Spring 2017

Sensor Characterization and Signal Fusion for InstantEye

Wyman T. Smith
University of New Hampshire, Durham, smithwyman@gmail.com

Sensor Characterization and Signal Fusion for InstantEye

Honors Thesis for Engineering Physics

Wyman Smith
Faculty Advisor: Dr. Marc Lessard

Department of Physics
University of New Hampshire
Durham, NH

May 20, 2017

Abstract

The practicality and effectiveness of using a TerraRanger Duo—a parallel sonar and infrared time-of-flight distance sensor—payload for obstacle detection is investigated for use with Physical Science Inc.'s InstantEye drone. A Python program was developed to interface with the serial data output before comparing the sensor's empirical performance against its data sheet. The two signals from the distinct sensor modules, each with their characterized strengths and weaknesses, were then fused with a Kalman filter. This was further refined by imposing conditional weighting based on the known sensor characteristics. The filter output, with conditional corrections, was able to accurately track a single object's position and velocity within a maximum range of 14 meters.

Table of Contents

1 Introduction 3

1.1 Kalman Filter Overview 3

2 Sensor Characterization 5

2.1 USB Interface 6

2.2 Sensor Characteristics 6

3 Signal Fusion 9

3.1 Kalman Filter Design 10

3.2 Filter Tuning 12

4 Future Work 13

Acknowledgements 14

References 14

Appendix A: Serial Interface Code 15

A.1 sensor_read.py 15

A.2 ie_sense.py 15

Appendix B: Kalman Filter Code 17

1 Introduction

InstantEye is a high-performance, low-cost drone developed by Physical Sciences Inc. (Andover, MA) for military and commercial applications. By adding aftermarket payloads to the base vehicle, the functionality of InstantEye can be expanded and tailored to specific applications. Existing supported payloads include infrared floodlights, thermal imagers, and terrain mapping camera systems. This report examines a prospective payload that is intended to augment InstantEye's ability to navigate in close-quarters around large objects for military reconnaissance exercises. Specifically, we investigate the characteristics of the TerraRanger Duo distance sensor, by TerraBee (Saint-Genis-Pouilly, France).

The TerraRanger Duo sensor package houses both an infrared (IR) Time-of-Flight (ToF) rangefinder and a sonar module operating in parallel. We will refer to the infrared sensor as both the IR sensor and the ToF sensor for the remainder of this report. The TerraRanger Duo returns data via an included USB serial interface. Both sensors exhibit distinct properties and vary in their ability to handle different environments. For example, the ToF rangefinder has greater distance resolution, but struggles in broad daylight or when observing a transparent or reflective surfaces. The sonar, on the other hand, does not suffer from performance degradation under these conditions, but it does have issues when pointed at anechoic surfaces and corners.

This insight lends itself naturally to the signal fusion problem of creating a robust and accurate signal from the two imperfect sources. Ultimately, it was decided that a Kalman filter would provide the necessary flexibility, computational efficiency, and accuracy to reliably fuse the two signals. The information at the output of the filter can be used as an independent means of collision avoidance to distinguish obstacles and their proximity to the InstantEye drone with a high degree of certainty.

Thus, the goal of this project is four-fold: to develop a means of reading and parsing the module's serial data output, to acquire sample data and characterize the sensors' performance, to design a Kalman filter to fuse the sonar and IR sensor readings, and to tune this filter using the previously determined characteristics. Each step in this process is outlined and further explained in the following sections.

1.1 Kalman Filter Overview

The Kalman filter, primarily developed by Rudolph Kalman, is one of the most popular state estimation techniques due to its flexibility, robustness, and uncanny ability to resolve useful information from noisy signals. In practice, it is a recursive formulation and system of equations that act on a system state space model to minimize the error covariance, thereby converging upon the actual state. It is an optimal solution to the observer design problem for a well-defined linear system with uncorrelated noise, but it also works surprisingly well for non-ideal systems. The mathematical foundation of the algorithm lies in Bayesian inference and the solution to a discrete-time matrix Riccati difference equation. We present the final solution and an intuitive understanding to the technique in this paper. For a rigorous derivation and proof, refer to R. Kalman's original paper.[1]

The Kalman filter algorithm is described in terms of the state space model of a system, which is commonly written as

$$\dot{x}(t) = Ax(t) + Bu(t) \quad (1)$$
$$y(t) = Cx(t) + Du(t). \quad (2)$$

[1] "A new approach to linear filtering and prediction problem", Journal of Basic Engineering, ASME, 82 (March 1960).

Here, $x(t)$ is the state vector containing the system state variables, and $u(t)$ is the input vector corresponding to the control variables. $y(t)$ is the output vector containing the observable variables of interest. $A, B, C,$ and D are coefficient matrices that relate the state, input, and output vectors. The time-evolution of the system state is defined by Eq. 1 as a linear combination of the current state and the current input, and the system output is defined by a different linear combination. Any linear, time-invariant, and finite-dimensional differential or difference equation can be written in this matrix form.

The state equations used by the Kalman filter are given in discrete time and explicitly incorporate signal noise. They are

$$x_k = Ax_{k-1} + Bu_k + w_{k-1} \quad (3)$$
$$z_k = Hx_k + v_k, \quad (4)$$

where the index, k, refers to the k'th time step. The current state vector (x_k) is a function of the previous state (x_{k-1}), the current input (u_k), and the noise associated with process inaccuracies (w_{k-1}). A measurement of the state (z_k), is modeled as a function of the current state and the noise associated with the measurement (v_k). Again, $A.B$ and H are predefined coefficient matrices. If the noise is assumed to be Gaussian, then w and v have covariance matrices of Q and R respectively. That is, the probability densities are given by

$$p(w) \sim N(0, Q) \quad (5)$$
$$p(v) \sim N(0, R). \quad (6)$$

Before proceeding to the Kalman filter formulation, it is important to understand the approach from a holistic standpoint as well as the notational conventions. The filtering process can be divided into two recurring stages: time update or prediction, and measurement update or correction. The time update portion uses knowledge about the system model from Eq. 3 to predict the current state from the previous state. This gives the *a priori* state estimate. Then, we utilize information from the sensor measurements to correct this prediction, giving the *a posteriori* estimate. The amount by which the prediction is corrected by the measurement information depends on the relative accuracy in the state model versus that of the sensor reading. This is largely determined by the covariance values of Q and R from Eq. 5 and Eq. 6. To keep track of the relative error, we compute an error covariance matrix (P_k) that is updated with each *a priori* and *a posteriori* estimate. Note that error is defined as the difference between the actual state and the state estimate, and the error covariance is the expectation value of this error squared; see Eq. 7 and 8. The Kalman filter actively works to minimize this error covariance by weighting the measurement update correction more heavily in favor of the estimate source with less error. Since it is a recursive algorithm, the filter only stores information about the previous state and its error covariance.

$$e_k = x_k - \hat{x}_k \quad (7)$$
$$P_k = E[|e_k|^2]. \quad (8)$$

In the following mathematical description, we use an accent circumflex to indicate that the state predictions are just an estimate of the actual state, though over time the estimate will converge to the actual state. A superscript minus indicates the *a priori* prediction estimate, otherwise a variable is of the *a posteriori* estimate. Subscript k indicates the matrix or vector corresponding to the current state, and $k-1$ means the previous state.

We begin with a discussion of the time update equations.

$$\hat{x}_k^- = A\hat{x}_{k-1} + Bu_k \quad (9)$$
$$P_k^- = AP_{k-1}A^T + Q. \quad (10)$$

Eq. 9 predicts the *a priori* state by using the previous state, current input, and their relationship matrices defined by the system model from Eq. 3. We also compute the *a priori* error covariance matrix, P_k^-, from the previous *a posteriori* error covariance, the state transition matrix, A, and the process noise covariance, Q. Obviously, initial values need to be set for the first iteration.

Once we have a prediction, we compare it to the measured state, z_k, to form an updated *a posteriori* state estimate. Specifically, this entails computing the Kalman gain, K_k, a factor between 0 and 1 which is derived from the aforementioned Ricatti equation. In essence, it indicates the relative certainty in the measurement versus the estimate. A value close to 1 means that the measurements are more accurate, and a value close to 0 means that the state prediction is more accurate. The Kalman gain is then used to calculate the *a posteriori* state and error covariance as shown by Eq. 11-13.

$$K_k = P_k^- H^T (HP_k^- H^T + R)^{-1} \quad (11)$$
$$\hat{x}_k = \hat{x}_k^- + K_k(z_k - H\hat{x}_k^-) \quad (12)$$
$$P_k = (I - K_k H)P_k^- \quad (13)$$

Then, the process repeats for a new time step. $\hat{x}_k$ and P_k become $\hat{x}_{k-1}$ and P_{k-1} respectively and a new *a priori* estimate is calculated. The process is appropriately summarized by Fig. 1.

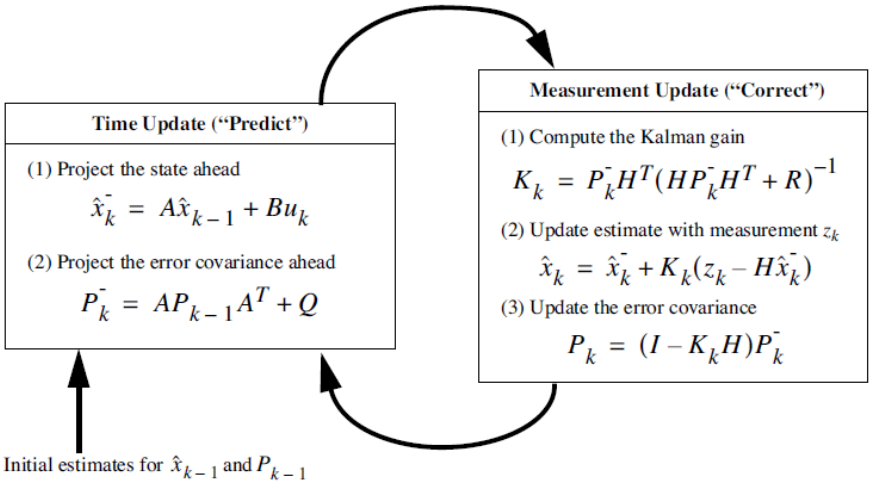

Figure 1: Kalman filter recursion diagram with significant equations[2]

2 Sensor Characterization

A primary component of this project is the characterization of the TerraRanger Duo's sonar and ToF IR distance sensors. Sensor performance was evaluated by the following metrics: distance resolution, polling frequency, field of view, material effects, target velocity, and valid range.

[2] Diagram from Welch, G., & Bishop, G. "An introduction to the Kalman filter", UNC Chapel Hill, (1995), p. 24.

2.1 USB Interface

Before measurements could be made towards characterizing the sensor, we required a means of interfacing with the sensor and parsing the output data into something useable. Some Python scripts were written to accomplish this. See Appendix A for the full code; sensor_read.py provides a user interface for calling functions from ie_sense.py which contains the tools for opening a serial connection with the sensor, parsing the data, and either displaying the data to the user or saving it to a file for post-processing.

As mentioned previously, the TerraRanger Duo comes with a built-in USB interface. The serial port settings are given in Table 1 and can be opened with the PySerial library.[3] For windows, the serial port is usually designated by a COM number, i.e. 'COM3'. For Linux, the serial port is often '/dev/ttySX' or '/dev/ttyUSBX' where 'X' is the corresponding port number. Additional connection settings are given by Table 1. Errors in the connection are handled by the 'SerialException' flag.

Table 1: Serial configuration settings for TerraRanger Duo interface

Port	COMX or /dev/ttyUSBX
Baud Rate	115200
Parity	None
Stop Bits	2
Byte Size	8

The serial connection returns the distance measurement data as a continuous stream of seven-byte messages. The first byte is 'T' (0x54 hex), and it followed by two bytes representing the binary distance value in millimeters as measured by the ToF sensor. This is followed by a byte 'S' (0x53 hex), and another two bytes representing the binary distance value in millimeters as measured by the sonar sensor. Finally, the message is terminated with a CRC8 checksum byte of the previous six bytes. Thus, the data output has the byte-wise form: 'TXXSXXC'.

Since the measurement data is being returned continuously, it is critical that the messages be aligned. To accomplish this, we simply wait until a 'T' is detected in the data stream. Then after two bytes pass, we check that the next byte is an 'S' followed by another two bytes and the correct checksum. Sometimes the distance measurement happens to also read 0x54 and 0x53 which throws off the initial alignment. This can be solved by checking the alignment continuously and reinitiating the process whenever an error occurs.

The user has the option to display a live feed of the sensor data or to save the data to a file for post-processing.

2.2 Sensor Characteristics

With the ability to read and interpret data from the sensor module, we begin the sensor characterization process. Again, we hope to evaluate the sensor performance by the following metrics: distance resolution, polling frequency, field of view, material effects, target velocity, and valid range. The numerical results are displayed in Table 2. This is followed with a discussion of each metric and an explanation of the data used to determine the parameters. Each parameter is presented in the table as its experimentally determined value to the left with the corresponding value from the datasheet to the right if available.[4]

[3] https://pythonhosted.org/pyserial/

[4] http://www.teraranger.com/wp-content/uploads/2015/04/TeraRangerDuospecificationsheet.pdf

Table 2: Sensor characteristic summary—format: "experimental value" / "data sheet value" (if available)

Distance Resolution (mm)	***Infrared ToF***	***Sonar***
Min	5 | 5	10 | 10
Polling Frequency (Hz)	***Infrared ToF***	***Sonar***
Min	19	0.70 | 1
Max	1000 | 600	1.05 | 20
Field of View (deg)	***Infrared ToF***	***Sonar***
Mean	5 | 3.4	30
Range (m)	***Infrared ToF***	***Sonar***
Min	0.20	0.20
Max	14.0 | 14.0	7.65 | 7.65

Distance Resolution

The distance resolution is the minimum incremental distance that the sensor will detect. This parameter is set by the analog-to-digital converter internal to the TerraRanger Duo. The IR sensor detects and reports changes in 5 mm steps while the sonar has a slightly poorer resolution of 10mm.

Polling Frequency

The polling frequency is the rate at which each sensor's reported measurements are updated with new information. It should be noted that the TerraRanger Duo supports both a precision and a speed mode. All measurements were made under the precision mode of operation. Data was acquired and timestamped over several one-minute intervals in which the sensor distance readings were continuously changing. The minimum and maximum polling rates were calculated by analyzing the longest and shortest times between reported changes in distance.

The wide variation in update frequencies can be attributed to a few main factors. First, and perhaps most significantly, is the delay caused by buffers in the serial port. Reading from and writing to the serial port is asynchronous and due to the nature of the data link, there is likely burstiness associated with the message transmission. Additionally, the data is timestamped upon arrival by the ie_sense.py script, not by the sensor itself. The timing method uses Python's time.time() function which is limited to approximately 1ms precision on Windows. Indeed, the precision varies with computer and operating system. Together, these effects compound to make measurement timing inaccurate, thus affect the polling frequency consistency.

Field of View

The field of view is the solid angle in which the sensor can detect valid measurements. Without specialized equipment, however, it is difficult to accurately characterize. For the IR sensor, placed a 1m square white target at 5m and 10m away and rotated the sensor until it failed to register the correct target distance. Doubling this angle gives the corresponding field of view solid angle. The sonar sensor works on a different principle and is less dependent of direction. It will return the distance of any object in a wide area in front of it. For this experiment, we used a narrower 1m by 0.2m target and placed in at varying angles away from dead-center and noted the locations at which the sonar failed to register the correct distance.

The IR sensor gives an accurate measurement of distance for objects immediately in its line of sight, while the sonar scans a wide flame-shaped area in front of it but returns their distances with lesser resolution. An example of the flame shape characteristic field of view of a similar sonar sensor is given by Figure 2.

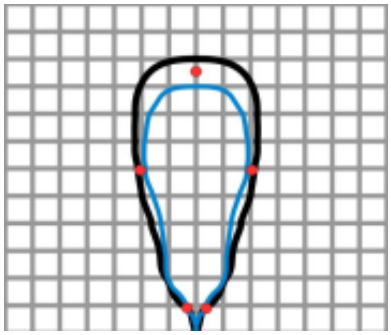

Figure 2: Flame-shaped sonar detection range—blue = full detection, black = partial detection[5]

Range

The range specifies the minimum and maximum valid distances that each sensor reports. Neither sensor reported values less than 20cm. For the IR sensor, once its maximum range of 14m was surpassed, the measurement would wrap back to 0. This is likely an aliasing artifact caused by the Time-of-Flight pulse rate. When the max range of the sonar was surpassed, the sensor reading locked at 7.65m.

Data samples from the sensor were captured in 1-minute intervals at constant distances increasing by 1-meter increments. An example histogram of the sensor measurements at 2 meters is given by Figure 3. The sonar tends to undershoot the distance, and the IR has a tendency to overshoot slightly.

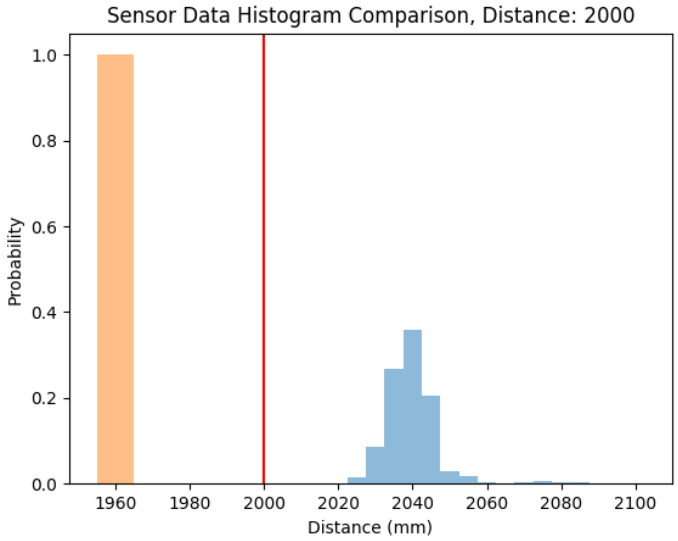

Figure 3: Sensor measurement samples at 2 meters as a roughly Gaussian probability distribution
Red = exact, Orange = sonar, Blue = IR

[5] See HRLV-MAXSonar-EZ Data Sheet, http://www.maxbotix.com/documents/HRLV-MaxSonar-EZ_Datasheet.pdf

Gaussian Measurement Approximation

Referring to Figure 3, the sensor measurement distribution is approximately Gaussian in nature due to the central limit theorem. We will use this observation in the design of the Kalman filter to fuse the two sensor signals. More specifically, we require the covariance matrix of the two measurements. This can be computed experimentally by Eq. 14 from the data collected at each of the 1-meter intervals. Here, X and Y are n-dimensional vectors containing the sample points, and $\bar{x}$ and $\bar{y}$ are the mean values of X and Y.

$$cov(X,Y) = \frac{1}{n-1}\sum_{i=1}^{n}(X_i - \bar{x})\,(Y_i - \bar{y}) \tag{14}$$

Target Velocity

In addition to the parameters given in Table 2, the qualitative effect that a moving target has on the accuracy of the distance measurement was evaluated. After several trials of tracking a car traveling between 2 and 15 miles per hour, it was determined that the velocity of the target has no discernable effect on accuracy. An object moving at higher speeds would likely leave the valid range of the sensors before any noticeable error could be noticed.

Material

Highly reflective or transparent surfaces cause the infrared sensor to yield sporadic invalid measurements. Similarly, the sonar has difficulty with anechoic geometries. The results appear randomly distributed and invalid. Figure 4 shows the effect of placing a translucent object directly in front of the TerraRanger Duo. The sonar accurately detects the object, but the IR data is scattered.

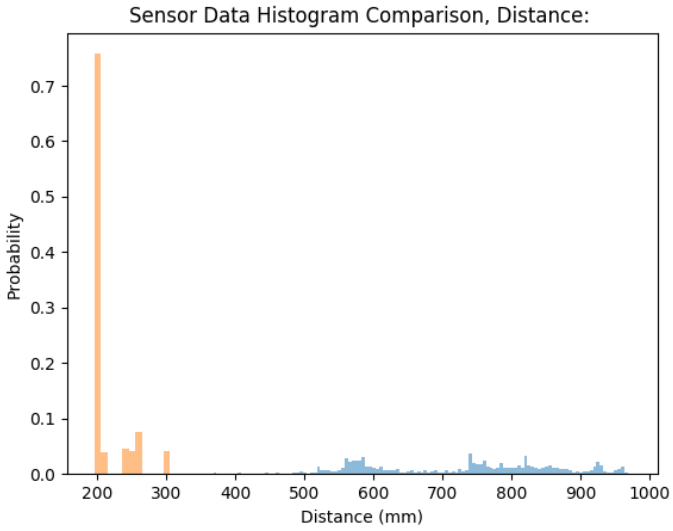

Figure 4: Effect of placing a transparent object directly in front of sensor
Orange = sonar, Blue = IR

3 Signal Fusion

Armed with knowledge of the sensor's tendencies and characteristics, we attempt to fuse the distinct sonar and IR measurements into a more reliable signal with a Kalman filter. Not only is the Kalman filtering

algorithm computationally efficient, but it also lends itself nicely to calculating the velocity of an object tracked by the distance sensors. Additionally, we can incorporate the known location of the InstantEye drone into the state model and calculate the exact location of an object simply by adding in the filter's relative position information.

We then expand on this filter in the Filter Tuning section by implementing conditional corrections aimed at compensating for the effects we just characterized. As an example, when measuring outside the valid range, the filter should ignore all measurement information.

3.1 Kalman Filter Design

First, we set the state vector of our state space model to track the relative distance (s) and velocity ($\dot{s}$) of the object to the sensor. That is,

$$x = \begin{bmatrix} s \\ \dot{s} \end{bmatrix}. \tag{15}$$

From the laws of motion, we can relate the time evolution of this state with itself via the state transition matrix, A. Since we do not have control over the object we observe with the sensors, the control matrix, B, is zero.

$$A = \begin{bmatrix} 1 & dt \\ 0 & 1 \end{bmatrix} \tag{16}$$

$$B = 0. \tag{17}$$

Both the ToF and the sonar sensors directly measure an object's relative distance, thus the measurement matrix, H, is given by

$$H = \begin{bmatrix} 1 & 0 \\ 1 & 0 \end{bmatrix}. \tag{18}$$

To initialize the filter for the first iteration, we set the state distance to the first IR sensor position measurement, and the velocity to zero. The specific values are not important in the long-run, as the filter will converge to the correct values, but by setting the initial state to the first measurement value—which we assume to be decently accurate, we ensure this convergence occurs almost immediately. Likewise, the initial value of the error covariance matrix is not critical, with the one caveat that it cannot be a zero matrix. If it is zero, the filter assumes there is no error, and will never change its prediction. For simplicity, we assume the error covariance matrix to be the identity matrix at the start.

$$x_0 = \begin{bmatrix} z_0(1) \\ 0 \end{bmatrix} \tag{19}$$

$$P_0 = I. \tag{20}$$

The final pieces of information we need to implement the Kalman filter are the process and measurement noise covariance matrices, Q and R from Eq. 5 and 6. R is experimentally calculated by Eq. 14. The measurement covariance matrix result at 1m is shown by Eq. 21. The first element represents the self-covariance of the IR sensor, the last element is of the sonar, and the off-diagonals are the cross covariance of the two sensor measurements. For the process covariance, precise calculation requires careful calibration and analysis of the entire system model. Due to lack of resources and time constraints, we simply assume that there is no correlation between the calculation of the position and the velocity by setting the

off-diagonal terms to zero. The diagonal elements were experimentally determined by running the filter with different values until a reasonable response was obtained; see Eq. 22.

$$R_{1m} = \begin{bmatrix} 1.237e^{-4} & 4.747e^{-5} \\ 4.747e^{-5} & 5.710e^{-5} \end{bmatrix} \tag{21}$$

$$Q = \begin{bmatrix} 0.01 & 0 \\ 0 & 0.01 \end{bmatrix}. \tag{22}$$

We now present the core code that runs the Kalman filter algorithm; the method is written in Python and utilizes functions from the numpy library. This script is called each time there is an updated measurement, and runs through the computations shown in Figure 1 to compute a new output state. For the full implementation, see Appendix B.

```
# Perform Next Filter Iteration
def Step(self, z):
     # A Priori
     x_prior = self.A.dot(self.X)
     p_prior = (self.A.dot(self.P)).dot(self.A.T) + self.Q

     # A Posteriori
     residual = (self.H.dot(p_prior)).dot(self.H.T) + self.R
     k_gain = (p_prior.dot(self.H.T)).dot(np.linalg.inv(residual))
     x_post = x_prior + k_gain.dot(z - self.H.dot(x_prior))
     size = self.P.shape[0]
     p_post = (np.identity(size) - k_gain.dot(self.H)).dot(p_prior)

     # Update
     self.X = x_post
     self.P = p_post
```

A plot of the filter output versus time is given by Figure 5, below. The capture is of arbitrary measurements made by pointing the sensor at random objects varying in size, shape, and distance. The top graph is of distance, and the bottom is of velocity.

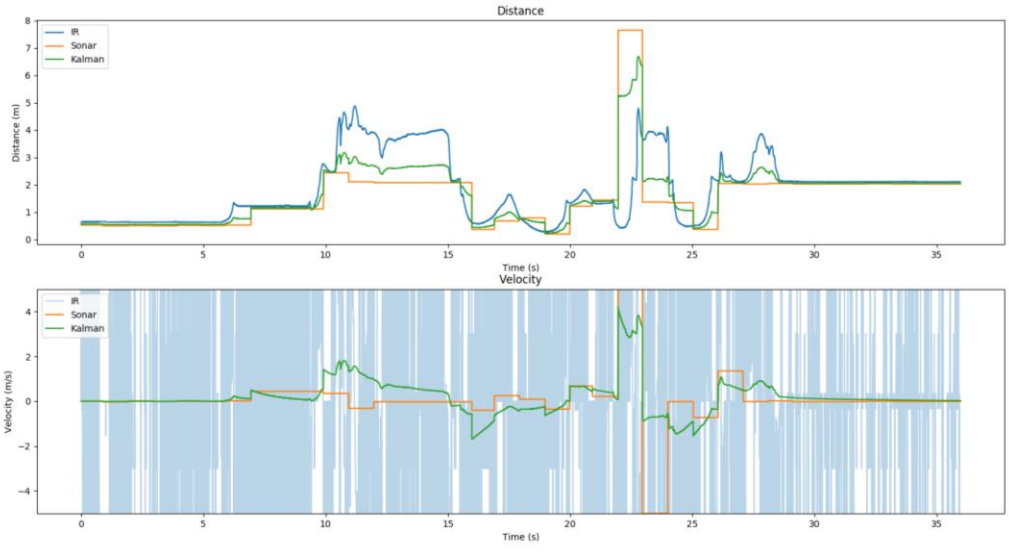

Figure 5: Kalman filter arbitrary distance and velocity state comparison with sensor inputs
Green = Kalman, Blue = IR, Orange = sonar

From the distance plot, the Kalman filter output appears to directly follow the distance measurements. When there is a discrepancy between the sonar measurement and the IR measurement, the filter finds a weighted average of the two values. One of the primary sources of this error arises from the low polling rate of the sonar as compared to the IR sensor. Also, the sonar has a wider field of view and may be detecting different objects that are not registered by the IR sensor.

From the velocity plot, we can see that calculating velocity from the raw IR sensor data yields very noisy results. Taking a derivative of the rapidly changing signal amplifies the high frequency component. Interestingly, the Kalman filter pulls out a somewhat usable velocity signal by incorporating the signal from the sonar, which by itself is too coarse to be useful.

The trend is clear, and the Kalman filter appears to fuse the two signals reliably, however there is definite room for improvement, and we attempt to optimize the filter with conditional gates in the following section.

3.2 Filter Tuning

In the previously described version of the Kalman filter, the weighting of each sensor input is determined by the Kalman gain which is set in part by the previously calculated measurement covariance. Thus, by altering these covariance matrices when certain conditions are met, we can modify the Kalman gain to be more or less receptive to signals from either of the two sensors. Overall, three conditions were tested: distance-dependent measurement covariance matrices, disabling measurements outside the valid range, and decreasing the sonar wait until next update. See Apenndix B for the full code implementation.

Distance-Dependent Measurement Covariance

From the data gathered during the range characterization of the sensor, we computed covariance matrices corresponding to various distances through the entire valid range of sensor operation. By checking the *a priori* state for an approximate range, we select the appropriate covariance matrix to apply in the Kalman gain calculation. In total, there are 14 distinct experimental covariance matrices spaced in 1-meter intervals. For example, if the *a priori* estimate indicates a predicted distance of 5.2-meters, the covariance matrix corresponding to the 5-meter range would be used in calculating the gain for the *a posteriori* state. This condition is described mathematically by Eq. 23, where the measurement covariance becomes a conditional function of the *a priori* state.

$$R \rightarrow R_i | \hat{x}_k^-, \qquad i \in \{1,2, \dots 14\} \tag{23}$$

In practice, the effect this condition had on the filter response was negligible as the sensor covariance matrices did not vary significantly with distance as anticipated.

Disable Measurements Outside Valid Range

The sonar sensor has a maximum range of 7.65 meters, and the IR sensor has a maximum range of 14 meters. When the reported values surpass this valid range, the sensor data is invalid and should be ignored. The implementation of this condition was simple for the sonar as the returned values cap at 7.65m. Any time the sonar reports 7.65, the measurement is ignored by setting the variance associated with the sonar ($R_{2,2}$) to a large value (100). We were unable to determine an elegant solution for ignoring the IR sensor data; see section on Future Work.

Decrease Sonar Weight Until Next Update

The sonar has a much slower polling rate than the IR sensor. For fast moving objects, this means that the position information of the sonar quickly becomes invalid until the next measurement update is received. Thus, we weight antiquated sonar measurements less by scaling its variance value by a factor proportional to the velocity state. The scaling factor was chosen somewhat arbitrarily, but it proves effective based on empirical results. The factor is given by

$$R_{new} = R * (1 + vel)^2. \tag{24}$$

Incorporating these three conditions into the filter yields significantly improved results, as shown in Figure 6. The new filtered signal uses the frequent updates of the IR sensor to give a more realistic interpretation of the distance at any point in time, and it realigns itself with the sonar when an update is available. Not shown in Figure 6—as it is difficult to reproduce errors—invalid measurements (7.65m) from the sonar sensor are properly ignored. Finally, the velocity signal has greater detail and is not bounded to the low time-resolution sonar signal as it was before. However, some artifacts from the distance state appear to creep into the velocity state, especially noticeable around the 10-20 second mark.

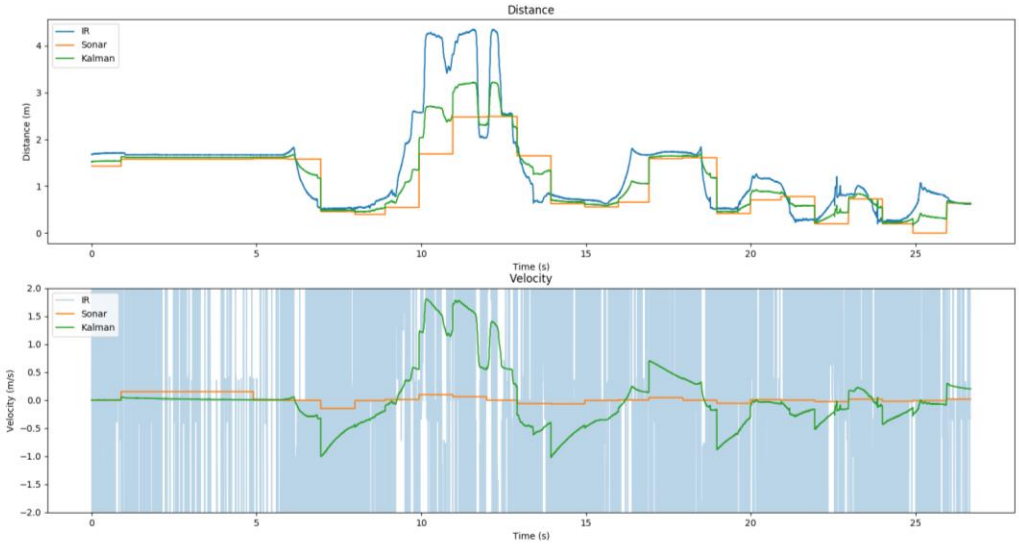

Figure 6: Modified Kalman filter arbitrary distance and velocity state comparison with sensor inputs Green = Kalman, Blue = IR, Orange = sonar

4 Future Work

The logical next step in the development of this collision avoidance system for InstantEye is to test the payload on the drone itself and to combine the state information from the drone with the relative state information of objects tracked by the sensor. Further improvements can also be made with regards to conditional tuning of the filter. Specifically, it would beneficial to have a robust system for detecting when the IR sensor is returning a value outside its valid range. Additionally, when the IR sensor and sonar signals diverge, as between 10-20 seconds in Figure 6, they are likely tracking two different objects. In this case, it would be beneficial for the filter to return information about both objects, and not just a weighted average of the two. That is, given two distinct signals from the IR and sonar sensors, the filter should also return two distinct signals.

Acknowledgements

I am deeply grateful to Dr. Marc Lessard, the faculty advisor to this project and the person whom I must thank for bringing the engineering physics program to fruition at UNH. I would also like extend gratitude to Marc Merritt and James Glynn of Physical Science, Inc. This project would not have been possible without their support.

References

[1] Einicke, G. A. (2012). Smoothing, filtering and prediction: estimating the past, present and future. InTech.

[2] Faragher, Ramsey (2012). Understanding the basis of the kalman filter via a simple and intuitive derivation [lecture notes]. IEEE Signal processing magazine, 29, 128-132.

[3] Kalman, R. E. (1960). A new approach to linear filtering and prediction problems. Journal of basic Engineering, 82(1), 35-45.

[4] Labbe, R. R. (2015). Kalman and bayesian filters in python.

[5] Nise, N. S. (2007). Control systems engineering. John Wiley & Sons.

[6] Terejanu, G. (2008). Crib sheet: Linear Kalman smoothing.

[7] Welch, G., & Bishop, G. (1995). An introduction to the Kalman filter.

Appendix A: Serial Interface Code

A.1 sensor_read.py

```
# Wyman Smith
# April 2, 2017
#
# InstantEye: TerraRanger Duo Serial Interface
#
# Serves as the serial interface menu for the TerraRanger Duo.

import ie_sense, sys

# Connect to sensor
ser = ie_sense.connect()

# Read and manipulate sensor data
if __name__ == '__main__':
     read_type = raw_input("(S)tream or save to (F)ile\n").lower()

     # Display live sensor data
     if read_type in ['s', 'stream']:
          ie_sense.readc(ser)

     # Save sensor data to file
     elif read_type in ['f','file']:
          filename = raw_input("Enter the desired file name\n")
          ie_sense.save_data(ser,filename)

     else:
          print("Command not recognized.")
          sys.exit()
```

A.2 ie_sense.py

```
# Wyman Smith
# April 2, 2017
#
# InstantEye: TerraRanger Duo Serial Interface
#
# Contains functions for reading and manipulating the sensor connection and data.

import serial, sys, time, os
from threading import Thread
from serial import SerialException

# Current time for time-stamping data (ms)
current_time = lambda: int(round(time.time()*1000))

# ======================
# Serial connection setup
# ======================
def connect(COM = None):
     # If none specified, default serial port is COM3
     if COM is None:
          COM = 'COM3'

     # Attempt serial connection
     while True:
          try:
               ser = serial.Serial(
                    port=COM,
                    baudrate=115200,
```

```
                    parity=serial.PARITY_NONE,
                    stopbits=serial.STOPBITS_TWO,
                    bytesize=serial.EIGHTBITS
               )

          except SerialException:
               print("Connection failed. Make sure the device is on, or try changing the serial port.\n")
               raw_input("Press Enter to continue...")
               continue

          print('Connected to: ' + ser.portstr)
          break

     # (B)inary output and (P)recision mode
     ser.write("B\r\n")
     ser.write("P\r\n")
     time.sleep(.1)

     ser.close()
     return ser

# ============================
# Continuously read sensor data
# ============================
def readc(ser):
     ser.open()

     # Align output
     while True:
          if ser.read() == 'T':
               ser.read(2)
               if ser.read() == 'S':
                    ser.read(3)
                    break

     # Stop on user input
     stop_cond = {'status': True}
     Thread(target=stop_capture, args=(stop_cond,)).start()

     # Display output
     while stop_cond['status']:
          data = ser.read(7)
          T = int(''.join( [bin(ord(data[1]))[2:].zfill(8), bin(ord(data[2]))[2:].zfill(8)] ), 2)
          S = int(''.join( [bin(ord(data[4]))[2:].zfill(8), bin(ord(data[5]))[2:].zfill(8)] ), 2)

          CRC = ord(data[6])

          sys.stdout.flush()
          sys.stdout.write("\rT (cm): {0:05d} | S (cm): {1:05d}".format(T/10, S/10))

# ============================
# Save sensor data to text file
# ============================
def save_data(ser,filename):
     # Save in data directory
     cwd = os.getcwd()
     directory = os.path.dirname(cwd + '/data/')
     if not os.path.exists(directory):
          os.makedirs(directory)
          os.chdir(directory)
     else:
          os.chdir(directory)

     # Create file
     file = open(filename+'.txt','w+')

     time.sleep(1)

     # Stop recording and save on user input
```

```
    stop_cond = {'status': True}
    Thread(target=stop_capture, args=(stop_cond,)).start()

    ser.open()

    # Align output
    while True:
        if ser.read() == 'T':
            ser.read(2)
            if ser.read() == 'S':
                ser.read(3)
                break

    # Save data
    while stop_cond['status']:
        data = ser.read(7)
        T = int(''.join( [bin(ord(data[1]))[2:].zfill(8), bin(ord(data[2]))[2:].zfill(8)] ), 2)
        S = int(''.join( [bin(ord(data[4]))[2:].zfill(8), bin(ord(data[5]))[2:].zfill(8)] ), 2)

        # File format: 'IR_DATA(mm) SONAR_DATA(mm) TIME_STAMP(ms)\n'
        data = str(T) + ' ' + str(S) + ' ' + '{}'.format(current_time())

        # Display sensor reading live to user
        sys.stdout.flush()
        sys.stdout.write("\rT (cm): {0:05d} | S (cm): {1:05d}".format(T/10, S/10))

        file.write(data + '\n')

    # Return to original directory
    file.close()
    os.chdir(cwd)

# ========================
# Stop data capture thread
# ========================
def stop_capture(stop_cond):
    raw_input("Press any Enter to stop recording...\n")   # Stop on user input
    #time.sleep(30)         # 30 second capture
    stop_cond['status'] = False
```

Appendix B: Kalman Filter Code

```
# Wyman Smith
# April 15, 2017
#
# InstantEye: TerraRanger Duo Sensor Fusion Kalman Filter v1
#
# Functions for Kalman filter sensor fusion with conditional and time-varying corrections.

from __future__ import division
import numpy as np
import matplotlib.pyplot as plt
import threading, time, serial, Queue, sys # Remove sys later
from serial import SerialException

# Current time for time-stamping data (ms)
current_time = lambda: int(round(time.time()*1000))

# =============
# Kalman Filter
# =============
class KalmanV1:
    def __init__(self, X):
        dt = 1/600                                      # Time step
        self.X = X                                      # Initial State
        self.P = np.array([[1,0],[0,1]])       # Initial Error Covariance
        self.A = np.array([[1,dt],[0,1]])      # State Evolution
```

```
            self.H = np.array([[1,0],[1,0]])       # Observation Matrix
            self.Q = np.array([[0.01,0],[0,0.01]])      # Process Covariance
            self.R = np.array([[0.0002,0.0001],[0.0001,0.00015]])      # Measurement Covariance

            # Corrections: class init
            self.kfc = Kf_Corrections()

        # Return Current State
        def State(self):
            return self.X

        # Perform Next Filter Iteration
        def Step(self, z):
            # A Priori
            x_prior = self.A.dot(self.X)
            p_prior = (self.A.dot(self.P)).dot(self.A.T) + self.Q

            # Corrections: conditional covariance
            # self.R = self.kfc.conditional_cov(x_prior)
            self.R = np.array([[0.0002,0.0001],[0.0001,0.00015]])
            # Corrections:  decrease sonar weight with time until next update
            self.R = self.kfc.sonar_weight(self.R,z,x_prior)
            # Corrections: outside valid range
            self.R = self.kfc.disable_invalid(self.R,z)

            # A Posteriori
            residual = (self.H.dot(p_prior)).dot(self.H.T) + self.R
            k_gain = (p_prior.dot(self.H.T)).dot(np.linalg.inv(residual))
            x_post = x_prior + k_gain.dot(z - self.H.dot(x_prior))
            size = self.P.shape[0]
            p_post = (np.identity(size) - k_gain.dot(self.H)).dot(p_prior)

            # Update
            self.X = x_post
            self.P = p_post

# ========================
# Kalman Filter Corrections
# ========================
class Kf_Corrections:
    def __init__(self):
        # Load covariance matrix database
        # cwd = os.getcwd()
        # self.cov = np.load(cwd+'\covariance_db.npy')                  # COME BACK AND CHANGE THIS TO CWD
        self.cov = np.load('D:\InstantEye\Distance Sensor\covariance_db.npy')
        self.prev_vel = 0

    # Set covariance based of a priori position
    def conditional_cov(self,x_prior):
        i = x_prior[0]

        if i < 0.75:   # 0.5m
            return self.cov[0]
        elif i < 1.5:  # 1m
            return self.cov[1]
        elif i < 2.5:  # 2m
            return self.cov[2]
        elif i < 3.5:  # 3m
            return self.cov[3]
        elif i < 4.5:  # 4m
            return self.cov[4]
        elif i < 5.5:  # 5m
            return self.cov[5]
        elif i < 6.5:  # 6m
            return self.cov[6]
        elif i < 7.5:  # 7m
            return self.cov[7]
        elif i < 8.5:  # 8m
            return self.cov[8]
```

```
            elif i < 9.5:  # 9m
                return self.cov[9]
            elif i < 10.5: # 10m
                return self.cov[10]
            elif i < 11.5: # 11m
                return self.cov[11]
            elif i < 12.5: # 12m
                return self.cov[12]
            elif i < 13.5: # 13m
                return self.cov[13]
            elif i < 14.5: # 14m
                return self.cov[14]
            elif i < 15.5: # 15m
                return self.cov[15]
            else:               # >15m
                return self.cov[16]

        # Disable measurements outside of valid range
        def disable_invalid(self,R,z):
            R = np.array(R)
            s = z[1]  # Sonar reading
            if s == 7.65:       # Max range
                R[1,1] *= 100  # Set variance of sonar to *100

            return R


        # Decrease sonar weight until next update
        def sonar_weight(self,R,z,x_prior):
            if self.prev_vel == z[1]:    # If no update from sonar
                R[1,1] *= (1+z[1])*(1+z[1])       # Increase covariance the larger the velocity

            else:    # Else set new previous velocity
                self.prev_vel = z[1]
            return R

# ================
# Sensor Interface
# ================
class Sensor:
    def __init__(self, COM = None):
        # If none specified, default serial port is COM3
        if COM is None:
            COM = 'COM3'

        # Attempt serial connection
        while True:
            try:
                self.ser = serial.Serial(
                    port=COM,
                    baudrate=115200,
                    parity=serial.PARITY_NONE,
                    stopbits=serial.STOPBITS_TWO,
                    bytesize=serial.EIGHTBITS
                )

            except SerialException:
                print("Connection failed. Make sure the device is on, or try changing the serial port.\n")
                raw_input("Press Enter to continue...")
                continue

            print('Connected to: ' + self.ser.portstr)
            break

        # (B)inary output and (P)recision mode
        self.ser.write("B\r\n")
        self.ser.write("P\r\n")
        time.sleep(.1)
```

```
            self.ser.close()

        # Read data live from sensor
        def read(self, q):
            self.ser.open()

            # Align output
            while True:
                if self.ser.read() == 'T':
                    self.ser.read(2)
                    if self.ser.read() == 'S':
                        self.ser.read(3)
                        break
            # Properly format output ('ctrl+c' to stop)
            while True:
                data = self.ser.read(7)
                # Distances in millimeters
                T = int(''.join( [bin(ord(data[1]))[2:].zfill(8), bin(ord(data[2]))[2:].zfill(8)] ), 2)
                S = int(''.join( [bin(ord(data[4]))[2:].zfill(8), bin(ord(data[5]))[2:].zfill(8)] ), 2)

                # Not currently being used, but serial error correction possible with this
                CRC = ord(data[6])
                # Format data array
                data = np.array([T, S, current_time()])
                # Add sensor data to output queue
                q.put(data)

# ========================
# Plot distance versus time
# ========================
def dvt_plot(data):
    y_t, y_s, y_k, x, v_k, v_t, v_s = [],[],[],[],[],[],[]

    # Start from t=0
    t_0 = data[0][2]
    dt2 = 0

    # Needed to make sonar velocity calc work
    v_s.append(0)

    for i in data:
        y_t.append(i[0])        # IR-ToF Distance (m)
        y_s.append(i[1])        # Sonar Distance (m)
        y_k.append(i[3])        # Filter Distance (m)
        x.append(i[2] - t_0)    # Time (ms)
        v_k.append(i[4])        # Filter Velocity (m/s)

        # Calculate Raw Velocities
        if len(x) == 1:
            continue
        dt = np.absolute(x[-2]-x[-1])
        if dt == 0:
            dt = 1/600
        v_t.append((y_t[-1]-y_t[-2])/dt)

        # Sonar has longer dt, only update v when new value present
        if y_s[-2] == y_s[-1]:
            dt2 += dt
            v_s.append(v_s[-1])
            continue

        v_s.append((y_s[-1]-y_s[-2])/dt2)


    # Makes velocity and time vectors the same size
    v_t.append(v_t[-1])

    # Plot IR and Sonar Distance vs. Time
    plt.figure(1)
```

```
     plt.subplot(211)
     plt.plot(x, y_t, label='IR')
     plt.plot(x, y_s, label='Sonar')
     plt.plot(x, y_k, label='Kalman')
     plt.legend(loc='upper left')
     plt.xlabel('Time (s)')
     plt.ylabel('Distance (m)')
     plt.title('Distance')

     # Plot Velocity vs. Time
     ax1 = plt.subplot(212)
     plt.plot(x, v_t, label='IR', alpha=0.3)
     plt.plot(x, v_s, label='Sonar')
     plt.plot(x, v_k, label='Kalman')
     ax1.set_ylim([-2,2])
     plt.legend(loc='upper left')
     plt.xlabel('Time (s)')
     plt.ylabel('Velocity (m/s)')
     plt.title('Velocity')
     plt.show()

# ========================
# Stop data capture thread
# ========================
def stop_capture(stop_cond):
     raw_input("Press any Enter to stop recording...\n")
     stop_cond['status'] = False

# Run Kalman filter
if __name__ == '__main__':
     # Connect to sensor
     sense = Sensor('COM3')

     # Initialize queue to output sensor data
     q = Queue.Queue()

     # Continually read sensor output in thread
     read_worker = threading.Thread(target=sense.read, args=(q,))
     read_worker.start()

     # Kalman filter initialization information
     data = q.get()/1000
     x_init = np.array([data[0],0])         # Initial state, X = Z

     # x = state, z = measurement
     kf = KalmanV1(x_init)

     stop_cond = {'status': True}
     threading.Thread(target=stop_capture, args=(stop_cond,)).start()

     datalist = []

     # Run filter
     while stop_cond['status']:
          data = q.get()
          timestamp = data[2]
          data = data[:2]/1000
          kf.Step(data)
          K = kf.State()
          T = data[0]
          S = data[1]
          # Display sensor reading live to user
          sys.stdout.flush()
          sys.stdout.write("\rT (m): {0:.3f} | S (m): {1:.3f} | K (m): {2:.3f} | V (m/s): {3:.3f}".format(T,
S, K[0], K[1]))

          datalist.append([T,S,timestamp/1000,K[0],K[1]])

     dvt_plot(datalist)
```