# AAE 568 Project Proposal: Trajectory Optimization with Obstacle Avoidance for Unmanned Aerial Vehicles

Atharva Gunda, Hsin-Ying Lin, Falak Mandali
*School of Aeronautics and Astronautics, Purdue University, West Lafayette, USA*
algunda@purdue.edu , lin1783@purdue.edu , fmandali@purdue.edu

**Abstract**

The objective of this project is to optimize the path of a UAV to reach its destination using the shortest path. The UAV detects the environment in real-time to identify any obstacles, static or dynamics, and modify its future trajectory to maintain a safe distance from the obstacles at all times. A nonlinear model predictive controller is used to optimize the control input to the UAV to control its lateral and longitudinal motion. A satellite is used to provide the UAV with location data. An extended Kalman filter aboard the UAV is used to predict the next state of the UAV and correct its estimate using the satellite data. The successful control and estimation of the UAV are displayed in three different environments: no obstacle, static obstacle, and dynamic obstacle.

## I. Introduction

Unmanned aerial vehicles (UAVs) have gained popularity for their wide range of applications, such as rescue missions, package deliveries, and military purposes. Path planning plays an important role in such applications, typically categorized into global path planning and local path planning [1] [2]. For global path planning, obstacles are known and static. Some commonly used algorithms include the A* and RRT* algorithms. However, real scenarios are usually more complicated, and the environment is often unknown and dynamic. For instance, there may be other moving vehicles and pedestrians acting as dynamic obstacles in urban cities. In these cases, local path planning would be ideal as it is implemented in an online fashion. In the scope of this project, we aim to perform real-time trajectory optimization for UAVs, which can be considered as local path planning.

We can optimize our path and control our drone in real-time using control methods such as *model predictive control* (MPC). There are two types of MPC formulations: linear MPC, and nonlinear MPC. Both types of MPC perform an optimization at every time step to minimize a specified cost function. This optimization takes into account predicted future trajectories of the system and provides the optimal control input to apply until the next time step. This type of controller appears to be suitable for our application as we need to correct the position of our rotor changes in real time to ensure it does not collide with obstacles while learning about the environment in real time.

There are advantages and disadvantages to implementing either a linear or nonlinear MPC. The use of a linear MPC is encouraged when the dynamics of a system can be modeled as linear equations and when there are fixed linear constraints on the input or output of the system. It can find an optimal input for a global minimum cost in the constrained region whilst being computationally less expensive. However, it may not handle dynamic constraints so additional controllers may be needed to achieve this [3]. The nonlinear MPC, on the other hand, is more applicable to real-world systems as it can consider nonlinear system dynamics, and nonlinear and time-varying constraints on the input and the states, despite being computationally expensive. Additionally, one must be aware that a nonlinear MPC may not always converge to a globally minimal solution. Inspired by the flexibility of nonlinear MPC, the authors of [4] proposed an algorithm that minimizes the distance to the target point in real time while maintaining a safe distance from the obstacles, particularly dynamic obstacles.

It is worth noting that most of the literature on simulation-based target tracking or trajectory optimization assumes that the position of the UAV at each time step is known. However, UAVs need GPS data to navigate in the real

world [5]. While the UAV can also predict its future state, the GPS data sent by satellite may encounter noise due to atmospheric reflections, modulation, and demodulation of the data. The UAV can predict its next state using its dynamics, and an extended Kalman filter (EKF) can be used to correct the estimate by accounting for the error in the predicted and actual locations of the UAV. [6].

The objective of this work is to optimize the trajectory of the UAV in real time so that it takes the shortest path from the initial to the final point. The UAV should be able to correct its course upon detecting obstacles in the environment to maintain a safe distance from them. The paper is organized as follows: II discusses the motion of the UAV modeled using linear dynamics which is later used to show the control of the UAV in an obstacle-free environment; III discusses the motion of the UAV modeled using nonlinear dynamics and nonlinear constraints to measure distance from obstacles; IV discusses the formulation of the EKF to correct the predicted states; VI shows the results of the integrated control and estimation of the UAV.

## II. Linear Model Predictive Control (MPC)

### A. Linear Dynamics

The kinematic motion of the UAV with respect to the ground frame can be modeled using Eqn. (1) - (4), where ($p$) stands for the position of the UA(V, ($v$) stands for the velocity of the UAV, and ($a$) stands for the acceleration of the UAV along the x, y, and z axes measured from the ground frame. The acceleration of a UAV is indirectly controlled by changing the thrust, pitch, and roll of the UAV. Each model can be used to control the UAV in an obstacle-free environment as the only constraints are on the acceleration of the system, which is linear. The quadratic nature of dynamics enables the formulation of the quadratic cost function as shown in Eqn. (5). The objective of the MPC is to find the optimal input ($u^*$) that minimizes the cost function ($J_N$) within the constraint on the input described as in Eqn. (6).

$$\text{state, } x = \begin{bmatrix} p & v \end{bmatrix}^T \tag{1}$$

$$\text{input, } u = \begin{bmatrix} a \end{bmatrix}^T \tag{2}$$

$$\dot{x} = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} x + \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} u \tag{3}$$

$$\text{outupt, } y = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix} x \tag{4}$$

$$\text{cost: } \min J_N(U) = \frac{1}{2} U^T L U + F^T U \tag{5}$$

$$\text{subject to: } \begin{bmatrix} a_x^{min} \\ a_y^{min} \\ a_z^{min} \end{bmatrix} \leq \begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix} \leq \begin{bmatrix} a_x^{max} \\ a_y^{max} \\ a_z^{max} \end{bmatrix} \tag{6}$$

where

$$p = \begin{bmatrix} p_x & p_y & p_z \end{bmatrix}^T$$
$$v = \begin{bmatrix} v_x & v_y & v_z \end{bmatrix}^T$$
$$a = \begin{bmatrix} a_x & a_y & a_z \end{bmatrix}^T$$

## III. Nonlinear Model Predictive Control (NLMPC)

### A. Nonliner Dynamics

A linear MPC can not be used to incorporate nonlinear and dynamic constraints on the state. Hence, a nonlinear MPC is required to accommodate such constraints. The nonlinear dynamics of the UAV used in this work are inspired from the model created by the authors of [7]. The model is split into two body frames, where the thrust ($T$), roll angle ($\phi$), and pitch angle ($\theta$) are calculated in the body frame, but the position ($p$) and velocity ($v$) of the UAV is measured in the inertial frame. The authors of [7] also assume that there is a low-level controller aboard the drone that is maintaining the desired roll ($\phi_{ref}$) and pitch angle ($\theta_{ref}$), and hence the roll rate ($\dot{\phi}$) and pitch rate ($\dot{\theta}$) are modeled as a first order system modeled by the gains ($K_\theta, and K_\phi$) and time constants ($\tau_\theta, and \tau_\phi$) for the controller. The values displayed in Eqn. (10). Finally, the authors of [7] have modeled the inertial resistance of the drone as the damping terms denoted by $\begin{bmatrix} A_x & A_y & A_z \end{bmatrix}$.

$$x = \begin{bmatrix} p & v & \phi & \theta \end{bmatrix}^T \tag{7}$$

$$u = \begin{bmatrix} T & \phi_{ref} & \theta_{ref} \end{bmatrix}^T \tag{8}$$

$$\begin{bmatrix} A_x & A_y & A_z \end{bmatrix} = \begin{bmatrix} 0.1 & 0.1 & 0.2 \end{bmatrix} \tag{9}$$

$$\begin{bmatrix} K_\phi & K_\theta & \tau_\phi & \tau_\phi \end{bmatrix} = \begin{bmatrix} 1 & 1 & 0.5 & 0.5 \end{bmatrix} \tag{10}$$

$$\dot{p} = v \tag{11}$$

$$\dot{v} = \begin{bmatrix} \cos(\theta) & \sin(\theta)\sin(\phi) & \sin(\theta)\cos(\phi) \\ 0 & \cos(\phi) & -\sin(\phi) \\ -\sin(\theta) & 0 & \cos(\theta)\cos(\phi) \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ T \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ -g \end{bmatrix} - \begin{bmatrix} A_x & 0 & 0 \\ 0 & A_y & 0 \\ 0 & 0 & A_z \end{bmatrix} v \tag{12}$$

$$\dot{\phi} = \frac{1}{\tau_\phi}(K_\phi \phi_{ref} - \phi) \tag{13}$$

$$\dot{\theta} = \frac{1}{\tau_\theta}(K_\theta \theta_{ref} - \theta) \tag{14}$$

$$y = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix} x \tag{15}$$

### B. Obstacle Definition and Constraint

In the scope of this report, we model the obstacle as a sphere for simplicity. The obstacle avoidance constraint can then be formulated as an inequality constraint:

$$[(p_x - p_x^{\text{obs}})^2 + (p_y - p_y^{\text{obs}})^2 + (p_z - p_z^{\text{obs}})^2 - (r^{\text{obs}} + r_s)^2] \geq 0 \tag{16}$$

where $\begin{bmatrix} p_x^{\text{obs}} & p_y^{\text{obs}} & p_z^{\text{obs}} \end{bmatrix}^T$ denotes the position of the obstacle, which can vary over time. $r^{obs}$ and $r_s$ represent the radius of the obstacle and the safety radius, respectively. The safety radius serves as an additional buffer zone around the obstacle to ensure safe navigation, especially when the UAV is operating within proximity to the obstacle.

To follow the constraint formulation structure for Python's Optimization Engine (OpEn), we utilize the function $[h]_+ = \max\{0, h\}$ to convert Eqn. (16) to an equality constraint, where h can be expressed as:

$$h = (r^{\text{obs}} + r_s)^2 - (p_x - p_x^{\text{obs}})^2 - (p_y - p_y^{\text{obs}})^2 - (p_z - p_z^{\text{obs}})^2 \tag{17}$$

Then, the equality constraint can be formulated as:

$$[h]_+ = 0 \tag{18}$$

This implies that the equality constraint is satisfied when there is no collision between the UAV and the obstacle.

### C. Problem Formulation

Since our constraint for obstacle avoidance includes the position of the obstacle $\begin{bmatrix} p_x^{\text{obs}} & p_y^{\text{obs}} & p_z^{\text{obs}} \end{bmatrix}^T$, we introduce an extended state so the obstacle avoidance constraint can be formed as a constraint on the state. The extended state is defined as:

$$x_{\text{ext}} = [p_x \quad p_y \quad p_z \quad v_x \quad v_y \quad v_z \quad \phi \quad \theta \quad p_x^{\text{obs}} \quad p_y^{\text{obs}} \quad p_z^{\text{obs}}]^T \tag{19}$$

Let $N$ denote the prediction horizon, and $x_{ext}(k)$ denote the extended state at time step k, then the cost function can be formulated as:

$$J = \sum_{k=0}^{N-1} \left( (x_{\text{ext}}(k) - x_{\text{target}})^T Q (x_{\text{ext}}(k) - x_{\text{target}}) + (x_{\text{ext}}(N) - x_{\text{target}})^T Q_N (x_{\text{ext}}(N) - x_{\text{target}}) \right) \tag{20}$$

where $x_{target} = \begin{bmatrix} p_x^{\text{target}} & p_y^{\text{target}} & p_z^{\text{target}} \end{bmatrix}^T$ is the position of the target. $Q \in \mathbb{R}^{8\times8}$, $Q_N \in \mathbb{R}^{8\times8}$ are positive definite weight matrices of the state for the running cost and final cost.

The optimization problem can then be formulated as:

$$\begin{aligned} \text{Minimize} \quad & J \\ \text{subject to} \quad & x_{ext,k+1} = f(x_{ext,k}, u_k) \\ & u_{\text{min}} \leq u(k) \leq u_{\text{max}} \\ & [h]_+ = 0 \end{aligned} \tag{21}$$

Note that the dynamics prediction $x_{ext,k+1} = f(x_{ext,k}, u_k)$ utilizes a discretized representation of the UAV's system dynamics, with a sampling time of $t_s$, using the Forward Euler discretization method.

**Algorithm 1** Nonlinear Model Predictive Control (NLMPC)

**Input:** Initial state $x_0$, guess of control input $u_0$, horizon length $N$, number of simulation steps $N_{\text{sim}}$
**Output:** Optimal control input $u$
$k \leftarrow 0$
**while** $k < N_{\text{sim}}$ **do**
  **Prediction Phase:**
  Formulate the constrained cost function over horizon $[k, k+N-1]$
  Solve the optimization problem (21) to find $u^* = [u_k^*, u_{k+1}^*, ..., u_{k+N-1}^*]$
  Extract the first control input: $u = u_k^*$
  **Time Step Increment:**
  $k \leftarrow k+1$
**return** $u$
Apply $u$ to the system: $x_{ext,k+1}$ = $f(x_{ext,k}, u)$

Algorithm 1 shows the steps of implementing NLMPC, where the optimization problem (21) is solved by Proximal Averaged Newton-type Method (PANOC) using OpEn. The general optimization problem for OpEn is in the form:

$$\begin{aligned} \text{Minimize} \quad & J(u, x) \\ \text{subject to} \quad & x_{,k+1} = f(x_k, u_k) \\ & u \in U \\ & F_2(u, x) = 0 \end{aligned} \tag{22}$$

The equality constraint $F_2$ is handled by using the penalty method, where the problem is then formulated as $Minimize : J(u,x) + c\|F_2(u,x)\|_2^2$, where $c$ is a positive penalty parameter. For each iteration, the penalty parameter increases and it uses the previous solution as the initial guess. By gradually escalating the penalty parameter, the method systematically drives the optimization towards minimizing the cost while satisfying the constraints within a specified tolerance level.

## IV. Extended Kalman Filter (EKF)

So far, the state of the UAV was assumed to be known. However, in real-life scenarios, the UAV position and velocity data is measured through sensors that tend to transmit data with certain noise [5]. Therefore, an estimation algorithm is necessary to minimize this process noise and optimally estimate the UAV's state at all times [6]. A satellite was chosen to get the location of the UAV in the ground frame and the noise associated with the data was modeled as zero-mean Gaussian. Due to the nonlinear nature of the motion of our UAV discussed in III, an extended Kalman filter (EKF) was used to correct the estimated states. Using the lecture notes as a reference, the following estimation algorithm was formulated:

Let the nonlinear dynamics be defined as follows:

$$\dot{x} = f(x, u, w, t) \tag{23}$$
$$y = h(x, v, t) \tag{24}$$
$$w \leftarrow \mathcal{N}(0, q^2) \tag{25}$$
$$v \leftarrow \mathcal{N}(0, r^2) \tag{26}$$

where, $x$ is the state, $u$ is the input, and $w$ and $v$ are noise.

In order to implement EKF, one can propagate the state using nonlinear dynamics, but one must propagate the error covariance matrix $P_k(+)$ using linearized dynamics only. The error covariance matrix shows the error in observing a state, and one must linearise the system dynamics at each estimated point to propagate to the next.

Linearized dynamics:

$$x_{k+1} = Ax_k + Bu_k + Cw_k \tag{27}$$
$$y = Hx_k + Gv_k \tag{28}$$

$$A = \frac{\partial f}{\partial x}|_{x=\hat{x}}, B = \frac{\partial f}{\partial u}|_{x=\hat{x}}, C = \frac{\partial f}{\partial}|_{x=\hat{x}} \tag{29}$$
$$H = \frac{\partial h}{\partial x}|_{x=\hat{x}}, G = \frac{\partial h}{\partial}|_{x=\hat{x}} \tag{30}$$

To correct the precited state and covariance, one must calculate Kalman gain as follows:

$$L_k = P_k(-)H^T(\hat{x}_k)[H(\hat{x}_k)P_k(-)H^T(\hat{x}_k) + R_k]^{-1} \tag{31}$$

The estimates are corrected as:

$$\hat{x}_k(+) = \hat{x}_k(-) + L_k[y_{sensor} - h(\hat{x}_k)] \tag{32}$$
$$P_k(+) = [I - L_k H(\hat{x}_k)]P_k(-) \tag{33}$$

The corrected state is propagated as:

$$\hat{x}_{k+1}(-) = \hat{x}_k(+) + \int_k^{k+1} f(\hat{x}_k, u_k, t)dt \tag{34}$$
$$P_{k+1}(-) = P_k(+) + \int_k^{k+1} [AP + PA^T + CQC^T]dt \tag{35}$$

To simulate the data sent from a satellite, the path of the UAV was optimized from an initial point to a final point without any estimator, and then the resultant trajectory was saved and is sample point was perturbed by a random amount.

## V. Integration

The NLMPC and EKF were integrated using the algorithm defined in Fig. 1. The algorithm is run repeatedly until we reach the target point. The NLPMC calculates the optimal input ($u^*_{k-1}$) at the current time step ($k-1$) to move to the next time step such that the next state does not violate any constraints. The state is then propagated to the next state ($x_k(-)$) using nonlinear dynamics discussed in Section III. An estimate of the error covariance matrix ($P_k(-)$) is calculated to understand the deviation of the predicted output from the measured output (satellite data). The Kalman gain ($L_k$) is then calculated to calculate the relation between predicted and measured output. The gain is then used to correct the estimate of $x_k(-)$ using nonlinear dynamics, and $P_k(-)$ using dynamics that are linearized as that estimate.

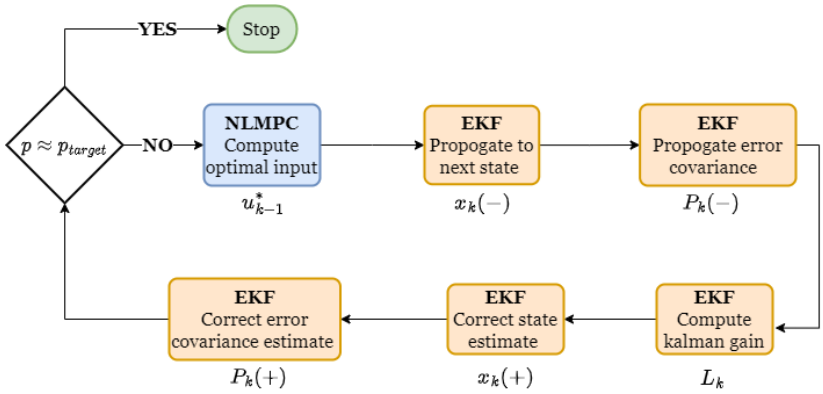



Figure 1: Algorithm for the control and estimation of the UAV

## VI. Results and Analysis

### A. UAV Control in obstacle-free environment

*1) Using linear dynamics:* In an obstacle-free environment, the UAV motion can be modeled using kinematic equations described in Eqn. (1) - (4). The target point is chosen to be the origin, and the initial point is randomly chosen within the radar range about the target point. As a regulatory problem with linear dynamics, the optimal control path for the UAV is obtained using linear MPC. The MPC was initialized with the values shown in the (36) - (39). No estimator was used here.

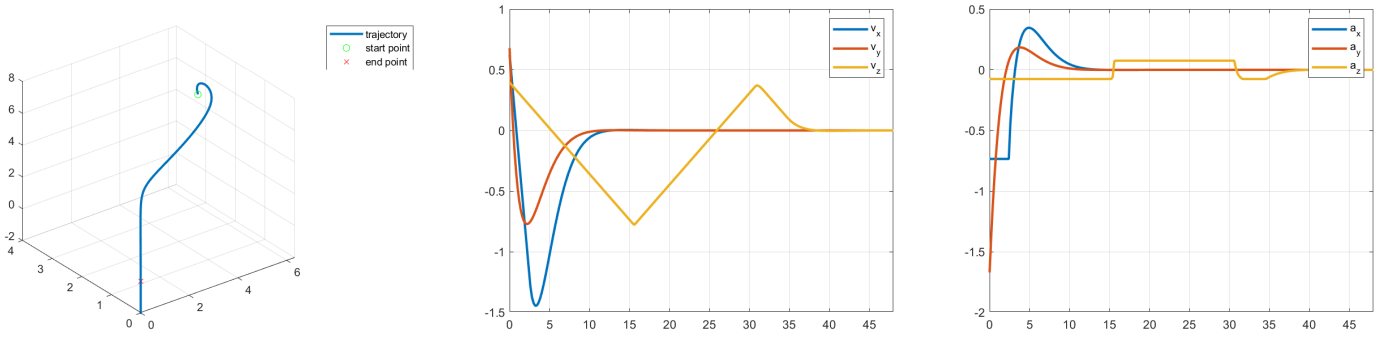
$$p_0 = \begin{bmatrix} 6 & 3 & 5 \end{bmatrix} \tag{36}$$

$$p_{target} = \begin{bmatrix} 0 & 0 & 0 \end{bmatrix} \tag{37}$$

$$v_0 = \begin{bmatrix} 0.6241 & 0.6791 & 0.3955 \end{bmatrix} \tag{38}$$

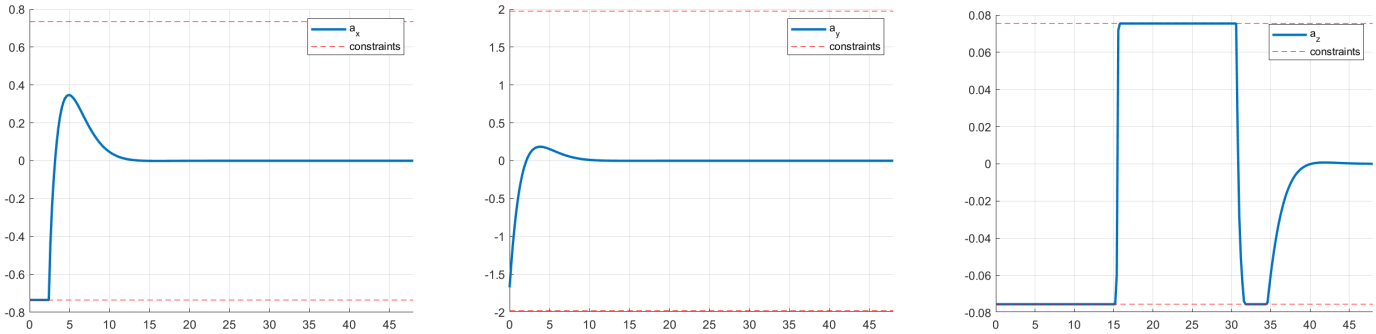
$$u_{constraints} = \pm \begin{bmatrix} 0.7349 & 1.9760 & 0.0755 \end{bmatrix} \tag{39}$$

It can be seen from Fig. 2a, that in the absence of any obstacles, a linear MPC controller is able to guide the UAV from the initial to the final point and can stay at the final point as can be seen from Fig. 2b and Fig. 2c that the UAV does not have any velocity or acceleration after reaching the target point. Due to the upper and lower bounds on the acceleration of the drone, the optimal path suggested by the MPC is not a straight line to the target point, but a curve that enables the UAV to fly within limits.




(a) Trajectory over time (b) Time history of velocity (c) Time history of acceleration

Figure 2: UAV position, velocity, and acceleration history

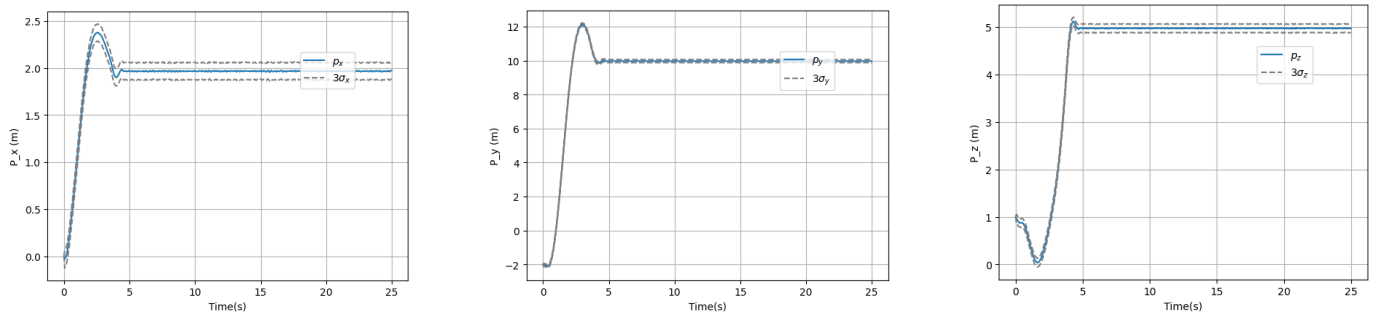



(a) Time history of acceleration in the x-direction (b) Time history of acceleration in the x-direction (c) Time history of acceleration in the x-direction

Figure 3: The UAV acceleration along each axis does not violate input bounds

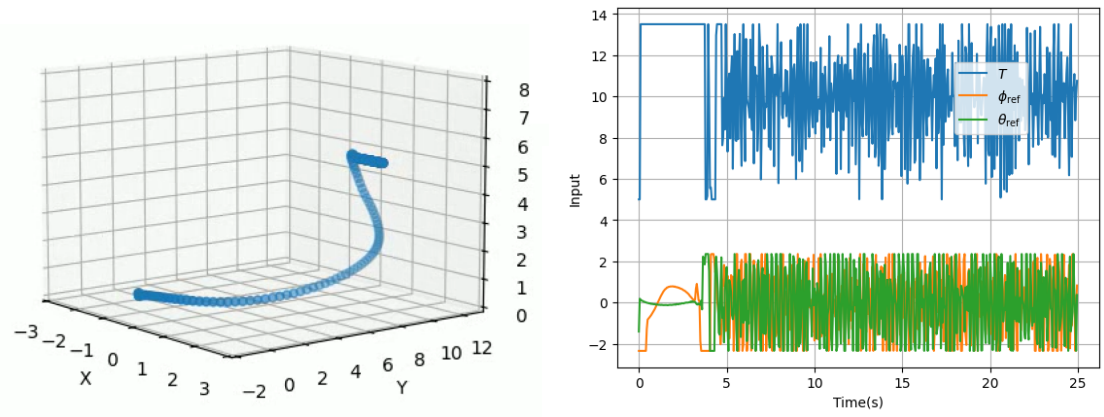
*2) Using nonlinear dynamics:* While the nonlinear dynamics were modeled particularly to control the UAV in an environment with obstacles, the NLMPC should be able to control the UAV in an obstacle-free environment. The EKF is used as an estimator to correct the prediction of the state matrix.

The results shown in Figs. 4 and 5 that the UAV reaches the target point without violating input constraints. The oscillation in the time history of the input plots is due to the continuous correction of the state/location of the UAV using the satellite data.




(a) Time history of x-position (b) Time history of y-position (c) Time history of z-position

Figure 4: The UAV is able to reach the desired point.




(a) Trajectory of UAV

(b) Time history of inputs

Figure 5: The UAV ris able to correct navigata to the final point with input limits. The

### *B. UAV control and estimation around static obstacle*

This section uses the NLMPC discussed in section III to navigate the UAV from the initial to the final point by learning its location using satellite data. The problem setup is described as follows:

$$r_{\text{obs}} = 2, r_s = 0.5 \tag{40}$$

$$p_{\text{target}} = \begin{bmatrix} 2 & 10 & 5 \end{bmatrix}^T \tag{41}$$

$$u_{\text{min}} = \begin{bmatrix} 5.0 & -0.35 & -0.35 \end{bmatrix}^T \tag{42}$$

$$u_{\text{max}} = \begin{bmatrix} 13.5 & 0.35 & 0.35 \end{bmatrix}^T \tag{43}$$

$$x_0 = [0 \quad -2 \quad 1 \quad 0.2 \quad 0.2 \quad 0.1 \quad 1 \quad 0.5 \quad 0.0 \quad 8.0 \quad 2.0]^T \tag{44}$$

Additionally, the horizon length $N$ and the $ts$ are chosen as 14 and 50 ms, respectively. The choice of horizon length $N$ is determined by balancing computational complexity with the need to anticipate obstacles in time and avoid collisions. Further, the weight in (20) has diagonal elements as 3 and 10 for $Q$ and $Q_N$. Note that the constraint of the extended state to enforce obstacle avoidance is shown as in (17) (18).

To simulate the satellite data, a previous trajectory of the UAV obtained with the assumption that the UAV position is known at all times is modified with some noise. More sensor noise described in Eqn. (25) - (26) were chosen as:

$$q = 0.1 \tag{45}$$

$$r = 0.1 \tag{46}$$

The results can be seen in Figs. 6 and 7. The results show that the UAV is able to navigate to the final point while avoiding the spherical static obstacle.

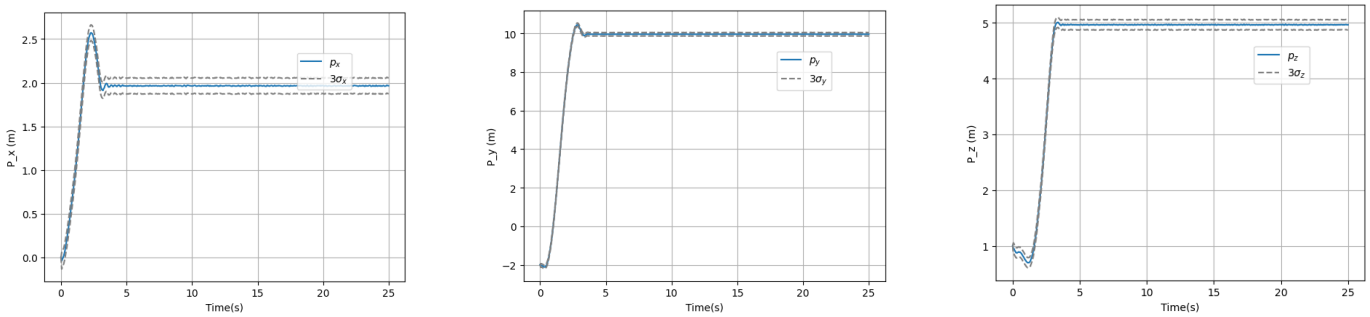
### *C. UAV control and estimation around dynamic obstacle*

This problem is set up the same as the static obstacle case, except now the obstacle can move and start at the origin. The trajectory of the dynamic obstacle is assumed to follow a constant velocity motion as follows:

$$\begin{bmatrix} v_x^{\text{obs}} & v_y^{\text{obs}} & v_z^{\text{obs}} \end{bmatrix} = \begin{bmatrix} -0.05 & 0.6 & 0.2 \end{bmatrix}^T \tag{47}$$

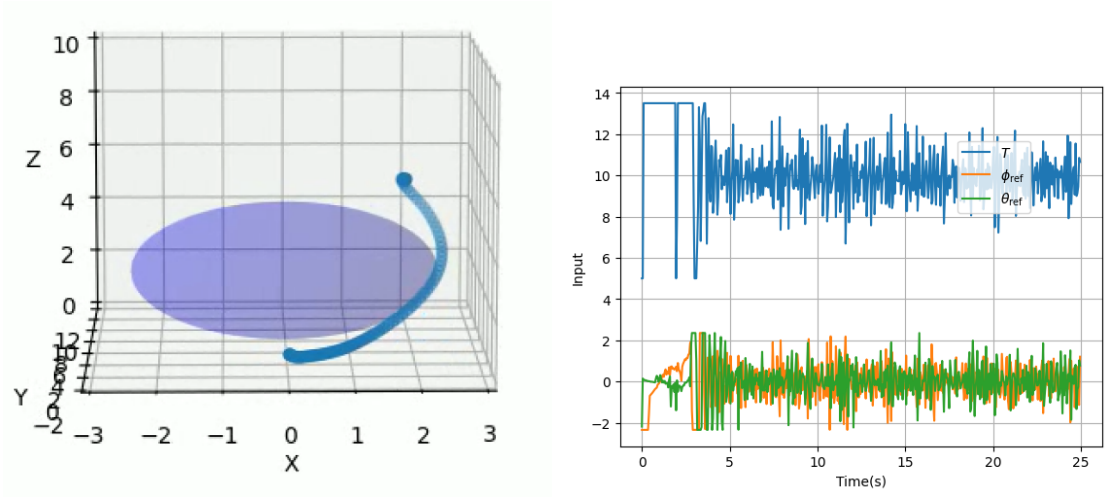
$$x_0 = [0 \quad -2 \quad 1 \quad 0.2 \quad 0.2 \quad 0.1 \quad 1 \quad 1 \quad 2 \quad 0 \quad 0]^T \tag{48}$$

The results can be seen in Figs. 8 and 9. One limitation of NLMPC is the computation time, so we want to validate our result by comparing the computation time and the discretization time step. We expect the simulation




(a) Time history of x-position (b) Time history of y-position (c) Time history of z-position

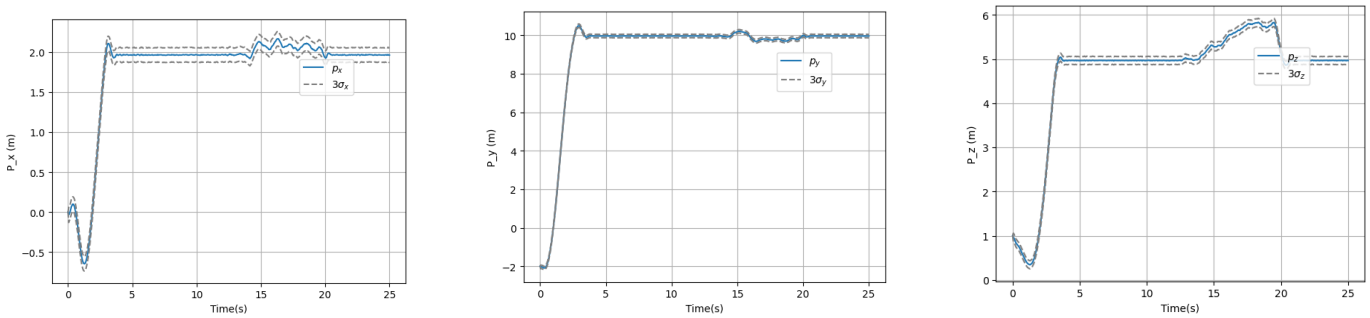
Figure 6: The UAV is able to reach the desired point and the variation in the position around the target point is reasonable.




(a) Trajectory of UAV (b) Time history of inputs

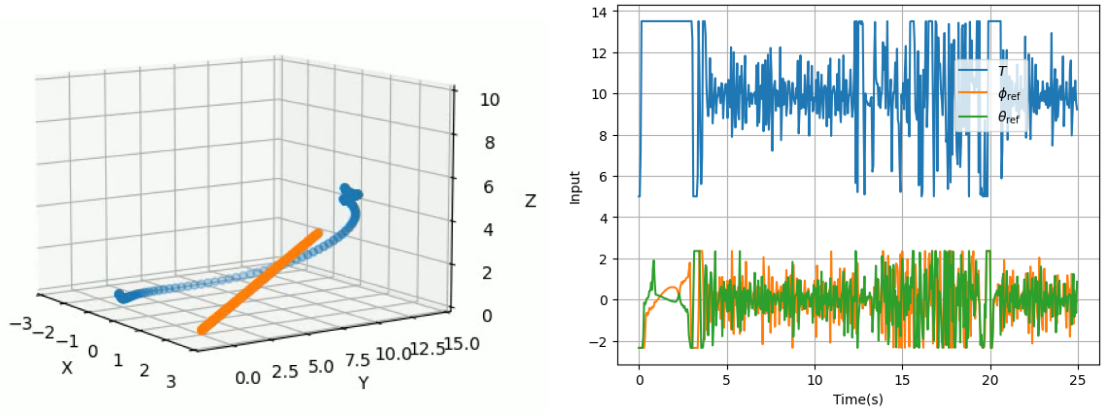
Figure 7: The UAV is able to detect the spherical obstacle and correct its course to maintain a safe distance from the obstacle while maintaining a short distance to the target point. The inputs show a lot of oscillation as the UAV is repeatedly correcting its course using satellite data.

time be shorter than the discretization time since the Proximal Averaged Newton-type Method (PANOC) has the advantage of being more efficient than other optimization algorithms using Sequential Quadratic Programming (SQP) [8]. For the three cases where the obstacle states are known, the results are validated. However, for the dynamic obstacle case with real measurement, the simulation time is longer than the discretization time.




(a) Time history of x-position (b) Time history of y-position (c) Time history of z-position

Figure 8: The UAV is able to reach the desired point and the variation in the position around the target point is reasonable. The UAV reaches the target point before the dynamic obstacle, but as the obstacle passes closer to the target point, the UAV corrects its course around the target point to the main a safe distance from the obstacle until it passes. This corresponds to the second set of oscillations.




(a) Trajectory of UAV (b) Time history of inputs

Figure 9: The UAV is able to detect the dynamic obstacle shown in orange and correct its course to maintain a safe distance from the obstacle while maintaining a short distance to the target point. The inputs show a lot of oscillation as the UAV is repeatedly correcting its course using satellite data.

## VII. Conclusion

This project showed the trajectory optimization of a UAV in an unknown environment. The UAV was modeled using nonlinear dynamics and a nonlinear model predictive control (MPC) was used to minimize the distance from the initial to the final point while detecting obstacles at each time point in time to maintain a safe distance from the obstacles, and to reach the target point within UAV flying limitations. The UAV was fed with simulated satellite data at each location to correct its estimate of its location using an extended Kalman filter (EKF). The results proved the ability of the controller and estimator to be able to detect obstacles in real-time and correct the course of the UAV toward the target along with correcting its estimate of its location using UAV to propose the best input to move to the next location.

## VIII. Contribution

1) **Atharva Gunda** : Assisted in the formulation and implementation of linear MPC, and implemented EKF.
2) **Hsin-Ying Lin** : The formulation of linear and nonlinear dynamics. Nonlinear MPC implementation with simulation. Assisted in EKF implementation.
3) **Falak Mandali**: Assisted in the formulation and validation of linear and nonlinear dynamics, in the implementation of the linear MPC, and in the formulation and implementation of the EKF.

# REFERENCES


[1] C. Cheng, Q. Sha, B. He, and G. Li, "Path planning and obstacle avoidance for AUV: A review," *Ocean Engineering*, vol. 235, p. 109355, Sep. 2021. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S002980182100771X

[2] J.-W. Lee, B. Walker, and K. Cohen, "Path planning of unmanned aerial vehicles in a dynamic environment," in *Infotech@Aerospace 2011*. American Institute of Aeronautics and Astronautics. [Online]. Available: https://arc.aiaa.org/doi/10.2514/6.2011-1654

[3] T. Baca, D. Hert, G. Loianno, M. Saska, and V. Kumar, "Model predictive trajectory tracking and collision avoidance for reliable outdoor deployment of unmanned aerial vehicles," in *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, pp. 6753–6760. [Online]. Available: https://ieeexplore.ieee.org/document/8594266/

[4] B. Lindqvist, S. S. Mansouri, A.-a. Agha-mohammadi, and G. Nikolakopoulos, "Nonlinear MPC for collision avoidance and control of UAVs with dynamic obstacles," vol. 5, no. 4, pp. 6001–6008. [Online]. Available: https://ieeexplore.ieee.org/document/9145644/

[5] N. Xue, L. Niu, X. Hong, Z. Li, L. Hoffaeller, and C. Pöpper, "DeepSIM: GPS spoofing detection on UAVs using satellite imagery matching," in *Annual Computer Security Applications Conference*. ACM, pp. 304–319. [Online]. Available: https://dl.acm.org/doi/10.1145/3427228.3427254

[6] S. Driessen, N. Janssen, L. Wang, J. Palmer, and H. Nijmeijer, "Experimentally validated extended kalman filter for UAV state estimation using low-cost sensors," vol. 51, no. 15, pp. 43–48. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S2405896318317488

[7] M. Kamel, T. Stastny, K. Alexis, and R. Siegwart, "Model predictive control for trajectory tracking of unmanned aerial vehicles using robot operating system," *Robot Operating System (ROS) The Complete Reference (Volume 2)*, pp. 3–39, 2017.

[8] L. Stella, A. Themelis, P. Sopasakis, and P. Patrinos, "A simple and efficient algorithm for nonlinear model predictive control," in *2017 IEEE 56th Annual Conference on Decision and Control (CDC)*. IEEE, 2017, pp. 1939–1944.

# IX. Appendix : Code

## A. Linear Model Predictive Control (MPC) [MATLAB]

```
clear; clc; close all;

% initial conditions
x0 = [randi(10) randi(10) randi(10) rand() rand() rand()]'

% constraints
uc = [2*rand() 2*rand() 2*rand()]'

% prediction horizon
N = 8;

% sampling time
ts = 0.2;   % s
tf = 60;    % 1 min

% drone dynamics
% xd = [px py pz vx vy vz]'
% 6x6
A = [eye(3) ts*eye(3); zeros(3) eye(3)];
% 6x3
B = [zeros(3); ts*eye(3)];
% 3x6
C = [eye(3) zeros(3)];
% 3x3
D = zeros(size(C,1),size(B,2));

% controllability
CO = ctrb(A,B);
% rank(CO)

% observability
OB = obsv(A,C);
% rank(OB)

% MPC
% constraint matrices
E = calcE();
W = calcW(-uc,uc,N);

% state prediction
[G,H] = getGH(A,B,N);

% cost weight
Q = eye(size(A,1));
R = eye(size(B,2));
P = Q;
Qb = blkdiag(Q,Q,Q,Q,Q,Q,Q,P);
Rb = blkdiag(R,R,R,R,R,R,R,R);

M = Q + (H'*Qb*H);
F = G'*Qb*H;
L = G'*Qb*G + Rb;
```

```
% initializations
t = 0:ts:tf;
tlen = length(t)-1;
uMPC = zeros(size(B,2),tlen + 1);
xMPC = zeros(size(A,1),tlen + 1);
xMPC(:,1) = x0;
yMPC = zeros(size(C,1),tlen + 1);
yMPC(:,1) = C*xMPC(:,1);
ytol = 1E-6*ones(3,1);

iA = false(size(W));
opt = mpcActiveSetOptions;
opt.IntegrityChecks = false;

% 1st iteration
i = 1;
[u, ~, ~] = mpcActiveSetSolver(L,F*(xMPC(:,i)),E,W,[],[],iA,opt);
uMPC(:,i) = u(1:3,1);
xMPC(:,i+1) = A*xMPC(:,i) + B*uMPC(:,i);
yMPC(:,i+1) = C*xMPC(:,i+1);

% all other iterations
while (norm(yMPC(:,i)) > ytol)
    [u, ~, ~] = mpcActiveSetSolver(L,F*(xMPC(:,i)),E,W,[],[],iA,opt);
    uMPC(:,i) = u(1:3,1);
    xMPC(:,i+1) = A*xMPC(:,i) + B*uMPC(:,i);
    yMPC(:,i+1) = C*xMPC(:,i+1);
    i = i + 1;
end
disp(i);

%plot_results(t(1:i),yMPC(:,1:i),uMPC(:,1:i),xMPC(:,1:i), ...
 %   uc.*ones(length(uc),length(t(1:i))),x0(1:3),t(i));

% Functions
function [G,H] = getGH(A,B,N)
    H = [];
    G = [];
    for r = 1:N
        H = [H; A^r];
        g = [];
        for i = 1:r
            g = [g (A^(r-1))*B];
        end
        for j = i+1:N
            g = [g zeros(size(A,1),size(B,2))];
        end
        G = [G; g];
    end
end

function W = calcW(u_min,u_max,N)
    W = [];
```

```
    for i = 1:length(u_min)
        W = [W;u_max(i);-u_min(i)];
    end
    W = repmat(W,N,1);
end

function E = calcE()
    E = [1;-1];
    E = blkdiag(E,E,E);
    E = blkdiag(E,E,E,E,E,E,E,E);
end

function plot_results(t,y,u,x,uc,x0,tf)
    % position
    figure;
    plot(t,y,LineWidth=2);
    grid on; xlim([0 tf]);
    legend('p_x','p_y','p_z');

    % velocity
    figure;
    plot(t,x(4:6,:),LineWidth=2);
    grid on; xlim([0 tf]);
    legend('v_x','v_y','v_z');

    % acceleration
    figure;
    plot(t,u,LineWidth=2);
    grid on; xlim([0 tf]);
    legend('a_x','a_y','a_z');

    % input constraints
    for i = 1:size(u,1)
        figure; hold on;
        plot(t,u(i,:),LineWidth=2);
        plot(t,-uc(i,:),'r--');
        plot(t,uc(i,:),'r--');
        hold off; grid on; xlim([0 tf]);

        if i == 1
            legend('a_x','constraints');
        elseif i == 2
            legend('a_y','constraints');
        else
            legend('a_z','constraints');
        end
    end

    % trajectory
    figure;
    plot3(y(1,:),y(2,:),y(3,:),LineWidth=2);
    hold on;
    scatter3(x0(1),x0(2),x0(3),'go');
    scatter3(0,0,0,'rx');
```

```
    hold off; grid on;
    legend('trajectory','start point','end point');
end
```

```
x0 =

    9.0000
   10.0000
    2.0000
    0.9134
    0.6324
    0.0975


uc =

    0.5570
    1.0938
    1.9150

   170
```

*Published with MATLAB® R2023b*

### *B. Nonlinear Model Predictive Control (MPC) [Python]*

In [ ]:
```
import opengen as og
import casadi.casadi as cs
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from autograd import jacobian
import sympy as sp
import numpy as np
import math
```

In [ ]:
```
# set up known obstacles
robs_1=2
rs=0.5 # safety radius
```

In [ ]:
```
# dynamics function for the system
# dynamics
def dynamics_ct(x, u):
    T=u[0]
    phi_r=u[1]
    theta_r=u[2]

    tau_phi=0.5
    tau_theta=0.5

    K_phi=1
    K_theta=1

    cp=math.cos(x[6])
    sp=math.sin(x[6])
    ct=math.cos(x[7])
    st=math.sin(x[7])
    vector_v = np.array([x[3], x[4], x[5]])
    R = np.array([[cp, -sp, 0],
              [ct*sp, ct*cp, -st],
              [st*sp, cp*sp, ct]])

    vector_T = np.array([0, 0, T])

    vector_gravity = np.array([0, 0, -9.81])
    Ax = 0.1  # Example value for Ax
    Ay = 0.1  # Example value for Ay
    Az = 0.2  # Example value for Az

    diagonal_matrix = np.diag([Ax, Ay, Az])
    vdot = np.dot(R, vector_T) + vector_gravity - np.dot(diagonal_matrix, vector_v)

    xdot0 = x[3]
    xdot1 = x[4]
    xdot2 = x[5]
    xdot3 = cs.cos(x[6])*cs.sin(x[7])*u[0]-Ax*x[3]
    xdot4 = -cs.sin(x[6])*u[0] -Ay*x[4]
    xdot5 = cs.cos(x[6])*cs.cos(x[7])*u[0] -Az*x[5]-9.81
    xdot6 = 1/tau_phi*(K_phi*u[1]-x[6])
    xdot7 = 1/tau_theta*(K_theta*u[2]-x[7])
```

```
    # the obstacle dynamic equation
    xdot8 = 0    # velocity of obstcale in x dir
    xdot9 = 0  # velocity of obstcale in y dir
    xdot10 = 0  # velocity of obstcale in z dir


    return [xdot0, xdot1, xdot2, xdot3, xdot4, xdot5, xdot6, xdot7, xdot8, xdot9, x


def dynamics_dt(x, u,ts,nx):
    dx = dynamics_ct(x, u)
    return np.array([x[i] + ts * dx[i] for i in range(nx)])
```

In [ ]:

```
# dynamics function for the system
def dynamics_ct_sp(x, u):
    T = u[0]
    phi_r = u[1]
    theta_r = u[2]

    tau_phi = 0.5
    tau_theta = 0.5
    K_phi = 1
    K_theta = 1

    # cp = sp.cos(x[6])
    # sp = sp.sin(x[6])
    # ct = sp.cos(x[7])
    # st = sp.sin(x[7])
    # vector_v = np.array([x[3], x[4], x[5]])
    # R = np.array([[cp   , -sp  ,   0],
    #               [ct*sp,  ct*cp, -st],
    #               [st*sp,  cp*sp,  ct]])

    # vector_T = np.array([0, 0, T])
    # vector_gravity = np.array([0, 0, -9.81])
    Ax = 0.1    # Example value for Ax
    Ay = 0.1    # Example value for Ay
    Az = 0.2    # Example value for Az

    # diagonal_matrix = np.diag([Ax, Ay, Az])
    # vdot = np.dot(R, vector_T) + vector_gravity - np.dot(diagonal_matrix, vector_

    xdot = [0]*nx
    xdot[0] = x[3]
    xdot[1] = x[4]
    xdot[2] = x[5]
    # xdot3 = -Ax*x[3]
    # xdot4 = -cs.sin(x[6])*u[0] -Ay*x[4]
    # xdot5 = cs.cos(x[6])*u[0] -Ay*x[4]-9.81

    xdot[3] = sp.cos(x[6])*sp.sin(x[7])*u[0]-Ax*x[3]
    xdot[4] = -sp.sin(x[6])*u[0] -Ay*x[4]
    xdot[5] = sp.cos(x[6])*sp.cos(x[7])*u[0] -Az*x[5]-9.81
    xdot[6] = 1/tau_phi*(K_phi*u[1]-x[6])
    xdot[7] = 1/tau_theta*(K_theta*u[2]-x[7])
```

```
    # the obstacle dynamic equation
    xdot[8] = 0     # velocity of obstcale in x dir
    xdot[9] = 0     # velocity of obstcale in y dir
    xdot[10] = 0    # velocity of obstcale in z dir

    return xdot

def dynamics_dt_sp(x, u,ts,nx):
    dx = dynamics_ct_sp(x, u)
    return sp.Matrix([x[i] + ts * dx[i] for i in range(nx)])
```

In [ ]:

```
def EKF(xk,uk,yk,Pk,ts,nx):
    # symbols
    xs = sp.MatrixSymbol('x',11,1)  # 11x1
    us = sp.MatrixSymbol('u',3,1)   # 3x1

    # noise
    q_noise = 0.01
    r_noise = 0.001
    Qk = q_noise * np.eye(11)        # 11x11
    Qk = np.array(Qk).astype(np.float64)
    Rk = r_noise * np.eye(3)         # 3x3
    Rk = np.array(Rk).astype(np.float64)

    # state jacobian
    fx = dynamics_dt_sp(xs,us,ts,nx)                                    # 11x1
    Ax = fx.jacobian(xs)                                                # 11x11
    # measurement jacobian
    gx = sp.Matrix(xs[0:3])                                             # 3x1

    # propagation
    xk_ = dynamics_dt_sp(xk,uk,ts,nx)   # 11x1
    Ak = Ax.subs({xs[0]:xk[0],
                  xs[1]:xk[1],
                  xs[2]:xk[2],
                  xs[3]:xk[3],
                  xs[4]:xk[4],
                  xs[5]:xk[5],
                  xs[6]:xk[6],
                  xs[7]:xk[7],
                  xs[8]:xk[8],
                  xs[9]:xk[9],
                  xs[10]:xk[10],
                  us[0]:uk[0],
                  us[1]:uk[1],
                  us[2]:uk[2]})         # 11x11
    Ak = np.array(Ak).astype(np.float64)
    Pk = np.array(Pk).astype(np.float64)
    Pk_ = Ak*Pk*Ak.transpose() + Qk    # 11x11
    hx = gx.jacobian(xs[0:3]).subs({xs[0]:xk[0],
                                    xs[1]:xk[1],
                                    xs[2]:xk[2]})         # 3x11
    Hk = np.column_stack((hx,np.zeros((3,8))))            # 3x11
    Hk = np.array(Hk).astype(np.float64)

    # kalman gain
```

```
    Lk = Pk_@Hk.T@np.linalg.inv(Hk@Pk_@Hk.T + Rk)
    Lk = np.array(Lk).astype(np.float64)

    # correction
    gx = gx.subs({xs[0]:xk[0],xs[1]:xk[1],xs[2]:xk[2]})
    gx = np.array(gx).astype(np.float64)
    xkp = np.array(xk_) + Lk@(np.array(yk) - gx)    # 11x1
    Pkp = Pk_ - Lk@Hk@Pk_                           # 11x11

    return [xkp, Pkp]
```

In [ ]:

```
# Build parametric optimizer
# ------------------------------------
(nu, nx, N, ts) = (3, 11, 20, 0.05)
(xref, yref, zref) = (2, 10, 5)
(q,r,qN)= (3,0,100)

u = cs.SX.sym('u', nu*N)
z0 = cs.SX.sym('z0', nx)

x_t=z0
cost = 0
f2 = []
for t in range(0, nu*N, nu):
    cost += q*((x_t[0]-xref)**2 + (x_t[1]-yref)**2 + (x_t[2]-zref)**2)
    u_t = u[t:t+3]
    # cost += r * cs.dot(u_t, u_t)
    x_t = dynamics_dt(x_t, u_t, ts, nx)
    # x_t [0:8]= dynamics_dt(x_t[0:8], u_t, ts)
    # x_t [8:11]= dynamics_obs(x_t [8:11],ts)

    f2 = cs.vertcat(f2,cs.fmax(0.0, (robs_1+rs)**2-(x_t[0]-x_t[8])**2-(x_t[1]-x_t[9
    #f2=cs.vertcat(f2, cs.fmax(0, 1 - x_t[0]**2 - x_t[1]**2 - x_t[2]**2))


cost += qN*((x_t[0]-xref)**2 + (x_t[1]-yref)**2+ (x_t[2]-zref)**2)

#umin = [-10.0] * (nu*N)
#umax = [10.0] * (nu*N)

umin = [5,-2.35,-2.35] * N
umax = [13.5,2.35,2.35] * N
bounds = og.constraints.Rectangle(umin, umax)

problem = og.builder.Problem(u, z0, cost).with_penalty_constraints(f2).with_constra

build_config = og.config.BuildConfiguration()\
    .with_build_directory("my_optimizers")\
    .with_build_mode("debug")\
    .with_tcp_interface_config()
meta = og.config.OptimizerMeta()\
    .with_optimizer_name("navigation")
solver_config = og.config.SolverConfiguration()\
    .with_tolerance(1e-5)\
    .with_delta_tolerance(1e-4)\
    .with_initial_penalty(100000.0)\
```

```
    .with_max_outer_iterations(20)
builder = og.builder.OpEnOptimizerBuilder(problem,
                                          meta,
                                          build_config,
                                          solver_config)
builder.build()



# # Use TCP server
# # ------------------------------------
# mng = og.tcp.OptimizerTcpManager('my_optimizers/navigation')
# mng.start()

# mng.ping()
# solution = mng.call([1.0, 2.0, 0.0, 1.0,1.0,1.0,1.0,1.0, 2,0,0], initial_guess=[5
# print(solution.is_ok())
# solution_data = solution.get()
# u_star = solution_data.solution
# exit_status = solution_data.exit_status
# solver_time = solution_data.solve_time_ms
# print("u_star: ",u_star)
# print("exit_status: ",exit_status)
# print("solver_time: ",solver_time)
# mng.kill()
```

In [ ]:
```
# Run this when port is not availiable
mng.kill()
```

In [ ]:
```
# import state and make it into measurement
yk_matrix = np.load('x_cache_static.npy')
# print(yk_matrix[0:3,:].shape)
yk_matrix_wn= yk_matrix[0:3,:]+0.01*np.random.rand(3,501)-0.05 # with noise
# print(yk_matrix_wn[0:3,1])
```

In [ ]:
```
# Use TCP server
# ------------------------------------

mng = og.tcp.OptimizerTcpManager('my_optimizers/navigation')
mng.start()

pong = mng.ping()
#print(pong)
def mpc_controller(state, nu, N):
    initial_guess = [1, 1, 2] * N
    #initial_guess = ic * (N)
    solver_status  = mng.call(state, initial_guess)

    #print(f"initial guess : ", initial_guess)
    #print(f"initial guess original: ", initial_guess_original)
    #print(f"solver_status: ", solver_status)


    if solver_status.is_ok():
        us = solver_status['solution']
```

```
        solution_data = solver_status.get()
        print(f"exit_status: ", solution_data.exit_status)
        print(f"solve_time_ms: ", solution_data.solve_time_ms)

        #print(f"f2_norm: ", solution_data.f2_norm)
        #print(us)
    else:
        solver_error = solver_status.get()
        error_code = solver_error.code
        error_msg = solver_error.message
        print(f"error_code: ", error_code)
        print(f"error msg: ", error_msg)
        us = "none"
    return us[0:nu]


Nsim = 500
x_cache = np.zeros((nx, Nsim+1))
x_cache[:, 0] = np.array([0, -2, 1, 0.2, 0.2, 0.1,1, 0.5,0.0, 8.0, 2.0])
P_cache = np.zeros((Nsim+1,3))
# states: px,py,pz,vx,vy,vz,phi, theta,px_obs,py_obs,pz_obs
u_mpc_array = []

# initialization for EKF
Pkp= np.eye(11) # 11*11
# yk= [[0], [-2], [1]] # starting point




for t in range(Nsim):
    # 0, 1 ...
    #print(f'Nsim:', Nsim)
    x_current = x_cache[:, t]
    # u_mpc = mpc_controller(x_current,nu,N)

    u_mpc = mpc_controller(x_current,nu,N)
    #print("current state: ",x_current[0:3])

    u_mpc_array.append(u_mpc)
    #ic = u_mpc #update initial guess using the current input
    #print("ic: ",ic)
    yk = yk_matrix_wn[:, t]
    [xkp, Pkp] = EKF(x_current,u_mpc,[[yk[0]], [yk[1]], [yk[2]]],Pkp,ts,nx)

    x_cache[:, t+1] = xkp[:,0]

    P_cache[t+1, :] = np.array([Pkp[0,0],Pkp[1,1],Pkp[2,2]])
    print(f'Pkp : {P_cache[t+1, :]}')


mng.kill()
# print(u_mpc)

# plot control input
```

```
u_mpc_array = np.array(u_mpc_array)
# print("Shape of input array:",u_mpc_array.shape)

time_array_2=np.arange(0, Nsim, 1)*ts
plt.figure()
plt.plot(time_array_2, u_mpc_array[:, 0], '-', label=r"$T$")
plt.plot(time_array_2, u_mpc_array[:, 1], '-', label=r"$\phi_{\text{ref}}$")
plt.plot(time_array_2, u_mpc_array[:, 2], '-', label=r"$\theta_{\text{ref}}$")

plt.grid()
plt.ylabel('Input')
plt.xlabel('Time(s)')
plt.legend(bbox_to_anchor=(0.7, 0.85), loc='upper left', borderaxespad=0.)
plt.show()


# plot drone states
plt.figure()
time_array_3=np.arange(0, Nsim+1, 1)*ts
plt.plot(time_array_3, x_cache[0, :], '-', label=r"$p_{x}$")
plt.plot(time_array_3, x_cache[1, :], '-', label=r"$p_{y}$")
plt.plot(time_array_3, x_cache[2, :], '-', label=r"$p_{z}$")

plt.grid()
plt.ylabel('States(m)')
plt.xlabel('Time(s)')
plt.legend(bbox_to_anchor=(0.7, 0.85), loc='upper left', borderaxespad=0.)
plt.show()


# plot drone x state with 3 sigma
plt.figure()
time_array_3=np.arange(0, Nsim+1, 1)*ts
plt.plot(time_array_3, x_cache[0, :], '-', label=r"$p_{x}$")
plt.plot(time_array_3, x_cache[0, :]+3*np.sqrt(P_cache[:,0].T), '--', color='grey',
plt.plot(time_array_3, x_cache[0, :]-3*np.sqrt(P_cache[:,0].T), '--', color='grey')

plt.grid()
plt.ylabel('P_x (m)')
plt.xlabel('Time(s)')
plt.legend(bbox_to_anchor=(0.7, 0.85), loc='upper left', borderaxespad=0.)
plt.show()


# plot drone y state with 3 sigma
plt.figure()
time_array_3=np.arange(0, Nsim+1, 1)*ts
plt.plot(time_array_3, x_cache[1, :], '-', label=r"$p_{y}$")
plt.plot(time_array_3, x_cache[1, :]+3*np.sqrt(P_cache[:,1].T), '--', color='grey',
plt.plot(time_array_3, x_cache[1, :]-3*np.sqrt(P_cache[:,1].T), '--', color='grey')

plt.grid()
plt.ylabel('P_y (m)')
plt.xlabel('Time(s)')
plt.legend(bbox_to_anchor=(0.7, 0.85), loc='upper left', borderaxespad=0.)
plt.show()
```

```
# plot drone z state with 3 sigma
plt.figure()
time_array_3=np.arange(0, Nsim+1, 1)*ts
plt.plot(time_array_3, x_cache[2, :], '-', label=r"$p_{z}$")
plt.plot(time_array_3, x_cache[2, :]+3*np.sqrt(P_cache[:,2].T), '--', color='grey',
plt.plot(time_array_3, x_cache[2, :]-3*np.sqrt(P_cache[:,2].T), '--', color='grey')

plt.grid()
plt.ylabel('P_z (m)')
plt.xlabel('Time(s)')
plt.legend(bbox_to_anchor=(0.7, 0.85), loc='upper left', borderaxespad=0.)
plt.show()

# plot obstacle states
plt.figure()
time_array_3=np.arange(0, Nsim+1, 1)*ts
plt.plot(time_array_3, x_cache[8, :], '-', label=r"$p_{x}^{obs}$")
plt.plot(time_array_3, x_cache[9, :], '-', label=r"$p_{y}^{obs}$")
plt.plot(time_array_3, x_cache[10, :], '-', label=r"$p_{z}^{obs}$")

plt.grid()
plt.ylabel('Obstacle States(m)')
plt.xlabel('Time(s)')
plt.legend(bbox_to_anchor=(0.7, 0.85), loc='upper left', borderaxespad=0.)
plt.show()
```

In [ ]:
```
# Check for interception
sphere_center = np.array([0, 8, 2])  # Center of the sphere
sphere_radius = 2.5  # Radius of the sphere
interception_points = []
for i in range(len(x_cache[0, :])):
    point = np.array([x_cache[0, i], x_cache[1, i], x_cache[2, i]])
    #print("point : ",point)
    distance = np.linalg.norm(point - sphere_center)
    #print("distance : ",distance )
    if distance <= sphere_radius:
        interception_points.append(point)
print("interception_points : ",interception_points )
```

In [ ]:
```
# plot trajectory in 3D
%matplotlib widget
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.scatter(x_cache[0, : ], x_cache[1, : ], x_cache[2, : ], label='Drone') # positio
    #ax.scatter(x_cache[8, :], x_cache[9, :], x_cache[10, :], label='Obstacle') # p
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('Z')
ax.set_title('Drone Trajectory with Static Obstacle')

sphere_center = np.array([0,8,2])  # Example coordinates
r = 2.5  # Radius of the sphere
u = np.linspace(0, 2 * np.pi, 100)
v = np.linspace(0, np.pi, 100)
```

```
x_sphere = sphere_center[0] + r * np.outer(np.cos(u), np.sin(v))
y_sphere = sphere_center[1] + r * np.outer(np.sin(u), np.sin(v))
z_sphere = sphere_center[2] + r * np.outer(np.ones(np.size(u)), np.cos(v))
ax.plot_surface(x_sphere, y_sphere, z_sphere, color='b', alpha=0.2)
```

In [ ]:

```
# plot trajectory in 3D for the video

%matplotlib widget
for n in range(x_cache.shape[1]):
    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')
    ax.scatter(x_cache[0, : n], x_cache[1, : n], x_cache[2, : n], label='Drone') #
    #ax.scatter(x_cache[8, :], x_cache[9, :], x_cache[10, :], label='Obstacle') # p
    ax.set_xlabel('X')
    ax.set_ylabel('Y')
    ax.set_zlabel('Z')
    ax.set_title('Drone Trajectory with Static Obstacle')

    sphere_center = np.array([0, 10, 2])  # Example coordinates
    r = 2.5  # Radius of the sphere
    u = np.linspace(0, 2 * np.pi, 100)
    v = np.linspace(0, np.pi, 100)
    x_sphere = sphere_center[0] + r * np.outer(np.cos(u), np.sin(v))
    y_sphere = sphere_center[1] + r * np.outer(np.sin(u), np.sin(v))
    z_sphere = sphere_center[2] + r * np.outer(np.ones(np.size(u)), np.cos(v))
    ax.plot_surface(x_sphere, y_sphere, z_sphere, color='b', alpha=0.2)

    # Set the viewpoint (angle of rotation)
    ax.view_init(elev=10, azim=n)  # Rotate around the y-axis

    # Set the same scale for the x, y, and z axes
    ax.set_xlim(-3, 3)
    ax.set_ylim(-2, 12)
    ax.set_zlim(0, 10)

    # Save the plot as an image
    plt.savefig(f'./output/{n}.png')
    # Close the current figure to suppress the output
    plt.close()
```

In [ ]:

```
import cv2

# choose codec according to format needed
fourcc = cv2.VideoWriter_fourcc(*'mp4v')
width = 640
height = 480
fps = 60  # desired frames per second
video = cv2.VideoWriter('video.mp4', fourcc, fps, (width, height))

for i in range(0, 500):
   img = cv2.imread('./output/' + str(i) + '.png')
   video.write(img)

cv2.destroyAllWindows()
video.release()
```

### C. Extended Kalman Filter (EKF) [Python]

```
import numpy as np
from autograd import grad, jacobian
import sympy as sp

# symbols
xs = sp.MatrixSymbol('x',11,1)
us = sp.MatrixSymbol('u',3,1)

def EKF(xk,uk,yk,Pk,ts,nx):
    # size(x) = 11x1
    # size(u) = 3x1
    # size(w) = 3x1
    # size(y) = 3x1
    # size(v) = 3x1

    # size(A) = 11x11
    # size(B) = 11x3
    # size(C) = 11x3
    # size(H) = 3x11
    # size(G) = 3x3

    # size(L) = 11x3
    # size(P) = 3x3
    # size(Q) = 3x3
    # size(R) = 3x3

    # noise
    q = 0.1
    r = 0.1
    Q = q*np.eye(3)
    R = r*np.eye(3)

    # jacobian
    fx = dynamics_dt(xs,us,ts,nx)
    Ax = jacobian(fx)
    gx = measurement_dt(xs,ts,nx)
    Hx = np.vstack(jacobian(gx),np.zeros(8,11))

    # propagation
    xk_ = dynamics_dt(xk,uk,ts,nx)
    Ak = Ax.subs([(xs,xk),(us,uk)])
    Pk_ = Ak*Pk*Ak.transpose() + Q
    Hk = Hx.subs(xs,xk)

    # kalman gain
    Lk = Pk_*Hk.transpose()*np.linalg.inv(Hk*Pk_*Hk.tranpose() + R)

    # correction
    xkp = xk_ + Lk*(yk - gx.subs(xs,xk_))
    Pkp = Pk_ + Lk*Hk*Pk_
```

```
return [xkp, Pkp]
```