# Autonomous Path Planning Under Position Uncertainty of a Multi-Rotor Platform with a Scratch Built Flight Controller

Vishal P. Gautam[Δ], Abhishek Jeyaprakas[Δ]

[Δ] Virginia Tech, Blacksburg, Virgina, USA 24060
abhishek1111@vt.edu & vishalgautam@vt.edu

## Abstract

Autonomous aerial vehicles typically employ GNSS and inertial sensors for state estimation, but these are noisy systems by nature and prone to errors such as drift, signal loss, and multipath effects. For this project, we propose a path planning framework that takes into account navigation uncertainty in GNSS/INS-based drones using Rapidly Exploring Random Trees (RRT*) simulations. A specially constructed drone using a Teensy flight controller board and a Raspberry Pi was used to capture real-world IMU readings while flying. This was used to model the state uncertainty of the vehicle as a multivariate Gaussian distribution, from which some samples are drawn to simulate potential deviations from the target path. By examining path deviation probability or collision with obstacles along these samples, we establish a "confidence corridor" and label risky segments on candidate paths. This knowledge is used to adapt or replan flight trajectories before flight so that safer and more robust navigation can be achieved in uncertain environments. This research completes the loop between simulation and physical deployment and offers an operational method for risk-aware autonomy on actual drone systems.

## Introduction

In this project, we present the development of a custom quadrotor platform named LX7-Blackbee, built entirely from scratch with the goal of enabling autonomous navigation through simplified yet robust onboard computation. The flight control system is based on a Teensy microcontroller, interfaced with an MPU6500 IMU for real-time state estimation. Rather than relying on complex, feature-heavy flight stacks such as ArduPilot or PX4, our approach focuses on building a lightweight and adaptable flight controller tailored for experimentation and educational autonomy research.

To offload high-level computation, a Raspberry Pi Zero is integrated onboard as an auxiliary processor, tasked with executing path planning algorithms and managing data logging. A key aspect of our methodology is to model the uncertainty in IMU measurements through in-flight testing and probabilistic error characterization, leveraging data collected during actual flight missions.

Using this empirically derived IMU error model, we applied RRT* algorithm to simulate trajectory deviations and identify risk-prone flight segments. The Raspberry Pi then computes safe, feasible paths using this uncertainty-aware planner. These paths are uploaded to the quadrotor before flight, where the

Teensy flight controller executes them under onboard control, closing the loop between physical flight testing and robust autonomous planning.

This modular setup provides a flexible platform for embedded autonomy research, enabling deeper understanding and customization of navigation, estimation, and control layers without the overhead of large autopilot frameworks.

## Model Details

### Autopilot

The autopilot for LX7-Blackbee was developed by adapting the open-source dRehmFlight codebase to run on a Teensy 4.0 microcontroller. This flight controller manages sensor acquisition, orientation estimation, and low-level stabilization. It uses a Madgwick filter for fusing data from the MPU6500 IMU, providing real-time roll, pitch, and yaw angles. A set of tunable PID controllers govern attitude and rate control, and the motor outputs are computed using a configurable mixer for quadrotor geometry. The system operates at a 2 kHz control loop frequency, ensuring tight feedback for responsive flight dynamics. Input commands are received via RC channels, and custom serial communication was implemented to interface with external computers for telemetry, mission commands, or real-time data logging. The simplicity and modularity of the dRehmFlight architecture allowed for direct insertion of experimental features like IMU error injection, bypassed control inputs for autonomous missions, and live parameter tuning without the overhead of a complex RTOS or middleware.

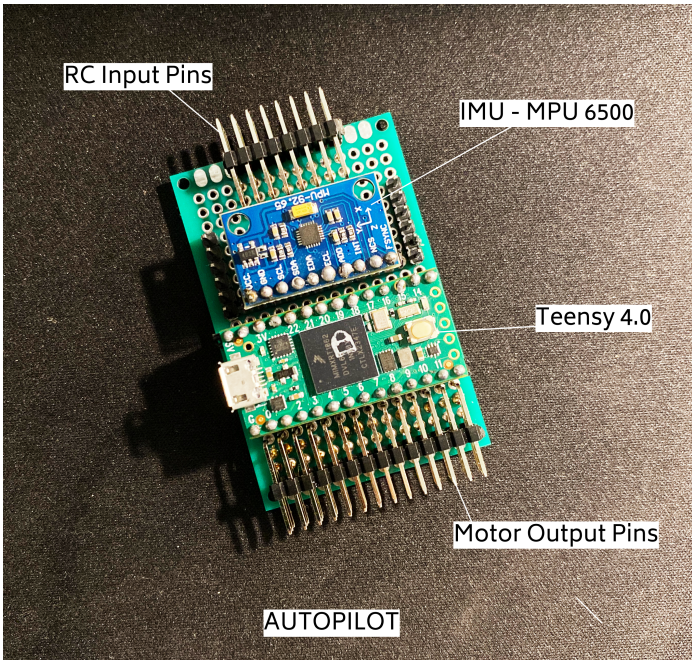



Figure 1: Autopilot

## Quadrotor Platform Overview

The LX7-Blackbee is a custom-built quadrotor platform designed for experimental autonomous flight and embedded path planning. It is structured around a 7-inch carbon fiber X-frame, chosen for its balance between structural stiffness, vibration resistance, and payload capacity. The propulsion system consists of four brushless motors mounted with bi-blade 7-inch propellers, delivering high thrust and efficient flight performance. Power is supplied by a 4-cell LiPo battery, secured at the center of gravity to maintain balance during flight.


Figure 2: Quadrotor Model

At the heart of the system lies a Teensy 4.0 microcontroller, functioning as the low-level flight controller. It runs a customized version of the dRehmFlight firmware and is responsible for sensor fusion, stabilization, and motor actuation. The Teensy interfaces with an MPU6500 IMU, which is mounted on a dedicated vibration isolation system to minimize high-frequency noise affecting orientation estimation.

For high-level tasks, a Raspberry Pi Zero is integrated on board. It acts as the offboard computer for tasks such as IMU data logging, real-time path planning, and autonomous waypoint command generation. Communication between the Pi and Teensy occurs over a serial interface, enabling coordinated control between estimation, planning, and actuation layers.

The LX7-Blackbee combines simplicity in hardware design with powerful onboard computation, making it an ideal testbed for real-time state estimation,

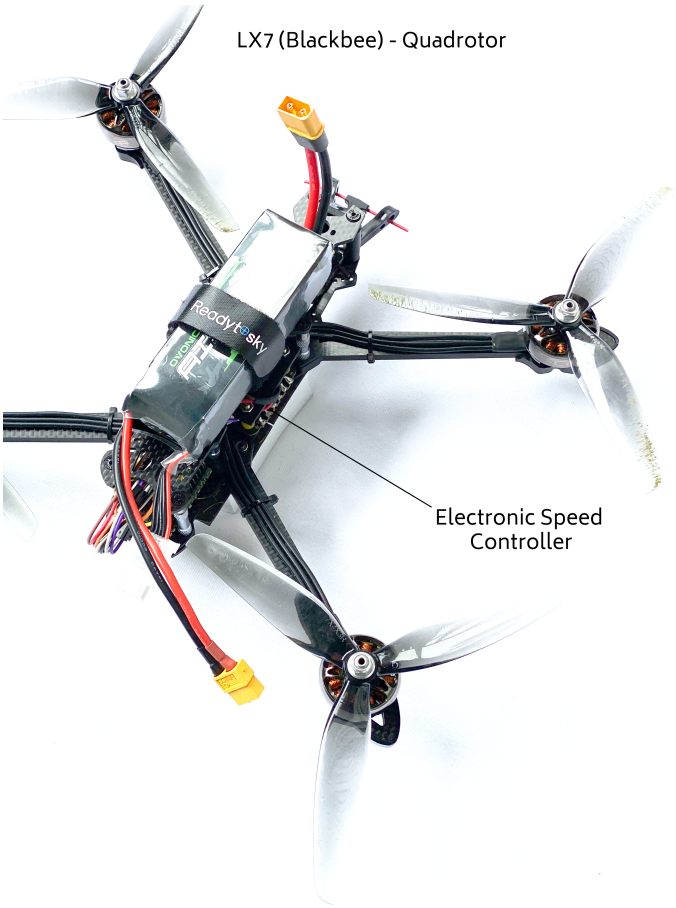



Figure 3: Quadrotor Model

uncertainty modeling, and experimental autonomy pipelines.

## Hardware and Control Code Implementation

| **Hardware** | **Usage / Implementation** |
|---|---|
| Teensy Microcontroller | Runs flight control code, receives motion commands |
| MPU-6050 IMU | Provides acceleration and angular velocity data |
| Raspberry Pi | Runs high-level path planner, logs data, sends commands to Teensy |
| Control Code (on Teensy) | Basic PID controller for heading and motion control |
| Communication | Serial interface between Pi and Teensy for data and commands |

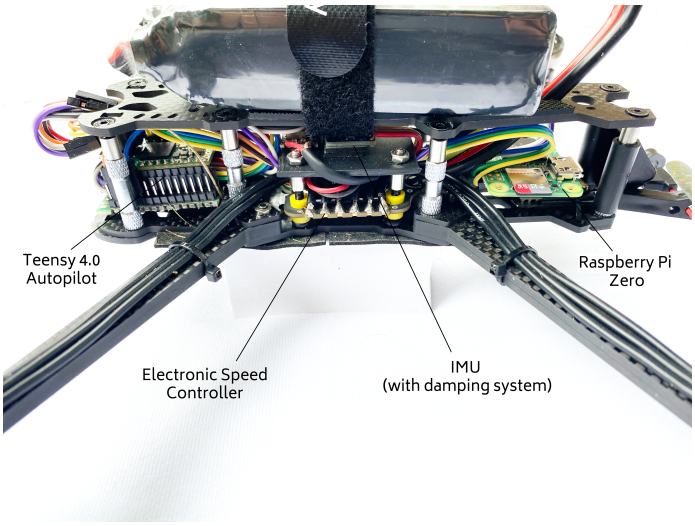



Figure 4: Avionics Layout

## Project Outline

| Phase | Description | Tools/Techniques |
|---|---|---|
| **Data Collection** | Log IMU data (acceleration and angular velocity) from the drone during predefined flight paths. | Teensy (for IMU logging), Raspberry Pi (data storage and transfer) |
| **Measurement Error Modeling** | Analyze IMU logs to model sensor noise, drift, and bias over time. Use this to define a probabilistic uncertainty model. | Statistical analysis (mean, variance), Gaussian modeling |
| **Algorithm Design** | Design a Rapidly Exploring Random Trees (RRT*) risk evaluation framework. Simulate possible trajectory deviations using noise models. | Uncertainty propagation via particle sampling; IMU kinematics model |
| **Algorithm Implementation** | Implement the path planner and particle-based simulator to evaluate path safety and confidence corridors. | MATLAB; 3D map simulation; obstacle checks; particle filter logic |
| **Algorithm Analysis** | Evaluate risk metrics (e.g., collision probability, deviation spread). Visualize particle cloud, error ellipses, and safe regions. | Risk plots, confidence visualizations, path overlay analysis |
| **Experimental Testing** | Compare real drone motion using IMU logs with RRT* predictions. Assess model accuracy and system robustness. | Path deviation analysis; real vs. simulated overlay comparison; drift evaluation |

# Quadrotor Dynamics Modelling

## Nonlinear State-Space Dynamic Model of the Quadrotor

### State and Control Vectors

The state vector $x \in \mathbb{R}^{12}$ is defined as:

$$x = \begin{bmatrix} x & y & z & v_x & v_y & v_z & \phi & \theta & \psi & p & q & r \end{bmatrix}^T \tag{1}$$

where:

- $(x, y, z)$: Position in the inertial frame
- $(v_x, v_y, v_z)$: Linear velocities in the inertial frame
- $(\phi, \theta, \psi)$: Roll, pitch, and yaw angles (Euler angles)
- $(p, q, r)$: Angular velocities in the body frame

The control input vector $u \in \mathbb{R}^4$ is:

$$u = \begin{bmatrix} u_1 & \tau_x & \tau_y & \tau_z \end{bmatrix}^T \tag{2}$$

where:

- $u_1$: Total thrust force
- $\tau_x, \tau_y, \tau_z$: Torques about body axes

### State-Space Equations

The nonlinear dynamics $\dot{x} = f(x, u)$ are:

### Translational Dynamics

$$\dot{x} = v_x, \quad \dot{y} = v_y, \quad \dot{z} = v_z \tag{3}$$

$$\begin{bmatrix} \dot{v}_x \\ \dot{v}_y \\ \dot{v}_z \end{bmatrix} = \frac{1}{m} R(\phi, \theta, \psi) \begin{bmatrix} 0 \\ 0 \\ -u_1 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ g \end{bmatrix} \tag{4}$$

The rotation matrix $R$ (body-to-inertial, ZYX Euler angles) is:

$$R = \begin{bmatrix} \cos\theta\cos\psi & \sin\phi\sin\theta\cos\psi - \cos\phi\sin\psi & \cos\phi\sin\theta\cos\psi + \sin\phi\sin\psi \\ \cos\theta\sin\psi & \sin\phi\sin\theta\sin\psi + \cos\phi\cos\psi & \cos\phi\sin\theta\sin\psi - \sin\phi\cos\psi \\ -\sin\theta & \sin\phi\cos\theta & \cos\phi\cos\theta \end{bmatrix} \tag{5}$$

**Rotational Kinematics**

$$\begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} = T(\phi, \theta) \begin{bmatrix} p \\ q \\ r \end{bmatrix} \tag{6}$$

where $T(\phi, \theta)$ is:

$$T = \begin{bmatrix} 1 & \sin\phi \tan\theta & \cos\phi \tan\theta \\ 0 & \cos\phi & -\sin\phi \\ 0 & \frac{\sin\phi}{\cos\theta} & \frac{\cos\phi}{\cos\theta} \end{bmatrix} \tag{7}$$

**Rotational Dynamics**

Euler's rotational dynamics:

$$\begin{bmatrix} \dot{p} \\ \dot{q} \\ \dot{r} \end{bmatrix} = I^{-1} \left( \begin{bmatrix} \tau_x \\ \tau_y \\ \tau_z \end{bmatrix} - \omega \times (I\omega) \right), \quad \omega = \begin{bmatrix} p \\ q \\ r \end{bmatrix} \tag{8}$$

Expanding the cross product:

$$\omega \times (I\omega) = \begin{bmatrix} (I_{zz} - I_{yy})qr \\ (I_{xx} - I_{zz})pr \\ (I_{yy} - I_{xx})pq \end{bmatrix} \tag{9}$$

**Jacobian Matrices for EKF:**

**State Jacobian**

$$F = fx \in \mathbb{R}^{12\times 12} \tag{10}$$

$$F = \begin{bmatrix} \mathbf{0}_{3\times3} & I_{3\times3} & \mathbf{0}_{3\times6} & \\ F_{vp} & \mathbf{0}_{3\times3} & F_{vR} & \mathbf{0}_{3\times3} \\ \mathbf{0}_{3\times6} & F_{\omega R} & F_{\omega\omega} & \\ \mathbf{0}_{3\times6} & \mathbf{0}_{3\times3} & F_{\alpha\omega} & \end{bmatrix} \tag{11}$$

**Control Jacobian**

$$G = fu \in \mathbb{R}^{12\times4} \tag{12}$$

$$G = \begin{bmatrix} \mathbf{0}_{3\times4} & \\ G_{vu} & \mathbf{0}_{3\times3} \\ \mathbf{0}_{3\times4} & \\ \mathbf{0}_{3\times1} & I^{-1} \end{bmatrix} \tag{13}$$

**Measurement Model:**

Measurement Vector

$$z = h(x) = \begin{bmatrix} R^T(a_{\text{world}} - g) \\ p \\ q \\ r \end{bmatrix} \tag{14}$$

where:

- $a_{\text{world}}$: Linear acceleration in the inertial frame
- $g = \begin{bmatrix} 0 & 0 & g \end{bmatrix}^T$: Gravity vector
- $R^T$: Transpose of rotation matrix, maps inertial to body frame

**Accelerometer Measurement Model**

$$\vec{y}_a = \vec{f}_b + \vec{b}_a + \vec{n}_a \tag{15}$$

In matrix form, for a 3-axis accelerometer:

$$\vec{y}_a = \begin{bmatrix} y_{a_x} \\ y_{a_y} \\ y_{a_z} \end{bmatrix} = \begin{bmatrix} f_{b_x} \\ f_{b_y} \\ f_{b_z} \end{bmatrix} + \begin{bmatrix} b_{a_x} \\ b_{a_y} \\ b_{a_z} \end{bmatrix} + \begin{bmatrix} n_{a_x} \\ n_{a_y} \\ n_{a_z} \end{bmatrix} \tag{16}$$

**Gyroscope Measurement Model**

$$\vec{y}_g = \vec{\omega}_{ib}^b + \vec{b}_g + \vec{n}_g \tag{17}$$

In matrix form, for a 3-axis gyroscope:

$$\vec{y}_g = \begin{bmatrix} y_{g_x} \\ y_{g_y} \\ y_{g_z} \end{bmatrix} = \begin{bmatrix} \omega_x \\ \omega_y \\ \omega_z \end{bmatrix} + \begin{bmatrix} b_{g_x} \\ b_{g_y} \\ b_{g_z} \end{bmatrix} + \begin{bmatrix} n_{g_x} \\ n_{g_y} \\ n_{g_z} \end{bmatrix} \tag{18}$$

We represent the quadrotor's state using a linearized state-space model:

$$\dot{x} = Ax + Bu + w, \quad w \sim \mathcal{N}(0, Q) \tag{19}$$

where $w$ is the process noise.

Position measurements are modeled as:

$$x_{\text{measured}} = x_{\text{true}} + \epsilon, \quad \epsilon \sim \mathcal{N}(0, \Sigma) \tag{20}$$

## Extended Kalman Filter (EKF) for State Estimation

The Extended Kalman Filter (EKF) is a recursive estimator used for nonlinear systems, where the process and measurement models are nonlinear but can be linearized about the current state estimate. The EKF operates in two main phases: **prediction** and **update (correction)**. We define the nonlinear system in discrete time as follows:

$$\mathbf{x}_{k+1} = f(\mathbf{x}_k, \mathbf{u}_k) + \mathbf{w}_k \tag{21}$$

$$\mathbf{z}_k = h(\mathbf{x}_k) + \mathbf{v}_k \tag{22}$$

where:

- $\mathbf{x}_k \in \mathbb{R}^n$: system state vector at time step $k$
- $\mathbf{u}_k \in \mathbb{R}^m$: control input vector
- $\mathbf{z}_k \in \mathbb{R}^p$: measurement vector
- $f(\cdot)$: nonlinear state transition function
- $h(\cdot)$: nonlinear measurement function
- $\mathbf{w}_k \sim \mathcal{N}(0, \mathbf{Q}_k)$: process noise
- $\mathbf{v}_k \sim \mathcal{N}(0, \mathbf{R}_k)$: measurement noise

### Prediction Step

Using the current state estimate $\hat{\mathbf{x}}_k$, we predict the next state and covariance:

**State Prediction**

$$\hat{\mathbf{x}}_{k+1|k} = f(\hat{\mathbf{x}}_k, \mathbf{u}_k) \tag{23}$$

**Jacobian of Dynamics**

$$\mathbf{F}_k = \left.\frac{\partial f}{\partial \mathbf{x}}\right|_{\hat{\mathbf{x}}_k, \mathbf{u}_k} \tag{24}$$

**Covariance Prediction**

$$\mathbf{P}_{k+1|k} = \mathbf{F}_k \mathbf{P}_k \mathbf{F}_k^T + \mathbf{Q}_k \tag{25}$$

### Update Step

When a new measurement $\mathbf{z}_k$ becomes available, the prediction is corrected.

**Measurement Prediction**

$$\hat{\mathbf{z}}_k = h(\hat{\mathbf{x}}_{k|k-1}) \tag{26}$$

**Jacobian of Measurement Model**

$$\mathbf{H}_k = \left.\frac{\partial h}{\partial \mathbf{x}}\right|_{\hat{\mathbf{x}}_{k|k-1}} \tag{27}$$

**Innovation (Residual)**

$$\mathbf{r}_k = \mathbf{z}_k - \hat{\mathbf{z}}_k \tag{28}$$

**Innovation Covariance**

$$\mathbf{S}_k = \mathbf{H}_k \mathbf{P}_{k|k-1} \mathbf{H}_k^T + \mathbf{R}_k \tag{29}$$

**Kalman Gain**

$$\mathbf{K}_k = \mathbf{P}_{k|k-1} \mathbf{H}_k^T \mathbf{S}_k^{-1} \tag{30}$$

**State Update**

$$\hat{\mathbf{x}}_{k|k} = \hat{\mathbf{x}}_{k|k-1} + \mathbf{K}_k \mathbf{r}_k \tag{31}$$

**Covariance Update (Joseph Form)** For improved numerical stability, the Joseph form of the covariance update is used:

$$\mathbf{P}_{k|k} = (\mathbf{I} - \mathbf{K}_k \mathbf{H}_k) \mathbf{P}_{k|k-1} (\mathbf{I} - \mathbf{K}_k \mathbf{H}_k)^T + \mathbf{K}_k \mathbf{R}_k \mathbf{K}_k^T \tag{32}$$

This ensures that $\mathbf{P}_{k|k}$ remains symmetric and positive semi-definite.

## Rapidly Exploring Random Tree Star (RRT*) for Path Planning

RRT, or Rapidly-exploring Random Tree Star, is an optimal path planning algorithm that improves on the basic RRT (rapidly exploring random tree) algorithm by performing path refinement over time. RRT grows a tree from the start point by connecting new nodes, brought in by random samples, to the nearest existing node provided that the path is collision free. In contrast to RRT, RRT* improves solutions by updating the nearby nodes to reduce the total cost of the path traversed. It is asymptotically optimal and probabilistically complete, meaning it will converge to the best path given enough samples.

### Map and Obstacle Definitions

- Workspace dimension: $\mathbb{R}^3$ with size $[X, Y, Z] = [100, 100, 100]$ meters.
- Start position: $\mathbf{q}_{\text{start}} = [x_s, y_s, z_s]^\top$
- Goal position: $\mathbf{q}_{\text{goal}} = [x_g, y_g, z_g]^\top$
- Hover height: $z = 10$ meters.
- Obstacles: $N = 7$ vertical cylinders, each defined by:

$$\mathcal{O}_i = \left\{ (x, y, z) \in \mathbb{R}^3 \mid \sqrt{(x - x_i)^2 + (y - y_i)^2} \leq r_i \right\}$$

### Sampling

A random sample $\mathbf{q}_{\text{rand}} \in \mathbb{R}^3$ is generated using uniform sampling:

$$\mathbf{q}_{\text{rand}} = \begin{cases} \mathbf{q}_{\text{goal}}, & \text{with probability } p_{\text{goal}} \\ [x, y, z]^\top, & x, y \sim \mathcal{U}(0, X),\ z = h \end{cases}$$

where $h = 10$ m is the hover height.

### Nearest Neighbor and Steering

For a tree with nodes $\mathcal{V}$, the nearest node $\mathbf{q}_{\text{near}}$ to $\mathbf{q}_{\text{rand}}$ is:

$$\mathbf{q}_{\text{near}} = \arg\min_{\mathbf{q}_i \in \mathcal{V}} \|\mathbf{q}_i - \mathbf{q}_{\text{rand}}\|_2$$

The new point is generated by moving a fixed distance $\delta$ towards $\mathbf{q}_{\text{rand}}$:

$$\mathbf{q}_{\text{new}} = \mathbf{q}_{\text{near}} + \delta \cdot \frac{\mathbf{q}_{\text{rand}} - \mathbf{q}_{\text{near}}}{\|\mathbf{q}_{\text{rand}} - \mathbf{q}_{\text{near}}\|}$$

### Collision Detection

- **Point collision:** A point $\mathbf{q}$ is in collision if:

$$\exists \mathcal{O}_i \text{ such that } \sqrt{(x - x_i)^2 + (y - y_i)^2} \leq r_i$$

- **Edge collision:** Discretize the edge between two points into steps of size $\epsilon$:

$$\mathbf{q}(t) = (1 - t)\mathbf{q}_1 + t\mathbf{q}_2, \quad t \in [0, 1]$$

  Check all samples $\mathbf{q}(t_k)$ for collision.

### Cost Function and Rewiring for Optimality

The cost function is used to generate optimal path from Start to Goal.

$$c(\mathbf{q}_i) = c(\mathbf{q}_{\text{parent}}) + \|\mathbf{q}_i - \mathbf{q}_{\text{parent}}\|_2$$

For each neighbor $\mathbf{q}_j$ within a ball of radius $r$:

$$\mathbf{q}_j \in \mathcal{B}(\mathbf{q}_{\text{new}}, r) = \{\mathbf{q}_j \in \mathcal{V} \mid \|\mathbf{q}_j - \mathbf{q}_{\text{new}}\| < r\}$$

Rewire if:

$$c_{\text{new}} + \|\mathbf{q}_j - \mathbf{q}_{\text{new}}\| < c(\mathbf{q}_j)$$

**Trajectory: Vertical ascent:** From $\mathbf{q}_{\text{start}}$ to $[x_s, y_s, h]$, **Planned path:** Output of RRT* from $[x_s, y_s, h]$ to $[x_g, y_g, h]$, **Descent:** From $[x_g, y_g, h]$ to $\mathbf{q}_{\text{goal}}$

### RRT* Algorithm Results and Visualization

3000 samples were used to generate the RRT* simulation in MATLAB. The simulation runs for the whole samples and tries to find the most optimal path from all the generated samples. Blue edges are the actual generated edges. Green edges are the rewired edges which introduces optimality in the path. All sampled nodes are plotted. Edges of the tree are shown. Final path is highlighted in red. Obstacles shown as gray cylinders. Drone animation uses a moving marker along the final trajectory. The offline RRT* generated waypoints are saved which are to be fed in the Raspberry Pi for Autonomous Navigation.

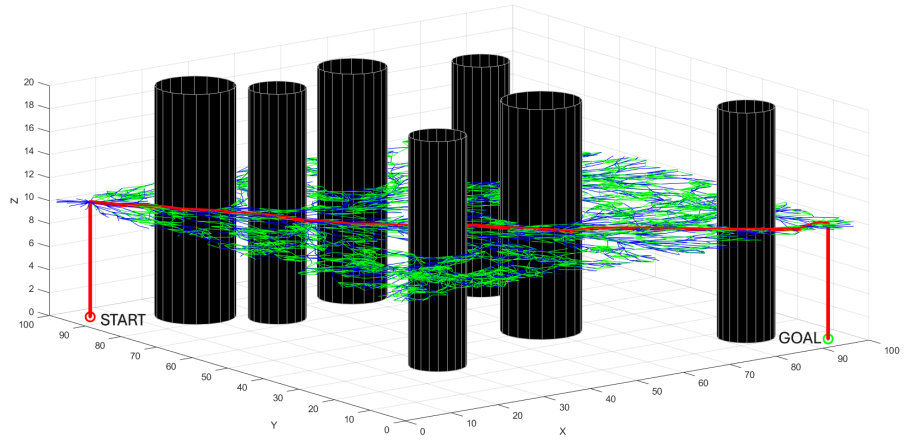



Figure 5: 3D RRT* Path Planning with Obstacles and Collision Avoidance

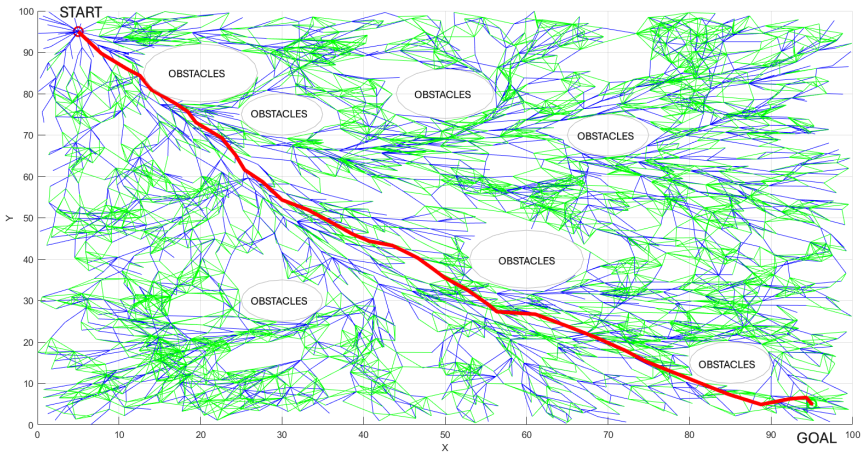



Figure 6: XY Plane View of the RRT* Path Planning

This image depicts the result of an RRT* (Rapidly-exploring Random Tree Star) path planning algorithm operating in a space populated with multiple elliptical obstacles. The environment spans from (0,0) to (100,100), with the START point in the top-left corner and the GOAL in the bottom-right. The algorithm begins by randomly sampling the configuration space and incrementally building a tree structure, represented here by the thin blue lines, which show the initial connections between nodes. As the tree grows, RRT* performs rewiring steps to improve path quality, and these rewired connections are shown in green. This process allows the algorithm to replace longer paths with shorter, more efficient ones whenever possible, while still avoiding the elliptical OBSTACLES shown in light gray. The final optimal path from START to GOAL is highlighted in a thick red line, showcasing the refined, lowest-cost route found after tree growth and rewiring. The presence of dense blue and green branches around obstacles indicates active exploration and optimization, resulting in a smooth and efficient path that safely navigates the cluttered environment.

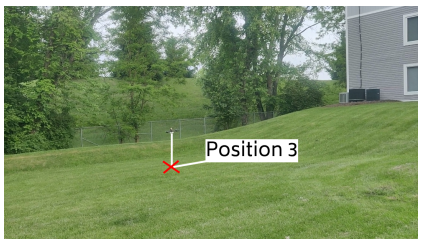
# Implementation and Results

### IMU Data Collection and Error Modeling

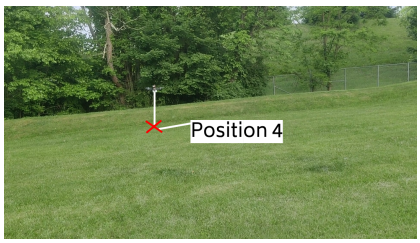
To assess the behavior and drift characteristics of the onboard MPU6500 IMU, we conducted a series of structured flight tests with the LX7-Blackbee quadcopter. The drone was flown in a closed-loop rectangular path through four designated positions in an open field (as shown in the Figure 5(a) to 5(d). At each leg of the path, the Raspberry Pi Zero interfaced with the Teensy-based flight controller to log real-time IMU data, including acceleration and angular rate measurements.


(a) Waypoint 1


(b) Waypoint 2




(c) Waypoint 3




(d) Waypoint 4

Figure 7: Flight test snapshots during IMU error modeling.

The objective of this exercise was to gather raw sensor output over repeatable motion profiles, which could then be used to identify systematic errors such as bias drift, scale factor inconsistencies, and integration noise. These recorded datasets form the basis for developing an error model, which we plan to integrate back into the autonomous flight system to improve state estimation accuracy during waypoint navigation.

The IMU measurements are modeled as the sum of the true signal, sensor bias, and measurement noise.

### Accelerometer Model

$$\mathbf{a}_{\text{meas}}(t) = \mathbf{f}_{\text{true}}(t) + \mathbf{b}_a(t) + \mathbf{n}_a(t) \tag{33}$$

- $\mathbf{a}_{\text{meas}}(t)$: measured acceleration (sensor output)
- $\mathbf{f}_{\text{true}}(t)$: true specific force (gravity-compensated)
- $\mathbf{b}_a(t)$: time-varying accelerometer bias

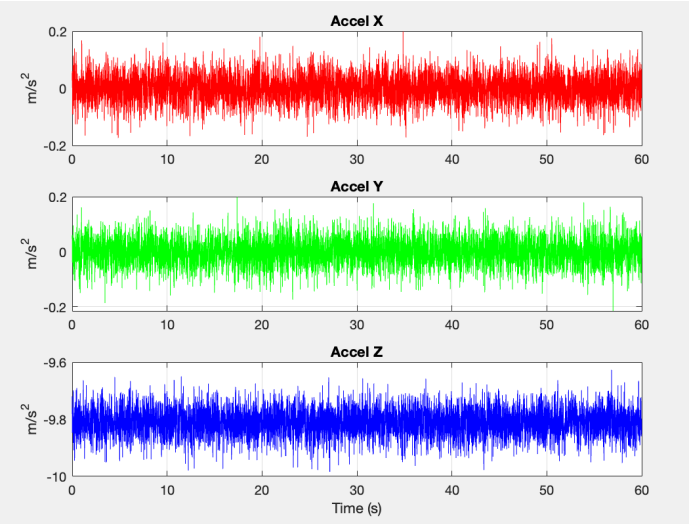



Figure 8: Accelerometer Data from Flight

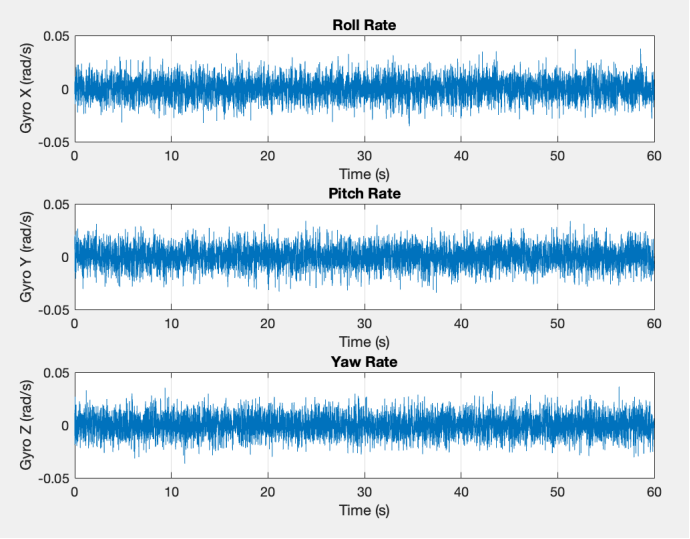



Figure 9: Gyro Data from Flight

- $\mathbf{n}_a(t)$: white Gaussian noise (zero-mean)

The accelerometer bias evolves over time as a random walk:

$$\dot{\mathbf{b}}_a(t) = \mathbf{w}_a(t), \quad \mathbf{w}_a(t) \sim \mathcal{N}(0, \sigma_{ba}^2 \mathbf{I}) \tag{34}$$

**Gyroscope Model**

$$\boldsymbol{\omega}_{\text{meas}}(t) = \boldsymbol{\omega}_{\text{true}}(t) + \mathbf{b}_g(t) + \mathbf{n}_g(t) \tag{35}$$

- $\boldsymbol{\omega}_{\text{meas}}(t)$: measured angular rate

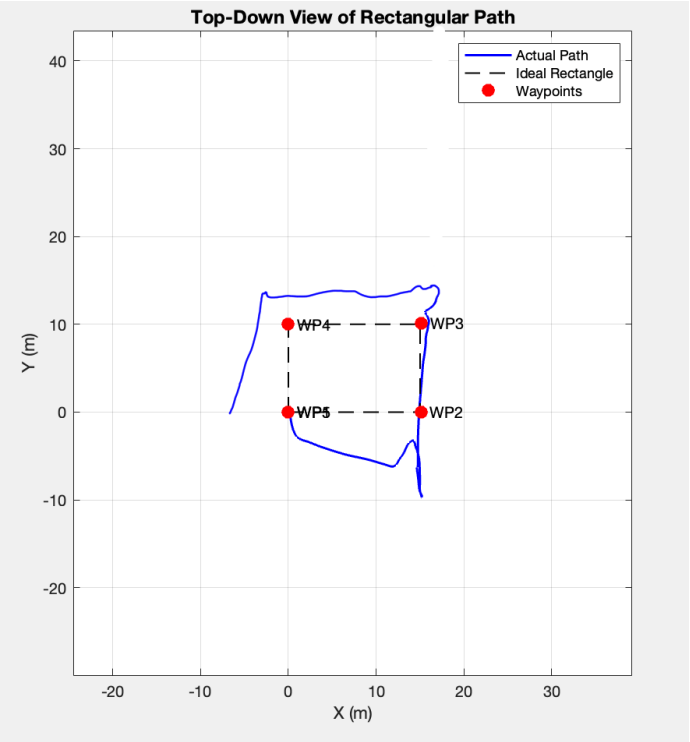



Figure 10: Position Estimation from Accelerometer Values

- $\boldsymbol{\omega}_{\text{true}}(t)$: true angular velocity
- $\mathbf{b}_g(t)$: time-varying gyroscope bias
- $\mathbf{n}_g(t)$: white Gaussian noise

The gyroscope bias also follows a random walk:

$$\dot{\mathbf{b}}_g(t) = \mathbf{w}_g(t), \quad \mathbf{w}_g(t) \sim \mathcal{N}(0, \sigma_{bg}^2 \mathbf{I}) \tag{36}$$

### Discrete-Time Approximation

In discrete time with sampling interval $\Delta t$, the bias can be updated as:

$$\mathbf{b}_a[k+1] = \mathbf{b}_a[k] + \mathbf{w}_a[k]\sqrt{\Delta t} \tag{37}$$

$$\mathbf{b}_g[k+1] = \mathbf{b}_g[k] + \mathbf{w}_g[k]\sqrt{\Delta t} \tag{38}$$

### Implementation of Error Model to Correct the Trajectory

To account for real-world IMU imperfections, we implemented an error model that simulates sensor bias and noise. The bias drifts slowly over time, while random fluctuations represent measurement noise. This model helps mimic how actual IMU data behaves during flight and provides a basis for developing correction methods. Figure 11 shows the corrected trajectory.

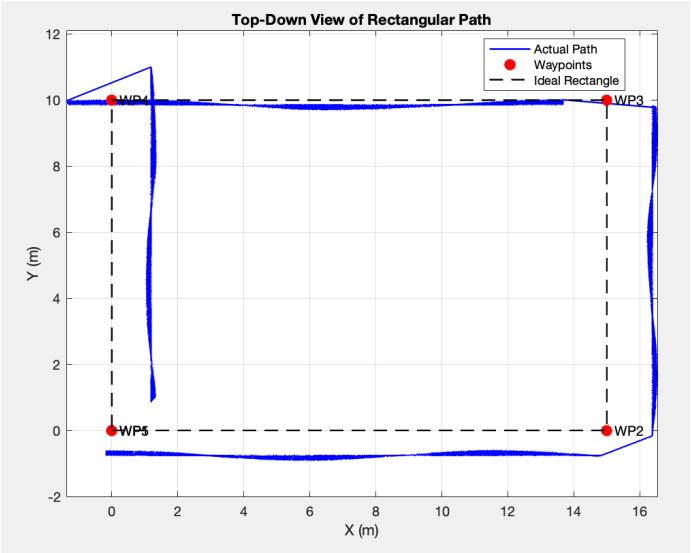



Figure 11: Corrected Path from Error Model

## Raspberry Pi based Waypoint Mission Using Generated Path Planned Coordinates

A waypoint mission was executed on a Raspberry Pi using coordinates generated by an RRT-based path planner. Given two predefined endpoints, the planner computed a feasible trajectory offline, which was then followed autonomously onboard the Raspberry Pi. To replicate realistic onboard sensing conditions, the IMU data was modeled with bias and drift. A simple correction method was applied to reduce the effect of these errors and keep the drone aligned with the intended trajectory—without the use of GPS.

Although the system conceptually demonstrated autonomous path following, the mission was not fully successful in practice. Limitations in the IMU error modeling, lack of flight test iterations, and the partial implementation of correction logic prevented the drone from strictly adhering to the path as planned.

## Uncertainty Quantification for Post Flight Analysis

In order to assess the estimator's accuracy against the anticipated RRT* trajectory, we performed a combined covariance and Monte Carlo analysis at four selected time intervals for the quadrotor flight. With the aid of an Extended Kalman Filter (EKF), we propagated state covariance utilizing the IMU-derived measurement model, visualizing the $1\sigma$ (68% confidence) uncertainty ellipses of position estimation. Concurrently, Monte Carlo simulations were conducted where the IMU data was perturbed based on the empirically determined sensor noise model. This approach yielded a distribution of estimated positions relative to each time frame for four estimates across the planned path, which was subsequently evaluated against the ground truth RRT* waypoint. The covariance ellipses with the Monte Carlo scatter generated estimative analytical bounds as

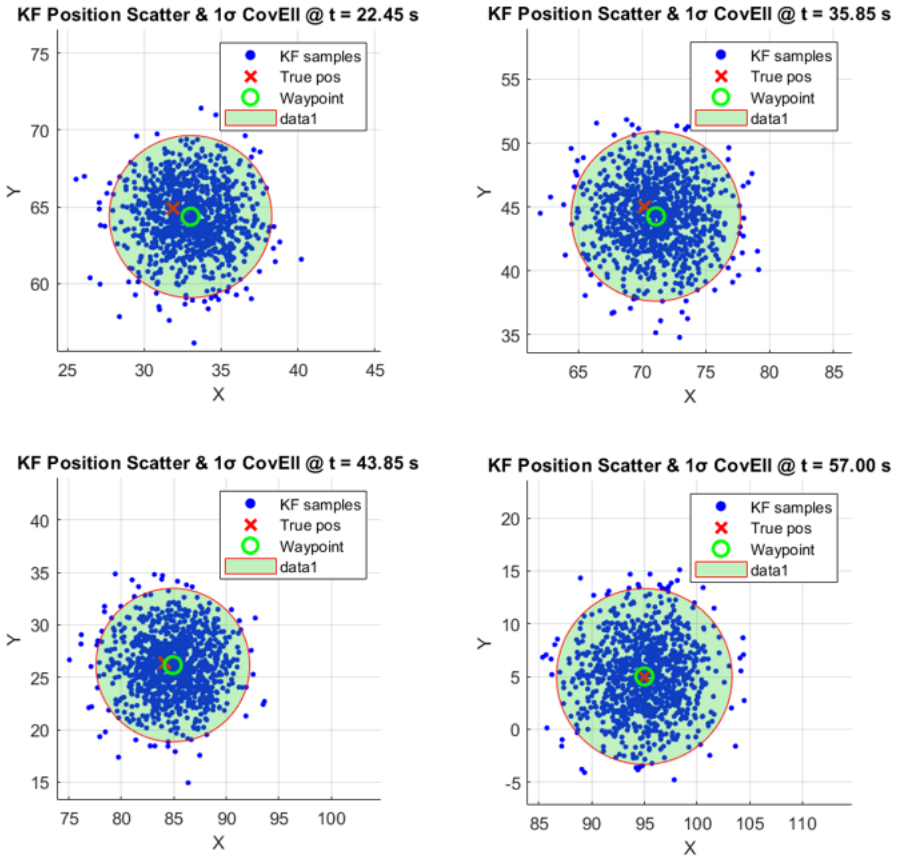



Figure 12: Statistical Analysis using Monte Carlo Simulation

well as empirical bounds on the estimator's performance. From the combination of the true waypoint, EKF estimate, and scatter, alignement allowed confidence levels to be assessed and estimation uncertainty tracked over time in relation to the planned path.

# Conclusion

The LX7-Blackbee project showcases the development of a fully custom quadrotor platform, with a focus on lightweight, onboard autonomy built around a Teensy microcontroller and MPU6500 IMU. A key experimental procedure involved flying the quadrotor in a controlled rectangular trajectory while collecting raw IMU data. This data was processed through double integration to estimate position, revealing drift and error characteristics inherent in inertial-only navigation. By systematically modeling these IMU measurement errors—particularly accelerometer bias, noise, and integration drift—we were able to construct a probabilistic error model that captures how position estimates degrade over time without correction.

This empirically derived model was then integrated with the quadrotor's

flight dynamics to simulate the impact of inertial uncertainty on path following. When applied to RRT*-planned trajectories, the model enabled us to assess how sensor error could cause deviations from planned paths, helping identify risk-prone segments in advance. The Raspberry Pi Zero, acting as an auxiliary processor, used this uncertainty-aware planner to compute safer paths that account for potential deviations due to IMU error. These refined paths were executed by the Teensy flight controller in real flight tests, creating a closed-loop system where real-world sensor behavior informs more robust and realistic planning. Overall, the project successfully demonstrates how raw sensor data, flight dynamics, and planning algorithms can be tightly integrated to improve the reliability of autonomous navigation in embedded platforms. The actual flight of the Blackbee could not exactly mimic the path produced by the RRT* trajectory due to errors in the error modeling. Which could have been reduced with more flight testing and a better sophisticated implementation algorithms.

## References


- LaValle, S. M. (2006). *Planning Algorithms*. Cambridge University Press.
- Karaman, S., Frazzoli, E. (2011). Sampling-based algorithms for optimal motion planning. *International Journal of Robotics Research*, 30(7), 846–894. [RRT*]
- Titterton, D. H., Weston, J. L. (2004). *Strapdown Inertial Navigation Technology*. IET.
- Grewal, M. S., Weill, L. R., Andrews, A. P. (2007). *Global Positioning Systems, Inertial Navigation, and Integration*. Wiley-Interscience.
- Mourikis, A. I., Roumeliotis, S. I. (2007). A multi-state constraint Kalman filter for vision-aided inertial navigation. *IEEE ICRA*.
- Mahony, R., Hamel, T., Pflimlin, J. M. (2008). Nonlinear complementary filters on the special orthogonal group. *IEEE Transactions on Automatic Control*, 53(5), 1203–1218.
- Meier, L. et al. (2015). PX4: A node-based multithreaded open source robotics framework for deeply embedded platforms. *IEEE ICRA*.


## AI Acknowledgement


AI was used for general guidance and clarification of concepts. All experimentations, critical decisions, analysis, and final implementations were made by the authors.