PSBC Project 2

March 29, 2019

1 Introduction

This report details the approach and implementation relating to the second coursework project of MATH36032. Please note that all programming and testing was carried out using GNU Octave rather than MATLAB.

2 The Problem

The goal of this project is to investigate the dynamics between a fox and a rabbit in a specific scene. The goal of the rabbit is to reach it is a burrow safely. Meanwhile, the goal of the fox is to catch the rabbit. We will aim to answer the question of which outcome occurs based on a set of input parameters.

To do this, we will use differential equations, solving them numerically using Octave's `ode45` solver.

3 The Scene

The scene we will be working with is as follows. Its main feature is a warehouse defined by its two westernmost corners and stretching out infinitely to the east. The two westernmost corners will be denoted as NW and SW for northwest and southwest respectively. Of course, neither creature can see through, or run through the warehouse. The scene will contain the rabbit's burrow (denoted B). We will denote the position of the rabbit by R and the fox by F. Figure 1 shows an example configuration.

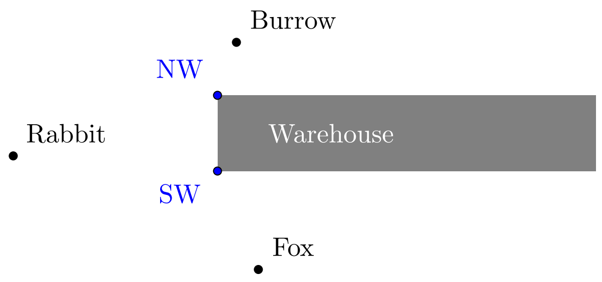
3.0.1 Rules

The rabbit moves in a straightforward way, directly towards the burrow with constant speed s_{r0}. The fox moves with a constant speed of s_{f0}, but its direction depends on the following rules;

- If the rabbit is within the fox's line of sight, the fox runs directly towards the rabbit.
- If a corner of the warehouse obscures the rabbit, then the fox runs towards that corner.
- If the fox reaches a corner and the rabbit still is not in sight, it follows the warehouse perimeter until it sees the rabbit.

Before continuing, we must make a few assumptions as to what our model can support. They are as follows.

Figure 1: An example configuration of the scene



- The rabbit is within the fox's line of sight initially.
- The burrow is within the rabbit's line of sight initially.

4 The General Approach

The approach we will use is to model the scene using Octave's built-in `ode45` function, which is a numerical solver for ODEs. We will take advantage of the fact that it is a numerical solver, not a symbolic one. This fact will allow us to change the direction of the fox within the ODE function based on the relative positions of the fox, rabbit and warehouse at every timestep.

We can do this because of the iterative nature of the solver, meaning that changing the direction in one timestep will not affect the values computed for previous timesteps.

5 The Solution

The first two inputs to our function defining the ODE are time t, and the vector z holding the solution at t, this is defined by

$$z = \begin{bmatrix} R_x \\ R_y \\ F_x \\ F_y \end{bmatrix}.$$

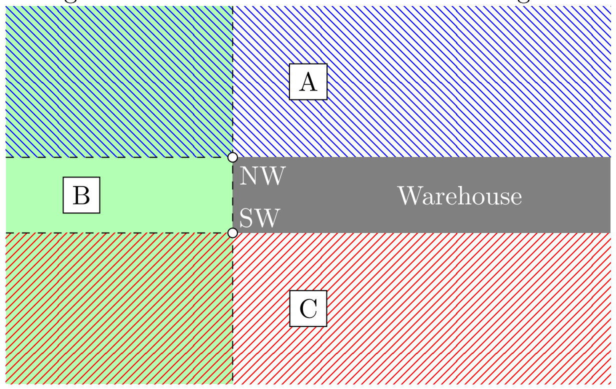
Within our ODE function, we must calculate the derivative of the elements of z with respect to time.

$$\dot{z} = \begin{bmatrix} \dot{R}_x \\ \dot{R}_y \\ \dot{F}_x \\ \dot{F}_y \end{bmatrix}.$$

In other words, we will calculate the velocity of the fox and rabbit in the x and y directions at time t.

We will break the problem up into smaller pieces and solve these small pieces first before looking at the solution as a whole. In practice, this means examining small helper functions which will contribute to the final solution.

Figure 2: The scene divided into three regions



5.1 Determining the Fox's Target

The most challenging aspect of modelling this scene is to determine the direction in which the fox should run at any instant, following 3.0.1. We will solve this problem by splitting our scene up into three overlapping regions A, B and C defined as follows (and visualised in Figure 2).

$$A = \{(x,y) \in \mathbb{R}^2 \mid y > NW_y\},$$
$$B = \{(x,y) \in \mathbb{R}^2 \mid x < NW_x\},$$
$$C = \{(x,y) \in \mathbb{R}^2 \mid y < SW_y\}.$$

The rules dictate that the fox always runs towards something, either the rabbit or one of the corners. We refer to this as the fox's target.

If both the fox and rabbit are in the same zone, then the fox's target is the rabbit since there must be a line of sight between them. Now suppose this is not the case and the fox and the rabbit are in different zones. Let L denote the line connecting the fox and the rabbit, with equation $y = mx + c$. Notice that since the two are in different zones, then if L intersects one of the warehouse walls, then there is no line of sight.

We can save on one intersection check by observing that the warehouse in convex and therefore if L intersects the north wall, then there must be an intersection with one of the other two. Hence we need only check if L intersects the south or west wall of the warehouse (it would be equally valid only to check the north and west walls).

Let Δ_x, Δ_y denote the horizontal and vertical distances respectively between the fox, and the rabbit. The gradient m of L therefore is $\frac{\Delta_y}{\Delta_x}$. In the equation above c is referred to as the intercept (the y coordinate when $x = 0$). This is calculated $c = F_y - m \times F_x$.

West wall

To check if L intersects the west wall, we must calculate the y coordinate of L at $x = SW_x$. If this value lies between SW_y and NW_y then we have an intersection. Mathematically, we check if the following holds.

$$SW_Y < m \times SW_x + c < NW_y.$$

South wall

Intersection with the south wall happens if the x coordinate where L intersects the line $y = SW_y$ is strictly greater than SW_x. By rearranging the equation of L, we see that we have an intersection when the following holds.

$$\frac{SW_y - c}{m} > SW_x.$$

We can conclude that if either case holds then the fox does not have a line of sight to the rabbit and hence the rabbit cannot be the fox's target. Therefore, the northwest corner and southwest corner of the warehouse the only two possibilities.

So suppose the fox cannot see the rabbit. If the fox is in zone A, then the target is NW clearly, since the closest corner to the fox is NW. Similarly, if the fox is in zone C then the target must be SW.

The final case is when the fox is in zone B but *not* in zones A or C. In this case, the fox's target depends on the location of the rabbit. If the rabbit is in zone A then the target is NW. Otherwise, the target is SW. A simple way to check this is to consider whether the gradient m is positive or negative. If it is positive, then the rabbit must be in zone A, and if negative it must be in zone C.

These cases also account for the rule that states that the fox follows the perimeter if it is at one of the corners but cannot see the rabbit. Let us look at the case where the fox is at NW and cannot see the rabbit. Then it must be the case that the rabbit is southeast of the warehouse, making the gradient negative, meaning the fox target will be SW.

Conversely, if the fox is at SW and cannot see the rabbit, then the rabbit is northeast of SW. Therefore the gradient must be positive, making the target NW as required.

The implementation of this function is listed in Listing 1. The first two lines of this function demand some explanation. They define variables `X` and `Y` to use when indexing the vectors. The reasoning is as follows. Given a vector `a = [xComponent, yComponent]`, it is far more readable to write code using `a(X)` and `a(Y)`, than `a(1)` and `a(2)`.

Listing 1: A function to compute the fox's target given the warehouse coordinates and the rabbit's position.

```
function foxTarget = computeFoxTarget(rabbitPos, foxPos, NW, SW)
  Y = 2; % index of y coordinate in the vectos
  X = 1; % index of x coordinate in the vectors

  foxTarget = rabbitPos;

  % The case when both in the same zone
  if foxPos(Y) > NW(Y) && rabbitPos(Y) > NW(Y) ... % Both north
      || foxPos(X) < NW(X) && rabbitPos(X) < NW(X) ... % Both west
      || foxPos(Y) < SW(Y) && rabbitPos(Y) < SW(Y) % Both south
    return;
  end

  intersectsSouth = false;
  intersectsWest = false;

  % Account for this case seperately to prevent division by zero
  % when calculating the gradient
  if rabbitPos(X) == foxPos(X)
    intersectsSouth = true;
  else
    % The gradient and intercept of the line between the fox and rabbit
    gradient = (rabbitPos(Y) - foxPos(Y)) / (rabbitPos(X) - foxPos(X));
    yIntercept = foxPos(Y) - gradient * foxPos(X);

    % Whether this line intersects the east or south of the warehouse
    intersectsSouth = ((SW(Y) - yIntercept) / gradient) > SW(X);

    intersectionWithWest = gradient * SW(X) + yIntercept;
    intersectsWest = intersectionWithWest < NW(Y) && intersectionWithWest > SW(Y);
  end

  if intersectsSouth || intersectsWest

    if foxPos(Y) > NW(Y) || (foxPos(Y) > SW(Y) && gradient > 0)
      foxTarget = NW;
    else
      foxTarget = SW;
    end
  end


end
```

5.2 Velocities in the x and y directions

We've seen in part 5.1 that the fox always runs towards some target with constant speed. Furthermore, the rabbit perpetually runs towards its burrow, also with constant speed. Therefore, we can say that a creature C runs towards a target T with a constant speed of u. We seek it's velocity in the x direction (v_x) and in the y direction (v_y).

Figure 3: The scenario when a creature C runs towards a target T.

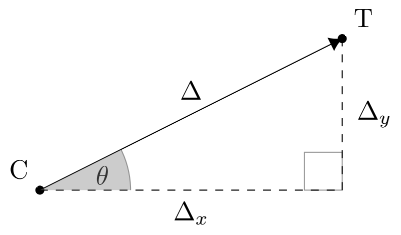


Figure 5.2 shows the scene. Using this we denote the angle between the creature and the target by θ and the distance between them by Δ. We can then easily deduce that

$$v_x = u\cos\theta,$$

$$v_y = u\sin\theta.$$

Furthermore, we see there's no need to compute θ. We can observe that

$$\cos\theta = \frac{\Delta_x}{\Delta}$$

$$\sin\theta = \frac{\Delta_y}{\Delta}.$$

We can then combine these equations giving our final equations

$$v_x = u\frac{\Delta_x}{\Delta},$$

$$v_y = u\frac{\Delta_y}{\Delta}.$$

The advantage of using these equations rather than computing θ using trigonometry is that we can implement both components in one line using Octave's vector based arithmetic. We can see this in our implementation in Listing 2.

Note, this function calls a function called `distance`, which simply computes the distance between two points. It is listed as Listing 7 in Apprendix A.

Listing 2: A function to compute the velocity of an entity facing a target with a given speed.

```
% For an entity directed at a target with a given speed
% return the x and y components of it's velocity
function velocity = computeVelocity(speed, pos, target)
  dist = distance(pos, target);
  velocity = speed .* ((target - pos) ./ dist);
end
```

5.3 Stopping the ODE solver

We can use an event function to stop the computation when our one of our end conditions is satisfied. Two events should stop the calculation, the fox catching the rabbit, and the rabbit reaching the burrow.

We will use a similar approach for both. The rabbit is considered caught if it is within 0.1 meters of the fox. Similarly, it is deemed to be in its burrow if it is within 0.1 meters of it. *Note* that this could present an issue in situations where the fox is very close to the rabbit at the same time as the rabbit is very close to the burrow. It could be ambiguous whether the rabbit is safe or not. In these cases, it may be appropriate to choose a smaller value than 0.1 to ensure the correct result is output.

Let the distance between the fox and the rabbit be $\Delta_{f,r}$. Our event function should trigger the ODE solver to stop when $\Delta_{f,r}$ becomes less than 0.1. In other words when $\Delta_{f,r} - 0.1$ becomes negative. The case when the rabbit reaches the burrow is similar. The implementation is listed in Listing 3.

Listing 3: The event function used to stop the ODE solver.

```
function [value,isterminal,direction] = stopEvent(t, z, burrowPos)
  tolerance = 0.1;
  % z = [Rx, Ry, Fx, Fy]

  % Fox catches the rabbit
  value(1) = distance(z(1:2), z(3:4)) - tolerance;
  isterminal(1) = 1;
  direction(1) = -1;

  % Rabbit reaches it's burrow
  value(2) = distance(z(1:2), burrowPos') - tolerance;
  isterminal(2) = 1;
  direction(2) = -1;
end
```

5.4 The ODE function

With this in place, we can now examine the ODE function itself, included in Listing 4. We can see that we compute the rabbit's velocity directly, and compute the fox's velocity based on its trajectory.

Listing 4: The ODE function used to model the fox chasing the rabbit

```
function dzdt = modelODE(t, z, burrowPos, rabbitSpeed, ...
                         foxSpeed, warehouseNW, warehouseSW)
  Y = 2; % index of y in pos
  X = 1; % index of x in pos

  % z = [Rx, Ry, Fx, Fy]
  dzdt = zeros(size(z));

  % Converted into row vectors to match input parameters
  rabbitPos = z(1:2)';
  foxPos = z(3:4)';

  % rabbit velocity
  dzdt(1:2) = computeVelocity(rabbitSpeed, rabbitPos, burrowPos)';

  % fox velocity
  foxTarget = computeFoxTarget(rabbitPos, foxPos, warehouseNW, warehouseSW);
  dzdt(3:4) = computeVelocity(foxSpeed, foxPos, foxTarget)';

end
```

5.5 An example

Putting together the components we have just explored and using the following values for the parameters, we can run the model using the code in Listing 5. This codes sets up the configuration for the scene and uses `ode45` to solve the ODE. The `MaxStep` option is used on the ODE solver to ensure that time increments of no more than `0.1` are used since the fox is continuously changing direction depending on its target and its essential to ensure these checks are done often to guarantee the accuracy of the model. Of course, it is also vital we do not make this value too small as this would negatively impact performance (in terms of speed). The value `0.1` was chosen as it produces a seemingly accurate simulation while not taking too long to complete the computation.

The results are displayed in Figures 4 and 5. As we can see, the fox fails to catch the rabbit in these conditions. The `plotScene` function is included in Appendix B in Listing 8 for completeness.

Parameter	**Value**
NW	(200, 0)
SW	(200, -400)
R	(0, 0)
F	(250, -550)
B	(600, 600)
s_{r0}	13m/s
s_{f0}	16m/s

Listing 5: The code to run the model with a set of parameters.

```
warning('off', 'integrate_adaptive:unexpected_termination');

% Speeds in meters per second
rabbitSpeed = 13;
foxSpeed = 16;

% Coordinates in meters
rabbitPos = [0, 0];
foxPos = [250, -550];
burrowPos = [600, 600];

warehouseNW = [200, 0];
warehouseSW = [200 -400];

messages = {'the fox caught the rabbit.', 'the rabbit reached the burrow.'};
printSol = @(time, index) printf("\nAt time %f, %s\n\n", time, messages{index});

options = odeset('Events', @(t,z)stopEvent(t,z, burrowPos), 'MaxStep', 0.1);

[~, z, tEvent, ~, iEvent] = ode45(@(t, z) modelODE(t, z, burrowPos, ...
                                          rabbitSpeed, foxSpeed,...
                                          warehouseNW, warehouseSW), ...
                 [0 300], [rabbitPos foxPos], options);


printSol(tEvent, iEvent);
plotScene(burrowPos, warehouseNW, warehouseSW, z, 1, "./report/simpleModel.eps");
```

Figure 4: The output from running the simple model.

```
>> part1

At time 65.263703, the rabbit reached the burrow.
```

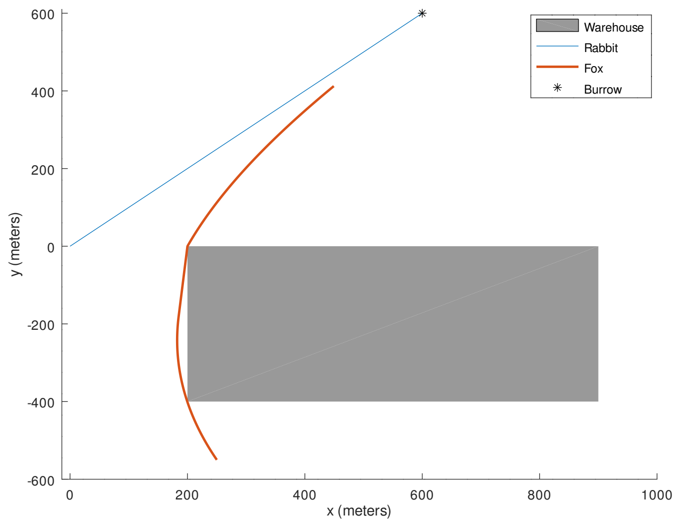
6 A more realistic scenario

Thus far, our model has assumed the fox and rabbit both run at constant speed. However, in reality, both will tire as they run, resulting in their speed diminishing. The amount by which their speed diminishes depends on the distance they've currently travelled. We will define their speeds at time t by the following.

$$\begin{aligned} s_f(t) &= s_{f0} \times e^{-\lambda_f d_f(t)}, \\ s_r(t) &= s_{r0} \times e^{-\lambda_r d_r(t)}, \end{aligned} \tag{1}$$

Where λ_f and λ_r are the rates by which the fox and rabbit's speeds diminish. As before, s_{f0} and s_{r0} are the initial speeds of the fox and rabbit respectively. The values $d_f(t)$ and $d_r(t)$ are the distance the fox and the rabbit respectively, have travelled up to time $t > 0$.

Figure 5: The paths of the fox and rabbit under this configuration.



Immediately, we can see that the vast majority of our model can remain unchanged. We must replace the constant speeds we've been using thus far with the equations above which is very straightforward. The only complexity comes in computing the distance that the fox and rabbit have travelled up to time t.

6.1 Computing the distance travelled

The method here is identical for the fox and the rabbit. For a creature C we can interpret this model as parametric equations, it's position given by $x(t)$ and $y(t)$, both functions of time.

As we've already seen visually in Figure 5, this traces a line in the $x - y$ plane. We're interested in computing the length of that line from $t = 0$ up to an arbitrary t. We know that the length of a line L defined in this way, between $t = 0$ and $t = T$ is equal to

$$d = \int_0^T \sqrt{(\frac{dx}{dt})^2 + (\frac{dy}{dt})^2}\, dt.$$

It then follows directly that $\dot{d} = \sqrt{\frac{dx}{dt}^2 + \frac{dy}{dt}^2}$. But $\frac{dx}{dt}$ is the horizontal velocity and $\frac{dy}{dt}$ is the vertical velocity which we already know how to compute. By taking the square root of the sum of their squares we get back to the speed of the creature.

Hence we can just set $\dot{d_f}(t)$ and $\dot{d_r}(t)$ equal to the speed calculated in Equation 1.

6.2 The new ODE function

Given this justification for the computation of the distance travelled, we can easily make these changes to our ODE function. We add the distance travelled by the fox and the rabbit to our `z` vector meaning we now have the following.

$$z = \begin{bmatrix} R_x \\ R_y \\ F_x \\ F_y \\ d_r(t) \\ d_f(t) \end{bmatrix}.$$

Which means the derivative of this that we must calculate is defined as

$$\dot{z} = \begin{bmatrix} \dot{R}_x \\ \dot{R}_y \\ \dot{F}_x \\ \dot{F}_y \\ s_r(t) \\ s_f(t) \end{bmatrix}.$$

The altered version is listed in Listing 6.

Listing 6: The ODE function taking into account diminishing speeds.

```
function dzdt = realisticModelODE(t, z, burrowPos, ...
                                rabbitSpeed, foxSpeed, ...
                                warehouseNW, warehouseSW, ...
                                rabbitSpeedRate, foxSpeedRate)
  % z = [Rx, Ry, Fx, Fy, Dr, Df]
  dzdt = zeros(size(z));

  % Converted into row vectors to match input parameters
  rabbitPos = z(1:2)';
  foxPos = z(3:4)';

  % rabbit velocity
  currentRabbitSpeed = rabbitSpeed * exp(-rabbitSpeedRate * z(5));
  dzdt(1:2) = computeVelocity(currentRabbitSpeed, rabbitPos, burrowPos)';

  % fox velocity
  foxTarget = computeFoxTarget(rabbitPos, foxPos, warehouseNW, warehouseSW);
  currentFoxSpeed = foxSpeed * exp(-foxSpeedRate * z(6));

  dzdt(3:4) = computeVelocity(currentFoxSpeed, foxPos, foxTarget)';

  % The current speeds to compute the distance travelled
  dzdt(5) = currentRabbitSpeed;
  dzdt(6) = currentFoxSpeed;

end
```

6.3 An example

Let's take the same example as we had in section 5.5. We must define values for the rates of diminishing speed of both creatures. These will be set as follows, $\lambda_f = 0.0002m^{-1}$ and $\lambda_r = 0.0007m^{-1}$. The code used to run the ode solver is almost identical to that used in our original problem; the code is included as Listing 9 in Appendix C for completeness.

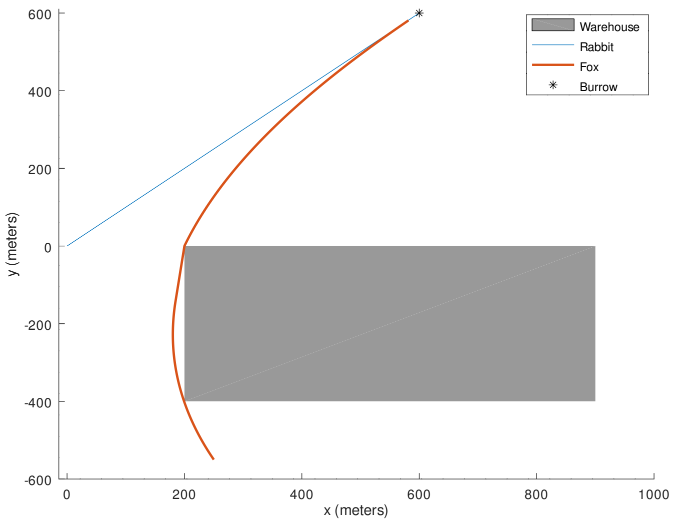
More interesting are the results which are shown in Figures 6 and 7. As we can see the fox still fails to catch the rabbit but does get much closer.

Figure 6: The output from running the more realistic model.

```
>> part2

At time 89.750793, the rabbit reached the burrow.
```

Figure 7: The paths of the fox and rabbit in the more realistic example.



Appendix A Distance

Listing 7: A function to compute the distance between two points.

```
% Distance between two points P1 and P2
function dist = distance(p1, p2)
  dist = sqrt(sum((p2 - p1) .^ 2));
end
```

Appendix B The `plotScene` function

Listing 8: A function to plot the results of the ODE solver and the rest of the scene.

```
function plotScene(burrowPos, warehouseNW, warehouseSW, z, fig, title)

  figure(fig);
  patch([warehouseSW(1) 900 900 warehouseSW(1)], [warehouseSW(2) warehouseSW(2) ...
      warehouseNW(2) warehouseNW(2)], 'FaceColor', [0.6 0.6 0.6], 'LineStyle', ...
      'none');
  hold on;

  plot(z(:,1), z(:,2))
  hold on;
  plot(z(:,3), z(:,4), 'LineWidth', 1.8);
  hold on;
  plot(burrowPos(1), burrowPos(2), 'k*');
  hold on;
  axis([-14 1000 -600 610]);
  hold on;
  xlabel('x (meters)');
  ylabel('y (meters)');
  legend({'Warehouse', 'Rabbit', 'Fox', 'Burrow'});

  print("-depsc", title);

  hold off;

end
```

Appendix C Running the more realistic model

Listing 9: The code used to run the more realistic model of the scene.

```
warning('off', 'integrate_adaptive:unexpected_termination');

% Speeds in meters per second
rabbitSpeed = 13;
foxSpeed = 16;

% Coordinates in meters
rabbitPos = [0, 0];
foxPos = [250, -550];
burrowPos = [600, 600];

warehouseNW = [200, 0];
warehouseSW = [200 -400];

% The rates of diminishing speeds
foxSpeedRate = 2e-4;
rabbitSpeedRate = 7e-4;


messages = {'the fox caught the rabbit.', 'the rabbit reached the burrow.'};
printSol = @(time, index) printf("\nAt time %f, %s\n\n", time, messages{index});

options = odeset('Events', @(t,z)stopEvent(t,z, burrowPos), 'MaxStep', 0.1);
[~, z, tEvent, ~, iEvent] = ode45(@(t, z) realisticModelODE(t, z, burrowPos, ...
                                       rabbitSpeed, foxSpeed, ...
                                       warehouseNW, warehouseSW, ...
```

```
                                          rabbitSpeedRate, foxSpeedRate), ...
                 [0 500], [rabbitPos foxPos 0 0], options);

printSol(tEvent, iEvent);

plotScene(burrowPos, warehouseNW, warehouseSW, z, 1, "./report/realisticModel.eps");
```