CMPUT 312: Differential Drive Robot Movement Using UVS

Mehrab Mehdi Islam
mehdiisl@ualberta.ca

Summary

For this project, I designed and implemented a differential drive mobile robot capable of locating a moving to specific target point in the image space. The robot is located and tracked using 2 trackers on it. The target point is also specified in the image space using a tracker. The robot uses differential drive movement to reach the target point (i.e rotating in place than move, repeating these steps until it is within the reach of the target point). The project's inspiration was to emulate, on a small scale, the mobile robot movement using vision- which can be used to Autonomously move a robot to different target points, to pickup and drop of heavy items, reducing human labour. The project overall, had a mixed result. The robot is able to move to the target point, but it is not completely autonomous as you still have to select the points on the robot and then specify it's target. There are also some complexities which sometimes prevent the robot from ever reaching its target

Robot Design

I utilize the LEGO Mindstorms EV3 kit to implement a mobile differential drive robot. The platform utilizes 2 of the motor ports: two for the drive. The full system also includes a host computer. The system utilizes a camera tethered to the host computer via a USB cable; the robot itself is connected to the host computer via bluetooth. The necessity for a separate, more powerful machine arises from computational complexity of the computer vision techniques employed by the robot and a relative weakness of the EV3

hardware. We send data from the host computer to the robot using a client-server protocol with a socket connection

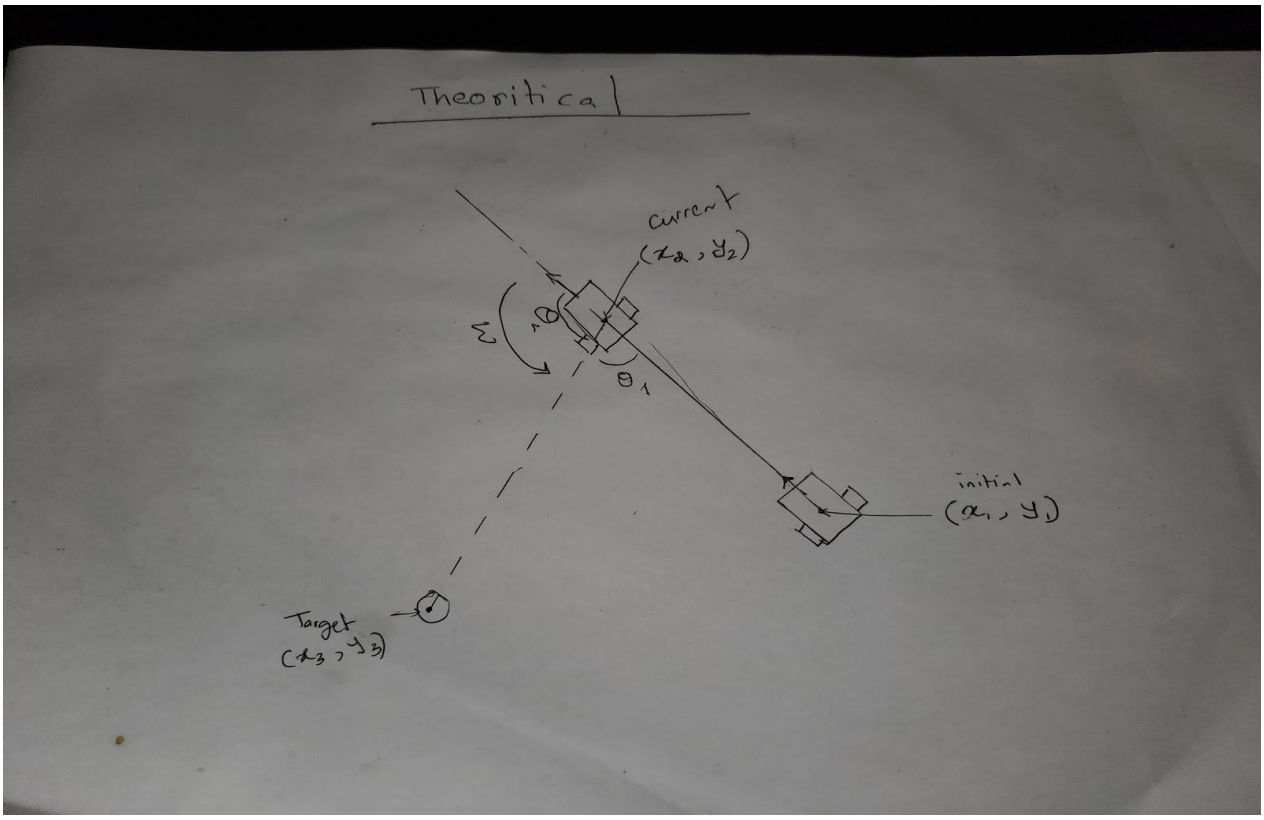
Setup

The setup of the system is quite simple. The robot is placed on the ground directly in front of the desk the camera stand of the system sits at. The camera is placed on the desk so that it has a decent sized image space where the robot can move in. The

camera is connected via USB to the computer, which has the server side programs. I tried to use my smartphone camera to get a vertical view of the image space, however the built in CV2 did not read such high resolution frames, and downsampling the frames made the KCF tracker lose track.

Inverse Kinematics Implementation

Initially I was using a Numerical Inverse Kinematics Implementation to move the robot to the specified target point. From its initial position, the Robot moves in a straight line with an arbitrary speed for an arbitrary amount of time(speed and time should be low values so that robot stays in work space) to reach a new position, recording the new position and previous position. These gives us 3 points, the current position, the initial/ previous position and the target position. Then we calculate the angle between 3 points using the cosine rule, with the current point being the angle point. We the get the angle to rotate using the angle we just calculated by subtracting it from Pi. We rotate the robot by this angle and then move in a straight line until we reach the target point. Using Visual serving, this method used a drive and look implementation. This was done by running the inverse kinematics function (d, looking if the target point and feature point is near each other using visual servoing. If they were not near each other, we would repeat the drive and look steps as specified above, till we reached the target point. This implementation however was not very flexible as you needed to specify different cases for all 4 of the quadrants in the image space, hence I opted to use a much more efficient method which relies completely on visual servoing



Tracking

Using Visual Servoing, I tried various different trackers, starting with an arbitrary colour tracker implementation i found online to all the 8 different trackers which is in Cv2.

Here is a description of what i got from testing each of the trackers

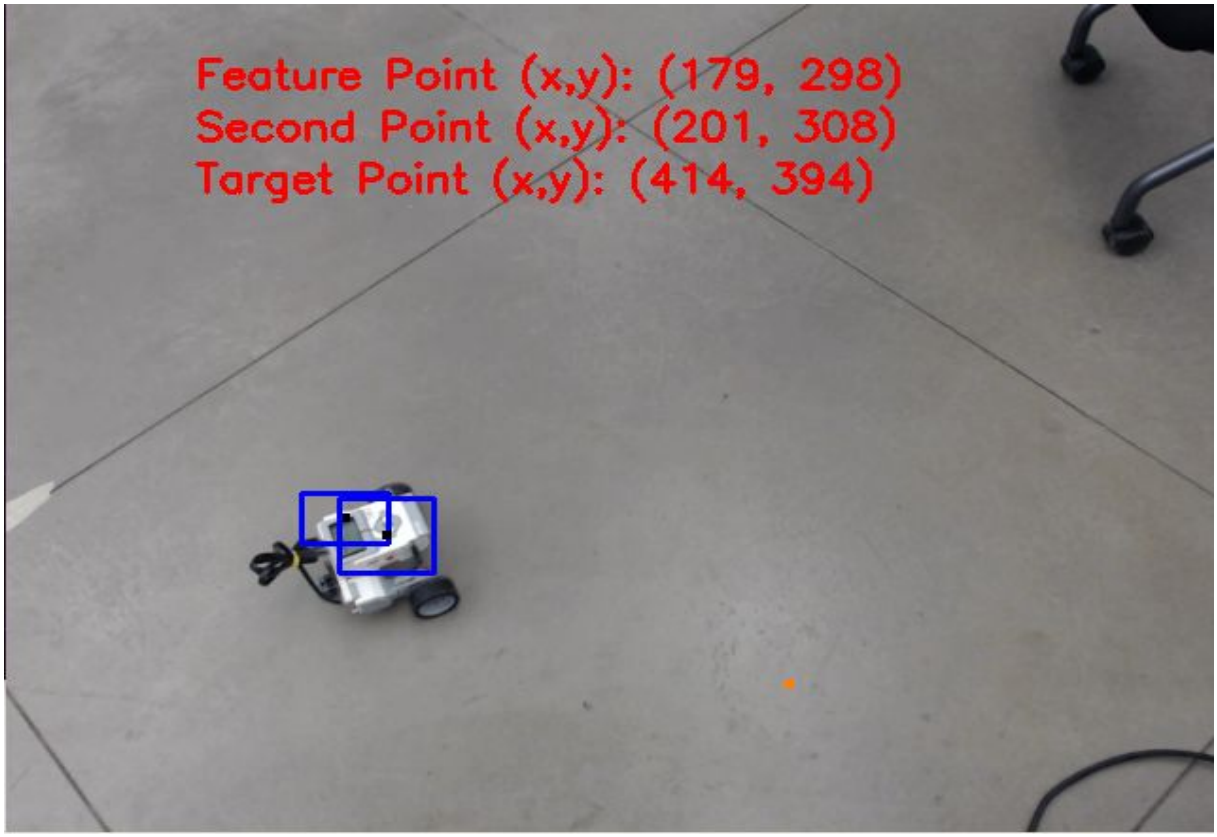
1. **BOOSTING Tracker:** This tracker is slow and doesn't work very well. Interesting only for legacy reasons and comparing other algorithms.
2. **MIL Tracker:** Better accuracy than BOOSTING tracker but does a poor job of reporting failure.
3. **KCF Tracker:** Uses Kernelized Correlation Filters. Faster than BOOSTING and MIL. Similar to MIL and KCF, does not handle full occlusion well. Reports failure

if it loses track of the robot when it moves too fast. Hence the robot needs to move at a lower speed

4. **CSRT Tracker:** Uses Discriminative Correlation Filter (with Channel and Spatial Reliability). Tends to be more accurate than KCF but much slower. Does not work well with a moving robots as it loses track fast and reports failures
5. **MedianFlow Tracker:** Does a nice job reporting failures; however, if there is too large of a jump in motion, such as fast moving objects, or objects that change quickly in their appearance, the model will fail
6. **TLD Tracker:**The TLD tracker does not stay on track with the differential drive. Misidentifies the differential drive after a few iterations as so the servoing fails
7. **MOSSE Tracker:** Very Fast. However very inaccurate. Loses track very quickly

In the end I decided on using the KCF tracker as it gives the best results with the fastest response time and simplest implementation mechanics, however it still has its fair share of problems as i stated above. As I used the KCF tracker, i needed to set the robot to move slowly so as to not make the tracker fail. I tested out various different speed, but ended up using 2% of the motor power as any higher than that more often than not fails. I used The KCF tracker to specify 2 bounding boxes on the robot, one at the head and the other at the center, so that it always has 2 points on it. I also used the KCF tracker to specify the target point on the image space.

Robot Movement



The feature point and Second point uses the KCF tracker to track them as stated above. The target point is a stationary point. The server_uws.py uses ComputeDelta to get the error vector between the feature and target point. Then in a loop, it checks if the norm of the delta is less than 10 or not (close distance to target point). If not, then in another loop nested inside the it checks, with the isBetween function if the second point(the point indicating the front end of the robot) located (or close enough) on a straight line between the feature point and the target point . If isBetween is false, the host computer

sends the (-2,2) through the socket connection to the client on the machine which is running testSocket.py. The client receives the data and moves the robot using these as the power to the left and right motors for 1 second> Visual servoing returns the new feature and the second point using the CV2KCF tracker bounding boxes, With the midpoint of the boxes being the points . This makes the robot rotate a bit in place. This rotation is done until the isBetween function returns true(when the second point is between feature and target point).Then then in another loop, still inside the initial loop it checks if the norm of the error vector is less than 10 or not, if not then it stores the error current error vector as previous error vector, moves the robot in a straight line by sending (5,5) to the robot using the socket connect, and then calculates the new error vector by getting the new feature point from the tracker, this goes on until you reach target point. However, if the norm of the previous error vector becomes greater than the norm of new error vector, it breaks out of this loop and goes back to the start of the initial while loop, entering the isBetween while loop again, repeating the steps above. This goes on till the robot reaches the vicinity of the target points (10 pixel range). Below is a sample pseudo code which shows how the loop structure explained above is like:-

```
While norm(err_vec)!=0:
    While !isBetween():
        Rotate
    While norm(err_vec)!=0:
        Prev_err = err_vec
        Do move
        Err_vec = target - feature
        If norm(Prev_err) > norm(err_vec)
            break
```

Testing

The image frame is 640 x 480. Hence all coordinates lie in that pixel height and width.

The only time the program doesn't end is when the tracker position shifts too much(the front point becomes the back point) and the 3 points can not align, hence rotation never ends. This is represented by DNE (Did Not End) in the Actual column

(x,y) (Feature Point Initial)	(x,y) (Target Point)	(x,y) (Feature Point Final)	(x,y)Error
(121, 111)	(131,142)	(135, 144)	(4, 2)
(135, 144)	(170, 255)	(167, 257)	(3, 2)
(167, 257)	(311, 444)	DNE	N/A
(347, 447)	(113, 114)	(109, 117)	(4,3)
(109, 117)	(600, 453)	(598,457)	(2,4)
(598,457)	(555, 335)	(557, 329)	(2,6)
(557, 329)	(431, 371)	DNE	N/A
(353, 354)	(404,347)	(402,350)	(2,3)
(402,350)	(217, 226)	(211, 233)	(6,7)
(211, 233)	(121, 312)	DNE	N/A
(132, 374)	(111, 312)	DNE	N/A
(123, 338)	(350, 350)	(350, 354)	(0, 4)
(350, 354)	(401, 349)	(402,350)	(1,1)
(402,350)	(214, 231)	(211, 233)	(3, 2)
(211, 233)	(333, 210)	DNE	N/A

Complexities & Solutions

The main problem that I faced was that the Tracker position shifts sometimes, making it so that the 3 points are never aligned in the right order. Hence the Robot never finds the target. This happens mainly due to rotation and occurs where there is a lot of steps (and angle turned is high) to reach the target point. One of the solutions I tried for this was to re-initialize the 2 tracker points after every rotation, this works quite well. However as I am aiming for a completely Autonomous robot, i had to drop this solution as it nears to a more teller operated system. Another thing that I tried was putting a Target with 2 unique symbols/drawings which stood out from the environment on top of the robot, so that the tracker could have much more success. However, there was not much success to this solution, so in turn this too was dropped. A probable solution to this would be to use an automated object tracking instead of selecting region with KCF. This can be done by Sticking 2 Coloured balls to the robot and use those as the points on the robot, with coloured object tracking. This would also make the robot much more Automate, I will be discussing this in the next section. Another minor complexity is that, when using the KCF tracker, I need to use low speeds as otherwise the KCF tracker loses track, and it fails. The only solution to this would be to use a different tracker. Another Complexity I encountered with the KCF tracker was that there was a tradeoff between the stability of it and the size of image space. The closer the camera is to the robot, the better the tracking is, however this also makes the image size smaller. I reached an optimum balance between these 2 aspects for this project, however im sure we can

increase the image size and still keep the stability by using a better resolution camera and a tracker that can support high resolution frames

Conclusion & Future Work

To conclude, I would say that the results of the project were quite successful, if you ignore the fact that the tracker i used for the project is quite unreliable in some aspects, such as image space, shifting when the robot rotates and failing when it moves at high speeds. All these however can be fixed by using a better tracker program in the Computer Vision part of the project. In the Future, I would like to improve upon this project by adding autonomous object detection as described above. Maybe even use deep learning to identify the object itself in the frame by using a training set, this would allow me to easily track and get the 2 points on the robot. Furthermore, for real life use, we can have the Computer Vision program also identify the target locations using deep learning. For example, this would allow it to identify 2 different types of target locations,such as pickup and drop off points. This would allow the robot to efficiently travel to the pick up point and get the cargo and then drop it off at the drop location, This would effectively reduce human labour.