Platform-Agnostic Model Predictive Control

Nikolai Flowers, Ardalan Tajbakhsh, Mike Turski
Department of Mechanical Engineering
Carnegie Mellon University
Pittsburgh, Pennsylvania
Email: {nflowers, atajbakh, mturski}@andrew.cmu.edu

Roshan Pradhan, Roberto Shu
Robotics Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania
Email: {roshanpr, rshu}@andrew.cmu.edu

***Abstract*—In this paper, we present the design and analysis of a platform-agnostic model predictive control (MPC) framework. The MPC is based on formulating a quadratic program (QP) that tracks the error between the desired reference trajectory and the actual robot state. The controller is based on a linear model of the controlled plant. We present *linear_mpc* an open-source library with MATLAB and C++ bindings that implement the QP based MPC proposed in this work. A key aspect of the proposed MPC formulation is that the input reference trajectory does not have to be dynamically feasible. Thus, eliminating the need for a high-level planner. We show on three different platforms (Tumbller, Subterrenean drone, and Ballbot) in simulation and hardware the usage of the library. Additionally, analysis of model predictive state estimator is presented comparing Kalman Filter and moving horizon state estimator. This paper also presents substantial hardware upgrades to the low-cost Tumbller platform to be able to realize the implementation of real-time MPC.**

I. Introduction

Model predictive control (MPC) has been gaining significant traction in real hardware applications in recent years [1], [2], [3]. Recent developments in optimization solver packages have led to enabling the application of MPC in systems with fast dynamics. However, setting up the MPC problem in software still requires some manual implementation which can be prone to user error and issues when interfacing with different solvers. Furthermore, there has been very little attention to the application of optimization-based techniques to the dual estimation problem. This project aims to develop a modular, user-friendly, and platform-agnostic software package that allows developers to prototype model predictive controllers with minimal effort. Additionally, we would like to shed light on the application of optimization-based techniques to the state estimation problem and demonstrate its superior performance compared to the Kalman filter.

II. Problem Description

Implementing MPC on a new robotic platform can be an arduous and time-consuming task; if you're testing out different solvers or different Quadratic Program (QP) formulations it can significantly slow the controller development process if each pairing is accompanied by extensive matrix arithmetic. We're proposing an MPC library that handles much of the backend matrix algebra natively and contains interfaces to a variety of off-the-shelf QP solvers. Using this framework, researchers and engineers will be able to rapidly deploy MPC controllers to new robots as well as efficiently test new combinations of solvers, cost functions, and constraints.

III. Related Work

There are several optimal control toolboxes that aim to provide a modular framework for MPC. Notable open-source examples are ACADO [4] and the Control Toolbox [5]. Though these libraries are applicable to a broad class of dynamical systems, it is very labor-intensive to set up a new system. Further, they are primarily designed for non-linear systems. Our proposed package is aimed at linear systems and leverage the power of a sparse QP formulation.

There has been significant research done on the design and implementation of model predictive controllers on segway-style ground vehicles as well as aerial vehicles. Both of these systems are quite suitable to MPC formulations and some related work is described below -

A. Ballbot

Many ballbot type robots have been built in the past years [6], [7], [8]. A common approach to controlling these underactuated robot platforms is to split the system's motion into two planes and neglect the interaction effects of the full dynamics along the different planes. Effectively linearizing the 3D nonlinear dynamics of the system. Decoupled linear controllers such as PID [9] or LQR [10], [7] are used to stabilize the ballbot along each plane.

Different approaches have been proposed to realize point-to-point motions while maintaining balance. In [11] shape trajectories are computed through an optimization given desired acceleration trajectories in position space. In [12] a differentially flat output of the ballbot is derived and used for trajectory optimization, enabling arbitrary motion. There have also been a few examples of controlling the ballbot position with a MPC framework. In [13] an optimization is formulated to control the highly non-linear underactuated ballbot system by using a linear representation of the system

which does not include the underactuated dynamic properties. Differently, in [14] a framework for real-time, full-state feedback, unconstrained, nonlinear model predictive control that combines trajectory optimization and tracking control in a single, unified approach is presented. Though this approach was shown to be effective to work in drones and ballbots, it required significant effort to implement on each platform.

In this work, we propose the use of the linear closed-loop dynamics of the system and formulate a QP tracking problem that can be solved iteratively in an MPC manner. The aim is to demonstrate the application of the platform-agnostic MPC formulation proposed in this project.

B. Quadcopter

MPC has been a popular research area for quadcopters in recent years [15][16][17]. Quadcopter models are quite simple and have a significant linearizable region of operation about hover condition. This enables the use of efficient linear convex QP solvers to solve both trajectory generation and trajectory tracking problems. In the case of trajectory generation, differential flatness is often exploited to plan in cartesian space instead of configuration space [18], [19] . Our project will focus on trajectory tracking, as is implemented in [20]. Other related work with slight modifications is [21], [22].

IV. System Modeling

A. Tumbller Dynamics

We use the parameterized state space model from Lab 2 to represent our Tumbller dynamics:

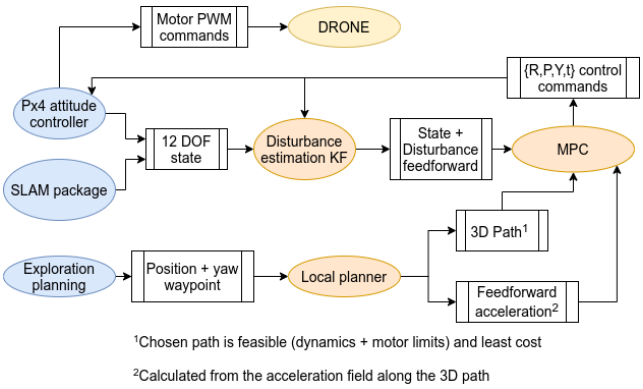
$$\dot{q} = Aq + Bu \tag{1}$$

where A represents the continuous time state update matrix and B represents the control authority matrix.

This model is given in continuous tine, but we convert to discrete time with a sampling time of T_s using MATLAB's *c2d* command with the zero order hold approximation. The state is represented as $q = [x, \dot{x}, \theta, \dot{\theta}]$ and the input is a single horizontal force applied at the base. We convert this force to a motor voltage using the linear transformation $V = \frac{Rr}{2k_T}u$.

B. Drone Dynamics

One of the the MPC implementations we investigate is position control and trajectory tracking for the SubT Drone. It is a custom-built drone weighing 5 kg, with an established autonomy stack and onboard state estimation. The current stack is equipped with a disturbance estimator that learns internal biases and external disturbances and tells the MPC so it can compensate. The control stack is also integrated with the local planner which provides future velocity and acceleration information to enable tight trajectory tracking. In general the



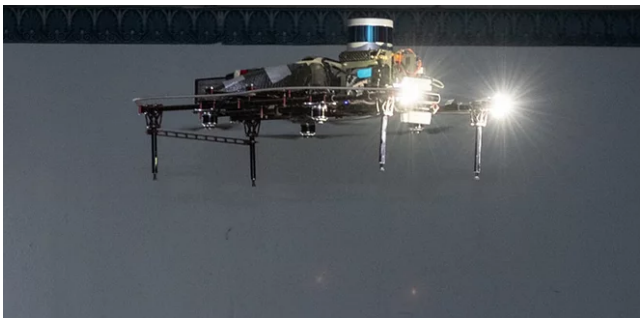


Figure 1. Schematic of SubT stack (top), Drone picture (bottom)

drone dynamics is nonlinear due to the presence of a rotation matrix pre-multiplying the thrust vector.

Notation: Here ψ is yaw, θ is pitch, ϕ is roll, subscript g represents ground frame, x, y, z are in ground frame, T is mass-normalized thrust, dt is MPC sampling time, k and τ represent parameters of a first order system.

$$m\begin{bmatrix}\ddot{x}\\ \ddot{y}\\ \ddot{z}\end{bmatrix} = \begin{bmatrix}0\\0\\-mg\end{bmatrix} + R\begin{bmatrix}0\\0\\Thrust\end{bmatrix} + F_{ext} \tag{2}$$

$$R = R_z(\psi)R_y(\theta)R_x(\phi) \tag{3}$$

$$R = \begin{bmatrix}c\psi c\theta & c\psi s\theta s\phi - s\psi c\phi & c\psi s\theta c\phi + s\psi s\phi\\ s\psi c\theta & s\psi s\theta s\phi + c\psi c\phi & s\psi s\theta c\phi - c\psi s\phi\\ -s\theta & c\theta s\phi & c\theta c\phi\end{bmatrix} \tag{4}$$

The state representation includes the translational position and velocity along with the roll and pitch. Since yaw only affects the heading, it is folded into the roll and pitch terms which are represented in the ground frame.

$$X = \{x, y, z, \dot{x}, \dot{y}, \dot{z}, \phi^g, \theta^g\}^T \tag{5}$$

$$U = \{\phi^g_{cmd}, \theta^g_{cmd}, T_{cmd}\}^T \tag{6}$$

We rely on a linear quadcopter model from ¡ref¿ to represent the dynamics of the drone. The nonlinear equations are linearized by using small-angle approximation and steady state hover assumption ($T \approx g$).

$$\begin{bmatrix}\ddot{x}\\ \ddot{y}\\ \ddot{z}\end{bmatrix} = \begin{bmatrix}T\theta^g\\ -T\phi^g\\ T-g\end{bmatrix} \tag{7}$$

$$\dot{\phi} = (k_\phi \phi_{cmd} - \phi)/\tau_\phi \tag{8}$$

$$\dot{\theta} = (k_\theta \theta_{cmd} - \theta)/\tau_\theta \tag{9}$$

After linearizing the continuous system, we can discretize using zero-order hold approximation to obtain A_d and B_d which we can then pass onto the solver.

$$A_d = \begin{bmatrix} 1 & 0 & 0 & dt & 0 & 0 & 0 & dt^2/(2g) \\ 0 & 1 & 0 & 0 & dt & 0 & -dt^2/(2g) & 0 \\ 0 & 0 & 1 & 0 & 0 & dt & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & gdt \\ 0 & 0 & 0 & 0 & 1 & 0 & -gdt & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 - dt/\tau_\phi & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 - dt/\tau_\theta \end{bmatrix} \tag{10}$$

$$B_d = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & dt^2/2 & -gdt^2/2 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & dt & -gdt \\ k_\phi dt/\tau_\phi & 0 & 0 & 0 \\ 0 & k_\theta dt/\tau_\theta & 0 & 0 \end{bmatrix} \tag{11}$$

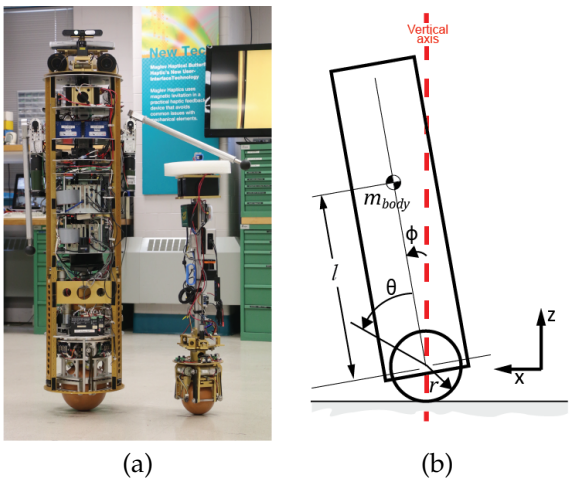
The internal and external un-modeled forces acting on the drone - which turn out to be quite significant - are modeled as control offsets, specifically $\tilde{\phi}$, $\tilde{\theta}$, and $\tilde{T}$. These offsets are estimated in real-time by the disturbance estimation package.

C. Ballbot Dynamics

Another hardware platform used in this project is the CMU ballbot shown in Fig. 2(a). The ballbot class of mobile robots is a unique type of dynamically stable robot that balances and locomotes on a single spherical wheel. Ballbot type robots belong to the family of *shape-accelerated underactuated balancing systems* [6]. In contrast to robots with multi-legged and multi-wheeled bases, ballbots have a single point of support rather than a polygon of support, which makes it harder to balance. Further, non-zero shape changes result in accelerations in position space.

In this work, the ballbot is modeled as a rigid cylinder on top of a rigid sphere. A planar model of the ballbot is used in the developed MPC. The following assumptions are made in the model: (i) there is no slip between the spherical wheel and the floor, (ii) the motion in the median sagittal plane and the median frontal plane is decoupled, and (iii) the equations of motion in these two planes are identical. With these assumptions, we can design two decoupled, independent planar controllers for the 3D system.

Euler-Lagrange equations are used to derive the equations of motion of the planar ballbot model shown in Fig. 2(b). The angle between the body and the vertical is referred to as the body lean angle ϕ while the angle between the ball and the body is referred to as the ball angle, θ. The relation between the ball angle and the linear displacement of the ball $x_{ball} = \theta * r$. The equations of motion for the simplified planar ballbot model can be written in matrix form as follows:



(a) (b)

Figure 2. (a) The human-size CMU ballbot next to one of the smaller ballbots called "shmoobots"; both balancing and station keeping. (b) Planar ballbot model.

$$M(q)\ddot{q} + C(q,\dot{q}) + G(q) + D(\dot{q})) = \begin{bmatrix} \tau \\ 0 \end{bmatrix} \tag{12}$$

where $q = [\phi, \theta]^T$ is the generalized coordinate vector, $M(q)$ is the inertia matrix, $C(q,\dot{q})$ is the vector of Coriolis and centrifugal forces, $G(q)$ is the vector of gravitational forces, $D(\dot{q})$ is the frictional torque vector and τ is the torque applied between the ball and the body in the direction normal to the plane. The expressions for the above mentioned terms are given below:

$$M(q) = \begin{bmatrix} \alpha & \alpha + \beta\cos(\phi) \\ \alpha + \beta\cos(\phi) & \alpha + \gamma + 2\beta\cos(\phi) \end{bmatrix}, \tag{13}$$

$$C(q,\dot{q}) = \begin{bmatrix} -\beta\sin(\phi)\dot{\phi}^2 \\ -\beta\sin(\phi)\dot{\phi}^2 \end{bmatrix}, \tag{14}$$

$$G(q) = \begin{bmatrix} 0 \\ -\frac{\beta g \sin(\phi)}{r} \end{bmatrix}, \tag{15}$$

$$D(\dot{q}) = \begin{bmatrix} D_c sng(\dot{\theta}) + D_v\dot{\theta} \\ 0 \end{bmatrix}, \tag{16}$$

where $\alpha = I_{ball} + (m_{ball} + m_{body})r^2$, $\beta = m_{body}rl$, and $\gamma = I_{body} + m_{body}l^2$. D_c and D_v are the coulomb friction and the viscous damping friction terms respectively. However, in this work we ignore the friction and damping effects and drop the term $D(\dot{q})$ from Eqn. (12). A complete list of the model parameters used are listed in Table I. We define the state vector $\mathbf{x} = [\theta, \phi, \dot{\theta}, \dot{\phi}]^T$ and

Parameter	Symbol	Value
Z-axis COM from ball center	l	0.5 $[m]$
Ball radius	r	0.0762 $[m]$
Ball mass	m_{ball}	1.3 $[kg]$
Ball inertia	I_{ball}	0.006 $[kgm^2]$
Body mass	m_{body}	16.0 $[kg]$
Body inertia	I_{body}	3.0 $[kgm^2]$
Coulomb friction torque	D_c	4.39 $[Nm]$
Viscous damping friction coefficient	D_v	0.17 $[Nms/rad]$

Table I
BALLBOT SYSTEM PARAMETERS

input $\mathbf{u} = \tau$. Then, Eqn. (12) can be re-written in the control affine form:

$$\dot{\mathbf{x}} = f(\mathbf{x}, \dot{\mathbf{x}}) + g(\mathbf{x}, \dot{\mathbf{x}})\mathbf{u}. \tag{17}$$

In order to use the Linear MPC formulation described in Section V-A the equations of motion have to be linearized to form

$$\dot{\mathbf{x}} = A_{bb}\mathbf{x} + B_{bb}\mathbf{u} \tag{18}$$

where

$$A_{bb} = \frac{\delta f(\mathbf{x}, \dot{\mathbf{x}})}{\delta \mathbf{x}} \quad \text{and} \quad B_{bb} = \frac{\delta g(\mathbf{x}, \dot{\mathbf{x}})}{\delta \mathbf{u}}. \tag{19}$$

In this project, we leverage the robust balancing controller developed for the CMU ballbot [9]. Thus, for the MPC we use the closed-loop dynamics of the ballbot system described by:

$$\dot{\mathbf{x}} = A_{bb,cl}\mathbf{x} + B_{bb,cl}\hat{\mathbf{u}} \tag{20}$$

where

$$A_{bb,cl} = A_{bb} - B_{bb}K \quad , \quad B_{bb,cl} = B_{bb} * K, \tag{21}$$

$\hat{\mathbf{u}} = [\theta_{mpc}, \phi_{mpc}, \dot{\theta}_{mpc}, \dot{\phi}_{mpc}]^T$, and $K \in \mathrm{R}^{4x1}$ is a vector of Proportional and Derivative gains of the balancing controller. The matrices $A_{bb,cl}$ and $B_{bb,cl}$ are the ones used for the MPC implemented in Sections VII-C and VIII-B.

V. CONTROLLER DESIGN

A. linear_mpc

1) Matrix Manipulation: The MPC $\rightarrow$ QP translation is prone to implementation errors and slows down the iterative process necessary for solid controller development. One of the primary goals of this project was to develop a software package that can simplify the process of designing and deploying model predictive control algorithms for and to real robots. We've written a MATLAB package called linear_mpc that significantly reduces this headache by wrapping state-of-the-art QP solvers (OSQP,qpOASES and quadprog) with functions that abstract away nearly all of the matrix manipulations that are required to massage an MPC problem in QP form.

We constrain our library to be used only on linear time-invariant MPC tracking problems where we have some reference trajectory that we're trying to follow. The generic MPC form we focus on is as follows:

$$\begin{aligned} \min_{q,u} \quad & ||q_N - q_{N,ref}||^2_{Q_N} + \sum_{k=0}^{N} ||q_k - q_{k,ref}||^2_Q + ||u_k||^2_R \\ \text{s.t.} \quad & q_{k+1} = A_d q_k + B_d u_k \quad \forall k \in [0, N] \\ & q_0 = q_{0,ref} \\ & q_{lo} \leq q_k \leq q_{hi} \quad \forall k \in [1, N+1] \\ & u_{lo} \leq u_k \leq u_{hi} \quad \forall k \in [0, N] \end{aligned} \tag{22}$$

The primary task of our library is to convert a problem of this form (specified by the weight matrices Q,Q_N and R, the state update matrices A_d and B_d, the state bounds q_{lo} and q_{hi}, the control bounds u_{lo} and u_{hi}, and the reference trajectory q_{ref} into a standard form QP that we can pass in to optimized QP solvers. We use the following as our standard form:

$$\begin{aligned} \min_{q,u} \quad & \frac{1}{2} x^T H x + f' x \\ \text{s.t.} \quad & l \leq Ax \leq u \end{aligned} \tag{23}$$

We compose the MPC problem into our standard form QP using the following set of transformations:

$$x = \begin{bmatrix} q_0 & q_1 & \dots & q_N + 1 & u_0 & u_1 & \dots & u_N \end{bmatrix}^T$$

$$H = \begin{bmatrix} Q & & & & & & & \\ & Q & & & & & & \\ & & \ddots & & & & & \\ & & & Q_N & & & & \\ & & & & R & & & \\ & & & & & \ddots & & \\ & & & & & & R \end{bmatrix}$$

$$f = \begin{bmatrix} -\begin{bmatrix} Q & & & \\ & Q & & \\ & & \ddots & \\ & & & Q_N \end{bmatrix}^T * q_{traj}^T \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

$$A = \begin{bmatrix} A_d & -A_d & & & & & B_d & \\ & \ddots & \ddots & & & & & \ddots \\ & & A_d & -A_d & & & & B_d \\ \mathbf{I_{N_x}} & & & & & & & \\ & \mathbf{I_{(N)N_x}} & & & & & & \\ & & \ddots & & & & & \\ & & & \mathbf{I_{(N)N_u}} & & & & \end{bmatrix}$$

$$l = \begin{bmatrix} 0 & \dots & 0 & q_{0,ref} & q_{lo} & u_{lo} \end{bmatrix}^T$$

$$u = \begin{bmatrix} 0 & \dots & 0 & q_{0,ref} & q_{hi} & u_{hi} \end{bmatrix}^T$$

This is the precise form desired by the OSQP solver [23]. Each solver we've tested expects a slightly different version of the linear constraint, but this standard form constraint set can be easily morphed into the others we've seen.

Quadprog wants linear constraints in the following form [24]:

$$\begin{aligned} \min_{q,u} \quad & \frac{1}{2} x^T H x + f' x \\ \text{s.t.} \quad & l \leq x \leq u \\ & Ax \leq b \end{aligned} \tag{24}$$

and qpOASES looks for [25]:

$$\begin{aligned} \min_{q,u} \quad & \frac{1}{2} x^T H x + f' x \\ \text{s.t.} \quad & lbA \leq Ax \leq ubA \\ & l \leq x \leq u \end{aligned} \tag{25}$$

Note that we can simply ignore the simple bounds on qpOASES and use the same constraint set as OSQP. For quadprog, we again ignore the simple bounds, but this time we must convert our double-sided inequality to single sided

$$l \leq Ax \leq u \rightarrow \begin{bmatrix} -A \\ A \end{bmatrix} x \leq \begin{bmatrix} -l \\ u \end{bmatrix} \tag{26}$$

2) Practical Considerations: A useful thing to recognize from this formulation is that the matrices H and A do not change from iteration to iteration; they can be set up on initialization and never modified unless the state space form or weights need to change. This reduces the matrix manipulation overhead at each iteration of our control loop; some active-set based solvers even have specialized forms that can warm-start the active-set approximation from previous iterations when the hessian matrix and linear constraint matrix are static [25].

3) Solver Benchmarking: One of the primary advantages of having this modular framework is it allows us to rapidly and accurately compare different QP solvers for different MPC applications. We performed a variety of efficiency tests comparing the solve times of *osqp*, *qpOASES* and *quadprog* (see table II) using our MPC framework on models of a drone and the tumbller. All solvers were set to the same or similar convergence properties during testing and all solvers were hotstarted with the previous solution. As a general rule, we find that *osqp* is considerably faster than *qpoases* which is slightly faster than *quadprog*.

We attribute the incredible performance of *osqp* as compared to *qpOASES* down to the sparse formulation we use for MPC. In future work we will add a denser (implicit representation of states) formulation to our library and compare these two again on a more apples to apples basis. We use *quadprog* with the interior-point solver, which in theory should give similar performance to *qpOASES*. On single solves, we find this to be true, but in the model predictive control domain we end up solving serial chains of QPs and we believe that *qpOASES* does a better job of initializing both the seed point and the active set from the previous solve, which helps it to converge faster.

VI. Estimator Design

We decomposed the state estimation problem into the sagittal plane and the longitudinal plane as the angle estimation relies on the IMU and the position estimation fuses the IMU with encoder measurements. In the first case, we present a comparison between a previously implemented complementary filter and a Kalman filter that fuses the accelerometer and gyro and show the superior accuracy of Kalman filter in simulation. In the longitudinal plane, we compared a kinematic Kalamn filter with an optimization-based moving horizon estimator (MHE) and show the superior accuracy of MHE at reasonable computation cost.

1) Kinematic Kalman Filter (KKF): In the state estimation problem, we formulate our Kalman filter for position estimation based on acceleration and position measurements from the accelerometer and encoder on the robot. Note that this formulation is compatible with any robotic system that is equipped with IMU and encoder. Furthermore, it can also include any external position measurements if available. We define the states of the filter as follows:

$$x_k = \begin{bmatrix} x, & y, & \dot{x}, & \dot{y}, & \ddot{x}, & \ddot{y} \end{bmatrix} \tag{27}$$

Where the dynamics of the filter are derived from the constant acceleration assumption as follows:

$$A = \begin{bmatrix} 1 & 0 & \Delta t & 0 & \frac{1}{2}\Delta t^2 & 0 \\ 0 & 1 & 0 & \Delta t & 0 & \frac{1}{2}\Delta t^2 \\ 0 & 0 & 1 & 0 & \Delta t & 0 \\ 0 & 0 & 0 & 1 & 0 & \Delta t \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{28}$$

where the meaurement function when measuring position and acceleration will be as follows:

$$H = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{29}$$

Equations (30), (31), (32), (33), and (34) are the main Kalman filter steps as follows:

$$\hat{x}_{k|k-1} = A\hat{x}_{k-1} + B_{k-1}u_{k-1} \tag{30}$$

$$P_{k|k-1} = A_{k-1}P_{k-1}A_{k-1}^T + Q_{k-1} \tag{31}$$

$$K_k = P_{k|k-1}H_k^T(H_kP_{k|k-1}H_k^T + R_k)^{-1} \tag{32}$$

$$\hat{x_k} = x_{k|\hat{k-1}} + K_k(z_k - H_k x_{k|\hat{k-1}}) \tag{33}$$

	Drone N = 10	Drone N = 20	Tumbller N = 10	Tumbller N = 20	Tumbller N = 40
quadprog, interior-point	6.71 ms	23.6 ms	2.59 ms	4.37 ms	11.7 ms
osqp	3.31 ms	4.75 ms	2.96 ms	3.35 ms	4.13 ms
qpOASES	5.72 ms	22.1 ms	1.51 ms	3.39 ms	18.3 ms

Table II
QP SOLVE TIME COMPARISON

$$P_k = (I - K_k H_k) P_{k|k-1} \tag{34}$$

where, the parameters are summarized in the table below.

Parameters	Description
A	State transition matrix
B	Input matrix
Q	Process noise cov
R	Measurement noise cov
P	State error cov
K	Kalman gain
H	Output matrix
z	Measured output

2) Angle Estimation Kalman Filter: For angle estimation, we have developed a Kalman filter that uses the gyro as the process model and accelerometer as the measurement. The Kalman filter update equations are identical to section 1. Figure below demonstrates the comparison between the Kalman filter and the complementary filter in simulation. As shown in the bottom figure, the Kalman filter performs significantly better in terms of state estimation error. This was consistent across various noise levels. The same filter was also implemented in C code and deployed on the Tumbller hardware. Figure 4 demonstrates the the Kalman filter (in blue) and the complementary filter (in red) running on the Tumbller hardware. The KF provides slightly faster tracking response. It is difficult to compare the accuracy as there are no ground truth measurements to reference.

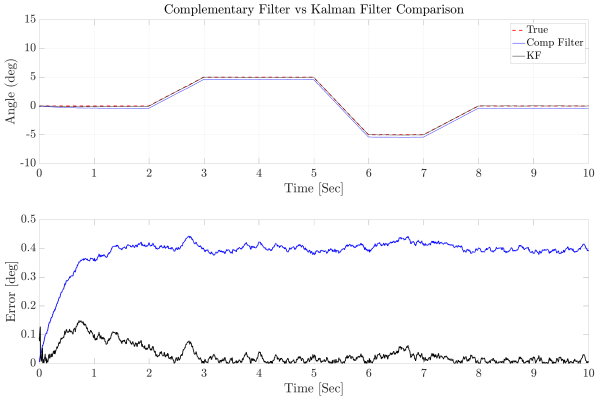


Figure 3. Complementary filter (red) vs Kalman Filter Comparison (blue)

3) Moving Horizon Estimation: The moving horizon estimation problem is setup as the dual of MPC looking backwards in the past. The mathematical structure of the problem is almost identical to that of MPC except for the addition of the arrival cost term to the cost function, which summarizes the previous information at each estimation step. Equation (38) captures the system model and equation (39) is identical to the Kalman filter measurement function. Equation (40) summarizes any inequality constraints on either the states and the controls. p_v and p_w are measurement and process noise matrices respectively. The MHE framework allows for natural incorporation of constraints and relaxes the markov assumption as it relies on a window of measurements and not just the most recent measurement.

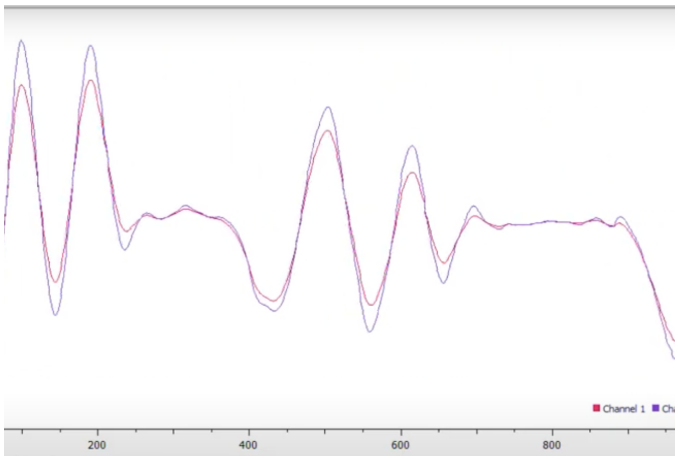


Figure 4. Complementary filter vs Kalman Filter Comparison on Tumbller

$$min_{x_{0:N+1}} \|x_0 - \tilde{x}_0\|^2_{p_x} + \|p - \tilde{p}\|^2_{p_p} + \sum_{k=0}^{N-1} (\|v_k\|^2_{p_v} + \|w_k\|^2_{p_w}) \tag{35}$$

$$x_{k+1} = f(x_k, u_k, z_k, ...) + w_k \tag{36}$$

$$y_k = h(x_k, u_k, z_k, ...) + v_k \tag{37}$$

$$g(x_k, u_k, z_k, ...) \leq 0 \tag{38}$$

To compare the KKF with the MHE we have conducted a series of experiments on the differential drive kinematics model as follows:

$$\begin{bmatrix} x(k+1) \\ y(k+1) \\ \theta(k+1) \end{bmatrix} = \begin{bmatrix} x(k) \\ y(k) \\ \theta(k) \end{bmatrix} + \Delta T \begin{bmatrix} v(k)cos(\theta(k)) \\ v(k)sin(\theta(k)) \\ \omega(k) \end{bmatrix} \tag{39}$$

In both cases, the measurements are acceleration and position and the measurement and process noises were

Horizon	# Var	Error (x)	Error (y)	Solve Time (ms)
6	33	0.158	0.189	1.56
8	43	0.146	0.178	1.79
10	53	0.143	0.132	2.13

Table III
STATE ESTIMATION AVERAGE RESULTS OF DIFFERENTIAL DRIVE KINEMATIC MODEL

identical for each run. Note that to incorporate the acceleration measurement in MHE, it was added in terms of the position. KKF naturally includes acceleration as part of the state definition. Table III summarizes the the average results of 10 experiments conducted on the differential drive kinematic model. We ran the forward simulation on the nonlinear model with the MPC controller in the loop from $x_0 = [0.1, 0.1, 0]$ to $x_f = [1.5, 1.5, 0]$ with various process and measurement noise matrices and averaged the results over all the runs. As shown, we were able to solve the MHE problem up to 1 second look-back horizon at below 5 ms which makes this approach feasible for real-time applications. The computation time could be further reduced with a C++ implementation [23]. Furthermore, figure 5 demonstrates the superior accuracy of MHE compared to the KKF in terms of state estimation error.

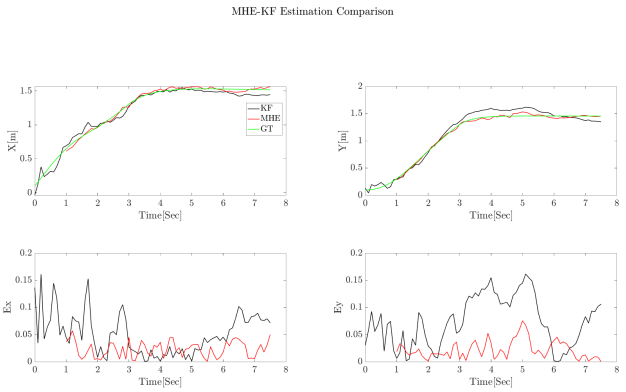


Figure 5. KKF vs MHE Comparison

VII. Simulation Results

A. Tumblller

We've developed a simple simulator to verify our controller on the linear Tumblller model. We run MPC at 100 hz with a horizon of 20 steps.

For the reference trajectory demonstrated below, we use the following weights (state weights corresponding to $[x, \dot{x}, \theta, \dot{\theta}]^T$, control corresponds to horizontal force:

$$Q_N = diag([1000, 0.1, 3, 0.1]) \quad (40)$$
$$Q = diag([10000, 1, 30, 1]) \quad (41)$$
$$R = 0.1 \quad (42)$$

Although not overly complex (i.e. we use the same linear model for simulation and control), this simulation

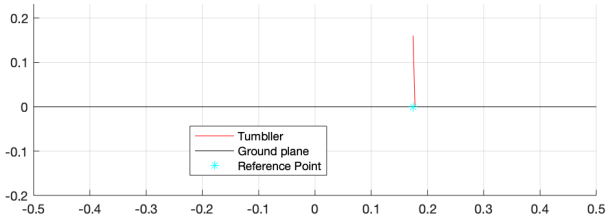


Figure 6. Tumblller Sine Wave Trajectory

allows us to verify that our controller finds adequate solutions that can stabilize the tumblller at reasonable control rates as well as effectively tracking a desired reference trajectory.

B. Drone MATLAB sim

We want to verify that our quadcopter controller is stable and effective before deploying to (and risking the safety of) hardware, so we've developed a high fidelity MATLAB simulation that allows us to test the MPC component in isolation pre-deployment. It uses a nonlinear model for simulation as well as a lower-level attitude PD controller. Our outer MPC runs at 50 hz (using *osqp* solver, horizon of 20 steps); at each timestep we use *ode45* with the attitude controller embedded in the state update function to update our state from iteration to iteration.

For each reference trajectory demonstrated below, we use the following weights (state weights corresponding to $[x, y, z, \dot{x}, \dot{y}, \dot{z}, roll, pitch]^T$, control corresponds to $[roll, pitch, thrust, grav]^T$:

$$Q_N = diag([200, 200, 1000, 50, 50, 100, 100, 10, 10]) \quad (43)$$
$$Q = diag([20, 20, 100, 5, 5, 10, 10, 1, 1]) \quad (44)$$
$$R = diag([10, 10, 0.2, 0]) \quad (45)$$

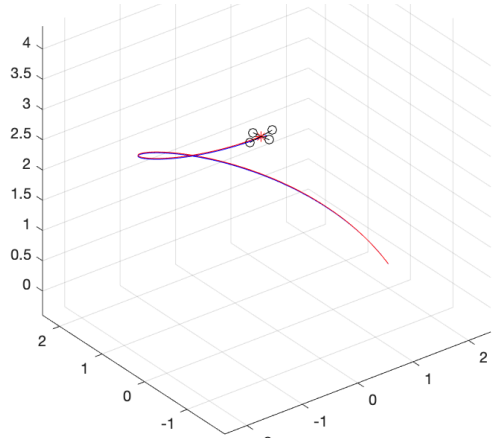


Figure 7. Drone Rising Spiral Trajectory

As can be seen from the sawtooth and straight trajectories, we can effectively track dynamically infeasible trajectories (red represents the achieved trajectory whereas

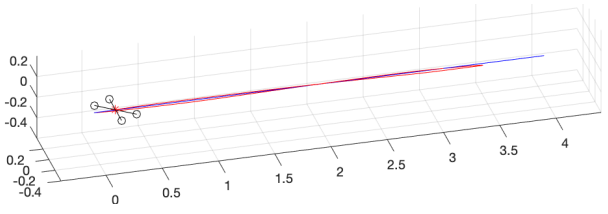


Figure 8. Drone out and back Trajectory

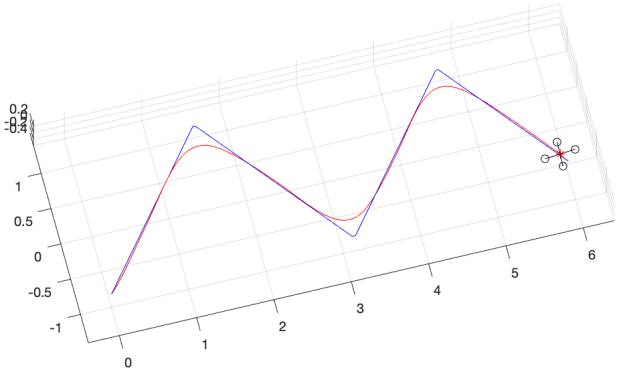


Figure 9. Drone Horizontal Sawtooth Trajectory

blue is the reference). This is an incredibly powerful feature of MPC control that can differentiate it from LQR or any pure feedback controller; the forward looking horizon allows us to round off tight corners and minorly deviate from the trajectory before we're forced to if it means we can avoid majorly deviating in the future.

C. Ballbot

An important task for robots is to perform rest-to-rest motions. However, for ballbot this task is not easy because of its under actuation. The control algorithm has to be able to weigh the use of its control input to either control the body lean angle to maintain balance or the ball position to track the desired trajectory. MPC is a good method to calculate control commands for a ballbot so that it can achieve both tasks of balancing and tracking simultaneously. We propose the use of a hierarchical structure with an inner loop balancing controller that consumes desired body lean angle and ball velocity setpoints. An outer loop controller generates the desired lean angle and velocity set-points to track a desired ball ground position. A schematic of the proposed control framework is shown in Fig. 10.

The proposed MPC framework was tested to perform different tasks in a dynamic simulation environment of the CMU ballbot using Matlab. We tested the framework in planar and 3D scenarios. For all the tasks performed we set the same QP and cost function weights. The only difference between tasks is the input reference trajectory. For all tasks we use the following weight matrices for the

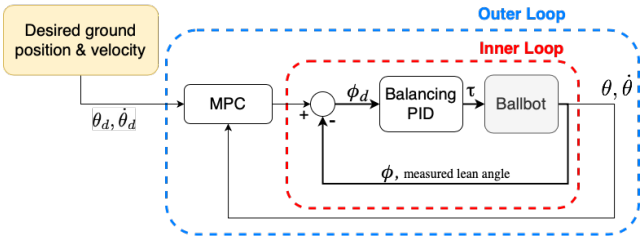


Figure 10. Overview of the implemented hierarchical control structure with an outer MPC loop and an inner balancing loop in simulation

quadratic cost function:

$$Q_N = diag([1000, 100, 100, 10]) \quad (46)$$

$$Q = diag([100, 10, 10, 1]) \quad (47)$$

$$R = diag([1, 1, 1, 1]). \quad (48)$$

The fact that we can use the weights for different tasks is a positive result towards our goal of formulating a platform-agnostic MPC.

D. Planar ballbot - Linear motion

In this experiment, we set the desired reference trajectory for the linear position of the planar ballbot model to be a ramp function with constant velocity $x_{ball} = 0.1m/s$. Screenshots of the simulated motion are shown in Fig. 11. As shown in Fig. 12, the state evolution follows very closely the reference trajectory but not perfectly. This is because the input reference trajectory is not dynamically feasible. Thus, the MPC outputs commands that cause the ballbot to diverge from the reference trajectory to satisfy the system dynamics. The non-zero lean angle motion at the beginning is necessary to induce a forward acceleration from rest. Similarly, the second non-zero lean angle motion is necessary to decelerate the robot so that it comes to a rest.

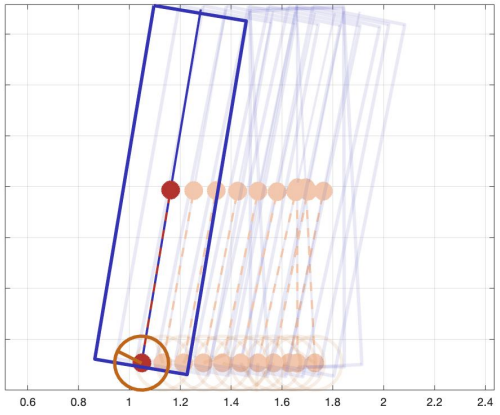


Figure 11. Screenshots of planar ballbot simulation tracking a ramp trajectory for the linear ground position, lighter color represents MPC optimal trajectory look ahead output.

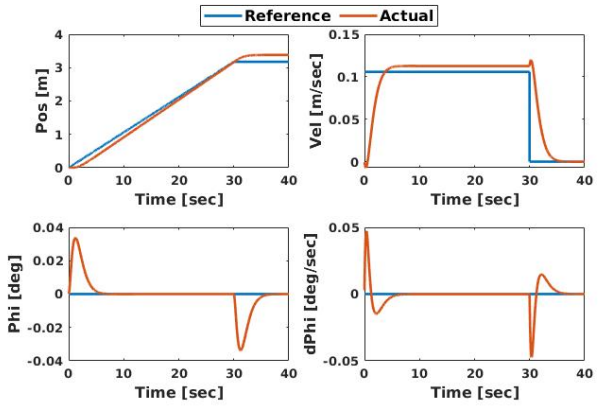


Figure 12. Planar ballbot simulation tracking a ramp trajectory for the linear ground position.

E. Planar ballbot - Step motion

A harder reference trajectory to track is a step response. Here a sudden change in ground position is desired. The MPC framework is capable to track a step response of 2 m as shown in Fig. 13. As desired the MPC outputs an optimal trajectory that first moves backward to induce a positive body lean angle towards the direction of motion followed by a negative body lean angle to stop the ballbot. The maximum step response that was possible to track was 2.5m.

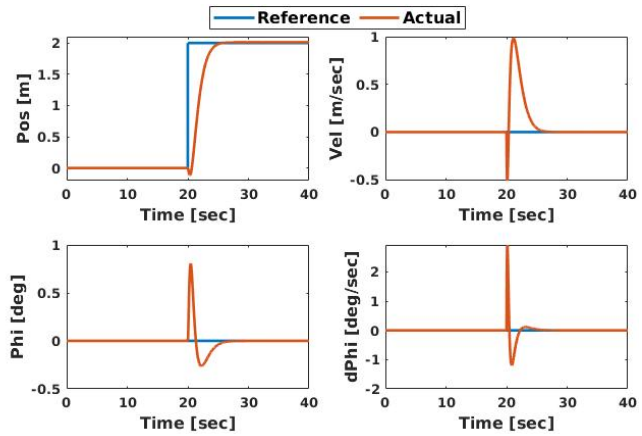


Figure 13. Planar ballbot simulation tracking a square reference trajectory for the linear ground position.

F. 3D ballbot - Circular motion

An extension of the planar simulation is a 3D simulation of ballbot. Following the assumption that the motion of the sagittal and frontal planes are decoupled we control the motion of each plane with separate instances of the MPC framework. Thus, two QPs are solved on each control loop iteration. A 100 Hz control rate loop is still attainable. We test the performances of this dual MPC set up by setting the task to track a circular motion, as shown in Fig. 14. It is important to highlight that the MPC generates the motion from the initial state of the robot in the center of the circle to the circular path. This motion is not included in the reference trajectory. Demonstrating the capability of the MPC to track dynamically infeasible trajectories. Ballbot can track the circular motion within $\pm$ 0.05 m.

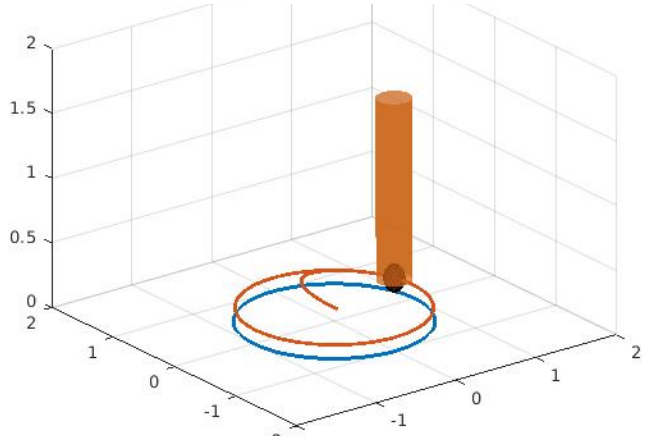


Figure 14. 3D ballbot simulation tracking a circular reference trajectory of 1 m radius for the planar ground position

VIII. Hardware Implementation & Testing

A. Tumbller

Throughout this course, we have implemented various controllers on the ELEGOO Tumbller self-balancing robot. During Lab 4, we encountered numerous hardware issues leading to difficulty in the implementation of MPC reference tracking on the Tumbller. As a part of this project, we propose an upgraded Tumbller centered around replacing the Arduino Nano with a Teensy 4.1 microprocessor.

Our main focus was to increase the wireless Bluetooth communication speed. The Arduino Nano connected to the HC-06 Bluetooth module (when connected using the SoftwareSerial library) was benchmarked as having a maximum baudrate of 38400 without data corruption. Pushing this speed any higher would result in nearly every message having corrupted bytes. In terms of MPC reference tracking, this speed limitation made it very difficult to send full state trajectories to the Nano from Matlab. Secondary issues that we hoped to address included the lack of use of the B channels of the quadrature encoders of the motors, as well as the general speed and program size allowed when using the Nano. By default, the B channels of the encoders are not connected to the Tumbller's PCB, and therefore are not accessible using the Nano in the default socket. Therefore, direction of the wheel rotations were determined using the sign of the motor voltage. While this did not present an issue for balancing control, for reference tracking we would like a more accurate estimate of our wheel velocity and position.

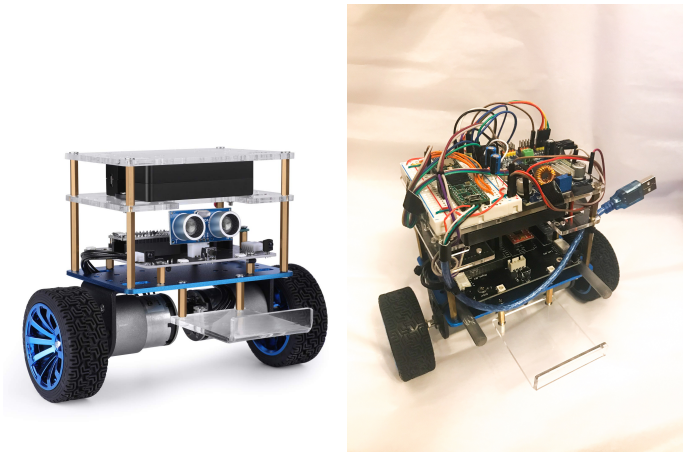


Figure 15. Original ELEGOO Tumbller (left), Modified Tumbller with Teensy 4.1 (right)

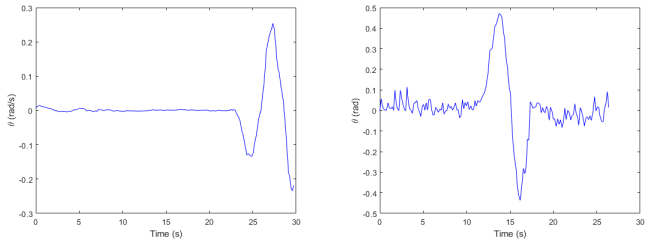


Figure 16. No PWM commands being sent (left), PWM commands being sent (right). Note that when PWM commands are not being sent, the θ estimate is smooth with very little noise, however, when PWM commands are being sent, there is upwards of 0.5 rad in noise on the signal. This noise, caused by the I2C corruption was leading to very shaky balancing.

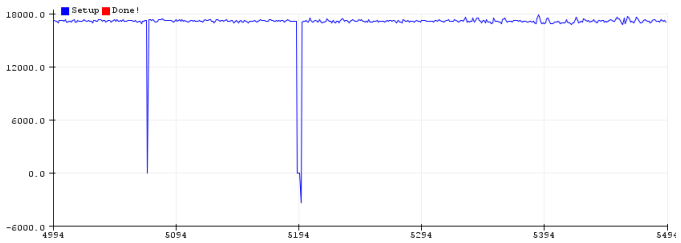


Figure 17. In this figure we see the raw value of the acceleration in the z direction from the IMU. We expect this value to be between 17000 and 18000 when the Tumbller is upright, however, we see large spikes in this measurement, dropping down to 0 and below 0. In this case, we are able to ignore these values as they are outside of the expected range and rate of change, but not all of the corrupted values followed this trend.

The hardware installation for this upgrade is relatively simple. The Teensy is mounted to a breadboard on top of the Tumbller, as well as all necessary electrical components to make this modification possible. Power is supplied to the Teensy through a voltage regulator connected to the external Tumbller battery. This regulator is necessary as the Teensy has a maximum voltage input of 5.5 V, whereas the Nano could handle the full 8 V from the battery. All necessary signals are passed into the Teensy through 3.3V to 5V logic converter boards, as the Teensy is a 3.3 V board. Finally, a HiLetgo PCA9685 servo motor driver module was connected to the PCB socket to handle sending PWM signals to the motor driver board. The addition of this module will be discussed in detail below. Before and after images of the Tumbller platform can be seen in Figure 15.

Using the Teensy, we were able to connect the Bluetooth HC-06 module directly to a hardware serial port. This, as well as the HC-06 now being connected with the proper 3.3V logic signals, was able to communicate at 1382400 baud to and from Matlab. This 36x speed increase enables us to communicate with Matlab at much faster speeds, allowing for full robot state and control reference trajectories to be sent from MPC in Matlab to the Tumbller. Also, we note that less than 1% of messages have corruption, and checks are made within the code to account for these. By soldering jumper wires directly to the motor connectors, we were able to connect the Teensy to the B channels of the quadrature encoders. Finally, we were able to bring the stabilizing controller execution time down from 2.5 ms on the Nano, to 0.4 ms on the Teensy.

Unfortunately, despite the hardware installation going smoothly, the upgraded Tumbller was plagued with internal communication errors. When implementing the balancing controllers from Lab 2, we noticed that the Tumbller was very shaky and struggled to balance smoothly. We discovered that the I2C signals from the IMU were being corrupted once or twice a second. These corrupted values were being used in our state estimation, thus creating rapid spikes in estimates of θ and $\dot{\theta}$. Upon much further investigation, we found that these corruptions were much more frequent while sending PWM commands directly from the Teensy to the motor driver module. A comparison of the θ estimate can be seen in Figure 16, and a plot of some example data corruptions can be seen in Figure 17. Note that for both of these trials, the motors were physically unplugged, and the motion of the robot was generated by holding and rocking the Tumbller by hand.

To fix this issue, we attempted to add checks on the ranges and rates of the IMU data to discard any corrupted values. Unfortunately, some of these corrupted signals fell within acceptable values. Next, we added the servo driver board to handle sending the PWM, to attempt to offload this potential issue from the Teensy. Luckily, this change nearly eliminated the IMU corruption issue, and the remaining corrupted values can be handled using range based elimination.

Aside from IMU data corruption, we also faced issues with our stabilizing control loop occasionally taking around 40x longer than expected. In Figure 18 we see the time to execute the code in the stabilizing controller. For a reason that we have not determined yet, on occasion

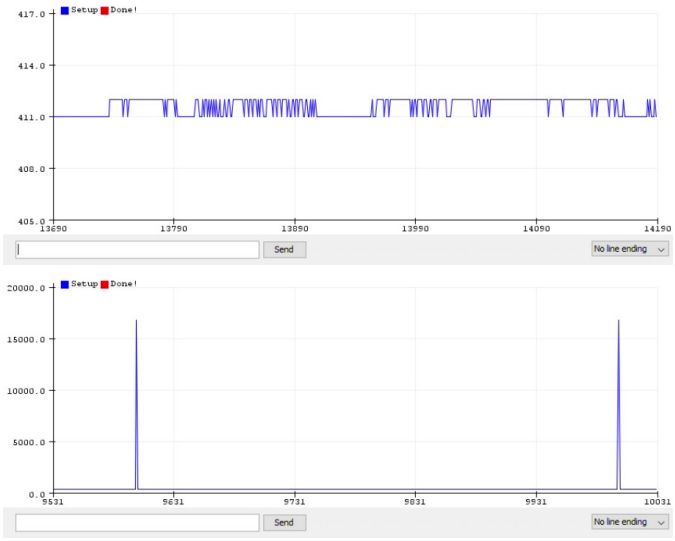


Figure 18. Expected control loop execution time (Top) Control loop time spikes (Bottom). Here we see occasional spikes in the execution of our stabilizing controller (nominally 0.4 ms, spiking to 15 ms). This caused our controller to hold the same motor voltage over this longer period of time, destabilizing the Tumbller for a moment. The frequency of the these spikes have been greatly reduced by adding capacitors on the power supply to the Tumbller, however, they still occur, and the cause is currently still unknown.

this execution time will jump from 0.4 ms to 15 ms for no apparent reason. This results in the controller holding the same motor voltage for this period of time, thus destabilizing the Tumbller for a moment. Upon inspection of the Tumbller power supplies, we noticed that we had voltage fluctuation while balancing. By adding two 100 μF capacitors to the 5V power rail, the execution delays were much less frequent. Instead of having multiple spikes every second, we only have a few per minute, which our balancing controller can handle without much issue. The cause of this issue has still not been determined, but its effects have been minimized as much as possible.

Finally, after mitigating the previously mentioned issues, we encountered another setback. During the tuning of a balancing controller for the finalized Tumbller, it went unstable, crashed, breaking the USB port on the Teensy. Without being able to get a replacement in time, we were forced to attempt to solder spliced USB communication wires directly to the pads on the underside of the Teensy. Fortunately, this fix went smoothly, and we were able to proceed with tuning. The break and fix can be seen in Figure 19.

Despite having many setbacks in the process of upgrading the Tumbller robot, we took advantage of the increase in performance of the robot and implemented multiple controllers on the Tumbller. We were able to take our Lab 2 PD controller and with some tuning generate a stable balancing behavior. The states and control of an example run of this behavior can be seen

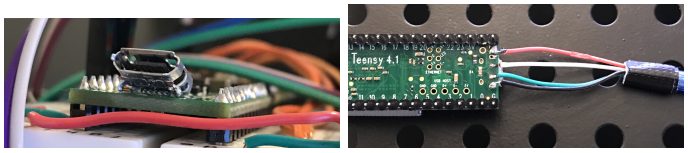


Figure 19. Broken USB port after crash during tuning (left) Soldering fix using a spliced USB cable (right). In this case we were fortunate to have access to a soldering iron, but this serves as a reminder to always have extra components in case of failure.

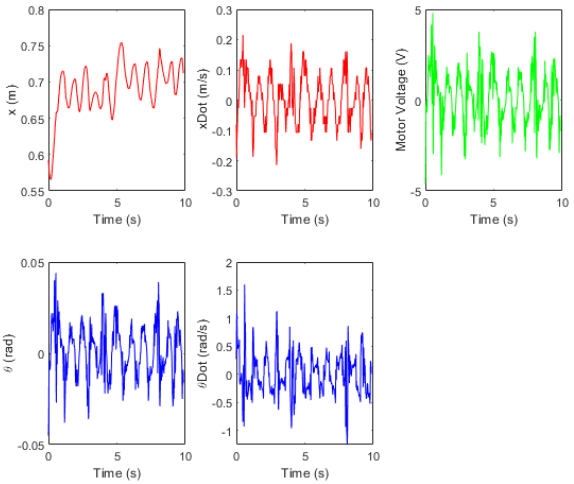


Figure 20. PD balancing control on the upgraded Tumbller. We see that the robot sways back and forth about the balancing point, with small deviations in θ, and averaging less than 3 V in control input.

in Figure 20. Next we implemented two forms of MPC: MPC balancing control, and MPC reference tracking with an inner stabilizing controller. The results of these controllers will be discussed below in the Final Results section of this report.

B. Ballbot

The Microdynamic Systems Lab has several different ballbot platforms. For this project we use the smaller ballbot version that is named *shmoobot*, shown in Fig. 2(a). The ballbot has two computers onboard; an Arm MBed LPC1768 microcontroller that runs the inner balancing controller at 500Hz, and an NVidia Jetson Nano running Ubuntu OS and the robotics middleware ROS [26].

To implement the trajectory tracking MPC we make use of the open-source C++ *linear_mpc* library we developed[1]. A dedicated ROS package that interfaces with the robot hardware, defines the robot dynamics, generates the reference trajectory, and instantiates the MPC instance was also developed and made open-sourced in github[2].

[1]https://github.com/nikolaif399/LinearMPC/tree/devel/cpp
[2]https://github.com/rshum19/ballbot_mpc_nav

The QP in each MPC iteration is solved on average in 5 ms. Thus, we comfortably set the MPC to run at 100Hz (10ms), to ensure that the optimization solver has sufficient time. The low-level balancing controller runs at a faster 500Hz. Thus, we hold the MPC command value for 5 iterations of the inner balancing controller. The same weight matrices Q_N, Q, R, and horizons length N values as the simulation are used here.

A block diagram of the control structure implemented in the hardware is shown in Fig. 21. There are three main differences between the hardware and simulation implementation of Section VII-C. Instead of passing the entire $\hat{\mathbf{u}} \in \mathrm{R}^{4\times 1}$ command signals from the MPC we pass only the body lean angle ϕ_{mpc} to the balancing controller. The balancing controller implemented on the hardware only consumes ball velocity $\dot{\theta}$ and body lean angle ϕ commands. Further, in practice, it was noticed that the MPC ball velocity output commands $\dot{\theta}_{mpc}$ deteriorated the performances of the controller.

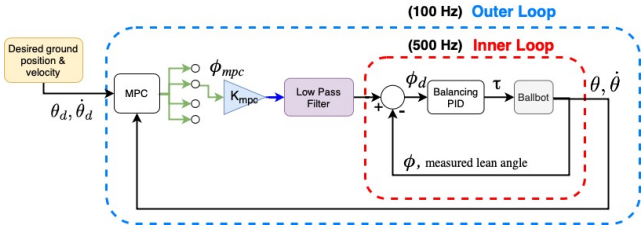


Figure 21. Overview of the implemented hierarchical control structure with an outer MPC loop and an inner balancing loop in hardware with the addition of a signal selector, proportional gain and low pass filter.

From experimentation, it was noted that the MPC command was too large causing the system to have a poor tracking performance. From hand-tuning it was determined that a proportion gain of $K_{mpc} = 0.5$ needed to be applied to the MPC output before the inner loop. This could be due to a mismatch between the actual hardware physical parameters and the parameter values used in the mathematical model.

Additionally, the MPC output had to be passed through a low pass filter as in Eqn. (49) to smooth out the command signal. The noisy raw signal sometimes caused undesirable instabilities.

$$\phi'[t] = \beta \cdot \phi_{raw}[t] + (1-\beta) \cdot \phi'[t-1] \tag{49}$$

IX. Final Results

A. Drone Gazebo sim

On the road to implementing MPC on the SubT drone, we first tested our formulation on the SubT Gazebo sim, which is based on the base Px4 simulator with modifications. The sim is integrated with the rest of the autonomy stack, hence the MPC can be tested out in conjunction with the Local Planner and Disturbance Estimator.

The implementation utilizes the drone state matrices in IV-B along with state and input constraints capping the attitude below a certain threshold. Additionally, the reference trajectory is set according to a feed-forward segment published by the local planner, containing velocities and accelerations.

We set up qpOASES with a ROS wrapper in C++ and call a hotstarted solve at every 25 Hz. The MPC ouputs a sequence of attitude and thrust commands, out of which only the first one is executed. The hotstarted implementation is much faster than full solve, reducing the average times from 400 ms to 20-30 ms. The MPC package runs along with the rest of the autonomy stack as well as the Gazebo sim, which slows the solve a lot compared to solving as a standalone program. However, the drawbacks of hotstarting are that the dynamics cannot be varied at every time step, hence disturbances have to be added a-posteriori. This is fine from a steady state error perspective, but doesn't guarantee constraint satisfaction. Additionally, fallbacks and safety conditions were coded in to ensure that if the solve fails or reaches max working set iterations, the MPC will revert to the baseline PD control.

After tuning using the MATLAB sim described in VII-B and some further tuning in Gazebo, the drone flights looked pretty good. The MPC smoothly deals with infeasible plans by optimizing local trajectories, doesn't make the Px4 attitude controller unstable, and solves pretty reliably. The only concern that needs to be addressed is occasional high jerks in angular rates that were observed in sim. In order to minimize this, X and Y response had to be set to be a bit overdamped. Including angular velocities in the state would enable us to penalize them explicitly, making tuning and deployment easier.

We tested step responses about hover state, racetrack style static trajectories with an abrupt stop at the end to test response to infeasible plans, and autonomous exploration with the local planner to test integration with the autonomy stack. Results with the step response and racetrack are presented in IX-A. When we run MPC with the local planner, there is near perfect tracking in sim upto 2.5 to 3 m/s, since the planner is guaranteed to provide feasible trajectories.

The param set from which results were generated is Q=[15, 15, 50, 5, 5, 10, 0.5, 0.5], Q_f=10Q, R=[10, 10, 0.1].

B. Tumbller MPC

Due to the significant increase in Bluetooth communication speeds and impressive solve times in Matlab with OSQP, we decided to attempt to run a MPC balancing controller real time on the Tumbller via Bluetooth communication. These speeds allowed us to run MPC at 100 Hz. The setup of this problem is identical to the simulation setup discussed earlier, but with gains tuned for the hardware system. We were able to achieve balancing for

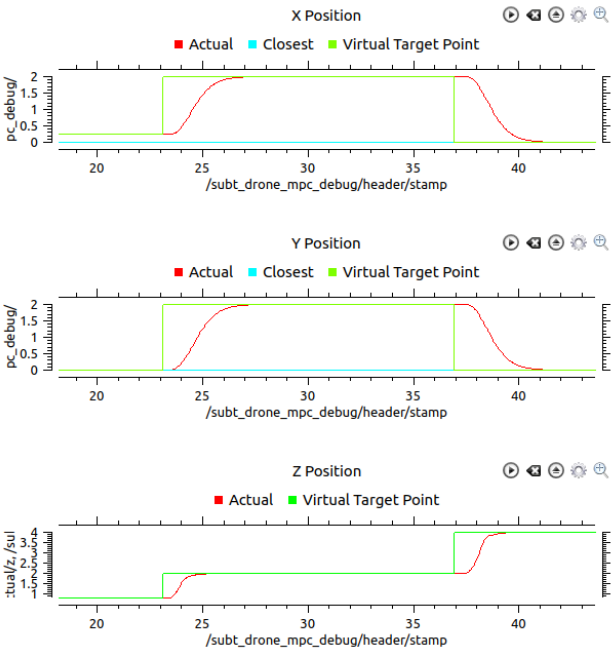


Figure 22. Step responses in Gazebo sim, X axis is seconds, Y axis is meters.

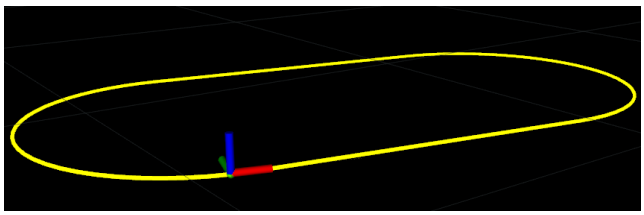


Figure 23. Racetrack in RViz, length = 6m, width = 3m

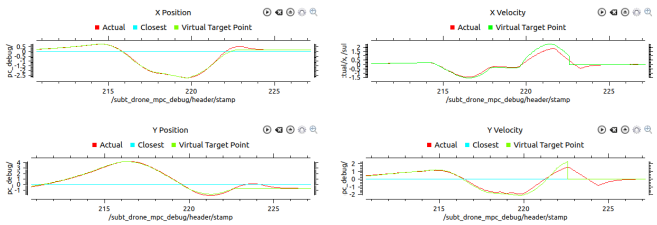


Figure 24. Position and velocity comparisons for racetrack, X-axis is seconds, Y-axis in SI units

5-10 seconds at a time, but the controller seemed to lag behind, and would occasionally go unstable. An example run of this behavior can be seen in Figure 25. Despite this relatively poor performance, we believe that further tuning would improve the balancing performance. It is also possible that the changes to the robot hardware have changed the mass and inertia properties enough to impact our controller performance. We believe that upon further tuning, reference tracking may be possible through this controller.

Finally, we attempted to run MPC reference tracking

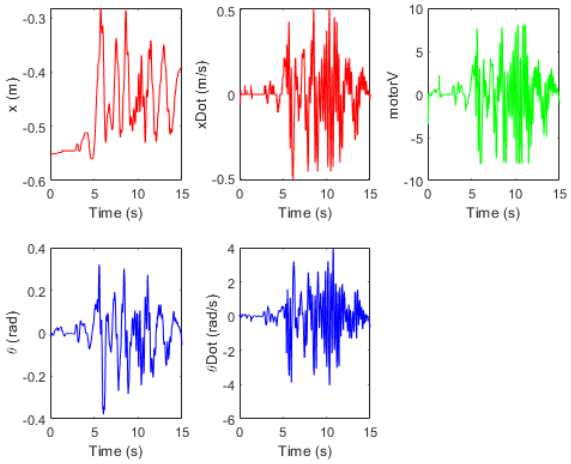


Figure 25. MPC balancing control on the upgraded Tumbller. We see that the robot sways back and forth about the balancing point, but with large deviations in θ and x. There is much room for improvement in this controller, but this result provides hope that more tuning will smooth out these issues.

on the Tumbller at 50 Hz, providing the PD stabilizing controller, running at 200 Hz, a full state and control reference trajectory, via Bluetooth from Matlab, for four time-steps in the future. Due to the numerous setbacks during the Tumbller upgrade process, and the time constraints of the project, we were unable to spend enough time tuning to show a significant reference tracking result for the Tumbller hardware. We believe that with more time and tuning that we will be able to achieve simple trajectory following behaviors at a minimum.

C. Ballbot Hardware

To validate the performance of our control algorithm on the real ballbot hardware we define the task to track a ramp trajectory for the ball position. The hierarchical control framework proposed managed to successfully track the trajectory within $\pm$ 0.05 m, as shown in Fig. 26. The MPC commands over time are shown in Fig. 27. These commands are of similar magnitude to those from other outer loop controllers implemented by MSL [12]. Screenshots of the entire motion are shown in Fig. 28. Unfortunately, the ballbot had a fall that broke the encoder in the yaw degree of freedom. Without, reliable data from this sensor it was very difficult to track a 3D motion such as a circular path in hardware.

X. Conclusion

In conclusion, we have presented the design and implementation of a model predictive control framework in simulation on three different robotic platforms. The designed controllers were successfully deployed on the Ballbot and upgraded Tumbller hardware. Furthermore, the developed open-source MPC package allowed for

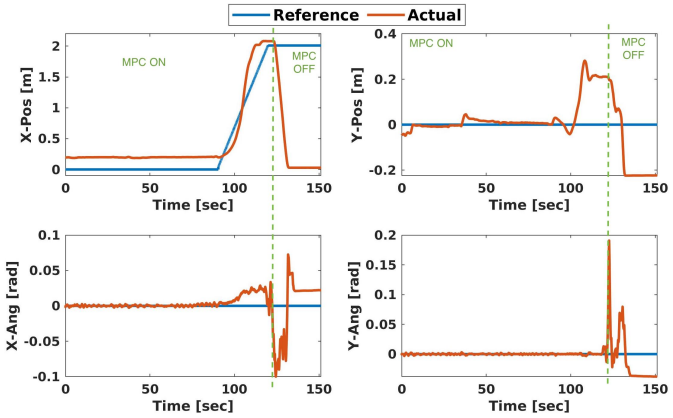


Figure 26. Evolution of state variables and reference trajectory under MPC.The green dash line indicates the separation between when MPC was active and when it deactivated for manual control.

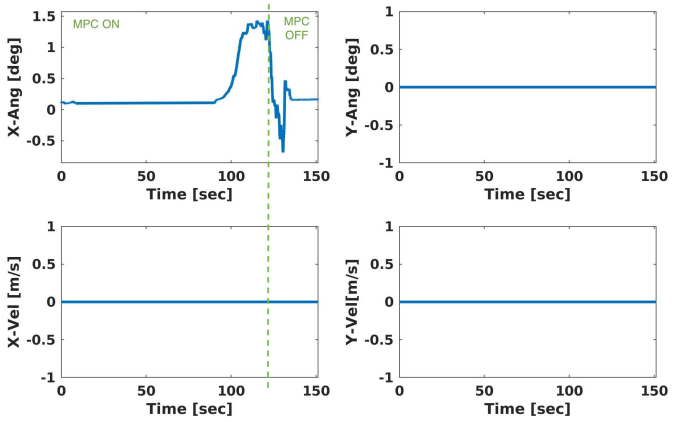


Figure 27. Evolution over time of MPC commands. The green dash line indicates the separation between when MPC was active and when it deactivated to manual control.

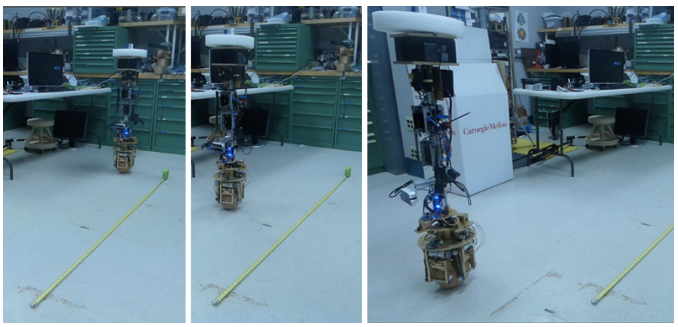


Figure 28. Screenshots of ballbot tracking a ramp trajectory with the MPC controller.

faster and more modular implementation across different hardware platforms. For the upgraded Tumbller platform, we acknowledge that our current MPC stabilizing controller results may not be as smooth as a simple PD control, but with tuning, we believe that we can improve its performance. We also believe that with extensive tuning, reference tracking will be possible for the system. On the ballbot only tracking of a linear motion was attainable. However, this does not show a deficiency in our proposed control framework, since the error was found to be a broken encoder that caused erroneous state feedback. We believe that 3D trajectory tracking is achievable. On the state estimation side, we have demonstrated that the Kalman filter performs more accurately compared to the complementary filter. Finally, it was shown that a moving horizon estimator performs more accurately compared to a Kalman filter for position estimation at a reasonable computation cost. For hardware implementation on the SubT drone, incorporating angular slew rate penalties and further tuning, followed by testing on previous ROS-bag data by running MPC in the background is planned. Once the ROS package passes these checks, it will be deployed on the drone and developed further for the DARPA competition.

References

[1] S. J. Qin and T. A. Badgwell, "A survey of industrial model predictive control technology," *Control engineering practice*, vol. 11, no. 7, pp. 733–764, 2003.

[2] A. Alessio and A. Bemporad, "A survey on explicit model predictive control," in *Nonlinear model predictive control*. Springer, 2009, pp. 345–369.

[3] K. Holkar and L. Waghmare, "An overview of model predictive control," *International Journal of Control and Automation*, vol. 3, no. 4, pp. 47–63, 2010.

[4] B. Houska, H. J. Ferreau, and M. Diehl, "Acado toolkit—an open-source framework for automatic control and dynamic optimization," *Optimal Control Applications and Methods*, vol. 32, no. 3, pp. 298–312, 2011.

[5] M. Giftthaler, M. Neunert, M. Stäuble, and J. Buchli, "The control toolbox—an open-source c++ library for robotics, optimal and model predictive control," in *2018 IEEE International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR)*. IEEE, 2018, pp. 123–129.

[6] U. Nagarajan, G. Kantor, and R. Hollis, "The ballbot: An omnidirectional balancing mobile robot," *The International Journal of Robotics Research*, vol. 33, no. 6, pp. 917–930, 2014.

[7] P. Fankhauser and C. Gwerder, "Modeling and control of a ballbot," B.S. thesis, Eidgenössische Technische Hochschule Zürich, 2010.

[8] M. Kumagai and T. Ochiai, "Development of a robot balancing on a ball," in *2008 International Conference on Control, Automation and Systems*. IEEE, 2008, pp. 433–438.

[9] U. Nagarajan, G. Kantor, and R. Hollis, "Integrated planning and control for graceful navigation of shape-accelerated underactuated balancing mobile robots," in *International Journal of Robotics Research*, vol. 32, no. 9-10, September 2013, pp. 1005–1029.

[10] T. Lauwers, G. Kantor, and R. Hollis, "One is enough!" in *Proc. Int'l. Symp. for Robotics Research*. San Francisco: Int'l. Foundation for Robotics Research, October 12-15 2005.

[11] U. Nagarajan, "Dynamic constraint-based optimal shape trajectory planner for shape-accelerated underactuated balancing systems," in *Robotics: Science and Systems*, Zaragoza, Spain, June 27-31 2010.

[12] M. Shomin and R. Hollis, "Fast, dynamic trajectory planning for a dynamically stable mobile robot," in *IEEE/RSJ Int'l. Symposium on Robotics and Systems (IROS)*, September 2014.
[13] T. K. Jespersen, M. al Ahdab, F. M. Juan de Dios, M. R. Damgaard, K. D. Hansen, R. Pedersen, and T. Bak, "Path-following model predictive control of ballbots," in *2020 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2020, pp. 1498–1504.
[14] M. Neunert, C. De Crousaz, F. Furrer, M. Kamel, F. Farshidian, R. Siegwart, and J. Buchli, "Fast nonlinear model predictive control for unified trajectory optimization and tracking," in *2016 IEEE international conference on robotics and automation (ICRA)*. IEEE, 2016, pp. 1398–1404.
[15] A. Manzoori and G. Vossoughi, "Control of quadrotors for tracking and landing on a mobile platform," in *2018 6th RSI International Conference on Robotics and Mechatronics (IcRoM)*. IEEE, 2018, pp. 46–52.
[16] S. Anoop and K. R. Sharma, "Model predictive control: Simulation studies for the implementation on vertical take-off and landing lab prototype," *Procedia computer science*, vol. 143, pp. 663–670, 2018.
[17] G. Ganga and M. M. Dharmana, "Mpc controller for trajectory tracking control of quadcopter," in *2017 International Conference on Circuit, Power and Computing Technologies (ICCPCT)*. IEEE, 2017, pp. 1–6.
[18] M. Faessler, A. Franchi, and D. Scaramuzza, "Differential flatness of quadrotor dynamics subject to rotor drag for accurate tracking of high-speed trajectories," *IEEE Robotics and Automation Letters*, vol. 3, no. 2, p. 620–626, Apr 2018. [Online]. Available: http://dx.doi.org/10.1109/LRA.2017.2776353
[19] D. Mellinger and V. Kumar, "Minimum snap trajectory generation and control for quadrotors," in *2011 IEEE International Conference on Robotics and Automation*, 2011, pp. 2520–2525.
[20] M. Kamel, M. Burri, and R. Siegwart, "Linear vs nonlinear mpc for trajectory tracking applied to rotary wing micro aerial vehicles," 2017.
[21] M. Kamel, K. Alexis, M. Achtelik, and R. Siegwart, "Fast nonlinear model predictive control for multicopter attitude tracking on so(3)," in *2015 IEEE Conference on Control Applications (CCA)*, 2015, pp. 1160–1166.
[22] D. Hentzen, T. Stastny, R. Siegwart, and R. Brockers, "Disturbance estimation and rejection for high-precision multirotor position control," 2019.
[23] B. Stellato, G. Banjac, P. Goulart, A. Bemporad, and S. Boyd, "Osqp: An operator splitting solver for quadratic programs," *Mathematical Programming Computation*, pp. 1–36, 2020.
[24] MATLAB, *R2019b*. Natick, Massachusetts: The MathWorks Inc., 2019.
[25] H. J. Ferreau, C. Kirches, A. Potschka, H. G. Bock, and M. Diehl, "qpoases: A parametric active-set algorithm for quadratic programming," *Mathematical Programming Computation*, vol. 6, no. 4, pp. 327–363, 2014.
[26] Stanford Artificial Intelligence Laboratory et al., "Robotic operating system." [Online]. Available: https://www.ros.org