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 **6**

4 Implementation **6**

5 Simulation Results and Discussion **6**

6 Summary and Outlook **6**

A Research Plan **6**

B MATLAB Code **6**

B.1 main.m 6

B.2 simulation.m 7

B.3 landscape.m 9

B.4 ant.m 11

C References **16**

1 Individual contributions

2 Introduction and Motivations

3 Description of the Model

4 Implementation

5 Simulation Results and Discussion

6 Summary and Outlook

A Research Plan

B MATLAB Code

B.1 main.m

```
%% 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

```
%% 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