ECE532 Project

Space Object Detection and Interception

Final Report

Team 36

Muaz Shash

Damian Pacynko

Marko Ciric

Youssef Elhadad

April 7, 2025

Contents

1.0 Overview

1.1 Motivation

Meteors and space debris entering Earth's atmosphere pose significant risks to human life, property, and critical infrastructure. With the accelerated growth of space exploration activities, satellite deployments, and orbital missions, the volume of debris orbiting Earth is projected to increase dramatically, intensifying these risks. Therefore, accurately detecting, tracking, and mitigating threats from falling space objects has become a crucial priority to safeguard public safety and infrastructure integrity.

This report introduces a proof-of-concept for a detection and interception system designed specifically to identify, track, and neutralize space debris entering Earth's atmosphere. Our proposed system utilizes an array of ultrasonic sensors strategically positioned within a defined monitoring area to accurately localize incoming objects. Once detected, a targeted laser mechanism simulates the neutralization or "destruction" of the object by pointing at it. Ultrasonic sensors were chosen due to their affordability, ease of deployment, and scalability. Although ultrasonic sensors inherently introduce some measurement uncertainty, this level of accuracy is deemed sufficient for a proof-of-concept demonstration.

A critical component of our approach involves leveraging an FPGA rather than a traditional microcontroller, primarily due to the need for real-time processing of multiple ultrasonic sensor inputs simultaneously. The inherent parallel-processing capabilities of an FPGA significantly reduce system latency, substantially enhancing detection accuracy and responsiveness.

For context, consider that a traditional microcontroller would process sensor inputs sequentially in a round-robin manner. With each ultrasonic sensor having an approximate latency of 50 milliseconds, the total latency quickly becomes substantial ($n \times 50$ ms for n sensors), negatively impacting the system's ability to accurately track rapidly moving objects. As the number of sensors—and therefore accuracy—increases, latency escalates proportionally.

In contrast, utilizing an FPGA allows simultaneous parallel processing of data from all sensors within a single 50 ms frame, drastically reducing total latency regardless of sensor count. This parallel architecture enables our system to achieve higher sensitivity and responsiveness, thus significantly enhancing its capability to track and respond to objects dynamically as they move through Earth's atmosphere.

1.2 Project Goal

The primary goal of this project is to create a proof-of-concept system that efficiently detects, tracks, and mitigates threats from meteors and space debris in real-time. Specific objectives include:

- Implementing an ultrasonic sensor array to accurately detect incoming objects.
- Designing custom IP blocks for the Nexys Video FPGA board to process detection data swiftly and efficiently.
- Using a servo-controlled laser to precisely target and simulate the destruction of identified threats.
- Providing real-time visual feedback via HDMI output, displaying predicted impact locations.

1.3 IP Overview

Table 1: IP Descriptions

IP	**Description**
Parser	Custom RTL core that triggers and reads 12 ultrasonic sensors at once, and calculates their detected distances.
Localization	Custom RTL core that uses the sensor distances to determine the object's (x, y, z) location.
Servo Controller	Custom RTL core that points a pair of pan and tilt servos at the determined location.
Frame Generation	Modified, existing HDMI core for displaying object location and landing position.
MicroBlaze	Xilinx soft processor for interfacing with the Localization IP through AXI, and controlling what is displayed through the HDMI IP

1.4 Block Diagram

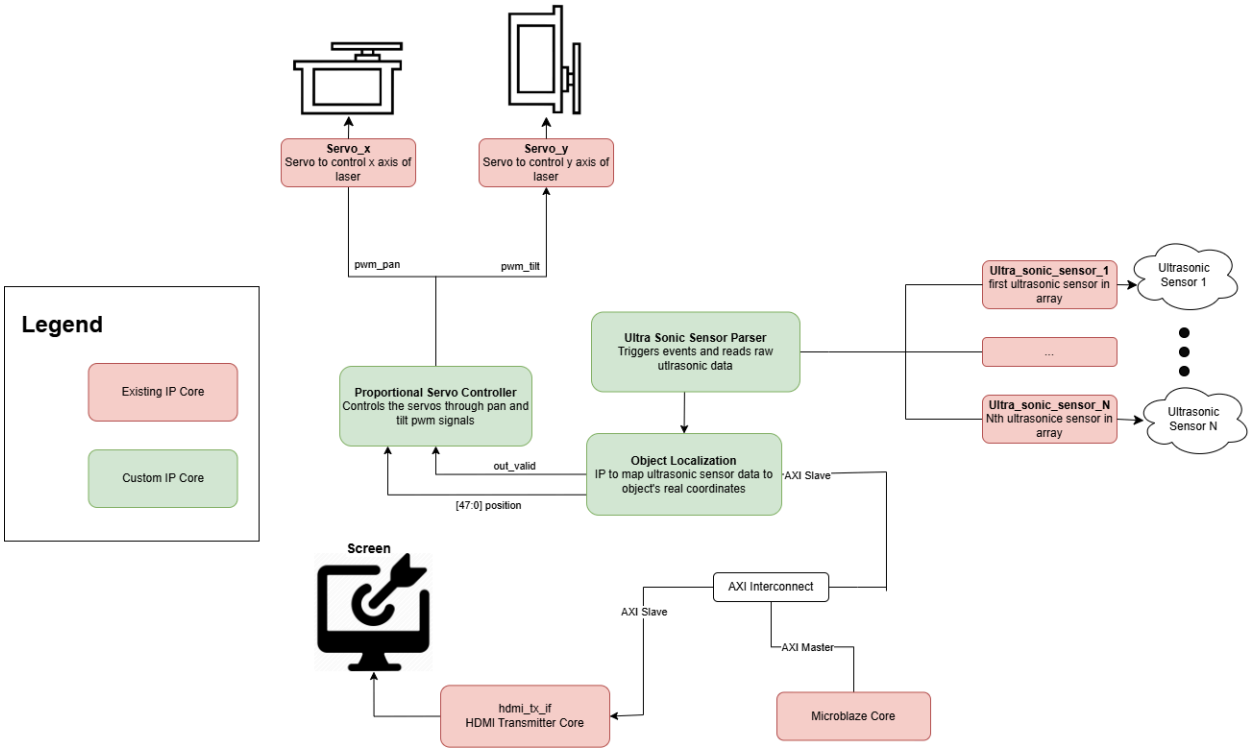


Figure 1: Final System Block Diagram

2.0 Outcome

2.1 Results

The section below provides a summary of the results achieved for each of the custom IP blocks.

2.1.1 Parser

The parser IP accurately captures sensor data with up to 6 mm accuracy for stationary objects within a two-meter range. A known source of error for accuracy is caused by our fixed-point representation, which introduces a ±2 mm deviation compared to double-precision operations. Despite this, the achieved accuracy is more than sufficient for our purposes, and we consider this a success. However, objects moving quickly in and out of frame can cause sporadic behavior.

This issue was also observed when using an Arduino Uno, indicating that it is a limitation of the sensor rather than the IP itself.

2.1.2 Localization

The localization IP went through significant rework to make better use of noisy and low-resolution sensor data (Figure 2). Techniques such as averaging the minimum three sensor values, restricting valid sensor inputs to a neighborhood, and finally applying time-averaging contributed to a tremendous improvement to the localizer IP's performance (Figure 3).

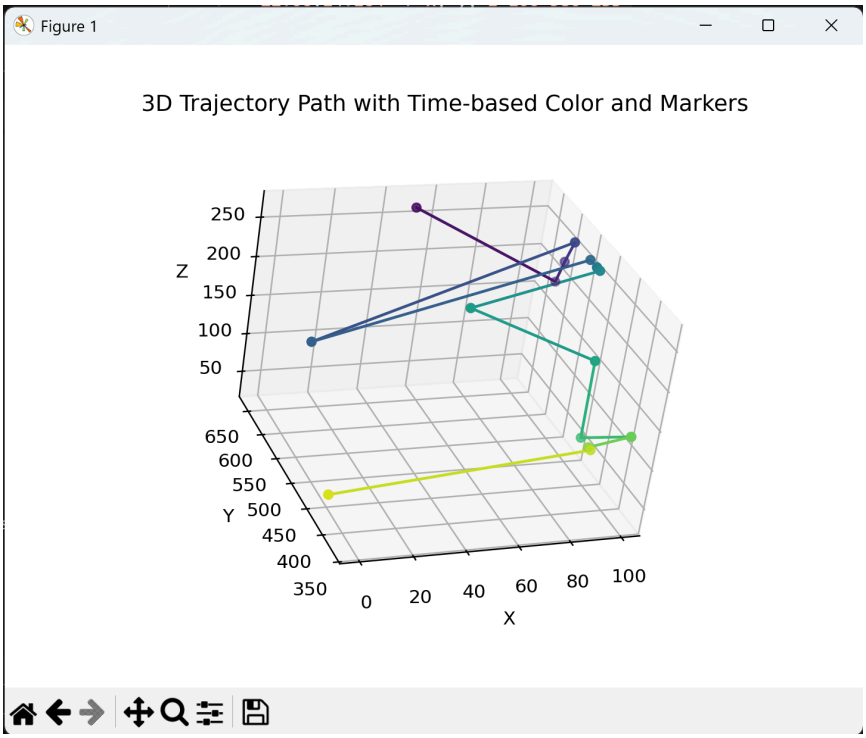


Figure 2: Trajectory path from localization outputs before optimizations

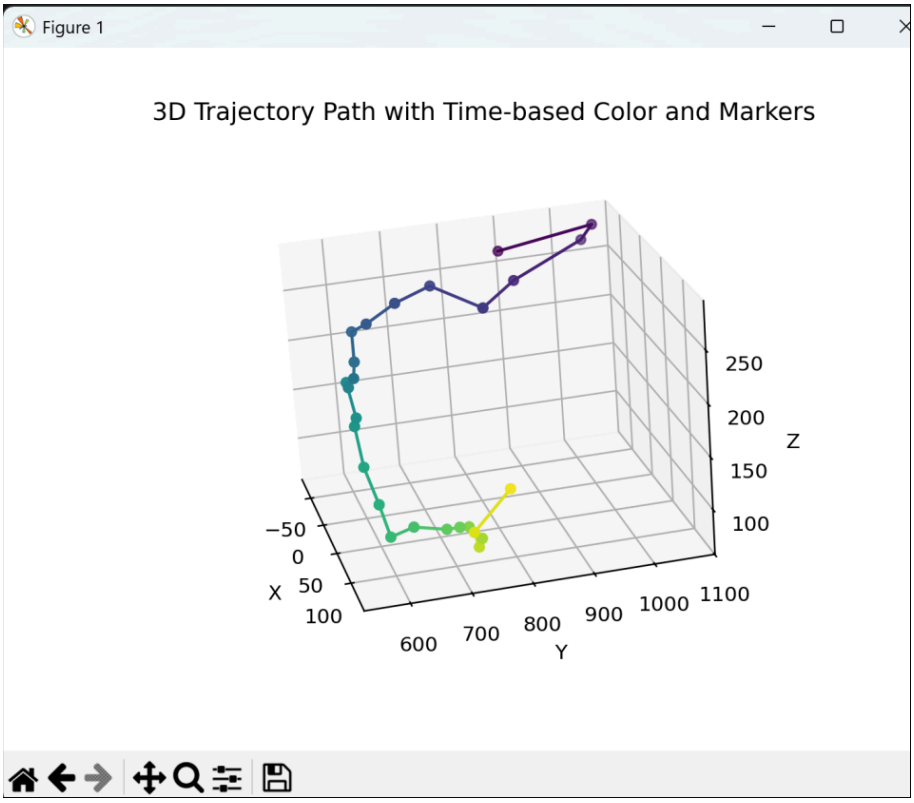


Figure 3: Trajectory path from localization outputs after optimizations

Validating the exact path of the balloon was challenging and not a priority at the time of testing, however, the general motion is in line with our expectations. For example, when the balloon was thrown diagonally from the left side to the right and toward the sensors, the data showed movement from -x to +x with a decreasing y-value—until it stopped accelerating and began to fall vertically (Figure 3).

We have tested the stationary accuracy of the IP, and we noticed that the localizer never settled at some x or z value. Instead, it would oscillate ± 50mm in either direction of the sensor locations. This can be attributed to different parts of the object being picked up by different sensors. With time averaging, however, this was no longer an issue as the oscillations would average between the two sensors.

Although we don't consider this IP a complete success (as we'll get into in section 2.1.5), we were still able to extract useful information from it to feed to HDMI and the servo controller IP.

2.1.3 Frame Generation

The original goal was for the trajectory path and landing position of the object to be displayed to a monitor in real-time using the MicroBlaze processor and existing Vivado IPs for HDMI. However, the complexity of the trajectory algorithm meant that the landing position could not be determined. So, the HDMI functionality was updated to support two different screen displays that can be toggled via interrupt with the board's buttons.

The first screen consisted of a dual grid, with the grid on the left displaying the x-z localized (real-world) coordinates of the object, while the grid on the right displayed the y-z localized coordinates of the object, with both displaying in real-time using a circle to denote the position, see Figure 4. However, the grid lines weren't accurately mapped to the dimensions, as is indicated by the z-axis text not aligning with the horizontal grid lines, meaning that the display was effectively only on the bottom half of the screen, resulting in a less meaningful GUI.

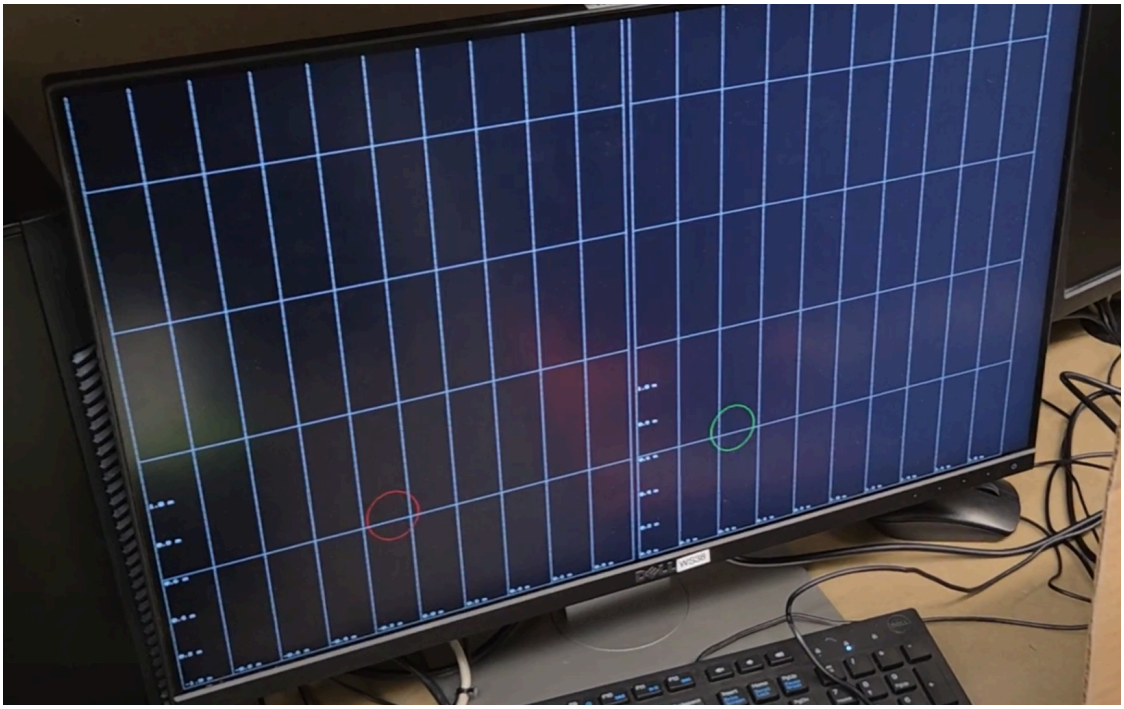


Figure 4: First screen x-z (left) and y-z (right) localized coordinates display.

The second screen consisted of a single grid for displaying the localized x-y coordinates of the object (see Figure 5). However there was a latency in the circle's position displaying because the static grid would be drawn each time the position was updated, this could be solved using the same double buffering technique as with the first screen which was original done in the testing, but the discrepancy of what should be displayed to the HDMI between the original goal and the contingency plan to show (x-z, y-z plots) and the time-constraints of the project led to this screen being inaccurate in displaying the object's real position. Also, like the first screen, the grid lines were not accurately mapped to the display, as the grid was only on the left side of the screen.

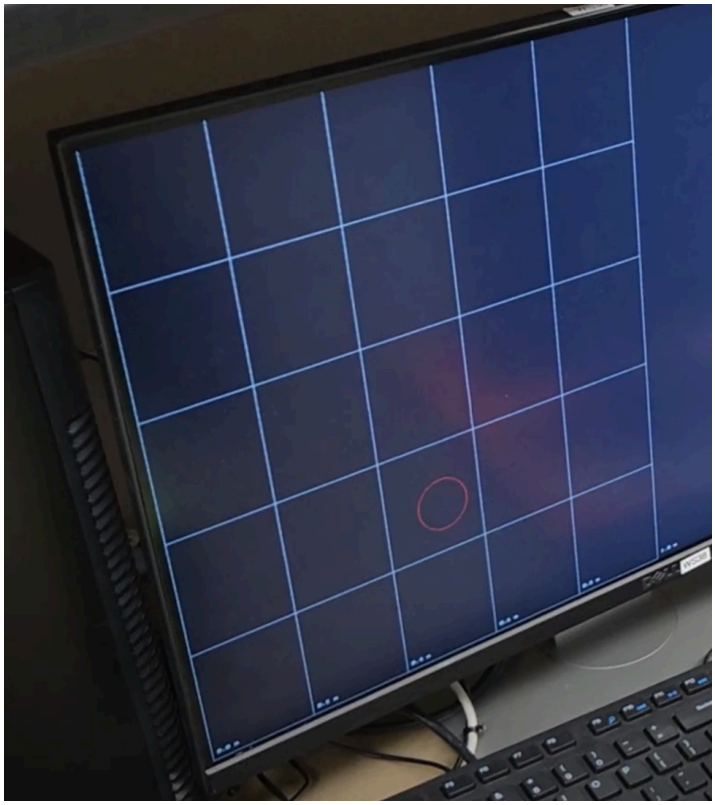


Figure 5: Second screen x-y localized coordinates display.

Both screens map the real-world coordinates in ranges from [0m…1m] on the z-axis, [-1m…1m] on the x-axis, and [0m…2m] on the y-axis to the pixel dimension with a granularity of 200mm for each axis on both screens.

Ultimately, many of the features to be displayed with the HDMI were not relevant in the final design, including Bresenham's line algorithm for the trajectory path, the filled circles for

denoting landing position, and a simpler but more cohesive GUI that plots the x-y localized coordinates after filtering for noise. The mapping of the real-world coordinates and the pixel coordinates was also not perfectly aligned, and could have been tuned more finely for a more accurate representation of the object's location.

2.1.4 Servo Controller

The Servo Control IP was able to successfully track and point a laser at test objects. To better view the qualitative results of the Servo Control, it's best to view the video demo in section 7.0; however, below are two screenshots demonstrating the servos' reaction to the movement of a test object. Figure 6 shows the pan and tilt servos pointing the laser at the object on the left side of the ultrasonic array, and Figure 7 shows the servos pointing at the object once it's moved towards the right side. We can see that after some calibration of the Servo Control IP, the laser successfully lands on the object.

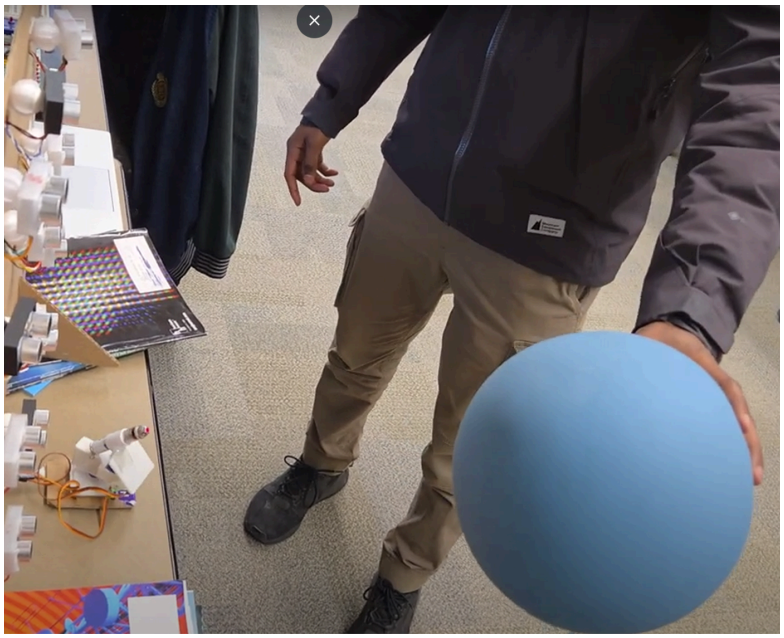


Figure 6: Servos point laser at object in one frame

Figure 7: Servos point laser at object in second position in the second frame

Some extra testing was done with an oscilloscope to give us extra confidence that the IP was working the way we wanted it to. We wanted to ensure that the pulse width, which is what controls the servo angles, is accurate, reflects the correct pan and tilt angles, and matches our simulations. A balloon was held in a position in front of the sensor array that matched our simulated positions, and the corresponding pulse width was measured. Since we used a lookup table for calculating the pulse widths, they almost exactly matched the ones in simulation.

2.1.5 Trajectory

The algorithm used to track the trajectory of the object did not perform as well as we had hoped. As seen in Figure 8, the predicted positions of the object had a large overshoot, and the predicted landing positions were very inaccurate. The goal for the landing position was to have at least 90% of the points within a 50 mm radius of where we expected the landing position to be. As seen in the plot in Figure 8, the landing positions varied across over 200 mm in the x-axis and by almost 1000 mm in the y-axis. As a result, the trajectory algorithm was abandoned as it yielded completely inaccurate results.

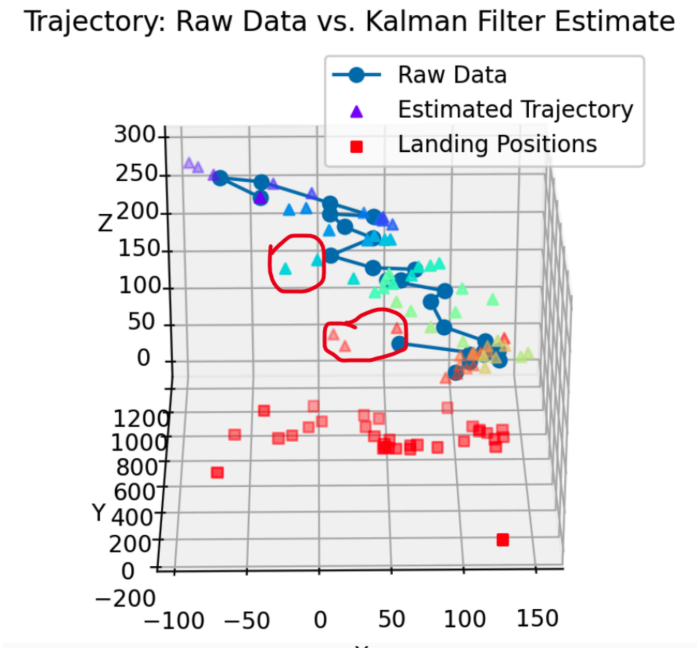


Figure 8: Predicted path of falling object in front of sensors (large overshoots present).

2.2 Future Work

To achieve our goal of tracking real time objects, we propose the following solutions. As we see it there are two routes: Either improve the field of view of the ultra sensors to provide trajectory with more data points, or combine the ultrasonic sensor data with a camera.

The first should be a simple solution. Assuming there are enough logic level shifters and I/O ports, adding more sensors will inherently provide a larger frame to work with. The IPs are already capable of working with any number of sensors. The only additional work is specifying the sensor configurations and neighbour maps. Also, considering we are intending to observe an object falling down rather than straight towards the sensors, it might be ideal to angle the sensors up towards the sky. This will take advantage of the axis that the sensors work best with, which is orthogonal to their plane of orientation.

The second solution will require a complete rework, but we believe this will provide the best results. Adding a camera can greatly improve the number of data points and resolution of localization outputs in the x-z plane. Combining this with the 10mm accuracy of the parser along

the y axis gives a tremendous opportunity to observe both depth and image plane data. This will provide trajectory with better data points to make accurate predictions of where the object is headed.

Lastly, we would recommend switching HDMI for VGA. The HDMI IP required much longer compile times which slowed down a lot of our integrated testing. The pixel and colour resolution of VGA is more than ample for our purposes and requires less area and effort to compile.

3.0 Project Schedule

Table 2. Overview of the project milestones and accomplishments

Milestone	Original Proposal Objectives	Actual Accomplishments	Evaluation of Differences
1: Initial Design Review & Test Bench Setup	- Finalize overall system design - Identify key hardware and software components	- Finalized system block diagram and basic interconnections in Vivado - Conducted initial sensor hardware tests with Arduino - Began outlining the trajectory prediction algorithm	Alignment: The team successfully completed their intended tasks. Early recognition of ultrasonic sensor latency and PMOD constraints led to immediate design optimizations (sensor pin usage).
2: Sensor Calibration	- Initial tests of sensor outputs	- Developed initial algorithms for sensor	Partially Ahead: More advanced tasks,

and Data Acquisition	- Begin sensor data mapping algorithm	mapping - Implemented HDMI IP and initial GUI for landing position - Simulated servo control logic - Developed initial custom IP for sensor parsing	such as HDMI visualization and servo control simulation, were initiated earlier than scheduled. Challenges arose with logic level shifting and the complexity of 3D localization.
3: Triangulation & Trajectory Prediction Development	- Algorithm for object triangulation and trajectory prediction	- Defined optimal sensor placement - Developed a program for object position prediction - Audio and tri-color LED system initial implementations began - Integrated ultrasonic sensor IP for simultaneous distance readings	Mostly on Track: Significant progress on triangulation and initial trajectory algorithms was made. Challenges included wave reflections between sensors and AXIS IP integration complexity.
4:	- Integrate servos and	- Developed Kalman	Deviations: More

Servo Control and Laser Targeting Integration	laser system with predictive module - Calibrate laser alignment	filter (initially linear, then quadratic) for trajectory prediction - Implemented lookup table instead of CORDIC IP for servo angles - Completed initial HDMI visualization with interactive features - Addressed sensor cross-talk and synchronization	complexity encountered in trajectory prediction than expected, causing a pivot to Python validation. Servo and HDMI integration have advanced significantly, but without direct laser integration yet.
5: HDMI Visualization & Audio Integration	- HDMI visualization of landing positions - Audio alert integration	- HDMI display adjusted and refined for accuracy - Integrated servo control successfully - Developed sensor fusion algorithm - Implemented reconfigurable sensor setup and DSP	Aligned with Adjustments: Strong advancement in HDMI visualization and servo control was achieved. Audio integration was slightly delayed. Focus shifted towards optimizing sensor fusion and accuracy.

		optimizations	
6: Final Demo Integration	- Final integrated system - Comprehensive demo	- Initial trajectory and landing position IP modules written - Integrated trajectory module into Verilog (from Python) - Servo adjustments for precise tracking - Progressed towards final end-to-end integration	Behind Schedule: Complexity in translating trajectory computations from Python to Verilog introduced delays. End-to-end integration was ongoing, not fully completed because of trajectory/landing position IPs. Final integration and smoothing noisy inputs were still in progress.

4.0 Description of IP Blocks

4.1 Parser

The sensors used for our system are the HC-SR04 ultrasonic sensors, as seen in **Figure** 9.

Figure 9: HCSR04 Sensor. The transmit channel on the left (T) and the receive channel on the right (R). Image from [1]

These are standard and low-cost sensors used in many beginner Arduino projects that provide up to 3 mm distance accuracy and 400 cm of range [1]. They operate in the same way as echolocation: Send a burst through the transmit channel and wait for the echo response on the receive channel. The sensor is controlled using the protocol in **Figure 10.**

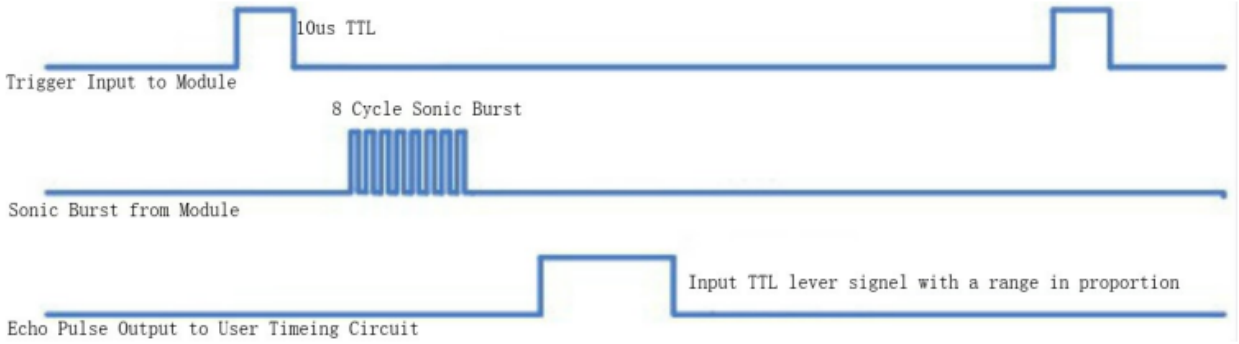


Figure 10: Timing diagram of HC-SR04 sensor. Image from [1]

The HC-SR04 has 4 pins: VCC (5V), trigger, echo, and ground. The trigger pin is used to initiate an acquisition from the sensor. It expects a 10-µs long high pulse, which will command the integrated circuits on the sensor to send an 8-cycle sonic burst. Once this is complete, the sensor

will immediately drive the echo pin high until it registers an echo response or it times out after 38 ms. Since the time for which the echo pin is high is equal to the time it takes for sound to travel to an object and back, the distance can be computed as:

$$distance\ (mm)\ = \frac{1000*(echo\ high\ time\ in\ s)*343}{2} \qquad (1)$$

To improve stability, a 60 ms interval is suggested between triggers, but we have found that a 10 ms buffer after the echo signal goes low is good enough. This means our worst-case cycle time is just over 48 ms.

Since the Nexys video board does not have a 5V output and the input and outputs are not 5V tolerant, an external power supply and PMOD logic level shifter is required. To save the number of inputs/outputs on these PMODs, we realized we can trigger all the sensors using a single wire, reducing the number of wires from 2*N per sensor to just N + 1 per sensor. The high fanout is not an issue in this case as well, since the trigger pin is low frequency (20Hz) and has a 10us pulse width. This means the signal would be unaffected regardless of whether the load capacitance was 1 uF and the wire resistance was 10 mOhms.

For our setup, we use the first pin of port JA as a trigger and the rest of the pins of port JA and JB for echo. These can be found in the constraints file in both the Parser and MeteorDestroyerSystem projects.

4.1.1 IP Description

The IP contains two modules. The first is hcsr04_sensor, which reads the data associated with a single sensor, and the other is array_parser; which serves as a top-level to instantiate a parameterized number of hcsr04_sensors, coordinate the trigger, and output the distances of each sensor in an array with a valid signal.

The hcsr04_sensor runs on a 100 KHz clock (slowed down from 100 MHz input) since 0.01 us is more than enough precision for the sensors, and the extra bit width only makes the

adder/multiplier size larger and slower. The control flow of an hcsr04_sensor is as follows (Figure 11):

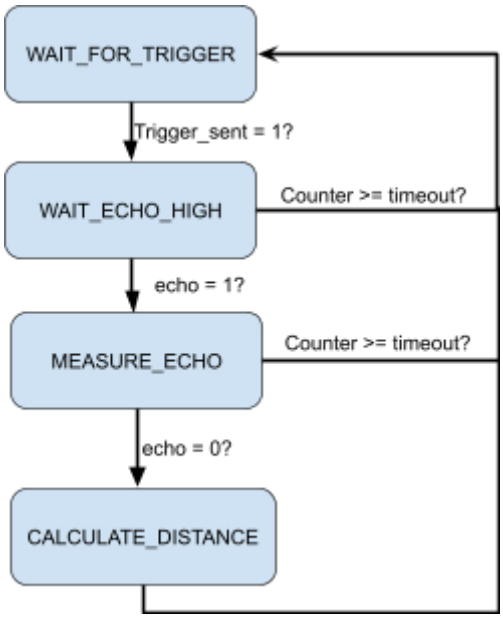


Figure 11: Parser state machine.

- **WAIT_FOR_TRIGGER:** The sensor starts in this state, where it waits for a trigger_sent input from the array parser. Once this is seen, it resets its counter and asserts not_ready to tell the array_parser not to begin another transaction.
- **WAIT_ECHO_HIGH:** The sensor requires some non-zero time to assert the echo signal high, so we wait here until that happens or we timeout; in which case, we set the distance_mm to the max value, assert valid, and go back to wait for a new trigger. This case is meant to handle a disconnected sensor, which should not be possible if a sensor is connected properly.
- **MEASURE_ECHO:** Start counting until the echo signal goes low. The counter value is constantly being multiplied by 343/100 (in U2.7 fixed point notation) and assigned to "product" to prepare for compute_distance. Note: We divide by 100 because echo_time is computed using counter_value/clk_freq(100,000), but we can simplify equation (1) by dividing by 1,000, leaving us with just 2*100 in the denominator.

- **COMPUTE_DISTANCE:** Compute the distance using equation (1). Since we still need to divide “product” by 2, and it is currently in U(ceil(log2(TIMEOUT)) + 2).7 format, we will discard the lower 8 bits to keep only integer bits and effectively right shift by 2.

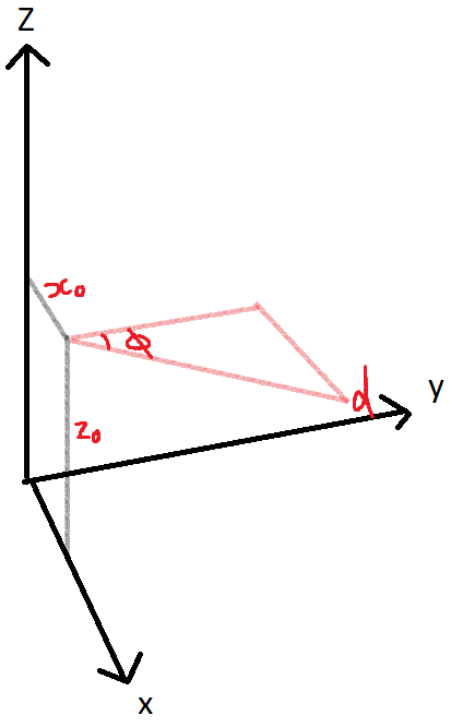
The array parser serves as a top-level component to coordinate and organize all sensor data. It has two states:

- **SEND_TRIGGER**: The parser starts here by driving the trigger signal high for 10us. Once this happens, it drives the trigger_sent signal high to all the hcsr04_sensor instantiations for one cycle and moves to idle.
- **IDLE:** The array parser waits here while the sensors are not ready. Once the OR of all sensor not_ready outputs is 0, it will begin counting to 10 ms and then resend another trigger.

To test this IP, we created a testbench that can be found under Parser/Parser.srcs/sources_1/tb/tb.sv. This testbench mimics real sensors by waiting for the trigger to go high and then responding by driving N echo signals with different delays. The output of the simulator was then validated by hand. Because of the long cycle times between trigger events, the simulator is not ideal for testing many test vectors. We quickly moved on to testing with the ILA debugger in real life with the FPGA connected to real sensors.

4.2 Object Localizer

The localizer IP’s purpose is to determine an object’s relative x, y, and z position given an array of sensor data. The basic operation to recover the object’s position is as follows (Figure 12):



$$\text{Object Position} = \begin{bmatrix} x_0 \\ 0 \\ z_0 \end{bmatrix} + \begin{bmatrix} d\sin(\theta) \\ d\cos(\theta) \\ 0 \end{bmatrix} \quad (2)$$

Figure 12: Mathematical model to derive location from distance

Unfortunately, this method results in very coarse information as we can't determine if an object is in between two sensors. To overcome this, we take the minimum three distances, compute their positions and then average the result. The IP also implements outlier rejection by confirming whether the minimum three distances are neighbours (within one sensor) to each other. If they are not, it takes two of the three sensors that are neighbors and recomputes the minimum three using the sensors that lie within that common neighborhood. The datapath is as follows (Figure 13):

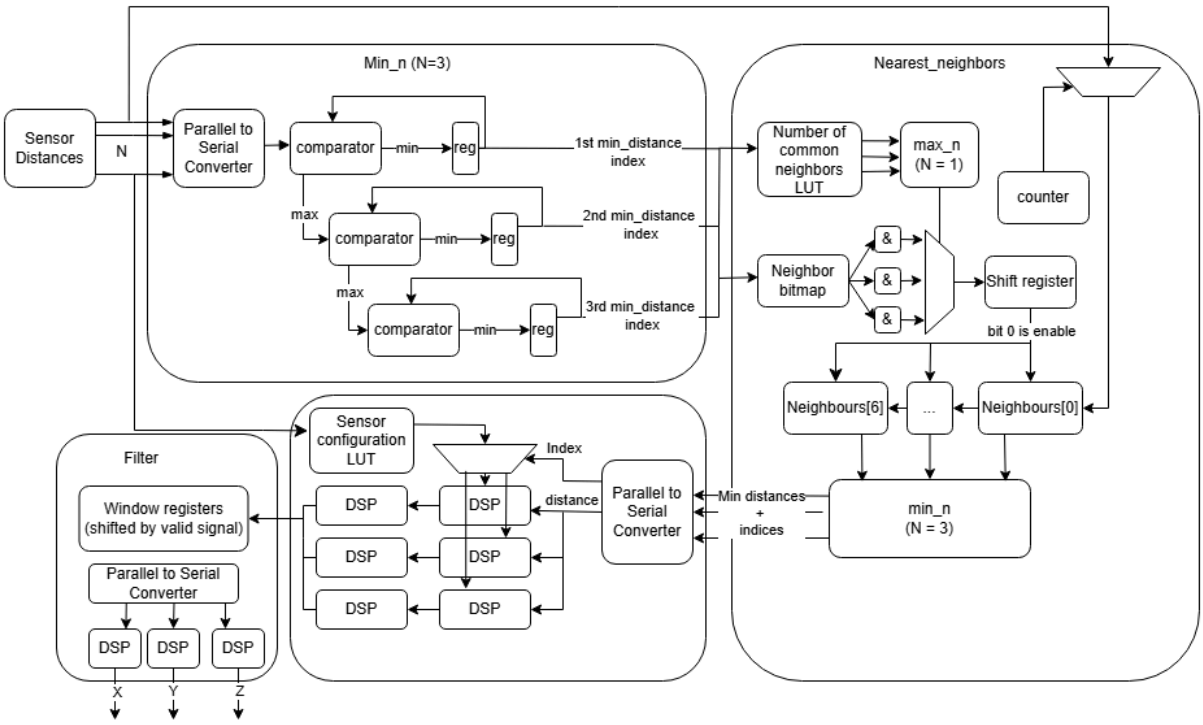


Figure 13: Data flow of localizer IP

4.2.1 Parallel to Serial Converter

The parallel to serial converter takes BUS_WIDTH elements and outputs them serially every clock cycle. It is implemented using a shift register, and the input is latched using a valid signal. The output valid signal is held high for BUS_WIDTH cycles as data is shifted to the output.

4.2.2 min_n/max_n

The min_n module takes some BUS_WIDTH elements and their corresponding indices as inputs and outputs the N minimum/maximum values. It does this by first converting the input into a serial stream and comparing each element with the subsequent element. The minimum element of the first comparator is held at the first level, and any value that is not the minimum so far is passed down as input to the next level. Once all elements have been processed, the output registers will hold the min_distance values, and the output valid will be set to high for one cycle. The max_n module operates in the same way but reversing min/max operations and wires.

4.2.3 Mult2_add and Mult2_accumulate

These two modules implement (A*B) + C and the sum of A*B over many cycles. These are wrappers for the DSP primitive that allow full control over which registers are enabled, operation mode, and seamless support for signed value and fixed-point conversion to integer.

4.2.4 Nearest Neighbors

This module is designed to receive an adjacency matrix of sensor neighbors, all the sensor distances and their indices, and the global minimum N distances. It then outputs the minimum N distances and their indices that all lie within a neighborhood. The motivation behind this module is to remove outliers if a sensor randomly receives a small value when it shouldn't.

The bitmap contains a 1 at bit j in register i if sensor i is a neighbor of sensor j, or if i = j. By performing an AND of any two bitmaps together, we can find the intersection of the sensors' neighborhoods. This will allow us to restrict our search for minimum distances to this neighborhood only and remove outliers that push our localization to the wrong neighborhood.

To get a good guess of where the object is, we take the global 3 minimum distances and find the intersection of every combination of their sensor neighborhoods. If there is no outlier, then all 3 will reside in a common neighbourhood of one of the sensors. If there is an outlier, then at least one sensor will have no common neighbours. Thus, the sensor with the maximum number of sensors in its intersecting neighbourhood will likely have no outliers.

Once we have the intersecting bitmap with the most neighbours, we shift the bitmap by one and select the corresponding distance. If there is a 1 at bit0 after shifting, then we enable the neighbors' shift register to capture the distance and index. The neighbours' shift registers are 6 registers since the intersection of two sensors can have at most 6 sensors. After we have exhausted the bitmap, we find the minimum distance using sensor distances restricted to some neighborhood intersection.

4.2.5 Localization

Now that we have a good guess of which sensors to use, we can look up their configurations (x0, y0, z0, and theta) to compute x, y, and z. We multiply each value by ⅓ and accumulate the results to derive the mean of the 3 minimum distances.

4.2.6 Filter

To combat noisy sensor data, we added a time filter at the end of the localizer. This contains a window of registers for x, y, and z and computes the mean over 5 valid localization outputs. Since our sensors run at 50ms, the window can not be too large or it will suffer from high latency.

4.2.7 AXI

A huge help in tuning our configuration was adding an AXI slave interface to update sensor configurations without recompiling. It also allowed us to read sensor values and localization outputs over UART in real time. This helped us not only validate that the localization worked, but also allowed us to record data, plot, and pass it to other algorithms, such as trajectory, to validate if they could even theoretically work

To test this IP and all of its modules, there are several testbenches. These can be found in Parser/Parser.srcs/sources_1/tb/. Testing the IP in real life consisted of throwing a slow-moving object, such as a balloon, at the sensors, as captured in the image below (Figure 14)

Figure 14: Testing setup for localizer IP

4.3 Servo Control

A crucial component of our system is the IP block responsible for controlling the laser's orientation towards the detected space debris. This module is implemented using a pair of pan and tilt servo motors. The pan servo adjusts the horizontal (x-y plane) angle, while the tilt servo adjusts the vertical (y-z plane) angle. Together, they enable precise positioning of the laser to target any location within our defined coordinate space. A custom 3D-printed mount designed to securely integrate these two servos is depicted in Figure 15.

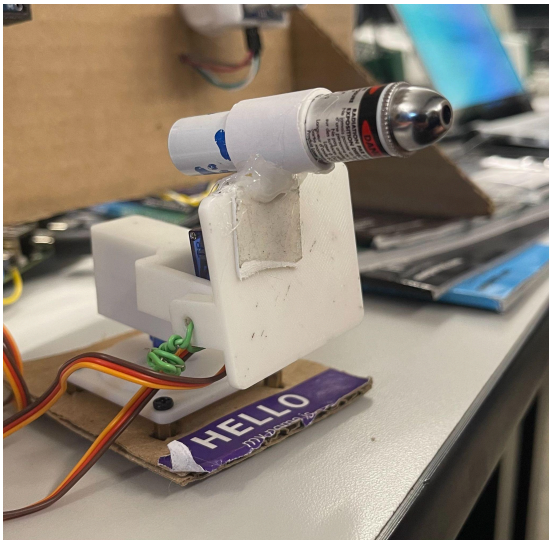


Figure 15: Pan and Tilt 3D-printed mount

4.3.1 Angle Calculation

First, we needed a reliable method for determining the required pan and tilt angles based on the object's detected position. The pan angle is calculated as the angle between the object's position and the y-x plane, while the tilt angle is derived from the angle between the object's position and the z-y plane. The geometric principles behind these calculations are illustrated in Figure 16.

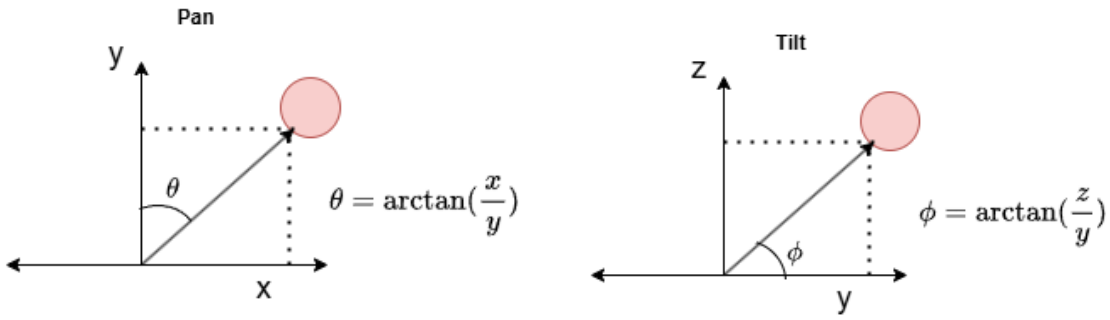


Figure 16: Geometry behind pan and tilt angles

4.3.2 Arctan Computation

Calculating the arctan function on an FPGA is inherently challenging. Initially, we attempted to implement this calculation using the Xilinx CORDIC IP core, known for efficiently performing various high-precision trigonometric operations. However, integrating the CORDIC IP proved complex due to its AXI-Stream interface and difficulties with fixed-point input/output data

representations. Given the moderate precision requirements and the benefit of simplicity in integration, we ultimately chose to use lookup tables (LUTs) for arctan computation.

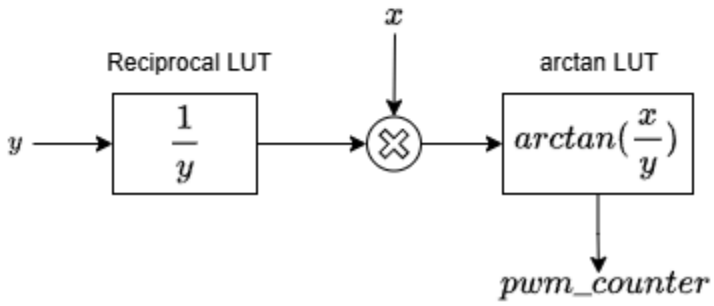


Figure 17: Arctan calculation flow diagram

The process of calculating the required angles is outlined in the flow diagram in Figure 17. The first step is to perform division (x/y). Since direct hardware division using the FPGA's DSP blocks results in long data paths and significantly lower operational frequency, we employed a reciprocal LUT method for efficient division emulation. Specifically, we created a reciprocal LUT whose input is 'y' and whose output is '1/y', represented in a 0.16 fixed-point format. Multiplying 'x' (int16) by this reciprocal provides the division (x/y) in a 16.16 fixed-point format. The result is then shifted to a 1.13 format as input for our arctan LUT.

Servo motors are controlled via Pulse Width Modulation (PWM) signals operating at a 50 Hz frequency. The servo angle corresponds directly to the PWM signal's pulse width. According to the specifications, a 1 ms pulse positions the servo at -90°, a 1.5 ms pulse centers it at 0°, and a 2 ms pulse achieves +90°. Intermediate angles are set by adjusting pulse widths between these extremes. To efficiently map calculated angles directly into servo PWM signals, we implemented an additional LUT that converts arctan results directly into PWM counter values, thus streamlining the process.

4.3.3 Angle Calibration

After functional verification through simulation, we integrated the servo control IP with our localization IP. During initial testing, we observed that although the servo followed our test object's direction, it failed to accurately reach the intended positions. After confirming the absence of logical errors within our IP's, we suspected inaccuracies in the servo specifications. To verify this, we used an Arduino to empirically test and calibrate the servo's actual pulse width-angle relationships beyond the manufacturer's stated ranges. As it turns out, the specifications were quite off, with a 0° angle corresponding to a pulse width of 1.83ms (rather than 1.5ms). Through this empirical calibration, we recorded accurate pulse width-angle pairings, adjusted the LUT accordingly using a polynomial fitting to the extreme width-angle pairings, and resolved the discrepancy. Consequently, the servos accurately tracked the object, enabling the laser to reliably follow and simulate object interception.

4.4 HDMI

HDMI functionality was implemented using an existing Vivado IP and demo code from the Digilent website [2]. Several versions of the IP were exported in a way that was not user-agnostic; specifically, the exported project used paths that were specific to the original designer, meaning that it was necessary to update all those paths to be congruent with our design. The 2022.1-1 version of the HDMI IP was selected as the optimal candidate because it introduced the least errors in upgrading, and that version of Vivado was also selected for the final design integration because the HDMI IP had the largest number of individual IPs when compared to the Vivado IPs used in the custom IPs written for this design, making integration simpler.

The only addition to the existing Vivado HDMI IP was adding the board's 5 push buttons as an input, and with interrupts so that the display on the screen can switch based on which button is pressed.

The data to be written to the screen was transmitted using HDMI and was done in C using the MicroBlaze processor. The localized position of the object is passed to the MicroBlaze processor using AXI, and that address is then read from in the MicroBlaze core with it represented on the

screen via a circle. The dimensions of the screen are scaled with respect to the real-world coordinates, i.e. the pixel coordinates and the real-world coordinates were mapped. The scaling was done on the pixel, circle, and grid "level" and was computed based on the granularity required. Specifically, the x and y axes both have a length of 2m, and 200m granularity was determined to be detailed enough, hence, that was used to generate the lines on the grid and map the position of the object on the screen to its position in real-life relative to the sensors.

The initial goal was to use Bresenham's line algorithm to draw the trajectory path based on what the trajectory IP sends via AXI to the MicroBlaze, hence, initially, we used a buffer array filled with arbitrary x-y values corresponding to hypothetical real-world coordinate values that would draw a line in the direction of the trajectory. Then, when the buffer becomes full after a period of time, a red circle would display as the "heat map," and the final x-y coordinates of the object will be saved in a secondary buffer for all the landing positions, seen in Figure 18. So, the locations of the landing positions are saved to a "second screen" that, upon a button interrupt, would trigger the switch and display a heat map of all the positions the object had landed at up until that point.

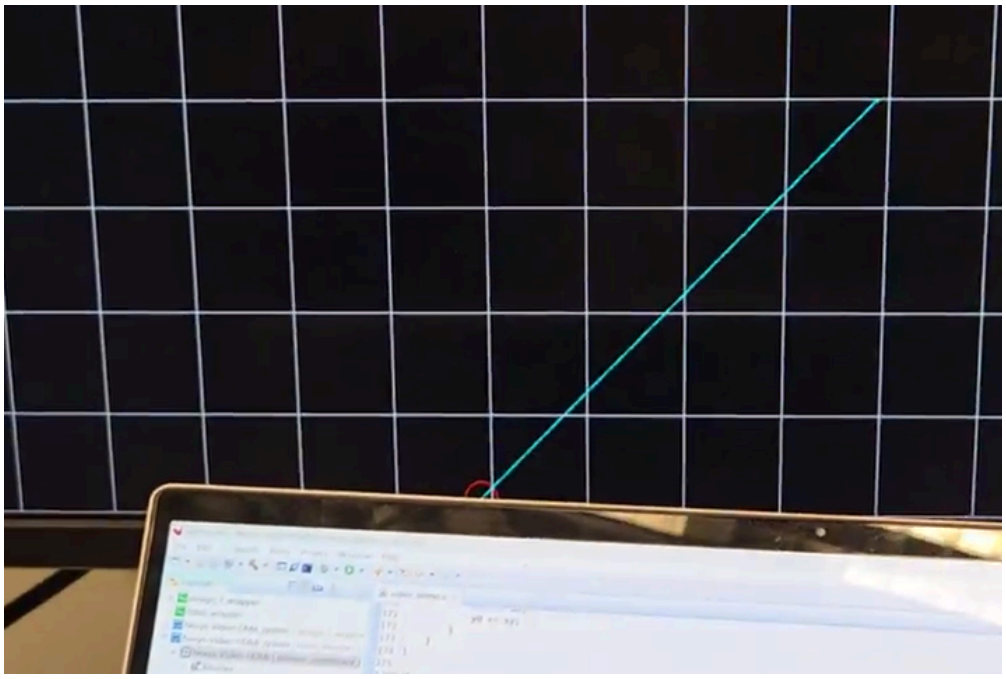


Figure 18: Trajectory path display with pseudo-values.

Then, when the trajectory and landing position custom IPs are ready to integrate with the HDMI and frame generation, the buffer storing the pseudo-trajectory values would be removed from the code, and the values from the IP would be directly fed to the MicroBlaze in the same way localization outputs are. The addresses with the predicted trajectory path and landing position would be read and drawn on the screen.

However, testing the display before the parser filtering to reduce noisy data was implemented, and before the trajectory IP was implemented, led to a very sporadic "trajectory" display. Hence, the line algorithm was removed from the display until it could be drawn in a more comprehensible and user-friendly way. Until then, the HDMI would simply display the localized x-y position of the object using a circle to denote position on the screen, as seen in Figure 19.

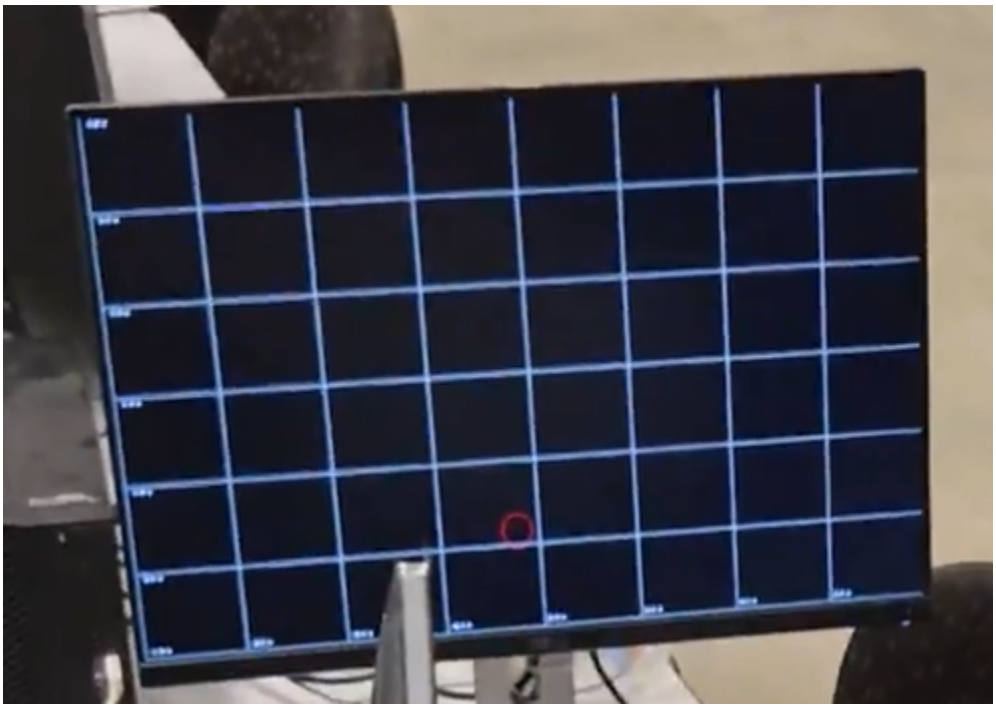


Figure 19: Initial x-y localized data display.

When the trajectory IP was not able to provide a clear predicted path, and it was not feasible to optimize it in time, the HDMI was updated to remove the original functionality of displaying the trajectory path and landing position, as they could not be reliably predicted, and instead display two screens. The first screen would display the x-z and y-z localized coordinates of the object, while the second would display a screen similar to Figure 19, which shows the x-y localized

coordinates. Both screens would display the position in real-time, using a double-buffer technique, and represent the position as a circle. With an 8×8 bitmap font being used to generate the labels for the axes.

4.5 Trajectory

One of the more ambitious aims of our project was to track the full trajectory of a moving object, which included not only estimating its real-time position but also predicting its acceleration, velocity, future location, and eventual landing spot. Achieving this goal required significant effort to determine an effective solution. Ultimately, we decided to implement a Kalman Filter (KF) to account for the high measurement uncertainties produced by the ultrasonic sensors [3]. By combining the KF with kinematic equations that describe the object's acceleration, velocity, and position, we hoped to produce robust predictions of the object's future state.

To validate the filter's accuracy and behavior, we started by simulating simple paths in Python and confirmed that the KF could track those paths appropriately. Next, we ported the filter into Verilog and tested it against the same paths to ensure both correctness and consistency in the hardware implementation. An important detail in the design of our filtering approach was how we handled the time between sensor inputs. Although a basic counter could track time steps, incorporating actual time measurements would have made the computations more complex and forced the system to run for significantly more cycles (for instance, handling equations like $d = 1/2at^2+vt$). Instead, we opted to maintain a consistent time delta of one unit between outputs, effectively removing the explicit time variable from the calculations and streamlining the process.

Despite this simplification, we still had to face the constraint that sensor updates arrived at only about 20 Hz. In essence, our system would use updated sensor readings and the Kalman Filter whenever a new measurement arrived, but it would otherwise rely on existing estimates of acceleration, velocity, and position for intermediate predictions. This means that output from the trajectory can take one of two paths: the first is when we get new sensor values and use them to update our KF, and the other is when we use existing values to predict the next steps. To ensure the same time delta, the number of cycles for each path needs to be exactly the same. Figure 20 shows detailed schematics drawn to determine the exact number of cycles of the update path.

However, once we began collecting real data from our localization outputs, we discovered that the ultrasonic sensors introduced far more noise than anticipated. In three-dimensional tests, the object's path did not behave reasonably in front of the sensing wall: the data zigzagged severely, especially in the x-y plane for a falling object. Our simulations in Python confirmed that the level of noise present was beyond what our KF approach could effectively handle. As mentioned in the results section, predictions often overshot the true path and gave unreliable landing positions (Figure 8).

Due to the combination of severe noise and time limitations toward the end of the project, we ultimately decided to scrap the trajectory tracking component and refocus our efforts on refining basic localization. We hoped this would at least provide a more stable foundation for tasks such as servo-based tracking, which depended on having reliable positional data. While we had to step back from the original plan for full trajectory prediction, the results from improving localization allowed us to accomplish the project's core goal, which was to track the object with the laser. The lessons learned about sensor noise and filtering will also be invaluable for future iterations of this work.

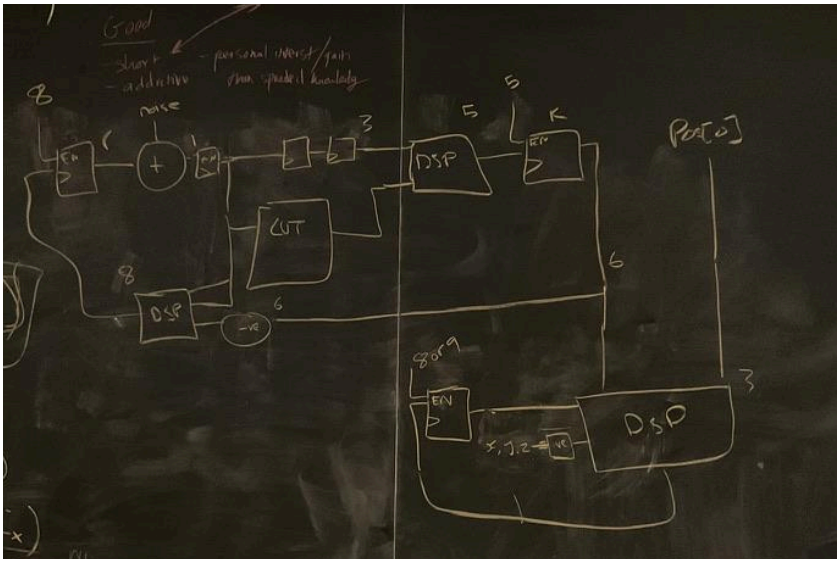


Figure 20: Detailed schematics of the update path of KF in trajectory estimation.

5.0 Description of Your Design Tree

The design is posted publicly on GitHub in a repository named “MeteorDestroyer” [4]. The repository contains the custom IP modules for the Parser (Parser_ip), Localization (Localizer_ip), and the Servo Controller (servo_controller_ip). The Vivado IP module for HDMI (hdmi), and the Vitis code (hdmi_vitis) for the MicroBlaze processor to write the localized sensor data to a screen using HDMI are also included.

The complete integrated design is located in the MeteorDestroyerSystem folder, while the design excluding HDMI is in the Parser folder, which was primarily used for testing servo functionality.

6.0 Tips and Tricks

A key piece of advice for future students tackling this project is to prioritize early planning and to break the workload into parallel tasks whenever possible. Identify each subsystem early on, for example, sensor acquisition, data processing, and communication, and assign them to different team members so that development can happen concurrently rather than in sequential isolation. When selecting sensors, take note of their accuracy, update frequency, and how easily they integrate with your chosen hardware and communication protocols. Poor sensor choices can introduce unnecessary noise and delays, whereas robust, well-documented components will save countless hours of debugging and calibration. Similarly, designing a clear communication architecture from the start ensures that every module works smoothly together. These steps, combined with careful scheduling of tasks and milestones, can greatly streamline development and help mitigate last-minute complications.

7.0 Project Video

A video of our project can be found on YouTube (https://youtu.be/uPklYhPbAa) or on Google Drive (https://drive.google.com/file/d/1fEneqZVOYIQjqXRhuuc_AZwqC5ROKEhK/view?usp=sharing).

8.0 References

[1] Ultrasonic ranging module HC - SR04 3.3V, https://www.gotronic.fr/pj2-ultrasonic-sensor-hc-sr04-3-3v-1912.pdf (accessed Apr. 7, 2025).

[2] NEXYS video HDMI demo - digilent reference, https://digilent.com/reference/programmable-logic/nexys-video/demos/hdmi (accessed Apr. 7, 2025).

[3] D. Tomer, "What I was missing while using the Kalman filter for object tracking," Towards Data Science, https://towardsdatascience.com/what-i-was-missing-while-using-the-kalman-filter-for-object-tracking-8e4c29f6b795/ (accessed Apr. 7, 2025).

[4] MuazShash, *MeteorDestroyer*. GitHub repository, 2022. [Online]. Available: https://github.com/MuazShash/MeteorDestroyer (accessed Apr. 7, 2025).