ETH Zürich

Modelling and Simulating Social Systems with MATLAB

Vector updating in path integration of desert ants

Authors:
Demian Rihs
Davide Frey
David Bernasconi
Priska Pietra

Supervisors:
S. Balletti
K. Donnay

Zürich
December 16, 2011

Contents

Abstract

Several studies have shown that ants have the capability to integrate the random path they follow while looking for a feeding site in order to return home in a straight way. To do this they have developed multiple strategies to optimize the amount of information their small brains have to handle: landmark visualization, control with optic flow, energy and locomotor activity, compass orientation and some others similar methods. Basing on the literature and on the results of other studies, we discard orientation based on amount of energy used and similar subjective methods, and we focus on the light compass and visual landmark methods: we will show the ongoing changes that the ants do on their global vector during the outgoing path, and how this permits to build a straight incoming path, only influenced by encountered landmarks.

Individual contributions

The introduction and the bibliographical notes have been written by Priska Pietra; for the setting of the model and its implementation Davide Frey was responsible. Organizer for results and summary is David Bernasconi. The report and the presentation have been edited by Demian Rihs.

1 Introduction and motivations

We will now explain why some of these options have been rejected, and how the others are utilized in order to compute the shortest path for returning home. A more technical explanation is given in the section Description of the Model, in this introduction we will only give an overall view on the hypothesis, and set some questions that we would like to answer with this work.

1.1 How ants see the world

It may not be obvious, but ants haven't a map based representation of the world: their brains are too small to permit the storing of information not immediately useful [4]. This means that they aren't able to determine their position relative to the place where they are, but only they distance and direction relative to the starting point: this is the rout integration model. The model permits an en-route orientation that doesn't depend on any local cues: in respect to an on site orientation, that requires to be always aware of the position in the space, being independent of already known signals while basing only on its own angular and linear displacements permits to have a continuous representation of its position relative to the starting point, even in places never visited before [1]. But how can this kind of orientation be possible? The calculus required to integrate a path seems to be too complicated to be done by ants. Because of this, other hypothesis gained attention in the past: we will now look at three of them, specifically the energy consume, the locomotor activity and the time-lapse.

1.1.1 The energy consume

The first hypothesis said that ants were able to detect the distance walked comparing the amount of energy spent during the outgoing path and during

the incoming one. This idea has come from studies about bees, but for ants it has been proved that the walking distances are assessed accurately no matter how big the load they are carrying is: this means that energy count isn't utilized for the orientation, or at least that it isn't the first method that ants use [3].

1.1.2 The locomotor activity

The second easily confuted hypothesis is the one about the control of locomotor activity: in multiple experiments the length of ants' legs has been manipulated (shortened or lengthened) and not only this didn't affect the capability of movement of the insects, but also they always took directly in count the modification. In fact ants with shorter legs stopped after less steps than ants with normal limbs, and ants with longer legs stopped after more steps, even if the ones shortened undershot the distance and the ones lengthened overestimate it. This clearly indicates that it's not the number of steps that ants base their navigation system on [2].

1.1.3 The time-lapse

The last rejected hypothesis is based on a sort of time-lapse, which can be seen as a variant of the step counter that we have just explained: in that case the number of steps wasn't relevant for the distance at which ants began to search for the nest, and in this case studies have shown that also speed doesn't affect homing performance. [2]

After these consideration, we have to go back at the initial discarded option: direct path integration. It's assessed that ants aren't able to compute a perfect integration of the path they've followed, they use instead an approximation of this method, supported with the use of visual landmarks for the orientation over short distance. [5]
As already said, this kind of orientation only requires information about the angular displacement and about the distance covered: thanks to the self induced optic flow and to a compass based on the polarized skylight, the insect is able to constantly update a global vector that will guide it towards home. [3]
This is followed until a known landmark appear, temporarily overwriting the global vector with a local one that points toward the next landmark, until the goal is reached. [6] This kind of approximation is clearly subject of small errors that can outcome in catastrophically wrong paths, but ants handle this

problem by compensating every error in a direction with a similar amount of errors in the other direction, resulting in an optimally calibrated navigation system [1].

1.1.4 What we want demonstrate

It's empirically understandable how ants follow the vectors for returning home, and the model will be explained more technically in the next section, but it's not so clear how these vectors are created and followed during the home-returning path. This is our goal: showing how an ant returning home follows the vectors that it updates all along the outgoing route, and how it handles the different orientation method that are at its disposition.

2 Description of the model

In order to continually update the representation of their position relative to home, ants averages angular displacement, measured thanks to the skylight compass, and weight these displacements with the covered distance. At each step, the recording vector is overwritten by a new one that counts the new displacement and angle. The celestial compass is a simplified version of polarized light, coming from the direct sunlight and the scattered light in the sky: during the path ants have been observed to stop and turning about their vertical axis, in this way they align in the best possible way the skylight pattern and are able to detect every angle variation of their path.

In fact, when the insect moves in a straight line no variation of the polarized pattern is perceived and only in case of a turn there is a change that is immediately interpreted and integrated to update the global vector. To note is that celestial patterns are used to detect direction, and not location. [1]

Seen that the aim of our work is to show how the vectors change during the path, and not providing the exact calculation on every moment, we will work with a simplified model that uses polarized light only to find the direction at the start and that find distance with a simple trigonometric calculation. The observation that only in case of turns the vector is updated is also applicable in case of visual patterns: straight motion only causes objects to become visually bigger, in contrast to angular motion that completely changes the profile of the horizon. This is also the way in which the insects updates their local vector: when a known landmark comes in sight, the global vector goes in a sort of background computation and ants starts to follow a new vector. Pose that global vector always points home and is created during the outgoing path basing on angular displacement as said before, the local vectors are coupled with a landmark: they point the next known landmark, but don't encode the distance between the two. This is a motive because they are only utilized in a known zone near to the nest, and this proximity causes them to be followed instead of the global vector: the last part of the path is always integrated in background, but is build as a set of known landmarks pointing one another until the nest is rejoined. The use of landmarks motivates also the sector fidelity shown by the ants: the best a zone is know, the more local vectors can be utilized to go home [6].
We've just said that only direction and not distance is computed by ants. This is true about the vectors, but more generally also distance is taken into account. A traveling ant always maintains the head in the same position and at a constant high above the ground, and also the speed of the movements is nearly constant: the insects measure traveled distance through self-induced

optic flow, where the esteem is based on image speed, and not influenced by the type of terrain or others characteristics such as contrast. [3] We proceed now with the actual implementation: we will give each function starting from the Main, providing explanation of what they does line per line at the end of each section.

3 Implementation

3.1 Main

```
1  it = 10000;
2
3  number_landmarks = 10;
4
5  number_foods = 10;
6
7  size = 40;
8
9  visibility = 10;
10
11 nest = create_nest(size,visibility);
12
13 foods = create_foods(size,number_foods,nest,visibility);
14
15 landmarks = create_landmarks(nest,visibility,
       number_landmarks);
16
17 global_vector = [0,0];
18
19 alpha = pi/2;
20
21 dens = 0.5;
22
23 errFood = 0.5;
24 errNest = 0.2;
25
26 pos = [nest(1),nest(2)];
27
28 length_to_food = 0;
29 length_to_nest = 0;
30
31 figure
32
```

```
33 h = subplot(2,1,1);
34 plot(nest(1),nest(2),'dc',landmarks(:,1),landmarks(:,2),'
       ok','Markersize',8)
35 for i = 1:number_landmarks
36     hold on;
37     quiver(landmarks(i,1),landmarks(i,2),landmarks(i,3)-
           landmarks(i,1),landmarks(i,4)-landmarks(i,2));
38 end
39 axis(h,[-size,size,-size,size]);
40
41 t = 2;
42 while (t < it)
43     j = subplot(2,1,2);
44     [X,Y,alpha,l] = search_food(pos(t-1,:),alpha,dens,
           visibility,errFood,size,foods);
45     pos(t,:) = [X,Y];
46     length_to_food = length_to_food + l;
47     [vx,vy] = update_glob_vect(global_vector(1,:),pos(t,:)
           -pos(t-1,:));
48     global_vector(1,:) = [vx,vy];
49     plot(pos(t,1),pos(t,2),'or',pos(1:t,1),pos(1:t,2),'-',
           nest(1),nest(2),'dc',landmarks(:,1),landmarks(:,2)
           ,'ok',foods(:,1),foods(:,2),'xg','MarkerSize',8);
50     hold on;
51     quiver(pos(t,1),pos(t,2),-global_vector(1),-
           global_vector(2));
52     axis(j,[-size,size,-size,size]);
53     if (check_position(pos(t,:),foods) == 1)
54       break;
55     end
56     pause(0.00001);
57     cla
58     t = t + 1;
59 end
60
61 land = landmarks;
62
63 check = 0;
64 last_landmark = [size+1,size+1];
65
66 Q = length(pos);
67 t = Q + 1;
68
69 while (t < 2*it)
70     [X,Y,check,last_landmark,l] = search_nest(visibility,
           dens,-global_vector(1,:),nest(1,:),pos(t-1,:),
           errNest,check,landmarks,last_landmark(1,:));
```

```
71        length_to_nest = length_to_nest + l;
72        pos(t,:) = [X,Y];
73        [vx,vy] = update_glob_vect(global_vector(1,:),pos(t,:)
              -pos(t-1,:));
74        global_vector(1,:) = [vx,vy];
75        plot(pos(t,1),pos(t,2),'or',pos(1:t,1),pos(1:t,2),'-r'
              ,nest(1),nest(2),'dc',land(:,1),land(:,2),'ok',
              foods(:,1),foods(:,2),'xg',pos(1:Q,1),pos(1:Q,2),'
              -','MarkerSize',8)
76        hold on;
77        quiver(pos(t,1),pos(t,2),-global_vector(1),-
              global_vector(2));
78        axis([-size,size,-size,size]);
79        if (check_position(pos(t,:), nest))
80            break;
81        end
82        pause(0.01);
83        cla
84        t = t + 1;
85    end
86
87    length_to_food
88    length_to_nest
89    Rapp = length_to_food/length_to_nest
```

1-30 Here we define some initial variables. In order we have the number of iteration `it`, then the number of food sources `number_food` and the number of landmarks `number_landmarks` that we want to put in our board (we imagine that our ant is closed in a sort of box), the size of our board `size`, the visibility of the ant (how much can see the ant in a circle where it is the center), the nest coordinate `nest` (generated in **create_nest**), the foods coordinates `foods` (generated in **create_foods**), the landmarks coordinates `landmarks` (generated in **create_landmarks**), the initial global vector `global_vector`, the initial angle `alpha` (the starting direction of the ant's movement), the density of the ground `dens` (if the density is close to zero, this means that the ant moves on a rocky ground and if the density is two means that it moves on a smooth surface: we use this to simulate the optical flow used to calculate the speed), the errors that occurs when the ants follows some vectors: `errFood` when the ant search the food and `errNest` when it follows the global vector, the matrix `pos` that tells us for each iteration where is the ant and what is the way that it has followed until now then finally we have the starting length `length_to_food` when

the ant search the food and length `length_to_nest` when it search the nest.

32-40 We create a figure defining the first subplot: this shows the local vectors of each landmark, pointing to each other and to the nest. We use a subplot because plotting each iteration will cause the program being really slowed down.

42-61 Here it's defined the loop for the research of the food. We create a new subplot, use `search_food` to update the position matrix (adding a new line each time, with the new position of the ant). We also update the distance that the ant has already covered, update the global vector, and plot all the stuff (landmarks, foods, nest, global vector, movement and ant). Finally we check if we have found some food (using `check_position`).

63-91 We plot the loop for the path from the food source to the nest. It this loop we use the function `search_nest` and then we update all the variables as done before (distance already covered, the position, the global vector, plot all the stuff, and finally check if we are arrived to the nest).

93-95 We display the length of the outgoing random path and the length of the incoming straight path, and we show a comparison between this two lengths.

3.2 Search_food

```
1  function [x,y,degree_f,path_length] = search_foods(
       position, degree, dens, visibility, err, size, foods)
2
3  next_length = visibility/8*rand*dens;
4
5  path_length = next_length;
6
7  if (is_coordinate_visible(visibility,foods,position(1,:))
       == 1)
8      [X,Y] = nearest_food(position,foods);
9
10     if (is_reachable([X,Y], position(1,:), next_length) ==
           1)
11         x = X;
12         y = Y;
13         degree_f = degree;
14     else
15         new_vec = [X,Y] - position(1,:);
16         degree_f = calc_degree(new_vec);
17
18         x = next_length * cos(degree_f + err) + position(1
               );
19         y = next_length * sin(degree_f + err) + position(2
               );
20         if (y < -size || y > size || x < -size || x > size
               )
21             for i = 1:3
22                 x = next_length * cos(degree_f + (pi/2)*i
                       + err) + position(1);
23                 y = next_length * sin(degree_f + (pi/2)*i
                       + err) + position(2);
24                 if (x >= -size && x <= size && y >= -size
                       && y <= size)
25                     break;
26                 end
27             end
28         end
29     end
30 else
31     alpha = (-pi/4 + degree):0.1:(pi/4 + degree);
32
33     n = length(alpha);
34     degree_f = alpha(randi([1,n]));
35
36     x = next_length * cos(degree_f + err) + position(1);
```

```
37      y = next_length * sin(degree_f + err) + position(2);
38      if (y < -size || y > size || x < -size || x > size)
39          for i = 1:3
40            x = next_length * cos(degree_f + (pi/2)*i + err)
                  + position(1);
41            y = next_length * sin(degree_f + (pi/2)*i + err)
                  + position(2);
42              if (x >= -size && x <= size && y >= -size && y
                    <= size)
43                  break;
44              end
45          end
46      end
47  end
48
49  end
```

This function randomly calculate the next movement of the ant, generating its casual path from the nest to the food.

1 The inputs required are the actual position, the previous angle, the density of the ground, the visibility of the ant, the error, the size of the board and the matrix with the coordinates of food sources. The outputs are the new position, the new angle (the direction of the ant) and the length of the path that has already been covered.

3-5 We calculate the length of the path the ant will cover in this iteration. This is calculated by multiplying the rand (a number between zero and one), the visibility of the ant divided by 8 and the density of the ground. Then we give this length as output.

7 We check if the ant sees a food source using `is_coordinate_visible`.

8 If there is a visible food source, then we use `nearest_food` to find its coordinates.

10-13 We check if the nearest food source that we've found is reachable (using `is_reachable`). If so we set the new position of the ant to be equal to the coordinate of the food source and we set `degree_f` equal to `degree`.

14-19 If the food source is not reachable, we calculate the vector and the angle (using **calc_degree**). The ant has to follow. Finally we update the position using trigonometry.

20-28 We check if the new position doesn't exceed the board bounds. If yes, we change the degree that the ant has to follow adding π radiant multiplied k times. This procedure guarantees that the ant will be "reflected" like a laser beam on a mirror.

30-37 If no food is visible, then we calculate a random angle and, using this new randomly found value, we update the new position.

38-46 We check again if the new position does not exceed from the board.

3.3 Search_nest

```
1  function [x,y,check_f,last_landmark_f,path_length] =
       search_nest(visibility,dens,global_vector,nest,
       position,err,check,landmarks,last_landmark)
2
3  next_length = visibility/8*rand*dens;
4
5  if (check == 0)
6      check_f = 0;
7  else
8      check_f = 1;
9  end
10
11 last_landmark_f = last_landmark;
12
13 path_length = next_length;
14
15 if (is_coordinate_visible(visibility,nest,position) == 1)
16     if (is_reachable(nest,position,next_length) == 1)
17         x = nest(1);
18         y = nest(2);
19     else
20         alpha_glob = calc_degree(global_vector);
21
22         x = next_length * cos(alpha_glob + err) + position
               (1);
23         y = next_length * sin(alpha_glob + err) + position
               (2);
24     end
25
26 elseif (is_coordinate_visible(visibility,landmarks,
       position) == 1 && check == 0)
27
28     [X,Y,~,~] = nearest_landmark(position,landmarks);
```

```
29
30      if (is_reachable([X,Y],position,next_length) == 1)
31          x = X;
32          y = Y;
33
34          last_landmark_f = [x,y];
35
36          check_f = 1;
37      else
38          local_vect = [X - position(1),Y - position(2)];
39
40          next_degree = calc_degree(local_vect);
41
42          x = next_length * cos(next_degree + err) +
                position(1);
43          y = next_length * sin(next_degree + err) +
                position(2);
44      end
45  else
46      if (check == 1)
47          [X,Y] = landmark_pointer(landmarks,last_landmark);
48
49          if (is_reachable([X,Y],position,next_length) == 1)
50              x = X;
51              y = Y;
52
53              last_landmark_f = [x,y];
54          elseif (check_position(position,landmarks) == 1)
55              [Xp,Yp] = landmark_pointer(landmarks,position(
                    1,:));
56
57              local_vect = [Xp - position(1), Yp - position(
                    2)];
58
59              next_degree = calc_degree(local_vect);
60
61              x = next_length * cos(next_degree + err) +
                    position(1);
62              y = next_length * sin(next_degree + err) +
                    position(2);
63
64              last_landmark_f = position;
65          else
66              local_vect = [X - position(1), Y - position(2)
                    ];
67
68              next_degree = calc_degree(local_vect);
```

```
69
70              x = next_length * cos(next_degree + err) +
                    position(1);
71              y = next_length * sin(next_degree + err) +
                    position(2);
72          end
73      else
74          alpha_glob = calc_degree(global_vector);
75
76          x = next_length * cos(alpha_glob + err) + position
                (1);
77          y = next_length * sin(alpha_glob + err) + position
                (2);
78      end
79  end
80
81  end
```

This is the function that makes the ant move from the food to the nest.

1 The inputs required are the visibility of the ant, the density of the ground, the global vector (that the ant follow to come back to the nest), the nest coordinates, the actual position, the error, a boolean check (that tells if the ant has found a landmark), the matrix with all the landmarks and the last landmark that the ant passed through. The outputs are the new position, the new boolean check, the last landmark and the length of the path that the ant covered.

3-5 We set `check_f` to 0 if the input `check` is zero and to one if the ant has already encountered a landmark.

5-9 We set `check_f` to 0 if the input `check` is zero and to one if the ant has already encountered a landmark.

15 We check if the ant can see the nest from the actual position, using `is_coordinate_visible`.

16-18 If the nest is visible, then we check if it's reachable (if the ant is so close to the nest that it can reach it) using `is_reachable`. If yes we set the new position equal to the coordinates of the nest.

19-24 If the nest is visible, but the ant can't reach the nest in this iteration we calculate the angle that it has to follow for reach the nest, using

`calc_degree` and using the trigonometry and the length of the path to calculate the new position.

26 If the nest isn't visible, we check if a landmark is visible (using `is_coordinate_visible`) and if check is equal to zero.

28 We look for the nearest landmark using (using `nearest_landmark`).

30-36 We check if the landmark is reachable (using is_reachable), if yes we set the new position to be equal to the coordinates of this landmark, set `last_landmark_f` to be equal to this one, and the set `check_f` to 1.

37-43 If the landmark is not already reachable, we calculate the local vector between the nearest landmark and the actual position, then using `calc_degree` we calculate the new angle that the ant has to follow and finally using the trigonometry we find the new position.

46 If the ant sees nor the landmark, neither the nest we check if check is equal to 1 (if the ant is already passed through a landmark).

47 We calculate the landmark pointed by `last_landmark` using `landmark_pointer`.

49-53 We check if the pointed landmark is reachable, if yes we set the new position to this landmark and we update `last_landmark_f`.

54-64 If the position is equal to a landmark (we check it using `check_position`), we calculate the new landmark that the ant has to reach (using landmark_pointer). Then we calculate the local vector, the angle, and the new position, and we update `last_landmark_f`.

66-71 If the ant isn't on a landmark, and can't reach a landmark, we calculate the new position using the landmark pointed by `last_landmark` and we update all the variables.

73-77 If *check* 0 we follow the global vector, updating the position as before.

3.4 Create_nest

```
1  function nest = create_nest(size,visibility)
2
3  range_nest = [-size + 1 + 2*visibility, size - 1 - 2*
       visibility];
4  nest = [randi(range_nest),randi(range_nest)];
5
6  end
```

This is the function that randomly create a nest on our board.

1 The required inputs are the size of our board and the visibility of the ant. The output is a couple (x, y) representing coordinate.

3-4 We define a range where the coordinates can be (we make it start from $-size + 1 + 2 * visibility$ because we need the nest to be so far in the function **create_landmarks**). Then we generate two random coordinates in this range.

3.5 Create_foods

```
1  function foods = create_foods(size, number_foods, nest,
       visibility)
2
3  range_foods = [-size + 1, size - 1];
4
5  foods = zeros(number_foods,2);
6
7  j = 1;
8  while (j <= number_foods)
9      fd = [randi(range_foods),randi(range_foods)];
10     if (is_in_circle(fd,nest,2*visibility) == 0)
11         foods(j,:) = fd(1,:);
12         j = j + 1;
13     end
14 end
15
16 end
```

This is the function that randomly create some food sources in our board.

1 The required inputs are the size of the board, the number of food sources that we want to create, the position of the nest and the visibility of the ant. The output is a couple (x, y) representing coordinate.

3-5 We set a range for the food sources (with a minimum distance from the edges) and we initialize a new $number\_foods \times 2$ matrix.

8-14 The loop that gives us as many food sources as we need. In each iteration we generate a (x, y) couple and we check if it isn't in a range of *2*visibility* from the nest (the reason is that in `create_landmarks` we will create landmarks in this range, and we thought that it will be better if the food is out of this range), using the function `is_in_circle`.

3.6 Create_landmarks

```
1  function landmarks = create_landmarks(nest, visibility,
       number_landmark)
2
3  range_landmark = [nest(1) - 2*visibility, nest(1) + 2*
       visibility];
4
5  landmarks = zeros(number_landmark,2);
6
7  for i = 1:number_landmark
8      x = randi(range_landmark);
9
10     if (x == nest(1))
11         x = x + 1;
12     end
13
14     lim = sqrt(4*visibility^2 - (x - nest(1))^2);
15     y = randi([ceil(-lim)+nest(2),floor(lim)+nest(2)]);
16
17     landmarks(i,:) = [x,y];
18 end
19
20 radius_land=landmark_matrix(landmarks,2*visibility,nest);
21 [k,l] = size(radius_land);
22
23 final_landmarks = [landmarks,zeros(number_landmark,2)];
24
25 for i = k:-1:1
26     for j = 2:l
27         if (radius_land(i,j) == 1)
```

```
28                dt = 3*visibility;
29                sol = [1000,1000];
30                for m = 1:i-1
31                    for n = 1:l
32                        if (n == 1 && radius_land(m,n) == 1 &&
                              norm(nest - landmarks(j-1,:)) <=
                             dt)
33                            dt = norm(nest - landmarks(j-1,:))
                                  ;
34                            sol = nest;
35                        elseif (radius_land(m,n) == 1 && norm(
                             landmarks(n-1,:) - landmarks(j-1
                             ,:)) < dt)
36                            dt = norm(landmarks(n-1,:) -
                                  landmarks(j-1,:));
37                            sol = landmarks(n-1,:);
38                        end
39                    end
40                end
41                final_landmarks(j-1,3:4) = sol(1,:);
42            end
43        end
44    end
45
46    landmarks = final_landmarks;
47
48    end
```

This function create the matrix $N \times 4$ which contains all the landmarks and the landmarks that are pointed.

1 The inputs are the visibility of the ant (we say that the ant recognize landmarks until *2*visibility* all around the nest), tthe number of landmarks that we want to create and the position of the nest. The output is the *number_landmark* $\times$ 4 matrix.

3-5 We define the range where the landmark can be created (as we sayd before, they can have a maximal distance from the nest equal to *2*visibility*) and then we initialize the landmarks output matrix.

7-17 In this for loop we create as many landmarks as we need. In each iteration we generate a random x value in the range setted before, then we check that this is not equal to the x coordinate of the nest (this

ensure that we will not have a food source on the coordinates of the nest), we set a limit for the y coordinate and then we generate a random y coordinate.

20-21 Here we generate a new matrix filled only with zeros and ones (using **landmark_matrix**), that tell us in which annulus of the circle of radius *2*visibility* are the landmarks. Then we calculate the size of this matrix.

23 We initialize the new landmark matrix, adding two coloumns filled with zeros on the right side.

25-44 Here we calculate which landmark points which other one. We enter in a double "for" loop that read the matrix `radius_land` reated before from the bottom to the top. In fact we search all the entries that are equal to 1 and then we compare it with all the other entries equal to 1 that belong to all the row above this one (the reasons will be clearer in the explanation of **landmark_matrix**). The goal is finding the nearest landmark to the one that we are comparing (if the nest is the nearest, then the landmark points the nest).

3.7 Landmark_matrix

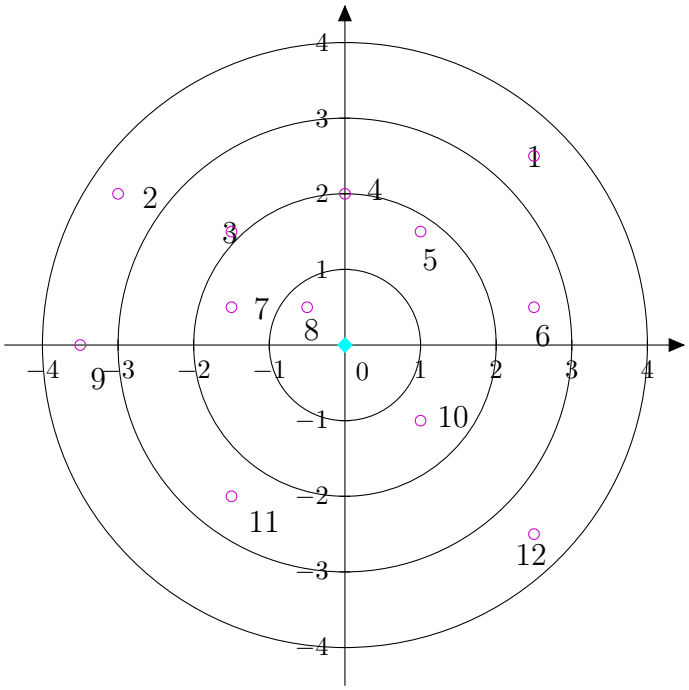
```
1  function pointer_matrix = landmark_matrix(landmarks,radius
       ,nest)
2
3  [m,~] = size(landmarks);
4
5  difference = 2;
6
7  pointer_matrix = zeros(radius, m + 1);
8  pointer_matrix(1,1) = 1;
9
10 i = 2;
11 while (i <= 2*radius + 2)
12     for j = 2:m+1
13         if (is_between_radius(landmarks(j-1,:), nest(1,:),
               i,difference) == 1)
14             pointer_matrix(i,j) = 1;
15         end
16     end
17     i = i + 2;
18 end
19
20 end
```

This function creates the matrix that tells in which annulus of a circle are situated the landmarks.

1 The inputs needed are the landmark matrix (when it's still an $n \times 2$ matrix), the maximal radius and the position of the nest. The output is a $M \times N$ matrix filled only with zeros and ones.

3-6 We calculate the number of landmarks that we have, we initialize the difference (the width of each annulus), then we initialize the output radius $radius \times (number\_landmark + 1)$.The matrix structure is as follow: each coloumn correspond to a landmark, except for the first one that corresponds to the nest, and each row corresponds to an annulus, starting for the one limitated by the circle of radius *difference* and the one of radius 0, and ending with the one limitated by the circle of radius radius and the one of radius *radius* *difference*.

10-18 We use **is_between_radius** to know if a point is in an annulus and we fill our matrix thanks to the double loop. In figure there is an example on how does it look.



In this imagine we see the nest (blue diamond in $(0,0)$), 12 numerated landmarks and 4 annulus of width 1. Let's put all the landmarks (in order, from the 1^{st} to the 12^{th}) in a 12×2 matrix:

$$landmarks = \begin{pmatrix} 2.5 & 2.5 \\ -3 & 2 \\ -1.5 & 1.5 \\ 0 & 2 \\ 1 & 1.5 \\ 2.5 & 0.5 \\ -1.5 & 0.5 \\ -0.5 & 0.5 \\ -3.5 & 0 \\ 1 & -1 \\ -1.5 & -2 \\ 2.5 & -2.5 \end{pmatrix}$$

Then we can start to fill our 10 matrix (that will be a 4×13 matrix) starting from the first row:

$$pointer\_matrix\_first\_row = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{pmatrix}$$

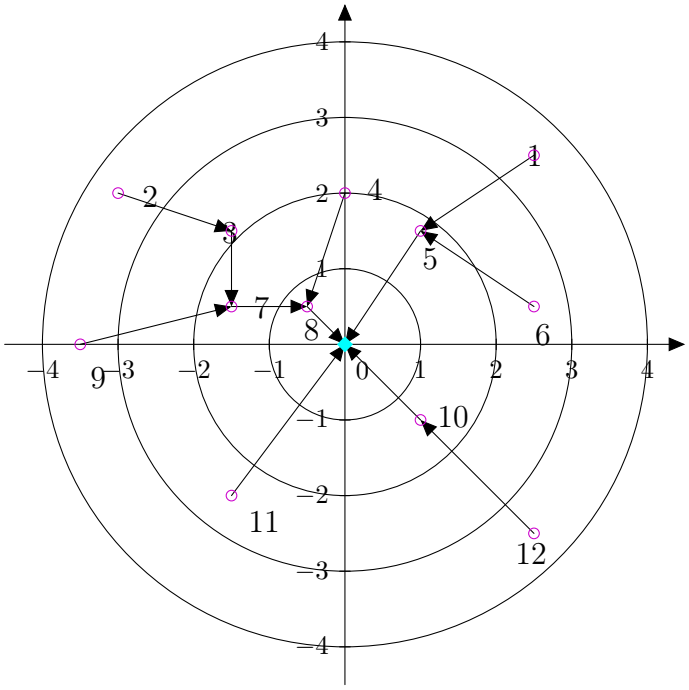
We put 1^s in the first and in the ninth place, because (as the image shows) there are the nest and the 8th landmark in the first annulus. Let's fill all the matrix:

$$pointer\_matrix = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \end{pmatrix}$$

The function do the following: it takes for example $pointer\_matrix(4, 2)$ (it's the first element of the last row) that correspond to the 1^{st} landmark and then compares it (calculate the distance) with all other landmarks that are not in the same annulus and are not in "out" annulus (nest too). When it finds the landmark or the nest that has the minimum distance, then the program updates the landmark matrix adding in the first row the two coordinates of the pointed landmark. In this case the resultant matrix will be:

$$landmarks = \begin{pmatrix} 2.5 & 2.5 & 1 & 1.5 \\ -3 & 2 & -1.5 & 1.5 \\ -1.5 & 1.5 & -1.5 & 0.5 \\ 0 & 2 & -0.5 & 0.5 \\ 1 & 1.5 & 0 & 0 \\ 2.5 & 0.5 & 1 & 1.5 \\ -1.5 & 0.5 & -0.5 & 0.5 \\ -0.5 & 0.5 & 0 & 0 \\ -3.5 & 0 & -1.5 & 0.5 \\ 1 & -1 & 0 & 0 \\ -1.5 & -2 & 0 & 0 \\ 2.5 & -2.5 & 1 & -1 \end{pmatrix}$$

You can see the graphial output in the next figure:



3.8 Calc_degree

```
1  function alpha = calc_degree(vector)
2
3  alpha = asin(vector(2)/norm(vector));
4
5  if (vector(1) <= 0 && vector(2) >= 0)
6    alpha = pi - alpha;
7  elseif (vector(2) <= 0 && vector(1) >= 0)
8    alpha = 2*pi + alpha;
9  elseif (vector(2) <= 0 && vector(1) <= 0)
10   alpha = pi - alpha;
11 end
12
13 end
```

This function calculate the angle of a vector bewteen itself and the vector $\begin{pmatrix}1\\0\end{pmatrix}$.

1 The only one input needed is the vector, and the output is the angle.

3 Here we calculate the angle using trigonometry: $\sin\alpha = \frac{a}{c}$, where $a = v_y$ and $b = \|v\|$.

5-11 We correct the angle we've just calculated, basing on how the vector is directed (for example, if the vector is nord-est directed - or rather, it has positive x coordinate and negative y - we have to subtract from π radiant the α calculated before).

3.9 Nearest_landmark

```
1  function [x,y,xp,yp] = nearest_landmark(position,landmarks
       )
2
3  dist = 10000;
4
5  [m,~] = size(landmarks);
6
7  for i=1:m
8      cond = norm(position(1,:) - landmarks(i,1:2));
9      if (cond < dist)
10         dist = cond;
11         x = landmarks(i,1);
12         y = landmarks(i,2);
13         xp = landmarks(i,3);
14         yp = landmarks(i,4);
15     end
16 end
17
18 end
```

This function give as output the 4-tuple of the nearest landmark from a position.

1 The inputs needed are the position and the landmarks matrix, the outputs are the coordinate of the nearest landmark, and the coordinate of the landmarks pointed by this one.

3-5 We set an initial "maximal distance" (we choosed a board of 40×40, so 10000 is enough, but if we choose a bigger board we should probably change this value) and then we calculate the number of landmarks.

7-16 This is the loop that calculate each time the distance between the ant's position and the i^{th} landmark, then it compares the distance between the shortest Landmark found until now, and if this one is shorter it updates the minimal distance and the outputs.

3.10 Is_reachable

```
1  function bool = is_reachable(coordinates, position,
       path_length)
2
3  bool = 0;
4
5  [m,~]  = size(coordinates);
6
7  dist = path_length;
8
9  for i = 1:m
10     cond = norm(position - coordinates(i,:));
11     if (cond <= dist)
12         bool = 1;
13         break;
14     end
15 end
16
17 end
```

This function tells if in some coordinates there is something that is reachable in one movement.

1 The inputs are the actual position, the coordinates that we want to compare with the position and the length of the path that the ant can cover. The output is a boolean value.

3-7 We set the initial value of the boolean value (set to 0), then we calculate how many coordinates we have to check and finally we set the initial"maximal distance" as *path_length*.

9-15 The loop checks if in the given coordinates there is one that can be reached by the ant. In fact this loop calculate for every i^{th} coordinate, the distance bewteen it and the ant's position and if this distance is less or equal than *path_length*, then break because we,ve found a point that is reachable by this position.

3.11 Is_coordinate_visible

```
1  function bool = is_coordinate_visible(visibility,
       coordinates,position)
2
3  [m,~] = size(coordinates);
4
5  bool = 0;
6
7  for i = 1:m
8      cond = (coordinates(i,1) - position(1))^2 + (
           coordinates(i,2) - position(2))^2;
9      if (cond <= visibility^2)
10         bool = 1;
11         break;
12     end
13 end
14
15 end
```

This function checks from some coordinates if there is something visible from the actual position.

1 The inputs are the position, the coordinates that we want to check and the visibility. The output is a boolean value.

3-5 We set the inital boolean value and we calculate how many coordinates we have to check.

7-13 We check if some coordinates are visible from the actual position, if yes break and set the boolean value to 1.

3.12 Is_between_radius

```
1  function bool = is_between_radius(coordinate,center,radius
       ,difference)
2
3  bool = 0;
4
5  cond = (coordinate(1) - center(1))^2 + (coordinate(2) -
       center(2))^2;
```

```
6
7  if (cond <= radius^2 && cond > (radius - difference)^2)
8      bool = 1;
9  end
10
11 end
```

This function checks if a point is between two circles of different radius (meaning that it checks if a point is in an annulus of a circle).

1 This function take as input the coordinates that we want to check if they are in an annulus, the center of the circle, the maximal radius of the annulus and the difference between the radius (in fact it's the width of the annulus choosen). The output is a boolean value.

3 Initialize boolean to 0.

5 Set the "condition value".

7-9 We check if the condition value is less or equal than $radius^2$ (we check if the point given by the coordinate is contained in the circle centered in *center* and with radius *radius*), then we check if this point is out of the circle centered in center but with radius *radius* *difference*. If so, then set bool to 1.

3.13 Is_in_circle

```
1  function bool = is_in_circle(coordinate,center,radius)
2
3  bool = 0;
4
5  cond = (coordinate(1) - center(1))^2 + (coordinate(2) -
       center(2))^2;
6
7  if (cond <= radius^2)
8      bool = 1;
9  end
10
11 end
```

This function is similar to the previous one (**is_between_radius**) but now we simply check if a point is contained in a circle.

1 The inputs are the coordinate that we have to check, the center of the circle and the its radius. The output is a boolean value.

3-5 Initialize boolean value to 0 and calculate the condition value.

7-9 Check if the condition value is less or equal than $radius^2$ (check if it's contained in the circle with radius *radius*).

3.14 Check_position

```
1  function bool = check_position(position, to_check)
2
3  bool = 0;
4
5  [m,~] = size(to_check);
6
7  for i = 1:m
8      if (position(1,:) == to_check(i,1:2))
9          bool = 1;
10         break;
11     end
12 end
13
14 end
```

This function checks a position, and tells if it corresponds to a coordinate given.

1 The inputs are the position where the ant is and the coordinates that we want to check. The output is a boolean value.

3-5 Set the boolean value to 0 and calculate the how many coordinates to check are given.

7-12 Start the loop for that check if in the matrix of coordinates given there is one corresponding to the position. If yes break and set the boolean value to 0.

3.15 Nearest_food

```
1  function [x,y] = nearest_food(position,foods)
2
3  dist = 10000;
4
5  [m,~] = size(foods);
6
7  for i=1:m
8      cond = norm(position(1,:) - foods(i,:));
9      if (cond < dist)
10         dist = cond;
11         x = foods(i,1);
12         y = foods(i,2);
13     end
14 end
15
16 end
```

This function gives back the nearest food source from a certain position.

1-16 It's almost the same as `nearest_landmark`, but in this case we have an output that gives only the coordinates of the food source (because the food doesn't point anything).

3.16 Update_global_vector

```
1  function [x, y] = update_glob_vect(prev_vect, new_vect)
2
3  x = prev_vect(1,1) + new_vect(1,1);
4  y = prev_vect(1,2) + new_vect(1,2);
5
6  end
```

This function updates the global vector on each iteration (on each movement of the ant).

1 The inputs are the previous global vector and the new vector created by the ant thanks to its movement from a i^{th} position to the $(i+1)^{th}$. The output are the updated components of the new global vector.

3-4 We simple add the previous global vector to this new one.

3.17 Landmark_pointer

```
1  function [dX,dY] = landmark_pointer(landmarks,landmark)
2
3  [m,~] = size(landmarks);
4  dX = 0;
5  dY = 0;
6
7  for i = 1:m
8      if (landmarks(i,1) == landmark(1) && landmarks(i,2) ==
           landmark(2))
9          dX = landmarks(i,3);
10         dY = landmarks(i,4);
11         break;
12     end
13 end
14
15 end
```

This function tells which is the landmark pointed by another given landmark.

1 The inputs are all the landmarks and the landmark that we are interested about. The outputs are the coordinates of the ponted landmark.

3-5 We count how many landmarks we have, and we set the standard output.

7-13 The loop passes through all the landmarks and when it finds the searched one, it overwrites the output and breaks.

4 The simulation output

Figure 1 shows the ant during the outogoing way: the path (in blue) is randomly created on each step the insect does, and at the same time the global vector (green arrow) is updated. The landmarks are ignorated because food sources could be anywhere, and the searching ant doesn't will to be influenced by any already known signal.

Figure 2 shows the outgoing path until the ant has found a food source. The ant starts to use the global vector it has calculated before, and follows it. By doing this, the insect covers a path that points directly toward the nest.

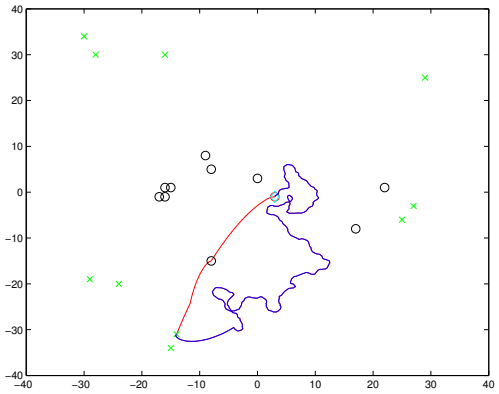


Figure 1: Simulation 1

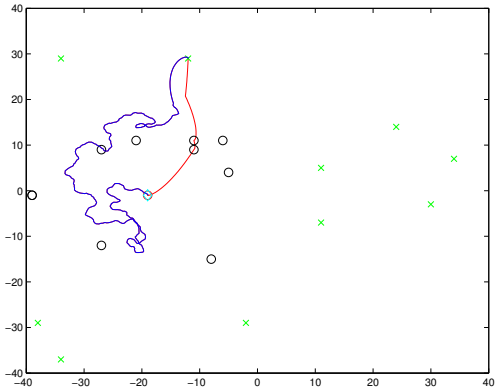


Figure 2: Simulation 2

During the way, the global vector is continuosly updated, so that in case of unexpected event the ant still has an idea about where the nest is.

During the path towards home it may happen that the ant encounter a known Landmark: as shown in Figure 2, if one of those landmarks comes in sight of the insect, this stops following the global vector (but still updates it) and goes to the Landmark. As shown in this image, each Landmark is coupeled with a local vector pointing to the next nearest Landmark, or to the nest: the ant starts following this series of local vector, that although doesn't lead to the shortest way, are a known sequence of signals towards home.

5 Summary and outlook

Our work has developed starting with some assumptions about the model used by ants to integrate their paths in order to return home following a straight way. Literature reports results of experiments showing how the insects use multiple instruments to know where the nest is in rapport to the actual position: polarized light is used to determine the amplitude of the turns ants do during the outgoing path, the distance they cover is measured thanks to the auto induced optic flow, and a simplified version of the path integration model is used to summarize all these information in a global vector that stores the direction and the distance of the nest from the current position.

An important concept is that ants have no proper representation of their position in the space, they only know where they are in terms of angle and distance from a starting point: in other words, they are not able to imagine the environment as a map and the instrument they use to find the way back home is mostly a compass based on the scattering of sun light.

The model we produced shows how the random path followed by the insects during the research of food is constantly summarized in a vector pointing to the nest: the ant will start following this vector to find the nest, and it will only change way if it meet an already known landmark. This landmark points to the next nearest landmark, until the nest is reached: even if this may not be the shortest path anymore, following a known signal represent a safer way to go home because this permits to avoid all the eventual errors in the computation of the global vector. To note is that the global vector is still updated, also when a landmark is followed: even in case of unexpected misleading landmarks, the ants always has an idea about where the nest is.

Our work only gives a general and simplified idea about the sophisticated method ants use to integrate their paths: we assumed that the insects live in a stable environment, with known landmarks and no unexpected variations during the way. The model may be improved by adding some features about how the insect will react in case of a misleading landmark (for example the rain may move known landmarks in new positions) or what happens when new landmarks need to be added to the list of known ones (like in case of building a path back home from a new feeding site). This last case can be assumed in another feature we left out of our model for reasons of simplicity: ants tend to show a sector fidelity. Implementing this will probably make more easy to implement also some of the other features we mentioned, but our model is randomly generated every time it's started, so we decided to

work only with instances that already know all the needed information, and isn't able to learn new ones.

For sure ants have optimized the method of path integration in a way that permits them to gain an astonishing precision in calculating a straight path out of the random way they follow during the research of food. The small brain of the insect has developed a capability to retain only the relevant information and to use it directly in order to minimize the number of elements that have to be remembered by the ant. We only covered a really small part of the topic, and big improvements are always possible: we would like to pose our project as a base to more detailed studies, the instruments that we've generated can be used in order to show a lot of numerical and graphical information that may be utilized to show an enlarged range of models that can be applied to further studies that remained out of our goals.

References

[1] Wehner, R.; Wehner, S.
Insect Navigation: use of maps or Adriane's thread
Ethology Ecology and Evolution 2:27-48, 1990

[2] Wittlinger, M.; Wehner, R; Wolf, H.
The desert ant odometer: a stride integrator that accounts for the stride length and walking speed
The Journal of Experimental Biology 210:198-207, 2007

[3] Wehner, R.; Ronacher, B.
Desert ants Cataglyphis fortis use self-induced optic flow to measure distance travelled
Comp Physiol A 177:21-27, 1995

[4] Wehner, R.
Desert ant navigation: how miniature brains solve complex tasks
Comp Physiol A 189:579-588, 2003

[5] Müller, M.; Wehner, R.
Path integration in desert ants, Cataglyphis fortis
Proc. Natl. Acad. Sci. USA 85:5287-5290, 1988

[6] Collett, M.; Collett, T. S.; Bisch, S.; Wehner, R.
Local and global vectors in desert ant navigation
Nature, Macmillan Publishers 394:269-272, 1998

[7] Wittlinger, M.; Wehner, R; Wolf, H.
The ant odometer: stepping on stilts and stumps
ScienceMag 312:1965-1967, 2006

[8] Wehner, R.; Wehner, S.
Path integration in desert ants: approaching a long-standing puzzle in insect navigation
Monitore zool. ital. (N.5) 20:309-331, 1986

Cover image sorce: http://pixdaus.com/pics/1289838580DsHcqZ6.jpg

Licence Agreement

We hereby agree to make our source code of 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.