isempty(S.l.landmarks)
                plot(S.l.landmarks(:,1), S.l.landmarks(:,2), 'o', ...
                    'Color', 'b');
            end

            % Initiates the Animation in "render"
            S.r_ant = plot(S.a.position(1), ...
                S.a.position(2),'.','Color','b');
            S.r_ant_view = plot(S.a.position(1) + ...
                S.a.detection_radius*cos(2*pi/20*(0:20)), ...
                S.a.position(2) + ...
                    S.a.detection_radius*sin(2*pi/20*(0:20)), 'Color', 'k');
        end

        %% Reset after complete run
        function reset(S)
            S.a.has_food = 0;
            S.a.nest = 0;
            S.a.obstacle_vector = zeros(100, 100, 2);

            % If render is true local vectors are plotted
            if S.r
                S.render_local_vectors;
            end
```

```
        end

        %% The simulation
        % if render is true the ant will be plottet on the landscape
        function run(S)
            % On the first run and if render is true rendering is initiated
            if S.r_init && S.r
                S.init_render();
            end

            % Some variables are reset bevore a new run
            S.reset();

            % Ant searces for food until a.has_food is true
            while S.a.has_food == 0
                S.a.findFood(S.l);

                % If render is true
                if S.r
                    S.render()
                end
            end

            % Ant returns to nest similar until a.nest is true
            while S.a.nest == 0
                S.a.returnToNest(S.l)

                % If render is true
                if S.r
                    S.render()
                end
            end

        end

        %% Render the simulation
        function render(S)
            figure(1)

            % Animation of ant and view-radius
            set(S.r_ant,'XData',S.a.position(1));
            set(S.r_ant,'YData',S.a.position(2));
            set(S.r_ant_view, 'XData', S.a.position(1) + ...
                S.a.detection_radius*cos(2*pi/20*(0:20)));
            set(S.r_ant_view, 'YData', S.a.position(2) + ...
                S.a.detection_radius*sin(2*pi/20*(0:20)));
            drawnow

            % If path plotting is true
            if S.r_path
```

```
                plot(S.a.position(1), S.a.position(2),'.','Color','w')
            end

%            F = getframe(1);
%            S.aviobj = addframe(S.aviobj,F);
% enable to create a movie (2/3)
        end

        %% Render local vectors
        function render_local_vectors(S)
            S.init_render();
            for i=1:length(S.l.landmarks)
                quiver(S.l.landmarks(i,1), S.l.landmarks(i,2), ...
                    S.a.local_vectors(i,1), S.a.local_vectors(i,2),'y')
            end
        end

    end % methods
end % classdef
```

B.3 landscape.m

```
%% Landscape class
% A class for handling the landscape of a simulation


classdef landscape < handle
    properties (SetAccess = public)
        size            % Sitze of quadratic Landcape

        plant           % Matrix storing free and taken points
        landmarks       % Position of landmarks
        feeder          % Position of Feeder
        nest            % Position of Nest
    end

    methods (Access = public)
        %% Initialize Landscape
        function L = landscape(N)
            L.size = N;
        end

        %% Gererate random Landcape
        function generateLandscape(L, n, num, size, prob)
            L.plant = zeros(n,n);
            L.plant(1,:) = ones(1,n);
            L.plant(n,:) = ones(1,n);
```

```
        L.plant(:,1) = ones(1,n);
        L.plant(:,n) = ones(1,n);

        % 1. Zuf llige Hindernisse Plazieren Anzahl der Hindernisse ...
            soll fest sein:
        posspeicher = zeros(num,1);

        for i = 1:num
            pos = n+1;
            % Finden eines geeigneten Ortes:
            while L.plant(pos) || L.plant(pos−1) || L.plant(pos+1) ...
                || L.plant(pos−n) || L.plant(pos+n)
                pos = randi([n+1,n*n−(n+1)]);
            end

            % Plazieren und speichern des Ortes  f r  Schritt 2:
            posspeicher(i) = pos;
            L.plant(pos) = 1;
        end

        % 2. Vergr ssern dieser Hindernisseauf eine bestimmte ...
            Gr sse (Hindernisse
        % wachsen  ber  R nder hinaus und auf der Anderen ...
            Spielfeldseite wieder
        % hinein.
        neigh = [−1 1 −n n];

        for i = 1:num
            dir = inf;
            for j = 1:size
                % Manchmal wird eine Richtungs nderung zugelassen:
                if rand < prob
                    dir = inf;
                end
                % W hlen einer zuf lligen Richtung zum Vergr ssern:
                while posspeicher(i) + dir < 1 || posspeicher(i) + ...
                    dir > n*n
                    dir = neigh(randi(4));
                end
                L.plant(posspeicher(i) + dir) = 1;
                posspeicher(i) = posspeicher(i) + dir;
            end
        end
    end

    %% Load a map (invoked from m−files)
    function load_map(L, P)
        L.plant = P;
        L.size = length(P);
    end % load_map
```

```

        %% Load a image-map
        function load_image(L, image, type)
            img = imread(image, type);
            L.size = length(img(:,:,1));
            L.plant = ¬img(:,:,1);                        % use hex #ffffff
            [y, x] = find(img(:,:,2) == 153);
            L.landmarks = [x, y];
            [y, x] = find(img(:,:,2) == 238, 1, 'first'); % use hex #1100ee
            L.nest = [x, y];
            [y, x] = find(img(:,:,3) == 238, 1, 'first'); % use hex #11ee00
            L.feeder = [x, y];
            L.plant(1,:) = ones(1,L.size);
            L.plant(L.size,:) = ones(1,L.size);
            L.plant(:,1) = ones(1,L.size);
            L.plant(:,L.size) = ones(1,L.size);
        end

    end % methods
end % classdef
```

B.4 ant.m

```
%% Ant class
% This class defines the behaviour/movement of an ant in a given landscape


classdef ant < handle
    properties (SetAccess = public)
        % Variables that may be set for testing:
        detection_radius = 20;          % View radius of the ant
        error_prob = 0.3;               % Error probability
        turn_prob = 0.3;                % Random turns

        % general variables
        position                        % Position of Ant
        global_vector                   % Global vector
        has_food                        % true or false
        nest                            % true or false

        % move-related Veriables
        move_direction                  % last move direction
        obstacle_vector                 % Matrix stores found obstacles
        rotation                        % Defines clockwise or ...
            counterclockwise turns
        move_radius                     % Moore neighbourhood (1st) of ...
            the ant
```

```
        local_vectors                   % stores local vectors
        updated_local_vectors           % boolean array
        last_local_vector               % stores the last landmark seen

        % result-storing
        step_counter                    % for counting the steps to nest ...
            or feeder
        results_food_finding            % results in steps
        results_nest_finding            % results in steps
    end

    methods (Access = private)
        %% Function to update local vectors (only when returning)
        function update_lv(A, landmarks)
            for i = 1:length(landmarks)
                if norm(landmarks(i,:) - A.position) < ...
                    A.detection_radius && ...
                        ¬A.updated_local_vectors(i) && ...
                        ¬isequal(A.last_local_vector, landmarks(end,:))

                    % "growth-factor" is calculated
                    gfac = 0.5 * exp(-norm(A.local_vectors(i,:))/10);


                    % Local vector is adjusted
                    A.local_vectors(i,:) = round(A.local_vectors(i,:) + ...
                        gfac * (- landmarks(i,:) + A.last_local_vector));

                    % Storing information about update
                    A.last_local_vector = landmarks(i,:);
                    A.updated_local_vectors(i) = true;
                end
            end
        end

        %% Function to calculate a second direction from given local vectors
        function temp = calc_lv_direction(A, landmarks)
            temp = [0 0];
            for i=1:length(landmarks)
                if norm(landmarks(i,:) - A.position) < ...
                    A.detection_radius && ...
                        ¬isequal(A.local_vectors(i,:), [0 0]) && ...
                        A.updated_local_vectors(i) == 0

                    % all local vectors in the detection radius are ...
                        summed up
                    temp = temp + A.local_vectors(i,:) + landmarks(i,:) ...
                        - A.position;

                    if norm(temp) ≤ 1
```

```
                        A.updated_local_vectors(i) = true;
                        disp('stop')
                    end

                end
            end
        end
    end % private methods

    methods (Access = public)
        %% Initalization of ant
        function A = ant()
            A.rotation = -1;
            A.move_direction = [0 1];
            A.obstacle_vector = zeros(100,100,2);
            A.move_radius = [1 1; 1 0; 0 1; 1 -1; -1 1; -1 0; 0 -1; -1 -1];
            A.step_counter = 0;
            A.nest = 0;
            A.has_food = 0;
        end

        %% Create the GlobalVector from Landscape
        function createGlobalVector(A, L)
            A.global_vector =  L.nest - A.position;
        end

        %% Initiate the local vectors
        function createLocalVectors(A, landmarks)
            A.local_vectors = zeros(length(landmarks), 2);
            A.updated_local_vectors = zeros(length(landmarks), 1);
        end

        %% FindFood
        % Moves ant randomly in landscape to find the feeder
        % calculates movevector from localvectors
        function findFood(A, L)

            % if the feeder is found:
            if isequal(A.position, L.feeder)
                A.has_food = true;
                A.last_local_vector = L.feeder;

                % results are stored and the stepcounter is reset
                A.results_food_finding = [A.results_food_finding, ...
                    A.step_counter];
                A.step_counter = 0;

                % some variables are reset or adjusted
                A.update_lv(L.landmarks)
                A.move_direction = -A.move_direction;
```

```
                A.updated_local_vectors(A.updated_local_vectors ≠ 0) = 0;
                return
            end

            % The Step-Counter is incremented
            A.step_counter = A.step_counter + 1;

            % All local vectors in detection radius are considered
            dir = A.calc_lv_direction(L.landmarks);

            % If there is no local_vector in sight the ant moves based on
            % its previous direction with a probability to trun 45
            % degree
            if isequal(dir, [0 0])
                dir = A.move_direction;
                if rand < A.turn_prob
                    phi = pi/4;
                    n = sign(rand-0.5);
                    err_rotation = [cos(phi), n*sin(phi); -n*sin(phi), ...
                        cos(phi)];
                    dir = round(dir * err_rotation);
                end
            end

            % If the ant can "see" the feeder all previous calcualations are
            % overwriten and the move direction points directly towards
            % the feeder.
            if norm(A.position - L.feeder) < A.detection_radius
                dir = L.feeder - A.position;
            end

            % move is invoked
            A.move(L, dir);
        end


        %% ReturnToNest
        % Ant returns to nest after it found food
        % The global vector is used
        function returnToNest(A, L)

             % if the nest is reached:
            if A.global_vector == 0
                A.nest = true;
                A.has_food = false;

                % results are stored and the stepcounter is reset
                A.results_nest_finding = [A.results_nest_finding, ...
                    A.step_counter];
                A.step_counter = 0;
```

```

                % some variables are reset or adjusted
                A.updated_local_vectors(A.updated_local_vectors ≠ 0) = 0;
                return
            end

            % The Step-Counter is incremented
            A.step_counter = A.step_counter + 1;

            % Local vectors are updated during the way home.
            A.update_lv(L.landmarks);

            % move is invoked
            A.global_vector
            A.move(L, A.global_vector);
        end

        %% move(A,L)
        % Moves ant in landmark, according to typical ant behaviour.
        % A: Ant
        % L: Landscape
        function move(A, L, move_vector)

            % All known obstacles are considered
            for i = 1:8
                move_vector(1) = move_vector(1)...
                    + A.obstacle_vector(A.position(1) + ...
                        A.move_radius(i,1), A.position(2) + ...
                        A.move_radius(i,2), 1);
                move_vector(2) = move_vector(2)...
                    + A.obstacle_vector(A.position(1) + ...
                        A.move_radius(i,1), A.position(2) + ...
                        A.move_radius(i,2), 2);
            end

            % if the given move_vector is zero a random move is chosen
            if isequal(move_vector, [0 0])
                move_vector = A.move_radius(randi([1,8]));
            end

            % The direction of the ant is given a certain random-error:
            if rand < A.error_prob
                move_vector(1) = move_vector(1) + (rand-0.5) * ...
                    move_vector(1);
                move_vector(2) = move_vector(2) + (rand-0.5) * ...
                    move_vector(2);
            end


            % Maindirection and seconddirection are calculated from the
```

```
        % direction given by the input vecor. The seconddirection ...
            gets a
        % Probability smaller than 0.5 based on the angle between
        % maindirection and global vector.
        maindir = round(...
            move_vector/max(abs(move_vector))...
        );
        secdir = sign(...
            move_vector - maindir * min(abs(move_vector))...
        );
        secprob = min(abs(move_vector)/max(abs(move_vector)));

        % the following tests make sure no error is produced because of
        % limit cases.
        if secdir(1) == 0 && secdir(2) == 0
            secdir = maindir;
        end
        if secprob == 0
            secdir = maindir;
        end
        if secprob <= 0.5
            tempdir = maindir;
            maindir = secdir;
            secdir = tempdir;
            secprob = 1-secprob;
        end

        temp = maindir;
        if rand < secprob
            temp = secdir;
        end

        % If there is no obstacle near the ant the rotation-direction
        % can change.
        count = 0;
        for i = 1:8
            count = count + L.plant(A.position(2) + ...
                A.move_radius(i,2), A.position(1) + A.move_radius(i,1));
        end
        if count == 0
            A.rotation = sign(rand-0.5);
        end

        phi = pi/4;
        rot = [cos(phi), A.rotation * sin(phi); -A.rotation * ...
            sin(phi), cos(phi)];

        % Obstacle-Avoiding: New maindirection until possible move ...
            is found!
```

```
            % 180deg-Turn-Avoiding: New maindirection if ant tries to ...
                turn around
            while L.plant(A.position(2) + temp(2), A.position(1) + ...
                temp(1)) ≠ 0 ...
                     || isequal(temp, -A.move_direction)

                % A obstacle_vector is created and helps the ant to ...
                    avoid the wall
                % and endless iterations.
                if abs(A.obstacle_vector(A.position(1) + temp(1), ...
                    A.position(2) + temp(2), 1)) < 40
                    A.obstacle_vector(A.position(1) + temp(1), ...
                        A.position(2) + temp(2), 1) = ...
                        A.obstacle_vector(A.position(1) + temp(1), ...
                            A.position(2) + temp(2), 1) ...
                        + 8*temp(1);
                end
                if abs(A.obstacle_vector(A.position(1) + temp(1), ...
                    A.position(2) + temp(2), 2)) < 40
                    A.obstacle_vector(A.position(1) + temp(1), ...
                        A.position(2) + temp(2), 2) = ...
                        A.obstacle_vector(A.position(1) + temp(1), ...
                            A.position(2) + temp(2), 2) ...
                        + 8*temp(2);
                end

                % The ant "turns" around 45deg.
                % rot is rotation matrix defined above
                temp = round(temp * rot);
            end

            % move direction is stored, position and global vector are
            % adjusted.
            A.move_direction = temp;
            A.position = A.position + temp;
            A.global_vector = A.global_vector - temp;
        end % move

    end % public methods
end
```

C References

References

[1] http://wissenschaft.de/wissenschaft/news/306514.html.

[2] M. Collett, T. S. Collett, S. Bisch, and R. Wehner. Local and global vectors in desert ant navigation. *Letters to nature*, 1998.

[3] K. Donnay and S. Balietti. www.soms.ethz.ch/teaching/matlabfall11 lecture 5 – agent-based modeling / game theory.

[4] M. Müller and R. Wehner. Path integration in desert ants, cataglyphis fortis. *Proc. Nati. Acad. Sci. USA*, 1988.

[5] R. Wehner. Desert ant navigation: how miniature brains solve complex tasks. *Karl von Frisch Lecture*, 2003.

[6] Wikipedia. http://en.wikipedia.org/wiki/ant_colony_optimization_algorithms.

[7] Wikipedia. http://en.wikipedia.org/wiki/graph_theory.

List of Figures