Utilization of Robot Localization and Computer Vision Techniques in the Static Void Ball Recollecting Autonomous

Nicola Muça Cirone, Trey J. Shaffer, and James A. Slade

Vista Ridge HS, Cedar Park TX , USA,
mucacirone.nicola@gmail.com, treyjshaffer@gmail.com, aslade1j@gmail.com
Website Home Page: https://twitter.com/staticvoid6990

Abstract. In this paper we will elaborate on the concepts behind Static Void's avant garde ball recollecting autonomous mode. Through the use of well established robot localization algorithms, such as Kalman filters, and other statistical analysis techniques, alongside the utilization of open source computer vision software, the reader will come to have a nuanced undestanding of the robot's algorithmic systems that allow for ball detection, recollection, velocity determination, etc ...

Keywords: Kalman Filter, Robot Localization, Computer Vision

1 Introduction

Firstly, we will begin to discuss the various issues presented by a ball recollecting autonomous mode. It should be noted that many of the presented problems are actively researched in the academic robotics community, so our provided solutions to them in later chapters should not necessarily be considered the de facto approach for all robotics teams. The subsequent subsections should be seen as an overview of the following chapters, where the given topic will be discussed in much greater depth.

1.1 Robot Localization

Robot Localization is the method of obtaining the robot's position in a field. In our ball recollecting autonomous, we implement a Kalman filter[3.2] for Robot Localization as it allows us to keep an optimal estimate of the robot location. Although it may sound simple, the question "where am I?" is a difficult, yet essential problem to solve in order to have a ball recollecting autonomous.

1.2 Ball Detection

There are many methods of determining the locations of balls in an environment. The most efficient modi utilize computer vision. Detecting objects in an image is very straightforward, however the difficult part of ball detection is differentiating between balls and other objects.

1.3 Path Finding

Path finding is the process of making robot movement decisions in careful consideration of the goals and obstacles on the field. In chapter 5, we discuss in further detail the multiple strategies one can take, which includes the A* heuristic and dynamic approaches, and precisely how these techniques work.

1.4 Filtering Noise

Many of the algorithms at the heart of our robot are centered around the filtering and processing of the information collected by our sensors. They are important because without reliable information of the environment, our robot is unable to respond to properly, and thus will be ineffective in completing any of the tasks it sets out to complete. In chapter 3, we examine the three primary filtering algorithms we considered for our design and how they work.

2 Statistical Foundations

Within this chapter we are going to introduce the fundamentals of statistics and the various concepts needed to understand the upcoming sections. We will start from the basis of this discipline defining mean, range, standard deviation, etc. and we will end up with a solid understanding of a probability function for standard or skewed distributions.

2.1 Basis

Given a list L of numbers such as:

$$L = -10, 0, 10, 20, 30. \tag{1}$$

We define the mean (μ) as the sum of all the elements of L divided by the number of the same elements:

$$\mu = \frac{-10 + 0 + 10 + 20 + 30}{5} = 10 \tag{2}$$

The variance (σ^2) is the average of the squared differences from the mean and gives us a way to measure how much the values differ from each other:

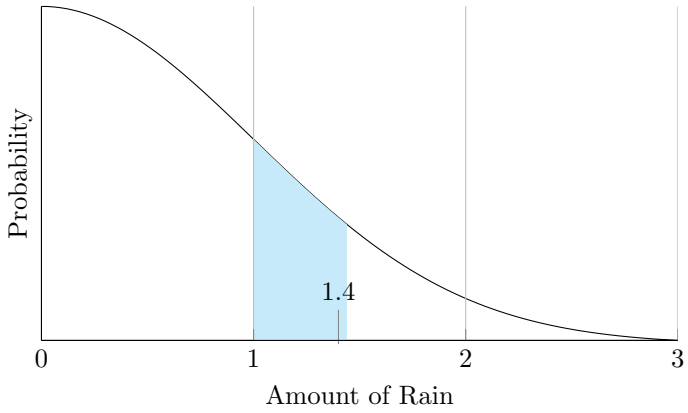
$$\sigma^2 = \frac{(-10 - 10)^2 + (0 - 10)^2 + (10 - 10)^2 + (20 - 10)^2 + (30 - 10)^2}{5} = 200 \tag{3}$$

Finally the Standard Deviation (σ) is the square root of the Variance and it's yet another way to measure how spread out the values are:

$$\sigma = \sqrt{\sigma^2} = \sqrt{200} = 10 * \sqrt{2} \tag{4}$$

2.2 Probability Density Function

A Probability Density Function is a function that returns the probability of a certain event happening. If, for example, we call Y the amount of rain tomorrow and we model a probability function plotting the amount of rain in mm on the x axis and the probability of it raining that exact amount on the y axis like this:



Where the probability that Y is between 1.0 mm and 1.4 mm is given by:

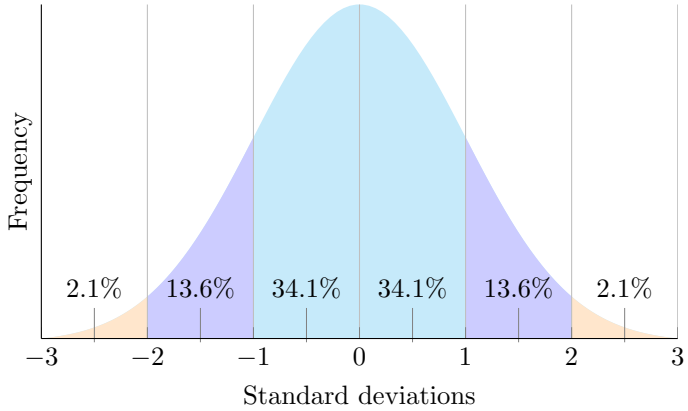
$$P(|Y - 1.2| < .2) = \int_{1.0}^{1.4} f(x)dx \tag{5}$$

Important to remember is the fact that the total probability must always add up to 1:

$$\int_{0}^{\infty} f(x)dx = 1 \tag{6}$$

2.3 Normal Distribution

The normal distribution function is a very common *continuous probability distribution* important is statistics as well as in other fields such as social studies and biology, commonly known as the "bell curve".



The equation for the Normal Distribution Function is:

$$p(x) := \frac{1}{\sigma\sqrt{2\pi}} e^{-(x-\mu)^2/2\sigma^2} \tag{7}$$

Where $(x - \mu)$ is the distance of x from μ and $(\frac{x-\mu}{\sigma})$ is the number of σs away from the mean. In the *Standard Normal Distribution* $\mu = 0$ and $\sigma = 1$. Closely related with the Normal Distribution Function is the *Cumulative Distribution Function* (CDF), a function that returns the probability that a random variate X will be less than or equal than x:

$$CDF(x) = P(X \leq x) := \int_{-\infty}^{x} p(z)dz \tag{8}$$

This means that:

$$P(a < x < b) = \int_{a}^{b} p(x)dx = CDF(b) - CDF(a) \tag{9}$$

2.4 Error Function and $\Phi(x)$

The Error Function (ERF) gives the probability that a positive random number in the Standard Normal Distribution will be in the range [$-x$, x]:

$$ERF(x) := \frac{2}{\sqrt{x}} \int_{0}^{x} e^{-t^2} dt \tag{10}$$

With this function we can get interesting results... The "*Normal Distribution Function*" $\Phi(x)$ gives the probability that a standard normal variate X assumes a value in the interval [0 , x]:

$$\Phi(x) := \int_{0}^{x} p(z)dz = \int_{0}^{x} \frac{e^{\frac{-z^2}{2}}}{\sqrt{2\pi}} dz = \frac{1}{\sqrt{2\pi}} \int_{0}^{x} e^{\frac{-z^2}{2}} dz \tag{11}$$

If we let $u = \frac{z}{\sqrt{2}}$ so that $du = \frac{dz}{\sqrt{2}}$, then:

$$\Phi(x) := \frac{\sqrt{2}}{\sqrt{2\pi}} \int_{0}^{\frac{x}{\sqrt{2}}} e^{-u^2} du = \frac{1}{2} ERF(\frac{x}{\sqrt{2}}) \tag{12}$$

The probability that a normal variate X assumes a value in the interval [x_1 , x_2] is therefore given by:

$$\Phi(x_1, x_2) := \frac{1}{2}(ERF(\frac{x_2}{\sqrt{2}}) - ERF(\frac{x_1}{\sqrt{2}})) \tag{13}$$

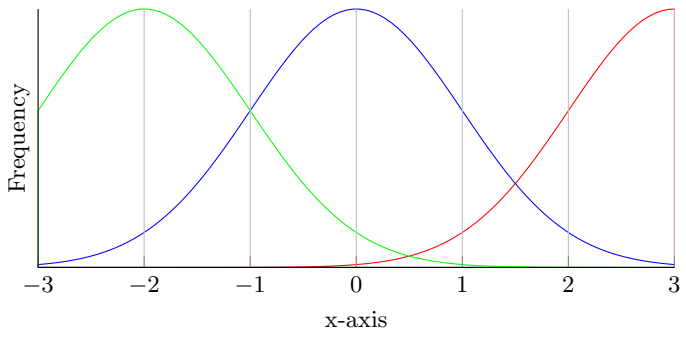
The CDF of a *Standard Normal Distribution*, defined as "THE" Normal Distribution Function ($\aleph(x)$) is:

$$\aleph(x) = \Phi(-\infty, x) := \frac{1}{2}(ERF(\frac{x}{\sqrt{2}}) - ERF(\frac{-\infty}{\sqrt{2}})) = \frac{1}{2}(ERF(\frac{x}{\sqrt{2}}) + 1) \tag{14}$$

An important thing to notice is the fact that Neither $\Phi(x)$ nor $ERF(x)$ can be expressed in terms of finite additions, subtractions, multiplications, and root extractions, and so must be either computed numerically or otherwise approximated.

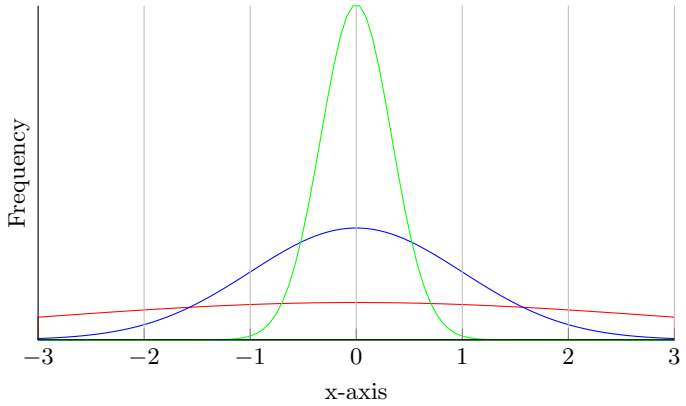
2.5 Skew Normal Distribution

In probability theory and statistics, the skew normal distribution is a continuous probability distribution that generalizes the normal distribution to allow for non-zero skewness. "ξ" denotes *Location*, this parameter simply shifts the graph left ($\xi < 0$) or right ($\xi > 0$):



$\xi = -2$ in the Green graph / $\xi = 0$ in the Blue graph / $\xi = 3$ in the Red graph

"ω" denotes *Scale*, if $\omega > 1$ this parameter will stretch $p(x)$, the greater the magnitude, the greater the stretching . On the other hand if $\omega < 1$ this parameter will compress $p(x)$. As $\omega \to 0$ the compressing approaches a Spike (ω cannot be negative); An $\omega = 0$ leaves $p(x)$ unchanged.



$\omega = \frac{1}{3}$ in the Green graph / $\omega = 1$ in the Blue graph / $\omega = 3$ in the Red graph

In the Standard Normal Distribution Function $\mu = \xi = 0$ and $\sigma = \omega = 1$; in a general distribution function however:

$$x \to (\frac{x - \xi}{\omega}) \tag{15}$$

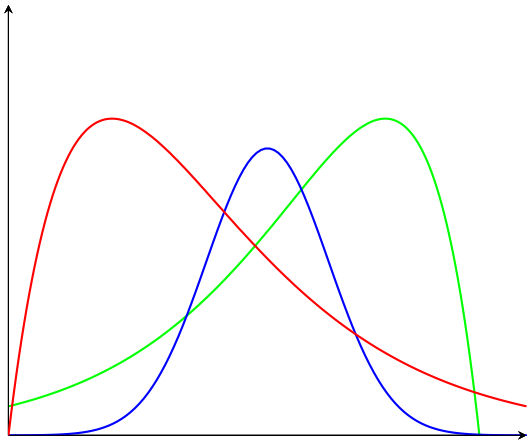
The general functions then are:

$$p(x) := \frac{1}{\omega\sqrt{2\pi}} e^{-(x-\xi)^2/2\omega^2} \tag{16}$$

$$CDF(x) := \int_{-\infty}^{(\frac{x-\xi}{\omega})} p(z)dz \tag{17}$$

$$\Phi(x) := \frac{1}{\sqrt{2\pi}} \int_{0}^{(\frac{x-\xi}{\omega})} e^{\frac{-z^2}{2}} dz \tag{18}$$

"α"denotes the Shape parameter, when $\alpha = 0$ then $p(x)$ stays unchanged; The distribution is *Right Skewed* if $\alpha > 0$ and *Left Skewed* if $\alpha < 0$.



$\alpha < 0$ in the Green graph / $\alpha = 0$ in the Blue graph / $\alpha > 0$ in the Red graph

While "α" may be chosen arbitrarily, the resulting *Skewness* "γ_1" of the distribution (a measure of the *Asymmetry* of the distribution about its mean) is limited to about [-1 , 1].

"δ" is a parameter related to the shape "α" by:

$$\delta := \frac{\alpha}{1 + \alpha^2} \tag{19}$$

and used in the expression for skewness:

$$\gamma_1 = \frac{4-\pi}{2} \frac{(\delta\sqrt{\frac{2}{\pi}})^3}{(1-2\frac{\delta^2}{\pi})^{\frac{3}{2}}} \tag{20}$$

The *Probability Density Function* of a skewed distribution with the parameters "ξ","ω","α" is given by:

$$f(x) = 2p(x)\Phi(\alpha x) = \frac{1}{\omega\pi} \exp(-\frac{1}{2}(\frac{x-\xi}{\omega})^2) \int_{-\infty}^{\alpha(\frac{x-\xi}{\omega})} e^{\frac{-t^2}{2}} dt \tag{21}$$

3 Filtering

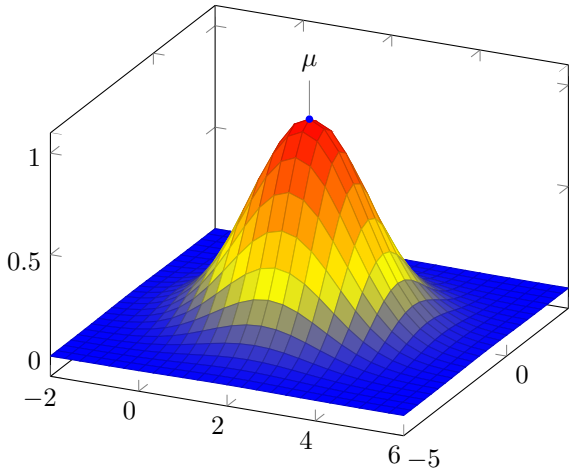
Filtering is the process of removing irregular variations in observations, typically called noise, that obscure the true state of whatever the robot is attempting to measure. We use filters frequently, as our sensors and movement are quite imprecise for the task at hand, and thus, an as-close-as-perfect state estimate is necessary to maintain our autonomous functionality.

3.1 Histogram Filter

During our planning for our robot localization algorithm we considered the Histogram filter, a filter that uses a discrete number of tiles with various statistical weights that are updated according to the state estimate probability model. As tiles may have no correlation depending on implementation specifics, the model is multi-modal, which can be beneficial for detecting objects of which there are multiple copies of on the field, such as particles and other robots.

Although there is no general implementation algorithm for histogram filters, one common design feature used in histogram filters includes a smoothening function, which can be modeled by a simple difference filter:

$$h_t(x) = h(x) - \alpha \frac{d^2 h(x)}{dx^2} \tag{22}$$



Thus allowing the peak probability at μ to be probabilistically smoothened along the nearby cells, as is evident in the above figure, where the smoothening function models a two-dimensional expansion on the Gaussian $\Phi(x, y)$ [2.4]

3.2 Kalman Filter

A Kalman filter is a state estimator, used to optimally determine the state of a given system using observations that are very noisy or inaccurate. The underlying assumptions are that true state and true measurements can be approximated by linear equations and that all noise has a continuous unimodal distribution as according to a Gaussian[2.3] or skewed normal[2.4] distribution.

Assumptions We assume that the true state is modeled by

$$x_t = F_t x_{t-1} + B_t u_t + w_t \tag{23}$$

such that x_t is the state at time t, F_t is the state transition model, and is a linear function of the state, and is thus why the Kalman filter is also be called a Linear Quadratic Estimation (LQE), B_t is the control input model, which models the change in the robot's state over the control vector, u_t. w_t is the process noise, and is random variations in the state as given by the distribution $\Phi(0, Q_t)$ [2.4.11] where Q_t is the covariance of the process noise.

We also assume that the true measurement function can be modeled by

$$z_t = H_t x_t + v_t \tag{24}$$

such that z_t is the true observation of state at time t, H_t is the observation model matrix, with the same width as the state estimate and height of the number of measurements concurrently taken. v_t is the observation noise as given by $\Phi(0, R_t)$, where R_t is the covariance of the observation noise.

Prediction To predict our state estimate, we use

$$\hat{x}_{t|t-1} = F_t \hat{x}_{t|t-1} + B_t u_t \tag{25}$$

Thus applying our state transition model to our previous state estimate based on prior measurements, alongside our control model, to determine the best prediction possible of the state estimate.

Another essential step during our predictions is to calculate the predicted estimate covariance. Given

$$P_t = cov(\hat{x}_t) \tag{26}$$

we can derive that

$$\begin{aligned} P_{t|t-1} &= cov(\hat{x}_{t|t-1}) \\ &= cov(F_t \hat{x}_{t-1}) \\ &= F_t cov(\hat{x}_{t-1}) F_t^T \\ &= F_t P_{t-1|t-1} F_t^T \end{aligned} \tag{27}$$

And thus after adding our covariance of the process noise Q_t, finally yielding

$$P_{t|t-1} = F_t P_{t-1|t-1} F_t^T + Q_t \tag{28}$$

Innovation Now that we have predicted the next state estimate, we must now correct the errors in our predictions during the correction stage, wherein, we use measurement data to refine the state estimate.

To do this, we must first calculate the "innovation", or measurement residual

$$\tilde{y}_t = z_t - H_t \hat{x}_{t|t-1} \tag{29}$$

Thus, with the innovation, we may quantify the discrepancy between the predicted measurements as according to our model and the actual observations.

We can derive the innovation covariance as

$$\begin{aligned} S_{t|t-1} &= cov(z_{t|t-1}) \\ &= cov(H_t \hat{x}_{t|t-1}) \\ &= H_t cov(\hat{x}_{t|t-1}) H_t^T \\ &= H_t P_{t|t-1} H_t^T \end{aligned} \tag{30}$$

After adding in the covariance of the observation noise R_t, we finally yield

$$S_t = H_t P_{t|t-1} H_t^T + R_t \tag{31}$$

Kalman Gain Next, we find ourselves at the heart of the Kalman filter with our calculation of the Kalman Gain.

The Kalman Gain can be thought of as the relative weight given to measurements and the state estimate, and is proportional to process noise, yet inversely proportional to measurement noise, or more formally

$$K_t \sim \frac{process\ noise}{measurement\ noise} \tag{32}$$

This is so because we want our weights to be low if our measurement noise is large relative to our process noise, as this would indicate a very untrustworthy measurement with minimal information gained and thus, we do not want to change our model significantly in response to it.

Yet on the other hand, if the process noise is large relative to the measurement noise, then we know that our prediction model is less correct, and thus we want to have a large weight to change our model more in response to a reliable observation.

We calculate the Optimal Kalman Gain as

$$K_t = P_{t|t-1} H_t^T S_t^{-1} \tag{33}$$

If we have a large Kalman Gain, then we are putting more emphasis on recent measurements in our model's predictions, and if we have a small Kalman Gain, then we consider our filter to be closely modeling the state estimate

Correction Now that we have calculated our Kalman gain, we can use these weights to correct our state estimate with the new measurements accordingly

$$\hat{x}_{t|t} = \hat{x}_{t|t-1} + K_t \tilde{y}_t \tag{34}$$

And then correcting our covariance,

$$P_{t|t} = (I - K_t H_t) P_{t|t-1} \tag{35}$$

where I is the identity matrix.

After this correction, another prediction happens, and then another correction, and then another prediction, and so on and so forth. Thus the Kalman filter algorithm continuously and recursively updates itself over time, allowing for the optimal model of the state estimate to be formed with the given observational data.

3.3 Particle Filter

Similar to other filters, the particle filter is used for state estimations, yet it does so in a very unique way. Instead of having one hypothesis that it continuously edits in order to approach an optimal hypothesis, the particle filter makes many random hypotheses and then tests the various hypotheses, assigning likelihoods to each hypothesis, and then pruning out less likely hypotheses in order to zero-in on what is the most near approximation of the state of the robot.

Assumption The particle filter's state estimate changes according to

$$x_t = f_t(x_{t-1}, v_{k-1}) \tag{36}$$

where x_t is the state estimate at time t, $v_t - 1$ is the noise vector, and f_t is a function describing the evolution of the state estimate.

One can observe that this filter is quite a bit simpler than other filter's we have touched on thus far. It is for this reason that particle filters are popular in modern robot localization algorithms, as the state estimate can be calculated with decent accuracy and at a very low computational cost.

The measurement of the state estimate x_t is given by

$$z_t = h_t(x_t, n_t) \tag{37}$$

where z_t is the measurement vector, h_t is the measurement modeling function, and n_t is the measurement noise.

Importance Sampling In importance sampling, we assign weights to particles according to their approximation of the state estimate. We do this by applying

$$p(x_{0:t-1}|z_{1:t-1}) \approx \sum_{i=1}^{N} w_{t-1}^i \delta_{x_{0:t-1}^i} \tag{38}$$

where $\delta_{x_{0:t-1}^i}$ is a Dirac delta distribution function centered on $x_{0:t-1}^i$ with w_{t-1}^i representing the weights of each particle i in the state estimate.

After the weights are normalized to sum to 1, we must update the state estimate and weights as according to the distribution $q(x_t|x_{t-1}^i, z_t)$

$$x_t^i \sim q(x_t|x_{t-1}^i, z_t) \tag{39}$$

$$w_t^i \propto w_{t-1}^i \frac{p(z_t|x_t^i)p(x_t^i|x_{t-1}^i)}{q(x_t^i|x_{t-1}^i, z_t)} \tag{40}$$

Resampling During resampling, the particle filter creates new hypotheses for the state estimate based on the weights of the particles of the previous generation. Particles with higher weights are more likely to be drawn from the previous generation for the new generation than are other particles. Particles are drawn accordingly:

$$p(x_t|z_{1:t}) \approx \sum_{i=1}^{N} w_t^i \delta_{x_t^i} \tag{41}$$

The particle filter is quite a simple, straightforward approach to localization and object detection as the entire algorithm is centered around the idea of recursively sampling and weighing and resampling again to approximate the state estimate.

4 Computer Vision

In this chapter we dissect the various image processing techniques and detection algorithms used in ball and obstacle detection.

4.1 Image Processing

Smoothing Image smoothing, also called blurring is a simple technique used to remove noise from an image. This is usually the first step of any object detection process. To implement smoothing, we apply a filter to our image. Linear filters are the most used. In a linear filter the value of the output pixel ($g(i,j)$) is a weighted sum of the values of the input pixels($f(i+k,j+l)$). The coefficients of the filter are called the kernel, in this case $h(k,l)$.

$$g(i,j) = \sum_{k,l} f(i+k,j+l)h(k,l) \tag{42}$$

In our application we decided to use a normalized box filter. The output pixels are the average of their kernel neighbors. This filter was chosen because of our desire for fast image processing. For applications where the expeditiousness of results does not hold as much weight, a Gaussian blur or an approximation of a Gaussian blur (through repeated box filters) may yield better results.

Morphological Operations Morphological operations are operations dealing with image shape. In our detection algorithm we used two morphological operations: erosion and dilation. Erosion erodes the boundaries of a foreground object, helping to remove noise. Dilation is the opposite of erosion. We erode the image to remove noise and then dilate the image to counteract the decrement in size caused by erosion.

Object Detection To detect objects in an image, we first use the Ramer-Douglass-Peuker algorithm (expected complexity: $T(n) = 2T(\frac{n}{2}) + O(n)$) to approximate a curve/polygon with another curve/polygon with less vertices. After this approximation is complete, we find the contours in the image. Contours are curves that connect the points (along an edge), having the same intensity or color.

4.2 Object Recognition

Although humans have an easy time recognizing what an object is, computers have no intuitive sense of what constitutes individual things. It is here where a large amount of the difficulty in computer vision lies because it is hard to describe an algorithm that does something humans do automatically and effortlessly, as we have very little idea how to replicate the process in our creations.

Object Differentiation There are many ways to go about object recognition/differentiation. The two solutions we came up with were a Kalman Gaussian Filter and a function to predict the area of a ball based on its x-value in the image (see fig. 1). In our application it was deemed unnecessary to use either of these approaches, as our ball detection algorithm yields very little noise.

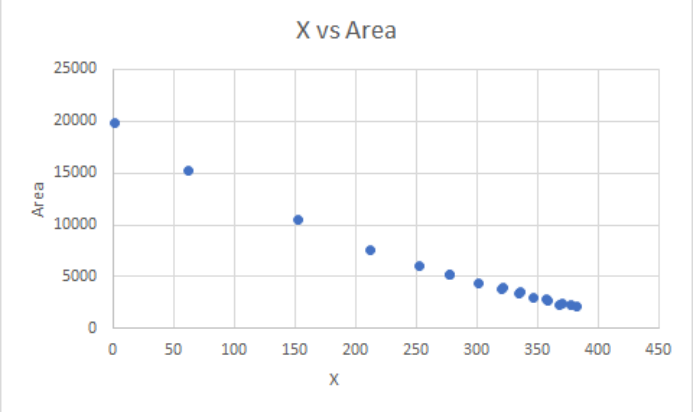


Fig. 1. Scatter Plot of x vs area

This data set can be modeled with the quadratic function:

$$f(x) = 0.06555722809948x^2 - 70.685550859x \ + 19678.458423313 \qquad (43)$$

In order to center the robot on the ball, we created an equation to model the relationship between a ball's x-value in the image and its optimal y-value for being gathered by the robot. We use this function and a precision parameter

to construct a range of y-values that the robot compares to the ball's current y-value in order to determine its course of action.

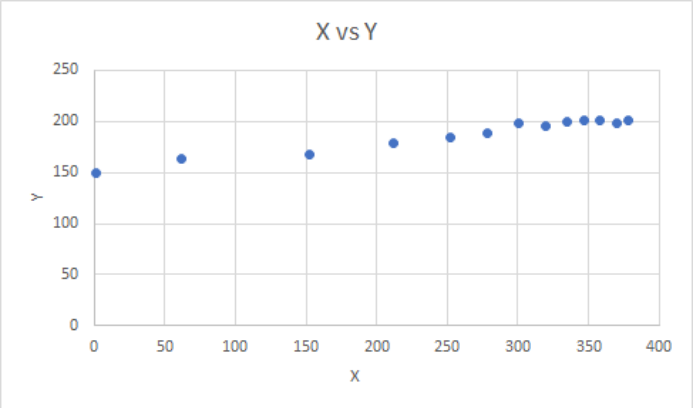


Fig. 2. Scatter Plot of x vs y

This data set can be modeled with the cubic function:

$$f(x) = -6.3412602732455\text{e}{-7}x^3 + 3.8459497753619\text{e}{-4}x^2 + 0.08027377881447x + 151.96016421995 \tag{44}$$

In order for the robot to autonomously intake balls, it needs to know the distance from its position to the position of the ball. We accomplished this by creating a function that predicts the balls distance based on its area in the image.

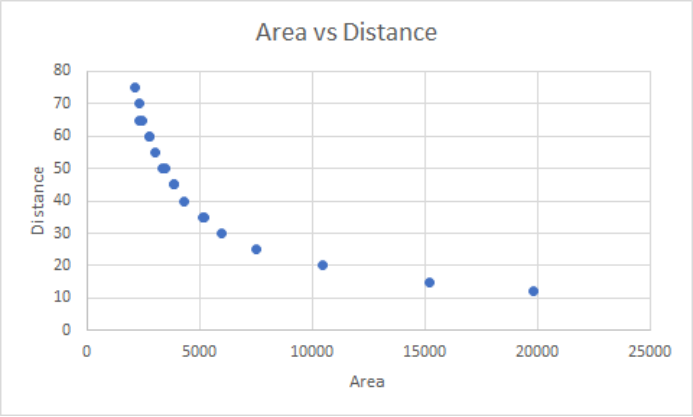


Fig. 3. Scatter Plot of area vs distance

This data set can be modeled with the reciprocal function:

$$f(x) = 39084.482095896x^{-}0.81944876716079 \tag{45}$$

The data we collected is shown in the table below:

X	Y	Area	Distance
1	150	19845	12
62	163	15210	15
152	168	10464	20
212	179	7533	25
252	185	5976	30
278	189	5168	35
278	104	5226	35
301	199	4340	40
320	195	3835	45
322	129	3894	45
335	200	3355	50
336	141	3465	50
347	201	3016	55
358	201	2805	60
359	151	2754	60
370	199	2448	65
368	155	2340	65
378	201	2300	70
382	202	2156	75

Fig. 4. Ball Detection Data

4.3 OpenCV

Our ball detection relies on the functionalities of the OpenCV library. OpenCV is an open source computer vision library that possesses various Image Processing functions. The OpenCV Java API can be found at http://docs.opencv.org/java/3.1.0/

5 Path Finding

5.1 Dijkstra's Algorithm

Dijkstra's algorithm is an algorithm for finding the optimal path between two nodes on a graph. The general algorithm can be understood as:

Dijkstra

```
unvisited = {node: None for node in nodes} #using None as +inf
visited = {}
current = 'B'
currentDistance = 0
unvisited[current] = currentDistance

while True:
    for neighbour, distance in distances[current].items():
        if neighbour not in unvisited: continue
        newDistance = currentDistance + distance
        if unvisited[neighbour] is None or unvisited[neighbour] >
            newDistance:
           unvisited[neighbour] = newDistance
    visited[current] = currentDistance
    del unvisited[current]
    if not unvisited: break
    candidates = [node for node in unvisited.items() if node[1]]
    current, currentDistance = sorted(candidates, key = lambda x:
        x[1])[0]

print(visited)
```

The general idea is that one always traverses the "path of least resistance", similar to many other natural phenomena in physics, such as lightning's path through the atmosphere. Similar to lightning, Dijkstra's algorithm branches out significantly and considers so many nodes, that in cases where the map is large, it is unfeasible for use in practical applications because of it's high computational cost.

5.2 A*

In cases where Dijkstra's algorithm cannot be implemented and it is not necessary that one has the shortest possible path between two points, one can use A* search. A* is quite different from Dijkstra's algorithm as it is a heuristic, meaning that instead of using well-defined precise steps for finding the path, it uses general guiding principles to lead it towards it's goal in a relatively efficient manner, and in this way acts much like a human would. To achieve this heuristic, A* can use an artificial intelligence that one must train with sample data beforehand, so that it may find general patterns in the samples and hopefully give a good estimate of solutions to new problems. We do not yet use artificial intelligence on our robot, although we do plan on implementing Google's TensorFlow for both computer vision and path finding through the use of A* in the latter case.

5.3 Dynamic Programming

Finding the shortest path from a given starting position to an end location is a good start, but it is not very efficient to find a new path every time the robot encounters an obstacle. It is instead better to create a map of actions that allows the robot to start from anywhere in order to get to the goal location (an optimum policy). This is accomplished by first creating a function that returns a map of values representing the cost of movement to get from their location to the goal location.

Dynamic

```
def compute_value(grid,goal,cost):
    value = [[99 for row in range(len(grid[0]))] for col in
        range(len(grid))]
    change = True

    while change:
        change = False
        for x in range(len(grid)):
            for y in range(len(grid[0])):
                if goal[0] == x and goal[1] == y:
                    if value[x][y] > 0:
                        value[x][y] = 0
                        change = True
                elif grid[x][y] == 0:
                    for a in range(len(delta)):
                        x2 = x + delta[a][0]
                        y2 = y + delta[a][1]

                        if x2 >= 0 and x2 < len(grid) and y2 >= 0 and y2 <
                            len(grid[0]) and grid[x2][y2] == 0:
                            v2 = value[x2][y2] + cost
                            if v2 < value[x][y]:
                                change = True
                                value[x][y] = v2

    for i in range(len(value)):
        print value[i]
    return value
```

To determine what action to take at each location, you look at the values of the surrounding grid cells and move in the direction of the lowest value.