Hit Me If You Can

U Chun Lao, Flavian Hautbois

May 16, 2012

Abstract

The task of estimating distance for a robot with a single camera is a difficult one. It has been addressed several times[1][2]. Our goal is to train a robot to avoid a yellow ball in an open, unknown environment. We use the known diameter of the ball to compute its distance to the camera, then predict its next positions with a Kalman filter, and we provide two methods to avoid the ball. Finally, we show the results of several experiments aiming at quantifying the accuracy of our methods and we give several steps forward that could be taken next.

1 Trajectory Estimation

1.1 Camera Model

In order to track a ball in the 3-D space with only one camera, there are existing models in computer vision that allows us to estimate its coordinates. Given the fact that the frame rate of the camera is not high enough and the processing power of the robot is also relatively low, we pick the simplest one called "pin hole model".
The pin hole model enforces that rays of light travel in straight lines from the object, through the pin hole, to the sensor plane, under the assumption that the lens is ideal. According to the model, any object at a distance y from the lens with a distance z from the principle axis, the corresponding image on the sensor with the height z' is governed by the equation

$$\frac{z'}{f} = \frac{z}{y} \tag{1}$$

where f is the focal length of the camera.
From which we can deduce the set of equation

$$\begin{bmatrix} x \\ z \end{bmatrix} = \frac{y}{f} \begin{bmatrix} x' \\ z' \end{bmatrix} \tag{2}$$

where x' and z' are the coordinate of the image on the sensor in pixels, which can be measured using computer vision algorithms such as Hough circle. Thus the actual x, z coordinate of the ball in 3-D space (in centimeters) can be easily computed if we can somehow obtain the distance between the ball and the camera in the y direction (in centimeters).

Other than the position on the image, the diameter of the ball on each frame can also be easily measured in pixels. Since we know the actual diameter of the ball, which is 6.7 cm, by equation 1 we can estimate the y coordinate of the ball in 3-D space using the equation

$$y = \frac{d \cdot f}{d'} \tag{3}$$

where d and d' are the diameter of the ball in 3-D space and on the image respectively.

1.2 Setting Up the Parameters

1.2.1 Exploiting the model

Before we can start tracking the ball in a frame captured by the camera in real time or extracted from a video, we first have to set up several parameters, which include the focal length of the camera, and the color thresholds for the ball.

Focal Length f

As mentioned in the previous section, the "pin hole camera model" requires the focal length of the camera in order to estimate the distance to any object. However, most of the cameras that we are going to use do not have the focal length given. Moreover, according to equation (3) the focal length has to be expressed in pixels instead of centimeters to match the units. Therefore we have to calibrate the camera at the very beginning. Fortunately, the calibration process is quick and simple for the "pin hole camera model". Since we know the real diameter of the ball, and the diameter in the image. By placing the ball at some fixed distance, say 15 cm from the camera and plugging the parameters into equation (3), we can compute the focal length of the that specific camera in pixels for position estimation.

Color Thresholds

Since the Hough circle algorithm only works on binary images, the captured frame must be thresholded before the algorithm can handle it. The process that we are going to use will transform the frame into a grayscale frame which represents the probability of each pixel being part of a ball. Thus we need a threshold to filter the probability frame into a binary image.

1.2.2 Correction to the model

The pinhole camera model is built on the ideal lens model, which is ideal and does not exist in this world. One of the issues that the ideal lens model ignores is the curvature of the lens. In OpenCV there is an extended pinhole camera model that takes care the problem. The position in 3D is related to the point in the 2D image in the following way:

$$\begin{bmatrix} x' \\ z' \end{bmatrix} = \frac{1}{y} \begin{bmatrix} x \\ z \end{bmatrix} \tag{4}$$

$$\begin{bmatrix} x'' \\ z'' \end{bmatrix} = (1 + k_1 r^2 + k_2 r^4 + k_3 r^6) \begin{bmatrix} x' \\ z' \end{bmatrix} \tag{5}$$

$$+ \begin{bmatrix} 2p_1 \\ 2p_2 \end{bmatrix} x'z' + \begin{bmatrix} p_2(r^2 + 2x'^2) \\ p_1(r^2 + 2z'^2) \end{bmatrix} \tag{6}$$

$$u = f_x \cdot x'' + c_x \tag{7}$$

$$v = f_y \cdot z'' + c_y \tag{8}$$

where $r^2 = x'^2 + z'^2$, (u, v) is a point on the image. This extended model handles the curvature of the lens but there is no easy inversion that maps from the image plane to the 3D location. Therefore in this project, we uses the original model with little modification.

As the focal length of the lens acts as a scaling factor for all three coordinates, we can adjust it based on the position and size of the ball in the image in order to correct the location estimation. To find a proper mapping between radius and the focal length, the simplest way is to fit a curve to the data. Based on the distance of the ball from the robot, we can divide it into three cases: close, medium and far away. To simplify the problem, we will only fit curves to the focal length to the medium range data as it is the most proper range for the robot to react.

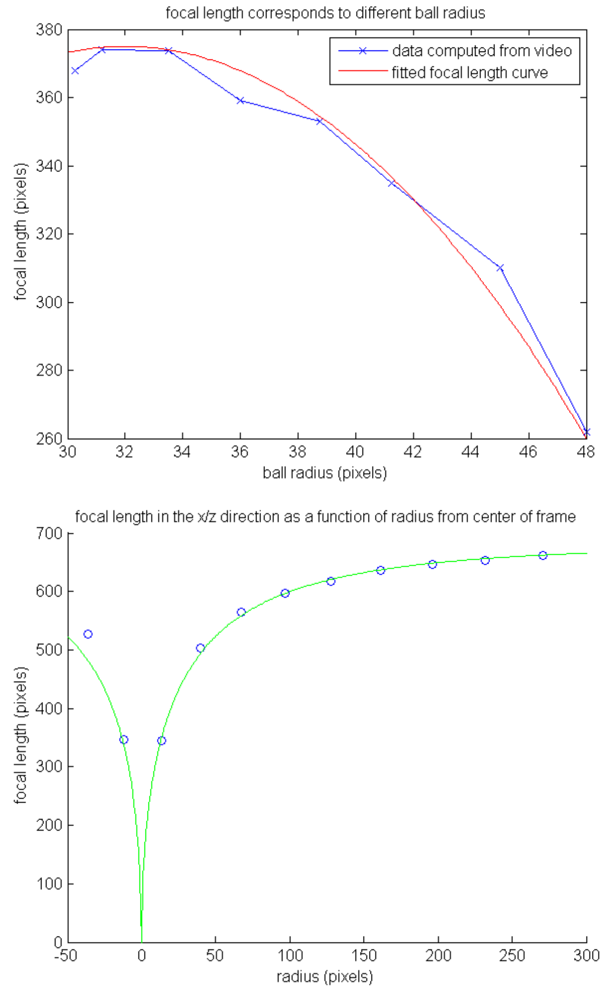

Figure 1: Fitted focal length in direction direction

1.2.3 Fitting Result

From the data collected and the testing results on color detection, the most reasonable range to work on turns out to be from 1.5 m to 2.5m. The focal length corresponds to the y-direction is given by

$$f_y = 375 - \frac{115}{256}(r - 32)^2$$

and the focal length corresponds to the x/z direction is given by

$$f = 680\left(1 - e^{-|x|^{0.5387}/5.5964}\right)$$

where r is the radius given by the Hough circle algorithm and x is the position of the ball measured from the center of the frame. It is similar for the z direction. Figure 1 shows the curve fitting to part of the data.

1.3 Using histograms for detection

As we are using OpenCV, we have access to different color spaces: HSV, YCrCb, or Lab. At first we assumed OpenCV opened the files in RGB, but in fact it opens them in BGR, which caused a lot of problems; we found out about this very late. To retrieve a ball on an image we assume that it has a particular histogram. We calculate the average histogram for a ball and then use it for a back projection. The back projection is basically assigning to each pixel of the source image a probability of belonging to a ball, based on the histogram. We then get a gray scale image that we threshold to get a binary image and then we apply a median filter and a blur filter.

By trying some color spaces and manipulating the number of bins for the histogram, we found out that YCrCb was a good color space for these bins values: 4 for Y and 20 for Cr and Cb. We have then a greater precision on the chrominance values than on the luminance values, as we do not want to rely on luminance: we want to compare the color information mostly.

We can then apply an algorithm to find the Hough circles in the filtered back projection, with a lower limit on the radius of 10px, and an upper limit on the radius of 60px, as we have a ball whose diameter is about 17cm (which is pretty big).

To select which Hough circles correctly match a ball, we compute the histogram of the region of the image they

circle, and compare it to another histogram of the ball (for example a histogram on HSV). We set a threshold for the distance between those two histograms (at the moment this threshold is 300 for the chi-square distance metric, but the average distance of the histograms from the training set to this average histogram is 1.5 with a standard deviation of 0.2).

Figures 2 shows how the initial image is changed by the back projection.

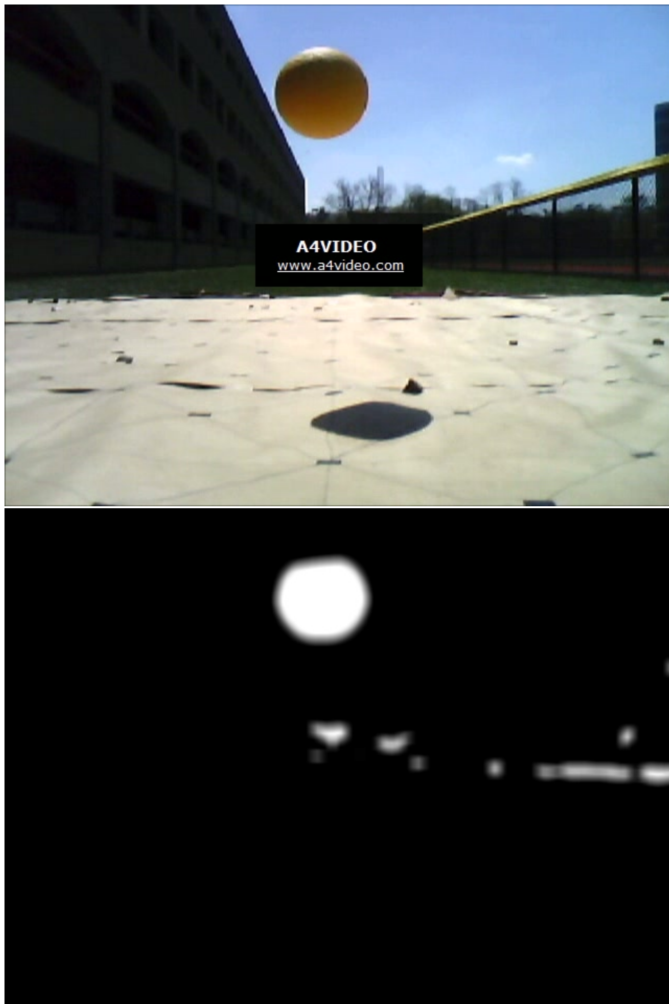

Figure 2: A frame from the video and its thresholded and filtered back projection.

1.4 Filtering The Results

The output of the Hough circle algorithm is a list of parameters of circles in the form of $[x_i,\ y_i,\ d_i]$ where x_i, y_i are the center of the ball and d_i is the diameter. Since the detection process depends heavily on the color, there are many fake circles that are also considered as a ball in the previous section. In order to remove these circles that are not really balls, we perform a filtering using the histogram distance and the process goes as follow:

1. For each circles we generate a binary mask, which is used for extracting the part of the frame that is considered as a ball in the previous step.

2. With the extracted part of the frame, we compute the histogram distance of that portion to the histogram file that we learn from the training data set. With a proper threshold, any circle that is too far from the training data is dropped.

1.5 Kalman filter

We take the following model as being our Kalman filter:

$$
\begin{aligned}
X_{k+1} &= AX_k + w_{k+1} \\
Z_k &= HX_k + v_k \\
X_k &= \begin{pmatrix} x_k \\ y_k \\ z_k \\ x_{k-1} \\ y_{k-1} \\ z_{k-1} \end{pmatrix} \\
Z_k &= \begin{pmatrix} x_k^m \\ y_k^m \\ z_k^m \end{pmatrix} \\
w_{k+1} &\sim N(0, Q) \\
v_k &\sim N(0, R) \\
A &= \begin{pmatrix} (1 + \frac{\Delta\tau_k}{\Delta\tau_{k-1}}) \times I & -\frac{\Delta\tau_k}{\Delta\tau_{k-1}} \times I \\ I & 0 \end{pmatrix} \\
H &= \begin{pmatrix} I & 0 \end{pmatrix} \\
Q &= \begin{pmatrix} \alpha & 0 & 0 & 0 & 0 & 0 \\ 0 & \alpha & 0 & 0 & 0 & 0 \\ 0 & 0 & \beta & 0 & 0 & 0 \\ 0 & 0 & 0 & \alpha & 0 & 0 \\ 0 & 0 & 0 & 0 & \alpha & 0 \\ 0 & 0 & 0 & 0 & 0 & \beta \end{pmatrix} \\
R &= \begin{pmatrix} \gamma & 0 & 0 \\ 0 & \delta & 0 \\ 0 & 0 & \gamma \end{pmatrix}
\end{aligned}
$$

$\Delta\tau_k$ it the time difference between the measurement of x_k and that of x_{k+1}. As we might not have a constant frame rate, we take this into account. Q and R are very simple for the moment but could be improved later on if needed. $\alpha, \beta, \gamma, \delta$ are variance constants, and we choose $\beta > \alpha$ and $\delta > \gamma$ as we have a greater uncertainty along the z direction for the time update and along the y direction for the measurement, which is the direction along which the camera is looking.

1.6 Post-processing

With the coordinates of the two sets of balls in two different frames, the velocity of each ball is simply given by

$$\begin{bmatrix} v_x \\ v_y \\ v_y \end{bmatrix} = \left(\begin{bmatrix} x_t \\ y_t \\ z_t \end{bmatrix} - \begin{bmatrix} x_{t-1} \\ y_{t-1} \\ z_{t-1} \end{bmatrix} \right) \times fps \tag{9}$$

where fps is the frame rate of the video captured from the camera, which is the same as the frequency.

Besides outputting the velocity of the balls for the control program and motion planning, it is necessary to save the position of each balls for computing the velocity in the next frame.

2 The decision algorithm

Our algorithm is given as inputs a position and a velocity for the ball. It is triggered when two conditions are true: the Kalman filter has been fed with at least four measurements and the ball is going to go inside a sphere of radius r within t seconds. We can define r and t as being respectively a fear factor and a reaction time. Those two parameters will need to be tuned when integrating the algorithm in a real robot. We developed two methods to move the robot: an 'optimal' method and a parallelepipedic robot method.

2.1 The 'optimal' method

This method uses a gradient descent algorithm to compute which point of the trajectory parabola will be the closest to the robot. Let us call $\vec{d}$ the unitary direction the robot is going to follow, $\vec{r}_{opt}$ the position of the closest point (the robot is at (0,0,0), and $\vec{v}_{opt}$ the velocity of the ball at this point.

The method has one parameter, $\theta \in [0; \frac{\pi}{2}]$ and we define

$$\vec{d} = -\cos\theta \frac{\vec{r}_{opt}}{\|\vec{r}_{opt}\|} + \sin\theta \frac{\vec{v}_{opt}}{\|\vec{v}_{opt}\|}$$

This definition forces the robot to move away from the parabola while moving in the direction of the ball with an angle of θ. The parameter θ has to be chosen in order to minimize the chances of being hit when the the uncertainty over the position of the ball is high (the ball is going to hit the robot later if it moves in the same direction). We'll come back on this parameter later in the paper.

2.2 The parallelepipedic robot method

In this method, we use a model of a parallelepipedic robot. We define three parameters R_x, R_y, and R_z, which are positive real numbers. They define six planes: $x = \pm R_x$, $y = \pm R_y$, $z = \pm R_z$. We only consider positive time values. For each dimension (x,y, or z) we get four position values that we sort according to the time they are going to be reached with increasing time. We get the first ones for each direction, denote them by x', y', z', and assign a weight to each dimension:

$$w(x) = -\frac{sign(x)}{1 + x^2}$$

The displacement vector $\vec{d}$ is then

$$\vec{d} = \frac{1}{\left\| \begin{bmatrix} w(x') \\ w(y') \\ w(z') \end{bmatrix} \right\|} \begin{bmatrix} w(x') \\ w(y') \\ w(z') \end{bmatrix}$$

.

2.3 Comparison

We compare the execution time of both our algorithms and the dot product of $\vec{d}$ with $\vec{v}_{opt}$ by looking at the mean of those statistics over 48020 trajectories in Table 1 for the mean, Table 2 for the variance, and Table 3 for the median.

We see that the gradient descent is 100 times slower than the intersection algorithm, which is not a surprise. We can also see that tuning the parameters R_{xyz} allow us to roughly control the angle between the optimal velocity and the chosen direction. However, the more we increase R, the more the standard deviation increases. For $R = 0$ the angle is 13 degrees and most of the values are between 0 degrees and 13 degrees (the median value is 10 degrees). We have the same thing for $R = 0.2$ but the maximum angle is smaller. See Figure 3 for a graphical view of two trajectories and two decisions for each of them.

The 'optimal' method is more reliable than the parallelepipedic method, but the latter is still robust and it is much less expensive to compute. Another comparison criteria is the number of parameters to tune, but we could take $R_x = R_y = R_z$ to make it even.

3 Designing a grid for an experiment

3.1 An experimental set

We will be throwing balls from various angles and record it with our computer's webcam which will be on the ground. We set an $n \times n$ square grid around it of length L. We record the time with a timer with absolute error $\Delta\tau$, and register the starting x_0 and ending x_1 points of the ball on the grid as well as the time τ it traveled.

We assume that the ground is a plane and that there is no air resistance. The movement equations for the ball are then:

Data	Optimal	Parallelepipedic ($R = 0.2$)	Parallelepipedic ($R = 0$)
Time (s)	10^{-2}	10^{-4}	10^{-4}
Dot product	$\cos\theta$	0.1141	0.2167

Table 1: Mean values over 48020 trajectories.

Data	Parallelepipedic ($R = 0.2$)	Parallelepipedic ($R = 0$)
Dot product	0.4422	0.3943

Table 2: Variance over 48020 trajectories.

Data	Parallelepipedic ($R = 0.2$)	Parallelepipedic ($R = 0$)
Dot product	0.0772	0.1626

Table 3: Median values over 48020 trajectories.

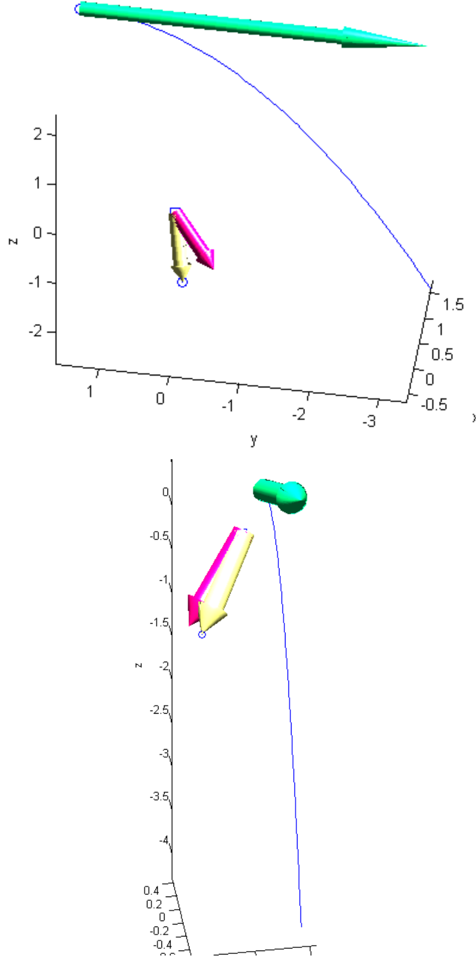

Figure 3: The parabolic trajectory, the 'optimal' direction in gold, and the parallelepipedic direction in purple.

$$\vec{a} = -g \cdot \vec{e}_z$$
$$\vec{v} = -gt \cdot \vec{e}_z + \vec{v}_0$$
$$\vec{x} = \frac{-gt^2}{2} \cdot \vec{e}_z + t \cdot \vec{v}_0 + \vec{x}_0$$

Very simply we get $\vec{v}_0 = \frac{\vec{x}_1 - \vec{x}_0}{\tau} + \frac{g\tau}{2} \cdot \vec{e}_z$ and

$$\vec{v} = -g(t - \frac{\tau}{2}) \cdot \vec{e}_z + \frac{\vec{x}_1 - \vec{x}_0}{\tau} = \vec{f}(\vec{x}_0, \vec{x}_1, \tau; t) \qquad (10)$$

3.2 The measurement error

To get the error for the velocity estimate, we use this formula:

$$\vec{\Delta v} = |\frac{\partial \vec{f}}{\partial \vec{x}_0}|\Delta \vec{x}_0 + |\frac{\partial \vec{f}}{\partial \vec{x}_1}|\Delta \vec{x}_1 + |\frac{\partial \vec{f}}{\partial \tau}|\Delta \tau$$

where $\Delta \vec{x}_0 = \Delta \vec{x}_1 = \begin{pmatrix} \Delta x \\ \Delta x \end{pmatrix}$ with $\Delta x = \frac{L}{n}$. The absolute value here for vectors and matrices means that we take the absolute value of each cell.

From (10) we end up with the following value for the velocity error:

$$\vec{\Delta v} = \frac{2}{\tau} \begin{pmatrix} \Delta x \\ \Delta x \\ 0 \end{pmatrix} + \frac{g}{2}\vec{e}_z - \frac{\vec{x}_1 - \vec{x}_0}{\tau^2}\Delta \tau \qquad (11)$$

From (11) we deduce an upper boundary for the norm of the velocity vector in (12).

$$\|\vec{\Delta v}\| \leq \underbrace{\frac{2}{\tau}\Delta x}_{\epsilon_{position}} + \underbrace{\frac{g}{2}\Delta \tau}_{\epsilon_{vertical}} + \underbrace{\frac{\|\vec{x}_1 - \vec{x}_0\|}{\tau^2}\Delta \tau}_{\epsilon_{horizontal}} \qquad (12)$$

We can then divide our error into three additive components: $\epsilon_{position}$, $\epsilon_{vertical}$ and $\epsilon_{horizontal}$. $\epsilon_{position}$ is exactly the error we make when we don't estimate the starting and ending points correctly (hence the 2). $\epsilon_{vertical}$ is the error on the vertical speed, which only depends on the acceleration g (times the error on time). $\epsilon_{horizontal}$ is the error on the speed on the place (x, y). It is the product of the average horizontal speed $\frac{\|\vec{x}_1 - \vec{x}_0\|}{\tau}$ times the relative error on time $\epsilon_{r,\tau} = \frac{\Delta\tau}{\tau}$.

3.3 Optimization of the set

To optimize our set we want to minimize the upper bound (12) of the norm of the speed vector. We will then balance our three ϵ. We give a more appropriate upper bound for the norm of the error on the speed measurement in (13) by assuming that $\|\vec{x}_1 - \vec{x}_0\| \leq \sqrt{2}L$ which is the length of the diagonal of the grid.

$$\|\vec{\Delta v}\| \leq \underbrace{\frac{2L}{\tau n}}_{\epsilon_{position}} + \underbrace{\frac{g}{2}\Delta\tau}_{\epsilon_{vertical}} + \underbrace{\frac{\sqrt{2}L}{\tau^2}\Delta\tau}_{\epsilon_{horizontal}} \tag{13}$$

First we will take $\epsilon_{position} = \epsilon_{horizontal}$. This yields equation (14).

$$n = \frac{\sqrt{2}}{\epsilon_{r,\tau}} \tag{14}$$

Then we group $\epsilon_{position}$ and $\epsilon_{horizontal}$ into $\epsilon_{planar} = \epsilon_{position} + \epsilon_{horizontal} = 2\epsilon_{horizontal}$. We now want $\epsilon_{planar} = \epsilon_{vertical}$, which yields (15).

$$L = \frac{g\tau^2}{2\sqrt{2}} \tag{15}$$

By choosing $\tau \approx 1$ and knowing that our maximum precision for τ is $\tau = \frac{1}{30}$ (30 is our frame rate), we get:

$$\begin{aligned}
\epsilon_{r,\tau} &= 0.033 \\
n_{opt} &= 42.42 \\
L_{opt} &= 3.47m \\
\Delta x_{opt} &= 0.08m \\
\epsilon_{planar,opt} &= 0.16 \\
\|\vec{\Delta v}_{opt}\| &\leq 0.32
\end{aligned}$$

If a ball is thrown at a speed of $3ms^{-1}$, the error we make when we estimate the speed is roughly 10%.

For practical purposes, here are the dimensions we chose for the grid: $L = 4m$ and $n = 40$.

4 The experiment

4.1 Building the grid

Our grid is made of craft paper. First, we drew a 10-by-10 grid on it.

To get more precision, we drew the diagonal lines joining all the intersections. This gives us a 14-by-14 equivalent grid. But because we have quite accurate eyes, for us this divides each cell into four cells, so we have a 20-by-20 'psychologically' equivalent grid. We can even go further, because the diameter of a ball is 17cm and our 40-by-40 grid has cells of length 10cm, then we can accurately still divide psychologically into smaller cells our grid. Our maximal precision is 8.5cm, the radius of the ball, so we can get up to the equivalent of a 45-by-45 grid for a $\Delta x = 0.085$. One cell is shown on Figure 4, and the real grid is shown on Figure 5.

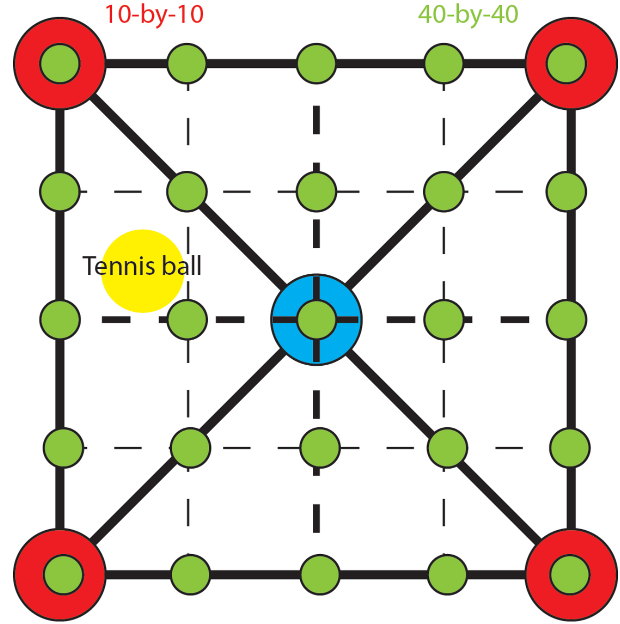

Figure 4: One cell of our grid, which consists of the red and blue vertices and the solid lines. The green vertices represent a grid that the brain can guess (the 100-by-100 grid is not represented here).

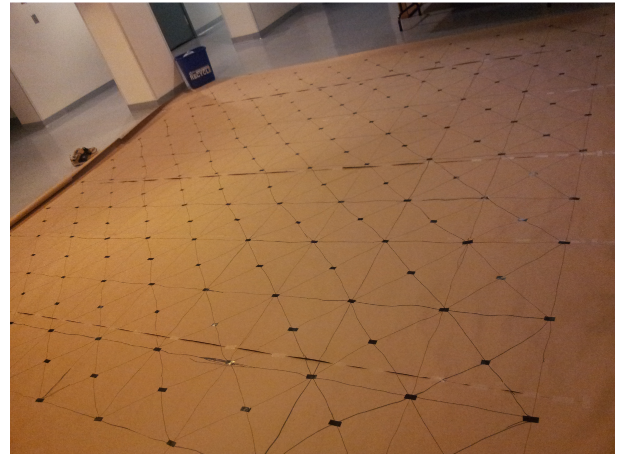

Figure 5: This is the 4m-by-4m grid we built with craft paper.

4.2 The measurements

We put the computer with the webcam on a vertex that we define as the origin of the grid (the point $(0, 0)$). We found it easier to bounce the ball instead of throwing it from one point. We recorded the experiment with another camera two get the positions of the first and second bounces as well as the time between both. We did the experiment in a very simple set up, without too much yellow objects in the frame.

Our results for the five videos are shown on Table 4 page 8.

The two facts that stand out from these figures are: the precision for the position is very good, and the precision for the velocity is awful. The velocity is computed by using the backward Euler method, but the measurements of the positions are very shaky, so the fact that they are good does not compensate the velocity error. We'll discuss in the last section how to improve on the velocity estimation.

Moreover, the position error on the third experiment is bigger than the others. As we can see in the plot of Figure 6 for the second experiment, the ball moved mostly in the medium range and the estimation is close to the actual position. However in the third experiment, when the ball was moving in the far region, the estimation becomes inaccurate and unreliable.

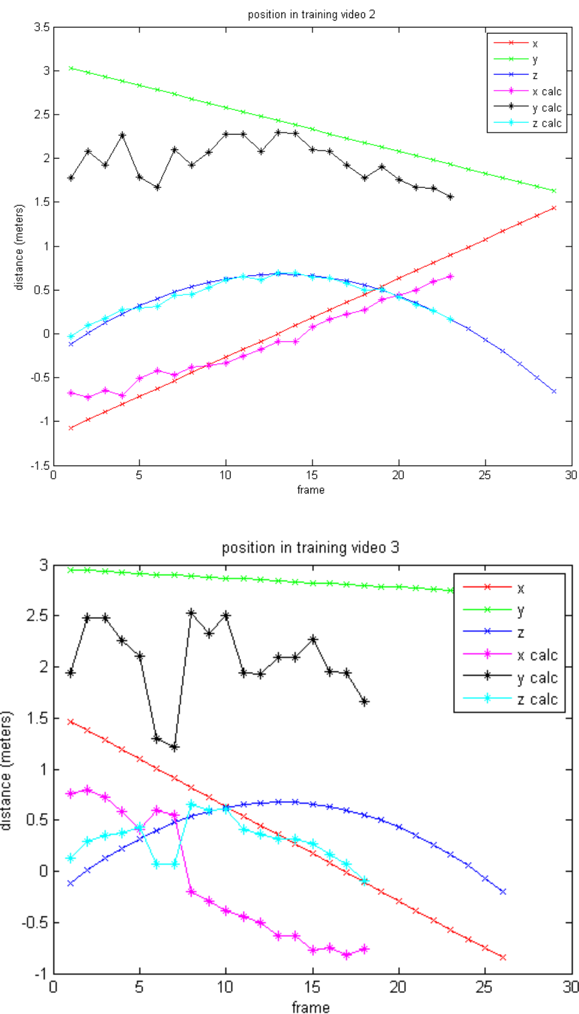

Figure 6: Actual ball position and the corresponding estimation for the second and the third experiments.

5 Future work

Our algorithms are implemented but we still need to plug them into the robot and test them on a real robot. Before that there we must get the velocity estimation right. This will be achieved by either changing the approximation method (for example a semi-implicit Euler method) or filtering the velocity in some way. One would be tempted to add a second Kalman filter to smooth the velocity estimation, but this might be globally equivalent to include in our already existing Kalman filter the velocity of the ball. This means that we would either have two Kalman filters with state dimensions 6 and 3, or one Kalman filter with state dimension 6 or 9.

Another bottleneck error is the error on the Hough circles. We will have to find a more reliable method for ball detection if we want to improve our precision over the position estimation.

At the moment, we tune the decision algorithms by hand, but we could use a method to optimize these parameters with a learning algorithm. The observations would be binary: 'the robot has been hit' and the state continuous. We are currently thinking of using Bayesian probabilities to optimize our parameters.

Acknowledgements

We would like to thank our professor Ashutosh Saxena, the Teatching Assistants for their interesting ideas: Mark Verheggen, Amit Sharma, Abhishek Anand, Hema Koppula, and especially Ian Lenz for his help with the aerial robot simulator.

References

[1] Jeff Michels, Ashutosh Saxena, and Andrew Y. Ng. High speed obstacle avoidance using monocular vision and reinforcement learning. In Luc De Raedt and Stefan Wrobel, editors, *ICML*, volume 119 of *ACM International Conference Proceeding Series*, pages 593–600. ACM, 2005.

[2] Cooper Bills, Joyce Chen, and Ashutosh Saxena. Autonomous mav flight in indoor environments using single image perspective cues. In *ICRA*, pages 5776–5783. IEEE, 2011.

Data	Experiment 1	Experiment 2	Experiment 3	Experiment 4	Experiment 5
$position_{error}$	0.567	0.395	1.449	0.522	0.796
$var(position_{error})$	0.327	0.096	1.025	0.289	0.383
Δv	0.533	0.576	0.606	0.544	0.488
v_{error}	8.994	5.931	25.459	12.811	35.202
$var(v_{error})$	169.196	20.58	749.366	481.694	693.428

Table 4: Our average results for the five experiments and their variance (SI units).