TEAM GUNDAM

Project 2

Visual Servoing

Larry Field, Farzon Lotfi, & Alex Strange

3/5/2011

Part 1:

There were two main tasks in this part of the project. First, identifying the robot's position, orientation, and the fruit locations. Second, calculating the best path and the rotations needed in order to get the robot to reach the lemon. For each task, we then thought up a few possible implementations. For the robot detection, we tried to use the six blue running lights on top of the Rovio to determine the robot's position and orientation. This could be done using the known relative distances between these points. The longest distance between any of the lights on the robot was the diagonals; using this we could construct a rectangle that represented the outer edges of the robot. The front of the robot was the shortest side of the rectangle. Using the center of the rectangle, we draw a line from the center of the square through the midpoint of the front line. This worked very well, and could calculate the orientation accurately to within a few degrees. However, a common problem was that we could only detect five of the six lights on the robot. This still allowed us to construct the overall position of the robot, but unfortunately, it often caused the orientation of the robot to be detected at about twenty to thirty degrees off.

This error could cause longer times to find the lemon because sometimes our path planner would correct for orientations that didn't need correcting. Because of this, we decided to change our detection method by attaching blue and green felt to the top of the robot. With blue on one half, and green on the other half, it became possible to detect the orientation by finding the two color blobs, and drawing a line between them. Then taking a line that is at a right angle to the line between the two color blobs. This gives a slightly larger error than the best case scenario of the six light point method, but significantly less than the average cases where we can't detect all of the lights. The average error was around 5-10 degrees using the new method. This still required periodic corrections as we got closer to our target object, but caused fewer incorrect adjustments and overcorrections.

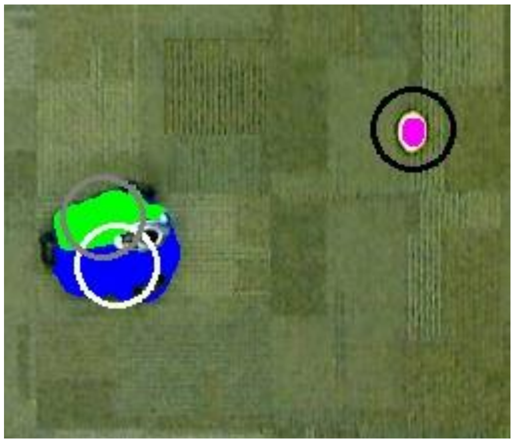


Figure 1: example of our blob detection algorithm detecting the different colored pieces of felt and the lemon.

Secondly, in order to determine the path to use, we first implemented a simple algorithm that uses the positions of the robot to determine the angle from the robot to the fruit. We would check at the next frame to see if we lined up and then proceed forward. Then you can find the difference between the orientations of the robot and the fruit. This tells you how far you need to turn to point at the fruit. Then you can merely turn this distance and then proceed forward checking the camera frames as a reference. Unfortunately, this tended to generate some problems, as we sometimes received slightly old images from the camera, so the robot would finish its turn, and we would receive an image from half way through the turn, which would cause us to calculate that we needed to keep turning, causing us to over correct. Eventually we would center in on the fruit and then proceed forward (correcting the angles periodically if the fruit either changed positions, or we were not pointed exactly at the fruit) until we had reached the fruit.

The premise of our improved algorithm was a straight line is the shortest path. We achieved this by setting our reference frame to the location of the robot. From here we use basic geometric features of circles and triangles to get position and orientation. The first step is to find the x and y location of the lemon. This line segment between the robot and the lemon is the radius. We then find the current orientation of the robot. This direction plus the length of the radius will be our current vector that we will need to translate to the point of the fruit (lemon). This vector will point at the point (Robot's x, Robot's + radius) I will refer to it as the parallel point from here on. Now that we have the point of the lemon and the parallel point, we can get the length of the cord between them. A perpendicular line from the midpoint of the parallel point and the lemon to the robot will allow us to turn this problem into a simple right triangle. Now we perform an arcsine on half the chord divided by the radius. This gives us an answer in radians so multiply by 180 and divided by π to get the answer in degrees. Then multiply by 2 to get the angle between the orientation of the robot and the fruit (lemon). Now we need to handle cases larger than 90 degrees. We can do this easily by breaking the problem into a semicircle. We now have a right side, a left, and a center (for special cases where a triangle can't be constructed). For degrees larger than 90 for either side just subtract the angle from 180 degrees. Using this Formula we get the angle of rotation and the distance of the robot from the lemon in pixel coordinates which can be converted into real-world coordinates since we know the height of the camera. Since we calculate distance on the first image we deal with lag time from the web camera a lot less. Also since we make sure our distances are slight underestimates we never run the risk of overshooting the lemon.

KEY

Robot point = rp	
Lemon point = lp	
Parallel point = pp	
Perpendicular line = pl	
Midpoint = mp	

1. $radius = distance(rp, lp) = \sqrt{(lp.x - rp.x)^2 + (lp.y - rp.y)^2}$
2. pp = Point(rp.x, rp.y+radius)
3. $chord = distance(pp, lp) = \sqrt{(lp.x - pp.x)^2 + (lp.y - pp.y)^2}$
4. $midpoint(pp, lp) = (\frac{lp.x+pp.x}{2}, \frac{lp.y+pp.y}{2})$
5. $pl = distance(rp, mp) = \sqrt{(mp.x - rp.x)^2 + (mp.y - rp.y)^2}$

$$\text{arcsine}\left(\frac{\frac{chord}{2}}{radius}\right) * \frac{180}{\pi} = \sigma \rightarrow \forall(2\sigma < 90) \Rightarrow 2\sigma \rightarrow \forall(2\sigma > 90) \Rightarrow 180 - 2\sigma$$

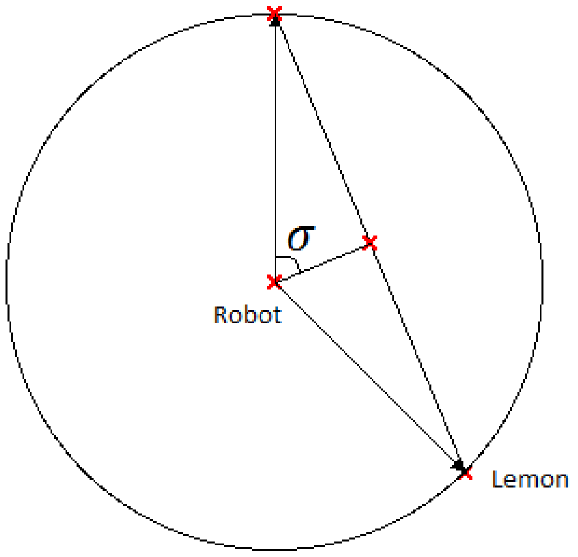


Figure 2: diagram of the formula above

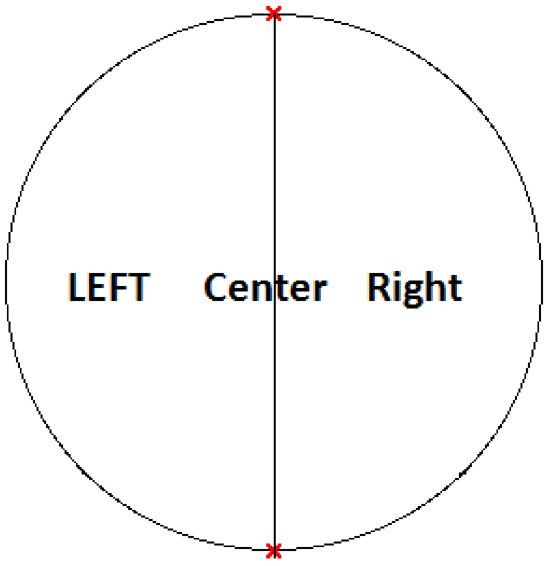


Figure 3: demonstrates how we divided the problem into two semicircles

We can compute an even more time and memory efficient algorithm (it would not matter on today's computers but would for a robot with an embedded processor). By getting the intersection of the robot's Y coordinates and the lemon's X coordinates. The resulting perpendicular line turns the problem into a triangle much faster than the above formula. All the angle computation will remain the same including dividing the problem up into two sides (ie the semicircles).

Formula (the key is the same as above):

1. $paallel\ point = (lp.x, rp.y)$
2. $hypotenuse = distance(rp, lp) = \sqrt{(lp.x - rp.x)^2 + (lp.y - rp.y)^2}$
3. $adjacent\ line = distance(rp, pp) = \sqrt{(pp.x - rp.x)^2 + (pp.y - rp.y)^2}$

Step 4 is not needed to solve this problem but it helps to think about the problem in terms of a triangle.

4. $opposite\ line = distance(lp, pp) = \sqrt{(pp.x - lp.x)^2 + (pp.y - lp.y)^2}$
5. $\arccos\left(\frac{adjacent\ line}{hypotenuse}\right) * \frac{180}{\pi} = \theta$ if $\theta > 90 \rightarrow \theta = 180 - a$

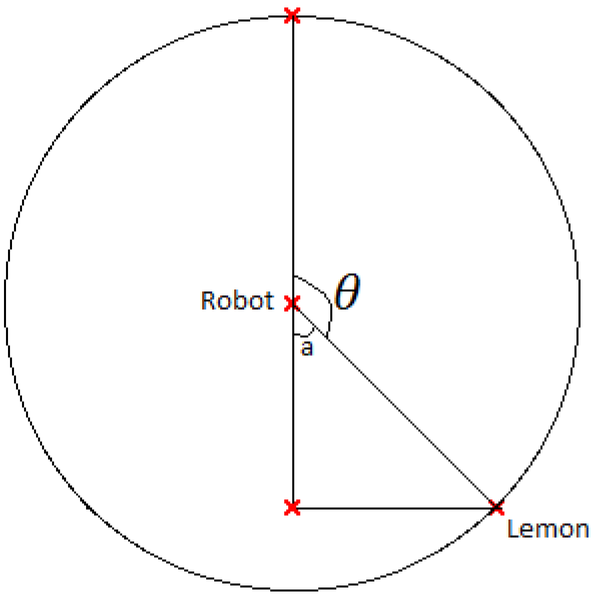


Figure 4: a visualization of the above formula. a = θ if $\theta \leq 90$

The algorithm that probably gave us the best improvement was the switch from the six light point detection to the generic color blobs. This prevented many cases where the robot would keep trying to correct its position even when it did not need to because one or two of the six lights would not be detected. These almost continuous unnecessary corrections would lead to much longer search times for the fruit. Once we removed this, the required number of corrections were much fewer, reaching the fruit much more quickly.

Given more time, we would probably have tried to perfect the point matching algorithm by using a threshold to find every point that *might* be a light on the robot, then using the known pattern of the lights to find the most likely position and orientation of the robot. This could be done by assigning every possible light point on the robot to every possible light source in the image, and find if there was a rotation around that point where the pattern would best match other possible light sources in the image. This would allow us to filter many more possible points, and find the ones that are most likely to be the robot. This method would give us one of the most accurate representations of the robot, which would make pathing both easier and more reliable.

If we had better control ability, we would probably have improved this algorithm by calculating the complete path (including appropriate distances using homogenous transforms) between the robot and the fruit. Then we would figure exactly how far to turn and move, and how to measure it using the optical encoders on the wheels. With that we would send the robot the entire path, and tell it to travel it on its own, using the internal optical encoders to ensure it was on its path. Once it had started its path, we would not recalculate a path unless we detect it was significantly off the path, it finishes its path, or we detect the lemon being significantly moved.

Basic algorithm

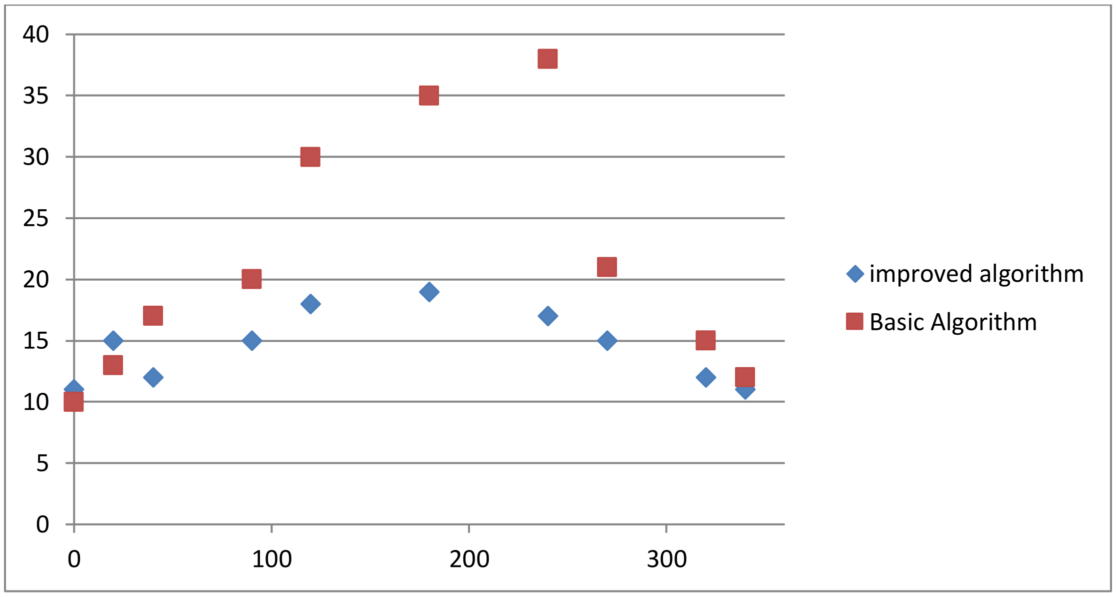
Difference in angle the robot is facing and the angle from the robot to the lemon. Zero means the robot is pointed straight at the lemon:	Time (approximate number of seconds):
0	10
20	13
40	17
90	20
120	30
180	35
240	38
270	21
320	15
340	12

The ones where it had to turn the most (considering that it always turns the shortest angle) generally had the longest times to reach the goal, because it would often over turn, because of the delay in receiving images. Sometimes the image received would be from before the robot finished the turn, causing our algorithm to think that the robot needed to turn some more. This usually wasn't a problem for the shorter distances (even the ones as large as 90 degrees), because the camera wouldn't update during the turn. However for the greater turning angles, this could become a serious problem where we could way overshoot our intended angle. Our algorithm that only turned a small amount at a time significantly reduced these times because it allowed us to wait for the most recent image, and thereby avoid overcorrection.

Improved pathing algorithm. The main difference is we would only rotate a short segment at a time before attempting to recalculate our desired orientation.

Difference in angle the robot is facing and the angle from the robot to the lemon. Zero means the robot is pointed straight at the lemon:	Time (approximate number of seconds):
0	11
20	15
40	12
90	15
120	18
180	19
240	17
270	15
320	12
340	11

We generally had better times with this algorithm, because we rarely over corrected. The main difference in time was merely the number of times we had to correct our path as we moved either due to minor orientation miscalculations, or our robot tending to drive slightly to the left, and getting further and further off angle as it drove.



2.1. Rovio Guard Questions

1. What is the most significant difference between overhead vision and onboard vision?

The most significant difference between the overhead camera and the Rovio's onboard camera is the limited field of view. Because the camera is on the front, it naturally limits vision to what happens to be ahead of the camera, whereas the overhead camera can see objects behind the robot. Furthermore, Rovio's FOV is narrow even in one direction, which makes it easy to lose sight of an object - if it's not in the center of view, driving forwards may drive past it, and raising or lowering the camera head may lose it as well.

A secondary problem we found is motion blur in the camera output; it was necessary to ignore camera frames for up to several seconds after issuing a movement, because the frames were blurred and we were unable to find objects in them.

2. What did you have to add to your controller to get it to work in the new perceptual context?

To make more efficient use of time, and because the view from the overhead and onboard cameras was so different, we implemented them as separate programs. Comparing the implementations, the largest difference is that it does not attempt to track the actual locations of target objects. It was difficult to estimate distance of an object without stereo vision or a non-moving camera, so instead we attempt to move the robot such that the object is centered in its vision, and then move straight ahead towards it.

3. Repeat the experiments in Part I for both the simple controller and your improved controller. Repeat the plots of relative orientation vs. time. What is the improvement?

The simple controller does not have the ability to issue turn commands or commands to change the camera height - it can only drive diagonally and the camera is fixed at mid-height.

1. Straight ahead 1m: 25 sec (drove slightly to the right and lost sight)
2. Robot started right of the object, facing forward, with the object at the edge of view: 20 sec (drove slightly to the right and lost sight afterwards)
3. Robot started left of the object, facing forward: 15 sec (it seems the robot drifts right)

We changed the controller to turn towards objects on the edge of vision, and to adjust the camera height to better see its target.

1. Straight ahead 1m: 15 sec
2. Robot started right of the object, facing forward: 24 sec
3. Robot started left of the object, facing forward: 33 sec
4. Which improvement to your algorithm was most significant in giving you better results?

The automatic adjustment of camera height was the most useful improvement, as moving closer to the object appears to make a lower-height camera more beneficial for tracking.

Although turning should be an improvement over moving diagonally, it did not improve times for some orientations. Turning the robot caused the targeting to sometimes choose a new incorrect target, as it is unable to estimate how the real target will move in its view after a turn. Also, issuing turn commands to Rovio appears to make it move much more slowly than only issuing drive commands, which affected the speed at which it could reach the target even when it saw it.

5. What other improvements might you consider if you had more time?

The largest problem with the current implementation is that, given any frame with an object in it, it will probably detect something as either a lemon or a face and try to head towards it, causing it to move forwards practically all the time. It needs better abilities to reject false positives, as well as an ability to head back to its home base when it is not currently chasing something. This would avoid the problem of having to manually bring it back after it finds something.

A second addition would be a scanning ability; since it only looks forwards, it can’t see things traveling at angles to it. Occasionally turning in place would make it more likely to notice objects in other orientations to it.