Autonomous Line Following and Landing of a Parrot Mambo Minidrone Using Simulink

Chinmay Amrutkar[1]
[1]Arizona State University, Tempe, AZ

Keywords:

Parrot Mambo
Drone
MATLAB
Color Detection
Color Threshold
Line Following
Autonomous Landing

ABSTRACT

Unmanned Aerial Vehicles (UAVs) offer significant potential for automated tasks requiring navigation in structured environments. This project focuses on developing an autonomous line following and landing system for the Parrot Mambo minidrone using MATLAB and Simulink. The core approach leverages onboard camera feed processed within a dedicated Simulink block using color thresholding to isolate the desired line and submatrix analysis to determine the line's position relative to the drone's center view. A Stateflow controller interprets this positional information to issue appropriate yaw and forward velocity commands, guiding the drone along the line. Landing is autonomously triggered when the line is no longer detected in the upper or side regions of the camera's view, indicating the drone is centered over the landing target. This report details the methodology, including the image processing pipeline, Stateflow control logic, and landing sequence, demonstrating a viable approach for vision-based autonomous navigation and landing for compact UAVs.

Corresponding Author:

Chinmay Ravindra Amrutkar
Arizona State University, Tempe, AZ, USA
Email: chinmay.amrutkar@asu.edu

1. INTRODUCTION

The proliferation of Unmanned Aerial Vehicles (UAVs) has opened avenues for automation across various domains. A fundamental capability for many autonomous tasks is the ability to follow predefined paths, such as lines marked on a surface. The Parrot Mambo drone, a compact and accessible platform with an onboard camera, serves as an excellent tool for developing and testing such autonomous navigation algorithms. This project addresses the challenge of enabling a Parrot Mambo minidrone to autonomously follow a colored line on the ground and execute a controlled landing at the line's end using MATLAB and Simulink.

The primary objective is to implement a robust system where the drone uses its downward-facing camera to detect the line, adjusts its heading (yaw) and position to stay centered over the line, and automatically lands when it reaches a designated point (indicated by the line's termination or a specific marker like circle). The system relies heavily on two key components within Simulink: an Image Processing block for line detection and position estimation, and a Stateflow block for managing the drone's behaviour and control logic based on the vision input. This integration demonstrates a practical application of computer vision and state-based control for UAV autonomy.

2. SETTING UP THE PARROT MAMBO DRONE

To successfully implement real-time autonomous navigation with the Parrot Mambo drone, it is essential to first establish a stable connection between the drone and the development environment. This setup

involves installing the necessary drivers, configuring the communication interface, and verifying hardware functionality through initial deployment tests.

1. Installing Bluetooth Drivers and Establishing Connection:

The Parrot Mambo drone connects to MATLAB and Simulink via Bluetooth, requiring proper driver installation on a Windows system. Following the guidelines from MathWorks, the appropriate Bluetooth drivers were installed to ensure stable wireless communication. Once the drivers were set up, the connection was verified by pairing the drone with the computer and confirming its availability in the Simulink support package.

2. Deploying the Parrot Getting Started Model:

To validate the connection and test the drone's hardware, the parrot_gettingstarted model from the MathWorks documentation was deployed. This model executes a basic motor test by spinning opposite motors alternately, ensuring that all four motors function correctly. Successful execution of this test confirmed that the drone was properly configured and ready for further experimentation.

By completing this setup, the foundation was established for integrating the drone with MATLAB-based image processing and control systems, enabling further development in color-based object detection and UAV automation.

Figure 1. Parrot Mambo Drone

3. METHODLOGY

Following the successful setup of the Parrot Mambo drone, this experiment was conducted in a structured manner to achieve autonomous navigation integrated with real-time color detection and autonomous landing. The core of this project lies in the Simulink model design, integrating real-time image processing with Stateflow-based control logic to achieve autonomous line following and landing.

3.1. Simulink Model Setup and Framework Analysis

- Launch and Template Selection:
    - Begin by launching Simulink and open *parrotMinidroneCompetitionStart* template, by entering *parrotMinidroneCompetitionStart* in Command Window. This template establishes the basic framework required for integrating image processing with flight control.

- Framework Analysis:
    - Carefully analyze the pre-configured image processing block to understand its workflow and ensure that it can interface seamlessly with the drone's downward-facing camera.

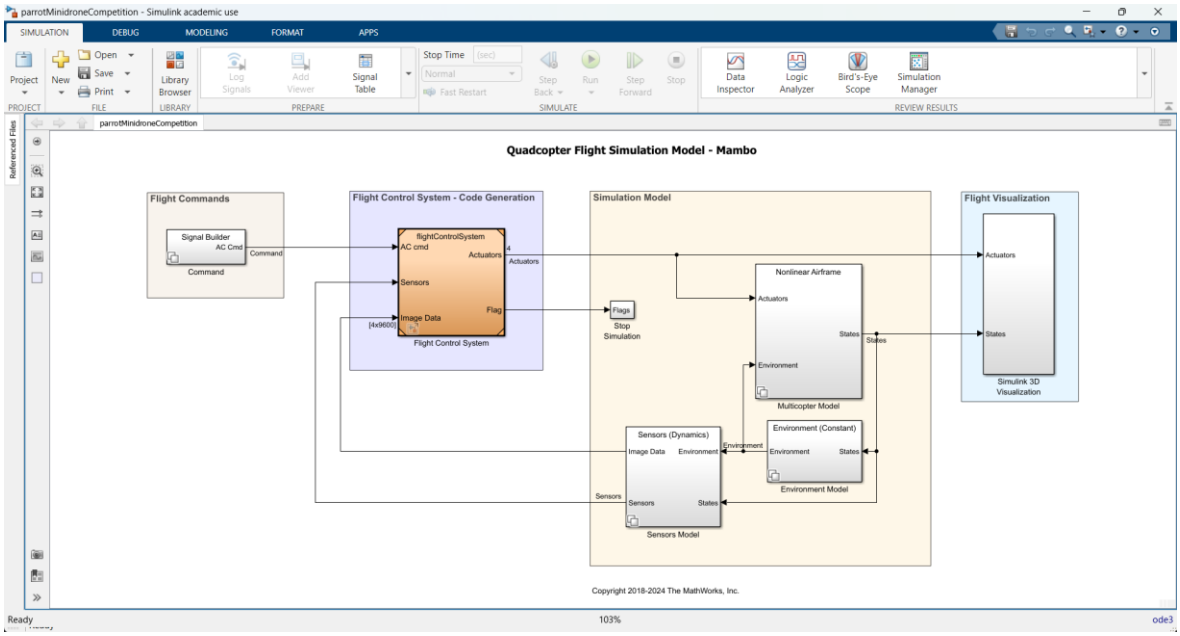

Figure 2. Quadcopter Flight Simulation Model – Mambo project

3.2. Image processing System

The image processing module is the visual core of the autonomous navigation and landing system for the Parrot Mambo minidrone. It processes live image data from the drone's onboard camera to identify the presence and location of a specific colored line, enabling the drone to follow that line accurately and detect the correct moment to land. This subsystem is implemented in Simulink and is composed of multiple blocks that handle image conversion, color segmentation, noise reduction, region-based analysis, and logic formulation for downstream flight control.

The process begins with the **PARROT Image Conversion** block, which takes the image input from the drone in the Y1UY2V format and converts it into a standard RGB format. This RGB image is essential because most image processing operations, including color thresholding, rely on the RGB or HSV color space for analysis. The converted R, G, and B components are then passed to a custom MATLAB Function block labeled red_color, which performs the thresholding operation. This function shown in Figure 5 was generated using MATLAB's Color Thresholding App Figure 4, where a red-colored path shown in Figure 3 was sampled and threshold bounds were carefully selected to isolate only the pixels corresponding to the line. The result is a binary mask: a black and white image where white pixels (value 1) indicate the presence of the desired color (red), and black pixels (value 0) represent the background.

Fig. 3. Captured Image

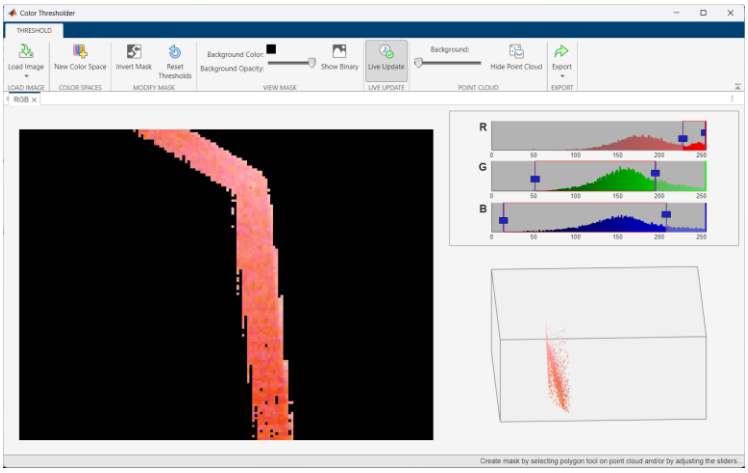

Fig. 4. Color Thresholding App (fine tuned for red color)

```
function BW  = red_color(R,G,B)
%createMask  Threshold RGB image using auto-generated code from colorThresholder app.
%  [BW,MASKEDRGBIMAGE] = createMask(RGB) thresholds image RGB using
%  auto-generated code from the colorThresholder app. The colorspace and
%  range for each channel of the colorspace were set within the app. The
%  segmentation mask is returned in BW, and a composite of the mask and
%  original RGB images is returned in maskedRGBImage.

% Auto-generated by colorThresholder app on 05-Feb-2025
%------------------------------------------------------


% Convert RGB image to chosen color space
I = R,G,B;

% Define thresholds for channel 1 based on histogram settings
channel1Min = 171.000;
channel1Max = 255.000;

% Define thresholds for channel 2 based on histogram settings
channel2Min = 83.000;
channel2Max = 127.000;

% Define thresholds for channel 3 based on histogram settings
channel3Min = 77.000;
channel3Max = 153.000;

% Create mask based on chosen histogram thresholds
sliderBW = (R >= channel1Min ) & (R <= channel1Max) & ...
    (G >= channel2Min ) & (G <= channel2Max) & ...
    (B >= channel3Min ) & (B <= channel3Max);
BW = sliderBW;

end
```

Fig.5. Code inside the function block auto-generated code from colorThresholder app for red color

To ensure robustness in the presence of small-scale noise or inconsistencies in color detection, the binary image is passed through a **Median Filter**. This filter smoothens the mask by removing salt-and-pepper noise while preserving the larger features of the segmented line. After this, the processed binary image is routed to a **Video Viewer** block, which displays the detection results in real time during both simulations and live flight. Another parallel Video Viewer is connected to the original RGB stream, allowing the operator to visually verify the accuracy of the segmentation.

The next critical phase in the logic is the spatial analysis of the image using **Submatrix blocks**. The filtered binary image is divided into six different regions of interest:

- **Up_left**, **Up_right**, and **Center** – These are smaller submatrices extracted from the upper part of the image. They help determine the relative position of the line in the drone's field of view.
- **Up_all**, **Left_all**, and **Right_all** – These cover broader sections of the image used specifically for landing logic.

Each of these submatrices is passed through a **Matrix Sum** block, which computes the total number of white pixels within that region—effectively counting how much of the desired line

color is visible. The resulting sum values are compared against constants using **Compare to Constant** blocks.

The navigation logic is derived from these comparisons as follows:

- **If Up_left > 150 pixels** → The line is to the **left**, so the drone should **yaw left**.
- **If Up_right > 150 pixels** → The line is to the **right**, so the drone should **yaw right**.
- **If Center > 150 pixels** → The line is centered, so the drone should **go straight**.

These Boolean decisions are clearly routed and visualized using **Display blocks** in the model and then passed forward through a **Bus Creator**.

Meanwhile, the **landing logic** is evaluated by analyzing the sum of white pixels in the broader regions:

- **If Up_all, Left_all, and Right_all are each ≤ 50**, it is concluded that there is no visible line ahead or on either side, meaning the drone is likely over the center of the landing target (e.g., a circle).
- These three conditions are combined using an **AND gate**, ensuring that the **Land_it signal is only true when all regions are void of line data**.

The final output of the entire image processing block is a structured Bus called **Vision-based Data**, which contains:

- **Left**: Boolean indicating line on the left.
- **Right**: Boolean indicating line on the right.
- **Center**: Boolean indicating line at the center.
- **Land_it**: Boolean flag to trigger landing logic.

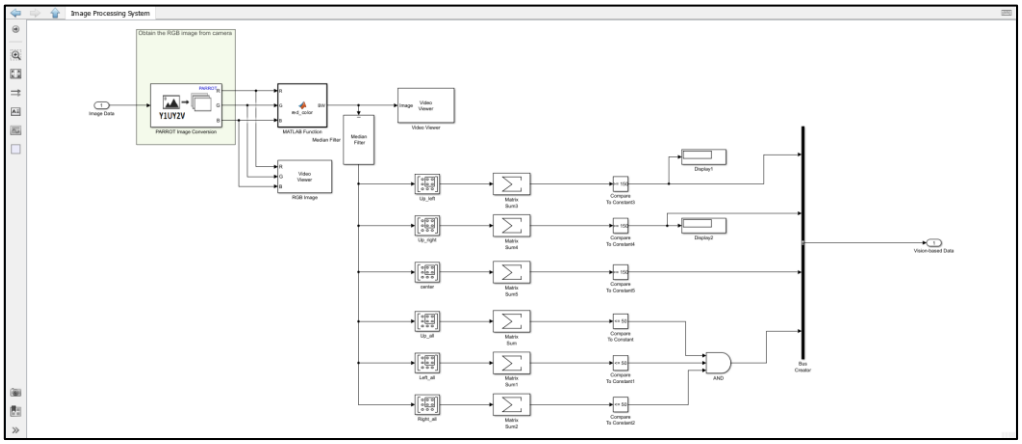

Fig.6. Image Processing Block Subsystems

These outputs are essential to the downstream **Stateflow charts**, where the drone's behavior—such as yawing, moving forward, and initiating landing—is determined. The full subsystem of image processing block is shown in **Figure 6**, which maps the flow from image acquisition, thresholding, and region-based analysis to decision output via vision-based data signals.

3.3. Stateflow Control System

The Stateflow-based control logic acts as the core decision-making and motion control center of the drone's autonomous behavior. This block receives processed outputs from the Image Processing Module, uses internal logic to determine the direction and velocity updates, and then outputs trajectory references that drive the drone's flight actuators. It is here that both **low-level flight behavior (motion calculation)** and **high-level commands (yaw, go straight, or land)** are executed in a structured and dynamic way.

The control system begins by receiving two essential inputs. First, the **EstimatedVal** bus contains the current drone state (position <X>, <Y>, <Z>, and orientation <yaw>, <pitch>, and <roll>) estimated by the onboard state estimator. These values are converted to a double type and routed into the main **Chart block**, which contains the top-level Stateflow logic (Chart 2, discussed next). The second input comes from the **Vision-based Data** bus, which contains four logical signals generated from the Image Processing module: Left, Right, Center, and Land_it. These are unpacked using a **Bus Selector**.

To determine the drone's yawing direction based on line position, a separate **Chart1 block** is employed. This smaller chart takes the Left and Right inputs and outputs a single integer value OUT, representing steering direction. The logic here is simple:

- If Left is true → OUT = -1 (yaw left)
- If Right is true → OUT = 1 (yaw right)
- Otherwise → OUT = 0 (continue straight)

This OUT value is then passed to a comparator and a **Switch block**, which ensures that path updates are only executed when a directional command is actually active (i.e., OUT ≠ 0). When steering is active, the command is multiplied by a small **Gain value (0.0012)**, and integrated over time using a **Unit Delay and Add block**. This simulates smooth yaw angle changes over time instead of instant rotations, helping to prevent abrupt drone movements.

Meanwhile, the primary **Chart** block (shown in tan) is responsible for calculating the updated drone position and control references. It receives the following key signals:

- yawin: updated yaw angle
- R and L: Right and Left line detection flags
- vel: a velocity constant (e.g., 0.00025) that determines speed
- Land_it: logical signal for landing
- C: Center detection flag

Inside this chart, mathematical equations are used to calculate the new X and Y coordinates:

$$dx = vel * \cos(yawin)$$
$$dy = vel * \sin(yawin)$$
$$x_{out} = x_{out} + dx$$
$$y_{out} = y_{out} + dy$$

These updates simulate smooth forward motion or yaw-based corrections depending on the detected line segment.

Three outputs—xout, yout, and zout—are emitted from this chart. These represent the drone's target position in 3D space. Additionally, a logical signal rot is used to handle orientation locking when needed. All of these values are bundled and sent through a **Bus Assignment block**, where:

- pos_ref holds the new position vector (x, y, z)
- orient_ref contains updated yaw or rotation command

The full output bus is labeled **UpdatedReferenceCmds**, which is finally routed to the drone's Flight Control System to drive its motors and actuators in real time.

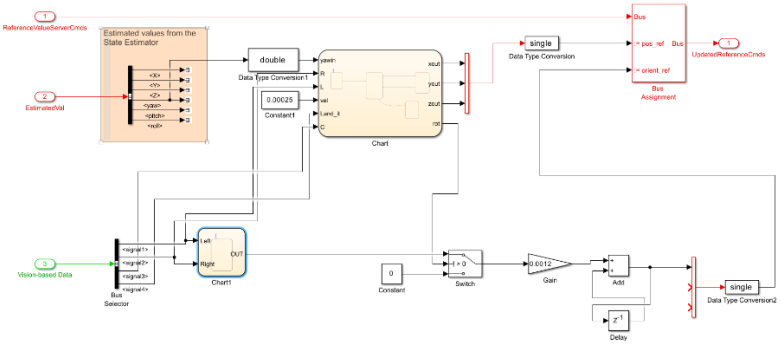

Fig 7. StateFlow Block

This entire control logic, as shown in **Figure 7**, forms the functional backbone of the drone's path-following behavior. It seamlessly integrates directional detection from vision data, position estimation, velocity control, and real-time motion reference updates. The elegance of this approach lies in its modularity—any component, whether image-based or control-based, can be modified or replaced without disturbing the entire architecture.

3.3.1. High-level behavioral State Chart

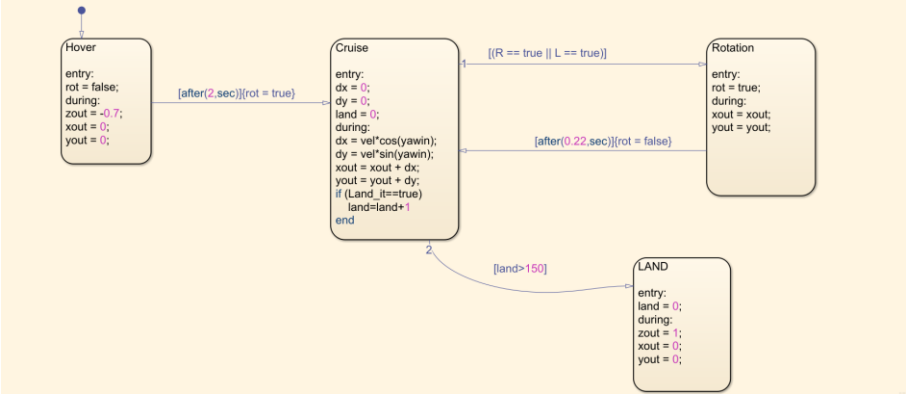

Fig. 8. High-level behavioral State Chart

First chart shown in **Figure 8** represents the high-level behavioral state machine of the Parrot Mambo drone and defines how it transitions between hovering, cruising, rotating, and landing states based on sensory input and timing conditions. This chart is the primary decision unit that transforms the perception data into actionable flight behavior.

The state machine starts in the **Hover** state. This state acts as the initial stabilization phase for the drone immediately after takeoff. During this period, the drone does not move horizontally and maintains a steady altitude with zout = -0.7, a negative value that ensures the drone ascends slowly to a hovering position. Horizontal positions xout and yout remain zero. Upon staying in this state for 2 seconds, the condition [after(2, sec)] is satisfied, triggering a transition to the **Cruise** state. Additionally, as part of this transition, the flag rot is set to true, which indicates that the drone is now permitted to begin motion planning and rotation behaviors.

Once in the **Cruise** state, the drone begins to update its position based on the yaw direction and velocity. In the entry action, variables dx, dy, and land are initialized to zero. Then, during every simulation cycle in the during action, the horizontal motion of the drone is computed using the following mathematical relations:

The values of dx and dy represent the drone's incremental movement in the x and y directions, respectively, based on its yaw orientation (yawin) and velocity (vel). The updated drone positions are then calculated as:

$$dx = vel * \cos(yawin)$$
$$dy = vel * \sin(yawin)$$

These expressions continuously update the output commands sent to the drone's actuators to enable forward motion. Simultaneously, the FSM checks whether the Land_it flag is activated. If true, the counter variable land is incremented (land = land + 1) every time step that Land_it remains active. This accumulation mechanism ensures that a landing is not triggered by a false or brief detection but only after sustained confirmation over multiple cycles.

The transition from **Cruise** to **Rotation** occurs under the condition [(R == true || L == true)], meaning that if a line is detected either on the right or left side, the drone needs to perform a corrective yaw maneuver. When this condition is met, the FSM transitions to the **Rotation** state. Upon entering this state, the rotation flag rot is set to true, indicating that orientation adjustment is underway. However, the drone's position does not change during this phase. Instead, xout and yout maintain their previous values to allow the drone to rotate in place without drifting laterally. After a short duration of **0.22 seconds** in the Rotation state, a timed transition returns the system back to the Cruise state. This brief rotation period is sufficient to reorient the drone toward the detected line direction.

Meanwhile, if the drone remains in the Cruise state and the land counter exceeds 150, the FSM executes a transition to the **LAND** state via the condition [land > 150]. This threshold ensures that landing is only initiated after persistent detection of the landing condition, avoiding premature descent due to transient vision errors.

In the **LAND** state, the drone initiates a controlled descent. The entry action resets the land counter to zero, while in the during action, the drone's vertical output command is set to zout = 1, instructing the drone to slowly descend. The horizontal outputs xout and yout are kept at zero to ensure the drone stays stationary while descending vertically onto the target landing zone. This careful design ensures that landing is stable and precise.

Overall, this FSM structure encapsulates the entire behavior of autonomous flight. It manages how the drone hovers to stabilize, cruises based on path direction, performs quick corrective rotations, and lands gracefully once the target is reached. The logical structure is straightforward, modular, and robust to noise, making it ideal for deployment on lightweight drones like the Parrot Mambo.

3.3.2. Steering Control Logic Chart

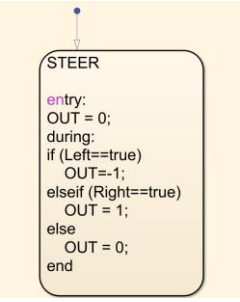

Fig. 9. Steering Control Logic Chart

The **Chart** block shown in Figure 9, named **STEER**, is responsible for interpreting visual inputs about the line's lateral position and converting them into a simple directional command for yaw control. This chart serves as a lightweight decision-making unit that complements the main FSM (Chart 2), by abstracting the steering decision into a single output value: OUT.

Upon entering the STEER state, the chart initializes the output variable OUT to 0, implying no steering command. This initialization occurs in the entry action of the chart and ensures that the drone remains stable unless a directional cue is explicitly detected.

The actual steering decision is made during the during phase of execution. At each cycle, the chart evaluates two Boolean inputs: Left and Right, which are obtained from the Image Processing module via the Vision-based Data bus. These inputs indicate whether a significant amount of the line is detected in the upper-left or upper-right region of the camera's field of view, respectively.

The logic is executed in a top-down order:

- First, it checks whether Left == true. If so, it sets OUT = -1, which corresponds to a command to **yaw left**. This decision has priority because the line being on the left is treated as a stronger cue to turn in that direction before checking right.
- If the line is not detected on the left, the chart proceeds to check if Right == true. If true, OUT = 1 is assigned, signaling a command to **yaw right**.
- If neither Left nor Right is active, then the drone is either aligned or the line is detected at the center. In this case, the system sets OUT = 0, instructing the drone to **continue straight**.

This output value OUT is passed to downstream logic where it is multiplied by a gain to determine the yaw increment and is then integrated over time for smooth motion. The design of this chart ensures responsiveness to real-time vision data while maintaining low computational overhead.

In summary, Chart 1 simplifies path following into a ternary steering decision:

- **OUT = -1** → Yaw Left
- **OUT = 0** → Go Straight
- **OUT = 1** → Yaw Right

The clarity and minimalism of this logic make it ideal for embedded deployment, ensuring that the drone can rapidly respond to line positioning changes detected by the onboard camera.

3.4. Deployment of the Integrated System on the Parrot Mambo Drone

After completing and testing both the image processing and Stateflow control logic in simulation, the final integrated model was deployed to the Parrot Mambo minidrone using Simulink's hardware support package. The target hardware was selected as the Parrot Mambo board under the *Hardware Settings*, and the build process was initiated using the **Build, Deploy & Start** button.

Fig. 10. Path

Upon successful deployment, the drone autonomously took off, entered the hover state for stabilization, and then transitioned to cruise mode. It continuously processed live video through the onboard camera, applied the color thresholding logic, and used submatrix analysis to follow the path shown in the Figure 10. The Stateflow logic generated smooth yaw and movement commands based on line location, and when the line was no longer visible in all three critical regions, the landing sequence was initiated.

This workflow allowed the entire control system to run onboard the drone without requiring a host computer during flight, showcasing the effectiveness of model-based design and real-time deployment using Simulink.

4. RESULTS AND DISCUSSION

The integrated system combining real-time image processing with Stateflow-based flight logic was successfully deployed and tested on the Parrot Mambo minidrone. Through a series of simulations and live flight trials, the drone demonstrated reliable performance in autonomously following a colored path and executing a controlled landing. During testing, the image processing block consistently detected the target color path under stable indoor lighting conditions. The thresholding and submatrix detection logic correctly interpreted whether the line was on the left, right, or center of the field of view. These signals were effectively translated into flight commands through the STEER and Cruise state logic, resulting in responsive yawing behavior and smooth trajectory control.

The steering logic implemented in Chart 1 proved to be both simple and robust. The drone quickly adjusted its yaw direction when the path veered off-center, and returned to straight motion when the line reappeared in the center region. The real-time computation of position updates using velocity and yaw inputs enabled the drone to maintain a consistent cruising behavior without abrupt movements.

The landing logic also functioned as intended. When the line was no longer present in the upper, left, and right segments of the visual field, the Land_it flag was activated. After being sustained for several cycles, this triggered the FSM transition into the LAND state, resulting in a smooth vertical descent. This mechanism reduced the risk of false positives from momentary detection glitches.

Overall, the results validate the effectiveness of combining lightweight image processing techniques with modular Stateflow logic to achieve autonomous behavior in a computationally constrained drone platform. While the system performed reliably under controlled conditions, it was observed that external factors like variable lighting or reflections could affect detection accuracy. This opens opportunities for future enhancements such as adaptive thresholding, dynamic region calibration, or the integration of machine learning-based classifiers to improve robustness.

5. CONCLUSION

This project successfully demonstrated the development and deployment of an autonomous line-following and landing system using the Parrot Mambo minidrone, entirely modeled and implemented in MATLAB Simulink. The approach leveraged a lightweight yet effective image processing pipeline, which performed real-time color thresholding and region-based detection to determine the path direction. These vision-based decisions were integrated with a modular Stateflow control architecture that governed the drone's motion, steering, and landing behavior.

The system achieved smooth path tracking and accurate directional control by combining submatrix logic with trigonometric motion updates. The landing mechanism, triggered by a lack of detected line segments across key visual zones, performed reliably and demonstrated the robustness of the control logic. The entire process—from stabilization and cruise to correction and autonomous landing—ran fully onboard the drone, showcasing the practicality of model-based design for embedded control systems. While the implementation worked well under controlled lighting conditions, challenges such as variable illumination and potential noise in color detection highlight areas for further improvement. Future iterations of this system could benefit from adaptive color segmentation, enhanced region analysis, or deep learning-based object detection models for even greater reliability in diverse environments. In summary, this project presents a replicable, resource-efficient methodology for achieving autonomous drone navigation using vision and control logic—a promising foundation for future work in UAV autonomy and robotics education.

6. REFERENCES

1. MathWorks. (n.d.). *Install Windows Bluetooth Drivers*. Retrieved from https://www.mathworks.com/help/simulink/supportpkg/parrot_ug/install-windows-bluetooth-drivers.html

2. MathWorks. (n.d.). *Getting Started with Simulink Support Package for Parrot Minidrones*. Retrieved from https://www.mathworks.com/help/supportpkg/parrot/ref/getting-started-with-simulink-support-package-for-parrot-minidrones.html

3. MathWorks. (n.d.). *External Mode for Parrot Minidrones*. Retrieved from https://www.mathworks.com/help/supportpkg/parrot/ref/external-mode-for-parrot-minidrones.html

4. MathWorks. (n.d.). *Communicating with Parrot Minidrone Using TCP/IP and UDP*. Retrieved from https://www.mathworks.com/help/supportpkg/parrot/ref/communicating-with-parrot-minidrone-using-tcpip-and-udp.html

7. BIOGRAPHIES OF AUTHOR

Chinmay Amrutkar is a master's student at Arizona State University pursuing Robotics and Autonomous Systems (AI). He has completed his Bachelors of Technology in Robotics and Automation at MIT World Peace University, Pune, India.
He can be contacted via email at camrutka@asu.edu