A Guidance and Control Framework for a Gliding Vehicle to Intercept a Moving Target

Ahmad Musallam, Michael Kerley, and Aadi Duggal

Abstract

A guidance and control algorithm has been developed for a point mass gliding vehicle using simplified non-linear dynamics. The vehicle is released at an altitude of 5000 m with an initial velocity of 100 m/s and the desired optimal trajectory to intersect a moving target at ground level is computed. An iterative Linear Quadratic Control (iLQR) is implemented in Python to obtain the optimal reference trajectory, which is updated every time there is new information about the target. The true trajectory is then perturbed by a wind gust model. To ensure our vehicle follows the optimal trajectory generated by the iLQR despite physical perturbations, we incorporate a Model Predictive Control Algorithm (MPC). The MPC computes the optimal control in the form of lateral accelerations of the vehicle (which represents lift caused by control surface deflections) to ensure our final desired state is reached. Sensor measurements of the vehicle state are produced at each discrete time which are processed by an Extended Kalman Filter (EKF) to estimate the state of the gliding vehicle. These estimates are fed into the MPC at each time step, thus defining our guidance and control loop. We show that this framework results in computationally efficient convergence, allowing the gliding vehicle to come close to intersecting the moving target. Results of accuracy, computation time, and technical discussions are presented.

I. Introduction

Target tracking of a gliding fixed wing aircraft has a wide array of practical applications. This includes but is not limited to autonomous operation, multi agent control, and search and rescue.

Simplified non-linear point-mass dynamics are used for this report. Due to the non-linear nature of the problem, traditional optimal control techniques such as Proportional-Integral-Control/Linear Quadratic Regulation are not applicable. Ideally, we want to pick a guidance law which can handle the non-linear nature of our problem, but also computes the optimal trajectory quickly enough such that it can be used on-board in a real system. Traditional non-linear control techniques such as the two point boundary value problem and dynamic programming are theoretically applicable, but practically useless due to computational costs. Our contribution to this problem is summarized below:

- On-Board Trajectory Optimization
  - The guidance law chosen to produce an optimal reference trajectory which intersects a moving target. differential dynamic programming (DDP) was the chosen method since it can handle the non-linear nature of the dynamics and the trajectories could be computed in under 0.2 seconds, allowing for an on-board guidance law.
- Robust Control Loop
  - Gusts were added in our simulation to deviate the vehicle from the optimal desired path. A Model Predictive Control algorithm was selected to output the control at each time step to ensure the vehicle follows the reference trajectory.
- State Estimation of the Vehicle
  - An Extended Kalman Filter was used to process simulated sensor measurements from the glider and produce state estimates at each time step. The estimate was fed into the MPC, thereby defining our guidance and control loop.

Differential dynamic programming is an iterative method for solving optimal trajectory problems introduced by David Mayne in 1966 [1]. Since then, it has seen many applications due to its robustness and computational efficiency. It was used in a formulation similar to ours such as in [2]. Here the authors used DDP to find the optimal

Purdue University, Aeronautics and Astronautics Engineering, West Lafayette, IN, 47907 USA (e-mail: amusalla@purdue.edu, kerley0@purdue.edu, duggal0@purdue.edu)

trajectory of a gliding missile. Similarly in [3], the authors used DDP for trajectory optimization for an engine-out transport aircraft.

The paper is organized as follows: in II we generate our problem statement and outline our method to solve it. We also provide the dynamics for our plant and disturbances such as wind which act of that plant. In III we use various methods to generate the optimal trajectory. In V we describe our sensing methods and our extended Kalman filter estimator. In IV we examine our nonlinear model predictive controller used to track the trajectory generated from III. In VI we examine the results of our trajectory optimizations and full simulation. In VII we draw conclusions based on those previous results.

II. Problem Formulation and setup

We consider an ideal point mass gliding vehicle with no autopilot delay. Then the three dimensional Kinematics model of the vehicle can be described by the dynamics used in the following papers [4] [5], [2]. They are rewritten below in (1) state space and (2) vector form for convenience.

$$\begin{aligned}
\dot{x} &= V\cos\psi\cos\gamma/s_p \\
\dot{y} &= V\sin\psi\cos\gamma/s_p \\
\dot{z} &= V\sin\gamma/s_p \\
\dot{V} &= \frac{-D}{m} - g\sin\gamma \\
\dot{\gamma} &= \frac{-a_z - g\cos\gamma}{V} \\
\dot{\psi} &= \frac{a_y}{V\cos\gamma}
\end{aligned} \tag{1}$$

$$\begin{aligned}
\dot{\vec{X}} &= [\dot{x} \quad \dot{y} \quad \dot{z} \quad \dot{V} \quad \dot{\gamma} \quad \dot{\psi}]^T = F(\vec{X},\vec{U}) \\
\vec{U} &= [a_y \quad a_z]^T \\
\vec{X} &= [x \quad y \quad z \quad V \quad \gamma \quad \psi]^T
\end{aligned} \tag{2}$$

where $[x, y, z]$ is the position vector of the vehicle in 3D space with represent to the inertial or earth frame. γ and ψ represent the flight path angle and the azimuth angle, and V is the velocity magnitude of the vehicle. g and m are the gravitational acceleration and the mass of the vehicle, respectively. s_p is a scaling parameter used to make plots more readable. Finally, D is the aerodynamic drag, which is modeled as...

$$D = \frac{1}{2}\rho V^2 S C_d \tag{3}$$

$$\rho = 1.15579 - (1.058e-4)\,z\,s_p + (3.725e-9)(z\,s_p)^2 - (6e-14)(z\,s_p)^3 \tag{4}$$

where ρ is the air density, S is the reference area, and C_d is the drag coefficient. The analytical solution for ρ (4) is an approximation of the international standard atmosphere or ISA+15.

The inputs a_y and a_z represent the acceleration that is a result of control surface deflection, here we are assuming that the autopilot is able to translate these acceleration requirements into control surface deflections. Hence, the inputs are limited to to physical constraints and they determine the vehicle maneuverability. This particular simplification was made out of convince.

$$|a_z| \le a_{max}, \quad |a_y| \le a_{max} \tag{5}$$

A wind gust model or disturbance is applied to the system to add complexity and prove that our setup is robust enough to work for imperfect conditions. We used the gust model used in MIL-F-8785C and its listed below for convince.

$$V_{gust} = \begin{cases} 0 & x > 0 \\ \frac{V_m}{2}(1 - cos(\frac{\pi x}{d_m})) & 0 \le x \le d_m \\ V_m & x > d_m \end{cases} \tag{6}$$

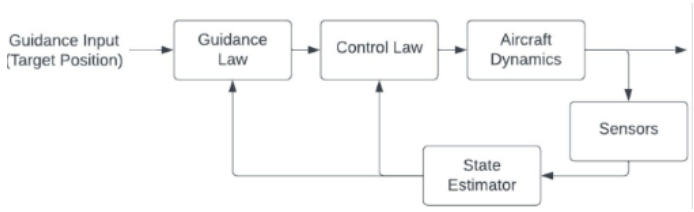


Fig. 1: Problem Framework

The general quadratic cost function is as follows...

$$J = \phi_f + \int_0^{t_f} L(X(t), U(t), t)dt \tag{7}$$

$$\phi_f = (\vec{X}_f - \vec{X_{rf}})^T Q_f (\vec{X} - \vec{X}_r) \tag{8}$$

$$L = (\vec{X} - \vec{X}_r)^T Q (\vec{X} - \vec{X}_r) + \vec{U}^T R \vec{U} \tag{9}$$

The accomplish the objectives outlined in section I we used the approach illustrated in figure 1. Where the Guidance law is generated by methods such as iLQR/DDP and nonlinear programming as outlined in III. For the control law to track the reference trajectory we used a nonlinear model predictive controller as noted in IV. The aircraft dynamics and notable disturbances are listed above in (1) and (6). Zero mean white noise is added to the outputs generated by (1) giving us our "measurements" which are then passed into a extended Kalman filter to provide an "estimate" as outlined in V. These estimates are passed into the Nonlinear MPC completing the control loop. The initial condition $\vec{X(0)}$ is known fully. For the final time condition $\vec{X(t_f)}$ only $[x \quad y \quad z]$ are known with $[V \quad \gamma \quad \psi]$ being free.

III. Trajectory Optimization

The general problem of trajectory optimization in discrete time can be summarized as follows

$$\textbf{Minimize} \qquad \phi(x_N) + \sum_{k=0}^{N-1} \mathcal{L}(x_k, u_k, \Delta t)$$

$$\textbf{Subject to}$$

$$\begin{aligned} & x_{k+1} = f(x_k, u_k, \Delta t), \\ & x(0) = x_0, \\ & g_k(x_k, u_k) \leq 0, \\ & h_k(x_k, u_k) = 0, \end{aligned} \tag{10}$$

where $\phi(x_N)$ is the cost at the final state, $\mathcal{L}$ is the running cost. g_k and h_k are inequality and equality constraints, respectively.

Methods for trajectory optimization are divided into two categories: "direct" and "indirect" methods. The direct approach involves considering both states and controls as decision variables and employing nonlinear programming (NLP) solvers like Sparse Nonlinear OPTimizers (SNOPT) or Interior Point OPTimizers (IPOPT). These methods leverage the resilience and versatility of general NLP solvers. Nevertheless, direct methods are typically slow and demand extensive optimization packages which makes them inefficient for online trajectory optimization on hardware constrained vehicles.

On the other hand, indirect methods break down the problem into smaller sub-problems by leveraging the Markovian structure of the problem. These techniques usually involve iteration, employing backward propagation of the cost function to determine the optimal control input, followed by a forward pass to enforce the system dynamics. Although less computationally demanding, the indirect methods are unable to handle constraints very well. examples of the indirect methods are differential dynamic programming (DDP) and iterative linear quadratic regulator (iLQR). In this section we will explore iLQR ability to solve our problem formulation and compare its performance with the direct method by using the IPOPT solver.

A. Direct Collocation

Direct methods for trajectory optimization discretize the trajectory into a finite set of points or segments, transforming the continuous problem into a finite-dimensional one. These methods directly parameterize the trajectory and solve for the optimal solution in the parameter space. Most direct collocation methods convert a continuous-time trajectory optimization challenge into a nonlinear program (NLP). NLP refers to a constrained parameter optimization problem with nonlinear terms in either its objective or constraint function. We setup our NLP problem as follows and solve it using IPOPT:

$$\begin{aligned}
&\textbf{Minimize} \quad && (x_N - x_f)^T Q_N (x_N - x_f) + \sum_{k=0}^{N-1} \Delta t_k, \\
&\textbf{Subject to} && \\
& && x_{k+1} = f(x_k, u_k, \Delta t), \\
& && x(0) = x_0, \\
& && u_{min} \leq u_k \leq u_{max}, \\
& && \Delta t_k = \Delta t_{k+1}, \\
& && (\mathtt{P}_{x,k} - \mathtt{P}_{x,c})^2 + (\mathtt{P}_{y,k} - \mathtt{P}_{y,c})^2 > \mathtt{r}^2
\end{aligned} \tag{11}$$

Where in the last constraint we assumed we have an cylindrical obstacle constraint with infinite height in the vertical direction. Here the big letters $\mathtt{P_x}, \mathtt{P_y}$ refer to the x and y positions with $\mathtt{P_{x,k}}, \mathtt{P_{y,k}}$ referring to the vehicle position while $\mathtt{P_{x,c}}, \mathtt{P_{y,c}}$ refer to the center of the cylinder with radius $\mathtt{r}$.

Here our cost function represents the fact that we are trying to hit the target with minimum time. The time minimization was achieved by having a fixed number of time steps and trying to minimize Δt by adding it as one of our control inputs. The constraint on Δt ensures that all time steps are equal length.

B. Iterative Linear Quadratic Regulator

iLQR is a DDP variant with the primary distinction between them being their approach to approximating the nonlinear dynamics. DDP employs a second-order Taylor series expansion for approximation, whereas iLQR relies on a first-order approximation. In theory DDP should converge in less iterations since it has a better approximation of the original function. But it was shown in practice that this is not the case because even though DDP converges in less iterations, each iteration is more expensive, making iLQR converge faster on some problems where the underlying dynamics can be approximated well by a linear function over a small region. In order to apply iLQR, the dynamics must be discretized. We approximate the discretize dynamics as a first-order Taylor series expansion about nominal trajectories.

$$x_{k+1} + \delta x_{k+1} = f(x_k + \delta x_k, u_k + \delta u_k) \approx f(x_k, u_k) + \left.\frac{\partial f}{\partial x}\right|_{x_k, u_k} (x - x_k) + \left.\frac{\partial f}{\partial u}\right|_{x_k, u_k} (u - u_k) \tag{12}$$

Which gives us the following discrete time linear system

$$\delta x_{k+1} = A(x_k, u_k)\delta x_k + B(x_k, u_k)\delta u_k \tag{13}$$

For LQR the cost function needs to be linear-quadratic. If it is not, a second-order Taylor-series expansion can be used to make it as such. In our formulation we assumed the cost function is quadratic of the following form

$$\begin{aligned}
J(x_0, U) &= \ell(x_N) + \sum_{k=1}^{N-1} \ell(x_k, u_k) \\
&= (x_N - x_f)^T Q_N (x_N - x_f) + \sum_{k=1}^{N-1} \frac{1}{2} u_k^T R_k u_k
\end{aligned} \tag{14}$$

1) Backward Pass: We begin by finding the augmented Lagrangian of (4):

$$\mathcal{L}_A = \ell_N(x_N) + (\lambda_N + \frac{1}{2}c_N(x_N)I_{\mu,N}) + \sum_{k=0}^{N-1}[\ell_k(x_k, u_k, \Delta t) + (\lambda + \frac{1}{2}c_k(x_k, u_k)^T I_{\mu,k})^T c_k(x_k, u_k)] \tag{15}$$

$$= \mathcal{L}_N(x_n, \lambda_N, \mu_n) + \sum_{k=0}^{N-1}\mathcal{L}_k(x_k, u_k, \lambda_k, \mu_k) \tag{16}$$

We now define the cost-to-go and the action-value function as follows

$$V_N(x_N)|\lambda, \mu = \mathcal{L}_N(x_N, \lambda_N, \mu_N) \tag{17}$$

$$V_k(x_k)|\lambda, \mu = \min_{u_k}\{\mathcal{L}_k(x_k, u_k, \lambda_k, \mu_k) + V_{k+1}(f(x_k, u_k, \Delta t))|\lambda, \mu\} \tag{18}$$

$$= \min_{u_k} G(x_k, u_k)|\lambda, \mu \tag{19}$$

then we take a second-order Taylor series expansion of the cost-to-go

$$\delta V_k(x) = \frac{1}{2}\delta x_k^T P_k \delta x_k + p_k^T \delta x_k \tag{20}$$

where P_k and p_k are the Hessian and gradient of the cost-to-go at time step k. Instead of optimizing the optimal trajectory directly, this Taylor series expansion means that we are now optimizing deviations about the optimal control trajectory. Since there are no controls to optimize at the final step, we can compute the cost-to-go immediately:

$$p_N = (\ell_N)_x = Q_N(x_N - x_f) \tag{21}$$

$$P_N = (\ell_N)_{xx} = Q_N \tag{22}$$

next we take the second order derivative of G_k with respect to the controls and states in order to find the relation between δV_{k+1} and δV_k,

$$\delta G_k = \frac{1}{2}\begin{bmatrix}\delta x_k \\ \delta u_k\end{bmatrix}^T \begin{bmatrix}G_{xx} & G_{xu} \\ G_{ux} & G_{uu}\end{bmatrix}\begin{bmatrix}\delta x_k \\ \delta u_k\end{bmatrix} + \begin{bmatrix}G_x \\ G_u\end{bmatrix}^T \begin{bmatrix}\delta x_k \\ \delta u_k\end{bmatrix} \tag{23}$$

and the block matrices are defined as (the time indices were dropped for brevity)

$$G_{xx} = \ell_{xx} + A^T P' A = A^T P' A \tag{24}$$

$$G_{uu} = \ell_{uu} + B^T P' B = R + B^T P' B \tag{25}$$

$$G_{ux} = \ell_{ux} + B^T P' A = B^T P' A \tag{26}$$

$$G_x = \ell_x + A^T p' = A^T p' \tag{27}$$

$$G_u = \ell_u + B^T p' = Ru + B^T p' \tag{28}$$

where $A = \partial f/\partial x|_{x_k,u_k}$, $B = \partial f/\partial u|_{x_k,u_k}$, and the $'$ indicates variables at the next time step (k+1).

Our objective is now to minimize (23) with respect to δu_k which gives a correction to the control trajectory. The result is a feedforward term d_k and a linear feedback term $K_k\delta x_k$. Here we also add regularization to ensure that G_{uu} is :

$$\delta u_k^* = -(G_{uu} + \rho I)^{-1}(G_{ux}\delta x_k + G_u) \equiv K_k \delta x_k + d_k \tag{29}$$

Substituting δu_k^* back into (23) gives us a closed form update for p_k and P_k:

$$P_k = G_{xx} + K_k^T G_{uu} K_k + K_k^T G_{ux} + G_{xu} K_k \tag{30}$$

$$p_k = G_x + K_k^T G_{uu} d_k + K_k^T G_u + G_{xu} d_k \tag{31}$$

$$\Delta V_k = d_k^T G_u + \frac{1}{2} d_k^T G_{uu} d_k \tag{32}$$

where ΔV_k is the expected cost improvement from the updated control trajectory.

Algorithm 1 Backward Pass

1: **function** BACKWARDPASS(X, U)
2: $P_N, p_n \leftarrow (21), (22)$
3: **for** $k = N-1 : -1 : 0$ **do**
4: $\delta Q \leftarrow (23), (24) - (26)$
5: **if** $Q_{uu} \succ 0$ **then**
6: $K_d, d, \Delta V \leftarrow (29), (32)$
7: **else**
8: Increase ρ and go to line 3
9: **end if**
10: **end for**
11: **return** $K, d, \Delta V$
12: **end function**

2) Forward Pass: We now use the obtained optimal control history to propagate they dynamics forward in time in order to ensure the dynamics are respected:

$$\delta x_k = \bar{x}_k - x_k \tag{33}$$
$$\delta u_k = K_k \delta x_k + \alpha d_k \tag{34}$$
$$\bar{u}_k = u_k + \delta u_k \tag{35}$$
$$\bar{x}_{k+1} = f(\bar{x}_k, \bar{u}_k) \tag{36}$$

where $\bar{x}_k$ and $\bar{u}_k$ are the updated nominal trajectories and $0 \leq \alpha \leq 1$ is a line search parameter. The algorithm for the forward pass and iLQR are given in [Algorithm 2] and [Algorithm 2] respectively.

Algorithm 2 Forward Pass

1: **function** FORWARDPASS($X, U, K, d, \Delta V, J$)
2: Initialize $\bar{x}_0 = x_0, \alpha = 1, J^- \leftarrow J$
3: **for** $k = 0 : 1 : N-1$ **do**
4: $\bar{u}_k = u_k + K_k(\bar{x}_k - x_k) + \alpha d_k$
5: $\bar{x}_{k+1} = f(\bar{x}_k, \bar{u}_k, \Delta t)$
6: **end for**
7: $J \leftarrow$ Using X, U
8: **if** J satisfies line search condition **then**
9: $X \leftarrow \bar{X}, U \leftarrow \bar{U}$
10: **else**
11: reduce α and go to line 3
12: **end if**
13: **return** K, U, J
14: **end function**

Algorithm 2 iLQR

1: initialize x_0, U, tolerance
2: $X \leftarrow$ simulate from x_0 using U
3: **function** ILQR(X, U)
4: $J \leftarrow$ Using X, U
5: **repeat**
6: $J^- \leftarrow J$
7: $K, d, \Delta V \leftarrow$ BACKWARDPASS(X, U)
8: $X, U, J \leftarrow$ FORWARDPASS($X, U, K, d, \Delta V, J^-$)
9: **until** $|J - J^-| \leq$ tolerance
10: **return** X, U, J
11: **end function**

IV. NONLINEAR MODEL PREDICTIVE CONTROL

Using the approaches in [6], [7], and [8] as a basis we developed a nonlinear MPC. The cost function used in the Nonlinear model predictive controller is listed below.

$$J = \sum_{i=k}^{k+p} (\hat{X} - \vec{X_{ref}})^T Q_{mpc} (\hat{X} - \vec{X_{ref}}) + \vec{U}^T R_{mpc} \vec{U} \tag{37}$$

$$Q_{mpc} = \begin{bmatrix} 10 I_{3\times3} & 0_{3\times3} \\ 0_{3\times3} & 0_{3\times3} \end{bmatrix} \tag{38}$$

$$R_{mpc} = \begin{bmatrix} 0.2 & 0 \\ 0 & 0.2 \end{bmatrix} \tag{39}$$

The MPC subscript is used to indicate a difference with R, Q used in section III. We used the typical MPC object in the Matlab optimal control toolbox with limits on the manipulated variables. In this setup fmincon is used to solve the nonlinear programming problem to find the optimal input. $\hat{X}$ refers to the optimal state estimate from the Extended Kalman Filter (EKF), as described in section V.

V. Extended Kalman filtering

An Extended Kalman Filter (EKF) was employed to estimate the state of the target that would be recursively fed into the MPC controller. The MPC takes in the optimal state trajectory from the iLQR, $\vec{X_{ref}}$, and the estimate from the EKF, $\hat{X}$, to minimize the cost function in (41).

A summary of the EKF is given in the table below. The rationale for the algorithm is that the chosen dynamics were non-linear, making the traditional Kalman Filter infeasible for this problem. As a brief overview, we begin with an initial state and state uncertainty estimate. These are 6x1 and 6X6 matrices which represent our state vector (X) and the state covariance matrix (P), respectively. The state estimate is propagated by the dynamics and the state covariance matrix is propagated by the continuous-time Ricatti equation using a linearized dynamics matrix, F. Both our state and state covariance were propagated using ODE45 from one discrete time-step to the next. With the inclusion of a new set of measurements, our propagated estimate and covariance is updated at each discrete time step by a Kalman Gain term, K. The Kalman Gain term comes from minimizing the uncertainty of our covariance matrix, giving us the best possible updated estimate from our measurement [9] [10]. This updated estimate is then fed into the MPC at each time. We use a variety of different initial estimates in a 100 run Monte-Carlo simulation to show that they all converge well within our 3-sigma uncertainty bounds.

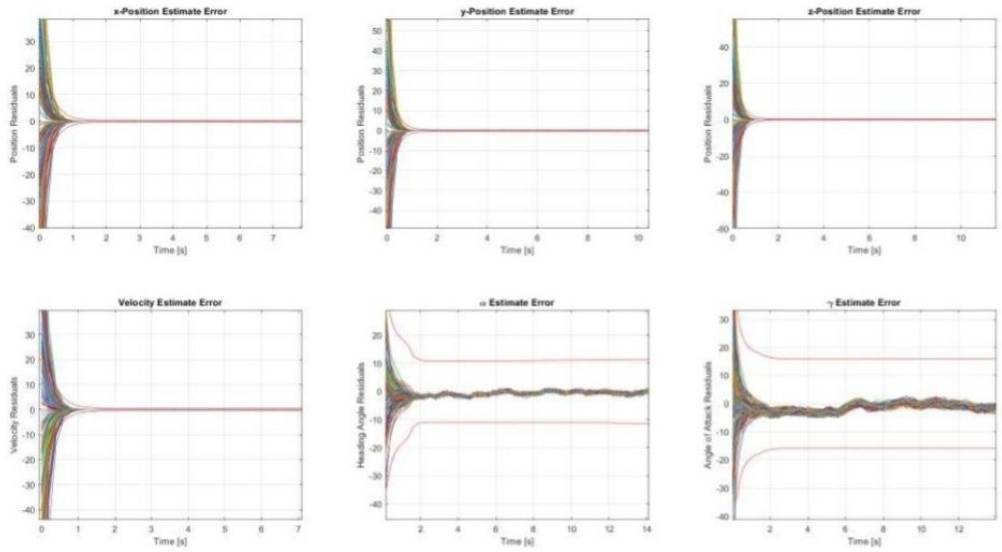


Fig. 2: Monte-Carlo of EKF Residuals

The measurement function was assumed to be linear, where we have the capability to measure each state directly with some measurement noise, which is captured by the R matrix. The true system dynamics were the set of equations used in (1) with the inclusion of the gust model in (7). In order to produce a reliable state estimate, we must take into account the uncertainties in the physical system, which define the process noise values, captured by the Q matrix.

Continuous-Discrete EKF Summary	
System Dynamics	$\dot{\mathbf{x}} = \mathbf{f}(x(t), t) + G(t)\mathbf{w}(t)$
Measurement Model	$\mathbf{z}_k = \mathbf{h}(x_k) + \mathbf{v}_k$
Initial Conditions	$\hat{\mathbf{x}} = \mathbf{f}(x(t), t) + G(t)\mathbf{w}(t)$ $\hat{\mathbf{x}}_0 = E[\mathbf{x}_0]$ $P_0 = E\left[(\mathbf{x}(t_0) - \hat{\mathbf{x}}_0)\,(\mathbf{x}(t_0) - \hat{\mathbf{x}}_0)^\intercal\right]$
Propagation	$\dot{\hat{\mathbf{x}}}_0 = \mathbf{f}(\hat{\mathbf{x}}(t), t)$ $\dot{P}(t) = F^\intercal(\hat{\mathbf{x}}, t)P(t) + P(t)F^\intercal(\hat{\mathbf{x}}, t) + G(t)Q_s(t)G^\intercal(t)$
Gain	$W_k = H_k(\hat{\mathbf{x}}_k^-)P_k^- H_k^\intercal(\hat{\mathbf{x}}_k^-) + R_k$ $C_k = P_k^- H_k^\intercal(\hat{\mathbf{x}}_k^-)$ $K_k = C_k W_k^{-1}$
Update	$\hat{\mathbf{z}}_k = \mathbf{h}_k(\mathbf{x}_k^-)$ $\hat{\mathbf{x}}_k^+ = \hat{\mathbf{x}}_k^- + K_k\,(\mathbf{z}_k - \hat{\mathbf{z}}_k)$ $P_k^+ = P_k^- - C_k K_k^\intercal - K_k C_k^\intercal + K_k W_k K_k^\intercal$

VI. Results

This section will present our results for trajectory optimization, state estimation, and optimal trajectory tracking using NLMPC under wind disturbances.

A. Simulation parameters

TABLE I: General Parameters

Parameter	Description	Value
x_0	Initial position	$[5000, 5000, 5000, 100, 0, \pi]^T$
x_f	Final Position	$[x_{target}, y_{target}, z_{target}, free, free, free]^T$
m	Mass [kg]	150
C_d	Drag Coefficient	0.0169
S	Reference surface area [m^2]	0.0324
u_{max}	Maximum input [m/s^2]	30
u_{min}	Maximum input [m/s^2]	-30

TABLE II: Trajectory Optimization Generation Parameters

Parameter	Description	Value
Q_N	Final state cost (used in both ILQR/DDP and NLP)	$diag[400, 400, 400, 0, 0, 0]$
R	Control Cost (used in ILQR/DDP only)	$diag[0.1, 0.1]$
u_{max}	Maximum input [m/s^2]	30
u_{min}	Maximum input [m/s^2]	-30

TABLE III: NLMPC Parameters

Parameter	Description	Value
Q_{mpc}		$diag[10, 10, 10, 0, 0, 0]$
R_{mpc}	Control Cost	$diag[0.2, 0.2]$
u_{max}	Maximum input [m/s^2]	30
u_{min}	Maximum input [m/s^2]	-30

TABLE IV: EKF Parameters

Parameter	Description	Value
R	Noise Covariance Matrix	$diag[0.025, 0.025, 0.025, 0.025, 0.0345, 0.0349]$
Q	State Noise Matrix	$diag[0.02, 0.02, 0.02, 0.02, 0.0175, 0.0175]$
P_0	Initial State Covariance	$diag[100, 100, 100, 100, 0.0873, 0.0873]$

B. Trajectory Optimization

For trajectory optimization, we simulated our problem using the parameters in tables I and II in the Appendix. where we assumed a point-mass fixed-wing vehicle in coordinated flight with no thrust available. With nonlinear programming, we are able to generate trajectories that meet our constraints for final position, control inputs and obstacle avoidance. The downside is the runtime to obtain such trajectory as in [Fig. 3] is more than 20 seconds. and considering that our entire trajectory is less than 30 seconds, it does not make sense to use in our moving target scenario.

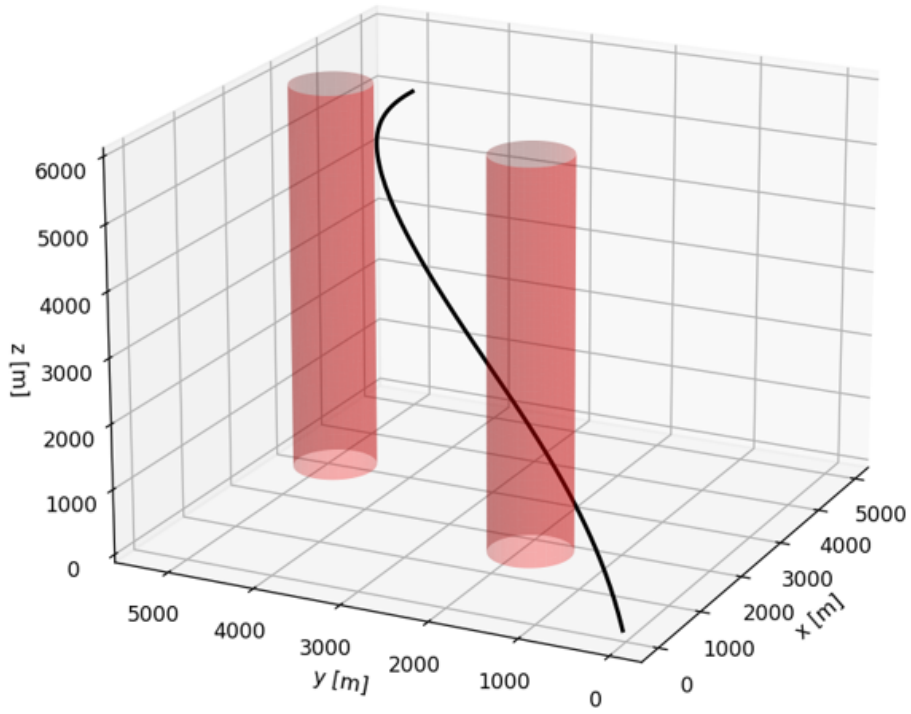
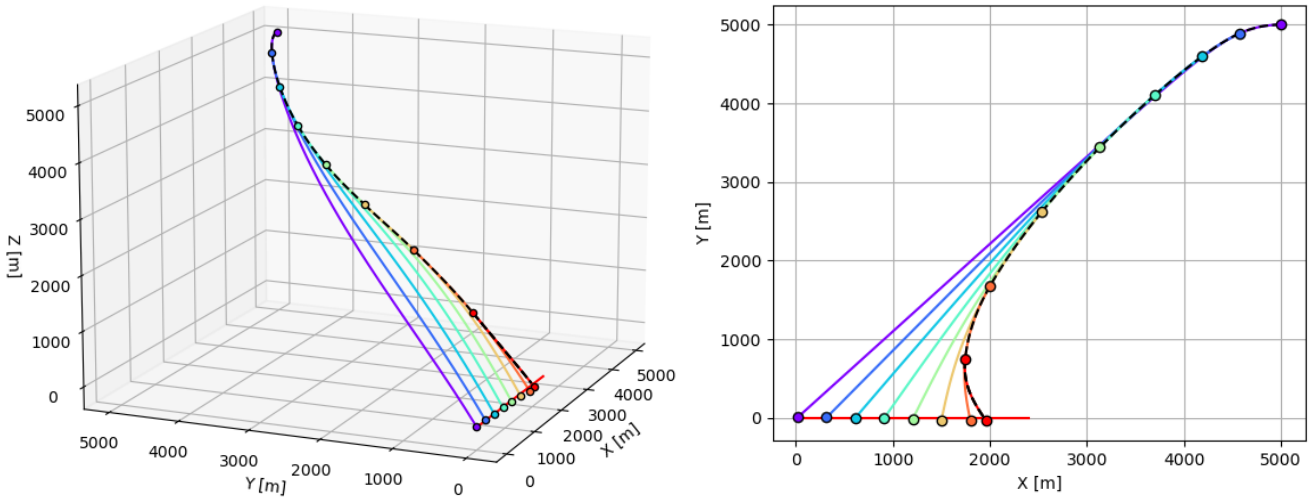


Fig. 3: Nonlinear Programming optimal trajectory in 3D with obstacle avoidance

Using iLQR we were able to simulate a moving target because the optimal trajectory generation time was less than 0.5 seconds while we assumed in our model that new measurements for the target are coming every 5 seconds. As can be seen in [fig. 4], iLQR was able to generate trajectories that lead to reaching the target position. These trajectories are smooth and obey the system dynamics. Here the target is assumed to be moving at a constant speed along the x-axis.



(a) 3D plot of iLQR trajectory

(b) 2D plot of iLQR trajectory

Fig. 4: a 3D and 2D plots of the optimal trajectory generated by iLQR, the colored line represent the trajectory generated each time a measurement of the target is received while the black dashed line represents the final optimal trajectory actually flown by the vehicle

The states behave as expected from the dynamics. Where the vehicle will drop altitude in order to gain speed then it levels off so it is able to fly for a larger range. this is reflected in the behavior of γ.

C. NLMPC results

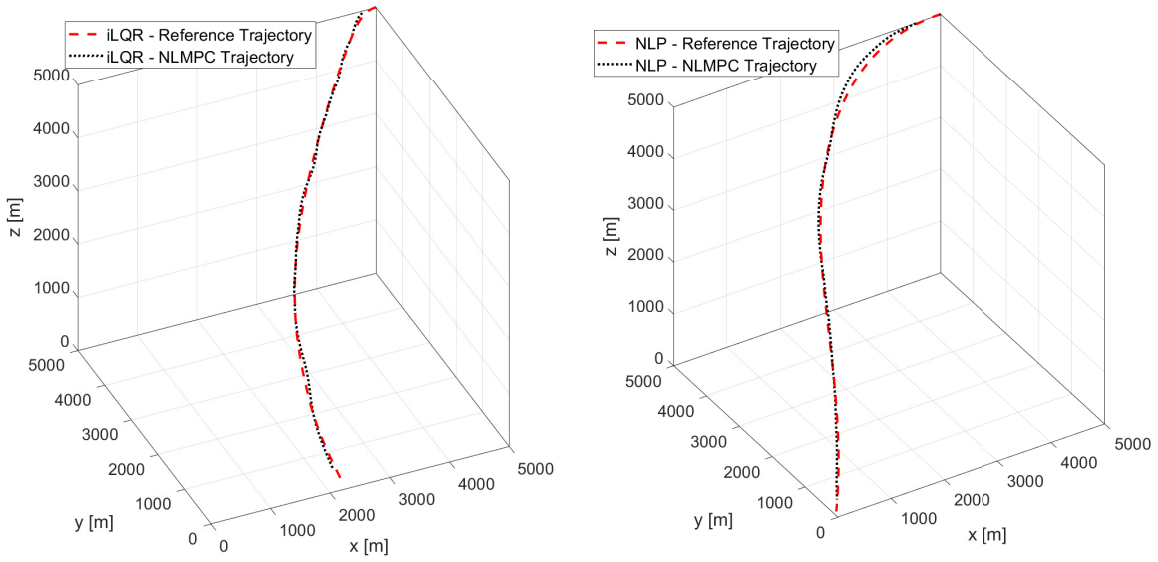


Fig. 5: NLMPC Path Following for iLQR and NLP with wind disturbances

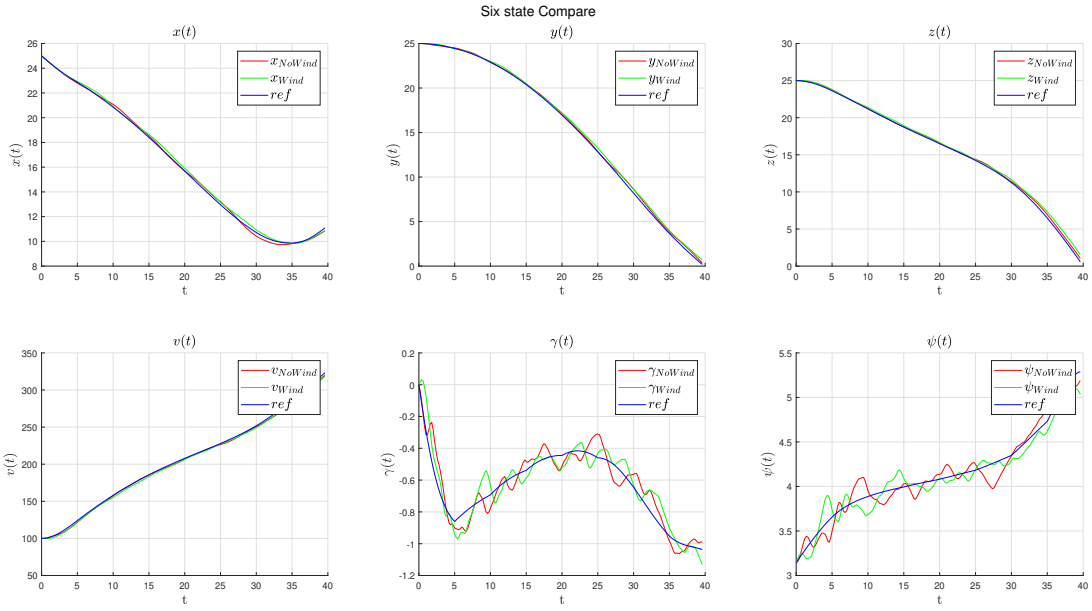


Fig. 6: NLMPC performance with and without wind for all six states

As seen in figure 6 we are able to track the reference states fairly well. However, it should be noted that the version with wind disturbances does not track as well but it still provides an adequate result.

From these results, we show that the objective of reaching our target is met to a satisfactory degree. We show that the nonlinear program was able to hit a fixed target while avoiding physical obstacle constraints. However, such a method was computationally expensive as it uses a similar approach to dynamic programming. This agreed with the results of [2] as it was the same set-up as their paper and they assumed the trajectories were computed offline. This also makes it infeasible for a tracking a moving target, since quick on-board trajectory generation would be required. This issue is solved using an iLQR which was implemented in Python, and we show that we can generate optimal trajectories in less than 0.5 seconds. This now makes it feasible to be implemented into an on-board system, as the time to solve for the desired trajectory is much less than the time interval to new information on the target.

VII. Conclusion

In this report, we developed and simulate a guidance and control framework for a gliding vehicle to intercept a moving target on the ground. We use a simplified non-linear point mass model of the dynamics for the system, which traditionally require more sophisticated and computationally intensive control techniques. While dynamic programming and the two point boundary value method should result in the correct trajectories, they would be completely infeasible to implement real-world application due to computational costs. Thus, we leverage the capability of differential dynamic programming to show that a reference optimal trajectory can be computed in less than 0.5 seconds. This demonstrates the potential for real world applications, as the guidance law can be computed on-board, thereby addressing a limitation of traditional optimal control formulations. To test our system, we add perturbations to our dynamics in the form of a gust model. We used a non-linear model predictive control algorithm as a path tracking controller to account for these disturbances and ensure we are able to follow the reference trajectory over the course of our simulation. Finally, we preform state estimation on our vehicle in the form of an Extended Kalman Filter whose estimates are fed into our NLMPC algorithm.

Based on our results, we show that we are able to intercept a moving target on the ground with an accuracy of 100 meters. For a fixed target, the intersection distance was negligible. This shows that our control and estimate preformed well, as we were able to reasonably reach our final state with realistic computational constraints. There is plenty of opportunity to expand on this effort by addressing some limitations from our report. Namely, one could use a more realistic dynamic model which takes into account the complex coupled rotational dynamics of the vehicles via Euler's Equations of motion. This would make the problem significantly more complicated and would test the

robustness of the iLQR solver. The measurement function of the EKF could also be more realistic by incorporating physical sensor data. IMU sensors, rate integrating gyros, and GPS sensors would reveal non-linearities of the measurements, which would add more computational constraints from the EKF.

VIII. Contributions of Individual Members

- **Ahmad Musallam**: Reference trajectory optimization using Nonlinear Programming and DDP, problem formulation, and literature review.
- **Michael Kerley**: Nonlinear MPC design, problem formulation, system simulation in Matlab/Simulink, and literature review.
- **Aadi Duggal**: State estimation using EKF, problem formulation, optimal trajectory formulation, and literature review.

Acknowledgments

We would like to thank the AAE 568 TA team for providing technical feedback and directions for the project. Their knowledge and engagement enhanced our learning experience and made the scope of this project possible.

References

[1] D. MAYNE. A second-order gradient method for determining optimal trajectories of non-linear discrete-time systems. *International Journal of Control*, 3(1):83–95, 1966.

[2] S. He X. Zheng and D Lin. Constrained trajectory optimization with flexible final time for autonomous vehicles. *IEEE TRANSACTIONS ON AEROSPACE AND ELECTRONIC SYSTEMS*, 58(3), 2022.

[3] Hongying Wu, Nayibe Chio Cho, Hakim Bouadi, Lunlong Zhong, and Felix Mora-Camino. Dynamic programming for trajectory optimization of engine-out transportation aircraft. In *2012 24th Chinese Control and Decision Conference (CCDC)*, pages 98–103, Taiyuan, China, May 2012. IEEE.

[4] Joao Fonseca Rui Dilao. Dynamic trajectory control of gliders. *Proceedings of the EuroGNC 2013*, 2013.

[5] H.-S. Shin S. He and A. Tsourdos. Computational guidance using sparse gauss-hermite quadrature differential dynamic programming. *IFAC-PapersOnLine*, 52(12):13–18, 2019.

[6] Thomas J. Stastny, Adyasha Dash, and Roland Siegwart. Nonlinear MPC for Fixed-wing UAV Trajectory Tracking: Implementation and Flight Experiments. In *AIAA Guidance, Navigation, and Control Conference*, Grapevine, Texas, January 2017. American Institute of Aeronautics and Astronautics.

[7] Yeonsik Kang and J. Hedrick. Design of Nonlinear Model Predictive Controller for a Small Fixed-Wing Unmanned Aerial Vehicle. In *AIAA Guidance, Navigation, and Control Conference and Exhibit*, Keystone, Colorado, August 2006. American Institute of Aeronautics and Astronautics.

[8] Merrill Edmonds and Jingang Yi. A Model Predictive Control Based Iterative Trajectory Optimization Method for Systems with State-Like Disturbances. In *2019 American Control Conference (ACC)*, pages 1635–1640, Philadelphia, PA, USA, July 2019. IEEE.

[9] Maria V Kulikova and Gennady Yu Kulikov. Square-root accurate continuous-discrete extended kalman filter for target tracking. In *52nd IEEE Conference on Decision and Control*, pages 7785–7790. IEEE, 2013.

[10] Louis Tonc and Geordie Richards. Orbit estimation from angles-only observations using nonlinear filtering schemes. 2019.

[11] Kwangjin Yang, Salah Sukkarieh, and Yeonsik Kang. Adaptive nonlinear model predictive path tracking control for a fixed-wing unmanned aerial vehicle. In *AIAA Guidance, Navigation, and Control Conference*, page 5622, 2009.

[12] Arturo De Marinis, Felice Iavernaro, and Francesca Mazzia. A minimum-time obstacle-avoidance path planning algorithm for unmanned aerial vehicles. *Numer Algor*, 89(4):1639–1661, April 2022.

[13] Kwangjin Yang, Salah Sukkarieh, and Yeonsik Kang. Adaptive Nonlinear Model Predictive Path Tracking Control for a Fixed-Wing Unmanned Aerial Vehicle. In *AIAA Guidance, Navigation, and Control Conference*, Chicago, Illinois, August 2009. American Institute of Aeronautics and Astronautics.

[14] Changkoo Kang and Craig A. Woolsey. Model-based path prediction for fixed-wing unmanned aircraft using pose estimates. *Aerospace Science and Technology*, 105:106030, October 2020.

[15] Jie Chen, Zhiwei Shi, Xi Geng, Qijie Sun, Quanbing Sun, and Zhenquan Yin. Optimal trajectory planning of a small UAV using solar energy and wind energy. *AIP Advances*, 13(5):055134, May 2023.

[16] Christopher Cotting and Timothy Cox. A Generic Guidance and Control Structure for Six-Degree-of-Freedom Conceptual Aircraft Design. In *43rd AIAA Aerospace Sciences Meeting and Exhibit*, Reno, Nevada, January 2005. American Institute of Aeronautics and Astronautics.

[17] Xiaobo Zheng, Shaoming He, and Defu Lin. Constrained Trajectory Optimization With Flexible Final Time for Autonomous Vehicles. *IEEE Trans. Aerosp. Electron. Syst.*, 58(3):1818–1829, June 2022.

[18] Rui Dilão and João Fonseca. Dynamic Trajectory Control of Gliders. In Qiping Chu, Bob Mulder, Daniel Choukroun, Erik-Jan Van Kampen, Coen De Visser, and Gertjan Looye, editors, *Advances in Aerospace Guidance, Navigation and Control*, pages 373–386. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[19] Yiming Zhao and Panagiotis Tsiotras. Time-Optimal Parameterization of Geometric Paths for Fixed-Wing Aircraft.

[20] Chen Li, Zhenglong Luo, Difei Xu, Weihong Wu, and Ziheng Sheng. Online Trajectory Optimization for UAV in Uncertain Environment. In *2020 39th Chinese Control Conference (CCC)*, pages 6996–7001, Shenyang, China, July 2020. IEEE.

APPENDIX

A. iLQR Code

```
import numpy as np


class iLQR:
    def __init__(self,dynamics,params,x0,xf,us):
        self.dynamics = dynamics
        self.parameters = {
                    'alphas'  : 0.5**np.arange(8), #line search candidates
                    'regu_init': 20,    #initial regularization factor
                    'max_regu' : 10000,
                    'min_regu' : 0.001
        }
        self.params = params
        self.x0 = x0
        self.xf = xf
        self.us = us

    def optimize(self):
        self.xs, self.us, self.cost_trace = run_ilqr(self.dynamics,self.params['dt'],self.x0,
    self.xf,self.us,self.parameters['regu_init'],self.params['Q_k'],self.params['Q_T'],self.
    params['R_k'],self.parameters['alphas'],self.params['nt'],self.parameters['min_regu'],self.
    parameters['max_regu'])
        return self.xs, self.us, self.cost_trace


def run_ilqr(dynamics,dt, x0, xf, u_init, regu_init, Q, Qf, R, alphas, max_iters, min_regu,
    max_regu):
    '''
       iLQR main loop
    '''
    us = u_init
    regu = regu_init
    # First forward rollout
    xs, J_old = rollout(dynamics.f, us, x0, xf, Q, Qf, R,dt)
    # cost trace
    cost_trace = [J_old]

    # Run main loop
    for it in range(max_iters):
        ks, Ks, exp_cost_redu = backward_pass(us,xs,xf,dynamics,dt,regu,Q,Qf,R)
        # Early termination if improvement is small
        if it > 3  and np.abs(exp_cost_redu) < 1e-5: break

        # Backtracking line search
        for alpha in alphas:
          xs_new, us_new, J_new = forward_pass(dynamics.f, dt, Q, Qf, R, xf, xs, us, ks, Ks,
    alpha)
          if J_old - J_new > 0:
              # Accept new trajectories and lower regularization
              J_old = J_new
              xs = xs_new
              us = us_new
              regu *= 0.7
              break
        else:
            # Reject new trajectories and increase regularization
            regu *= 2.0

        cost_trace.append(J_old)
        regu = min(max(regu, min_regu), max_regu)

    return xs, us, cost_trace
```

```
58
59
60 def rollout(f, us, x0, xf, Q, Qf, R, dt):
61     '''
62       Rollout with initial state and control trajectory
63     '''
64     xs = np.empty((us.shape[0] + 1, x0.shape[0]))
65     xs[0,:] = x0
66
67     for n in range(us.shape[0]):
68       xs[n+1,:] = f(xs[n,:], us[n,:],dt)
69
70     cost = Cost(xs,us,xf, Q, Qf, R)
71     return xs, cost
72
73
74
75 def forward_pass(f,dt, Q, Qf, R, xf, xs, us, ks, Ks, alpha):
76     '''
77        Forward Pass
78     '''
79     xs_new = np.empty(xs.shape)
80
81     cost_new = 0.0
82     xs_new[0,:] = xs[0,:]
83     us_new = us + alpha*ks
84
85     for n in range(us.shape[0]):
86         us_new[n,:] += Ks[n,:,:].dot(xs_new[n,:] - xs[n,:])
87         xs_new[n + 1,:] = f(xs_new[n,:], us_new[n,:],dt)
88
89     cost_new = Cost(xs_new, us_new,xf, Q, Qf, R)
90
91     return xs_new, us_new, cost_new
92
93
94 def backward_pass(us,xs,xf,dynamics,dt,regu,Q,Qf,R):
95     '''
96        Backward Pass
97     '''
98     ks = np.empty(us.shape)
99     Ks = np.empty((us.shape[0], us.shape[1], xs.shape[1]))
100
101    delta_V = 0
102    V_x = Qf@(xs[-1,:] - xf)
103    V_xx = Qf
104    regu_I = regu*np.eye(R.shape[0])
105    for n in range(us.shape[0] - 1, -1, -1):
106
107        f_x = dynamics.dfdx(xs[n,:], us[n,:],dt)
108        f_u = dynamics.dfdu(xs[n,:], us[n,:],dt)
109        l_x = Q@xs[n,:]
110        l_u = R@us[n,:]
111        l_xx = Q
112        l_ux = np.zeros((l_u.shape[0], l_x.shape[0]))
113        l_uu = R
114
115        # Q_terms
116        Q_x  = l_x  + f_x.T@V_x
117        Q_u  = l_u  + f_u.T@V_x
118        Q_xx = l_xx + f_x.T@V_xx@f_x
119        Q_ux = l_ux + f_u.T@V_xx@f_x
120        Q_uu = l_uu + f_u.T@V_xx@f_u
121
122        # gains
123
```

```
        Q_uu += regu_I
        k = -np.linalg.solve(Q_uu, Q_u)
        K = -np.linalg.solve(Q_uu, Q_ux)
        ks[n,:], Ks[n,:,:] = k, K

        # V_terms
        V_x  = Q_x + K.T@Q_u + Q_ux.T@k + K.T@Q_uu@k
        V_xx = Q_xx + 2*K.T@Q_ux + K.T@Q_uu@K
        #expected cost reduction
        delta_V += Q_u.T@k + 0.5*k.T@Q_uu@k

    return ks, Ks, delta_V


def Cost(x,u,xf,Q,Qf,R):
    cost = 0
    for i in range(u.shape[0]):
        cost += x[i,:].T@Q@x[i,:] + u[i,:].T@R@u[i,:]
    cost += (x[-1,:]-xf).T@Qf@(x[-1,:]-xf)
    return cost
```

B. Direct Method Code (NLP)

```
    # model Time Optimal Control using JuMP
using JuMP
import Ipopt
import Plots

# Create JuMP model, using Ipopt as the solver
model = Model(Ipopt.Optimizer)
n = 200    # Time steps

# model vehicle parameters
u_max = 30  # Maximum Angular Input
m = 150;
g = 9.8;
S = 0.0324;
Cd = 0.0169;
s_p = 200;

# Decision variables
@variables(model, begin
     t      0  # Time step
    # State variables
    30     x[1:n]     -1         # x-coordinate
    30     y[1:n]     -1         # y-coordinate
    30     z[1:n]     -1           # z-coordinate
    600     V[1:n]     200       # Velocity
    gama[1:n]                  # Flight path angle
    psi[1:n]                   # Heading angle
    # Control variables
    -u_max     a_y[1:n]     u_max    # Thrust
    -u_max     a_z[1:n]     u_max    # Thrust

end);


# Objective: Minimize time
@objective(model, Min,  t )

# Initial conditions
@NLconstraint(model, x[1] == 25);
@NLconstraint(model, y[1] == 25);
@NLconstraint(model, z[1] == 25);
@NLconstraint(model, V[1] == 280);
```

```
@NLconstraint(model, gama[1] == 0);
@NLconstraint(model, psi[1] == pi);

# Final time constraints
xf = 0.0;
yf = 0.0;
zf = 0.0;
gamaf = -pi/3;
psif = 270/180*pi;
@NLconstraint(model, (x[n] - xf)^2 + (y[n] - yf)^2 + (z[n] - zf)^2     2)
#@NLconstraint(model, (gama[n] - gamaf)^2 + (psi[n] - psif)^2      0.1)

# Helper functions
@NLexpression(model, rho[j = 1:n], 1.15579 - 1.058*10^(-4)*z[j]*s_p + 3.725*10^(-9)*(z[j]*s_p)
    ^2 - 6*10^(-14)*(z[j]*s_p)^3);
@NLexpression(model, D[j= 1:n], 0.5*rho[j] * V[j]^2 * S * Cd);

# Dynamics trapozoidal rule
@NLexpression(model,  x [j=1:n], V[j] * cos(gama[j]) * cos(psi[j]) / s_p);
@NLexpression(model,  y [j=1:n], V[j] * cos(gama[j]) * sin(psi[j]) / s_p);
@NLexpression(model,  z [j=1:n], V[j] * sin(gama[j]) / s_p);
@NLexpression(model,  V [j=1:n], -D[j] / m - g * sin(gama[j]));
@NLexpression(model,  gama [j=1:n],(-a_z[j] - g * cos(gama[j]))/ V[j]);
@NLexpression(model,  psi [j=1:n], a_y[j]/(V[j] * cos(gama[j])));

for j in 2:n
    i = j - 1
    @NLconstraint(model, x[j] == x[i] + 0.5 *  t  * ( x [i] +  x [j]));
    @NLconstraint(model, y[j] == y[i] + 0.5 *  t  * ( y [i] +  y [j]));
    @NLconstraint(model, z[j] == z[i] + 0.5 *  t  * ( z [i] +  z [j]));
    @NLconstraint(model, V[j] == V[i] + 0.5 *  t  * ( V [i] +  V [j]));
    @NLconstraint(model, gama[j] == gama[i] + 0.5 *  t  * ( gama [i] +  gama [j]));
    @NLconstraint(model, psi[j] == psi[i] + 0.5 *  t  * ( psi [i] +  psi [j]));
end


#Constraints for not passing through circles
rc = 500/s_p;
xc = [1800/s_p 2500/s_p 2500/s_p];
yc = [1800/s_p 4800/s_p 1800/s_p];
delta_dist = 100/s_p;
for i = 1:n
    @NLconstraint(model, (x[i] - xc[1])^2 + (y[i] - yc[1])^2     rc^2)
    @NLconstraint(model, (x[i] - xc[2])^2 + (y[i] - yc[2])^2     rc^2)
    @NLconstraint(model, (x[i] - xc[3])^2 + (y[i] - yc[3])^2     rc^2)
#    @NLconstraint(model, (x[i] - xc[2])^2 + (y[i] - yc[2])^2     rc^2)
end;

# Solve for the control and state
println("Solving...")
optimize!(model)
solution_summary(model)

## Display results
println("Time required: ", n * objective_value(model))
plt1 = Plots.plot(value.(x), value.(y),value.(z), labe="3D-path", aspect_ratio = 1, legend =
    false, framestyle = :box)
plt3 = Plots.plot(value.(V), label = "Velocity")
plt4 = Plots.plot(value.(z), label = "Altitude")
plt5 = Plots.plot(value.(x), value.(y), label="xy-path", aspect_ratio = 1, framestyle = :box)
# Circles
xt = zeros(n)
yt = zeros(n)
 t  = LinRange(0, 2  , n)
for j = 1:2
  global plt5
```

```
    for i = 1:n
        xt[i] = xc[j] + rc * cos( t [i])
        yt[i] = yc[j] + rc * sin( t [i])
    end
    Plots.plot!(plt5, xt, yt)
end

#Plotting final and initial locations
#Plots.scatter!(plt1, [0, xf], [0, yf])

display(plt1)
display(plt3)
display(plt4)
display(plt5)
```

C. EKF Code

```
% Inputs:
% zm:  Measurement at t
% xestInput: State Estimate at t-1
% PestInput: Covariance Estimate at t-1
% t, dt: Time index and dt
% pos_meas, angle_meas: 1 sigma STD of position and angle measurements
% sig_pos, sig_angle: 1 sigma STD of true state position and angles
%
% Outputs:
% xestPost: Posterior Estimate at time t
% PestPost: Posterior Covariance at time t
function [xestPost, PestPost] = full_state_EKF(zm, u, xestInput, PestInput, t,
    dt, pos_meas, angle_meas, sig_pos, sig_angle)

%Constants
m = 150;
g = 9.8;
S = 0.0324;
s_p = 200;
Cd = 0.0169;


R = diag([pos_meas^2 pos_meas^2 pos_meas^2 pos_meas^2 angle_meas^2 angle_meas
   ^2]);
H = eye(6);
% zm =  [x(4,:) + normrnd(0, pos_meas, [1, length(x(1,:))]);
%        x(5,:) + normrnd(0, angle_meas, [1, length(x(1,:))]);
%        x(6,:) + normrnd(0, angle_meas, [1, length(x(1,:))])];
% R = diag([pos_meas angle_meas angle_meas]);
% H = [0 0; 0 1]

rng(69420);
options = odeset('RelTol',1e-10,'AbsTol',1e-10);
Q = diag([sig_pos; sig_pos; sig_pos; sig_pos; sig_angle; sig_angle]);

%Estimate Propagation
xs = xestInput;
```

```
Ps = reshape(PestInput, [numel(PestInput),1]);
s = [xs; Ps];



tDom = linspace(t- dt, t, 5);
[~, s] = ode45(@(t, s) PropState(s, u, m, g, S, s_p, Q), tDom, s, options);
s = s';
state_pri = s(:,end);
xpri = state_pri(1:6);
Ppri = reshape(state_pri(7:end), [6,6]);

%Kalman Gain
p = 5/7;
C = Ppri*H';
if trace(H*Ppri*H') > p/(1-p)*trace(R)
    W = (1/p*H*Ppri*H' + R);
    K = C/(1/p*H*Ppri*H' + R);
else
    W = H*Ppri*H' + R;
    K = C/W;
end

%Measurememt and Update
zest = H*xpri;
xestPost = xpri + K*(zm-zest);
%Pest = Ppri - C*K' - K*C' + K*W*K';
PestPost = (eye(6) - K*H)*Ppri;
%Pest = validateCovMatrix(Pest);


function [sdot] = PropState(s, u, m, g, S, s_p, Q)
x = s(1); y = s(2); z = s(3);
v = s(4); gamma = s(5); psi = s(6);
P = reshape(s(7:end), [6,6]);
ay = u(1); az = u(2);

rho = 1.15579 - 1.058*10^(-4)*z*s_p + 3.725*10^(-9)*(z*s_p)^2 - 6*10^(-14)*(z*
    s_p)^3; % air density model
Cd = 0.0169;
D = 0.5 .* rho * v^2 * S * Cd;

dxdt =[v .* cos(gamma).* cos(psi) / s_p;
    v .* cos(gamma).* sin(psi) / s_p;
    v .* sin(gamma) / s_p;
    -D ./ m - g .* sin(gamma);
    ( -ay - g .* cos(gamma) ) ./ v;
      az ./ ( v .* cos(gamma) )];


F=  [0, 0,                                                                  0,
                                                                (cos(gamma)*cos(psi)
```

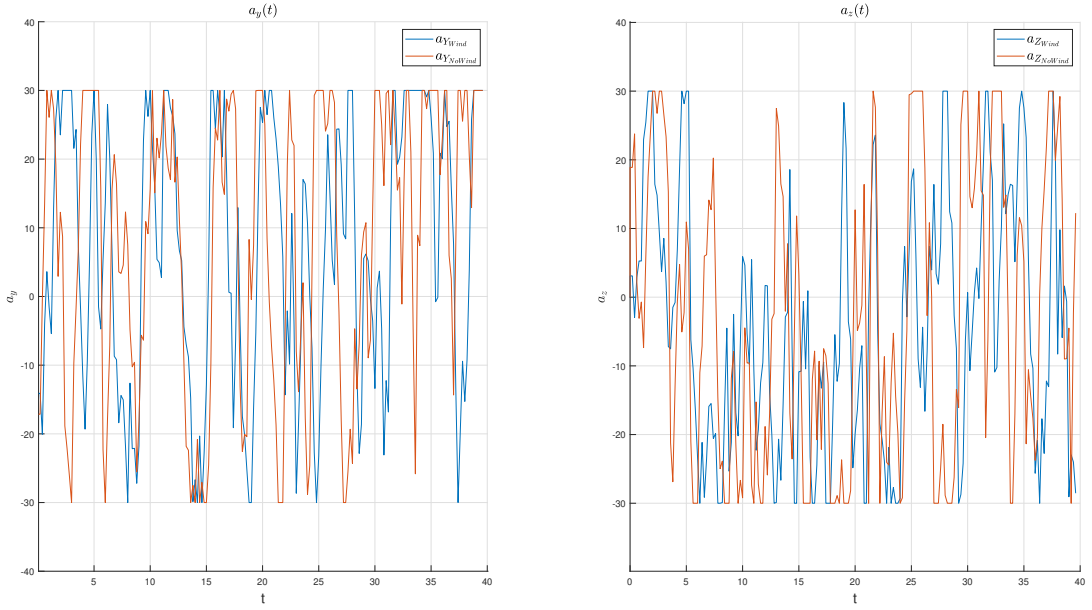
```
    )/s_p, -(1.0*v*cos(psi)*sin(gamma))/s_p, -(1.0*v*cos(gamma)*sin(psi))/s_p;
      0, 0,                                                                 0,
                                                          (cos(gamma)*sin(psi)
    )/s_p, -(1.0*v*sin(gamma)*sin(psi))/s_p,       (v*cos(gamma)*cos(psi))/s_p;
      0, 0,                                                                 0,
                                                                  sin(gamma
    )/s_p,                (v*cos(gamma))/s_p,                              0;
      0, 0, (Cd*S*v^2*(9.0e-14*s_p^3*z^2 - 3.73e-9*s_p^2*z + 5.29e-5*s_p))/m,
    -(2.0*Cd*S*v*(- 3.0e-14*s_p^3*z^3 + 1.86e-9*s_p^2*z^2 - 5.29e-5*s_p*z +
    0.578))/m,                 -1.0*g*cos(gamma),
     0;
      0, 0,                                                                 0,
                                                            (az + g*cos(gamma)
    )/v^2,                  (g*sin(gamma))/v,                               0;
      0, 0,                                                                 0,
                                                         -(1.0*ay)/(v^2*cos(
    gamma)), (ay*sin(gamma))/(v*cos(gamma)^2),
    0];
Pdot = F*P + P*F' + Q;
Pdot = reshape(Pdot, [numel(Pdot),1]);
sdot = [dxdt; Pdot];
end

end
```

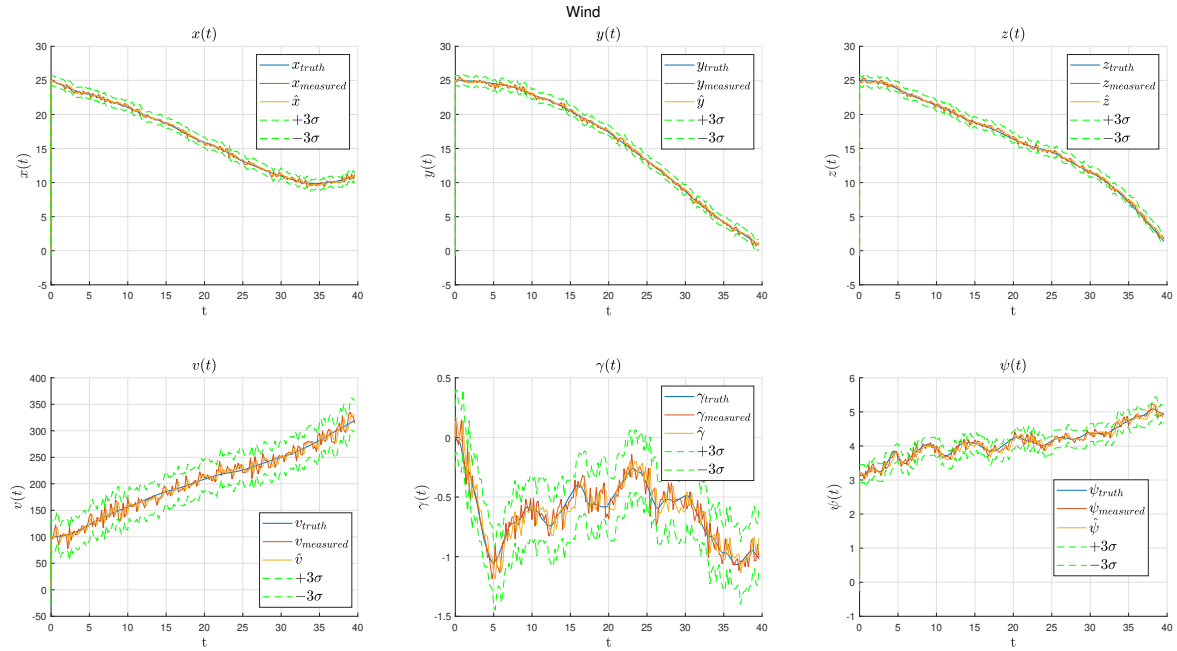
D. *Dynamics*

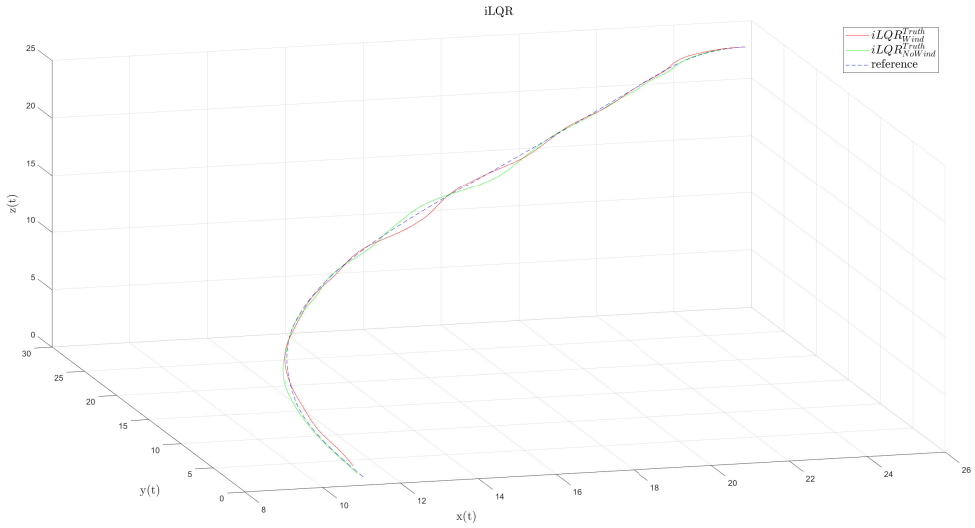
```
    function dxdt = StateFcn(x, u)
    % State equations of the flying robot.
    %
    % States:
    %   x(1)  x inertial coordinate of center of mass
    %   x(2)  y inertial coordinate of center of mass
    %   x(3)  z inertial coordinate of center of mass
    %   x(4)  V velocity Magnitude
    %   x(5)  Gamma inclination angle
    %   x(6)  Psi Bank angle

    s_p = 200;
    m = 150;
    g =9.81;

    rho = 1.15579 - 1.058*10^(-4)*x(3)*s_p + 3.725*10^(-9)*(x(3)*s_p)^2 -
    6*10^(-14)*(x(3)*s_p)^3; % air density model
    S = 0.0324;
    Cd = 0.0169;
    D = 0.5 .* rho * x(4)^2 * S * Cd;

    x_dot = x(4) .* cos(x(5)).* cos(x(6)) / s_p;
    y_dot = x(4) .* cos(x(5)).* sin(x(6)) / s_p;
    z_dot = x(4) .* sin(x(5)) / s_p;
```

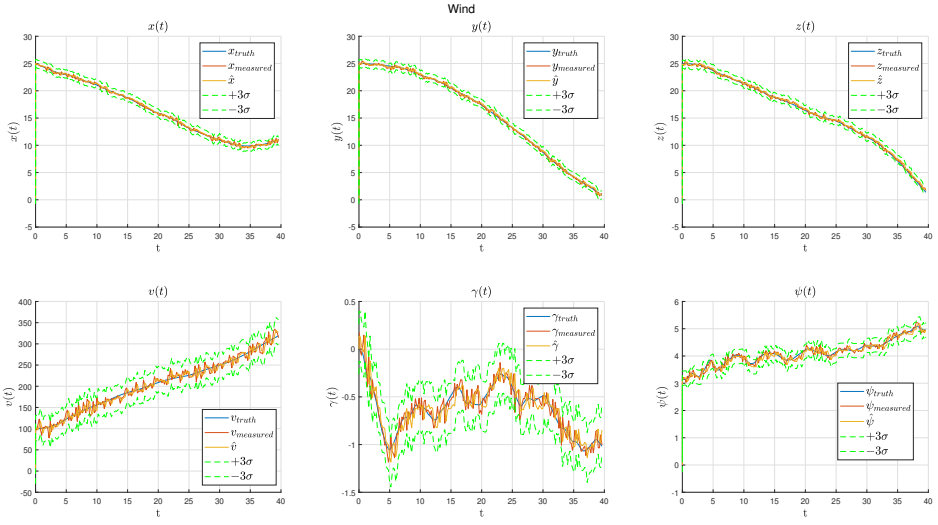
```
    V_dot = -D ./ m - g .* sin(x(5));
    gama_dot = ( -u(2) - g .* cos(x(5)) ) ./ x(4);
    psi_dot = u(1) ./ ( x(4) .* cos(x(5)) );

    dxdt = [x_dot; y_dot; z_dot; V_dot; gama_dot; psi_dot];
end
```

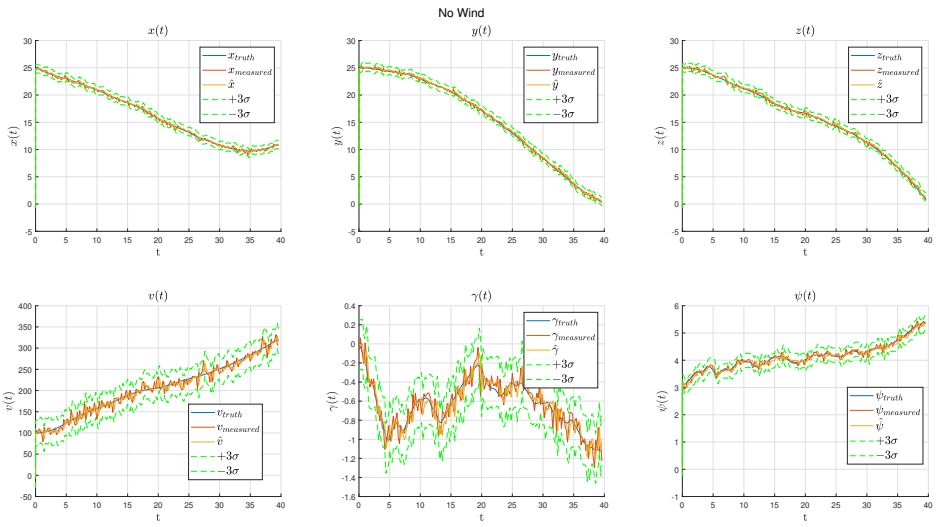
E. Simulink and Plots

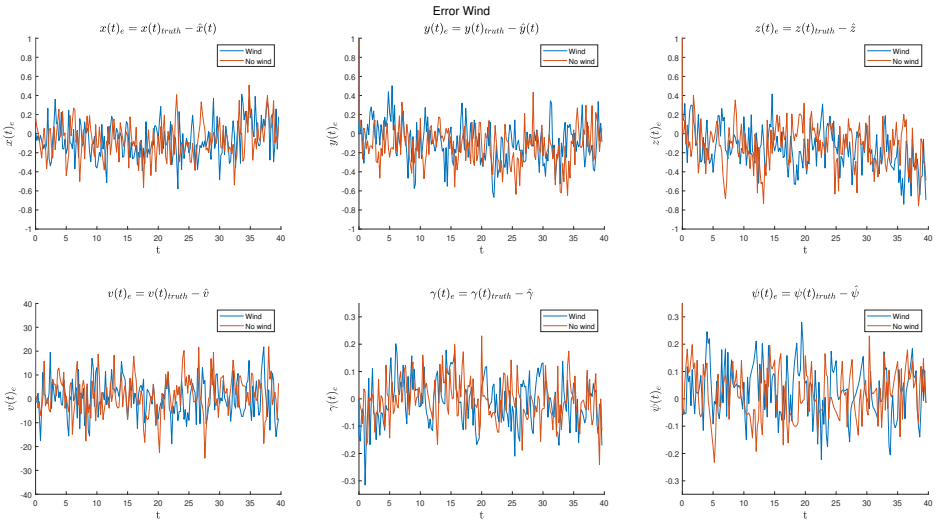




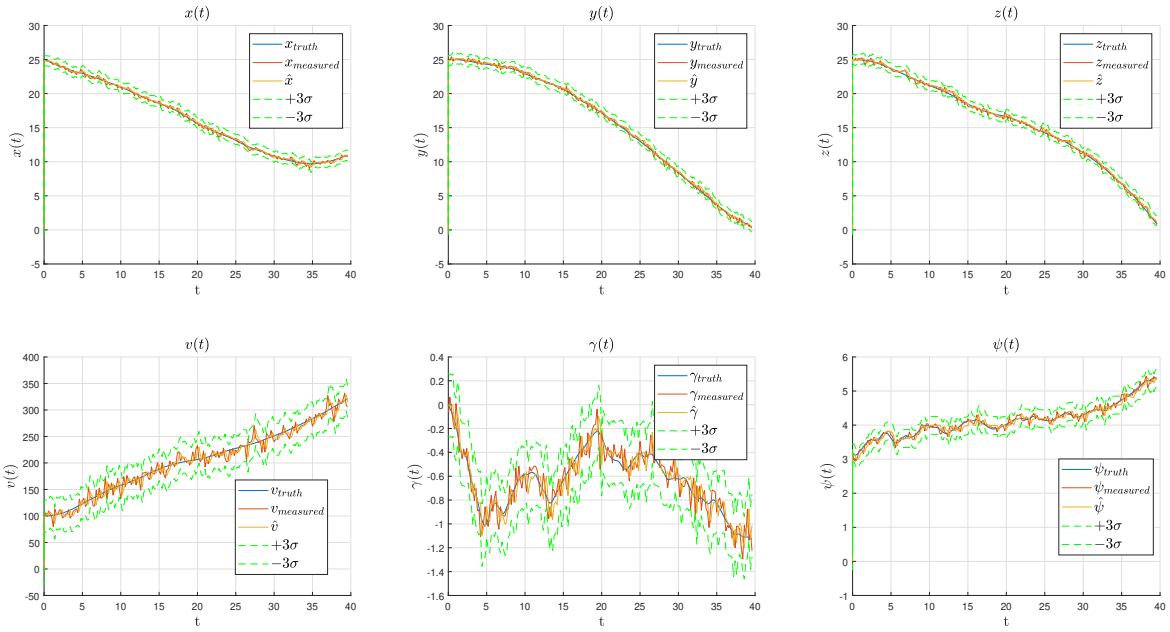

iLQR
$iLQR^{truth}_{Wind}$
$iLQR^{truth}_{noWind}$
reference
z(t)
y(t)
x(t)


Wind
$x(t)$
$y(t)$
$z(t)$
$v(t)$
$\gamma(t)$
$\psi(t)$
x_{truth}
$x_{measured}$
$\hat{x}$
$+3\sigma$
-3σ
y_{truth}
$y_{measured}$
$\hat{y}$
z_{truth}
$z_{measured}$
$\hat{z}$
v_{truth}
$v_{measured}$
$\hat{v}$
γ_{truth}
$\gamma_{measured}$
$\hat{\gamma}$
ψ_{truth}
$\psi_{measured}$
$\hat{\psi}$
t

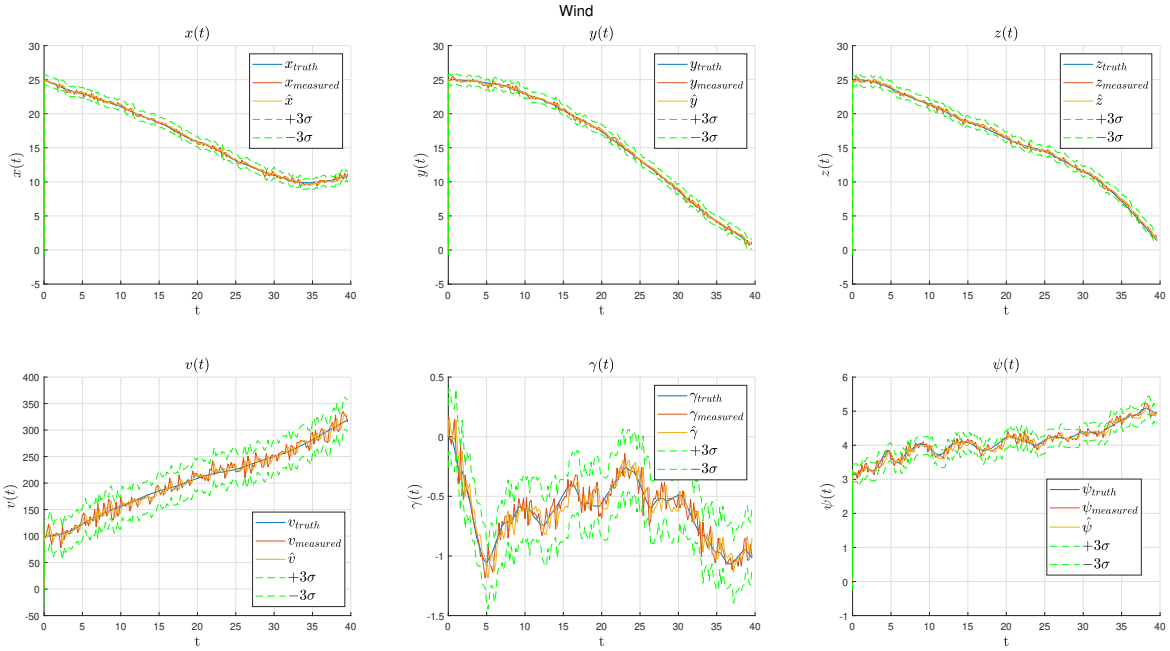

No Wind
$x(t)$
x_{truth}
$x_{measured}$
$\hat{x}$
$+3\sigma$
-3σ
$y(t)$
y_{truth}
$y_{measured}$
$\hat{y}$
$+3\sigma$
-3σ
$z(t)$
z_{truth}
$z_{measured}$
$\hat{z}$
$+3\sigma$
-3σ
$v(t)$
v_{truth}
$v_{measured}$
$\hat{v}$
$+3\sigma$
-3σ
$\gamma(t)$
γ_{truth}
$\gamma_{measured}$
$\hat{\gamma}$
$+3\sigma$
-3σ
$\psi(t)$
ψ_{truth}
$\psi_{measured}$
$\hat{\psi}$
$+3\sigma$
-3σ
t


Error Wind
$x(t)_e = x(t)_{truth} - \hat{x}(t)$
$y(t)_e = y(t)_{truth} - \hat{y}(t)$
$z(t)_e = z(t)_{truth} - \hat{z}$
$v(t)_e = v(t)_{truth} - \hat{v}$
$\gamma(t)_e = \gamma(t)_{truth} - \hat{\gamma}$
$\psi(t)_e = \psi(t)_{truth} - \hat{\psi}$
Wind
No wind
t

No Wind


$x(t)$
x_{truth}
$x_{measured}$
$\hat{x}$
$+3\sigma$
-3σ
$y(t)$
y_{truth}
$y_{measured}$
$\hat{y}$
$+3\sigma$
-3σ
$z(t)$
z_{truth}
$z_{measured}$
$\hat{z}$
$+3\sigma$
-3σ
$v(t)$
v_{truth}
$v_{measured}$
$\hat{v}$
$+3\sigma$
-3σ
$\gamma(t)$
γ_{truth}
$\gamma_{measured}$
$\hat{\gamma}$
$+3\sigma$
-3σ
$\psi(t)$
ψ_{truth}
$\psi_{measured}$
$\hat{\psi}$
$+3\sigma$
-3σ
t

Wind


$x(t)$
x_{truth}
$x_{measured}$
$\hat{x}$
$+3\sigma$
-3σ
$y(t)$
y_{truth}
$y_{measured}$
$\hat{y}$
$+3\sigma$
-3σ
$z(t)$
z_{truth}
$z_{measured}$
$\hat{z}$
$+3\sigma$
-3σ
$v(t)$
v_{truth}
$v_{measured}$
$\hat{v}$
$+3\sigma$
-3σ
$\gamma(t)$
γ_{truth}
$\gamma_{measured}$
$\hat{\gamma}$
$+3\sigma$
-3σ
$\psi(t)$
ψ_{truth}
$\psi_{measured}$
$\hat{\psi}$
$+3\sigma$
-3σ
t