



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 | 5 |
| 2 | Introduction and Motivations | 6 |
| 3 | Description of the Model | 7 |
| 3.1 | Simplifications | 8 |
| 4 | Implementation | 9 |
| 4.1 | Landscape | 9 |
| 4.2 | Ant | 10 |
| 5 | Simulation Results and Discussion | 10 |
| 6 | Summary and Outlook | 10 |
| A | Research Plan | 10 |
| B | MATLAB Code | 10 |
| B.1 | main.m | 10 |
| B.2 | simulation.m | 11 |
| B.3 | landscape.m | 13 |
| B.4 | ant.m | 15 |
| C | References | 20 |

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 don't think that a 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 autonomously 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

5 Simulation Results and Discussion

6 Summary and Outlook

A Research Plan

B MATLAB Code

B.1 main.m

```
1 %% Mainfile
2 % for common configurations of the simulation (mostly testing
3 % purposes
4
5 % clear everything
6
7 clc;
8 clear all;
9 clf;
10 close all;
11
12 runduration = 100; % Duration of simulation
13
14 addpath('Maps');
15
16 %% Option1 saved Map
17 % all saved Maps can be found in the code-folder/Maps
18
19 %% two Obstacles – Experiment 1
20 % map1
21
22
23 %% map2
24 % noch erstellen.
25
26 %% Option2 random Map
27 %mapsize = 100;
28 %s = simulation(mapsize);
29 %s.l.generateLandscape(50, 50, 0.8);
30 %s.a.position = [5 5];
31 %s.l.nest = [5 5];
32 %s.l.feeder_radius = 50;
33
34 s = simulation(100);
```

```

35
36 s.l.loadimage('test', 'png')
37 s.a.position = s.l.nest;
38
39 s.a.createGlobalVector(s.l);
40 s.a.createLocalVectors(s.l.landmarks);
41 s.init();
42 s.run(0);

```

B.2 simulation.m

```

1 %% Simulation Class
2 % Handles everything simulationwise e.g. run the simulation, define ...
   simulation wide parameters
3 %% Variables
4 % * l
5 %   Landscape
6 %   defines the Landscape of the simulation
7 % * a TODO decide if should/could be an array or not (simulate more than ...
   one ant in a given simulation)
8 %   Ant
9 %   defines the ant of the simulation
10
11
12 classdef simulation < handle
13     properties (SetAccess = private)
14         l;
15         a;
16         r_ant
17         r_ant_view
18     end
19     methods (Access = public)
20         %% Initialization
21         % Initalizes a simulation with landscape size N
22         % Ant is at the moment placed in the center of the map
23         function S = simulation(N)
24             if nargin == 0
25                 S.l = landscape(1);
26                 S.a = ant(1);
27             else
28                 S.l = landscape(N);
29                 S.a = ant(N);
30             end
31         end
32         %% Run
33         % Runs simulation for specified amount of iterations
34         function init(S)

```

```

35         S.init_render();
36     end
37     function reset(S)
38         S.a.has_food = 0;
39         S.a.nest = 0;
40         S.a.obstacle_vector = zeros(100, 100, 2);
41     end
42     function run(S, render)
43         S.reset();
44         while S.a.has_food == 0
45             S.a.findFood(S.l);
46             if render
47                 S.render()
48             end
49         end
50         while S.a.nest == 0
51             S.a.returnToNest(S.l)
52             if render
53                 S.render()
54             end
55         end % while ant is not at nest.
56     end % run
57     function init_render(S)
58         figure(1)
59         imagesc(S.l.plant)
60         axis off, axis equal
61         colormap ([0 1 0; 1 0 0; 1 0 0])
62         hold on
63         plot(S.l.nest(1), S.l.nest(2), 'o', 'Color', 'k')
64         plot(S.l.feeder(1), S.l.feeder(2), 'x', 'Color', 'k');
65
66         plot(S.l.landmarks(:,1), S.l.landmarks(:,2), 'o', 'Color', 'b');
67
68         S.r_ant = plot(S.a.position(1), ...
69             S.a.position(2), '.', 'Color', 'b');
70         S.r_ant_view = plot(S.a.position(1) + ...
71             S.a.view_radius*cos(2*pi/8*(0:8)), ...
72             S.a.position(2) + S.a.view_radius*sin(2*pi/8*(0:8)), ...
73             'Color', 'k');
74         hold on
75     end
76     %% Render
77     % renders the simulation (plant & ant)
78     function render(S)
79         figure(1)
80
81         %plot(S.a.position(1)-S.a.move_direction(1), ...
82             S.a.position(2)-S.a.move_direction(2),...
83             '.', 'Color', 'w')

```

```

81         set(S.r_ant, 'XData', S.a.position(1));
82         set(S.r_ant, 'YData', S.a.position(2));
83         set(S.r_ant_view, 'XData', S.a.position(1) + ...
            S.a.view_radius*cos(2*pi/20*(0:20)));
84         set(S.r_ant_view, 'YData', S.a.position(2) + ...
            S.a.view_radius*sin(2*pi/20*(0:20)));
85
86         drawnow
87         % Global Vector plotten?
88         % pause(0.01)
89     end % render
90
91     function render_local_vectors(S)
92         S.init_render();
93         for i=1:length(S.l.landmarks)
94             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)]);
95         end
96     end
97 end
98 end

```

B.3 landscape.m

```

1  %% Landscape class
2  % A class for handling the landscape of a simulation
3  %% Properties
4  % * size:
5  %   int, size of quadratic landscape
6  % * plant(size, size):
7  %   int-array map of landscape
8  % * feeder(1,1):
9  %   int-array position of
10
11 classdef landscape < handle
12     properties (SetAccess = public)
13         size;
14         landmarks;
15         plant;
16         feeder;
17         feeder_radius
18         nest;
19     end
20     methods (Access = private)
21     end
22     methods (Access = public)

```

```

23     %% Initialize Landscape
24     % size = n
25     function L = landscape(N)
26         L.size = N;
27         L.feeder = round([1/3*N 2/3*N]);
28         L.nest = round([2/3*N 1/3*N]);
29     end % init
30
31     %% set Feeder Radius for better observability;
32     function setFeederRadius(L, r)
33         L.feeder_radius = r;
34     end
35
36     %% Stump for external generateLandscape function
37     function generateLandscape(L, obstaclecount, obstaclesize, ...
38         obstacleprobability)
39         L.plant = generateLandscape(L.size, obstaclecount, ...
40             obstaclesize, obstacleprobability);
41     end
42
43     %% Function to set nest and feeder positions (not always required)
44     % Nest = nestposition, Feeder = feederposition
45     function setNestAndFeeder(Nest, Feeder)
46         L.nest = Nest;
47         L.feeder = Feeder;
48     end
49
50     %% Set Landmarks
51     function setLandmarks(Landmarks)
52         L.landmarks = Landmarks;
53     end
54
55     % Load a map with a specified plant and feeder/nest positions
56     function load_map(L, P)
57         L.plant = P; % Set plant
58         L.size = length(P);
59     end % load_map
60
61     function load_image(L, image, type)
62         img = imread(image, type);
63         L.size = length(img(:,:,1));
64         L.plant = ~img(:,:,1); % use hex #ffffff
65         [y, x] = find(img(:,:,2) == 153);
66         L.landmarks = [x, y];
67         [y, x] = find(img(:,:,2) == 238, 1, 'first'); % use hex #1100ee
68         L.nest = [x, y];
69         [y, x] = find(img(:,:,3) == 238, 1, 'first'); % use hex #11ee00
70         L.feeder = [x, y];
71         L.plant(1,:) = ones(1,L.size);
72         L.plant(L.size,:) = ones(1,L.size);

```

```

71         L.plant(:,1) = ones(1,L.size);
72         L.plant(:,L.size) = ones(1,L.size);
73     end
74
75     end % methods
76     methods (Static)
77     end % Static functions
78 end % classdef

```

B.4 ant.m

```

1  %% Ant class
2  % This class defines the behaviour/movement of an ant in a given landscape
3  %% Variables
4  % * position
5  %   1x2 int matrix
6  %   Position of ant in landscape
7  % * move_radius
8  %   nx2 int matrix
9  %   Defines "move radius" (neighbor fields for ant)
10 %   e.g. [-1 -1; -1 0; 0 -1; 0 1; 1 0; 1 1] ...
11 % * landmarks (TODO not implemented yet)
12 %   nxn int matrix
13 %   Defines local landmark-vectors for ant, should have the
14 %   size of the landscape
15 % * velocity
16 %   Is a 1x2 vector defining the x-y-velocity of our ant
17
18 classdef ant < handle
19     properties (SetAccess = public)
20         position
21         move_radius = [1 1; 1 0; 0 1; 1 -1; -1 1; -1 0; 0 -1; -1 -1];
22         move_direction
23         global_vector
24         has_food
25         nest
26         obstacle_vector
27         rotation
28         view_radius = 20;
29         local_vectors
30         updated_local_vectors
31         last_global_vector = [0 0]
32     end
33     methods (Access = private)
34         % creates the move_radius matrix
35         function create_moveradius(A, movewidth)
36             k = 1;

```

```

37         n = round(movewidth/2);
38         for i=-n:n
39             for j=-n:n
40                 if i == 0 && j == 0
41                     break
42                 end
43                 A.move_radius(k,1) = i;
44                 A.move_radius(k,2) = j;
45                 k = k + 1;
46             end
47         end
48     end
49     %% Function to update local vectors on seeable landmarks (only ...
50     when returning)
51     function update_lv(A, landmarks)
52         for i = 1:length(landmarks)
53             if norm(landmarks(i,:) - A.position) < A.view_radius && ~...
54                 A.updated_local_vectors(i)
55                 A.local_vectors(i,:) = A.global_vector - ...
56                     A.last_global_vector;
57                 A.last_global_vector = A.global_vector;
58                 A.updated_local_vectors(i) = true;
59             end
60         end
61     end
62     %% Function to calculate a second direction from given local vectors
63     function temp = calc_lv_direction(A, landmarks)
64         temp = [0 0];
65         for i=1:length(landmarks)
66             if norm(landmarks(i,:) - A.position) < A.view_radius
67                 temp = temp + A.local_vectors(i,:);
68             end
69         end
70         disp(temp);
71     end
72 end % private methods
73 methods (Access = public)
74     %% Initalization of ant
75     % x,y: starting positions
76     % movewidth: size for created generated move_radius matrix
77     function A = ant(x,y,movewidth)
78         if nargin == 1
79             A.position(1) = round(x/2);
80             A.position(2) = round(x/2);
81         elseif nargin > 1
82             A.position(1) = x;
83             A.position(2) = y;
84         end
85         A.rotation = -1;
86         A.move_direction = [0 1];

```



```

84         A.nest = 0; % True or False
85         A.has_food = 0;
86         A.obstacle_vector = zeros(100,100,2);
87     end
88
89     %% createGlobalVector from Landscape
90     function createGlobalVector(A, L)
91         A.global_vector = L.nest - A.position;
92     end
93     %% init local vectors
94     % only for coding & plotting convenience
95     % no ant predeterminately knows all landmarks on map
96     function createLocalVectors(A, landmarks)
97         A.local_vectors = zeros(length(landmarks), 2);
98         A.updated_local_vectors = zeros(length(landmarks), 1);
99     end
100     %% findFood
101     % Moves ant randomly in landscape to find the feeder
102     % Ant should learn landscapes and path integrate the global
103     % vector
104     % return true if found food
105     % return false if not
106     % calculate localvectors into move vector
107     function findFood(A, L)
108         if A.position(1) == L.feeder(1) && A.position(2) == L.feeder(2)
109             A.has_food = 1;
110             A.last_global_vector = A.global_vector;
111             disp('found food');
112             return
113         end
114         dir = A.calc_lv_direction(L.landmarks)
115         if dir(1) == 0 && dir(2) == 0
116             dir = A.move_radius(randi(length(A.move_radius)),:);
117             while dir * A.move_direction' ≤ 0
118                 dir = A.move_radius(randi(length(A.move_radius)),:);
119             end
120         end
121
122         if norm(A.position - L.feeder) < A.view_radius
123             dir = L.feeder - A.position;
124         end
125
126         A.move_direction = dir;
127         A.move(L, dir);
128         A.has_food = 0;
129     end
130
131     function init_returnToNest(A, landmarks)
132         A.update_local_vectors = zeros(length(landmarks), 1);
133     end

```

```

134
135 %% returnToNest
136 % Ant returns to nest after she found food
137 % Tries to go the most direct way with global_vector
138 % which points straight to the nest
139
140 function returnToNest(A, L)
141     % if the ant reached the nest no move is needed.
142     if A.global_vector == 0
143         A.nest = 1;
144         disp('reached nest')
145         return
146     end
147     A.update_lv(L.landmarks);
148     A.move(L, A.global_vector);
149
150 end
151
152 %% move(A,L)
153 % Moves ant in landmark, according to typical ant behaviour.
154 % A: Ant
155 % L: Landscape
156 function move(A, L, move_vector)
157     for i = 1:8
158         move_vector(1) = move_vector(1)...
159             + A.obstacle_vector(A.position(1) + ...
160                 A.move_radius(i,1), A.position(2) + ...
161                 A.move_radius(i,2), 1);
162         move_vector(2) = move_vector(2)...
163             + A.obstacle_vector(A.position(1) + ...
164                 A.move_radius(i,1), A.position(2) + ...
165                 A.move_radius(i,2), 2);
166     end
167     while move_vector(1) == 0 && move_vector(2) == 0
168         move_vector = A.move_radius(randi([1,8]));
169     end
170
171     % Maindirection and seconddirection are calculated from the
172     % direction given by the global vector. The seconddirection ...
173     % gets a
174     % Probability smaller than 0.5 based on the angle between
175     % maindirection and global vector.
176     maindir = round(...
177         move_vector/max(abs(move_vector))...
178     );
179     secdir = sign(...
180         move_vector - maindir * min(abs(move_vector))...
181     );
182     secprob = min(abs(move_vector)/max(abs(move_vector)));

```

```

179
180 % the following tests make sure no error is produced because of
181 % limit cases.
182 if secdir(1) == 0 && secdir(2) == 0
183     secdir = maindir;
184 end
185 if secprob == 0
186     secdir = maindir;
187 end
188 if secprob ≤ 0.5
189     tempdir = maindir;
190     maindir = secdir;
191     secdir = tempdir;
192     secprob = 1-secprob;
193 end
194
195
196 temp = maindir;
197 if rand < secprob
198     temp = secdir;
199 end
200
201 % If there is no obstacle near the ant the rotation-direction
202 % can change.
203 count = 0;
204 for i = 1:8
205     count = count + L.plant(A.position(2) + ...
206         A.move.radius(i,2), A.position(1) + A.move.radius(i,1));
207 end
208 if count == 0
209     A.rotation = sign(rand-0.5);
210 end
211
212 phi = pi/4;
213 rot = [cos(phi), A.rotation*sin(phi); -A.rotation*sin(phi), ...
214     cos(phi)];
215
216 % Obstacle-Avoiding: New maindirection until possible move ...
217 % is found!
218 % 180deg-Turn-Avoiding: New maindirection if ant tries to ...
219 % turn around
220 while L.plant(A.position(2) + temp(2), A.position(1) + ...
221     temp(1)) ≠ 0 ...
222     || ( temp(1) == -A.move.direction(1) && temp(2) == ...
223         -A.move.direction(2) )
224
225     % A obstacle_vector is created and helps the ant to ...
226     avoid the wall
227     % and endless iterations.

```

```

221         A.obstacle_vector(A.position(1) + temp(1), A.position(2) ...
           + temp(2), 1) = ...
222         A.obstacle_vector(A.position(1) + temp(1), ...
           A.position(2) + temp(2), 1) ...
223         + 10*temp(1);
224         A.obstacle_vector(A.position(1) + temp(1), A.position(2) ...
           + temp(2), 2) = ...
225         A.obstacle_vector(A.position(1) + temp(1), ...
           A.position(2) + temp(2), 2) ...
226         + 10*temp(2);
227
228         % The ant "turns" in direction of secdir. New secdir is old
229         % maindirection rotated over old secdir. (mirror)
230         % rot rotates
231
232         temp = round(temp * rot);
233     end
234
235     A.move_direction = temp;
236     A.position = A.position + temp;
237     A.global_vector = A.global_vector - temp;
238
239     end % move
240 end % public methods
241 methods (Static)
242
243 end % static methods
244 end

```

C References