

Fuzzy Logic and Basic Robotics Homework

T. Nathan Mundhenk and Laurent Itti
University of Southern California
Department of Computer Science

Copyright (c) 2006 T. Nathan Mundhenk, Laurent Itti

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

CS561 Homework 4

Due: By class Thursday, April 6, 2006

Guidelines:

This assignment has a written and programming component. You can see how much each part of the assignment is worth by the percentage next to it. For the written part, please turn in typewritten answers. You are not English majors, however blatant spelling and grammatical errors may cost you points, so be sure and run the spell check at least once. Answers on the written part will require justification and a simple yes/no answer will almost certainly garner you little or no points. In general, it is safer to write longer answers than shorter ones, but stay focused as a long but off-topic answer will not work either. This way, we can discern your train of thoughts better and grant partial credit for a wrong answer, if you were on the right track. Graphs must be neat and tidy. Hand-drawn graphs are OK, but computer-drawn graphs (e.g., made with Adobe Illustrator or Microsoft Power Point) are preferred. If you draw a graph by hand, be sure and use a straight edge (ruler) so it looks neat.

For the programming part, you will be provided sample inputs and at least one sample output. Since each homework is checked via an automated Perl script, your output should match the example format *exactly*. Failure to do so will most certainly cost some points. Since the output format is simple and there is an example on the web, this should not be a problem. Additionally, if your code generates a large number of warnings during compilation, you may lose points, so try and eliminate compile-time warnings. Additionally, your code should be well documented. If something goes wrong during compile and grading, if the fix proves easy, the amount of points lost will be far less. As such, documentation makes fixing easier, so it is to your advantage to do so.

You will be provided with a stub Perl script called “stubby.pl”, which works like the grading Perl script. You can run this script to make sure that your project will work properly. Thus, it is expected that your project will run through the grading Perl script without problems. Pedantically, we assert it will cost you points if your code does not run on the Perl script correctly. You will be able to tell if your output is correct if stubby shows each of the lines from your program output is exactly the same as from the comparison file which shows what output you should be getting.

To run stubby, copy it to your code directory along with the input, output and comparison files. Be sure to be in your code directory then type “perl stubby.pl”. The script is loaded with all sorts of output and feedback, so you should be able to see what it is doing if for some reason it isn’t working for you. If you don’t understand the script and want to know more about Perl, go to <http://www.perl.org/>.

Small amounts of extra credit (not more than 10% total) are given for all sorts of things. So long as you meet the base homework requirements anything creative, fun, interesting or outright cool

will most likely earn you extra credit. What is cool and earns you extra credit is somewhat subjective and bound to the whim of the grader.

Handing in the Assignment:

Since the class is very large it is important to hand in your homework as directed. *The homework is due before class on the due date shown.* There are two parts of the homework, which you must hand in. These are:

- (A) Your code tar/gzipped in **one** file. Do not send the binaries. - This should just include your uncompiled code and a readme file called readme.txt. When I run tar/gzip to uncompress your file, it should uncompress into its own directory. To keep things uniform, do not use bzip2. I should then be able to run my stubby Perl script and grade your code. See below on how to tar/gzip your code.
- (B) Your written part on paper. *Iff* you are a DEN student, you may submit your written part in the DEN digital drop-box (remember your cover sheet). Otherwise, you must submit a paper copy either in class or in a drop-box in front of my office in HNB (NOTE: the HNB building closes at 5:00pm). Late submissions will be noted. Be sure your homework is stapled together and is generally tidy. Electronic submissions of the written part from non-DEN students will not be accepted.

Be sure that your name is *clearly visible* on all material you hand in. To hand in your code you can compress it with tar/gzip with a command like:

```
“tar -czvf my.name.hw4.code.tgz my.name.hw4.code”
```

You might also try:

```
“tar -cvf my.name.hw4.code.tar my.name.hw4.code” then type “gzip my.name.hw4.code.tar”
```

This will compress the contents of the directory named “my.name.hw4.code” into a single file my.name.hw4.code.tgz. If you need more info on this, try “man tar”.

Electronic Submission of code:

You need to submit the assignment electronically via the DEN digital drop-box. Do this by accessing the DEN web page and then the DEN digital drop-box. Be sure to name your code with a name such as:

```
my.name.hw4.code.tgz
```

Be sure and substitute “my.name” with your own name. The reason for naming your submission is to make it much easier to match up projects with the students who submit them. Otherwise this task becomes difficult for the grader. When you have uploaded your code, be sure and click submit.

DEN students using electronic submission of the written part:

For DEN students handing in the written part electronically, if you use Word files, please compress the file using .zip or .rar.. Don't worry about compressing Acrobat documents since they are already compressed. Thus, I should see two files if you are doing an all-electronic submission, one with a name like bob.bobbinopolis.hw4.written.zip and the other with a name like bob.bobbinopolis.hw4.code.tgz. If you send me a file with a name like hw4.zip, I will put a hex on your credit cards. Of course this does not apply to your .cpp/.h files which should have names that work with the stubby Perl script. Also, be sure your name is on the documents you hand in since we print them up to grade them and it can be hard keeping documents matched. Please make sure all images and material for the written part are collated into one single file. Thus, *do not* send a word document and 30 image files, be sure and place them inside the word document itself.

Additionally, DEN students should remember to include a *cover sheet* with their submissions.

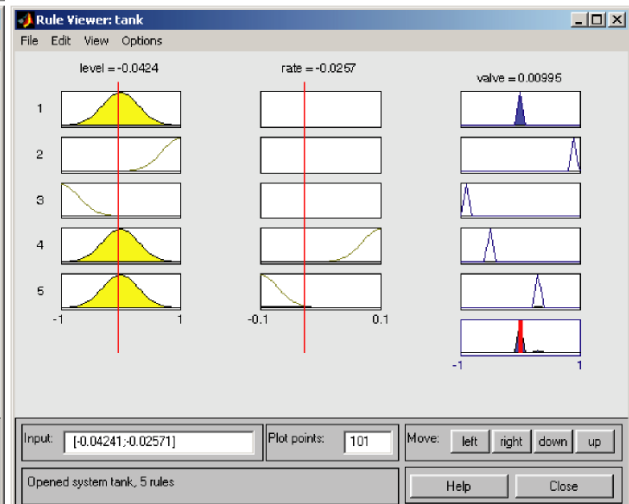
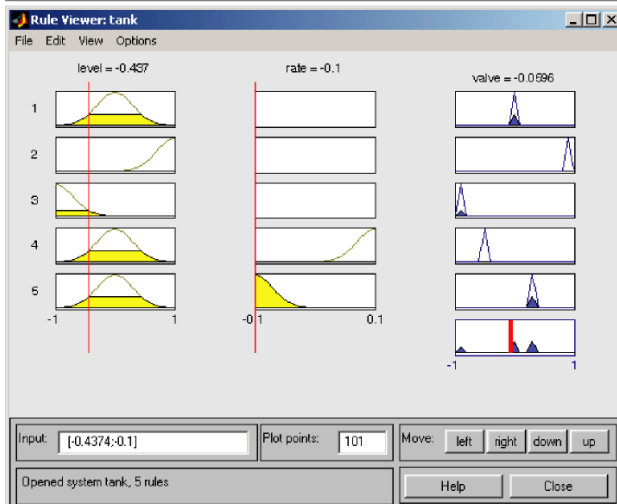
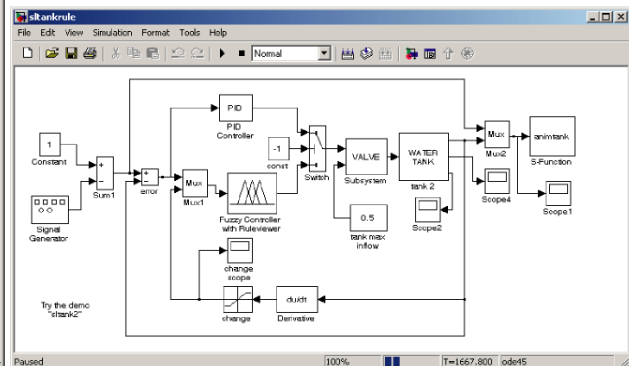
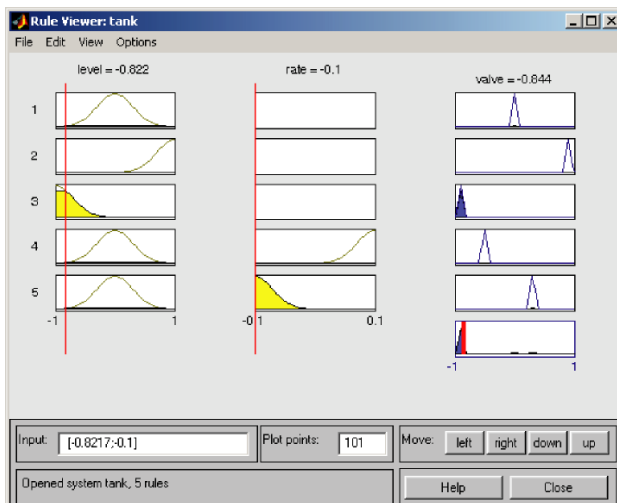
What exactly is Stubby?

This is very important. We use a Perl script to automate the grading process. Stubby is a Perl script we give to you so that you can check and make sure that your project conforms to specifications. That is, you can use Stubby to make sure that your assignment when handed in will run with our grading script. Thus, we have our own grading script *like* stubby, which we use to grade your project. By checking to make sure your project works with stubby, you make sure that your project will work with our automated grading script.

It is important to note that we will not use your version of stubby. We have our own script. As such, if you edit stubby to work with your code and not the other way around, this will not be very helpful. Additionally, since there are so many projects to grade, it is imperative that your program work with the grading Perl script. *If your program does not work with the grading Perl script you will lose points*

Question 1 (18%)

You are the manager of the Royal Canadian Walleye Conservation Group with the Eastern Great Lakes Committee. Your job is to make sure that all Walleye are happy and can swim to where they are supposed to swim. If a walleye is unhappy, then you are unhappy. To ensure that all the walleye from Lake Erie are happy, you have constructed a canal allowing them to swim to a new walleye resort you have constructed. After doing so you are approached by a representative of the Royal Canadian Moose Conservation Group with the Central Canadian Wilderness Committee. It seems that the canal you have built is preventing the moose on their natural migration route from reaching their winter-feeding grounds in Quebec from their summer grazing lands in Manitoba. They demand that you tear down your canal so that the moose can be on their way. However, you have a better idea. Having been educated at McGill, the Harvard of Canada, you thought ahead that moose might need to migrate through this area. You explain to the representative that you can meet him half way, every other day, you will let the water out of the canal so that the moose can pass over, then on the alternate days, the canal will fill back up so that Walleye can reach their resort. He's very skeptical of the idea. He doesn't believe that you can control so much water very well. You explain to him that you can control the level of water with a fuzzy logic controller. You show him this plan:



To complete this part of the assignment you will need to use Matlab (this demo is known to work on version 6.5 and higher), which can be run on Aludra or any of the public lab computers. You will also need access to the *Fuzzy Toolbox*, which may not come standard with the student version of Matlab you can purchase in the book store. A text file called matlab.readme is posted with this hand out. It will tell you how to configure Matlab to work with its license manager. Additionally, we suggest running Matlab on the Sun machines in the if you can. This seems to run smoother than via x terminal on Aludra. Please refer any questions on starting and running Matlab to ISD.

To run, in *Matlab* Click on:

Help -> Demos -> Toolboxes -> Fuzzy Logic -> Water Tank with Rule Viewer

(2 points each)

- A. What kind of membership functions are used in the fuzzy rule set for each of the three columns. Describe those functions.
- B. Describe each of the five fuzzy rule sets that relate level, rate and valve flow. In particular, show the fuzzy set memberships for measures and the fuzzy combination rules with operators.
- C. Describe how the five fuzzy rules aggregate and defuzzify to determine the rate of flow through the water valve.
- D. The representative asks if you can use standard first order logic rules for flow. What do you tell him and why?
- E. Open up the signal generator box by clicking on it and try the different types of signals (sine, square random). How do these different types of signals affect the system? How does the fuzzy controller react to them?
- F. In the signal generator box, use a square wave and set the frequency to > 100 . What happens to the water level and why?
- G. Reset the parameters to their original values, but change the max tank inflow to 50. What happens to the water level and why? How might you change the fuzzy controller to fix this?
- H. What would happen if you removed fuzzy rules 4 and 5?
- I. Name and describe two limitations of fuzzy logic.

Question 2 (Coding 60%, Related Questions 22%):

You are a researcher in Tallahassee Community College's prestigious Robotics Research Institute. You have just built the world's first robotic fly. The CIA is very interested in using it to spy on a diabolical cell of Fox News reporters. However, you must demo it for them first. Your mission (should you decide to accept it) is to design a fuzzy logic controller that will control the robot fly and demo it for the CIA. To demo your fly you will be given goal posts, which your fly must pass through. Additionally, you will be given a starting position and direction for your fly. You will then let your fly go and show the guys at the CIA that your fly can maneuver through any space.

This is the set up: You are given a position and orientation for your fly to start in. You can assume that your fly is not moving at the start. You are also *given several goal posts* to fly through. You can assume that your fly will only see the nearest set of goal posts until it flies through them, at which point it will see the next set of goal posts. You will *use a set of fuzzy rules* to determine the *speed* and *direction* of your fly so that it flies through the next set of posts.

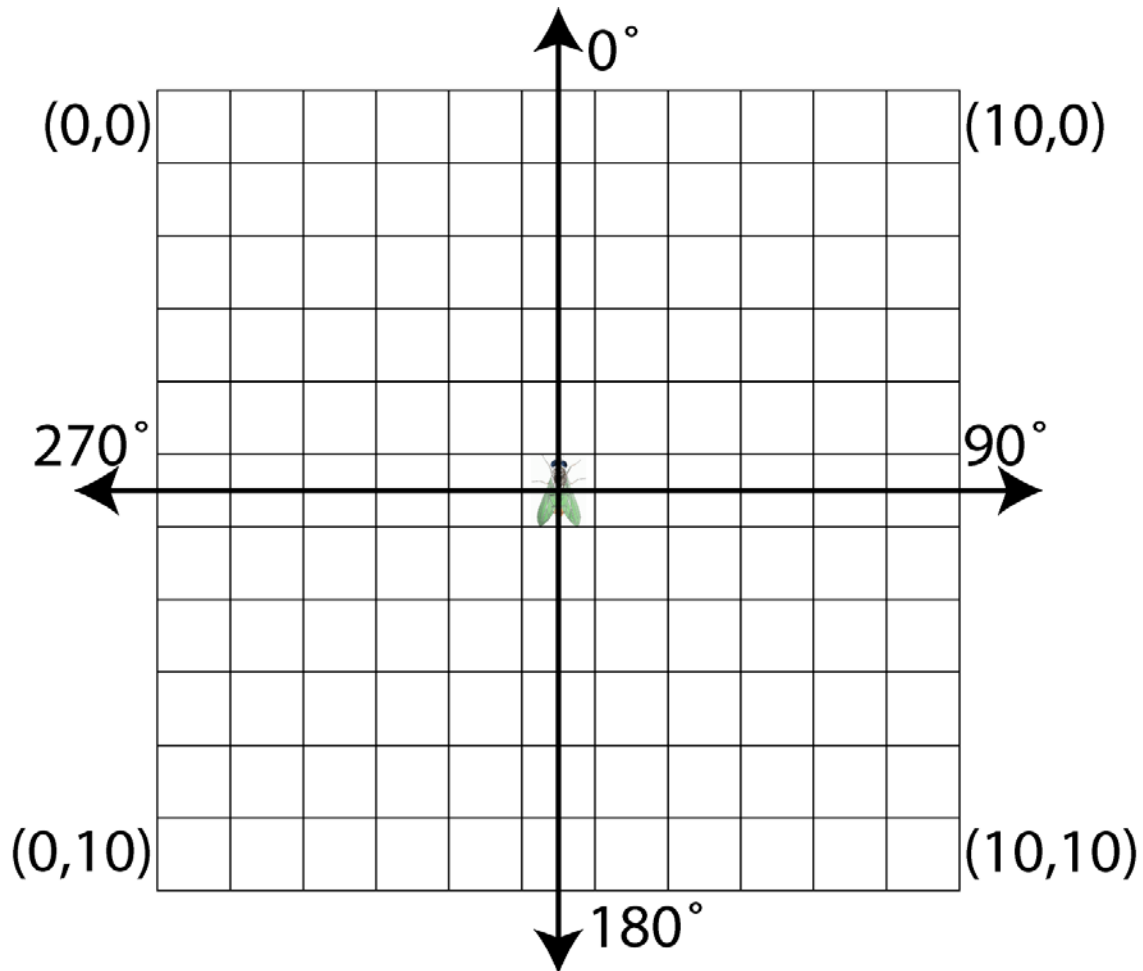


Figure 1: This is the set up for direction and heading in the arena. The top left hand corner is the 0,0 start coordinate. If a fly is pointed 0° then it is pointed towards the top of the arena.

For each set of goal posts, your fly sees two posts. Arbitrarily, one is post1 and one is post2. It is up to you which post gets which assignment. For each post, you compute the Euclidian distance to the post and the difference between the angle to the post and your current heading. Thus, you have four metrics, distance to post1, distance to post2, angle offset from post1 and angle offset from post2.

The goal posts and your fly are in an arena. The position of you fly is denoted by its spatial position. So, if your fly is at (10,6) we say its position is 10 cm to the right of the upper left hand corner and 6 cm down from the upper left hand corner.

We have put together a *suggested set of fuzzy rules*, you may change them or remove any of them you see fit if you believe it will improve the performance of your fly. You are also free to represent them with any function you think works (Sigmoid, Piecewise Linear, Gaussian, *whatever...*) The rule sets are:

```

Left(post1) & Left(post2) & far(post1) & far(post2) => turn left slowly
Right(post1) & Right(post2) & far(post1) & far(post2) => turn right slowly
Left(post1) & Left(post2) & normal(post1) & normal(post2) => turn left
Right(post1) & Right(post2) & normal(post1) & normal(post2) => turn right
Left(post1) & Left(post2) & near(post1) & near(post2) => turn left quickly
Right(post1) & Right(post2) & near(post1) & near(post2) => turn right quickly

```

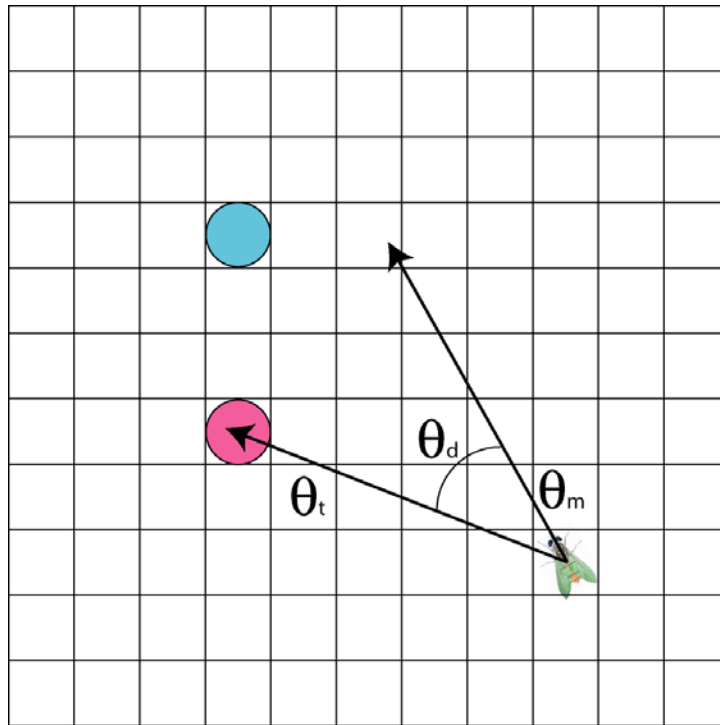


Figure 2: The fly is headed in direction θ_m . θ_t is the angle to a post and θ_d is the difference between the direction the fly is heading in and the direction to a post. You should plug θ_d into your rule set for left() and right().

If a post is to the left, then it should match with the rule Left(). Thus, if your heading is 75° and a post is at 55° from your heading, then it is to the left by 20° . Far(), normal() and near() are

instances of how close you are to a post in Euclidian distance. You should take these rules and aggregate the direction your fly should travel in. Notice that with these rules, as you approach a set of goal posts, one post will be to the left and the other to the right. Thus, in this situation, left() and right() should cancel each other out so that your fly travels straight through the posts.

For velocity, you might try the rules:

```
near(post1) & near(post2) => fly slowly
normal(post1) & normal(post2) => fly average
far(post1) & far(post2) => fly quickly
```

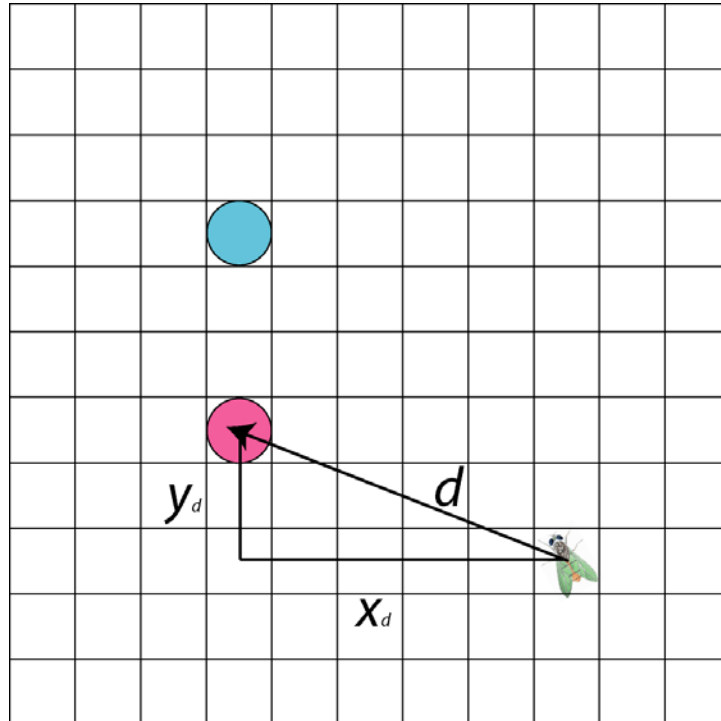


Figure 3: The fly is d distance from a post. This is just the Euclidian distance computed from the x_d and y_d position difference between your fly and a post. You should plug d into your rules for near(), normal() and far().

You should aggregate these rules into a velocity. Thus, compute the Euclidian distance to a post from your current position and apply the rules near(), normal() and far().

Simulating position and velocity: This is a discrete simulation. So for each step in your simulation, you need to move your fly a step. An easy way to do this is to pretend like each step is a second. So if your fly is moving at one cm per second, then your fly will be 1 cm from its last position in the last time step. In reality, we don't care about the size of your time slice, it might be milliseconds or hours. However, taking smaller steps will help you control your fly better at the expense of running your simulation longer. Since we only care that your fly moves through the post, you do not need to compute any real metric of time or space. It is important to note that the position and heading of your fly should be in *floating points* since your fly might be

at a position such as (12.234545,30.3882). Thus, while we set up the items in the arena using integer values, your simulation will use floating-point precision.

Clearing the posts: You can assume that your fly is myopic and will only see the next set of goal posts it is supposed to fly through. Thus, once you have cleared a set of posts, reset to fly through another set. The point at which your fly is considered “through” should be something like when it is at least past the line that intersects both goal posts.

Input file: You will be given a file, which contains the starting position of the fly, its orientation and the position of each post set. You will also be given an arena size, which will not vary by much for any simulation. An example input would look like:

```
5 100 90
2
50 95 50 105
75 95 75 105
200 200
```

This means that your fly starts at position $(x,y) = (5,100)$ and pointed in the heading 90° . There are two sets of goal posts as noted in the second line. The first has two posts at (50,95) and (50,105) while the second goal posts are at (75,95) and (75,105). The total arena size is on the last line and is 200 x 200 cm. This is an easy example in which your fly is pointed directly at the center of each set of posts and if it flies straight will fly through them.

The Arena size can vary from 200 cm to 1000 cm. However, each set of goal posts will be about 10 cm in distance from each other.

Path Points: Points are given based mostly on your fly’s ability to make it between each set of goal posts in order. If your fly comes too close to a post (scrapes it) you may lose some points. As such, aim for the center. It’s less of a problem if your fly’s path is a little wobbly so long as it makes it through the posts.

Output: You will output an image, which shows the arena and your fly’s path. Additionally, your fly’s path color will reflect its speed at that position on the path. Your output will be in an image format called *ppm* (portable pixel map). It’s an ASCII file where the value of each pixel in the image is composed of red, green and blue (RGB) values. You can use a program such as *Gimp* to draw or look at ppm files. Many graphical programs such as *xv* also support viewing of ppm files. An example is posted online. See also:

<http://netpbm.sourceforge.net/doc/ppm.html>

You should make sure that you can view your output file in *xv* and *gimp*. We use PPM images here because it is just about the easiest image format to read and write.

In your output image, draw your arena black $(R,G,B) = (0,0,0)$. Each post should be solid red $(R,G,B) = (255,0,0)$. Your path will be solid blue if your fly is moving slowly $(R,G,B) = (0,0,255)$ and light blue if it is moving fast $(R,G,B) = (0,255,255)$. Thus, your fly’s speed is

reflected in the value of the green channel in your image, which should range between 0 for very slow to 255 for very fast. One way to range your speed in this way and make it visually informative is to normalize the output based on the particular simulations minimum and maximum speed. Thus, if the fastest your fly travels during the simulation is Max , the slowest is Min , and your current speed is $Speed$. Then you can create a green integer value between 0 and 255 with:

$$\text{round}\left(\frac{Speed - Min}{Max - Min} \cdot 255\right)$$

Here `round` is just the rounding function. You can determine Max during each simulation by just taking the maximum speed value observed during that particular simulation. Thus, Max is relative and not absolute across simulations. Min is computed in the same way as Max but for the minimum value instead.

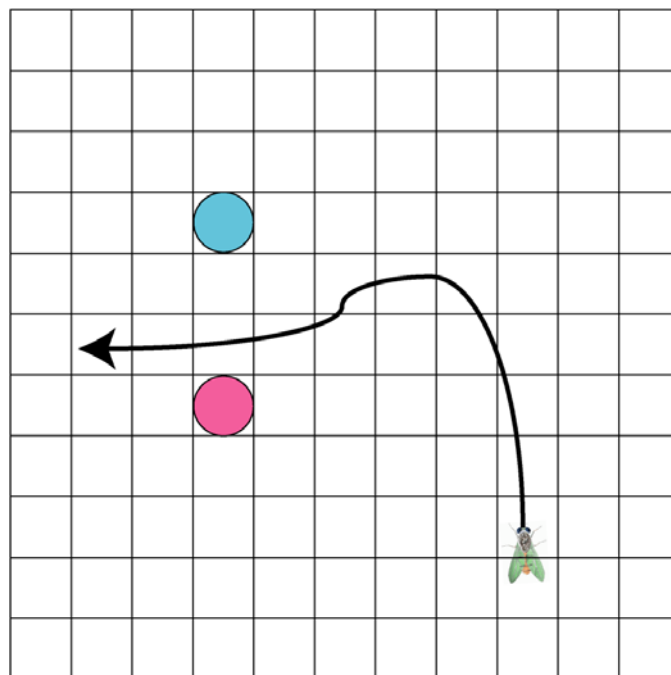


Figure 4: Your fly's path should look something like this. It should start out in some position pointed in some direction and move through the posts. It doesn't matter if the path wobbles a little. At this point, the main concern is that the fly actually passes through each set of goal posts. However, keep in mind that a large wobble will most likely prevent your fly from working correctly.

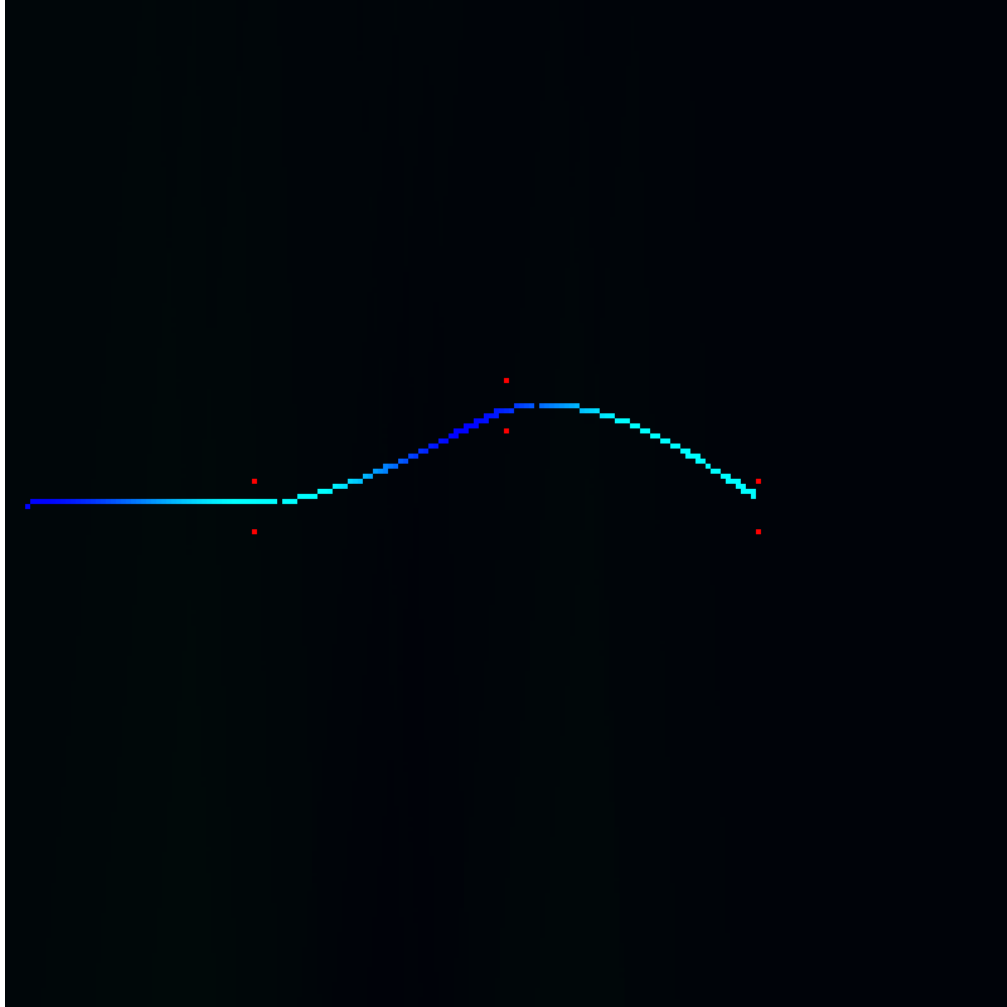


Figure 5: Your fly’s output image should look *something* like this. The posts are in red and your fly’s path is given with its speed. Don’t worry if your speed and path are not the same as this example and in general, everyone’s output should look a little different.

Naming:

You should name your image file in same way you have named your output files for the previous homework assignments. Thus, if the input file is *example.txt* your output file should be called *example.solution.ppm*.

Compiling:

This is the same as in the previous projects. There will be a very simple stubby to make sure naming conventions are adhered to.

New Stubby “revision” Function:

In addition to what stubby has been doing until now; we have added another function so that stubby creates a “revision” file every time you run it. This just creates a file with the last time

you ran stubby and what revision number it recorded. Submit the file *revision.txt* with your other code files. Students are sometimes unsure if we have received the latest version of their code, we can use this file to help make sure we received the final revision of your code. It is advised that you save your last revision number for your own records as well.

Questions:

(11 points each)

- (A) Draw the functions for your fuzzy rule set which you used in your final solution. This should look something like rule set from the water tank example. Make sure to account for all your rules. Also make sure that it is obvious how your rules interact and derive the final output. Since this is worth 11 points it should look *very* spiffy.
- (B) Explain for each rule set, what it does and how it effects your simulation.