CS100R Assignment 6

out: 6 AM, November 12, 2007.
due: **6PM, November 30th, 2007.**

1 Introduction

Recall from lecture that interpolation involves figuring out a value at a point where you don't have data, but inside the range where you do. In the last assignment we found a line of best fit for distance vs. time data from the robots. This line of best fit would be useful if we wanted to interpolate where the robot might have been at any time during the interval over which we collected data. We would simply have taken the time we wanted and plugged it into the equation for the line of best fit to get a distance. This would have given us a very reasonable guess despite the fact that the robot did not give us continuous data.

Assignment 6 has two parts. In the first part we will learn about a few methods for morphing images. Because there won't always be a nice one-to-one mapping of pixels from the original image to the new one, we will need to use interpolation to decide what value to give pixels in the new image. In the second part we will be playing a sort of "guess who" game with the robots.

2 Images in Matlab

Before we can let you loose on this assignment, we must make a digression into a few ugly Matlab topics pertaining to images and the way they're represented.

Though you may not know it, you've probably already encountered one strange aspect of Matlab: its *data types*. By default, when Matlab sees a number, it interprets it as a *double* – a positive or negative real number with a fixed number of potential digits before and after the decimal point. Another data type that you've encountered is the *logical* data type, which represents a value that is either 0 or 1 (false or true). And, as you may have found out, Matlab uses logical values in Boolean expressions and in binary images. Also of interest to us is the *uint* data type, which represents an unsigned (non-negative) integer. It is possible to specify the number of bits that Matlab will use to represent the integer by specifying `uint8`, `uint16`, etc. Logically, the value of a `uint8` must fall in between 0 and 255 (00000000 and 11111111 in eight-bit binary form), and the value of a `uint16` must fall between 0 and 65535. There are also data types for signed integers, characters, Java objects, and more, but they are beyond the scope of this assignment.

Images are represented in Matlab by arrays of intensity values. Binary images are particularly simple: they're just matrices with logical-valued entries corresponding to pixels. Grayscale images are the same, but with double-valued or integer-valued entries. In the former case, an entry of 0.0 indicates that a pixel has zero intensity, and an entry of 1.0 gives a pixel full intensity; in the latter case, an entry of 0 gives a pixel zero intensity, but the value indicating full intensity is the maximum value for

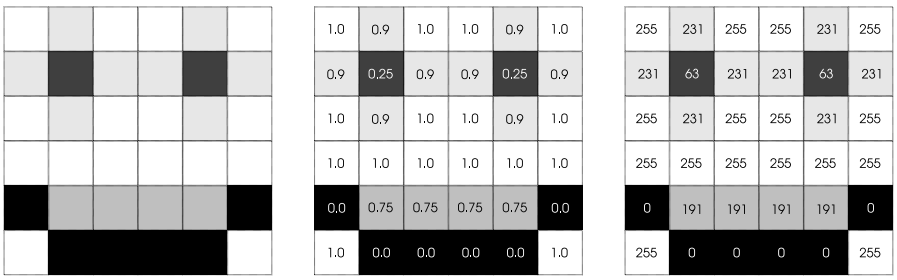


Figure 1: Images in Matlab can be represented using a variety of data types. Pixel intensities fall in the range $[0, 1] \in \mathbb{R}$ *when the data type is* `double`*, and in the range* $[0, 255] \in \mathbb{Z}$ *when the data type is* `uint8`*.*

the particular integer data type (e.g., 255 for `uint8`). An example greyscale image is presented in Figure 1.

The issue becomes somewhat more complicated when color is involved. Matlab supports many different techniques for representing color in images, but we will be solely concerned with RGB. Unlike a greyscale image, which gives one intensity value to each pixel, an image in RGB format gives three intensity values to each pixel: one for red, one for blue, and one for green. We say that such an image has three color *planes*. In Matlab, this is represented with a three-dimensional array, where the first two dimensions index the pixels in the image, and the third indexes the color plane (1 = red, 2 = blue, 3 = green). Unfortunately, as elegant as Matlab syntax is for handling two-dimensional arrays, it is surprisingly unpleasant for three-dimensional ones. You can set and access the color values for a pixel independently without much trouble:

```
image(i,j,1) = 50;
image(i,j,2) = 25;
image(i,j,3) = 75;
```

However, if you'd like to set all three color values of a pixel at the same time, your only option is to do something like this:

```
image(x,y,:) = reshape([R G B], 1, 1, 3);
```

You can look up Matlab's `reshape` command for details if you're curious. You may use whichever method you like; however, be careful where you use `reshape`, as it can slow down your code quite a bit if you're not careful!

Like their greyscale counterparts, color images can come in a number of different data types. We load images from files and cameras using the `uint8` data type. However, you may use the `double` data type for your images if you find it more convenient. You can easily convert between data types using the `double` and `uint8` functions in the following manner:

```
doubleImage = double(uint8Image);
uint8Image = uint8(doubleImage);
```

But remember, the intensity values will fall in different ranges *depending on the data type*. If you convert an image from `uint8` to `double` without rescaling the intensity values, you'll produce some strange results. Matlab is very strict about types and will yell at you if you try to mix them up very much, so don't be surprised if you run into errors if you're not careful about your conversions.

One other thing you should keep in mind when you're working with images in this assignment is that the y-axis of an image in Matlab is inverted. Matlab indexes the image like a matrix, so increasing values of i move you down in the image, rather than up. This has some slightly annoying consequences when performing geometric operations on images.

3 Image morphing

3.1 How to demo your work for credit

For credit, please write a demonstration function for each morphing algorithm you implement. The demonstration functions should display the original image and the new image side by side. They should take arguments based on the type of transformation they demonstrate. So for example the header for the centering function might look something like this:

```
function [ ] = demoCenter(image, centerPixel)
```

How you go about displaying images "side by side" is up to you. Showing them next to each other in the same figure window is possible if you make a big image and copy the two images into it. You could also accomplish this by manually setting parameters on the figure and the axes for the image plots, but we don't recommend trying (if using `timer` in the last assignment was an adventure, this is a horror mystery). Another option would be simply to display the image in two separate `figure` windows. You can use the `figure` command to accomplish this, bearing in mind that `imshow` draws an image in the most recently opened figure window (or creates one of its own if none is open).

3.2 Horizontal Inversion

To get used to moving pixels around, we'll start with a horizontal inversion. Given an image, write a function which creates a new image of the same size with the left and right sides reversed. That is, flip the image from left to right.

3.3 Centering an Object

Write a function `center_student` that takes three arguments: an image and the coordinates of a pixel that will become the center of the image. The function will return a

larger image with the center pixel at its center, the old image in one corner, and black pixels filling the space outside the original image.

Note, the center pixel need not have an integer location, so for example a valid center pixel might have coordinates (30.43, 41.2). It might be tempting to just round the center pixel and shift the image by an integer amount, but this will be less useful to you when you do the rest of the assignment.

You can accomplish such a fractional translation of the image using "reverse interpolation." Thas is, we will get the value for each pixel in the new image by looking at the old one (as opposed to looking at each pixel in the old image and sticking it in the new one). First, make an array that is the correct size for the new picture (how big does it need to be so that you don't lose any information?) Next, find an equation which, given the location of a pixel in the new image, computes the position of the corresponding pixel in the old image – this position may be fractional. Then, for each spot in the new image, use the equation to figure out where in the old image each pixel's value should come from. If the location from which you are taking the pixel's value is not in the old image, make the pixel black; otherwise, use interpolation to estimate what the value of the pixel should be.

In lecture we learned about bilinear interpolation. Use this method to find the value of a pixel in the new image by looking at the nearest four pixels to its location in the old image. You may want to write a helper function and use it to interpolate values for all the pixels in the image. While we do recommend writing a helper function, you should note that calling a function in Matlab is extremely slow. If you do write a helper function to do bilinear interpolation on a pixel-by-pixel basis, you can expect performance to be cut by a factor of approximately two. We will discuss various ways to optimize Matlab code in section.

3.4 Stretching

Write a function `stretch_student` which takes three arguments: an image, a vertical stretch factor, and a horizontal stretch factor. It will return a larger image which is a stretched version of the original image.

The stretch factors should be percentages of the original size. So giving the function a horizontal stretch factor of .50 should yield an image half the width of the original image.

3.5 Rotation

Write a function `rotate_student` which takes two arguments: an image, and an angle in degrees. This function will rotate the image counterclockwise around its center point by the given amount.

Computing the coordinates of a rotation in the counterclockwise direction can be viewed as an angle addition (Figure 2) if we make the simplifying assumption that we're rotating about the origin (we will revisit this in a moment). We're interested in in the math for rotating in the clockwise direction, since we'll be doing reverse inter-

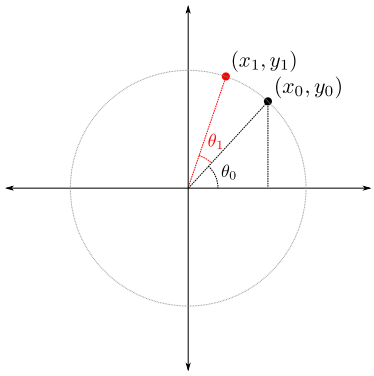


Figure 2: Rotating (x_0, y_0) by θ_1 to obtain (x_1, y_1) can be viewed as an angle addition. By the same token, given a point in the rotated image (x_1, y_1), computing its source $(x_0, y_0$ in the original image can be viewed as an angle subtraction.

polation. Recall that the angle addition identities for sine and cosine can be stated in the following way:

$$\begin{aligned}\sin(\alpha+\beta) &= \sin\alpha\cos\beta + \sin\beta\cos\alpha,\\ \cos(\alpha+\beta) &= \cos\alpha\cos\beta - \sin\alpha\sin\beta.\end{aligned}$$

Now consider that the source and destination points both fall on a big circle with radius $r = \sqrt{x_1^2 + y_1^2}$. We may therefore express our reverse rotation using the following trigonometric relations:

$$\begin{aligned}x_1 &= r\cos(\theta_0+\theta_1),\\ y_1 &= r\sin(\theta_0+\theta_1),\\ x_0 &= r\cos\theta_0,\\ y_0 &= r\sin\theta_0.\end{aligned}$$

Since we only know the quantities θ_1 (the angle we're rotating by) and $\theta_0+\theta_1$ (the angle of the point in the rotated image), we rewrite the equations for (x_0, y_0) (the points in the source image) in the following way:

$$\begin{aligned}x_0 &= r\sin(-\theta_1 + [\theta_1+\theta_0]),\\ y_0 &= r\cos(-\theta_1 + [\theta_1+\theta_0]).\end{aligned}$$

Applying the angle addition identities, we obtain

$$\begin{aligned}x_0 &= r\cos-\theta_0\cos(\theta_1+\theta_0) - r\sin-\theta_1\sin(\theta_1+\theta_0),\\ y_0 &= r\sin-\theta_0\cos(\theta_1+\theta_0) + r\sin(\theta_1+\theta_0)\cos-\theta_1.\end{aligned}$$

Of course, since $\cos-\alpha = \cos\alpha$ and $\sin-\alpha = -\sin\alpha$, we get the following:

$$\begin{aligned}x_0 &= r\cos\theta_1\cos(\theta_1+\theta_0) + r\sin\theta_1\sin(\theta_1+\theta_0),\\ y_0 &= -r\sin\theta_1\cos(\theta_1+\theta_0) + r\sin(\theta_1+\theta_0)\cos\theta_1.\end{aligned}$$

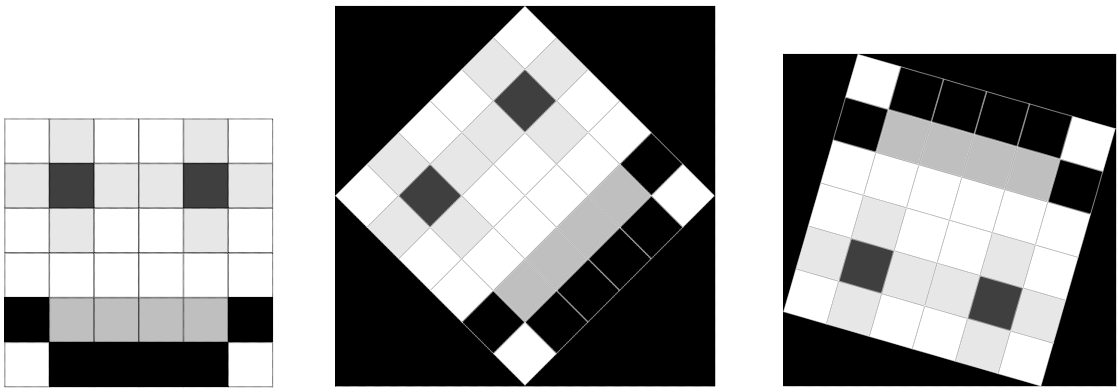


Figure 3: The dimensions of a rotated image may vary from the those of the original.

Substituting $x_1 = r\cos(\theta_0 + \theta_1)$ and $y_1 = r\sin(\theta_0 + \theta_1)$, we get that

$$\begin{aligned} x_0 &= x_1\cos\theta_1 + y_1\sin\theta_1, \\ y_0 &= -x_1\sin\theta_1 + y_1\cos\theta_1. \end{aligned}$$

Now we can adjust these equations for the fact that we're rotating around the center of the image rather than the origin by subtracting the center point's coordinates from each point (with (c_{x0}, c_{y0}) being the center point in the source image and (c_{x1}, c_{y1}) being the center point in the return image, which may have different dimensions than the input image), yielding

$$\begin{aligned} x_0 &= (x_1 - c_{x1})\cos\theta_1 + (y_1 - c_{y1})\sin\theta_1 + c_{x0}, \\ y_0 &= -(x_1 - c_{x1})\sin\theta_1 + (y_1 - c_{y1})\cos\theta_1 + c_{y0}. \end{aligned}$$

There is yet another issue we must account for: y-axis inversion. As we mentioned above, since Matlab represents images using matrices, and the i-indices start from the top of the matrix, the y-axis of the image is inverted with respect to the Cartesian y-axis we assumed in our calculations. We therefore must subtract our y values out of the total height h of their respective images, which has the following effect (remember that $h_0 - c_{y0} = c_{y0}$):

$$x_0 = (x_1 - c_{x1})\cos\theta_1 - (y_1 - c_{y1})\sin\theta_1 + c_{x0}, \quad (1)$$

$$y_0 = (x_1 - c_{x1})\sin\theta_1 + (y_1 - c_{y1})\cos\theta_1 + c_{y0}. \quad (2)$$

We must also be careful that our output image is big enough to hold the rotated image. As shown in Figure 3, the output image's dimensions will usually be bigger than the input image's dimensions. You can use equations (1) and (2) to solve for the minimal dimensions of your output image.

4 Who moved?

Imagine that you're working on a RoboCup team and you're responsible for writing the code which connects the overhead camera with the robots' onboard cameras. At

some regular interval, you grab one frame from each camera, and you'd like to know which robot is which on the overhead camera display so you can estimate its position on the field or tell if someone's sneaking up behind it.

Let's start out by making some simplifying assumptions. First, we'll guarantee that between any two successive frame grabs, no more than one robot changes its position or orientation. Second, we'll require that all of the robots be in view of the overhead camera at all times. This simplified problem can be easily addressed using *image differencing*.

4.1 Getting frames from the A6 library.

Since this is CS100R and not the real RoboCup, we'll be using Roombas (sucking up the soccer ball with a vacuum cleaner is probably against the RoboCup rules anyway). We've built up a small library of image data which contains several sequences of frame grabs from an overhead camera and a number of robots on the ground (See Figure 4 for an example set of frames). You can access this data using the following two functions:

1. `getA6Info` returns information about a particular sequence of grabs. You call it with the sequence number (there are currently five sequences you can use) and it returns the number of robots on the floor and the number of sets of frames in the following way:

   ```
   [numRobots, numSets] = getA6Info( sequenceNumber );
   ```

2. `getA6Frames` returns a set of frames given a sequence number and an index into the sequence. It returns the frames in a special Matlab data type called a *cell array*. Cell arrays are like regular arrays, except you can put whatever you want inside of them (try putting a Java object and an image into one regular array!), and you access the elements using curly braces ({ and }) rather than parentheses. The first element in the returned array is always the image from the overhead camera. The subsequent images are from the onboard cameras of the robots. You might handle this in the following way:

   ```
   F = getA6Frames( 4, 1 );

   overheadImg = F{1};
   for i = 2:length(F)
       robotImage = F{i};
       doSomething(robotImage);
   end
   ```

4.2 Imaging differencing

Now that you know how to get frames out of a sequence, let's return to the problem

Figure 4: An example set of frames from the A6 data collection. The first one is from an overhead camera, and the remaining images come from onboard robot cameras. (Yes, that is a rather embarrassing picture of me with my shirt hanging out.)

of figuring out which robot is which. We can pull two successive sets of frames out of a sequence and compare them using image differencing to isolate the motion of one robot. To get a difference image, we simply subtract two images – sounds easy, right? Unfortunately, Matlab data types rear their ugly heads again. The images come out of the cameras in `uint8` format, which has range $[0, 255] \in \mathbb{Z}$. If we subtract two intensity values, say, $40 - 60$, Matlab will clamp the result to the range of the data type, returning 0. To preserve all information, we need to convert the image to `double` format. Remember that images in `double` format have intensity values in the range $[0, 1] \in \mathbb{R}$, so the code to convert the images and compute their difference image might look something like this:

```
doubleImage1 = double(uint8Image1) / 255;
doubleImage2 = double(uint8Image2) / 255;
diff = doubleImage1 - doubleImage2;
```

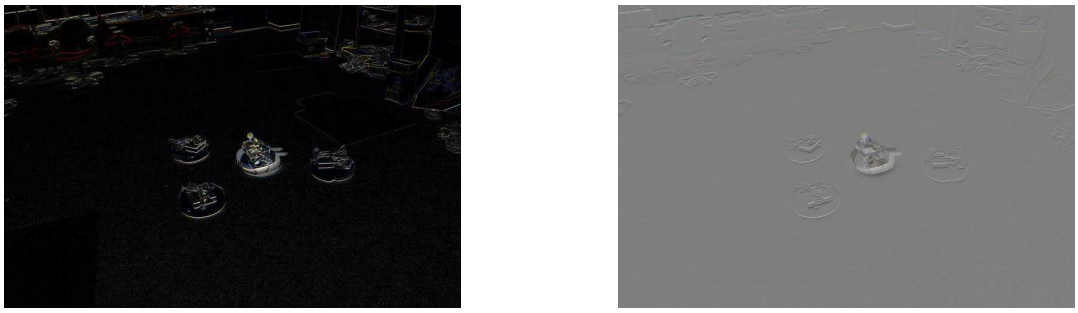
This produces an output image like the one shown in Figure 5.

Intuitively, we'd like to apply the same techniques to the difference image that we applied to wand images in previous assignments: threshold the difference image, find a big blob, and conclude it's the robot. We might try to threshold the difference image in the following way[1]:

```
threshImg = double(abs(diff) > 0.2);
```

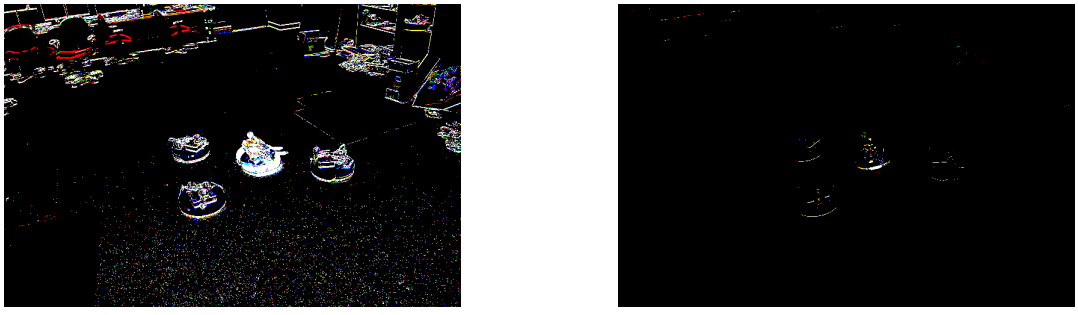
The problem with this is that there's so much noise in the difference image that it's hard to isolate the robot (see Figure 6).

[1]Can you figure out why it's convenient to convert the result of the boolean operation to a `double` again? What does Matlab say if you try to plot a `logical` image with three channels?



(a) (b)

Figure 5: A difference image. There's no logical way to display negative intensities, so we must map the values into the positive $[0, 1]$ *range before displaying. The image in (a) is obtained by taking the absolute value of the difference, and the image in (b) is obtained by centering values around* 0.5 *and squashing them into* $[0, 1]$.



(a) (b)

Figure 6: Just thresholding the difference image yields results that are too noisy, no matter what the thresholding value is: (a) was thresholded at 0.1, and (b) was thresholded at 0.6.

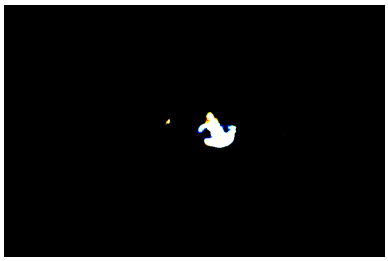


Figure 7: This is the result after blurring the difference image with `blurBoxFilter(abs(diff), 15)` *and then thresholding it at 0.1.*

This problem is rather like the dense box problem we formulated earlier this year. We need some way of capturing the density of the pixels at each point. One way to simple and straightforward way to encode this information is to blur the image. More specifically, we average the value of each pixel with the values of some number of neighboring pixels.

This operation is a specific kind of *convolution*, and Matlab is particularly fast for these operations, so we've provided a function `blurBoxFilter` which uses Matlab's built-in functionality to blur an image. It takes as arguments an image and an integer which specifies how big of a region around each pixel the function should include in the average. Thresholding after blurring produces much cleaner-looking results, as shown in Figure 7.

So we can now blur and threshold an image, and, if we use the built-in command `im2bw` to collapse the red, green, and blue channels of the image into one, we can do this entire operation in one line:

```
out = im2bw(double(blurBoxFilter(abs(diff), 15) > 0.1));
```

This leaves us with the same kind of binary image we've been working with all year. We can find the biggest connected component and compute its median point and bounding box or convex hull. These are all logical ways to represent its position and shape, and they'll all give us a good idea of where the robot is in the image.

4.3 So who moved?

So now that we've got a way to locate a moving robot in an image, we just need a way to figure out which robot our blob corresponds to using the images from the onboard cameras. Since at most one robot has moved, we can again look at difference images: the difference between the images grabbed from the moving robot should be great, while the difference for the other robots should be fairly small. Like above, we'd like to filter out the noise and threshold, but we no longer care about finding the biggest connected component, since features will have moved around all over the image. We can guess which robot moved by counting the number of ones in the thresholded dif-

ference image for each robot and picking the one with the most.

4.4 Getting credit

1. Write a function `whoMoved` which takes two sets of frames as input. The function will compare them using the techniques we discussed, and return three things: the number of the robot that moved (the first robot has the second image in each set, the second has the third, and so on), the robot's estimated position in the overhead image, and the estimated convex hull of the robot.

2. Write a function `watchRobots` which takes a data sequence number, looks it up in the library, and then reads all of the frames out of the library for the sequence, running `whoMoved` on each pair and keeping track of each robot's last known position. At every frame, it should display everything it knows about the robots and their positions, as shown in Figure 8.

   You can plot nice-looking text labels on your images in the following way:

   ```
   % Name the robot who moved.
   name = sprintf('robot %d', r);

   % Plot the current robot (what we last knew about it).
   % More info using doc text
   text(robotX(r), robotY(r), name, ...
       'HorizontalAlignment', 'center', ...
       'VerticalAlignment', 'middle', ...
       'BackgroundColor', [0 0 0], ...
       'Color', [1 1 1], ...
       'EdgeColor', [1 0 0]);
   ```

   If you're curious, you can learn about all of the crazy things you can do with the `text` command by running `doc text` – the link at the bottom to "Text Properties" is particularly useful.

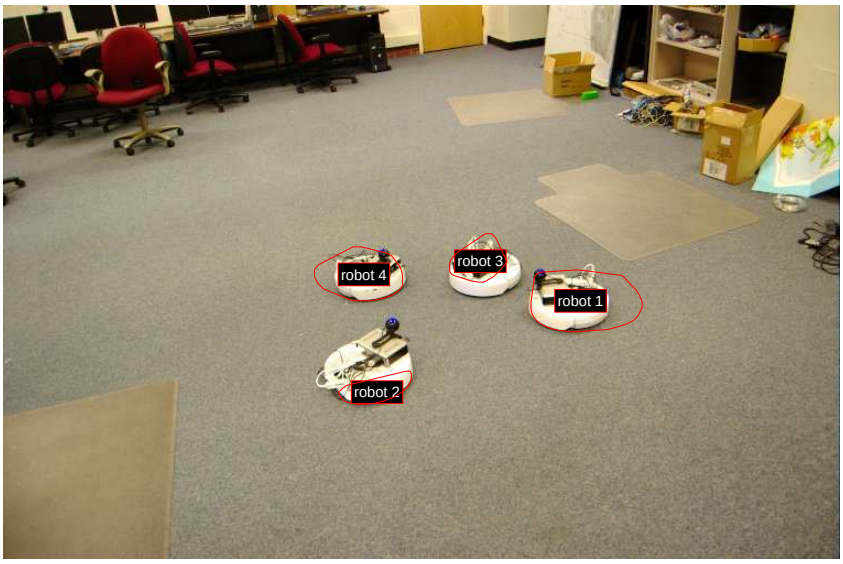


Figure 8: Output from our reference implementation of `watchRobots`*. It's not critical that your function produce perfect results (it may not even be possible), but it should be fairly accurate.*