K.O.L.T.: Known Object Localization and Tracking

Submitted by:
Nathan Rosenberg, Robotics Engineering

Advised by:
William R. Michalson, Professor
Robotics Engineering

Submitted in partial fulfillment of the requirements for the degree of Bachelor of Science

April 25, 2019
Abstract

An important task performed by many robots is detecting, localizing, and tracking objects in the environment. All sorts of robots—from humanoid robots to autonomous cars and drones—need to be able to find objects around them and track their location. KOLT delivers a drop-in solution to this problem. A software package build for ROS, KOLT consists of a deep neural network for object detection in RGBD images coupled with a Kalman filter for tracking and filtering of detecting objects. The ultimate goal is to develop a drop-in solution for most vision tasks that roboticists encounter.
Acknowledgements

This MQP would not have been possible without the generous support and guidance of my advisor William Michalson. Without his insight and knowledge I could not have completed this project.
Contents

Abstract I

Acknowledgements II

1 Introduction 1

2 Background 2
  2.1 History of Computer Vision ........................................ 2
  2.2 Traditional Object Detection Methods ............................ 2
  2.3 Machine Learning for Object Detection .......................... 3
    2.3.1 Convolutional Neural Networks (CNNs) ........................ 3
    2.3.2 The Inception Network ....................................... 4
    2.3.3 R-CNN .............................................................. 4
    2.3.4 YOLO ............................................................... 5

3 Project Requirements 7

4 Design 8
  4.1 Overall Software Design ........................................... 8
    4.1.1 ROS ................................................................. 8
  4.2 Detection ............................................................. 9
    4.2.1 Detection Pipeline ............................................ 9
    4.2.2 YOLOv2 .......................................................... 10
  4.3 Tracking and Filtering ............................................ 11
    4.3.1 Tracking and Filtering Pipeline ............................... 11
    4.3.2 Hungarian Algorithm ......................................... 12
    4.3.3 Kalman Filter .................................................. 13

5 Results 16
  5.1 Detection Performance ............................................. 16
  5.2 Kalman Filter Performance ....................................... 17
  5.3 Visualization ..................................................... 17

6 Future Work 18

Appendix A API Documentation 20

Appendix B Code Repository 21


List of Figures

2.1 Traditional object detection method for an orange traffic cone. . . . . . . . . 2
2.2 Haar-like facial feature recognition. . . . . . . . . . . . . . . . . . . . . . . . 3
2.3 Max and average pooling. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4
2.4 Inception module, naïve version [1]. . . . . . . . . . . . . . . . . . . . . . . . 5
2.5 YOLO grid showing prediction boxes. . . . . . . . . . . . . . . . . . . . . . . 5
2.6 The YOLO prediction pipeline. . . . . . . . . . . . . . . . . . . . . . . . . . . 6
4.1 High-level KOLT processing pipeline. . . . . . . . . . . . . . . . . . . . . . . 8
4.2 Object classification vs. detection example. . . . . . . . . . . . . . . . . . . 9
4.3 KOLT detection pipeline. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10
4.4 Accuracy and speed on VOC 2007 [2]. . . . . . . . . . . . . . . . . . . . . . 10
4.5 KOLT tacking and filtering pipeline. . . . . . . . . . . . . . . . . . . . . . . 12
4.6 Process noise covariance matrix $Q$. . . . . . . . . . . . . . . . . . . . . . . 14
4.7 Measurement noise covariance matrix $R$. . . . . . . . . . . . . . . . . . . 14
5.1 YOLOv2 traffic cone dataset results. . . . . . . . . . . . . . . . . . . . . . . 16
5.2 YOLOv2 performance with a mostly obscured traffic cone. . . . . . . . . . . 16
5.3 Kalman gain over 1000 timesteps. . . . . . . . . . . . . . . . . . . . . . . . . 17
5.4 Kalman estimated state over 1000 timesteps. . . . . . . . . . . . . . . . . . . 17
5.5 Kalman uncertainty over 1000 timesteps. . . . . . . . . . . . . . . . . . . . . 18
5.6 KOLT visualization. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18
1 Introduction

The concept of a robot, a machine that could do the bidding of its creator, has been around for millennia. While the idea has been around for a while, the technology has not been available to do so. With computers doubling in capacity every 18 months and the artificial intelligence (AI) revolution gaining momentum, this is changing [3]. With this we are constantly picking apart our own brains to try to understand how they work, and why we do the things we do in order to artificially emulate them. Through advanced artificial intelligence, we are able to do things with computers only humans have been able to do until now.

Advancements in AI have allowed for numerous advancements in robotics in the past decade. Traditional algorithms are being replaced with more advanced deep-learning models that offer far more capability and robustness. Deep neural networks allow for robots to become far more adaptable and able to learn from experiences, just as humans do.

In recent years many strides have been made on the computer vision side of AI. Computer vision allows robots to interact with their environment through observation in the same way humans do. This MQP, KOLT, attempts to bring us one step closer to this goal by combining object detection with 3D localization and tracking. It is designed with the goal of allowing robots to interact with their environment by finding objects and their positions.

KOLT (known object localization and tracking) combines a deep convolutional neural network with a tracker and Kalman filter and is able to localize and track multiple objects at once. It is designed as a complete vision package for roboticists at WPI completing their MQPs that require a vision solution as part of their project. For example, there were four other robotics MQPs this year that needed a solution that KOLT could have provided. Because creating a computer vision system is very time consuming and is usually not in itself the final product for an MQP, the goal of KOLT is for it to empower these projects and allow them to be more successful by giving groups needing them more time to work on the rest of their project.
2 Background

This section looks at the background of computer vision and the use of machine learning. This section also explores some of the technologies used in KOLT, such as YOLOv2 and convolutional neural networks (CNNs).

2.1 History of Computer Vision

In the late 1960s computer vision started as research at universities as a novel way of empowering robots with human-like abilities. In 1966 this began as a summer research project at MIT by having a computer describe what it saw through a connected camera [4].

At the time the main difference between computer vision and the prevalent field of image processing was the goal to gain a three-dimensional structure from images to be used to detect objects. These algorithms for edge-detection, background extraction, and optical flow are still used today [5].

Research continued in the 1990s to include shape recognition from detected blobs and edges. This was also the time when the concept of camera calibration came about when work on 3D-reconstruction from multiple images began. Towards the end of the 90s, work on a software library under the umbrella of Intel began that would later become known as OpenCV [6].

2.2 Traditional Object Detection Methods

Before machine learning was used, object detection was a much harder task. Object detectors used hand-picked features to make detections. This would usually involve filtering the image, extracting potential objects as blobs, and doing a hard-coded feature detection. For example, if orange traffic cones were the object that needed to be detected, first a color filter would be applied to the image to find “orange”. Next, the resulting image would be put through a threshold filter to clearly distinguish between the “orange” object and the background. Finally, a blob detector would be run on the resulting image to find the position of the detected “orange” object.

This method works fine indoors where the lighting is usually consistent, but when this same detection model is taken outdoors, it has no guarantee of working. In addition, this technique does not take into account the shape of the detected object or any other potential

Figure 2.1: Traditional object detection method for an orange traffic cone.
features. As such, it can easily mistake another orange object as an orange traffic cone and provide a false positive detection.

2.3 Machine Learning for Object Detection

A lot of the initial work on object detection methods using machine learning began with facial detection. These initial efforts can be classified as feature-based and appearance-based. Feature-based detection methods focus on identifying specific features within the image, such as the eyes, nose, and mouth and their location with respect to one another. Appearance-based methods look through grid sections of an image to find faces. These methods often combined with more advanced techniques in what is known as a cascade [5].

2.3.1 Convolutional Neural Networks (CNNs)

Convolutional neural networks (CNNs) are a type of deep neural network that is specifically inspired by the visual cortex in animals [7]. CNNs offer a great improvement over traditional object detection methods because instead of using hand-engineered features and filters, they learn and apply these themselves. They are able to take an unaltered image and produce a classification from what they see.

CNNs are different from normal deep neural networks (DNNs) in that they are able to make use of spacial and temporal properties of an image. CNNs are also faster at making predictions on images. Whereas a DNN will make use of every pixel in an image, a CNN will reduce the number of parameters to better fit an image. This is important because images are highly complex with an extreme amount of variance.

CNNs typically consist of two parts:

• **Convolution layers**: Extract features from an input, reducing the input complexity. This step is essential to providing fast predictions.
• **Pooling layers**: From the output of the convolution layer, these layers determine the dominate features. Pooling layers come in two types:

  – *Average pooling*: Returns the average of all the values from the part of the image covered by an output from the convolution layer.
  
  – *Max pooling*: Returns the maximum value from the part of the image covered by an output from the convolution layer.

![Max and average pooling](image)

In addition to returning the maximum value from each portion of the image, the max pooling layer performs a noise suppression in that it does not include the activations that are not as strong. Because max pooling performs de-noising in addition to dimensionality reduction rather than dimensionality reduction as noise reduction, max pooling usually performs better than average pooling.

**2.3.2 The Inception Network**

Before the Inception network, when better performance was required from a CNN, more layers were added. This, however, can lead to several problems. Deeper networks can be easily overfit to data and lose the ability to generalize. The inception network took a smarter approach. Instead of only adding layers and hoping for higher performance, inception took the approach of adding multiple filters per layer, essentially *widening* instead of *deepening*.

Using this inception network, a larger neural network was built and was known as GoogLeNet, or Inception v1. Within it are nine inception layers, totaling 22 layers.

**2.3.3 R-CNN**

One of the first real-time object detection and bounding box estimation methods using CNNs was R-CNN [8]. This system runs a CNN on each region of interest in order to classify them.
For each image received, there could be a thousand predictions made. This results in a very low framerate even with large amounts of processing power.

A 10x speed improvement was made with Faster R-CNN, but still operates at the 1-2 FPS (Frames Per Second) level, which is still not enough to be helpful for any real-time use case [9].

### 2.3.4 YOLO

The YOLO (You Only Look Once) algorithm goes in a different direction from the work R-CNN and Faster R-CNN have done. YOLO uses a modified version of the Inception network and, as its name suggests, it only runs a single prediction over each image. Doing this drastically increases the efficiency and thus the performance over the other two. The way YOLO does this is by splitting an input image into a grid of size $S \times S$ and doing the following on each:

![YOLO grid showing prediction boxes.](image)
• It predicts \( B \) number of boundary boxes, each having a confidence score.
• It detects only one object regardless of the number of boxes.
• It predicts \( C \) number of class probabilities.

Because of this, the prediction for the YOLO algorithm has a shape of \((S, S, Bx5+C)\) which when using the standard values it becomes \((7, 7, 30)\) \([2]\). YOLO then uses a CNN

![Figure 2.6: The YOLO prediction pipeline.](image)

network to reduce the dimensionality to 7x7 with 1024 outputs at each. A linear regression is performed to get 7x7x2 boundary box predictions, and a high-pass filter is used to get high box confidence scores for the final prediction.

YOLOv2 is an updated version of the original YOLO algorithm with several speed and feature improvements.
3 Project Requirements

A common task required for robots is the ability to find and track objects in three-dimensional space. Currently, at WPI there exist numerous robotics projects that require this capability but the groups working on them do not have the time required to build a robust software package to be able to do this. To help with this problem, this MQP has been tasked with the creation of a software package to be able to be used in other projects that has the ability to:

• Robustly detect multiple classes objects in real-time from a video stream.
  – Achieve a framerate $\geq 25\text{fps}$ with a desktop GPU.
  – Achieve a mAP score of $\geq 45.0$.

• Robustly find bounding boxes for detected objects in each video frame.

• Track detected objects across video frames.

• Filter out extraneous detections and produce an estimated position for each object in three dimensions.

• Support multiple architectures.
  – Both CPU and GPU compute capabilities. This is important on the NVIDIA Jetson which is an embedded computer with a CUDA-enabled GPU.

• Be easily adaptable and configurable to support the needs of a variety of applications.
4 Design

This section details the design and inner workings of KOLT.

4.1 Overall Software Design

There are three main parts to KOLT:

- **Detection**: The searching through a video stream to find target objects, such as ‘person’ or ‘traffic cone’ and finding their bounding box in each image.

- **Tracking**: Once an object has been detected in each individual image, each detection in each frame needs to be correlated to a detection in a previous frame. These are known as tracked objects.

- **Filtering**: Once detections are matched up to a tracked object the detected positions need to be filtered using a Kalman filter. This process filters out noise and extraneous detections.

![High-level KOLT processing pipeline.](image)

Detection, tracking, and filtering comes together to create an object detection and tracking pipeline as seen in figure 4.1.

4.1.1 ROS

KOLT was built on top of ROS (Robot Operating System) so it could be integrated into existing robotics projects easily. In the creation of a ROS-compatible YOLOv2, a few modifications and adaptations were made to the original algorithm, which was written in C++ using a custom framework called Darknet [10]. Instead of this framework, it was decided to use Python 2.7 (as it is included with ROS) and the Keras framework with a TensorFlow backend. These tools are widely used and would thus further aid in a software package that could be easily extended and integrated into other projects.
4.2 Detection

It was decided early on that instead of using traditional computer vision techniques for object detection, a deep neural network would be used. The specific neural network architecture selected is YOLOv2. This architecture is specifically an object detection framework, not an object classification framework, an important distinction. An object classification framework takes in a whole image and classifies it.

![Object Classification](image1.png) ![Object Detection](image2.png)

Figure 4.2: Object classification vs. detection example.

Figure 4.2 (a) is an example of a classification. The whole image would be labeled as ‘cat’. However, figure 4.2 (b) is an example of a detection. The object is found in the image, usually with a bounding box.

4.2.1 Detection Pipeline

During the design of the detection pipeline several considerations were made. The first was the ability to be able to detect objects in multiple connected cameras. For example, if there was a robot with four cameras mounted on top giving it a 360° view and the ability to detect and track objects in each was needed.

The second was the need to have the neural network loaded in memory only once. The YOLOv2 network is very large and takes a significant amount of computational resources. Most, especially embedded, computers cannot handle multiple instances running at once. In addition, loading weights into a GPU’s memory takes a relatively long time. So, if real-time performance is required it is important to have memory pre-allocated beforehand.

The last consideration was the need to make the detection in a separate process. This is important because it allows one to restart only the detector without needing to restart the rest of the KOLT pipeline.

With these considerations the chosen architecture consists of a detection service that takes in an image and produces an array of detections. The advantage of this is, as the name suggests, the detector runs in a separate process, taking in a queue of images to run the detector on. This can be seen in figure 4.3.

Each RGBD camera is paired with a detector node so that for n cameras there are n detection nodes. Each of these detection nodes has a persistent connection to a detection
service proxy. Each detection node then publishes a detection message to a shared `rostopic` with the ID of the associated camera attached to each.

### 4.2.2 YOLOv2

The implementation of YOLOv2 in KOLT, as said before, is built with Keras with a TensorFlow backend. YOLOv2 was chosen because of its high framerate and high accuracy. This can be seen in figure 4.4 where it was tested on the VOC 2007 dataset.

![Accuracy and speed on VOC 2007](image)

YOLOv2 consists of a 24 layer convolutional neural network with two fully connected layers at the end. The last convolution layer has an output with the shape $(7, 7, 125)$ [2].
With YOLOv2 there are multiple predictions per grid cell, however we want to find the true positive prediction. To do this, we select the grid cell with the highest IoU (intersection over union) and the ground truth training labels. The sum-squared error between predictions and ground truth is used to calculate loss [2]. There are three parts to the loss function:

- **Classification loss**: If an object is detected the loss is the sum-squared error of the conditional class probabilities.
- **Localization loss**: The error between the predicted box and the ground truth.
- **Confidence loss**: The objectness of the predicted box. If an object is detected within the box there is one loss function, if not then there is another.

These parts are combined into the following loss function

\[
\lambda_{\text{coord}} S^2 \sum_{i=0}^{B} \sum_{j=0}^{B} L_{ij}^{\text{obj}} \left[ (x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2 \right] \\
+ \lambda_{\text{coord}} S^2 \sum_{i=0}^{B} \sum_{j=0}^{B} L_{ij}^{\text{obj}} \left[ \left( \sqrt{w_i} - \sqrt{\hat{w}_i} \right)^2 + \left( \sqrt{h_i} - \sqrt{\hat{h}_i} \right)^2 \right] \\
+ \sum_{i=0}^{B} \sum_{j=0}^{B} L_{ij}^{\text{obj}} \left( C_i - \hat{C}_i \right)^2 \\
+ \lambda_{\text{noobj}} S^2 \sum_{i=0}^{B} \sum_{j=0}^{B} L_{ij}^{\text{noobj}} \left( C_i - \hat{C}_i \right)^2 \\
+ \sum_{i=0}^{B} L_{i}^{\text{obj}} \sum_{c \in \text{classes}} (p_i(c) - \hat{p}_i(c))^2 \tag{1}
\]

In training, three warm up epochs are performed during which the sizes of the five boxes in each cell are forced to match the sizes of the 5 anchors. After warm up the network is trained until the loss does not change over three epochs.

### 4.3 Tracking and Filtering

Once a detection has been made, it is fed to the tracking and filtering step. In this step detections are correlated to known tracked objects and their position estimated using a Kalman filter.

#### 4.3.1 Tracking and Filtering Pipeline

The tracking and filtering step is handled by the `vision_pose` node. This node waits for new detections to be published and processes them by:

1. Finding the centroid of each detected bounding box.
2. Using the centroid and the associated RGBD image to get its \((x, y, z)\) position in the camera’s frame.

3. Using a tracker to correlate each new calculated position to already tracked objects using the Hungarian algorithm.
   - If a new position can not be correlated to an already tracked object, a new tracked object will be created with a unique integer ID.
   - Each tracked object has its own Kalman filter that filters incoming raw poses and produces a predicted position.

4. The filtered poses get published as a pose array with each pose having an orientation of \((0, 0, 0, 0)\).

This process can be seen in figure 4.5.

### 4.3.2 Hungarian Algorithm

The Hungarian algorithm is used to correlate new detections to tracked objects. This algorithm takes in a bipartite graph of the sum of the square distance between the last filtered pose and the current detected pose.

\[
C = \begin{bmatrix}
\sqrt{\sum (d_1 - t_1)^2} & \ldots & \sqrt{\sum (d_m - t_1)^2} \\
\vdots & \ddots & \vdots \\
\sqrt{\sum (d_1 - t_n)^2} & \ldots & \sqrt{\sum (d_m - t_n)^2}
\end{bmatrix}
\]  

(2)

Equation 2 shows this where \(C\) is a \(n \times m\) matrix in which \(n\) is the number of tracked objects, \(m\) is the number of detections, \(d\) is a horizontal vector of raw detections, and \(t\) is a vector of the last tracked positions.

Let \(X\) be a boolean matrix where \(X[i, j] = 1\) iff \(i\) is assigned to \(j\) in the bipartite graph. The optimal assignment for this graph would is shown in equation 3

\[
\min \sum_i \sum_j C_{i,j} X_{i,j}
\]  

(3)

where each row is assignment to at most one column, and each column to at most one row.
4.3.3 Kalman Filter

Filtering is necessary because raw detections can be noisy, give false-positives, or lose sight of the tracked object. In these circumstances, raw detections need to be fed into a Kalman filter to predict the actual pose of a tracked object.

The Kalman filter in KOLT uses a CV (constant velocity) model, which assumes the velocity of the tracked object remains constant. This model is ideal for stationary objects and the short-term tracking of moving objects. If more functionality is needed from the filter then a more complicated filter will be required, such as an adaptive Kalman filter or an extended Kalman filter (EKF).

The Kalman filter’s state vector is

\[
x_k = \begin{bmatrix} x & y & z & \dot{x} & \dot{y} & \dot{z} \end{bmatrix} = \begin{bmatrix} \text{X position} \\ \text{Y position} \\ \text{Z position} \\ \text{X velocity} \\ \text{Y velocity} \\ \text{Z velocity} \end{bmatrix}
\]  

where the projected state is

\[
x_{k+1} = A \cdot x_k
\]

which is

\[
x_{k+1} = \begin{bmatrix} 1 & 0 & 0 & \Delta t & 0 & 0 \\ 0 & 1 & 0 & 0 & \Delta t & 0 \\ 0 & 0 & 1 & 0 & 0 & \Delta t \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ \dot{x} \\ \dot{y} \\ \dot{z} \end{bmatrix}_k
\]

The observation model is

\[
y = H \cdot x
\]

which is

\[
y = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix} \cdot x
\]

The velocity of the tracked object has the possibility of changing over time, such as changing directions. This is known as process noise, \( Q \). \( Q \) can be calculated as

\[
Q = G \cdot G^T \cdot \sigma_v^2
\]

with \( G = \begin{bmatrix} 0.5dt^2 & 0.5dt^3 & d \tau & dt & dt & dt \end{bmatrix}^T \) and \( \sigma_v \) as the velocity process noise. \( Q \) ends up being

\[
Q = \begin{bmatrix} 0.25\Delta t^4 & 0.25\Delta t^4 & 0.25\Delta t^4 & 0.5\Delta t^3 & 0.5\Delta t^3 & 0.5\Delta t^3 \\ 0.25\Delta t^4 & 0.25\Delta t^4 & 0.25\Delta t^4 & 0.5\Delta t^3 & 0.5\Delta t^3 & 0.5\Delta t^3 \\ 0.25\Delta t^4 & 0.25\Delta t^4 & 0.25\Delta t^4 & 0.5\Delta t^3 & 0.5\Delta t^3 & 0.5\Delta t^3 \\ 0.5\Delta t^3 & 0.5\Delta t^3 & 0.5\Delta t^3 & \Delta t^2 & \Delta t^2 & \Delta t^2 \\ 0.5\Delta t^3 & 0.5\Delta t^3 & 0.5\Delta t^3 & \Delta t^2 & \Delta t^2 & \Delta t^2 \\ 0.5\Delta t^3 & 0.5\Delta t^3 & 0.5\Delta t^3 & \Delta t^2 & \Delta t^2 & \Delta t^2 \end{bmatrix}
\]
which looks like figure 4.6. The measurement noise covariance, $R$, tells the Kalman filter how inconsistent the position measurements are from the camera. This ends up being

$$R = \begin{bmatrix} \sigma_x^2 & 0 & 0 \\ 0 & \sigma_y^2 & 0 \\ 0 & 0 & \sigma_z^2 \end{bmatrix}$$  \hspace{1cm} (11)$$

where $\sigma$ is the standard deviation of the measurements. This ends up looking like figure 4.7.

The dynamics of the system need to be modeled in order for the Kalman filter to make a prediction. These are as follows

$$x_{k+1} = x_k + \dot{x}_k \cdot \Delta t \hspace{1cm} (12)$$

$$y_{k+1} = y_k + \dot{y}_k \cdot \Delta t \hspace{1cm} (13)$$

$$z_{k+1} = z_k + \dot{z}_k \cdot \Delta t \hspace{1cm} (14)$$

$$\dot{x}_{k+1} = \dot{x}_k \hspace{1cm} (15)$$

$$\dot{y}_{k+1} = \dot{y}_k \hspace{1cm} (16)$$

$$\dot{z}_{k+1} = \dot{z}_k \hspace{1cm} (17)$$
The initial state, $x_0$, needs to be set before the Kalman filter can make a prediction. The initial state is

$$x_0 = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$  \hspace{1cm} (18)$$

where both the position and velocity are 0. In addition, the initial uncertainty, $P_0$, also needs to be set as this also changes over each timestep. The initial uncertainty is

$$P_0 = \begin{bmatrix} \sigma_x^2 & 0 & 0 & 0 & 0 & 0 \\ 0 & \sigma_y^2 & 0 & 0 & 0 & 0 \\ 0 & 0 & \sigma_z^2 & 0 & 0 & 0 \\ 0 & 0 & 0 & \sigma_x^2 & 0 & 0 \\ 0 & 0 & 0 & 0 & \sigma_y^2 & 0 \\ 0 & 0 & 0 & 0 & 0 & \sigma_z^2 \end{bmatrix}$$  \hspace{1cm} (19)$$

where $\sigma$ is the standard deviation.

At each timestep the Kalman filter makes a prediction and a correction to the state. During the prediction step the following calculations are made in this order:

1. Project the state ahead

$$x_{k+1} = A \cdot x_k$$  \hspace{1cm} (20)$$

2. Project the error covariance ahead

$$P_{k+1} = A \cdot P_k \cdot A^T + Q$$  \hspace{1cm} (21)$$

Next, during the correction step the following calculations are made in this order:

1. Compute the Kalman gain $K$

$$S_k = H_k \cdot P_k \cdot H_k^T + R_k$$
$$K_k = (P_k \cdot H_k^T)S_k^{-1}$$  \hspace{1cm} (22)$$

2. Update the state estimate via $z$

$$y_k = Z_k - (H_k \cdot x)$$
$$x_k = x_k + (K_k \cdot y)$$  \hspace{1cm} (24)$$

3. Update the error covariance

$$P_k = (I_k - (K_k \cdot H_k))P_k$$  \hspace{1cm} (26)$$
5 Results

Throughout designing and testing, KOLT has been able to meet all of the original design requirements.

5.1 Detection Performance

Detections with KOLT are made at $33 \pm 3$fps on a desktop GPU using the full YOLOv2 network. On an embedded computer this would be much lower. For this application the Tiny YOLOv2 network would work better and be able to achieve real-time performance.

After being trained on a custom traffic cone dataset of 850 images, YOLOv2 was able to easily the cones from test images and achieves a mAP score of 48.1. It is even able to find traffic cones that are mostly obscured and discolored, such as in figure 5.2.

Figure 5.1: YOLOv2 traffic cone dataset results.

Figure 5.2: YOLOv2 performance with a mostly obscured traffic cone.
5.2 Kalman Filter Performance

The Kalman filter with a CV model has fairly good performance, especially with stationary targets. With moving targets, especially ones that change direction, the predictions tend to lag behind but eventually they catch up.

Testing with gaussian noise of a $\sigma = 0.3$, the Kalman filter has a gain over 1000 timesteps as shown in figure 5.3. The estimated state settles to $\mu = 0.0$ over $\approx 20$ timesteps. This can be seen in figure 5.4. The uncertainty can be seen in figure 5.5 Because the $\sigma$ is the same for all axes, the uncertainty is the same for all three, for both position and velocity. With KOLT this is not the case. The Z-axis tends to have a higher $\sigma$ than the other axes because the RGBD camera has more noise in depth than in the image.

![Figure 5.3: Kalman gain over 1000 timesteps.](image1)

![Figure 5.4: Kalman estimated state over 1000 timesteps.](image2)

5.3 Visualization

An important component of any vision system is the visualization system for debugging purposes. KOLT uses Rviz as the visualization tool with the position of the detected objects shown as axes and the path as a marker array. This can be seen in figure 5.6.
Future Work

While KOLT was ultimately successful and was able to meet all project requirements, there is still a lot of work that can be done. This ranges from performance improvements to stability upgrades, to feature additions and technology updates. Here is a list of potential future work:

- In pursuit of real-time applications, add a priority to each detection submitted to the prediction server a priority queue. This will allow more important images to be analyzed first and achieve a higher framerate. This would be helpful, for example, if there were two cameras on a robot with one facing forwards and one backwards. It might be important for the forward-facing camera to have a higher framerate than the backwards-facing one.

- Implement a FIFO queue for path point storage within the tracking code, which would
reduce number of for-loops required.

- Vectorize everything with Numpy while keeping data on the GPU’s memory with something like Cupy, a CUDA-accelerated Numpy library. In addition, simplifying the prediction pipeline would aid in increasing performance.

- The largest performance bottleneck is the YOLOv2 prediction step. The KOLT implementation of YOLOv2 is significantly slower than the Darknet implementation. Something to look into is specifying the prediction batch size or implementing the Keras `predict_generator` function into the prediction service.

- Modification of the base YOLOv2 algorithm to allow for the input and training on RGBD data to try and enhance the accuracy of detections. This will also allow for the possibility of 3D bounding box detection and orientation.
A API Documentation

API documentation can be found on the Github Wiki page for KOLT. This resource will be continually updated as more is added to the code base. This can be found at: https://github.com/diggerdata/kolt_ros/wiki
B Code Repository

The code for KOLT can be found on Github. As this is an active project, the code base will be continually updated. The lastest code can be found at: https://github.com/diggerdata/kolt_ros
References


