Monocular Vision Assisted Odometry

Project for Center and Laboratory for Intelligent Systems

Giorgio Tabarani and Christian Galinski

April 2015

Summary of project in combination of wheel encoders odometry and vision in order to both improve the odometry accuracy, and to detect obstacles

Contents

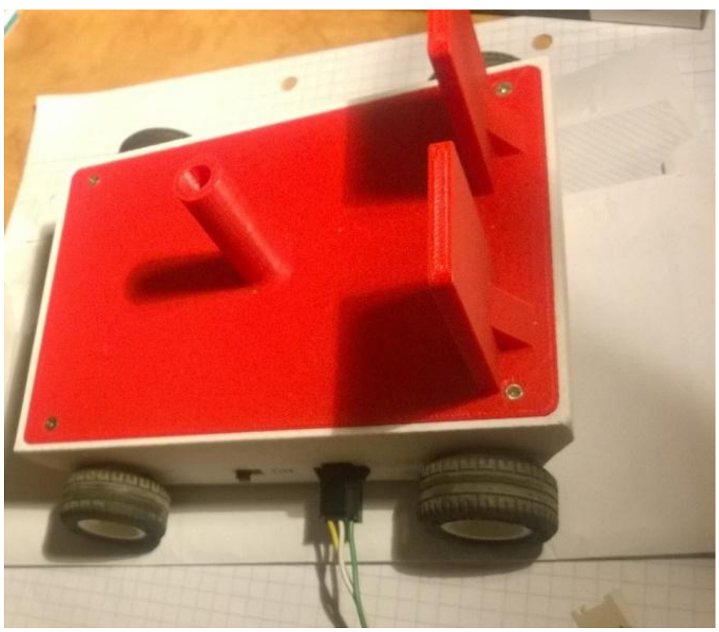
Introduction

Most common instruments for obstacle detection on robots are laser sensors and ones based on signals broadcasted and received by the robot itself to determine the distance to obstacles around it. This is a great manner to avoid obstacles but costly since these sensors are relatively expensive.

The goal in this project was to come up with an algorithm based on a somewhat cheap accessory one can attach to a robot and determine the distance to an obstacle. This accessory is a camera which can be found anywhere nowadays from mobile phones to tablets and wearables and standalone cheap cameras which can cost no more than $5.

After attaching a camera to the robot, with every move the robot will take a single frame from the camera and feed it to our algorithm which will in turn analyze this frame with the previous one in mind and determine the distance between the robot's current location and the closest obstacle to it.

We hereby explain the tools and the algorithms we used and developed and lastly present theoretical results as well as practical results.



Tools

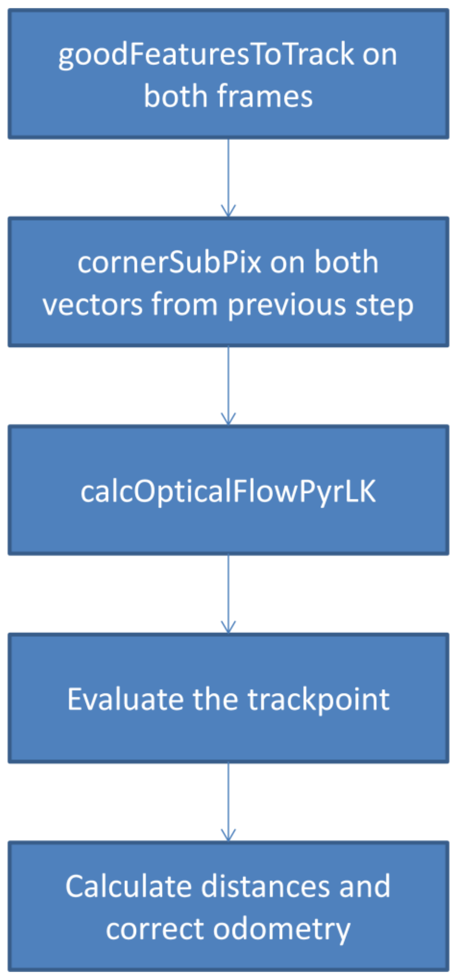
The algorithm was implemented in C++ using Visual Studio 2012 and with the help of OpenCV Library version 2.4.9.

Interfacing with the robot was in an Android application written in Java and sending/receiving data from the above C++ code using JNI while the pictures were taken from a Nexus 5's rear camera and downgraded to 640x480 pixels.

Algorithm

We implemented the required interface available in `VisualAssistedOdometry.h` using C++ under the class `FramesAnalyzer`.

The algorithm was simple and straight forward and consisted of the following steps:



GoodFeaturesToTrack

After receiving two frames as input, we run OpenCV's goodFeaturesToTrack algorithm in order to find the best points we should keep track of which will help us calculate the distance each object has moved.

The algorithm takes 8 arguments: relevant image, an empty vector which it fills with the good points it finds, maximal number of corners to find, the quality level, minimum distance in pixels between corners, mask, block size which it uses to determine if a point is in fact a corner and a boolean which determines whether to use HarrisDetector.

The parameters we have supplied to each frame's analysis were:

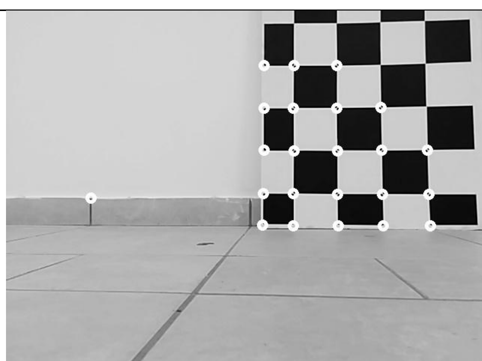
The frame itself, an empty vector so it can fill it up, maximum number of corners is 10000, quality level of 0.05, minimum distance of 2 pixels, an empty mask, a block size of 7 and true for using HarrisDetector.

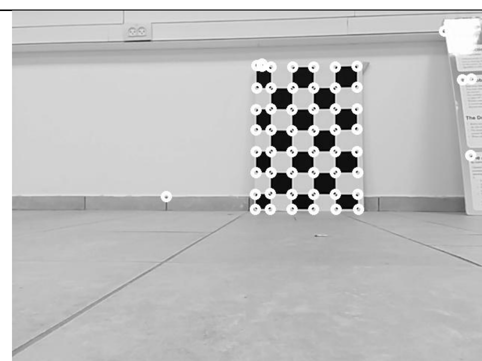
The quality level is used in the following manner: the highest measurement of Harris's function response is multiplied by this value, in our case 0.05, and all the points which have a quality level less than the calculated value are rejected. This is one of the screening processes we use to ignore malfunctioning points.

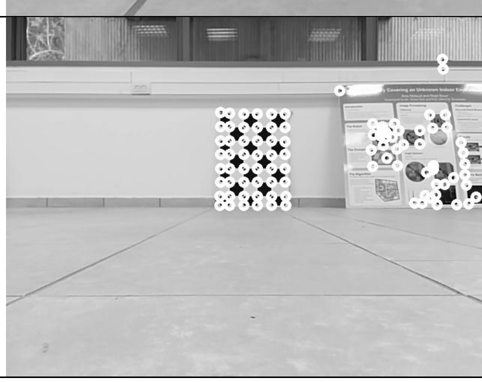
Minimum distance is in fact the Euclidean distance between two points; if the value is less than 2 pixels then the points are ignored since they're considered the same.

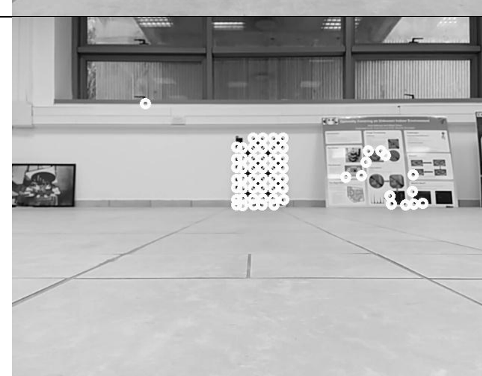
Block size is the size of a block to use in computing the derivative.

Few of results of this algorithm:







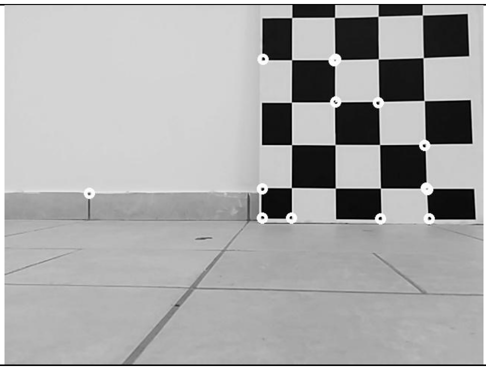
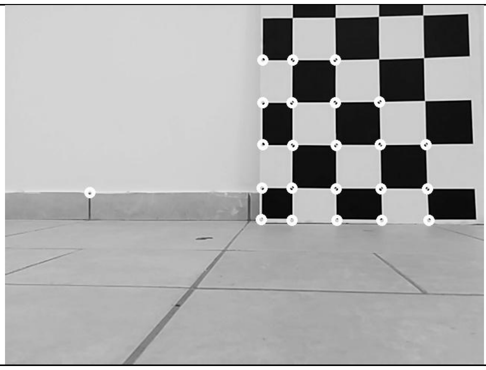


CornerSubPix

The results of GoodFeaturesToTrack, the two vectors of corners, are passed to a follow up function called cornerSubPix. Its main purpose is to refine the corners' locations in order to get more accurate results and avoiding hiccups due to low image resolution. Its results give us an accuracy of up to 3 digits after the decimal point.

The algorithm is an iterative one which we terminate after 30 iterations or when the corner position moves by less than 0.01.

Here are results of before and after applying the algorithm on the same frame.

Before	After
	

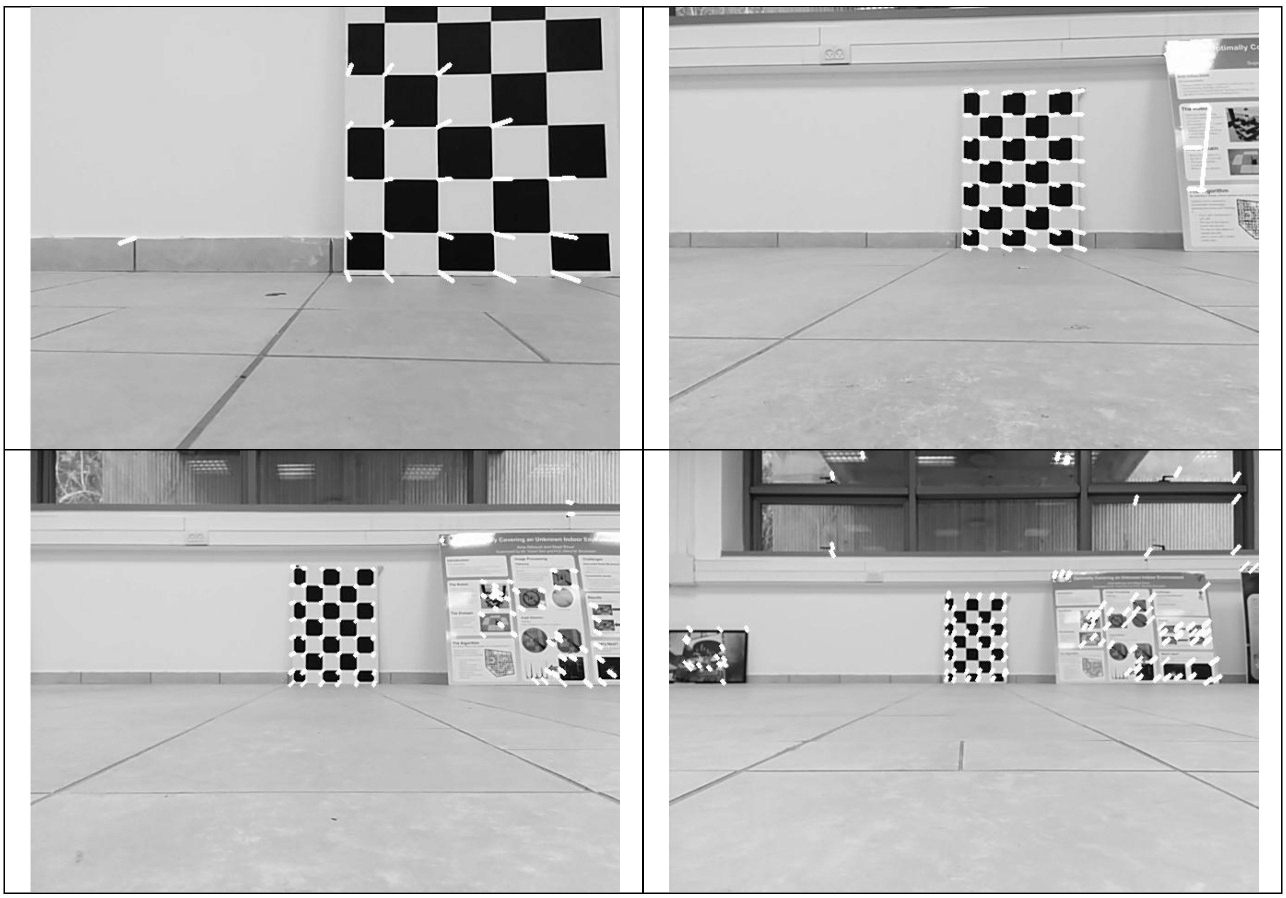
CalcOpticalFlowPyrLK

The upgraded corners from CornerSubPix are passed to CalcOpticalFlowPyrLK function in order to see each corner from one vector to where it was mapped in the second vector, meaning each corner from one frame to the other.

As output, it returns both vectors we fed to it along with a status vector and an error vector which indicate how accurate the match was, if at all.

This algorithm is also an iterative one which we terminate after 20 iterations or when the search window moves by less than 0.3.

Here are examples of running the algorithm on 4 sets of 2 frames (i.e each picture here was made out of 2 frames).
The drawn lines represent where each corner has moved from the first frame to the second.



Evaluate track point

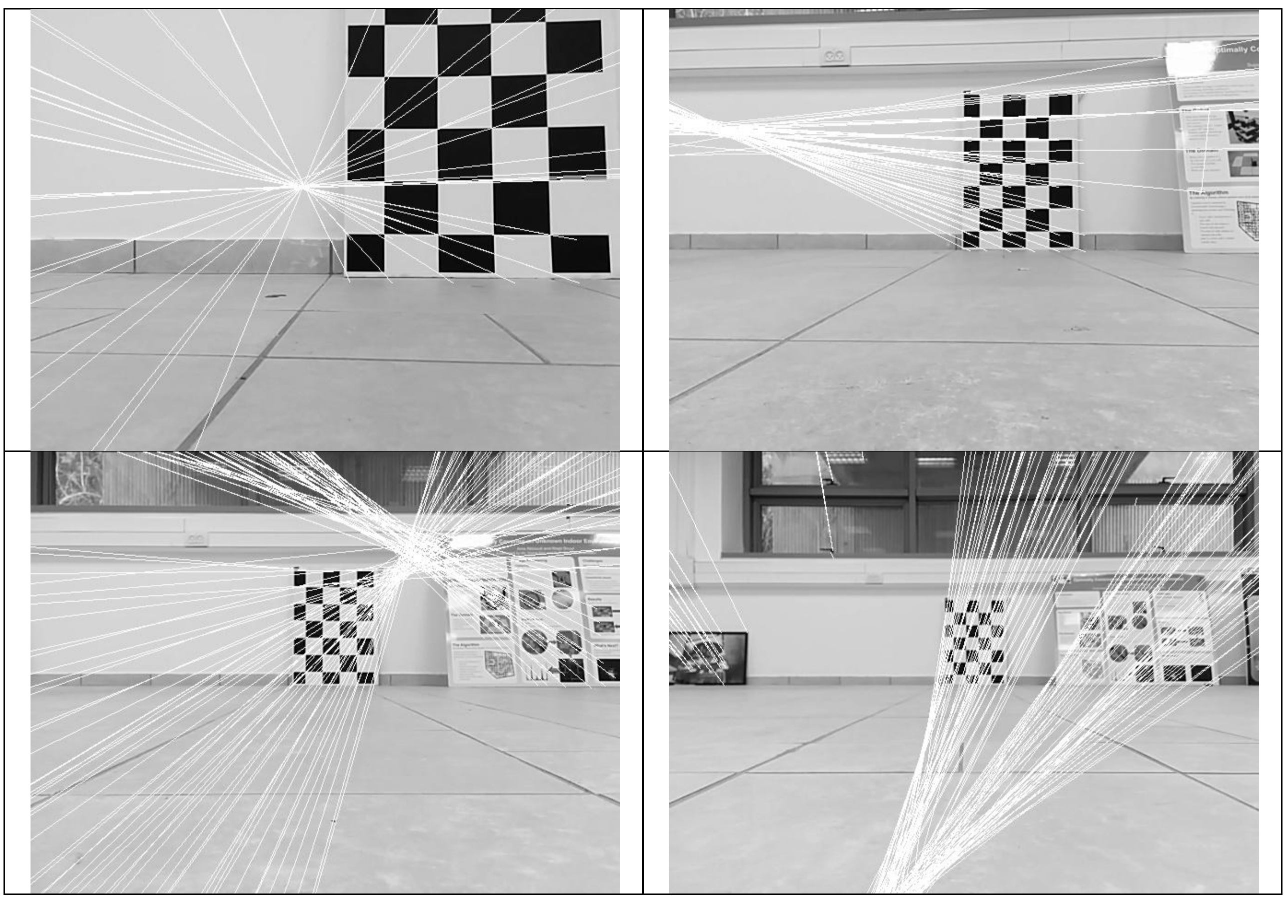
In order to find the point which the robot is moving towards, we took each two matching points and calculated the line's equation which passes through them. Eventually we intersected all the lines together to find the initial center point.

In a second pass, we intersected the lines once again but rejected lines whose intersection point was 4-6% (in each dimension) away from the initial center point.

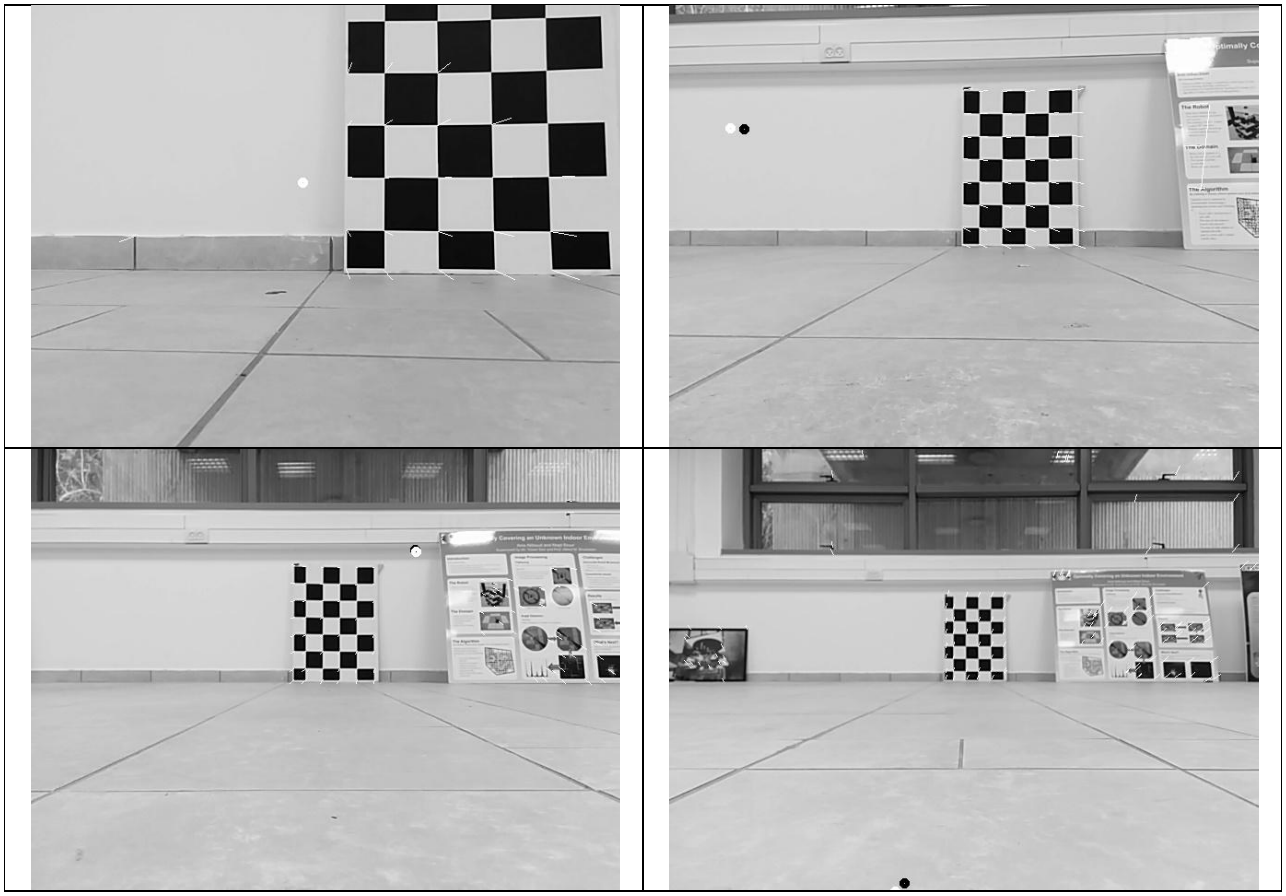
The second pass would incur in results better in at least 2 pixels in each dimension and this was critical considering the low resolution of the images.

We chose this point to assist us in calculating the distance later on since this is the only point which does not change its position between two consecutive frames.

Here we see 4 different sets of images where each set is composed of 2 frames.
The continuation of the vectors from CalcOpticalFlowPyrLK drawn to give an estimate where the track point is located.



The black point is the initial intersection point whereas the white point is the updated one as described above.



Calculate distances and correct odometry

Distance calculation

Using simple geometry, we can conclude the following formula to calculate a distance of an object:

$$d_1 = s \cdot \frac{x_2}{x_2 - x_1}$$

Where S is the step size, x_1 is the object's size in pixels in the first frame, x_2 is the object's size in pixels in the second frame and d_1 is the object's distance from the point where the first frame was taken.

This formula came from the pinhole projection formula like so:

$\frac{x}{f} = \frac{X}{d}$ where x is the size of the object on the sensor, f is the focal length, X is the size of the object and d is the distance from nodal point to the object.

If we write this formula for the two frames, using subindex 1 for the first frame and subindex 2 for the second frame, we get the following, given X and f do not change for the frames:

$$\frac{x_1}{f} = \frac{X}{d_1}, \qquad \frac{x_2}{f} = \frac{X}{d_2}$$

If we consider $d_1 = d_2 + s$ then after reordering the equation such that the constants are in one side and the variables in the other, we get:

$$\begin{aligned}
&x_1 \cdot d_1 = f \cdot X, \qquad x_2 \cdot d_2 = f \cdot X \\
&\to x_1 \cdot d_1 = x_2 \cdot d_2 \\
&\to x_1 \cdot d_1 = x_2 \cdot (d_1 - s) \\
&\to \boldsymbol{d_1 = s \cdot \frac{x_2}{x_2 - x_1}}
\end{aligned}$$

This forces us to pay careful attention to images from a large distance as the difference between $x_1 and\ x_2$ can be very small and thus we divide by a small number and cause numerical errors which we'll address in our analysis later on.

In our implementation, we took a constant point – namely the track point, explained above – and calculated each corner's distance relatively to it in pixels. Therefore, a corner which moved between the frames "creates" two objects $x_1 and\ x_2$, where x_1 is (defined by) the distance between the corner in the first frame and the track point, and x_2 is the distance between the corner in the second frame and the same track point (which does not move between the frames).

In this way, since the track point stays fixed, its distance to the nodal point is not determined, so we can choose it and set it as equal to each corner's distance to the nodal point, and so we are able to get the pure distance to these corners without being dependent on any other specific corners.

As it stands, the results from this algorithm were not sufficient as we shall demonstrate in the **Theoretical results** section later on, and so we had to improve it.

Odometry correction

In this section we shall explain how we can calculate the improved odometry step size per move and thus achieve a more accurate estimate for the distance from an obstacle per move.

This analysis is relevant only for theoretical approximation since it depends on having an average for the odometry step size.

Using the above mentioned equation of $d_1 = s_1 \cdot \frac{x_2}{x_2 - x_1}$ and if we denote $z_i = \frac{x_i}{x_i - x_{i-1}}$, where x_i, x_{i-1} were calculated from the frame's analysis as explained before, we can write:

$$\boldsymbol{d_1 = s_1 \cdot z_2 \rightarrow s_1 = \frac{d_1}{z_2}}$$

$$\forall i \geq 1, \qquad d_{i+1} = d_i - s_i, \qquad d_{i+1} = s_{i+1} \cdot z_{i+2}$$

$$\rightarrow \boldsymbol{s_{i+1} = \frac{d_{i+1}}{z_{i+2}} = \frac{d_i - s_i}{z_{i+2}} =}$$

$$\boldsymbol{= \frac{s_i \cdot z_{i+1} - s_i}{z_{i+2}} = s_i \cdot \left(\frac{z_{i+1} - 1}{z_{i+2}}\right) =}$$

$$\boldsymbol{= s_{i-1} \cdot \left(\frac{z_i - 1}{z_{i+1}}\right) \cdot \left(\frac{z_{i+1} - 1}{z_{i+2}}\right) = \cdots = s_1 \cdot \prod_{j=2}^{i+1} \frac{z_j - 1}{z_{j+1}}}$$

$$\boldsymbol{average\ of\ s_i = \frac{s_1 + \sum_{i=1}^{n-1} s_{i+1}}{n} =}$$

$$\boldsymbol{\frac{s_1 + \sum_{i=1}^{n-1} \left(s_1 \cdot \prod_{j=2}^{i+1} \frac{z_j - 1}{z_{j+1}}\right)}{n} = s_1 \cdot \frac{1 + \sum_{i=1}^{n-1} \prod_{j=2}^{i+1} \frac{z_j - 1}{z_{j+1}}}{n} = average\ odometry\ step\ size}$$

Thus, by knowing the average odometry step size for n moves, we can get the value of s_1 and afterward calculate the value of every s_i *and* $\boldsymbol{d_i}$, $\forall i\colon 2 \leq i \leq n$ depending on a realistic step size for the move itself and not an average one which depends on all the other moves as well.

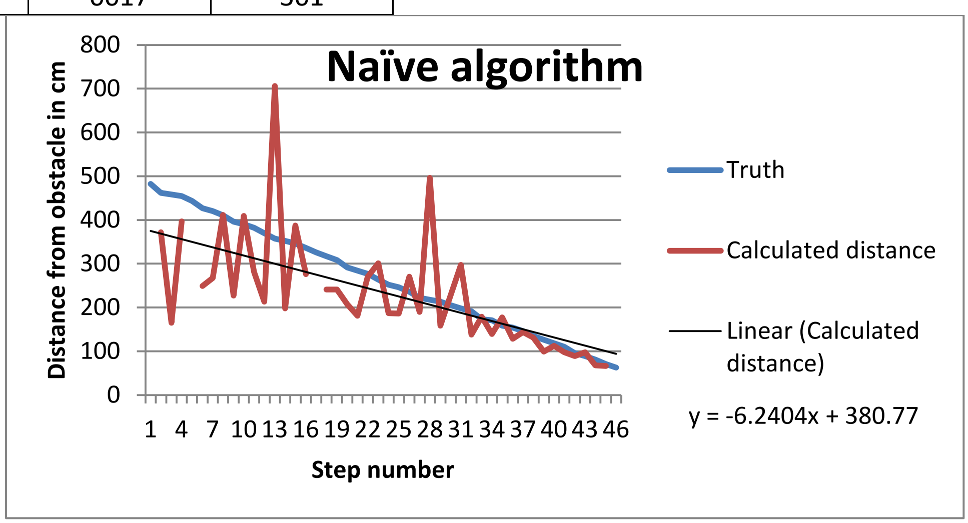
Theoretical results

In order to test the algorithm's performance we ran several simulations on the provided "Calibration" data set and measured the goodness of the algorithm by the quality of the graph and the trend line.

In the beginning, we assumed each step's size was the given encoder step approximation (8.32cm) and ran the algorithm described in Distance calculation section.
As stated earlier, the results were far from accurate as we can see in the graph below which has a lot of inconsistencies in its slope:

Frame1	Frame2	Distance
0000	0001	
0001	0002	372
0002	0003	165
0003	0004	397
0004	0005	
0005	0006	249
0006	0007	267
0007	0008	411
0008	0009	227
0009	0009a	409
0009a	0009b	281
0009b	0009c	213
0009c	0009d	706
0009d	0009e	198
0009e	0009f	387
0009f	0010	276
0010	0011	
0011	0012	241
0012	0013	241
0013	0014	207
0014	0015	181
0015	0016	270
0016	0017	301

Frame1	Frame2	Distance
0017	0018	187
0018	0019	186
0019	0019a	270
0019a	0019b	190
0019b	0019c	496
0019c	0019d	158
0019d	0019e	228
0019e	0019f	297
0019f	0020	138
0020	0021	179
0021	0022	139
0022	0023	177
0023	0024	128
0024	0025	144
0025	0026	131
0026	0027	99
0027	0028	112
0028	0029	98
0029	0029a	89
0029a	0029b	98
0029b	0029c	68
0029c	0029d	66

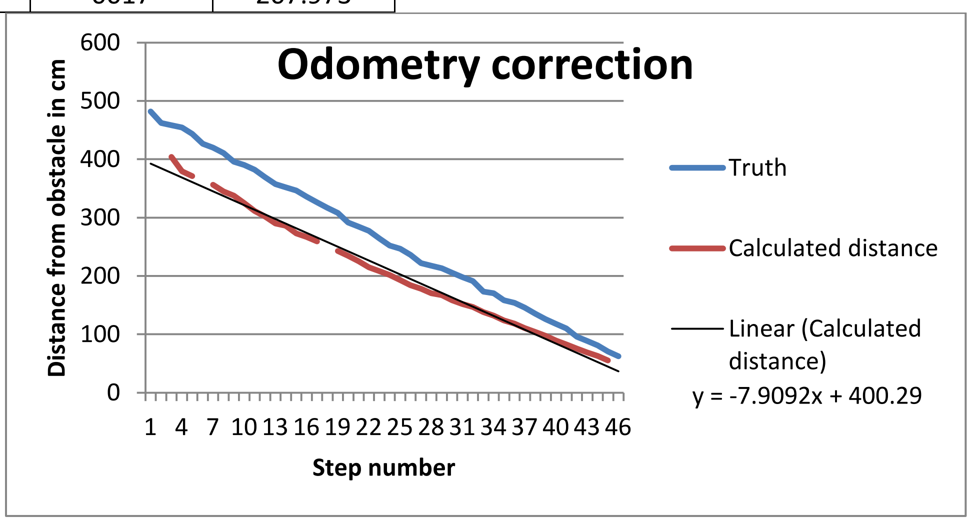


Later on we turned to the improved algorithm described in Odometry correction section, where **n** is the number of transitions between frames (45 in this dataset) and **average odometry step size** was set to be the encoder step approximation (8.32).

We see an outstanding improvement in the results as can be seen in the graph below:

Frame1	Frame2	Distance
0000	0001	
0001	0002	
0002	0003	404.016
0003	0004	379.114
0004	0005	371.06
0005	0006	
0006	0007	356.393
0007	0008	344.937
0008	0009	337.455
0009	0009a	324.775
0009a	0009b	311.035
0009b	0009c	301.831
0009c	0009d	289.823
0009d	0009e	285.844
0009e	0009f	273.114
0009f	0010	267.055
0010	0011	259.007
0011	0012	
0012	0013	242.925
0013	0014	234.376
0014	0015	224.882
0015	0016	214.574
0016	0017	207.973

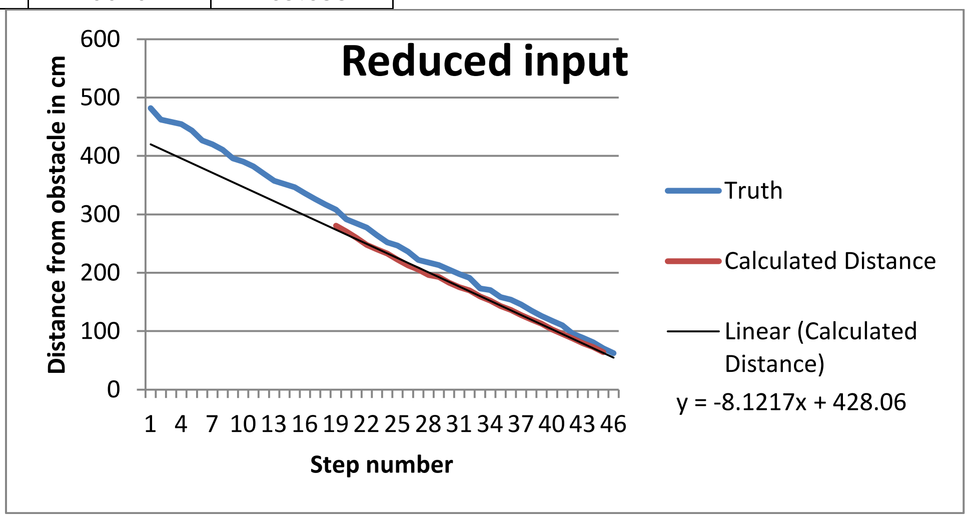
Frame1	Frame2	Distance
0017	0018	201.732
0018	0019	192.785
0019	0019a	184.201
0019a	0019b	177.971
0019b	0019c	170.108
0019c	0019d	167.256
0019d	0019e	158.465
0019e	0019f	151.724
0019f	0020	146.992
0020	0021	138.173
0021	0022	131.77
0022	0023	123.926
0023	0024	118.114
0024	0025	110.466
0025	0026	104.046
0026	0027	97.4389
0027	0028	89.2522
0028	0029	82.627
0029	0029a	75.6615
0029a	0029b	68.6109
0029b	0029c	62.8302
0029c	0029d	55.1711



Lastly, we ran a simulation on reduced input which is closer to the obstacle to see how the algorithm behaves with smaller distances and received even better results!

Frame1	Frame2	Distance
0012	0013	280.379
0013	0014	270.512
0014	0015	259.554
0015	0016	247.657
0016	0017	240.037
0017	0018	232.835
0018	0019	222.509
0019	0019a	212.6
0019a	0019b	205.411
0019b	0019c	196.335
0019c	0019d	193.043
0019d	0019e	182.897
0019e	0019f	175.117
0019f	0020	169.655

Frame1	Frame2	Distance
0020	0021	159.476
0021	0022	152.086
0022	0023	143.033
0023	0024	136.325
0024	0025	127.498
0025	0026	120.087
0026	0027	112.462
0027	0028	103.013
0028	0029	95.3662
0029	0029a	87.3268
0029a	0029b	79.1892
0029b	0029c	72.5173
0029c	0029d	63.6773



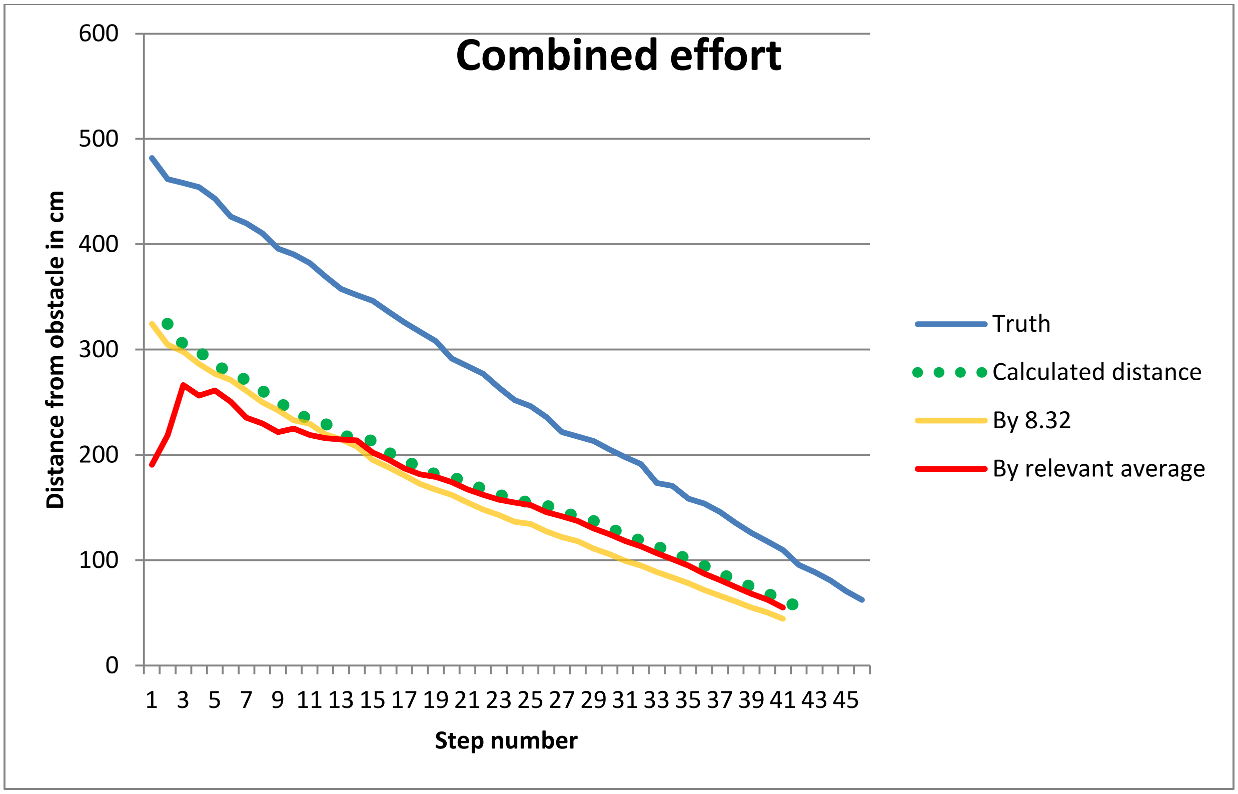
Further attempts of improvement led us to use the given step size (namely **S** in the above equations) for the first few steps and after that to start improving it and using the updated value for the distance calculation.

In the table below we display the calculated distances after multiplying with the Total average of all steps so far versus multiplying by the average until each step.

Naïve multiplied by 8.32	Multiplied by relevant average	Best out of two worlds	Multiplied by final average
332.8616		332.8616	414.3726
324.5416	190.669	324.5416	404.0152
304.5386	218.4739	304.5386	379.114
298.069	266.2336	298.069	371.0601
286.287	256.2238	286.287	356.393
277.0843	261.1759	277.0843	344.9366
271.0748	250.6858	271.0748	337.4555
260.8886	235.3359	260.8886	324.7749
249.8513	229.9427	249.8513	311.0348
242.4581	221.7877	242.4581	301.8312
232.8119	225.1316	232.8119	289.8228
229.6154	218.8659	229.6154	285.8435
219.3892	215.9431	219.3892	273.1132
214.5229	214.5159	214.5229	267.0552
208.0574	213.6737	213.6737	259.0065
195.1398	202.1177	202.1177	242.9256
188.2724	195.525	195.525	234.3766
180.6455	187.2905	187.2905	224.8819
172.3654	181.6392	181.6392	214.5743
167.0623	179.0213	179.0213	207.9724
162.0495	174.2131	174.2131	201.7321
154.8627	167.2461	167.2461	192.7854
147.967	162.071	162.071	184.2012
142.9626	157.6104	157.6104	177.9712
136.646	154.6353	154.6353	170.1079
134.3547	152.2575	152.2575	167.2554
127.2935	145.5728	145.5728	158.4651
121.8788	141.6348	141.6348	151.7245
118.0774	137.2637	137.2637	146.9922
110.993	130.1725	130.1725	138.1729
105.8495	124.5623	124.5623	131.77
99.5488	118.3299	118.3299	123.9263
94.87962	113.1753	113.1753	118.1137
88.73696	106.6198	106.6198	110.4668
83.57856	101.0597	101.0597	104.0453
78.27173	94.75301	94.75301	97.4389
71.69544	87.29781	87.29781	89.2522
66.37347	81.18464	81.18464	82.62699
60.7781	74.64529	74.64529	75.66143
55.11451	68.20023	68.20023	68.61094
50.47087	62.57781	62.57781	62.83016
44.31839	55.17107	55.17107	55.17107

The green marked rows in the first two columns represent better values than their counterpart in red which means they are closer to the wanted values which are represented in yellow.

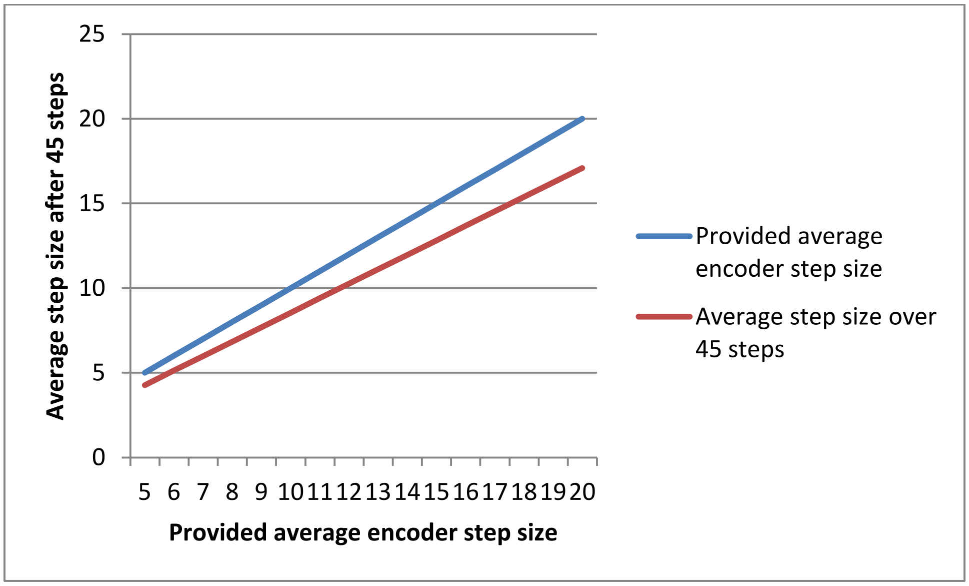
The third column is our output which combines the first two.



We notice in all graphs that when we are closer to the obstacle we get more accurate results.

Dependency on initial step size

By providing a different average encoder step size, we receive the following graph which represent the actual average step size value after 45 steps.



We notice couple of things: First of all the definite average is usually lower than the provided average and that's probably due to irregularity in the pictures such as rotations and not using the reset function, second of all it's a semi-linear graph and that's due to the fact our algorithm is based on multiplication and therefore when you double the provided average encoder step size, the total average will also be doubled, and lastly we do not see any convergence and that's because our algorithm already tries to match the step size to the provided average step size.

Installation on a live robot and results

When finally arriving to integrate the image analysis and processing code with the robot, we started facing real-world problems such as drift and hardware issues with the robot's encoders which affected its behavior and compliance with our commands.

Such issues can be noticed when rotating the robot by giving its encoders opposite values yet the wheels don't turn in the same speed, this means the robot's rotation wasn't always in the same size despite giving it the same parameters and the solution to this problem was to provide it with a constant speed over a fixed period of time which resulted in a relatively constant rotation angle eventually. Yet, this solution caused another problem; the robot's encoders were in incomplete circle state, which resulted in relatively short and unpredictable first forward step after the rotation. This problem is unsolvable since we don't have direct access to each encoder separately in order to modify it and complete the circle.

In order to deal with (few) possible unwanted rotations caused by image analysis hiccups, in case the reported distance is below the minimal distance threshold – but not critical – we let the robot move forward in safe mode, such that every following distance-too-close report will result in immediate rotation.
We use the following pseudo-code:

```
if(dist < 75)
       Enter safe mode
if(dist <= 25)
       Rotate
if(in safe mode)
       if(dist >= 55)
              Move forward
       if(dist >= 75)
              Exit safe mode
       if(dist < 55)
              Rotate
//75 - minimal distance threshold
//55 - distance-too-close
//25 - critical
```

Sample run in the table in the next page.

Step number	Distance	If statement
0	Start	
1	Mismatch	
2	Mismatch	
3	114.009	
4	85.74315	
5	78.42592	
6	75.06543	
7	69.20378	25<dist<75
8	61.56432	25<dist<75
9	25.31468	
10	Rotate	
11	93.41955	
12	80.11607	
13	68.71044	25<dist<75
14	56.69225	25<dist<75
15	46.32505	
16	Rotate	
17	53.7094	25<dist<75
18	23.59244	dist<25
19	Rotate	
20	47.02141	25<dist<75
21	37.87338	
22	Rotate	
23	75.69654	
24	63.37202	25<dist<75
25	55.0888	25<dist<75
26	45.83493	
27	Rotate	
28	42.85822	25<dist<75
29	33.74571	
30	Rotate	
31	57.69447	25<dist<75
32	47.89338	
33	Rotate	
34	53.06975	25<dist<75
35	45.12167	
36	Rotate	
37	101.3752	
38	89.00514	
39	78.29432	

Step number	Distance	If statement
40	61.86887	25<dist<75
41	54.91204	
42	Rotate	
43	69.40255	25<dist<75
44	43.79698	
45	Rotate	
46	501.7636	
47	362.4899	
48	289.064	
49	244.7373	
50	200.835	
51	168.0163	
52	140.5826	
53	112.0302	
54	89.17937	
55	66.37161	25<dist<75
56	47.54035	
57	Rotate	
58	45.23246	25<dist<75
59	33.81655	
60	Rotate	
61	112.7145	
62	96.65104	
63	83.25559	
64	66.07921	25<dist<75
65	52.95801	
66	Rotate	
67	75.224	
68	51.27864	25<dist<75
69	51.27864	

Conclusion

Camera's prices are falling more rapidly than any other sensor (such as lasers) despite them being just as rich (in data), if not richer, than normal sensors even in a monocular vision system.

This project has described a system capable of estimating a robot's distance from its closest obstacle while taking into account many variables such as low resolution images, robot's drift and encoders' issues.

This system is easily accessible to everyone when the only requirements are a robot and a camera, even from a smart-phone.

Future work

As we saw there have been a lot of irregularities in the steps made between the frames from the dataset which could easily affect the algorithm. It will be nice if we had frames taken in specific step sizes and angle and develop the algorithm to detect whether the obstacle is in fact on the right side or left side of the robot and rotate him accordingly. With that in mind, we can develop it further and implement some of the Bug algorithms or map a room, just as robots usually do with SONAR and LIDAR devices.

It will also be interesting to see how low the images' resolutions can be and still be able to detect an obstacle.

Bibliography

GoodFeaturesToTrack - Shi and C. Tomasi. Good Features to Track. Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pages 593-600, June 1994.

HarrisDetector - C. Harris and M. Stephens (1988). "A combined corner and edge detector". Proceedings of the 4th Alvey Vision Conference: pages 147—151

CornerSubPix - http://docs.opencv.org/modules/imgproc/doc/feature_detection.html#cornersubpix

CalcOpticalFlowPyrLK - J. Y. Bouguet, (2001) . Pyramidal implementation of the affine lucas kanade feature tracker description of the algorithm. Intel Corporation, 5

OpenCV - opencv.org