Simulations of Robotic Pursuit Using Matlab and Simulink

by Joshua Lioi
Advisor:

UG 2006-11

Simulations of Robotic Pursuit Using Matlab and Simulink

Joshua Lioi

ISR REU Program

Summer 2006

Advisors:

Dr. P. S. Krishnaprasad

P. Vishwanada Reddy

Abstract

A pursuit curve is the path one creature takes while following another, and these can be used to model predator/prey chases, missiles homing in on a target, or even robot movement during a rendezvous. This paper will focus on the mathematical modeling and subsequently the simulation of these curves in Matlab's Simulink. Two other students who were working in the lab interacted directly with the robots, and in particular, they worked with experimental robotic pursuit. Using the Cricket system, which works like an indoor GPS, the robots in the Intelligent Servosystems Lab can communicate their locations to one another, and then run pursuit protocols. The Matlab programs, though idealized, are successfully able to simulate the realistic trials of robotic pursuit.

1. Introduction

As the name implies, the Intelligent Servosystems Laboratory is concerned with what could be called “smart robots.” A servosystem is another word for a mechanism that can manage itself through feedback. At the same time, the word “intelligent” refers to intelligent control, which is a set of protocols a robot uses to navigate, problem solve, or even learn in a manner of speaking. One major aspect of intelligent control from the recent past is that of robot formations, also called swarms. In order to make such robots work efficiently, they must have control laws in place that allow them to move together or follow one another without colliding[1]. This paper describes simulations of pursuit laws that robots could utilize in rendezvous or swarming situations. These could also be used to model more aggressive attack or shadowing strategies that unmanned military robots could use in the future. Although the simulations involve only two or three robots at a time, they could also be expanded to better model larger swarms.

1.1 Robots Modelled

The majority of the robots in the lab are wheeled, so they can only control their tangential speed and angular velocity. However, a robot has three variable values, or generalized coordinates of the system, which describe its location: x, y, and orientation angle. This implies that the wheeled robots are nonholonomic, meaning that there are fewer controlled inputs than generalized coordinates[2]. These wheeled robots can use various systems to navigate and determine their location; systems such as ultrasound sensors, microphones for sound localization, and onboard GPS are a few examples.

1.2 Cricket System

A more recently implemented system used in the lab is the Cricket system, which is the most closely related to my project. Each robot has two Cricket beacons on it that send out ultrasound and RF pulse signals at regular intervals. Ten equally spaced Cricket listeners on the ceiling map out a coordinate system and receive the RF pulse and the ultrasound signals from the robot's beacons. The listener units then bounce the signal packet back, along with their ID number. When a robot acquires the return signal packet from a listener, it can calculate the horizontal distance between the two beacons based on the time difference between the RF pulse and the slower ultrasound signal. By using signals from multiple listeners each second, a robot can easily use triangulation to approximate its location in the xy coordinates and its orientation in the lab[3]. The pursuit simulations are based on the wheeled robots in the lab that utilize the Cricket system.

1.3 Classical Pursuit

To understand more advanced methods of pursuit, one must first learn what pursuit is and how it started. What is now called "classical pursuit" was first studied by Pierre Bouger in 1732. Classical pursuit involves two moving particles: one of which (pursued or prey) moves along a known curve, while another particle (pursuer or predator) follows the first. The pursuer always has an instantaneous velocity pointed directly toward the pursued, and the path that the pursuer traces out is called a pursuit curve. Since then much has been analyzed about the pursuit curve, including the logarithmic shape that the predator traces out and the total time before a capture occurs[4]. A more physical example of a pursuit curve would be the trajectory of a torpedo or missile, when the projectile tracks a target by moving towards it at all times. However, it can be concluded that this might not be the most efficient trajectory, and a strategy that leads a target would be more likely to succeed. In fact, with classical pursuit a predator can only overtake the prey if its velocity is more than that of the prey[5]. The greatest contribution that classical pursuit can make to swarms is the "bugs problem," which is directly related to rendezvous.

1.4 Bugs Problem

The bugs problem, also known as the mice or dogs problem, was first considered in 1877 by Eduard Lucas. He analyzed what trajectories would occur if three dogs started out at the points of an equilateral triangle, and then each dog chased the next one counterclockwise to itself. Henri Brocard provided his analysis a few years later and determined that the dogs would trace out the logarithmic spiral of classical pursuit, and that all dogs/mice would meet simultaneously at the Brocard point of the triangle[6]. Analysis was later expanded to regular n-polygons and even irregular n-polygons, instead of just the equilateral triangle case. Further examination discovered that the simultaneous meeting always occurs in non-collinear triangles, but not for other polygons where premature head-on collisions can occur[7]. The final important note that concerns us is the idea of relative speeds for irregular triangles. Figure 1 shows an example of an irregular triangle with bugs at each point chasing counterclockwise. Klamkin and

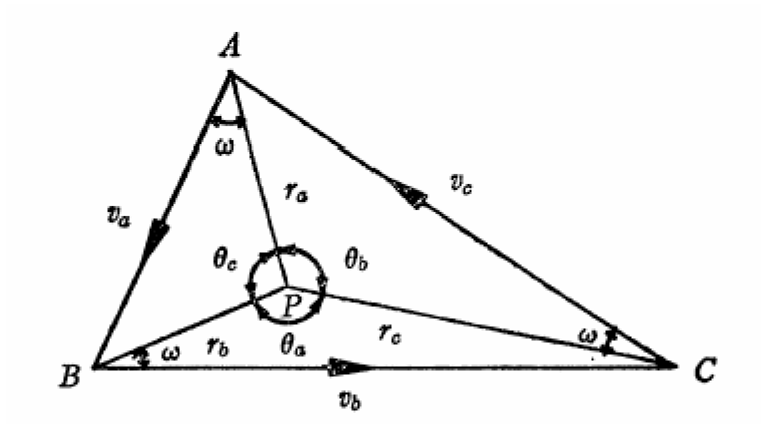


Figure 1: Bugs problem triangle with counterclockwise pursuit[8]

Newman derive speeds based on ratios of side lengths that will allow the three bugs to converge simultaneously at the Brocard point, regardless of the shape of the initial triangle, excluding all collinear cases. The speeds for each bug would be

$$v_a = n\frac{b}{a} \quad v_b = n\frac{c}{b} \quad v_c = n\frac{a}{c}$$

where a, b, and c are the side lengths of the triangle as shown in Figure 1, and n is any scalar constant[8]. Although useful later, the bugs problem is still referring to classical pursuit and does not account for the nonholonomicity of the robots, and so another type of pursuit is necessary.

1.5 Cyclic Pursuit

Marshall, Broucke, and Francis consider a slightly different bugs problem by adding a constraint to each bug so that they can only move like a unicycle, which correlates directly to our nonholonomic robots. The unicycles could not perform classical pursuit, as they require time to orient themselves so that they are facing their prey. The traditional feedback law for modeling such a nonholonomic vehicle is by using simultaneous differential equations for each of the generalized coordinates, as follows where x and y determine position, θ describes the orientation,

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} \cos(\theta) & 0 \\ \sin(\theta) & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} u \\ \omega \end{bmatrix}$$

while u and ω are the control inputs of tangential speed and angular velocity, respectively. However, this equation simply describes the motion of one vehicle and does not account for the pursuit laws needed. To remedy this, Marshall, Broucke, and Francis define alternative coordinates that determine the location of vehicle *i* based on its position relative to its prey (vehicle *i*+1). As can be seen in Figure 2, the new generalized variables are α, β, and r. The

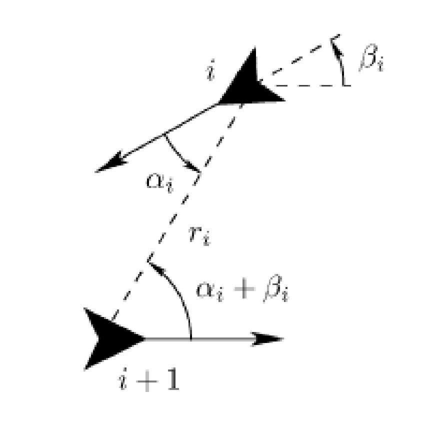


Figure 2: Alternative Coordinates for cyclic pursuit strategy[6]

variable r_i is simply the Euclidean distance between the i and i+1 vehicles, and α_i is the difference in radians between the pursuer's current heading and the heading it would have in a classical pursuit situation. Unfortunately, β_i is defined, somewhat awkwardly, as the angle between the prey's current heading and the reverse vector of the predator's heading. Marshall, Broucke, and Francis also derived the equations analogous to the traditional feedback law which use the alternative coordinates, and thus the cyclic pursuit equations are defined as

$$\dot{r}_i = v_i \cos(\alpha_i) - v_{i+1}\cos(\alpha_i + \beta_i)$$

$$\dot{\alpha}_i = \frac{1}{r}\left[v_i \sin(\alpha_i) + v_{i+1}\sin(\alpha_i + \beta_i)\right] - \omega_i$$

$$\dot{\beta}_i = \omega_i - \omega_{i+1}$$

In order to automate the control inputs, they suggest making each vehicle's angular velocity proportional to the current value of α ($\omega_i = k*\alpha_i$ where k is a constant). This means that a vehicle will turn more sharply when its heading is significantly different from the direction towards its prey[6]. This system of feedback will henceforth be referred to simply as "cyclic pursuit." Cyclic pursuit is the direct nonholonomic analogy to classical pursuit, and it is employed in many of the robot simulations for pursuit and rendezvous. However, a more sophisticated method of pursuit is also considered, which is called motion camouflage.

1.6 Motion Camouflage

Motion Camouflage is a pursuit strategy first observed in insects such as dragonflies and hoverflies, and later in the attack patterns of bats. This elaborate technique allows animal (called the shadower) to trick another creature (called the shadowee) into thinking it remains still, even while both are moving continuously. The key to this pursuit lies in the predator/shadower deceiving the prey's optic flow, which is the ability to discern movement with the eyes[9]. Consider the example where you are driving and you observe a car parked on the side of the road; as you approach it, you perceive the car moving further away from the center of your vision until you travel past it, and this observation is done by optic flow. Now consider driving the same speed as a car next to you; the other car appears to be stationary even though both cars are moving, which is the idea behind motion camouflage.

There are two varieties of motion camouflage: one is for the shadower to move such that it appears stationary relative to some object in the background, while the other is to appear stationary relative to a point at infinity, meaning that the shadower "always appears to be at the same bearing"[10]. This second style of motion camouflage is the one considered in the simulations of the robots, and it is the same method that bats use to quickly close in on prey when hunting. The simulation of the robots is based primarily on Dr. Krishnaprasad's model, and the same constraints are used. The predator moves with tangential speed always equal to one, while the prey has a speed between zero and one, and any reasonable angular velocity. Then the angular velocity of the predator is determined to be

$$\omega_P = \mu\left[W \cdot \left(\frac{\vec{r}}{|\vec{r}|}\right)^{\perp}\right] \quad with \quad W = \dot{\vec{r}}_p - \left(\frac{\vec{r}}{|\vec{r}|} \cdot \dot{\vec{r}}\right)\frac{\vec{r}}{|\vec{r}|}$$

where r_p is the vector from the origin to the predator, r is the vector from the prey to the predator (so that |r| is the distance between the two), and μ is a constant called the gain. Also, the perpendicular symbol ($\perp$) here means to rotate a vector counterclockwise by π/2 radians. A larger value of the gain variable allows a pursuer to achieve motion camouflage, even when it must estimate the prey's angular velocity[10]. Implementing these equations allows simulations to be created for robot movement of this kind, and with sufficient onboard sensors a robot could actually attain a motion camouflage pursuit strategy.

2. Modeling Methods

Although accurate simulations of realistic robot pursuit were the initial goal of this project, the ideal cases were primarily the ones modeled. The robots will always have some error in the measurement of their positions, but with sufficient onboard sensors and approximation algorithms, the ideal cases could be nearly attainable. In addition, these ideal simulations could act as a control variable when compared to the experimental pursuit tests run by two other students with whom I collaborated, Hosam Haggag and Golbarg Mehraei.

2.1 Matlab and Simulink

Matlab is generally considered to be a great tool in mathematical computation, and it served the simulation purposes rather well. In conjunction with Matlab, I also worked with Simulink, an extension of Matlab which is capable of modeling and analyzing dynamic systems with relative ease. Although I had never used Simulink before, it was fairly easy to learn the basics through various tutorials. Unfortunately, Simulink is lacking in its graphing capabilities, so I later incorporated Matlab functions with commands that called my Simulink models and did the graphing. This became an excellent and rather user-friendly approach to the simulations. Following are descriptions of each of the models I created and how they evolved as I modified and improved them.

2.2 Classical Pursuit

Classical pursuit simulations seemed like the easiest place to begin, as it was also the first pursuit strategy I learned. These were primarily just practice for the more advanced versions of pursuit. I created two versions of the Simulink model; practice1_v2.mdl had the prey particle moving along a line, which was designated by the constants in the model, while practice2.mdl instead had the prey move in circular or elliptical patterns centered at zero (See Appendix A).

By changing the constant or gain values, the prey's movement changed. The "x & y solver" subsystem determined the predator's movement using fairly simple equations that solved based on parametric expressions of the x and y movements of the prey, called p & q respectively[11]. In order to work with these models more easily, a simple Matlab function, called pursuitgraph.m, was created that could alter the initial xy position of the predator and set the value of the relative speed, which is the ratio of predator speed to prey, with a value of one corresponding to equal speeds. These models were simplistic, but they provided an excellent start for the more complicated models.

2.3 Robot Movement

In order to try and implement robotic pursuit, basic robot movements had to first be modeled. Using the traditional nonholonomic feedback laws, a simple Simulink model (robotmove.mdl) was built that could graph the movement of a robot based on speed and angular velocity values[6]. However, this model was cumbersome and difficult to utilize, so I refined it by creating a Matlab function called interactrobot.m that could call the robotmove.mdl model (See Appendix B). The interactrobot.m code included a loop which allowed for changes of the robot's speed and angular velocity each second (previous values could be repeated to have the robot continue along the same curve), and these values could be entered by the user easily. As an experiment to approach robotic pursuit, the robotmove2.mdl Simulink model was produced, in conjunction with a separate Matlab function called interactrobot2 (See Appendix C). The robotmove2.mdl model was simply a juxtaposition of the robotmove.mdl and practice2.mdl models, in that the "x & y solver" block was attached to robotmove.mdl. Subsequently, this allowed for user-input speed and angular velocities of the prey robot and derived the movement of a classical pursuit predator following the robot. This had very little real world significance, but there was a predator/prey pursuit with at least the prey moving as one of the robots would, which was a step closer to simulating the robots' motions.

2.4 Cyclic Pursuit

To attain a better pursuit representation, the initial thought was to simply add constraints to the classical solver that would prevent the predator from turning too sharply. However, this idea was quickly dismissed as impossible after a few modification attempts. Shortly thereafter, I found the paper on cyclic pursuit by Marshall, Broucke, and Francis and made the cyclerobotmove.mdl model using their concepts (See Appendix D). Retaining the same nonholonomic control for the prey (seen in the Prey Solver subsystem), I implemented a different solver for the predator that used the cyclic equations. Again, the predator was given a relative speed, which is accounted for in the gain block titled "rel speed (c)." As previously discussed, there was also the quantity k to account for in the angular velocity ratio of $\omega=k\alpha$, which is done in the "rel ang speed (k)" block[6]. As before, a corresponding Matlab function called cyclicrobot.m, was also constructed, including the loop where the prey's controls can be changed by the user. The only substantial change this code required was transformations for the relative coordinates of α, β, and r from x, y, and θ. Marshall, Broucke, and Francis used only random initial positions, but positioning of the robots in the simulation was important, so I had to discern these transformations and use them to convert both ways (See lines 18-21 and 59-60 of cyclicrobot.m). This successfully simulated cyclic pursuit of a pursuer robot following a user-controlled pursuee.

The next step taken was a modification of these models to create robotpursuit.mdl and cyclicrobotpursuit.m, which better approximated the Robot Pursuit experiments done by Hosam Haggag and Golbarg Mehraei (See Appendix E). These tests were to have a robot go to a target location, and constants were used for speed and angular velocity of the predator, so this change had to be made in the models. The pursued was made to remain still in this code and the predator had to follow the constraints set in the Robot Pursuit tests. Also, stopping criteria were introduced as the user-inputs for the prey were removed. The cyclicrobotpursuit.m/ robotpursuit.mdl modeling was very successful in providing an ideal case comparison for Haggag and Mehraei's Robot Pursuit trials.

2.5 Mice/Bugs Problem

Having simulated two robots fairly well, I hoped to create an alternate version of the bugs problem, in which each bug was actually a nonholonomic wheeled robot. For the sake of comparison, I had to create code for the classical version of the bugs problem. Initially, I began trying to alter robotmove2.mdl and add another predator to follow the first, but the prey was still a robot, so I had to find another route. Starting over with just the "x & y solver" block, I made a concise Simulink model called micesolver.mdl which worked perfectly for the mice/bugs problem (See Appendix F). I then worked with a Matlab function called classicmice.m to try and implement a system for calling the micesolver model. Unfortunately, classicmice.m was insufficient and it went through various iterations until it was perfected in classicmice_v4.m (See Appendix F).

Initially I set an amount of time for the simulation to run, but later improved it with a loop similar to the interactrobot.m idea. This loop was different, in that no user-inputs were needed and the user could simply press enter to continue with the simulation or type "quit" when they were finished. In addition, classicmice.m only worked for the equilateral triangle and the relative speeds discovered later had to be included[8]. In order to verify that the three mice met at the Brocard point, I implemented an algorithm to plot this point for any given triangle[12]. For ease, I based the algebra on triangles with one side along the positive x-axis and beginning at the origin; this way all the angles were less cumbersome to calculate. After this was accomplished, classicmice_v4.m depicted the bugs problem flawlessly.

2.6 Robot Mice/Bugs Problem

While working on the classical version of the bugs problem, I was also undertaking the task of creating a parallel simulation where each bug moved as the robots do. The first attempt was to simply add another predator solver to the cyclerobotmove.mdl model, but this proved impossible as the second predator's location was lost because of the relative coordinates involved[6]. As before, the cyclerobotmove models went through various versions until it was perfected with cyclerobotmove6.mdl (See Appendix G). An approximation of the second predator's movement was first implemented but it was found to be insufficient. Eventually, the prey was made to imitate the other robots, which allowed the equilateral triangle case to be simulated, but any other triangle was impossible. Finally, by solving for the prey's movement through first the cyclic pursuit predator solver and then the traditional robot movement solver, the prey acted like a predator and the positions were not lost due to too many relative coordinates. For each attempt at this simulation, there were also successive versions of robotmice.m until it too was finalized with robotmice_v6.m (See Appendix G). The code plots the Brocard point, just as the classicmice_v4.m does, and the relative coordinate transformations of cyclicrobot.m are also present (lines 34-49 and 87-92). These perfected versions ultimately provided a simulation analogous to the classical case, and the two styles of the bugs problem could be compared.

2.7 Motion Camouflage

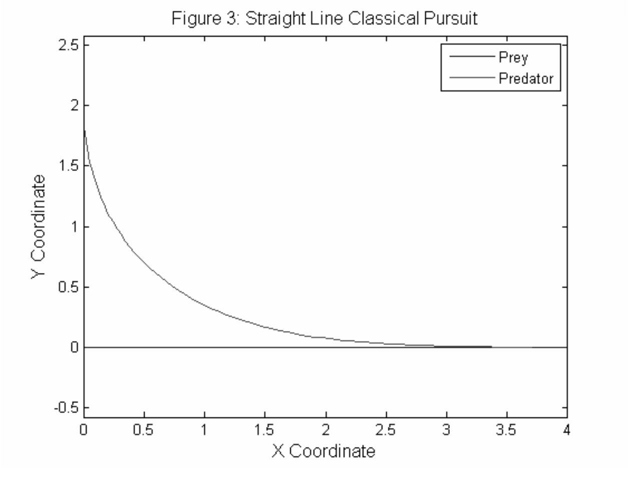
After thoroughly studying Dr. Justh and Dr. Krishnaprasad's equations governing motion camouflage, I was ready to implement it for the robots. For this simulation, I returned to a situation involving only two robots, similar to the cyclerobotmove.mdl model. The traditional feedback laws were again implemented for the shadowee robot and the shadower also used them, except that the angular velocity had to be derived via the equations. After learning how to represent vectors in Simulink, I was able to assemble the motioncamouflage model (See Appendix H). The model is a fairly direct execution of the control laws, except that I had to devise a way of rotating the vector by pi/2 radians, which involved using the atan2 Matlab function and then finding the components after subtraction[10]. In order to analyze this model more easily, a corresponding Matlab function was created. MotionCamoRobot.m works very similarly to cyclicrobot.m except that the predator always has unit speed, and the gain variable had to be set in the model. The same user-input loop returned and allowed for better control of the predator. The final touch was to periodically plot a line connecting the two robots (line57); the predator works to make these green lines parallel as he moves, since it wants to be at the same bearing relative to the shadowee at all times[10]. With this model created, a true simulation of wheeled robots performing a motion camouflage strategy could be analyzed.

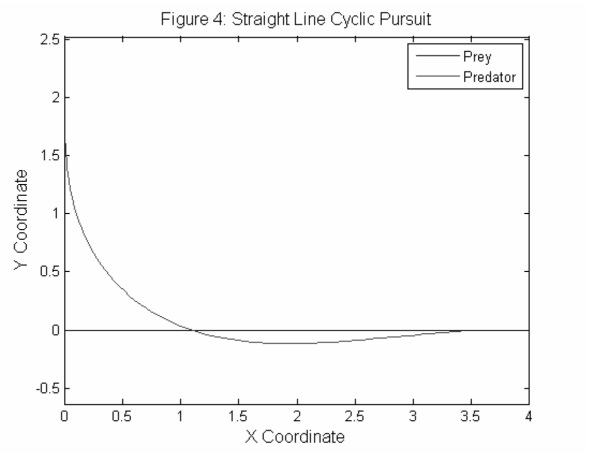
3. Discussion

Even though the goal was to simulate realistic robot movement, I decided that I also wanted to compare and contrast some of the types of pursuit. For example, classical pursuit is entirely implausible in practice, but it serves as an excellent comparison, especially for cyclic pursuit. Although a robot can't orient itself to be facing the prey at every given instant due to movement constraints, in cyclic pursuit it attempts to do just that. Subsequently, comparisons yielded some interesting results, especially in the mice problem. Also, based on the simulations, motion camouflage does not seem to work well on its own when applied to the robots. Generally insects and bats seem to use this unique strategy in nature, but a robot can simply not travel with the same maneuverability. However, comparisons with this strategy are still interesting and some conclusion can be made that would allow motion camouflage to be useful for robots in conjunction with other strategies.

<u>*3.1 Straight Line Pursuit*</u>

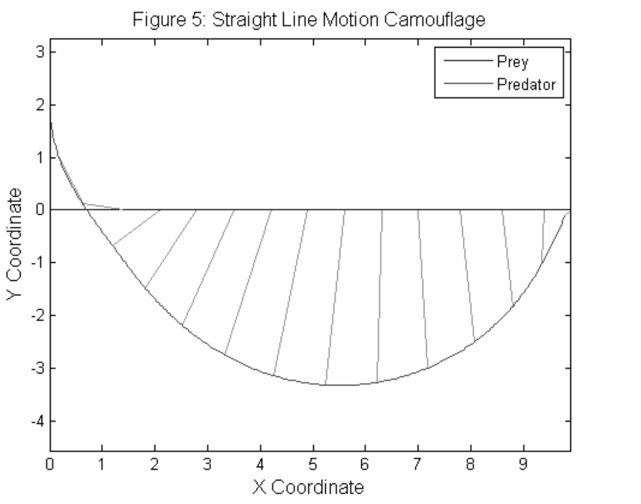
The classical case of the prey moving on a straight line was thoroughly analyzed by Arthur Bernhart, and it serves as an excellent comparison for these simulations[4]. As a classical pursuit predator automatically orients itself to be facing toward the prey from the beginning, I decided to make the robot versions face the same way for the purpose of the first comparison. Below are figures 3 and 4 which have the prey moving in a straight line along the x-axis,



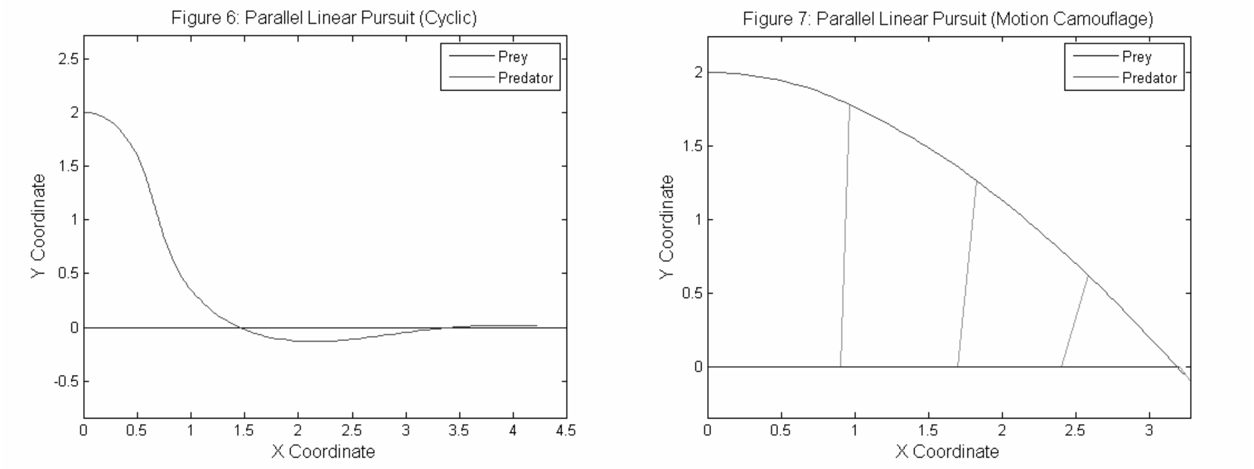


beginning at the origin, while the predator begins at the point (0,2) and is turned toward the negative y direction (facing the prey). As can be seen, the cyclic version is very similar to the classical, except that the predator overshoots the prey's trajectory because of the turning limitations of the robot. Also, the prey has unit speed for the whole simulation, while a value of

1.2 is set for the predator's relative speed; the angular speed gain (k) is set to a value of two. A similar test of the motion camouflage pursuit strategy can be seen below in figure 5, although it is slightly different. The velocities are different for this test, as the prey has a speed of 0.8 and



the predator has unit speed. The green lines connect the current positions of the robots at regular intervals, and since the predator wants to keep the same bearing in the eyes of the prey, these lines are close to parallel. However, one can see that the robot overtakes the prey more slowly under this strategy, which is evidence for its difficulty in using motion camouflage. A better comparison of cyclic pursuit versus motion camouflage is shown in figures 6 and 7. The starting positions are the same as the previous examples, except that the predator is facing the positive x-



axis, meaning that the two robots are initially moving parallel to one another. In this particular situation, the robot following motion camouflage actually does reach the prey first, so motion camouflage can be the time optimal strategy for constrained systems when used appropriately.

3.2 Circular Pursuit

Another traditional arrangement for pursuit is when the prey is moving along a circular path and the predator begins in the center of the circle, facing the prey. Below are graphs of the simulations for both the classical and cyclic versions of this configuration (Figures 8 and 9). In

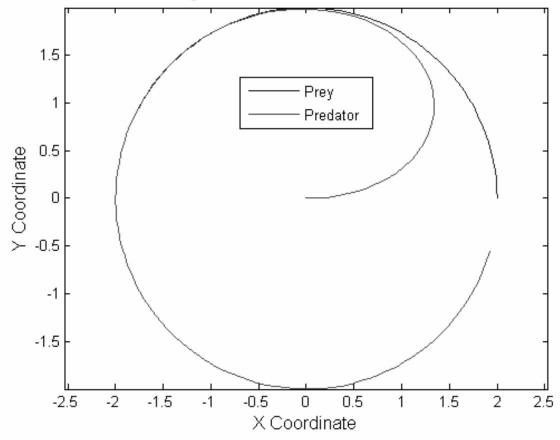


Figure 8: Classical Pursuit - Circle

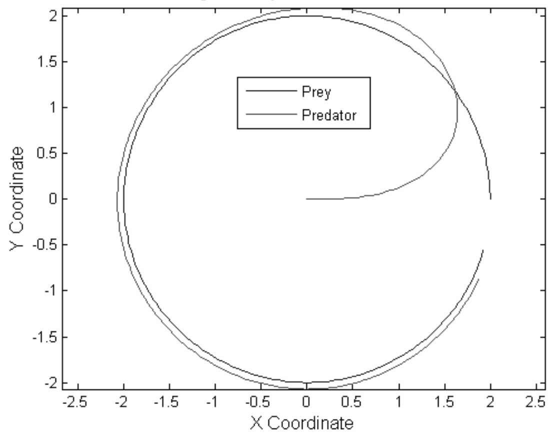
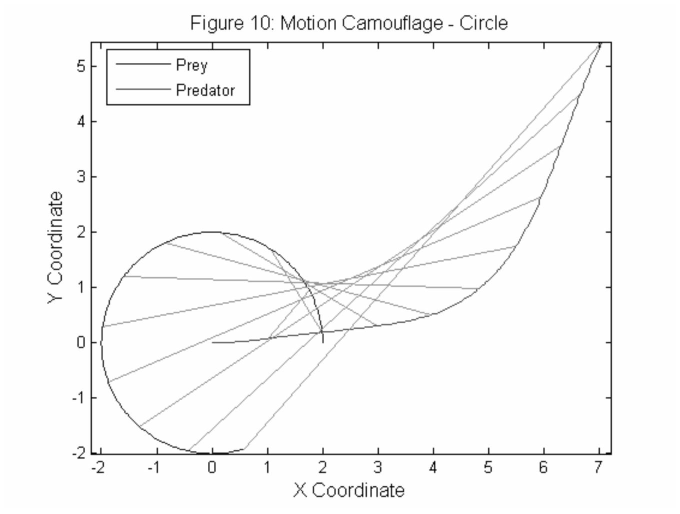


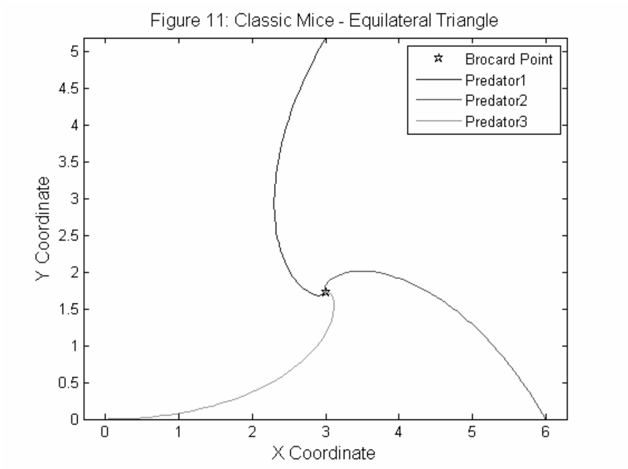
Figure 9: Cyclic Pursuit - Circle

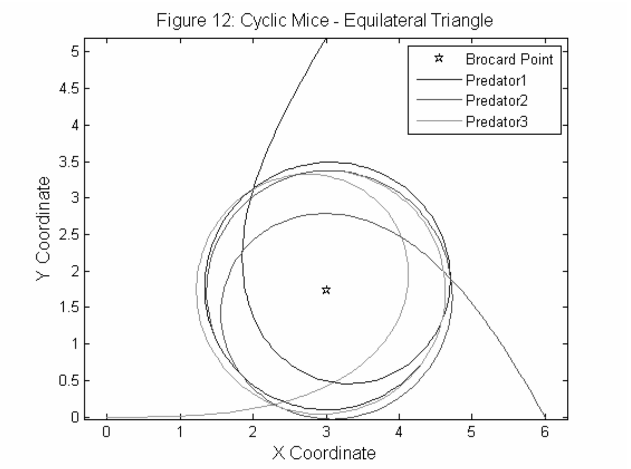
both cases, the prey starts at the point (2,0) and is oriented toward the positive y direction, while the predator is facing toward the prey (positive x direction) and is initially located at the origin. As before, the predator overshoots the prey's trajectory and compensates to correct itself. Also, the predator and prey swiftly become collinear in the classical trial while the cyclic version never quite accomplishes that. After seeing the linear examples, this is largely what would be expected from these pursuit laws. The motion camouflaging robot does not seem to work very well in the circular case, as can be seen in figure 10 on the next page. In its attempts to remain at the same bearing from the prey, the predator ends up moving very far away from its target. Allowing the simulation to run further only increased the distance between them, though the predator always tried to get back to the point where the green lines were parallel. This failing of motion camouflage is due either to a problem in the model that must be improved, or the inability of these robots to utilize this strategy.



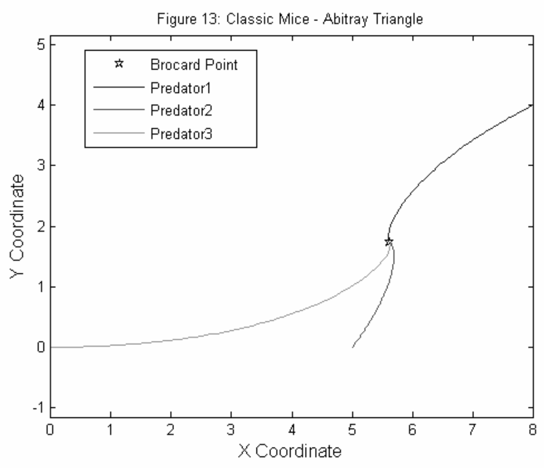
3.3 Mice/Bugs Problem

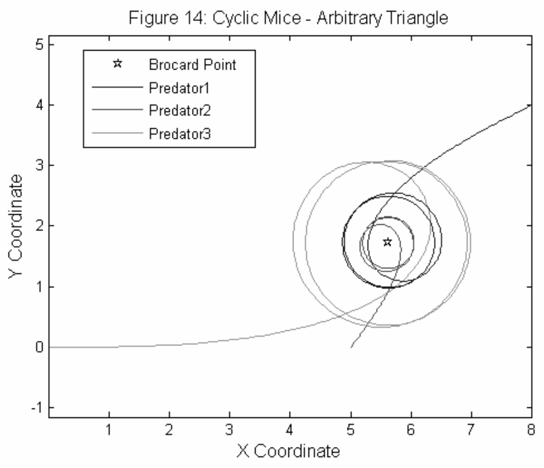
A much more interesting network of predators to consider is the mice problem, in which every entity involved is both predator and prey. Accordingly, this makes an excellent framework for multi-robot rendezvous experiments. Historically, the mice problem was first considered in the case for the equilateral triangle, and my analysis began there as well[6]. Included below are figures 11 and 12, which portray the equilateral triangle case for the mice problem in classical





and cyclic pursuit, respectively. All of the “mice” involved have the same speed and they begin at the points of an equilateral triangle with a side length of six. As is expected, the classical pursuit version has all three mice converge simultaneously at the Brocard Point. Although the cyclic version is different, there are some striking similarities to the classical; while the robots do not meet at the Brocard point, they converge to a circular path centered at this point. Having seen this, I was curious if it remained true for other arbitrary triangles, and after finding the relative speeds required I immediately analyzed other cases[8]. Figures 13 and 14 depict the mice problem for classical and cyclic pursuit using an arbitrary triangle with points located at (0,0), (5,0), and (8,4). Again, the classical instance converges with a simultaneous meeting at the

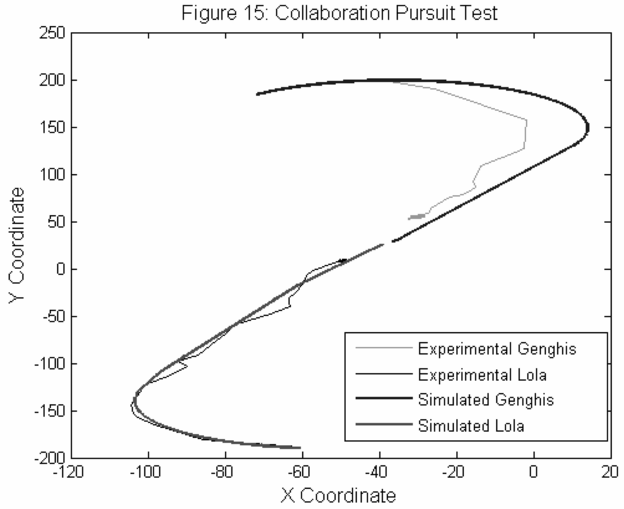




Brocard point, while the cyclic version looks somewhat different. The three robots converge to circles, but each is of different radius; however, intuition states that this difference is due entirely to the relative speeds, which are unequal. Thus, the mice problem is represented and the Brocard point remains central to convergence, even in the cyclic case. Unfortunately, due to time constraints, I was unable to implement a mice problem model that utilized motion camouflage.

3.4 Collaboration

Though I was unable to complete the mice problem comparison, I was able to collaborate with Haggag and Mehraei on their tests. They were working on pursuit problems as well, but in the experimental sense instead of in simulation. This allowed us to compare our results successfully, but required a fair amount of adjustments to the preexisting cyclic pursuit model. After the alterations were complete, we produced results like those in figure 15. The two robots



involved are called Genghis and Lola, and they were made to pursue one another under the same protocol from various starting positions around the room. Then, using their starting positions, I simulated the same situation and we superimposed the graphs to compare. According to this, the simulations are an excellent approximation for the experimental results. The only delineation is that Genghis turned more sharply than expected, but this was due to its battery being low, meaning that its forward speed was not as high as it should have been. Overall, the simulations and experiments coincided and further study could be done with more robots, or other pursuit strategies.

4. Conclusion

By studying various types of pursuit, I was able to implement a variety of simulation models for theoretical classical pursuit and more realistic variants involving the robots. Comparisons could then be made between the versions and even between the different strategies of cyclic pursuit and motion camouflage for the robots. There was also a successful collaboration that simulated a genuine rendezvous test with the robots. Given more time, I would like to further study the motion camouflage model to determine if there is an error in it or if it is simply difficult for these mobile robots to implement it perfectly. There are also minor modifications that could make some of the simulations run more smoothly. As an example, having a time input for the cyclic pursuit models to dictate an amount of time for the given controls to be performed would make things less tedious for the user. Also, stopping criteria could be changed to mimic the cyclicrobotpursuit.m code by automatically stopping when the robots reach a certain range from one another. This project could be the basis for further study by modeling even more pursuit strategies, and ultimately these pursuit strategies could be implemented on the robots for comparison. Another improvement would be to make the simulations less ideal by adding “noise” to simulate the statistical inaccuracy of the robot position measurements. These accomplishments could be used to further the study of rendezvous for swarms and also implement pursuit strategies for military application. By combined cyclic pursuit and motion camouflage, a robot or unmanned aerial vehicle could spy and even attack more effectively. Even missiles and other targeting projectiles could benefit from using motion camouflage to make their trajectories more optimal.

References

1) Institute for Systems Research University of Maryland. (2002). *A Simple Control Law for UAV Formation Flying* (TR 2002-38). College Park: ISR.

2) Holonomic. Retrieved August 3, 2006, from Wikipedia: The Free Encyclopedia Web site: http://en.wikipedia.org/wiki/Holonomic

3) Kushleyev, A, Young, T. (2005) "Cricket as a Positioning System for Control Applications". Merit Program Summer Research Paper

4) Pursuit Curve. Retrieved June 20, 2006, from Wolfram MathWorld Web site: http://mathworld.wolfram.com/PursuitCurve.html

5) Mungan, C. E. (2005). "A Classic Chase Problem Solved from a Physics Perspective". *Eur. J. Phys.* **26**, 985-990.

6) Marshall, J. A., Broucke, M. E., Francis, B. A. (2004). "Formations of Vehicles in Cyclic Pursuit". *IEEE Trans. Automat. Contr.* **49**, 1963-1974.

7) Behroozi, F., Gagnon, R. (1979). "Cyclic Pursuit in a Plane". *J. Math. Phys.* **20**, 2212-2216.

8) Klamkin, M. S., Newman, D. J. (1971). "Cyclic Pursuit or 'The Three Bugs Problems'". *Amer. Math. Monthly* **78**, 631-639.

9) Mizutani A., Chahl, J. S., Srinivasan, M. V. (2003). "Motion Camouflage in Dragonflies". *Nature* **423**, 604.

10) Justh, E. W., Krishnaprasad, P. S. (2006). "Steering Laws for Motion Camouflage". *Proc. R. Soc. A* FirstCite Early Online Publishing.

11) Kunda, J. Myers, M. (2003). Pursuit Curves. Retrieved June 20, 2006, Web site: http://oxygen.fvcc.edu/~dhicketh/Math222/spring03projects/MarkJohn/newpursiut.htm

12) Brocard Points. Retrieved August 3, 2006, from Wolfram MathWorld Web site: http://mathworld.wolfram.com/BrocardPoints.html

Appendix A

pursuitgraph.m

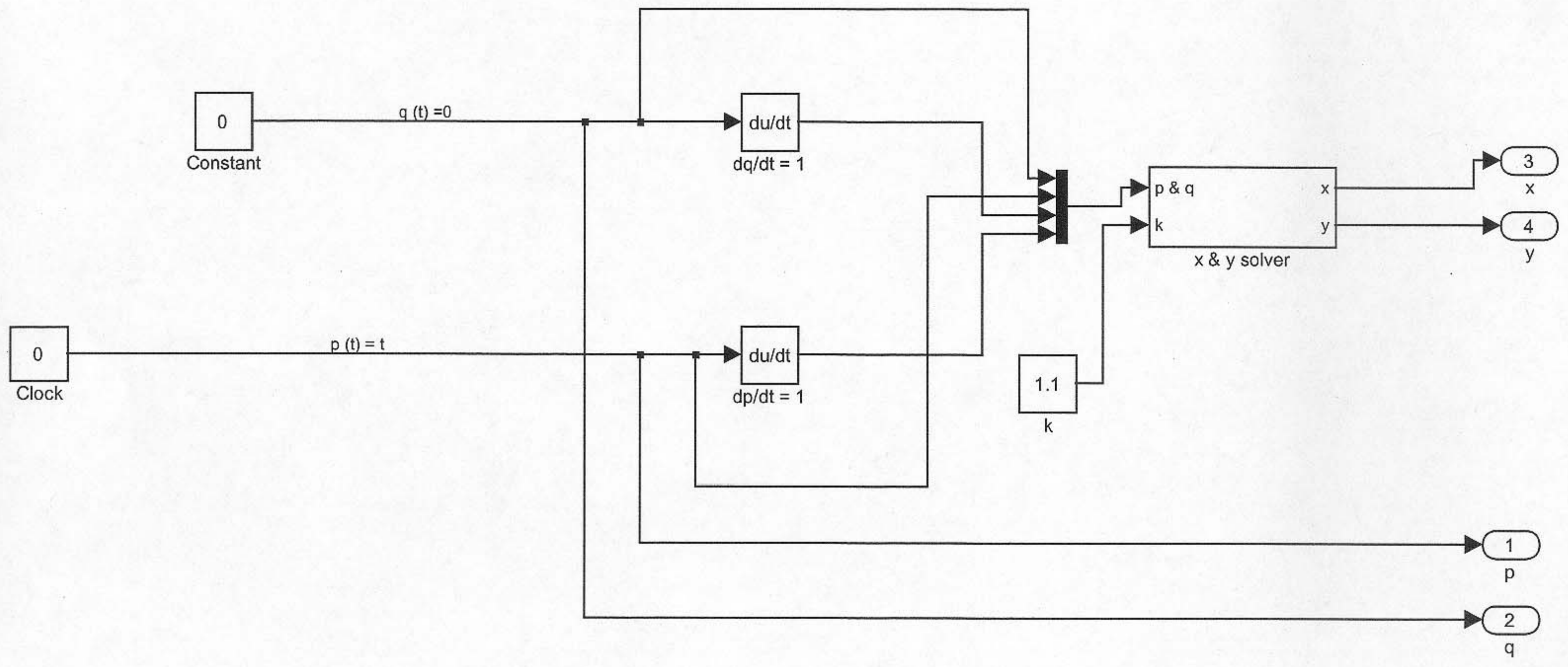
practice1_v2.mdl

practice2.mdl

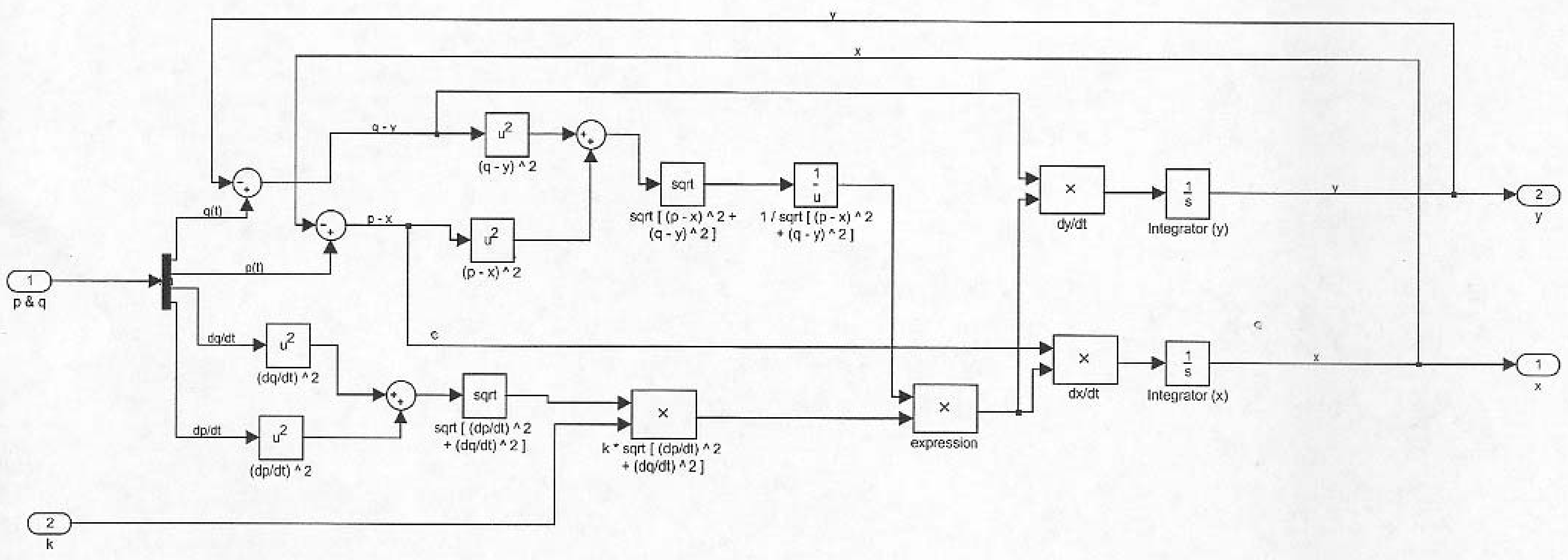
```
function pursuitgraph(StopTime,relspeed,initialx,initialy)

% This function will call the classical linear pursuit simulink model
% (practice1_v2, practice2) and graph the encounter on an xy
% plane where the pursued is in red and the pursuer is in blue
%
%  Inputs:  StopTime - end of simulation (start is 0.0)
%           value - relative speed of pursuer versus pursued
%           initialx - initial x position of pursuer
%           initialy - initial y position of pursuer
%
%

set_param('practice2','StopTime',num2str(StopTime));
set_param('practice2/k','value',num2str(relspeed));
set_param('practice2/x & y solver/Integrator (x)','initial',num2str(initialx));
set_param('practice2/x & y solver/Integrator (y)','initial',num2str(initialy));
t = 0:0.05:StopTime;
[t,x,y] = sim('practice2',t);
plot(y(:,1),y(:,2),'bl',y(:,3),y(:,4),'r');
axis equal;
legend('Prey','Predator');

end
```

practice1_v2

Models classical pursuit with the prey moving linearly.
This model is called by pursuitgraph.m

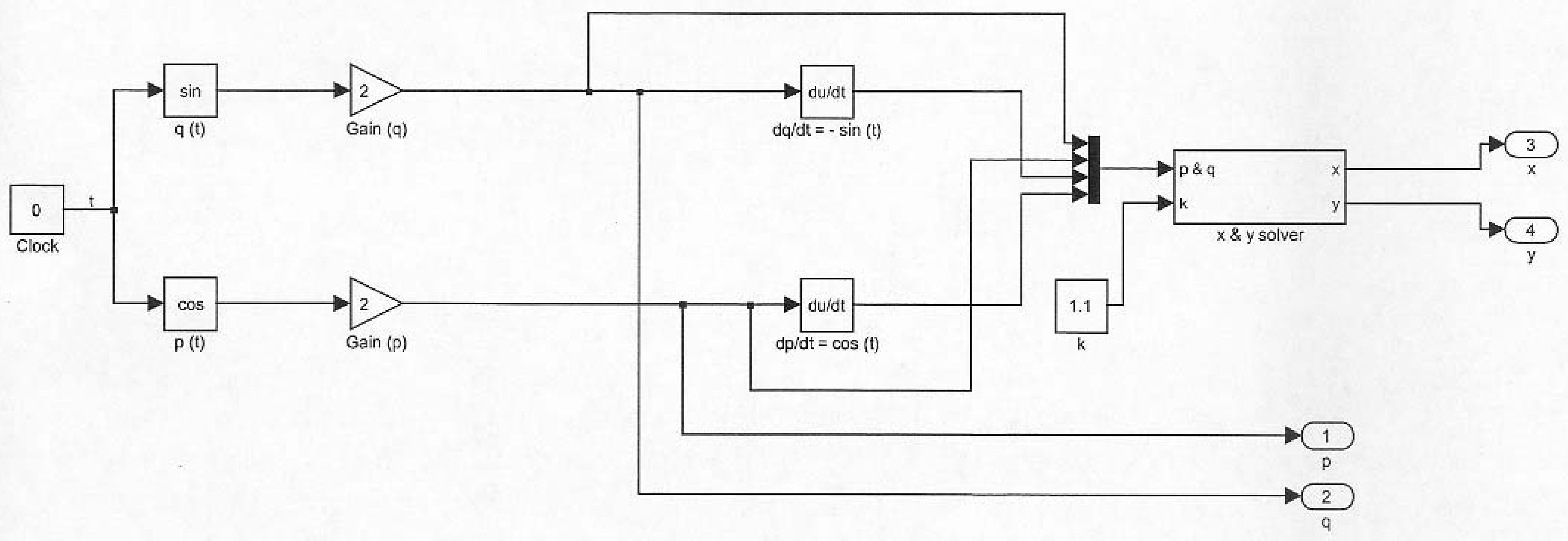


practice1_v2 / x & y solver



practice2

Simulates classical pursuit with the prey moving
in a circle or ellipse. This model is called by pursuitgraph.m



Appendix B

interactrobot.m

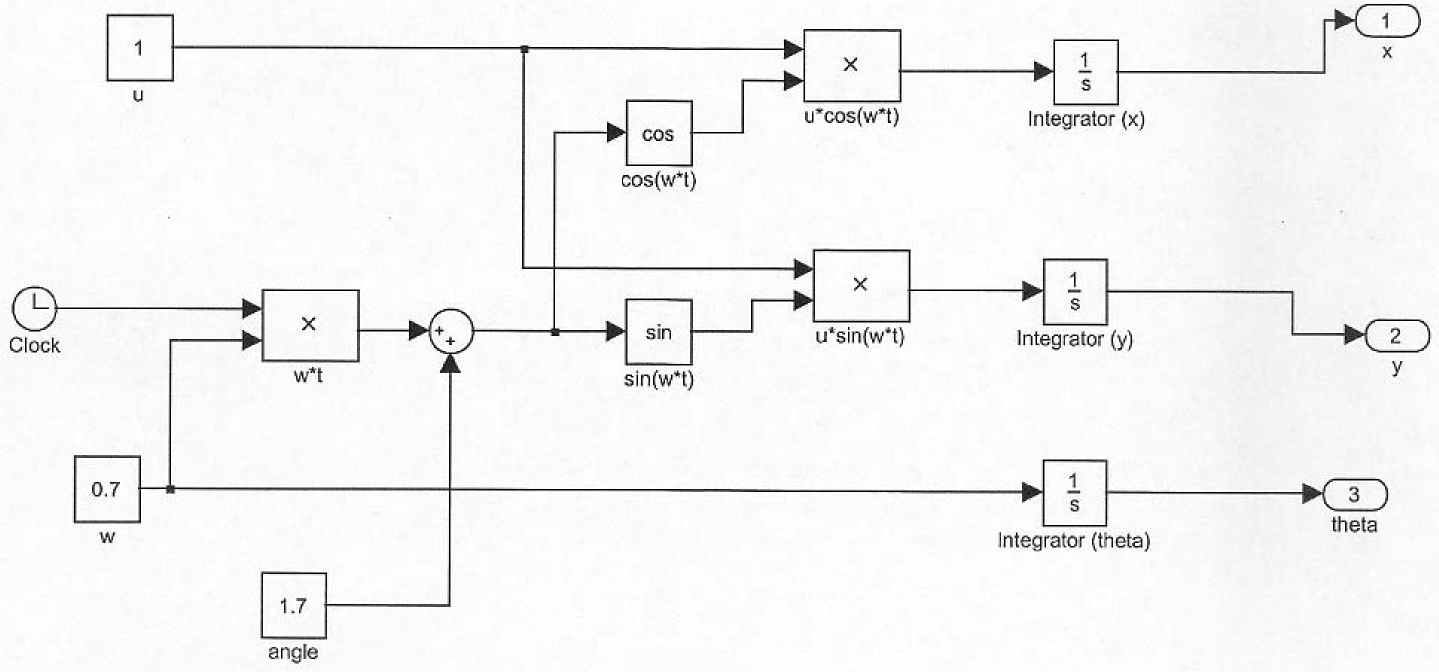
robotmove.mdl

```
function interactrobot(ini_x,ini_y,ini_theta)

% This function calls the robotmove simulink model to chart how the robot
% will move over time. In addition this function will allow user changes to
% the values of omega and u during the simulation.
%
%  Inputs:  ini_x - initial x position of robot
%           ini_y - initial y position of robot
%           ini_theta - initial angle of robot
%

start = [ini_x,ini_y,ini_theta];
check = true;
begintime = 0;
angle = 0;


%enter starting robot parameters
disp('Enter the word "quit" at prompt to end simulation');
u = input('Enter robot''s speed\n','s');
if(strcmpi(u,'quit') == true)
    check = false;
end %if
if(check == true)
    w = input('Enter robot''s angular velocity\n','s');
    if(strcmpi(w,'quit') == true)
        check = false;
    end %if
    hold on;
end %if


%loop for movement
while(check == true)
    t = begintime:0.1:(begintime+1);
    set_param('robotmove','StartTime', num2str(begintime));
    set_param('robotmove','StopTime', num2str(begintime+1));
    set_param('robotmove/u','value',u);
    set_param('robotmove/w','value',w);
    set_param('robotmove/angle','value',num2str(angle));
    set_param('robotmove/Integrator (x)','initial',num2str(start(1)));
    set_param('robotmove/Integrator (y)','initial',num2str(start(2)));
    set_param('robotmove/Integrator (theta)','initial',num2str(start(3)));
    [t,x,y] = sim('robotmove',t);
    start = [y(end,1),y(end,2),y(end,3)];
    plot(y(:,1),y(:,2),'bl');

    %user enters new values
    u = input('Enter robot''s speed\n','s');
    if(strcmpi(u,'quit') == true)
        check = false;
    end %if
    if (check == true)
        w = input('Enter robot''s angular velocity\n','s');
        if(strcmpi(w,'quit') == true)
            check = false;
        end %if
    end %if

    angle = y(end,3);
    begintime = begintime+1;
end %while
hold off;
close;

end %interactrobot

```

robotmove

This model simulates the basic movement of a single robot
based on the classic nonholonomic model.
Orientation is then taken into account.

(This model is called from the matlab function interactrobot.m)



Appendix C

interactrobot2.m

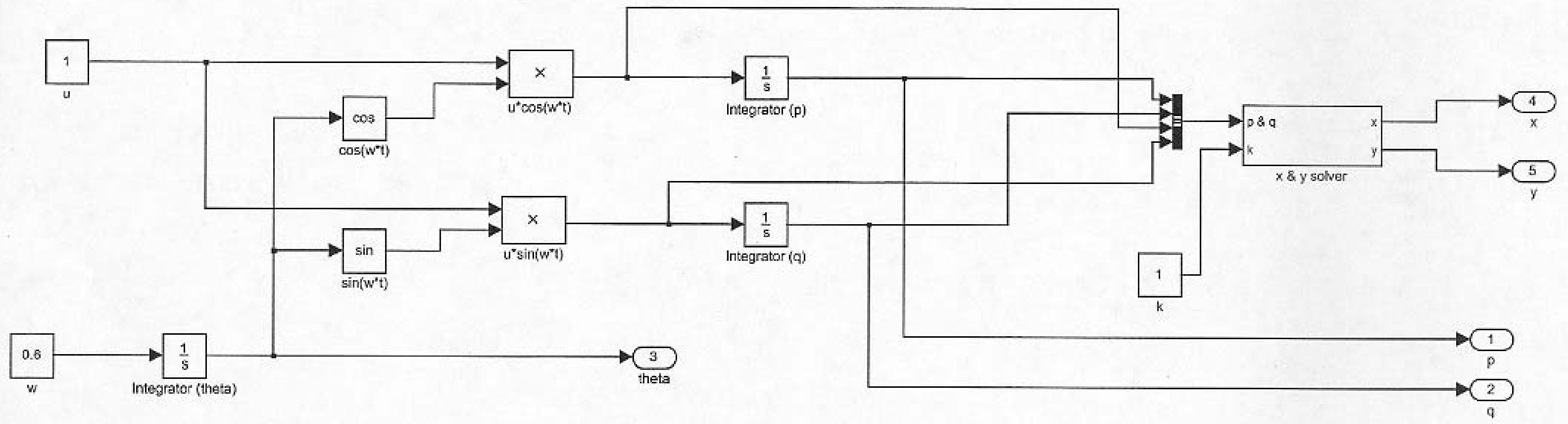
robotmove2.mdl

```
function interactrobot2(ini_p,ini_q,ini_theta,ini_x,ini_y)

% This function calls the robotmove simulink model to chart how the robot
% will move over time. In addition this function will allow user changes to
% the values of omega and u during the simulation. Also has a point
% particle following the robot in the classical pursuit fashion
%
%  Inputs:  ini_p - initial x position of prey robot
%           ini_q - initial y position of prey robot
%           ini_theta - initial angle of prey robot
%           ini_x - initial x position of predator robot
%           ini_y - initial y position of predator robot
%

start = [ini_p,ini_q,ini_theta,ini_x,ini_y];
check = true;
begintime = 0;


%enter starting robot parameters
disp('Enter the word "quit" at any prompt to end simulation');
u = input('Enter robot''s speed\n','s');
if(strcmpi(u,'quit') == true)
    check = false;
end %if
if(check == true)
    w = input('Enter robot''s angular velocity\n','s');
    if(strcmpi(w,'quit') == true)
        check = false;
    end %if
    hold on;
end %if

set_param('robotmove2/k','value',num2str(1.0));

%loop for movement
while(check == true)
    set_param('robotmove2','StartTime', num2str(begintime));
    set_param('robotmove2','StopTime', num2str(begintime+1));
    set_param('robotmove2/u','value',u);
    set_param('robotmove2/w','value',w);
    set_param('robotmove2/Integrator (p)','initial',num2str(start(1)));
    set_param('robotmove2/Integrator (q)','initial',num2str(start(2)));
    set_param('robotmove2/Integrator (theta)','initial',num2str(start(3)));
    set_param('robotmove2/x & y solver/Integrator (x)','initial',num2str(start(4)));
    set_param('robotmove2/x & y solver/Integrator (y)','initial',num2str(start(5)));
    t = begintime:0.05:(begintime+1);
    [t,x,y] = sim('robotmove2',t);
    start = [y(end,1),y(end,2),y(end,3),y(end,4),y(end,5)];
    plot(y(:,1),y(:,2),'bl',y(:,4),y(:,5),'r');
    axis equal;
    pause(0.01);

    %user enters new values
    u = input('Enter robot''s speed\n','s');
    if(strcmpi(u,'quit') == true)
        check = false;
    end %if
    if (check == true)
        w = input('Enter robot''s angular velocity\n','s');
        if(strcmpi(w,'quit') == true)
            check = false;
        end %if
    end %if

    begintime = begintime+1;
end %while

hold off;
close;

end %interactrobot2

```

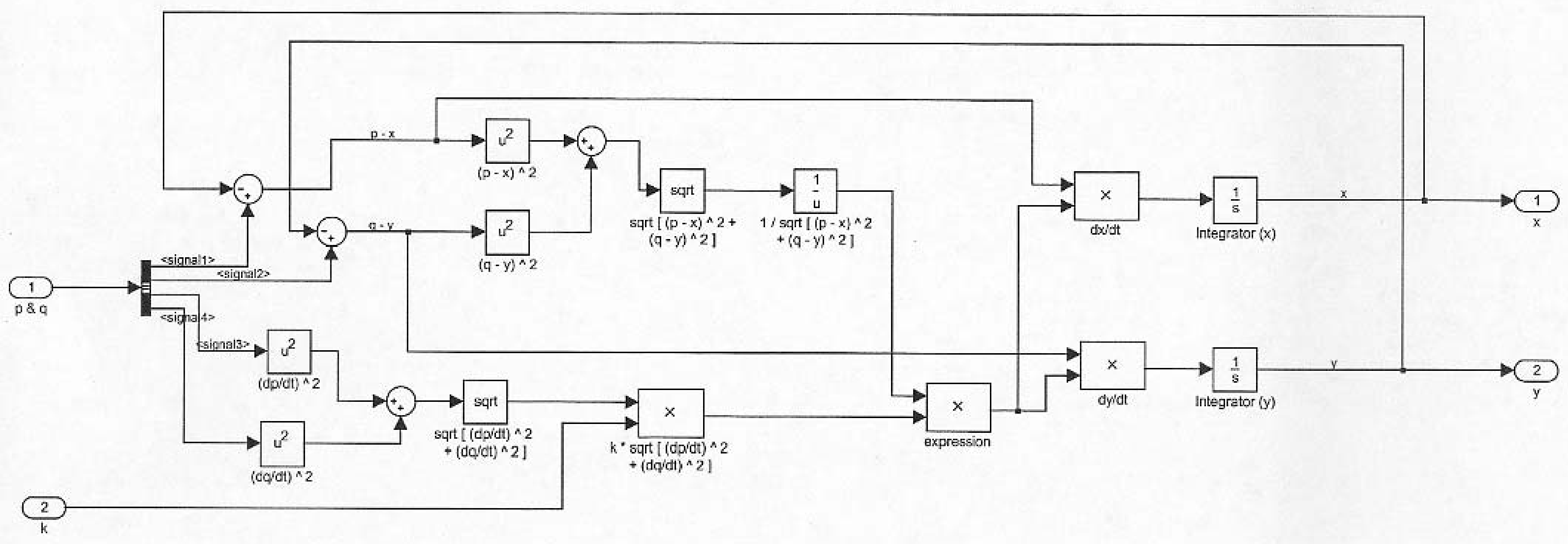
robotmove2

This model simulates pursuit movement where the prey is like a robot but the predator acts as a particle in the classical pursuit method. The movement of the prey is user-input.

(This model is called from the matlab function interactrobot2.m)



robotmove2 / x & y solver



Appendix D

cyclicrobot.m

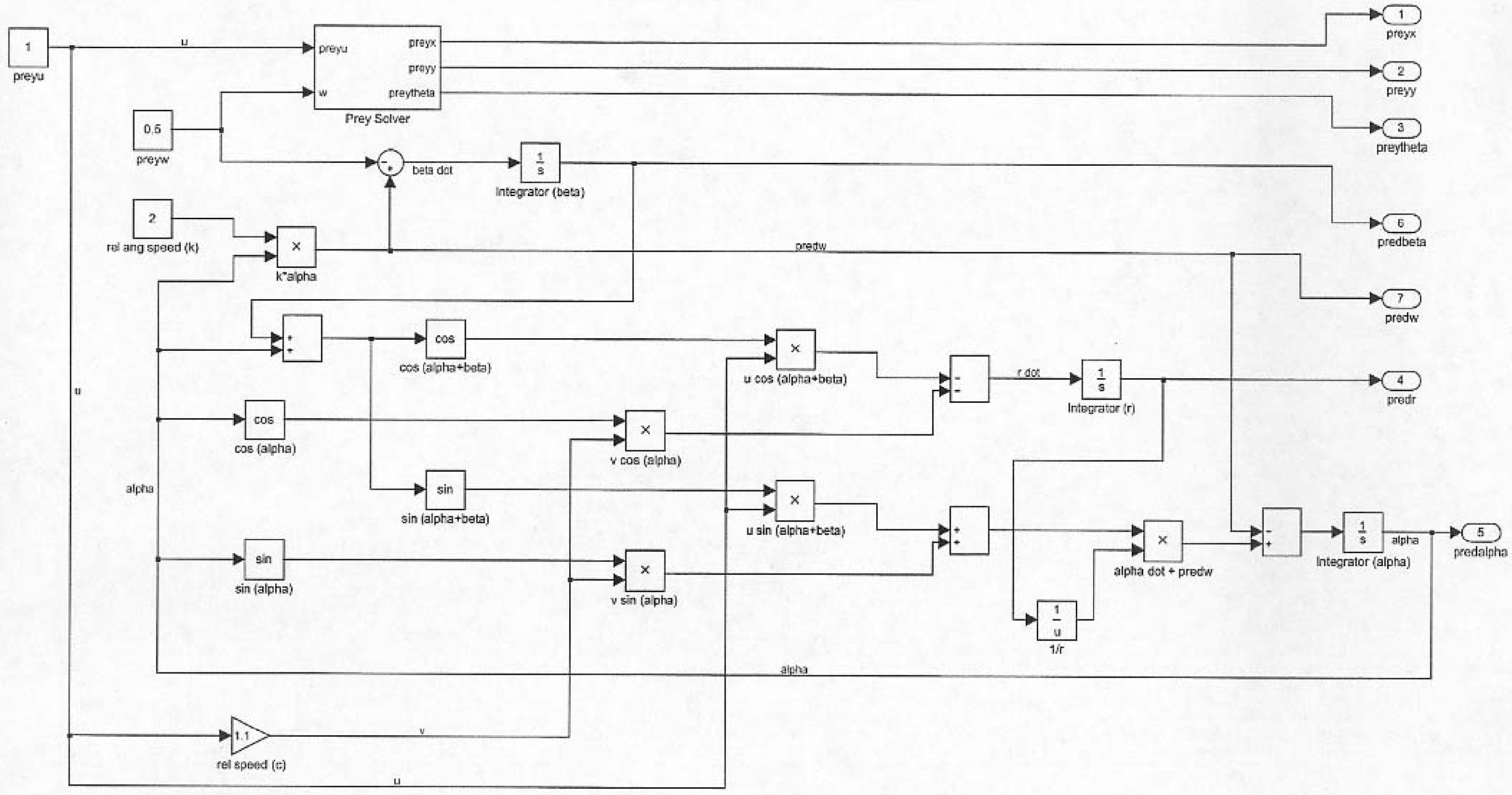
cyclerobotmove.mdl

```
function cyclicrobot(preyx,preyy,preytheta,predx,predy,predtheta)

% This function calls the cyclicrobotmove simulink model to chart how a
% pair of robots will move in a predator/prey situation. This model
% accounts for orientation in both predator and prey robots. In addition
% this function will allow user changes to the values of omega and u during
% the simulation.
%
%  Inputs:  preyx - initial x position of prey robot
%           preyy - initial y position of prey robot
%           preytheta - initial orientation (angle) of prey robot
%           predx - initial x position of predator robot
%           predy - initial y position of predator robot
%           predtheta - initial orientation (angle) of pred robot
%                       (-pi <= predtheta < pi)
%

ini_r = sqrt( (preyx-predx)^2 + (preyy-predy)^2 );
ini_beta = predtheta - preytheta - pi;
phi = atan2(preyy-predy,preyx-predx);
ini_alpha = phi - predtheta;


start = [preyx,preyy,preytheta,ini_r,ini_alpha,ini_beta];
check = true;
begintime = 0;

%enter starting prey robot parameters
disp('Enter the word "quit" at any prompt to end simulation');
u = input('Enter prey robot''s speed\n','s');
if(strcmpi(u,'quit') == true)
    check = false;
end %if
if(check == true)
    w = input('Enter prey robot''s angular velocity\n','s');
    if(strcmpi(w,'quit') == true)
        check = false;
    end %if
    hold on;
end %if

set_param('cyclerobotmove/rel speed (c)','Gain',num2str(1.1));
set_param('cyclerobotmove/rel ang speed (k)','value',num2str(2));

while (check == true)
    set_param('cyclerobotmove','StartTime',num2str(begintime));
    set_param('cyclerobotmove','StopTime',num2str(begintime+1));
    set_param('cyclerobotmove/preyu','value',u);
    set_param('cyclerobotmove/preyw','value',w);
    set_param('cyclerobotmove/Prey Solver/Integrator (preyx)','initial',num2str(start↙
(1)));
    set_param('cyclerobotmove/Prey Solver/Integrator (preyy)','initial',num2str(start↙
(2)));
    set_param('cyclerobotmove/Prey Solver/Integrator (theta)','initial',num2str(start↙
(3)));
    set_param('cyclerobotmove/Integrator (r)','initial',num2str(start(4)));
    set_param('cyclerobotmove/Integrator (alpha)','initial',num2str(start(5)));
    set_param('cyclerobotmove/Integrator (beta)','initial',num2str(start(6)));
    t = begintime:0.05:(begintime+1);
    [t,x,y] = sim('cyclerobotmove',t);
    start = [y(end,1),y(end,2),y(end,3),y(end,4),y(end,5),y(end,6)];
    predxdata = y(:,1) + (y(:,4).*cos(y(:,5)+y(:,6)+y(:,3)));
    predydata = y(:,2) + (y(:,4).*sin(y(:,5)+y(:,6)+y(:,3)));
    plot(y(:,1),y(:,2),'bl',predxdata,predydata,'r');
    axis equal;
    pause(0.01);

    %user enters new values
    u = input('Enter robot''s speed\n','s');
    if(strcmpi(u,'quit') == true)
        check = false;
    end %if
    if (check == true)
        w = input('Enter robot''s angular velocity\n','s');
        if(strcmpi(w,'quit') == true)
            check = false;
        end %if
    end %if

    begintime = begintime+1;
end %while

hold off;
close;

end %cyclicrobot

```

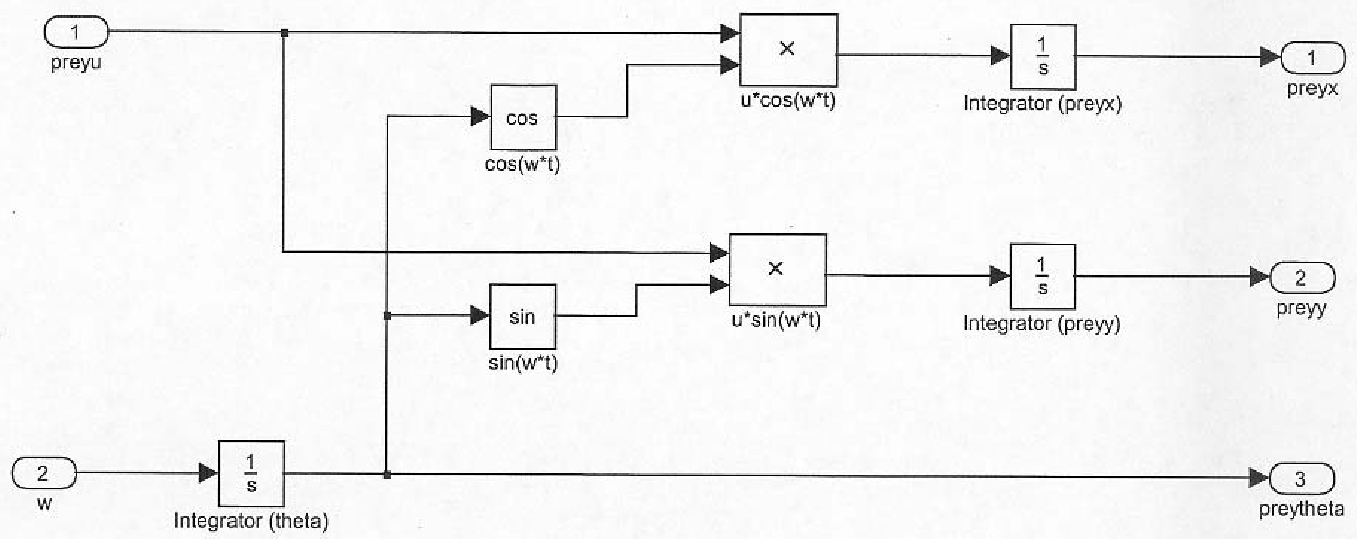
cyclerobotmove

This model is based on the Francis paper about cyclic pursuit.
Orientation is accounted for in both the prey and predator.
Also, the prey's movement is input by the user.

(This model is called from the matlab function cyclicrobot.m)



cyclerobotmove / Prey Solver



Appendix E

cyclicrobotpursuit.m

robotpursuit.mdl

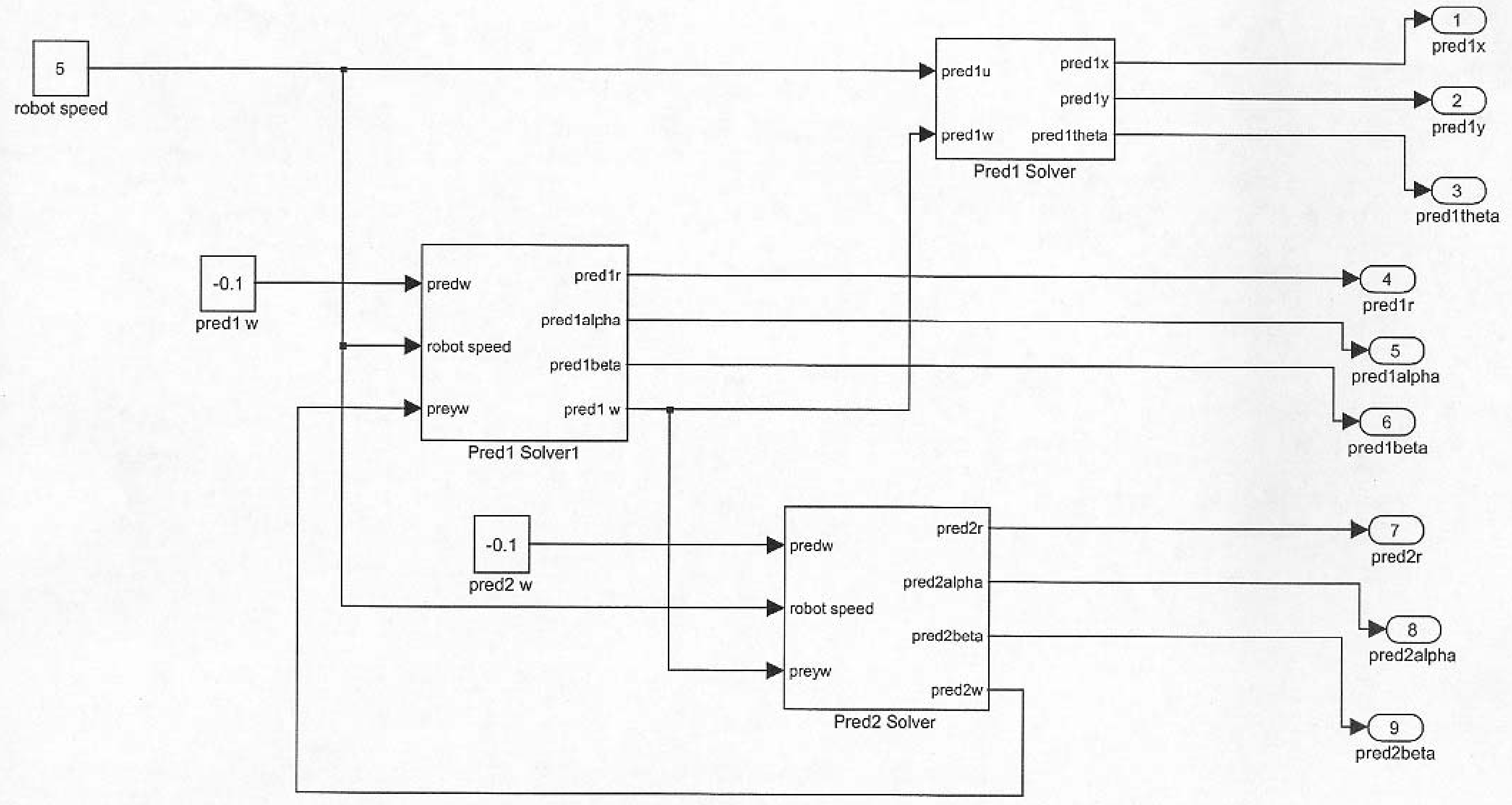
```
function cyclicrobotpursuit(pred1x,pred1y,pred1theta,pred2x,pred2y,pred2theta)

% This function calls the cyclicrobotmove simulink model to chart how a
% pair of robots will move in a mice problem type of situation. This model
% accounts for orientation in both predator and prey robots. and both
% robots are predators in a sense
%
%  Inputs:  pred1x - initial x position of pred1 robot
%           pred1y - initial y position of pred1 robot
%           pred1theta - initial orientation (angle) of pred1 robot
%           pred2x - initial x position of pred2 robot
%           pred2y - initial y position of pred2 robot
%           pred2theta - initial orientation (angle) of pred2 robot
%

ini_r = sqrt( (pred1x-pred2x)^2 + (pred1y-pred2y)^2 );
ini_pred1beta = pred1theta - pred2theta - pi;
ini_pred2beta = pred2theta - pred1theta - pi;
phi1 = atan2(pred2y-pred1y,pred2x-pred1x);
phi2 = atan2(pred1y-pred2y,pred1x-pred2x);
ini_pred1alpha = phi1 - pred1theta;
ini_pred2alpha = phi2 - pred2theta;


start = [pred1x,pred1y,pred1theta,ini_r,ini_pred1alpha,ini_pred1beta,...
    ini_r,ini_pred2alpha,ini_pred2beta];
check = true;
begintime = 0;
hold on;
set_param('robotpursuit/robot speed','value',num2str(5));


while (check == true)
    set_param('robotpursuit','StartTime',num2str(begintime));
    set_param('robotpursuit','StopTime',num2str(begintime+1));

    set_param('robotpursuit/Pred1 Solver/Integrator (pred1x)','initial',num2str(start↙
(1)));
    set_param('robotpursuit/Pred1 Solver/Integrator (pred1y)','initial',num2str(start↙
(2)));
    set_param('robotpursuit/Pred1 Solver/Integrator (theta)','initial',num2str(start↙
(3)));
    set_param('robotpursuit/Pred1 Solver1/Integrator (r)','initial',num2str(start(4)));
    set_param('robotpursuit/Pred1 Solver1/Integrator (alpha)','initial',num2str(start↙
(5)));
    set_param('robotpursuit/Pred1 Solver1/Integrator (beta)','initial',num2str(start↙
(6)));
    set_param('robotpursuit/Pred2 Solver/Integrator (r)','initial',num2str(start(7)));
    set_param('robotpursuit/Pred2 Solver/Integrator (alpha)','initial',num2str(start↙
(8)));
    set_param('robotpursuit/Pred2 Solver/Integrator (beta)','initial',num2str(start↙
(9)));
```

```
    if abs(start(5)) < 0.0872665
        set_param('robotpursuit/pred1 w','value',num2str(0));
    else
        if start(5) > 0
            set_param('robotpursuit/pred1 w','value',num2str(0.1));
        else
            set_param('robotpursuit/pred1 w','value',num2str(-0.1));
        end
    end
    if abs(start(8)) < 0.0872665
        set_param('robotpursuit/pred2 w','value',num2str(0));
    else
        if start(8) > 0
            set_param('robotpursuit/pred2 w','value',num2str(0.1));
        else
            set_param('robotpursuit/pred2 w','value',num2str(-0.1));
        end
    end
    t = begintime:(begintime+1);
    [t,x,y] = sim('robotpursuit',t);
    start = [y(end,1),y(end,2),y(end,3),y(end,4),y(end,5),y(end,6),y(end,7),y(end,8),y↙
(end,9)];
    pred2xdata = y(:,1) + (y(:,7).*cos(y(:,8)+y(:,9)+y(:,3)));
    pred2ydata = y(:,2) + (y(:,7).*sin(y(:,8)+y(:,9)+y(:,3)));
    plot(y(:,1),y(:,2),'bl',pred2xdata,pred2ydata,'r','LineWidth',2);
    for count = 1:length(pred2xdata)
        if dist2d(y(count,1),y(count,2),pred2xdata(count),pred2ydata(count)) < 10
            check = false;
        end
    end
    pause(0.01);

    begintime = begintime+1;
end %while

hold off;
close;

end %cyclicrobotpursuit

%------------------------------------------------------------------------
function d = dist2d(x1,y1,x2,y2)

d = sqrt((x2-x1)^2+(y2-y1)^2);

end %dist2d
```

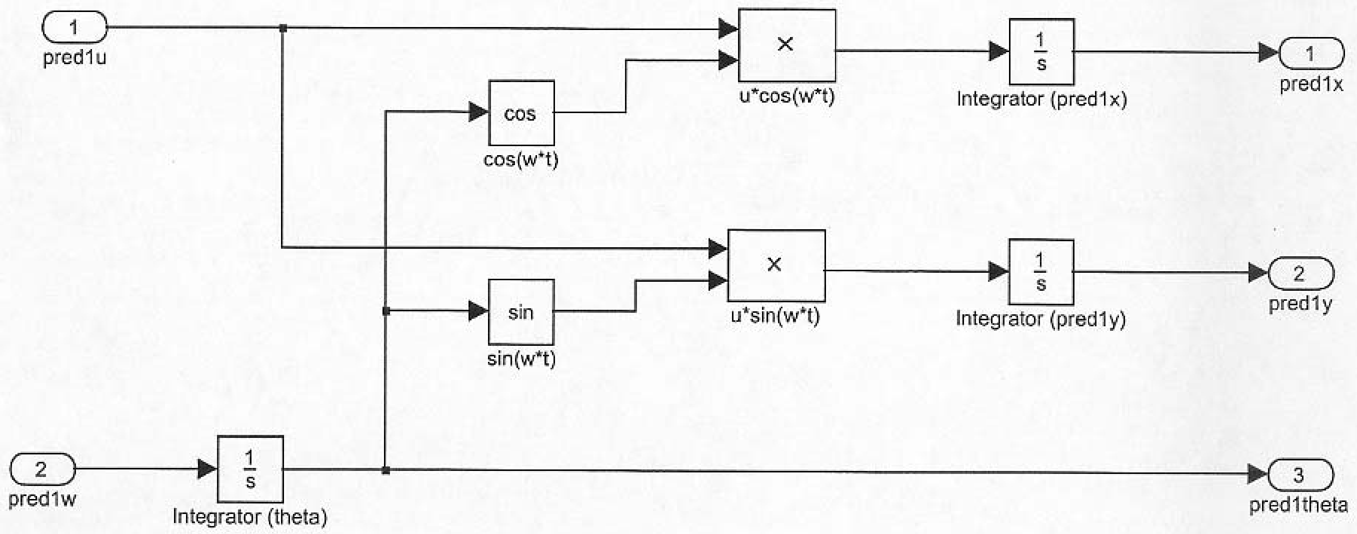
robotpursuit

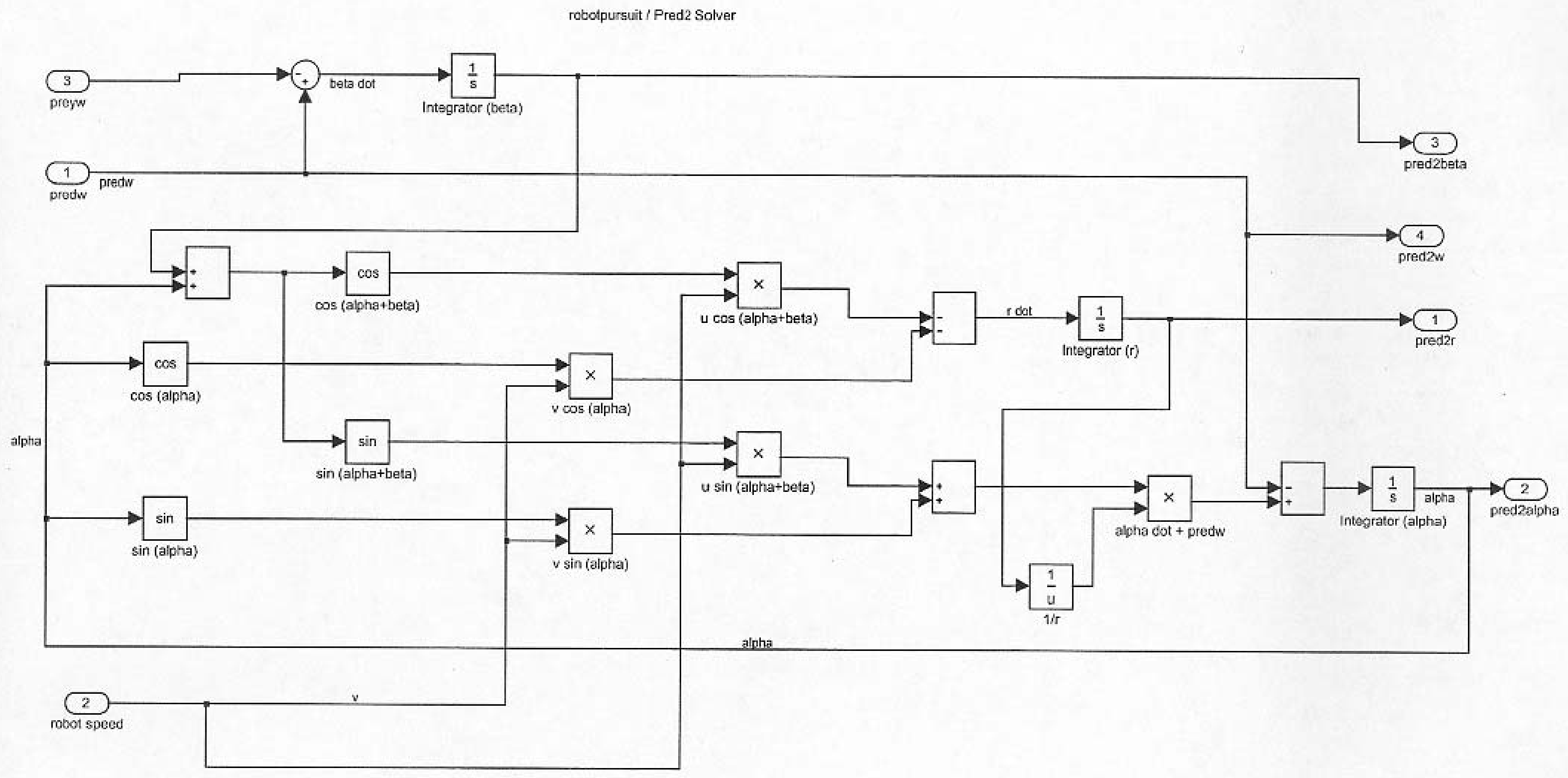
This model uses cyclic pursuit to simulate the real world trials done by Haggag and Mehraei. All controls are entered through the Matlab function.

(This model is called from the matlab function cyclicrobotpursuit.m)



robotpursuit / Pred1 Solver




robotpursuit / Pred2 Solver
3
preyw
beta dot
1/s
Integrator (beta)
1
predw
predw
cos
cos (alpha+beta)
u cos (alpha+beta)
r dot
1/s
Integrator (r)
cos
cos (alpha)
v cos (alpha)
sin
sin (alpha+beta)
u sin (alpha+beta)
alpha
sin
sin (alpha)
v sin (alpha)
1/u
1/r
alpha dot + predw
1/s
alpha
Integrator (alpha)
alpha
2
robot speed
v
3
pred2beta
4
pred2w
1
pred2r
2
pred2alpha

Appendix F

classicmice_v4.m

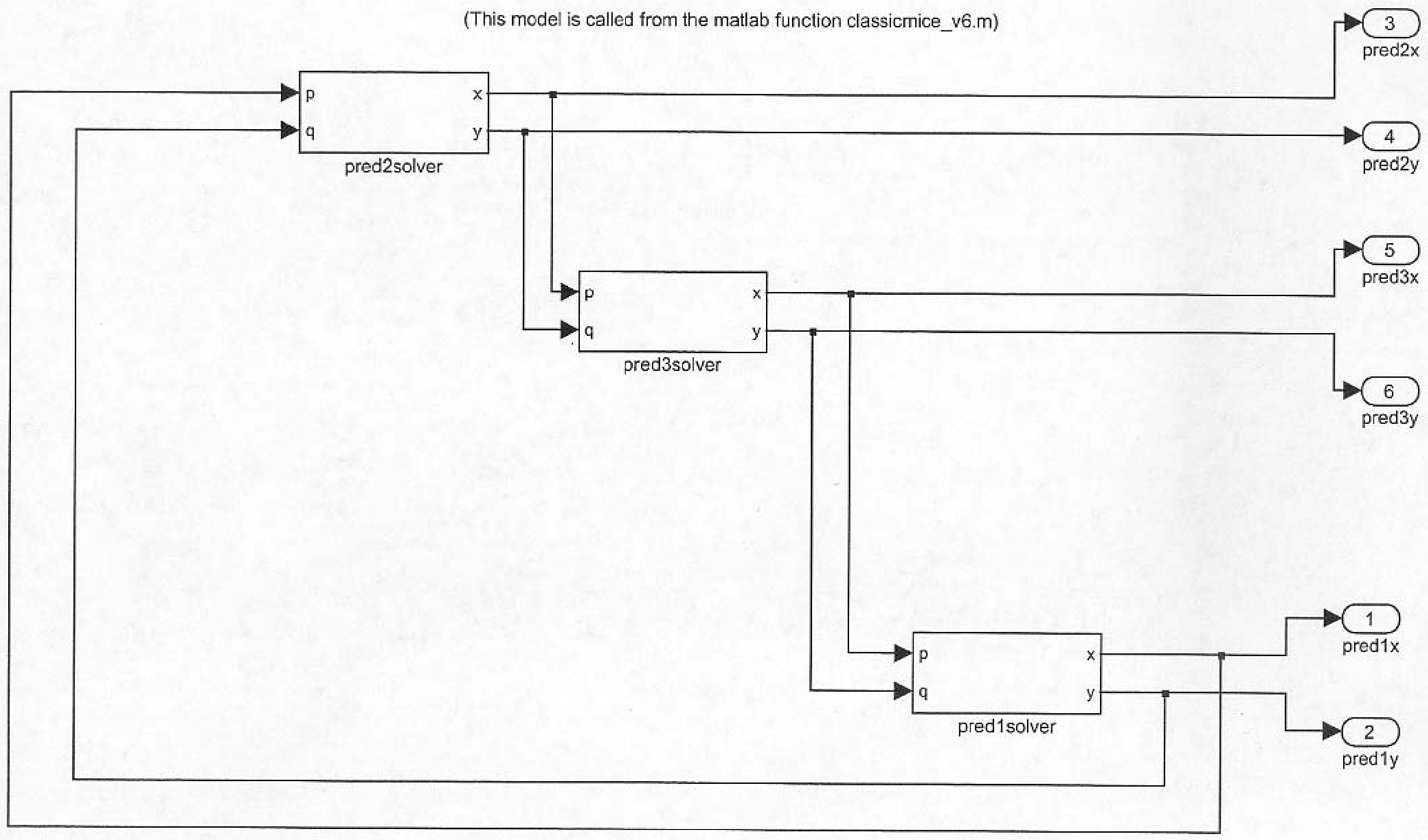
micesolver.mdl

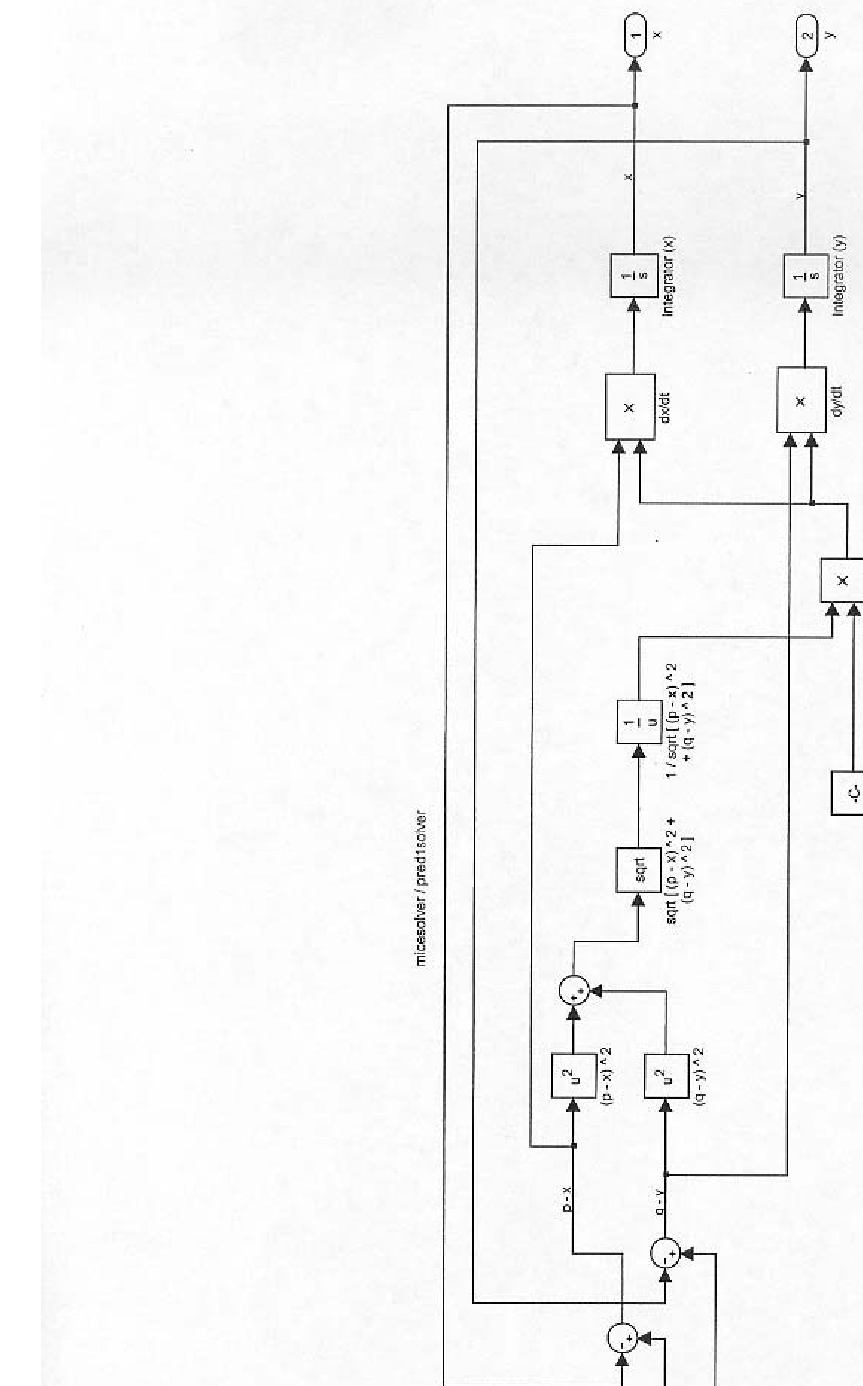
```
function classicmice_v3(x1,y1,x2,y2,x3,y3)

% This function calls the micesolver simulink model to simulate the
% Mice Problem. This model utilizes classical pursuit modelling. It also
% adds relative speeds for all three mice, so it works for more than just
% the equilateral case
%
%  Inputs:  x1 = x position of first particle
%           y1 = y position of first particle
%           x2 = x position of second particle
%           y2 = y position of second particle (must be zero)
%           x3 = x position of third particle (must be zero)
%           y3 = y position of third particle (must be zero)
%

check = true;
begintime = 0;
start = [x1,y1,x2,y2,x3,y3];


%triangle information
a = dist2d(x1,y1,x2,y2);
b = dist2d(x3,y3,x1,y1);
c = dist2d(x3,y3,x2,y2);
s = (a+b+c)/2;
area = sqrt(s*(s-a)*(s-b)*(s-c));
angle = atan((4*area)/(a^2+b^2+c^2));
k = (2*area)/(((c*a)/b)+((a*b)/c)+((b*c)/a));
brocardy = (k*b)/a;
brocardx = brocardy/tan(angle);
hold on;
plot(brocardx,brocardy,'kp');

%relative speeds
pred1speed = k*(a/c);
pred2speed = k*(c/b);
pred3speed = k*(b/a);
set_param('micesolver/pred1solver/pred1 speed','value',num2str(pred1speed));
set_param('micesolver/pred2solver/pred2 speed','value',num2str(pred2speed));
set_param('micesolver/pred3solver/pred3 speed','value',num2str(pred3speed));

while (check == true)
    set_param('micesolver','StartTime',num2str(begintime));
    set_param('micesolver','StopTime',num2str(begintime+1));
    set_param('micesolver/pred1solver/Integrator (x)','initial',num2str(start(1)));
    set_param('micesolver/pred1solver/Integrator (y)','initial',num2str(start(2)));
    set_param('micesolver/pred2solver/Integrator (x)','initial',num2str(start(3)));
    set_param('micesolver/pred2solver/Integrator (y)','initial',num2str(start(4)));
    set_param('micesolver/pred3solver/Integrator (x)','initial',num2str(start(5)));
    set_param('micesolver/pred3solver/Integrator (y)','initial',num2str(start(6)));
    t = begintime:0.05:begintime+1;
    [t,x,y] = sim('micesolver',t);
    start = [y(end,1),y(end,2),y(end,3),y(end,4),y(end,5),y(end,6)];
    plot(y(:,1),y(:,2),'bl',y(:,3),y(:,4),'r',y(:,5),y(:,6),'g');
    axis equal;
    response = input('type ''quit'' to end the simulation\n','s');
    if(strcmpi(response,'quit') == true)
        check = false;
    end %if
    begintime = begintime+1;
end %while

hold off;
close;

end % classicmice_v4

%-------------------------------------------------------------------------
function d = dist2d(x1,y1,x2,y2)

d = sqrt((x2-x1)^2+(y2-y1)^2);

end %dist2d

```

micesolver

This model uses separate classical pursuit solvers to simulate the mice problem. Can work for any triangle using the relative speeds and other information transferred from the Matlab function

(This model is called from the matlab function classicmice_v6.m)




micesolver / pred1solver
1 p
2 q
p - x
q - y
u^2
(p - x)^2
(q - y)^2
sqrt
sqrt [(p - x)^2 + (q - y)^2]
1/u
1 / sqrt [(p - x)^2 + (q - y)^2]
-C-
pred1 speed
Product
dx/dt
dy/dt
1/s
Integrator (x)
Integrator (y)
x
y
1 x
2 y

Appendix G

robotmice_v6.m

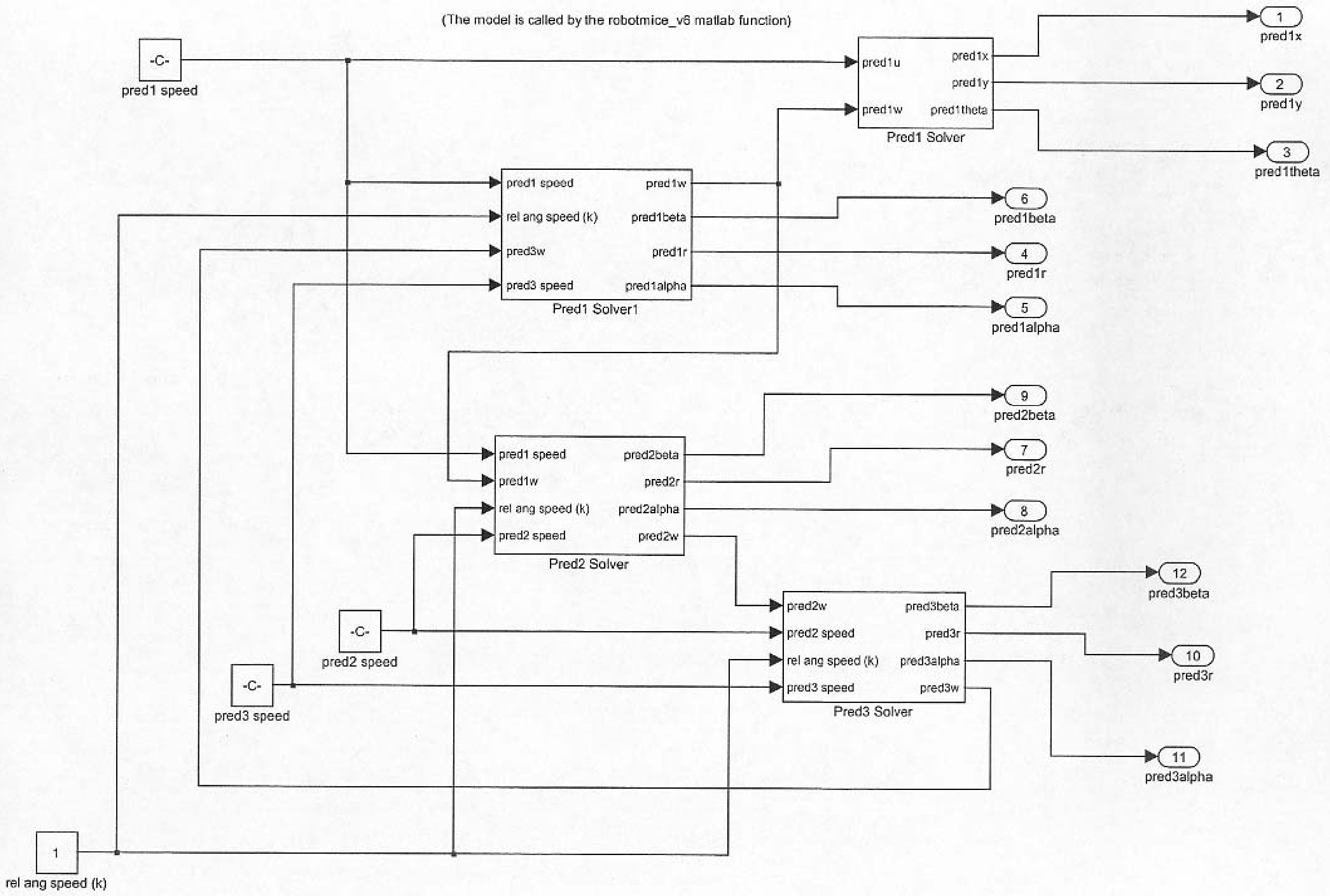
cyclerobotmove6.mdl

```
function robotmice_v6(pred1x,pred1y,pred2x,pred2y,pred3x,pred3y)

% This function calls the cyclicrobotmove4 simulink model to chart how
% three robots will move in a pursuit situation (Mice Problem). This model
% accounts for orientation in all three robots. (this model improves on
% robotmice_v5 by adding in relative speeds for the three robots)
%
%  Inputs:  pred1x - initial x position of pred1 robot
%           pred1y - initial y position of pred1 robot
%           pred2x - initial x position of pred2 robot
%           pred2y - initial y position of pred2 robot (must be zero)
%           pred3x - initial x position of pred3 robot (must be zero)
%           pred3y - initial y position of pred3 robot (must be zero)
%

check = true;
begintime = 0;

%triangle information
pred3theta = 0;
pred2theta = atan2(pred1y-pred2y,pred1x-pred2x);
pred1theta = -pi + atan2(pred1y-pred3y,pred1x-pred3x);
a = dist2d(pred1x,pred1y,pred2x,pred2y);
b = dist2d(pred3x,pred3y,pred1x,pred1y);
c = dist2d(pred3x,pred3y,pred2x,pred2y);
s = (a+b+c)/2;
area = sqrt(s*(s-a)*(s-b)*(s-c));
angle = atan((4*area)/(a^2+b^2+c^2));
k = (2*area)/(((c*a)/b)+((a*b)/c)+((b*c)/a));
brocardy = (k*b)/a;
brocardx = brocardy/tan(angle);


%determine all initial conditions (r values first)
ini_pred1r = sqrt( (pred3x-pred1x)^2 + (pred3y-pred1y)^2 );
ini_pred2r = sqrt( (pred1x-pred2x)^2 + (pred1y-pred2y)^2 );
ini_pred3r = sqrt( (pred2x-pred3x)^2 + (pred2y-pred3y)^2 );
%now beta values
ini_pred1beta = pred1theta - pred3theta + pi;
ini_pred2beta = pred2theta - pred1theta - pi;
ini_pred3beta = -pred3theta - pred2theta - pi;
%now phi values (used to find alpha)
phi1 = atan2(pred3y-pred1y,pred3x-pred1x);
phi2 = atan2(pred1y-pred2y,pred1x-pred2x);
phi3 = atan2(pred2y-pred3y,pred2x-pred3x);
%and last are the alpha values
ini_pred1alpha = phi1 - pred1theta;
ini_pred2alpha = phi2 - pred2theta;
ini_pred3alpha = phi3 - pred3theta;

start = [pred1x,pred1y,pred1theta,...
    ini_pred1r,ini_pred1alpha,ini_pred1beta,...
    ini_pred2r,ini_pred2alpha,ini_pred2beta,...
    ini_pred3r,ini_pred3alpha,ini_pred3beta];


%relative speed
pred1speed = k*(a/c);
pred2speed = k*(c/b);
pred3speed = k*(b/a);
set_param('cyclerobotmove6/pred1 speed','value',num2str(pred1speed));
set_param('cyclerobotmove6/pred2 speed','value',num2str(pred2speed));
set_param('cyclerobotmove6/pred3 speed','value',num2str(pred3speed));
set_param('cyclerobotmove6/rel ang speed (k)','value',num2str(1));

hold on;
plot(brocardx,brocardy,'kp');

while (check == true)
    set_param('cyclerobotmove6','StartTime',num2str(begintime));
    set_param('cyclerobotmove6','StopTime',num2str(begintime+1));
    set_param('cyclerobotmove6/Pred1 Solver/Integrator (pred1x)','initial',num2str(start(1)));
    set_param('cyclerobotmove6/Pred1 Solver/Integrator (pred1y)','initial',num2str(start(2)));
    set_param('cyclerobotmove6/Pred1 Solver/Integrator (theta)','initial',num2str(start(3)));
    set_param('cyclerobotmove6/Pred1 Solver1/Integrator (r)','initial',num2str(start(4)));
    set_param('cyclerobotmove6/Pred1 Solver1/Integrator (alpha)','initial',num2str(start(5)));
    set_param('cyclerobotmove6/Pred1 Solver1/Integrator (beta)','initial',num2str(start(6)));
    set_param('cyclerobotmove6/Pred2 Solver/Integrator (r)','initial',num2str(start(7)));
    set_param('cyclerobotmove6/Pred2 Solver/Integrator (alpha)','initial',num2str(start(8)));
    set_param('cyclerobotmove6/Pred2 Solver/Integrator (beta)','initial',num2str(start(9)));
    set_param('cyclerobotmove6/Pred3 Solver/Integrator (r)','initial',num2str(start(10)));
    set_param('cyclerobotmove6/Pred3 Solver/Integrator (alpha)','initial',num2str(start(11)));
    set_param('cyclerobotmove6/Pred3 Solver/Integrator (beta)','initial',num2str(start(12)));
    t = begintime:0.05:(begintime+1);
    [t,x,y] = sim('cyclerobotmove6',t);
    start = [y(end,1),y(end,2),y(end,3),y(end,4),y(end,5),y(end,6),y(end,7),y(end,8),y(end,9),y(end,10),y(end,11),y(end,12)];
    pred2xdata = y(:,1) + (y(:,7).*cos(y(:,8)+y(:,9)+y(:,3)));
    pred2ydata = y(:,2) + (y(:,7).*sin(y(:,8)+y(:,9)+y(:,3)));
    pred2phi = atan2(y(:,2)-pred2ydata,y(:,1)-pred2xdata);
    thetavalues = pred2phi - y(:,8);
    pred3xdata = pred2xdata + (y(:,10).*cos(y(:,11)+y(:,12)+thetavalues));
    pred3ydata = pred2ydata + (y(:,10).*sin(y(:,11)+y(:,12)+thetavalues));
    plot(y(:,1),y(:,2),'bl',pred2xdata,pred2ydata,'r',pred3xdata,pred3ydata,'g');
    axis equal;
    response = input('type ''quit'' to end the simulation\n','s');
    if(strcmpi(response,'quit') == true)
        check = false;
    end %if
    begintime = begintime+1;
end %while

hold off;
close;

end %robotmice_v6

%--------------------------------------------------------------------------
function d = dist2d(x1,y1,x2,y2)

d = sqrt((x2-x1)^2+(y2-y1)^2);

end %dist2d

```

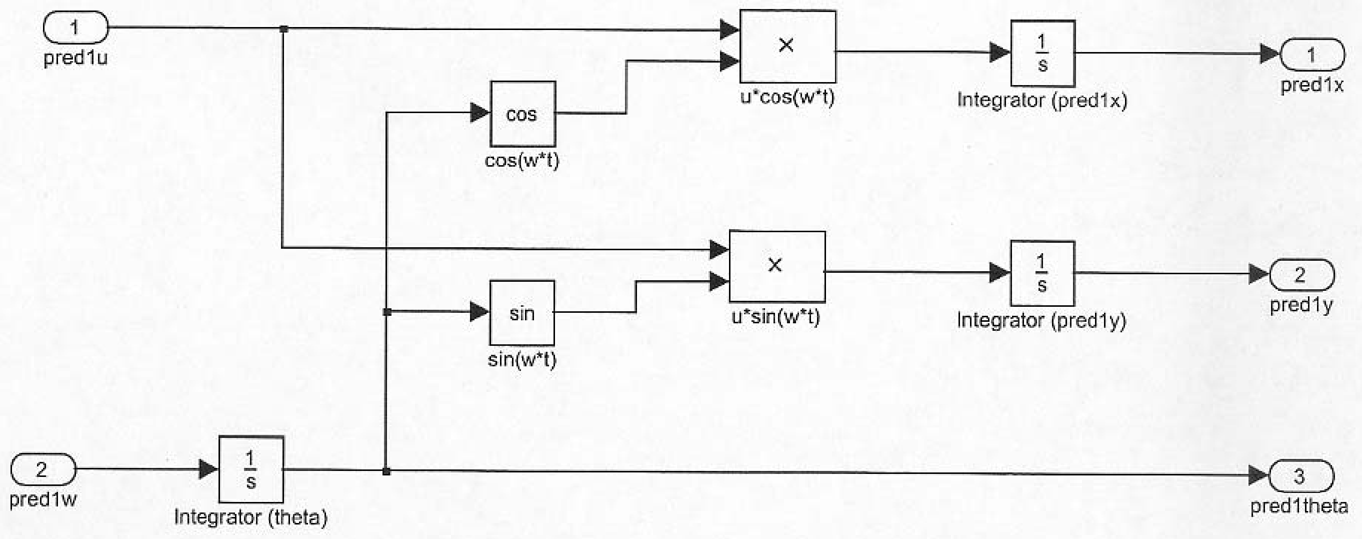
cyclerobotmove6

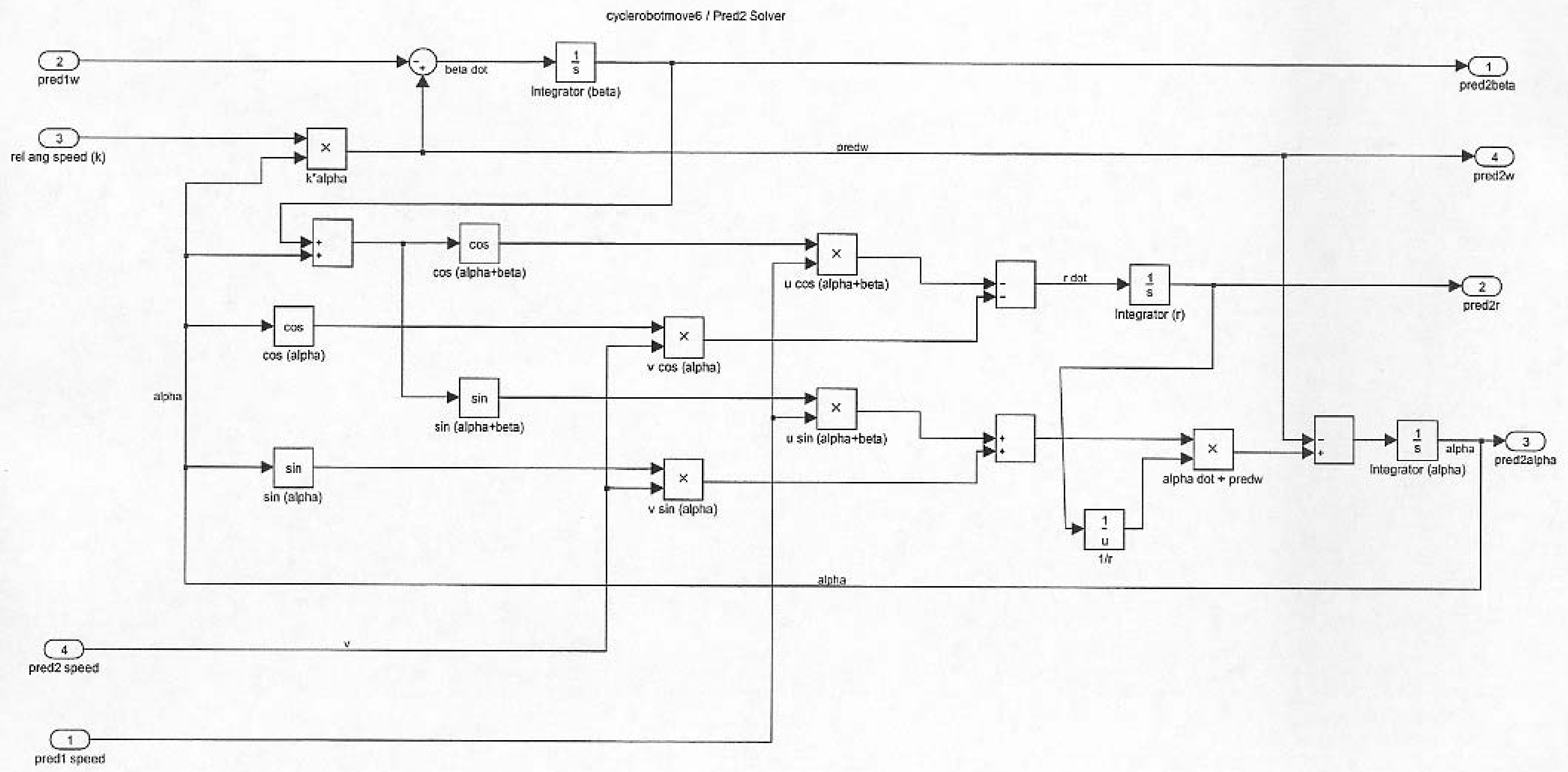
This model accurately works for the cyclic version of the mice problem. All necessary relative speeds are taken into account. The mice problem is now completed for the cyclic motion and can be compared to the fixed classical case as well.

(The model is called by the robotmice_v6 matlab function)



cyclerobotmove6 / Pred1 Solver




cyclerobotmove6 / Pred2 Solver
2 pred1w
3 rel ang speed (k)
4 pred2 speed
1 pred1 speed
beta dot
1/s
Integrator (beta)
k*alpha
predw
cos (alpha+beta)
cos (alpha)
sin (alpha+beta)
sin (alpha)
alpha
v
u cos (alpha+beta)
v cos (alpha)
u sin (alpha+beta)
v sin (alpha)
r dot
Integrator (r)
1/u
1/r
alpha dot + predw
Integrator (alpha)
1 pred2beta
4 pred2w
2 pred2r
3 pred2alpha

Appendix H

MotionCamoRobot.m

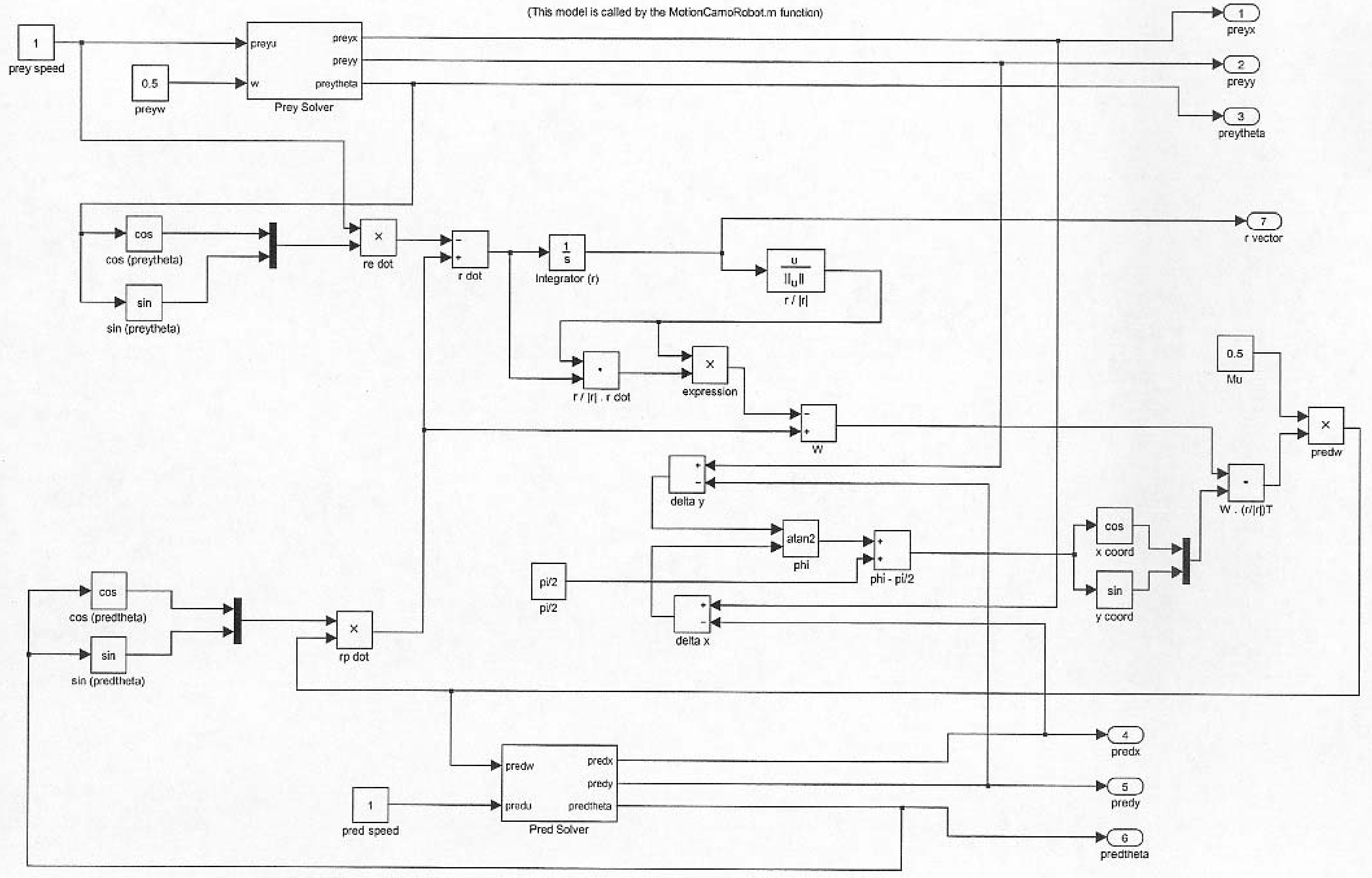
motioncamouflage.mdl

```
function MotionCamoRobot(preyx,preyy,preytheta,predx,predy,predtheta)

% This function calls the motioncamouflage simulink model to chart how a
% pair of robots in a predator/prey situation would move if the predator
% robot follows a motion camouflage control scheme. This model fully
% accounts for orientation in both robots. In addition this function will
% allow user changes to the values of speed and angular velocity (for the
% prey) during the simulation.
%
%  Inputs:  preyx - initial x position of prey robot
%           preyy - initial y position of prey robot
%           preytheta - initial orientation (angle) of prey robot
%           predx - initial x position of predator robot
%           predy - initial y position of predator robot
%           predtheta - initial orientation (angle) of pred robot
%

ini_rvector = [predx-preyx,predy-preyy];

start = [preyx,preyy,preytheta,predx,predy,predtheta,ini_rvector];
check = true;
begintime = 0;

%enter starting prey robot parameters
disp('Enter the word "quit" at any prompt to end simulation');
u = input('Enter prey robot''s speed\n','s');
if(strcmpi(u,'quit') == true)
    check = false;
end %if
if(check == true)
    w = input('Enter prey robot''s angular velocity\n','s');
    if(strcmpi(w,'quit') == true)
        check = false;
    end %if
    hold on;
end %if

set_param('motioncamouflage/Mu','value',num2str(0.5));

while (check == true)
    set_param('motioncamouflage','StartTime',num2str(begintime));
    set_param('motioncamouflage','StopTime',num2str(begintime+1));
    set_param('motioncamouflage/prey speed','value',u);
    set_param('motioncamouflage/pred speed','value',num2str(1));
    set_param('motioncamouflage/preyw','value',w);
    set_param('motioncamouflage/Prey Solver/Integrator (preyx)','initial',num2str(start↙
(1)));
    set_param('motioncamouflage/Prey Solver/Integrator (preyy)','initial',num2str(start↙
(2)));
    set_param('motioncamouflage/Prey Solver/Integrator (theta)','initial',num2str(start↙
(3)));
    set_param('motioncamouflage/Pred Solver/Integrator (predx)','initial',num2str(start↙
(4)));
    set_param('motioncamouflage/Pred Solver/Integrator (predy)','initial',num2str(start(5)));
    set_param('motioncamouflage/Pred Solver/Integrator (theta)','initial',num2str(start(6)));
    set_param('motioncamouflage/Integrator (r)','initial',num2str(start(7)));
    t = begintime:0.05:(begintime+1);
    [t,x,y] = sim('motioncamouflage',t);
    start = [y(end,1),y(end,2),y(end,3),y(end,4),y(end,5),y(end,6),y(end,7)];
    plot(y(:,1),y(:,2),'bl',y(:,4),y(:,5),'r');
    plot([start(1),start(4)],[start(2),start(5)],'g');
    axis equal;
    pause(0.01);

    %user enters new values
    u = input('Enter robot''s speed\n','s');
    if(strcmpi(u,'quit') == true)
        check = false;
    end %if
    if (check == true)
        w = input('Enter robot''s angular velocity\n','s');
        if(strcmpi(w,'quit') == true)
            check = false;
        end %if
    end %if

    begintime = begintime+1;
end %while

hold off;
close;

end %MotionCamoRobot
```

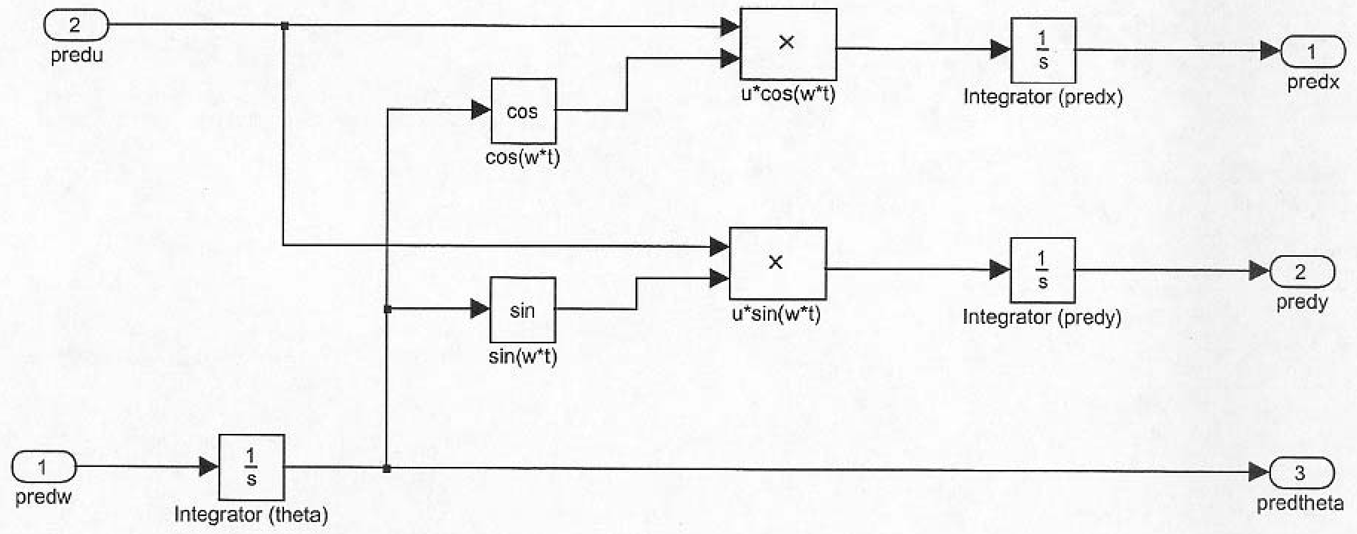
motioncamouflage

This model simulates the motion camouflage strategy for the robots.
Controls for the prey can be entered by the user through the
Matlab function. Traditional solvers are used except for the predator's
angular velocity

(This model is called by the MotionCamoRobot.m function)



motioncamouflage / Pred Solver


2
predu
1
predw
1/s
Integrator (theta)
cos
cos(w*t)
sin
sin(w*t)
×
u*cos(w*t)
×
u*sin(w*t)
1/s
Integrator (predx)
1/s
Integrator (predy)
1
predx
2
predy
3
predtheta