Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

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

Project Report

# Modelling Desert Ant Behaviour with a special focus on desert ant movement

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

## 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 different approaches. The timeframe of the course is one semester.
In this paper we will try to show how to replicate the behaviourr 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 very huge and complex social structures. Very large numbers of them work together efficiently like one body. This requires a high level of coordination. We have already seen some videos which show the great achievements of ant colonies in building and hunting. Now we found out about their navigation abilities and are curious to learn how ants are able to cover extreme distances. 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.

Ants have been subject of modern research since 1848, the motivations were often interest in their instincts, society and of course 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 Theory). The class of ant colony optimization algorithms was introduced 1992 and has since been a field of active study.

However, those algorithms are using 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 finding 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 laid out before them, like forest ants do. Also they make use of a path-integrator with which they are able to track their position in reference to where they started the journey, most likely the nest.

Results of interest are:

- How optimized is navigation by vectors
- What is the most energy-consuming task
- Out of which states is it possible for the ant to find the nest (e.g. dropping the ant somewhere else, outside of her regular path etc.)
- How well does the ant 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

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 efficiency when searching and finding food.

Because of the nature of our problem we choose to design our simulation around a time-discrete step-based model of an ant. We chose to let only one ant run 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).

The simulation should be capable of finding a good path between nest and feeder and use a simple learning process to achieve that. We want to create a model, that can autonomous avoid obstacles and not get stuck in a corner. In order to meet this requirements we split our simulation in two parts:

#### Landscape

Our landscape should contain all the information about

- Position of the nest
- Position of the feeder
- Obstacles (stones, trees, cacti, oases, sand dunes and many more), 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 matrix stores information about taken and free points by the values true or false, where false stand for an obstacle. Nest, feeder, landmarks and local vectors are saved separately as vectors, to make them easy to reach.

#### Ant

Our ant should follow certain, simple rules to move according to the studies we received as part of the project description. Such are basic rules like avoiding obstacles or a little more specific rules like following the global vector when returning to the nest and using the local vectors of the landmarks when finding the food again. During the simulation and after the ant has had success in finding food our local vectors should as well change according to the new found and better path.

### 3.1 Simplifications

There will be simplifications and assumptions, 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). This is due to the fact that this behaviour can differ from ant to ant and there is no consistent result true for all desert ants.
- The model will have a detection-radius in which landmarks, nest and feeder are considered for moving and navigating.

## 4 Implementation

As described above our simulation consists of two main parts: The landscape and the ant. Both of these were implemented as separate classes. A third class the simulation-class should handle the rendering, initialising and iterations. We also used a main-file in which we declared variables that would have impact on the outcome of our simulation like the detection-radius of the ant or information on the map, that should be loaded.

### 4.1 Landscape

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

We implemented different versions of loading landscapes into our simulation. Beside the possibility of creating the landscape-matrix in a separate m-file and the random-map generator we often used a simple but elegant method for generating maps out of arbitrary made generic Portable Network Graphics. This method finds specific color values and translates them into their meaning in the context of the landscape.

| | 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

### 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 should always point to the nest (as long as the ant moves are coherent). We built our ant around the most important method: `move`. The `move` function is called out of two different methods the `find_food` and the `return_to_nest`. In the following all methods of the class `ant` are described:

#### 4.2.1 Find food

This loop iterates the move-method until the ant reaches the food. 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 should follow to reach the food sooner. As soon as the feeder is in a certain distance (the detection radius) the ant runs straight towards it.

#### 4.2.2 Calculate the direction from landmarks

$$\vec{v}_{direction} = \sum_{i=0}^{n} \vec{l_i} \quad \forall ||\vec{l_i}||_2 < r_{detection} \tag{1}$$

where $\vec{l_i}$ are the local vectors, $i$ ranging from the first to the last landmark and $r_{detection}$ is the view 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 it updates all local vectors while passing the related landmarks. In our implementation the local vectors always points to the last landmark the ant has passed or are adjusted toward this position. Thereby the ant develops a steady route that is a close to the optimal route. Of course there is no possibility to find out how real ants remember the exact direction and length of the local vectors and therefore this way of implementation must be tested for reliability later on.

#### 4.2.4 Updating the local vectors on all landmarks

For the implementation we decided, that our model of the ant, would be able to remember only the last global vector where a landmark was spotted. This simplification seems to be adequate because of the limited brain complexity of real ants. 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 *grow-factor*, which only allows a small adjusting every time the ant passes the landmark. As a result the learning curve of the ant became interesting, as described below in the experimental results.

$$
\begin{aligned}
a &= 0.5 * \exp\left(-\frac{||\vec{l_i}||_2}{10}\right) && (2)\\
\vec{l_i} &= a * \operatorname{round}\left(\vec{l_i} + (\vec{g_i} - \vec{g}_{i-1})\right) && (3)
\end{aligned}
$$

#### 4.2.5 move

The move method is 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 must calculate a new direction vector to one of the first order Moore neighbours:

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

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

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 (s. picture below). 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 allowed to walk directly towards a target. Some limit cases are handled separate.

$$
\begin{aligned}
\vec{s} &= \vec{m} - \vec{dir} * \min\left(|dir_i|\right) && (5)\\
p &= \frac{\min|dir_i|}{\max|dir_i|} && (6)
\end{aligned}
$$

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 following picture was taken with a low turning-probability (10 percent).

In the picture 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, then the desired position is checked again until a possible step is found.

## 4.3 Simulation

The simulation class main purpose is to serve as a holder for the landscape and the ant. 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):

```
while the ant has no food
    search for 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. 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 and an the detection-radius. Local vectors are rendered in a different method after each successful returning to the nest.

## 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 will try 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 are 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[**?**]. 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 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.

## 6 Summary and Outlook

## A Research Plan

## B MATLAB Code

### B.1 main.m

```matlab
%% Mainfile
% for common configurations of the simulation (mostly testing
% purposes

% clear everything

clc;
clear all;
clf;
close all;

runduration = 100;  % Duration of simulation

addpath('Maps');

%% Option1 saved Map
% all saved Maps can be found in the code-folder/Maps

%% two Obstacles - Experiment 1
% map1


%% map2
% noch erstellen.

%% Option2 random Map
%mapsize = 100;
%s = simulation(mapsize);
%s.l.generateLandscape(50, 50, 0.8);
%s.a.position = [5 5];
%s.l.nest = [5 5];
%s.l.feeder_radius = 50;

s = simulation(100);

s.l.load_image('test', 'png')
s.a.position = s.l.nest;

s.a.createGlobalVector(s.l);
s.a.createLocalVectors(s.l.landmarks);
s.init();
s.run(0);
```

## B.2 simulation.m

```matlab
%% Simulation Class
```

```
% Handles everything simulationwise e.g. run the simulation, define ...
    simulation wide parameters
%% Variables
% * l
%   Landscape
%   defines the Landscape of the simulation
% * a TODO decide if should/could be an array or not (simulate more than ...
    one ant in a given simulation)
%   Ant
%   defines the ant of the simulation


classdef simulation < handle
    properties (SetAccess = private)
        l;
        a;
        r_ant
        r_ant_view
    end
    methods (Access = public)
        %% Initialization
        % Initalizes a simulation with landscape size N
        % Ant is at the moment placed in the center of the map
        function S = simulation(N)
            if(nargin == 0)
                S.l = landscape(1);
                S.a = ant(1);
            else
                S.l = landscape(N);
                S.a = ant(N);
            end
        end
        %% Run
        % Runs simulation for specified amount of iterations
        function init(S)
            S.init_render();
        end
        function reset(S)
            S.a.has_food = 0;
            S.a.nest = 0;
            S.a.obstacle_vector = zeros(100, 100, 2);
        end
        function run(S, render)
            S.reset();
            while S.a.has_food == 0
                S.a.findFood(S.l);
                if render
                    S.render()
                end
            end
            while S.a.nest == 0
                S.a.returnToNest(S.l)
                if render
                    S.render()
                end
            end % while ant is not at nest.
        end % run
        function init_render(S)
            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');

            plot(S.l.landmarks(:,1), S.l.landmarks(:,2), 'o', 'Color', 'b');

            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.view_radius*cos(2*pi/8*(0:8)), ...
                S.a.position(2) + S.a.view_radius*sin(2*pi/8*(0:8)), ...
                    'Color', 'k');
            hold on
        end
        %% Render
        % renders the simulation (plant & ant)
        function render(S)
            figure(1)


            %plot(S.a.position(1)-S.a.move_direction(1), ...
                S.a.position(2)-S.a.move_direction(2),...
            %     '.','Color','w')
            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.view_radius*cos(2*pi/20*(0:20)));
            set(S.r_ant_view, 'YData', S.a.position(2) + ...
                S.a.view_radius*sin(2*pi/20*(0:20)));

            drawnow
            % Global Vector plotten?
            % pause(0.01)
        end % render

        function render_local_vectors(S)
            S.init_render();
            for i=1:length(S.l.landmarks)
```

```
                line([S.l.landmarks(i,1)  S.l.landmarks(i,1) + ...
                    S.a.local_vectors(i,1)], [S.l.landmarks(i,2) ...
                    S.l.landmarks(i,2) + S.a.local_vectors(i,2)]);
            end
        end
    end
end
```

## B.3 landscape.m

```
%% Landscape class
% A class for handling the landscape of a simulation
%% Properties
% * size:
%   int, size of quadratic landscape
% * plant(size, size):
%   int-array map of landscape
% * feeder(1,1):
%   int-array position of

classdef landscape < handle
    properties (SetAccess = public)
        size;
        landmarks;
        plant;
        feeder;
        feeder_radius
        nest;
    end
    methods (Access = private)
    end
    methods (Access = public)
        %% Initialize Landscape
        % size = n
        function L = landscape(N)
            L.size = N;
            L.feeder = round([1/3*N 2/3*N]);
            L.nest = round([2/3*N 1/3*N]);
        end % init

        %% set Feeder Radius for better observability;
        function setFeederRadius(L, r)
            L.feeder_radius = r;
        end

        %% Stump for external generateLandscape function
        function generateLandscape(L, obstaclecount, obstaclesize, ...
            obstacleprobability)
            L.plant = generateLandscape(L.size, obstaclecount, ...
                obstaclesize, obstacleprobability);
        end

        %% Function to set nest and feeder positions (not always required)
        % Nest = nestposition, Feeder = feederposition
        function setNestAndFeeder(Nest, Feeder)
            L.nest = Nest;
            L.feeder = Feeder;
        end

        %% Set Landmarks
        function setLandmarks(Landmarks)
            L.landmarks = Landmarks;
        end

        % Load a map with a specified plant and feeder/nest positions
        function load_map(L, P)
            L.plant = P;     % Set plant
            L.size = length(P);
        end % load_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
    methods (Static)
    end % Static functions
end % classdef
```

## B.4 ant.m

```
%% Ant class
% This class defines the behaviour/movement of an ant in a given landscape
%% Variables
% * position
%   1x2 int matrix
%   Position of ant in landscape
% * move_radius
%   nx2 int matrix
%   Defines "move radius" (neighbor fields for ant)
%   e.g. [-1 -1; -1 0; 0 -1; 0 1; 1 0; 1 1] ...
% * landmarks (TODO not implemented yet)
%   nxn int matrix
%   Defines local landmark-vectors for ant, should have the
%   size of the landscape
% * velocity
%   Is a 1x2 vector defining the x-y-velocity of our ant

classdef ant < handle
    properties (SetAccess = public)
        position
        move_radius = [1 1; 1 0; 0 1; 1 -1; -1 1; -1 0; 0 -1; -1 -1];
        move_direction
        global_vector
        has_food
        nest
        obstacle_vector
        rotation
        view_radius = 20;
        local_vectors
        updated_local_vectors
        last_global_vector = [0 0]
    end
    methods (Access = private)
        % creates the move_radius matrix
        function create_moveradius(A, movewidth)
            k = 1;
            n = round(movewidth/2);
            for i=-n:n
                for j=-n:n
                    if i == 0 && j == 0
                        break
                    end
                    A.move_radius(k,1) = i;
                    A.move_radius(k,2) = j;
                    k = k + 1;
                end
            end
        end
        %% Function to update local vectors on seeable landmarks (only ...
            when returning)
        function update_lv(A, landmarks)
            for i = 1:length(landmarks)
                if norm(landmarks(i,:) - A.position) < A.view_radius && ¬...
                    A.updated_local_vectors(i)
                    A.local_vectors(i,:) = A.global_vector - ...
                        A.last_global_vector;
                    A.last_global_vector = A.global_vector;
                    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.view_radius
                    temp = temp + A.local_vectors(i,:);
                end
            end
            disp(temp);
        end
    end % private methods
    methods (Access = public)
        %% Initalization of ant
        % x,y: starting positions
        % movewidth: size for created generated move_radius matrix
        function A = ant(x,y,movewidth)
            if nargin == 1
                A.position(1) = round(x/2);
                A.position(2) = round(x/2);
            elseif nargin > 1
                A.position(1) = x;
                A.position(2) = y;
            end
            A.rotation = -1;
            A.move_direction = [0 1];
            A.nest = 0; % True or False
            A.has_food = 0;
            A.obstacle_vector = zeros(100,100,2);
        end

        %% createGlobalVector from Landscape
        function createGlobalVector(A, L)
            A.global_vector =  L.nest - A.position;
        end
        %% init local vectors
        % only for coding & plotting convenience
        % no ant predeterminately knows all landmarks on map
        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
        % Ant should learn landscapes and path integrate the global
        % vector
        % return true if found food
        % return false if not
        % calculate localvectors into move vector
        function findFood(A, L)
            if A.position(1) == L.feeder(1) && A.position(2) == L.feeder(2)
                A.has_food = 1;
                A.last_global_vector = A.global_vector;
                disp('found food');
                return
            end
            dir = A.calc_lv_direction(L.landmarks)
            if dir(1) == 0 && dir(2) == 0
                dir = A.move_radius(randi(length(A.move_radius)),:);
                while dir * A.move_direction' ≤ 0
                    dir = A.move_radius(randi(length(A.move_radius)),:);
                end
            end

            if norm(A.position - L.feeder) < A.view_radius
                dir = L.feeder - A.position;
            end

            A.move_direction = dir;
            A.move(L, dir);
            A.has_food = 0;
        end

        function init_returnToNest(A, landmarks)
           A.update_local_vectors = zeros(length(landmarks), 1);
        end

        %% returnToNest
        % Ant returns to nest after she found food
        % Tries to go the mist direct way with global_vector
        % which points straight to the nest

        function returnToNest(A, L)
             % if the ant reached the nest no move is needed.
            if A.global_vector == 0
                A.nest = 1;
                disp('reached nest')
                return
            end
            A.update_lv(L.landmarks);
            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)
            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
            while move_vector(1) == 0 && move_vector(2) == 0
                move_vector = A.move_radius(randi([1,8]));
            end


            % Maindirection and seconddirection are calculated from the
            % direction given by the global veor. 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 ...
                    || ( temp(1) == −A.move_direction(1) && temp(2) == ...
                        −A.move_direction(2) )

                % A obstacle_vector is created and helps the ant to ...
                    avoid the wall
                % and endless iterations.
                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) ...
                    + 10*temp(1);
                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) ...
                    + 10*temp(2);

                % The ant "turns" in direction of secdir. New secdir is old
                % maindirection rotated over old secdir. (mirror)
                % rot rotates

```

```
                temp = round(temp * rot);
            end

            A.move_direction = temp;
            A.position = A.position + temp;
            A.global_vector = A.global_vector - temp;

        end % move
    end % public methods
    methods (Static)

    end % static methods
end
```

## C References