Fox Rabbit Pursuit Simulation

Yann McLatchie
Student ID: 10161640

Introduction

In this project, we consider a pursuit problem in which a rabbit follows a predefined and known path $(r_1(t), r_2(t))$ towards its burrow from the origin at velocity $s_r(t)$ as it is chased by a fox. Between the fox's initial position at $(250, -550)$ and the rabbit's burrow is an impenetrable warehouse through which the fox can neither see nor run. While the fox can see the rabbit, that is there exists a line segment connecting the two that does not intersect with the walls of the warehouse, it runs directly towards it with velocity $s_f(t)$. If the fox's view of the rabbit is impeded by the South wall of the warehouse, then the fox runs directly towards the South West corner, located at $(200, -400)$. Upon reaching this corner, while the rabbit is still not in sight, the fox runs parallel to the West wall of the warehouse until there is a clear line of sight between the two, at which point the fox runs directly towards the rabbit once more.

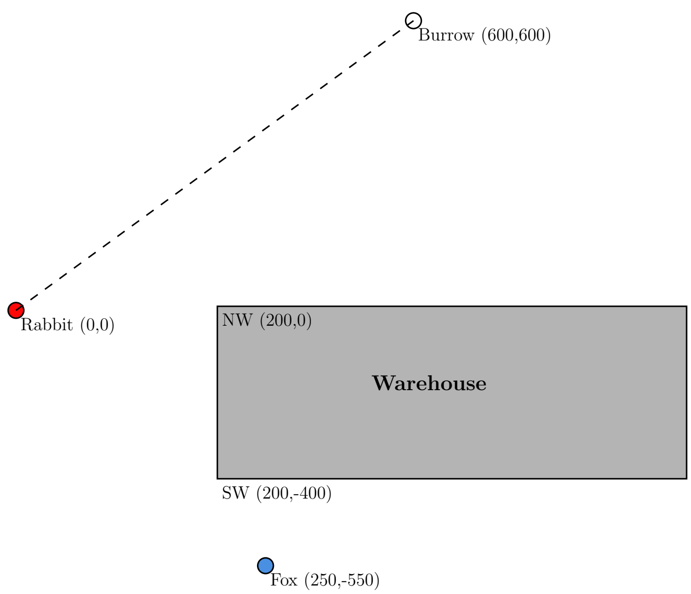

Figure 1: Map showing the fox and rabbit in their initial positions, the warehouse and the burrow.

While the warehouse is assumed to extend an infinite distance Eastwards such that the fox will always run around it clockwise, we will later define a South East corner to determine whether or not the fox's view is blocked. The rabbit's path throughout this project will be a straight line towards its burrow, not engaging in any evasive manoeuvres and hoping its head start is sufficient to get it back to safety. Therein lies our goal: to determine whether, given the initial position of the fox, the animals' initial speeds, and the rate of speed decrease per meter travelled, the rabbit returns to its burrow before being caught by the fox. Over the course of this project, we will investigate this with both constant and decreasing speeds for both animals. The full code for the project is given in the Appendix.

1 Constant Speeds

To begin with, we assume that both the fox and the rabbit travel at constant speeds $s_f = 16ms^{-1}$ and $s_r = 13ms^{-1}$ respectively. We consider the rabbit to be captured if the fox gets to within $0.1m$ of its position, which we define as the `mindist` in the simulation. The rabbit's position at time t is given by

$$\begin{pmatrix} r_x(t) \\ r_y(t) \end{pmatrix} = \sin(\pi/4) \begin{pmatrix} s_r t \\ s_r t \end{pmatrix},$$

or in MATLAB as `r = sin(pi/4) * [speedRab*t;speedRab*t]` such that theoretically after 65.2714 seconds, if it is not caught in that time, the rabbit is back in its burrow. Meanwhile, the fox's path is split up into several legs: first it must clear the SW corner, then the NW corner, and finally it must chase the rabbit in the final sprint to the burrow. To begin with the fox can see the rabbit, and so runs directly at it. Letting $\vec{F}(t)$ be the location of the fox at time t, and $\vec{R}(t)$ that of the rabbit, the derivative of the fox's location - its velocity - is formally defined as

$$\frac{\vec{F}'(t)}{\|\vec{F}(t)\|} = \frac{\vec{R}(t) - \vec{F}(t)}{\|\vec{R}(t) - \vec{F}(t)\|}$$

to be directed towards the rabbit, where $\|\vec{F}(t)\| = s_f$ (Cabrera and Negrón-Marrero, 2011). Rearranging, we are able to input this into the `ode45` function call as the following derivative.

```
    r = sin(pi/4)*[speedRab*t;speedRab*t]; % rabbit's location at time t
    dist = max(norm(r-z),1e-6); % to avoid division by zero
    dzdt = (s_f*(r-z)) / dist; % fox's velocity vector at time t
```

Using the `odeset` functionality, we can define the events at which we wish to stop the integration. We set four of these initially: the fox catches the rabbit, the rabbit makes it back to the burrow, the view of the fox becomes obstructed by the South wall, and its view is obstructed by the West wall. If the fox catches the rabbit at this point, or if the rabbit returns to its burrow, the integration ends. These two events are given as

```
    % Fox catches rabbit
    value(1) = dist - mindist;
    isterminal(1) = 1;
    direction(1) = -1;
    % Rabbit back to burrow
    value(2) = r(1) - burrow(1);
    isterminal(2) = 1;
    direction(2) = 1;
```

respectively. `direction(1) = -1` indicates that we stop the simulation only when `value(1)` is decreasing through zero, and similarly `direction(2) = 1` asserts that we want `value(2)` to be growing through zero (Higham and Higham, 2017).

However, if the view of the fox is obstructed by the South wall, we define a new ODE to describe the fox's trajectory. In this case, letting the corner be $(C_x, C_y) = (200, -400)$, the fox's velocity is now given as

$$\frac{\vec{F}'(t)}{\|\vec{F}(t)\|} = \frac{\vec{C}(t) - \vec{F}(t)}{\|\vec{C}(t) - \vec{F}(t)\|},$$

which can once more provide our `ode45` solver as

```
    sw = [200 -400]; % South West corner
    dist = max(norm(sw-z),1e-6);
    dzdt = (speedFox*(sw-z)) / dist;
```

In order to determine whether the fox's view has been obstructed, we check if the two line segments joining the fox and the rabbit, and the South West and South East corners intersect. In order to check this, we use the native `polyxpoly` function that tests whether the two line segments intersect, and if so returns the intersection coordinates.

We can check these conditions without the use of MATLAB functions using the algorithm given in Cormen (2001) by taking any one of the four endpoints of the two segments, let's say p_1 from the line p_1p_2, and investigating its orientation with respect to the two endpoints of the other line, p_3 and p_4. Without loss of generality, let p_1 lie to the left of the segment p_3p_4, if we were to draw a line connecting p_1 to p_3 this would be an anti-clockwise rotation of the vector $\overrightarrow{p_3p_4}$. We check this by calculating the cross-product of the two. By checking if the opposite is true for p_2, then we know that p_1 lies to the left of p_3p_4 and p_2 lies to its right, meaning the line segment p_1p_2 straddles p_3p_4. Repeating these checks for p_3 and p_4 we check that p_3p_4 straddles p_1p_2. Therefore, we can check if the view of the fox is blocked by the South wall with the following code, modified from Stafford (2015).

```
dt1=det([1,1,1;x(1),x(2),x(3);y(1),y(2),y(3)]) * ...
    det([1,1,1;x(1),x(2),x(4);y(1),y(2),y(4)]);
dt2=det([1,1,1;x(1),x(3),x(4);y(1),y(3),y(4)]) * ...
    det([1,1,1;x(2),x(3),x(4);y(2),y(3),y(4)]);
if(dt1<=0 && dt2<=0)
    intercept(1) = 1;
else
    intercept(1) = 0;
end
```

This runs very quickly since the matrices are small, although for maintenance and brevity, we will use `polyxpoly`. We terminate this integration when either the fox reaches the South West corner, or the rabbit reaches its burrow.

If the fox makes it past the South West corner and the rabbit is in sight, it will run straight towards it as before. If, however, it finds that the rabbit is no longer in sight it will run due North, parallel to the West wall of the warehouse. The ODE governing the fox's trajectory for this leg is simply

$$\vec{F}'(t) = \begin{pmatrix} F'_x(t) \\ F'_y(t) \end{pmatrix} = \begin{pmatrix} 0 \\ s_f \end{pmatrix}$$

since the fox only moves vertically at its initial speed. In MATLAB, this is given as `westPerimeterODE = @(t, z)[0; speedFox]`. The fox continues along this path until it can see the rabbit once more, at which point it runs straight towards it according to the first ODE discussed. In order to determine this event, we apply the same methology and algorithm as was used to check the visibility through the South wall with `polyxpoly`, only using the North West and South West corners instead of the South East and South West; if the two lines don't intersect, the fox has a clear view of the rabbit.

In order to handle these different legs, we execute various calls of `ode45` with the different ODEs and events function, and keep track of which event triggers their termination. When we call `ode45`, we are required to input a time span to run the integration over. At each leg we make a guess at this time frame by estimating the endpoint with the equation time = distance/speed, using the distance between the fox and the rabbit as the numerator, and the difference in their respective speeds as the denominator. The endpoint of an `ode45` call becomes the starting time of the next one, `tspan = [te norm(ze-re)/(initSpeedFox-initSpeedRab)]`, or $t = 0$ for the first function call.

We first run `ode45`, with the fox running directly at the rabbit. When this terminates, if the rabbit has not been caught and is not in its burrow, we know that the fox's view has been obscured by one of the warehouse walls, and we enter a while loop. This is shown in the psuedocode below.

```
fox runs directly at rabbit
while rabbitNotCaught and rabbitNotHome:
    if viewObscuredBySouthWall
        run towards SW corner
    if viewObscuredByWestWall
```

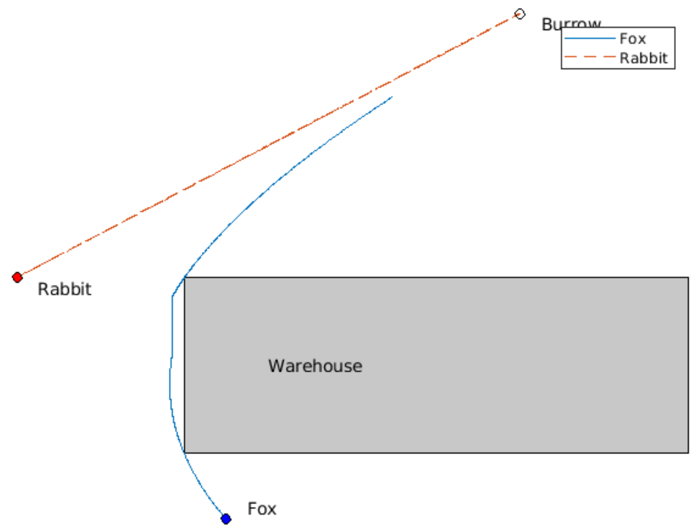

Figure 2: Constant Speeds Pursuit Curves

```
            run vertically
        else
            run directly at rabbit;
```

At every iteration of the `while` statement, an event termination code is generated in the `zi` component of the output `[t,z,te,ze,zi] = ode45(...)`. We re-enter the `while` loop as long as this event termination code does not correspond with the rabbit being caught or making it back to its burrow. Note that it is possible to make it past the South wall, run directly towards the rabbit, and then later to have the fox's view obscured by the West wall.

Running this simulation, we achieve the pursuit paths shown in Figure 2, and find that the rabbit does in fact make it back to its burrow unscathed. The simulation terminates at `te = 65.2714`, the theoretic value, indicating that the simulation ran with a great deal of accuracy. The fox's final coordinates are at `ze = [448.4068 410.7413]`.

2 Variable Speeds

In the second section of this project, we alter the speeds of the animals such that they diminish with distance travelled in an inverse exponential manner. Formally, the speeds of the fox and rabbit are

$$s_f(t) = s_{f_0} e^{-\mu_f d_f(t)} \qquad \text{and} \qquad s_r(t) = s_{r_0} e^{-\mu_r d_r(t)}$$

respectively, with $s_{f_0} = 16ms^{-1}, s_{r_0} = 13ms^{-1}, \mu_f = 0.0002m^{-1}, \mu_r = 0.0008m^{-1}$, and $d_f(t)$ and $d_r(t)$ the distances run by the animals in meters up to time t. We can write the rabbit's velocity in component form as

$$\begin{pmatrix} r_x'(t) \\ r_y'(t) \end{pmatrix} = \begin{pmatrix} r_x'(0)\exp(-\mu_r r_x(t)) \\ r_y'(0)\exp(-\mu_r r_y(t)) \end{pmatrix} = \sin(\pi/4) s_{r_0} \begin{pmatrix} \exp(-\mu_r r_x(t)) \\ \exp(-\mu_r r_y(t)) \end{pmatrix},$$

resulting in an ODE we can solve using MATLAB's native `dsolve` function as follows.

```
    syms r(t);
    r_t = dsolve(diff(r) == initSpeedRab*exp(-mu_r*r), r(0) == 0);
    r = sin(pi/4)*[r_t;r_t]
```

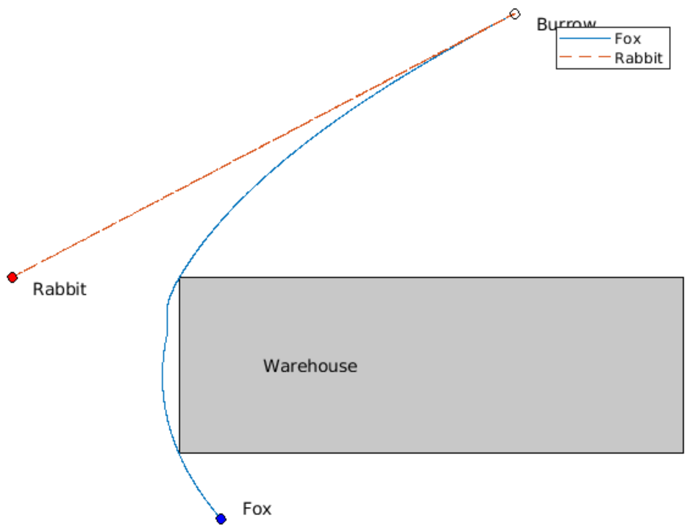

Figure 3: Decreasing Speeds Pursuit Curves

We achieve the following formula describing the rabbit's position at time t,

$$\vec{R}(t) = \begin{pmatrix} 625\sqrt{2}\log\left(\frac{s_{r_0}t}{1250}+1\right) \\ 625\sqrt{2}\log\left(\frac{s_{r_0}t}{1250}+1\right) \end{pmatrix}.$$

Solving for the rabbit's position using `dsolve` means that we can use this code for various values of μ_r, including $\mu_r = 0$ for the constant speeds case. For the fox, we measure the distance travelled up to time t as the arc length distance the fox travelled along the pursuit curve up to time t. Thus we simply initiate a matrix and a scalar at the beginning of our program as

```
    foxDistMat = [z0(1) z0(2)]; totalFoxDist = 0;
```

and then add the following to the `ode45` functions.

```
    foxDistMat = [foxDistMat; reshape(z,[],1)']; % add current location to matrix
    % cumulatively sum the total distance travelled at each time step
    totalFoxDist = totalFoxDist + norm(foxDistMat(end,:)-foxDistMat(end-1,:));
    s_f = initSpeedFox * exp(-mu_f * totalFoxDist);
```

The organisation of the code and the events management functions are the same as in question one, so having made these two changes we are ready to run the simulation. We find that once more the rabbit makes it back to its burrow in time, as shown in the following pursuit curves in Figure 3. The simulation terminates at `te = 93.4187`, which concurs with the theoretical time if the rabbit is not caught, calculated in MATLAB with the following code.

```
    syms r(t);
    r_t = dsolve(diff(r) == initSpeedRab*exp(-mu_r*r), r(0) == 0);
    double(solve(600 == sin(pi/4)*r_t)) % returns 93.4187
```

The fox terminates the simulation at coordinates `ze = [592.7695 592.6999]`, just short of catching the rabbit and once more the rabbit makes it back to the burrow safely.

3 Critical Analysis

We structured the code to be able to accommodate for both constant and variable speeds in the same function. The stability and maintainability of the code are ensured by the use of native MATLAB functions and by performing symbolic computations outwith the `while` loop. These calculations increase run time compared to hard-coded alternatives, but are not more expensive for variable speeds compared to constant speeds. We could perhaps speed up the run time by using `for` or `parfor` (parallel `for`) loops instead of the `while` loop, but this would undermine the logical structure of the code.

Some of the time values in the simulations were compared with theoretic values where we found that they matched. Further, setting the `options` for each `ode45` call to stricter `AbsTol` and `RelTol` than default, and setting `Refine` to 8 the same results were obtained as before, implying that the solution is independent of the parameters and is accurate. We thus have confidence in the methodology used and the results produced by our code.

Appendix

```
function [tfox, zfox, te, ze, zi] = ...
    fox_rabbit(initSpeedRab, initSpeedFox, mu_f, mu_r, z0, burrow, mindist)
% Fox-rabbit pursuit simulation.

% Guess time span
tspan = [0 norm(z0)/(initSpeedFox-initSpeedRab)];
% Get component-wise rabbit's path
y = []; x = []; tfox = []; zfox = [];
syms y(x); % y is the distance covered by the rabbit at time x
r_t = dsolve(diff(y) == initSpeedRab * exp(-mu_r*y), y(0) == 0);
rab = sin(pi/4) * [r_t; r_t];
% Chase rabbit
foxDistMat = [z0(1) z0(2)]; totalFoxDist = 0;
options = odeset( ...
    'Events', @(t, z)chaseEvents(t, z, rab, mindist, burrow), ...
    'RelTol',1e-10,'AbsTol',1e-10,'InitialStep',1e-16,'Refine',8);
[tfox1,zfox1,te,ze,zi] = ode45( ...
    @(t, z)chaseODE(t, z, initSpeedFox, mu_f, rab), tspan, z0, options);
tfox = [tfox;tfox1]; zfox = [zfox;zfox1];

% while the fox has not caught the rabbit and the rabbit is not home
while (zi ~= 1) && (zi ~= 2)
    if zi == 3 % if view obscured by South wall
        func = @(t,z)SWCornerODE(t, z, mu_f, initSpeedFox);
        options = odeset('Events', @(t,z)SWCornerEvents(t, z, rab, burrow));
    elseif zi == 4 % if view obscured by West wall
        func = @(t, z)verticalODE(t, z, initSpeedFox);
        options = odeset('Events', @(t,z)verticalEvents(t, z, rab, burrow), ...
            'RelTol',1e-10,'AbsTol',1e-10,'InitialStep',1e-16,'Refine',8);
    else
        func = @(t,z)chaseODE(t, z, initSpeedFox, mu_f, rab);
        options = odeset('Events', @(t,z)chaseEvents(t, z, rab, mindist, burrow), ...
            'RelTol',1e-10,'AbsTol',1e-10,'InitialStep',1e-16,'Refine',8);
    end
    te = te(end,:); ze = ze(end,:);
    x = te; re = eval(rab);
    tspan = [te norm(ze-re)/(initSpeedFox-initSpeedRab)];
    [tfox2,zfox2,te,ze,zi] = ode45(func, tspan, ze, options);
    tfox = [tfox;tfox2]; zfox = [zfox;zfox2];
    zi = zi(end);
```

```
end

x = tfox; rabPath = [eval(rab) eval(rab)];
% visualise the pursuit curves
figure1 = figure; axes1 = axes('Parent', figure1); hold(axes1,'on'); set(axes1, 'Visible','off');
rectangle('Parent',axes1,'Position',[200 -400 600 400],'FaceColor',1/255*[200,200,200]);
text(300,-200,"Warehouse");
plot(zfox(:,1),zfox(:,2)), hold on;
plot(rabPath(:,1),rabPath(:,2),'--'), hold on;
plot(600, 600, 'ok'), hold on; text(625, 575, "Burrow");
plot(z0(1), z0(2),'ok','MarkerFaceColor','blue'), hold on; text(z0(1)+25, z0(2)+25, "Fox");
plot(0, 0, 'ok','MarkerFaceColor','red'), hold on; text(25, -25, "Rabbit");
legend('Fox','Rabbit');

    % ------------------------ Nested functions ------------------------

    function dzdt = chaseODE(t, z, initSpeedFox, mu_f, rab)
    % Fox-rabbit pursuit simulation ODE.
    % rabbit's path
    x = t; r = eval(rab);
    % fox's path
    foxDistMat = [foxDistMat; reshape(z,[],1)'];
    totalFoxDist = totalFoxDist + norm(foxDistMat(end,:)-foxDistMat(end-1,:));
    s_f = initSpeedFox * exp(-mu_f * totalFoxDist);
    dist = max(norm(r-z),1e-6);
    dzdt = (s_f*(r-z)) / dist;
    end

    function dzdt = SWCornerODE(t, z, mu_f, initSpeedFox)
    % Fox-rabbit pursuit simulation ODE.
    sw = [200;-400];
    foxDistMat = [foxDistMat; reshape(z,[],1)'];
    totalFoxDist = totalFoxDist + norm(foxDistMat(end,:)-foxDistMat(end-1,:));
    s_f = initSpeedFox * exp(-mu_f * totalFoxDist);
    dist = max(norm(sw-z),1e-6);
    dzdt = (s_f*(sw-z)) / dist;
    end

    function dzdt = verticalODE(t, z, initSpeedFox)
    % Fox-rabbit pursuit simulation West wall ODE.
    foxDistMat = [foxDistMat; reshape(z,[],1)'];
    totalFoxDist = totalFoxDist + norm(foxDistMat(end,:)-foxDistMat(end-1,:));
    s_f = initSpeedFox * exp(-mu_f * totalFoxDist);
    dzdt = [0; s_f];
    end

    function [value, isterminal ,direction] = ...
        verticalEvents(t, z, rab, burrow)
    % Define simulation termination events for the west perimeter run
    value=ones(1,6); isterminal=ones(1,6); direction=ones(1,6);
    % rabbit's path
    x = t; r = eval(rab);
    % line segments
    nw = [200 0]; sw = [200 -400];
    x1=[r(1) z(1)]; y1=[r(2) z(2)]; x2=[sw(1) nw(1)]; y2=[sw(2) nw(2)];
    % Rabbit back to burrow
    value(1) = burrow(1) - r(1); isterminal(1) = 1; direction(1) = -1;
    % Fox can see the rabbit again
    if polyxpoly(x1,y1,x2,y2)
```

```
        value(6) = 1;
    else
        value(6) = 0;
    end
    isterminal(6) = 1; direction(6) = -1;
    end

    function [value, isterminal ,direction] = chaseEvents(t, z, rab, mindist, burrow)
    % Define simulation termination events
    x = t; r = eval(rab); dist = max(norm(r-z),1e-6);
    nw = [200 0]; sw = [200 -400]; se = [800 -400];
    x1=[r(1) z(1)]; y1=[r(2) z(2)]; x2=[sw(1) nw(1)]; y2=[sw(2) nw(2)];
    x3=[sw(1) se(1)]; y3=[sw(2) se(2)];
    % Fox catches rabbit
    value(1) = dist - mindist; isterminal(1) = 1; direction(1) = -1;
    % Rabbit back to burrow
    value(2) = r(1) - burrow(1); isterminal(2) = 1; direction(2) = 1;
    % View obscured by South Wall
    if polyxpoly(x1,y1,x3,y3)
        value(3) = 0;
    else
        value(3) = 1;
    end
    isterminal(3) = 1; direction(3) = -1;
    % View obscured by West Wall
    [xi,yi] = polyxpoly(x1,y1,x2,y2);
    if [xi,yi] & (yi < nw(2) - mindist)
        value(4) = 0;
    else
        value(4) = 1;
    end
    isterminal(4) = 1; direction(4) = -1;
    end

    function [value, isterminal ,direction] = ...
        SWCornerEvents(t, z, rab, burrow)
    % Define simulation termination events for SW corner run
    sw = [200 -400];
    x = t; r = eval(rab);
    % Rabbit back to burrow
    value(1) = r(1) - burrow(1); isterminal(1) = 1; direction(1) = 1;
    % Fox reaches corner
    value(5) = z(1) - sw(1); isterminal(5) = 1; direction(5) = -1;
    end
end
```

References

G. Cabrera and P. Negrón-Marrero. Pursuit problems: Generalizations and numerical simulations. 2011.

T.H. Cormen. *Introduction to algorithms*. The MIT press, 2001.

Desmond J. Higham and Nicholas J. Higham. *MATLAB Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, third edition, 2017. ISBN 978-1-61197-465-2.

Roger Stafford. How to check whether two lines intersect or not? https://stackoverflow.com/questions/27928373/how-to-check-whether-two-lines-intersect-or-not, 2015. Accessed: 06/04/2021.