TinyMPC: A Model Predictive Control Framework for Embedded Applications

Khai Nguyen, Anoushka Alavilli, and Sam Schoedel

***Abstract*— We introduce a novel model predictive control (MPC) framework designed for running on low-resource hardware. Our approach utilizes an Augmented Lagrangian-iLQR optimization method to handle trajectory optimization and constraint management. The framework is optimized to run on several microcontrollers, including the Teensy 4.1, STM32F401, and the Crazyflie 2.1 drone, which uses an STM32F405 processor. We detail the algorithm implementation, hardware optimization techniques, and provide a comparison of the framework's runtime performance with other state-of-the-art solvers. Additionally, we present the results of testing the framework on the Crazyflie 2.1 drone, which shows that it can solve the full-state quadrotor problem with a minimum frequency of 16Hz, while a complementary LQR is implemented to stabilize the drone at 50Hz.**

I. Introduction

In recent years, significant strides have been made in the field of robotics, particularly in deep learning (DL) and reinforcement learning (RL), resulting in a range of improved capabilities and applications [5] [2] [3]. However, these advanced techniques often demand significant computing resources, which can pose challenges for many robotic applications. This stands in contrast to a significant portion of robotics that utilizes embedded systems with severely limited computing resources. These systems include palm-sized toy robots like the Crazyflie quadrotor [4] and the Petoi Bittle quadruped [9], as well as mission-critical robots like NASA's Mars Perseverance rover [10]. The Crazyflie and Bittle employ affordable and accessible commercial microcontrollers, such as families of STM32 and Teensy, respectively. Meanwhile, the Perseverance rover relies on RAD750, a radiation-hardened version of the PowerPC 750 processor from the previous century. Clearly, the need for resource-constrained robotic applications is widespread. Operating within a tight computational budget, these embedded systems require innovative approaches to enable the efficacy of advanced techniques.

MPC is a popular method for controlling complex robotic systems, which has achieved remarkable success in recent years due to the miniaturization of powerful computational platforms [6], [12], [7]. This method formulates the control problem into an optimization problem with specific objectives and constraints, enabling the achievement of desired behaviors while accounting for complex dynamics, torque limits, and obstacle avoidance. Like every other control module, the computation time from sensor feedback to control signals must be small enough to stabilize the system without any significant latency. Hence, these optimization problems must be solved at high frequencies (tens to hundreds of Hertz) on the robot, a reliable and efficient solver algorithm is the core of successful MPC implementation.

For optimal control problems, convex optimization offers the advantage of finding a global optimal solution, or determining whether a solution exists, even in the presence of constraints. Numerous numerical techniques have been implemented in both open-source and proprietary software, including popular solvers such as SNOPT, Ipopt, OSQP, KNITRO, and ALTRO. Despite their effectiveness, limited computing resources pose a challenge for these solvers and their model predictive control (MPC) implementations. Many MPC approaches employ direct transcription methods, which involve large matrices and can be computationally expensive. To address this issue, high-performance MPC implementations use specialized sparse-matrix routines to exploit the problem structure and efficiently leverage warm-start strategies. One such implementation is ALTRO [11], which employs an Augmented Lagrangian Iterative LQR approach and has demonstrated excellent performance

While some MPC implementations have been designed for FPGAs and microcomputers like Jetson, they are primarily focused on process plants and not highly dynamic systems like robots. Additionally, some MPC implementations rely on Matlab/Simulink to generate C code. However, some of these implementations have not yet been applied to real-world control applications and are only tested on small problems through simulation. Therefore, there is a need for MPC implementations that can be deployed on resource-constrained hardware and have been tested in real-world control applications for highly dynamic systems like robots.

We present TinyMPC, a novel and lightweight solution that offers a user-friendly interface for implementing Model Predictive Control (MPC) on embedded systems with extremely limited resources. To achieve this, we build on the foundation established by [1] and [11] to develop an efficient and effective Augmented Lagrangian-based solver in the C programming language. The implementation includes optimized linear algebra routines, variable step sizes, and infinite horizon Linear Quadratic Regulator terminal cost. Our solution not only provides a speedy solver but also a complete framework that can be readily deployed for a range of control applications.

Our contributions to the field of optimal control are three-fold. Firstly, we have developed a highly efficient and extensible open-source optimal control solver in C. Secondly, we have created a user-friendly MPC interface that simplifies the setup and compilation of our solver on various hardware platforms. Finally, we have provided benchmark results to

demonstrate the efficacy of our approach and a hardware demonstration to showcase its practicality.

This paper is organized as follows. In Section II, we provide background information on optimal control and introduce the LQR method. In Section III, we present our Augmented Lagrangian LQR algorithm for solving optimal control problems. Section IV details the implementation of the TinyMPC framework. In Section V, we discuss the results of our experiments. Finally, we summarize our work and suggest directions for future research in Section VI.

II. Background

A. Notation

We denote $x \in \mathbb{R}^n$ as the state and $u \in \mathbb{R}^m$ as the control input. For a finite horizon length N, we will have N states and $N-1$ control inputs and denote, for example, the state at the k^{th} time step as x_k.

B. Nominal Trajectory

At a high level, the objective in trajectory optimization is to track a pre-specified nominal trajectory while obeying a given a set of state and input constraints. Assume we are interested in a time period T, which can be discretized by N intervals of dt. We have the following:

$\bar{X} = \{\bar{x}_0, \ldots, \bar{x}_N\}$, which is our nominal state trajectory, and $\bar{U} = \{\bar{u}_0, \ldots, \bar{u}_{N-1}\}$, which is our nominal input trajectory.

These can be obtained from a higher-level planner using sample-based, grid-based, or optimization-based methods. These trajectories may or may not be dynamically feasible (i.e. satisfies or does not satisfy (1)) and may or may not obey all input constraints.

C. Linearized Dynamics

We define general, nonlinear dynamics of a system as follows:

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

From this, we can define the difference between the actual and nominal trajectory as $\delta x_k = x_k - \bar{x}_k, \delta u_k = u_k - \bar{u}_k$.

We locally approximate the nonlinear dynamics with a first-order Taylor expansion with respect to the nominal trajectory

$$\begin{aligned} x_{k+1} &= \bar{x}_{k+1} + \delta x_{k+1} && (2)\\ &= f(\bar{x}_k + \delta x_k, \bar{u}_k + \delta u_k) && (3)\\ &\approx f(\bar{x}_k, \bar{u}_k) + \left.\frac{\partial f}{\partial x}\right|_{\bar{x}_k, \bar{u}_k} \delta x_k + \left.\frac{\partial f}{\partial u}\right|_{\bar{x}_k, \bar{u}_k} \delta u_k && (4) \end{aligned}$$

with

$$A_k \equiv \left.\frac{\partial f}{\partial x}\right|_{\bar{x}_k, \bar{u}_k}, \quad B_k \equiv \left.\frac{\partial f}{\partial u}\right|_{\bar{x}_k, \bar{u}_k} \tag{5}$$

Then,

$$\bar{x}_{k+1} + \delta x_{k+1} = f(\bar{x}_k, \bar{u}_k) + A_k \delta x_k + B_k \delta u_k \tag{6}$$

Now, we have two options to represent state and input, i.e. absolute x_k or delta δx_k. This will lead to two different formulations for our optimal control problem as well. In our implementation, we started with the absolute formulation but moved to the delta formulation in order to add line search.

Delta formulation:

$$\begin{aligned} \delta x_{k+1} &= A_k \delta x_k + B_k \delta u_k + f(\bar{x}_k, \bar{u}_k) - \bar{x}_{k+1} \\ \delta x_{k+1} &= A_k \delta x_k + B_k \delta u_k + f_k \end{aligned} \tag{7}$$

Absolute formulation:

$$\begin{aligned} x_{k+1} &= A_k x_k + B_k u_k + f(\bar{x}_k, \bar{u}_k) - A_k \bar{x}_k - B_k \bar{u}_k \\ x_{k+1} &= A_k x_k + B_k u_k + f_k \end{aligned} \tag{8}$$

In both formulations, we denote f_k as the affine term in the linearized dynamics. In (7), a nonzero f_k implies the dynamical infeasibility of the nominal trajectory; otherwise, it is zero. In (8), f_k includes the dynamics linearization error as well. The dynamics Jacobians A_k and B_k only depend on nominal trajectory therefore the derived system is basically linear time-varying and can be precomputed, which reduces the number of online calculations in our real-time system.

D. Cost Function

The tracking linear-quadratic cost (excluding the mixed and constant terms) is as follows:

$$J = \ell_f(x_N) + \sum_{k=1}^{N-1} \ell(x_k, u_k) \tag{9}$$

$$\begin{aligned} &= \frac{1}{2}(x_N - \bar{x}_N)^\intercal Q_f (x_N - \bar{x}_N) \\ &\quad + \sum_{k=1}^{N-1} \frac{1}{2}(x_k - \bar{x}_k)^\intercal Q_k (x_k - \bar{x}_k) \\ &\quad + \frac{1}{2}(u_k - \bar{u}_k)^\intercal R_k (u_k - \bar{u}_k) \end{aligned} \tag{10}$$

$$\begin{aligned} &= \frac{1}{2} x_N^\intercal Q_f x_N + q_f^\intercal x_N \\ &\quad + \sum_{k=1}^{N-1} \frac{1}{2} x_k^\intercal Q_k x_k \\ &\quad + \frac{1}{2} u_k^\intercal R_k u_k + q_k^\intercal x_k + r_k^\intercal u_k \end{aligned} \tag{11}$$

where $q_f = -Q_f \bar{x}_N, q_k = -Q_k \bar{x}_k, r_k = -R_k \bar{u}_k$

E. Dynamic Programming Problem

Cost-to-go is supposed to be in the linear-quadratic form

$$V_k(x_k) = \frac{1}{2} x_k^\intercal P_k x_k + p_k^\intercal x_k \tag{12}$$

At terminal state, $P_N = Q_f, p_N = q_f$

Bellman equation:

$$\begin{aligned} V_k &= \min_{u_k} \{\ell(x_k, u_k) + V_{k+1}(f(x_k, u_k))\} \\ &= \min_{u_k} \{Q_k(x_k, u_k)\} \end{aligned} \tag{13}$$

Action-value function:

$$Q_k(x_k, u_k) = \frac{1}{2} x_k^\intercal Q_k x_k + \frac{1}{2} u_k^\intercal R_k u_k + q_k^\intercal x_k + r_k^\intercal u_k + \frac{1}{2}(A_k x_k + B_k u_k + f_k)^\intercal P_{k+1}(A_k x_k + B_k u_k + f_k) + p_{k+1}^\intercal (A_k x_k + B_k u_k + f_k) \quad (14)$$

Group Q_k into linear and quadratic terms:

$$Q_k(x_k, u_k) = \frac{1}{2} \begin{bmatrix} x_k \\ u_k \end{bmatrix}^\intercal \begin{bmatrix} Q_{xx} & Q_{xu} \\ Q_{ux} & Q_{uu} \end{bmatrix} \begin{bmatrix} x_k \\ u_k \end{bmatrix} + \begin{bmatrix} Q_x \\ Q_u \end{bmatrix}^\intercal \begin{bmatrix} x_k \\ u_k \end{bmatrix} \quad (15)$$

$$\begin{aligned} Q_x &= q_k + A_k^\intercal (P_{k+1} f_k + p_{k+1}) \\ Q_u &= r_k + B_k^\intercal (P_{k+1} f_k + p_{k+1}) \\ Q_{xx} &= Q_k + A_k^\intercal P_{k+1} A_k \\ Q_{uu} &= R_k + B_k^\intercal P_{k+1} B_k \\ Q_{ux} &= B_k^\intercal P_{k+1} A_k = Q_{xu}^\intercal \end{aligned} \quad (16)$$

Apply the necessary optimality condition to achieve optimal control:

$$0 = \frac{\partial Q_k}{\partial u_k} = Q_{uu} u_k + Q_{ux} x_k + Q_u \quad (17)$$

Problem (13) has a closed-form solution:

$$\begin{aligned} u_k^* &= -K_k x_k - d_k \\ d_k &= Q_{uu}^{-1} Q_u \\ K_k &= Q_{uu}^{-1} Q_{ux} \end{aligned} \quad (18)$$

Plug these back into (14) to get the cost-to-go:

$$\begin{aligned} p_k &= Q_x + K_k^\intercal Q_{uu} d_k - K_k^\intercal Q_u - Q_{ux}^\intercal d_k \\ P_k &= Q_{xx} + K_k^\intercal Q_{uu} K_k - 2 K_k^\intercal Q_{ux} \end{aligned} \quad (19)$$

All of these result in Riccati recursion (backward pass) to calculate cost-to-go and control gains. Because the system is locally linear, we only need to perform one iteration to obtain optimality.

F. Forward Pass

Following the delta formulation, the forward pass is as below:

$$u_k \leftarrow u_k - K_k(\bar{x}_k - x_k) - d_k \quad (20)$$

$$x_{k+1} \leftarrow A_k x_k + B_k u_k + f_k \quad (21)$$

III. Constrained Optimal Control Problem

Up to this point, we have not considered any constraints on the state x or input u. Real-world systems have many types of physical or desired limitations. Some popular constraints are box constraints on the state and input at each time step (to ensure that the solution is realizable on hardware and to encourage smoothness in the solution), equality constraints on the goal state (to ensure that that a desired final position is reached), and second-order cone constraints to, for example, enforce thrust limits.

To handle constraints, we use the augmented Lagrangian method (ALM) which enforces constraints into the cost function. Generally, our tracking problem is convex, and a global solution can be found with only one constrained backward pass. However, multiple iterations are needed to satisfy dual feasibility.

Below is the pseudo-code of our AL-TVLQR algorithm.

Algorithm 1: Augmented Lagrangian TVLQR

```
initialization;
for i ← 0 to maxIters do
    K, k, P, p ← ConstrainedBackwardPass;
    X, U ← ForwardPass;
    CalculateConstraintViolation();
    if augmented Lagrangian gradient is zero then
        return;
    end
    UpdateDuals();
    UpdatePenalty();
    if objective function KKT conditions are met then
        return;
    end
end
```

A. Augmented Lagrangian Cost Function

First, consider linear equality and inequality constraints as follows:

$$h_k^x(x) = H_k^x x_k - h_{0k}^x = 0, \quad (22)$$

$$g_k^x(x) = G_k^x x_k - g_{0k}^x \leq 0 \quad (23)$$

$$h_k^u(u) = H_k^u u_k - h_{0k}^u = 0, \quad (24)$$

$$g_k^u(u) = G_k^u u_k - g_{0k}^u \leq 0 \quad (25)$$

Let's look at the constrained backward pass. In essence, the ALM method will add Lagrangian and penalty terms into the original cost function, turning it into an unconstrained optimization problem. Interestingly, these linear-quadratic terms fit nicely into the LQR formulation.

The augmented Lagrangian applies to the action-value function (assuming only constraints on u_k)

$$Q_k(x_k, u_k, \lambda_k, \mu_k) \leftarrow Q_k + \lambda_k^\intercal h_k(u_k) + \frac{1}{2}\rho h_k(u_k)^\intercal h_k(u_k) + \mu_k^\intercal g_k(u_k) + \frac{1}{2} g_k(u_k)^\intercal I_\rho g_k(u_k) \quad (26)$$

Apply the necessary optimality condition to achieve opti-

mal control:

$$0 = \frac{\partial Q_k}{\partial u_k} = Q_{uu}u_k + Q_{ux}x_k + Q_u + \lambda_k^\intercal \frac{\partial h_k}{\partial u_k} + \rho h_k(u_k)^\intercal \frac{\partial h_k}{\partial u_k} + \mu^\intercal \frac{\partial g_k}{\partial u_k} + g_k(u_k)^\intercal I_\rho + \frac{\partial g_k}{\partial u_k} \quad (27)$$

Substitute the linear constraint cases (24) and (25) into (27) to obtain:

$$\begin{aligned} 0 &= Q_{uu}u_k + Q_{ux}x_k + Q_u + \lambda_k^\intercal H_k + \rho(H_k u_k - h_{0k})^\intercal H_k \\ &\quad + \mu_k^\intercal G_k + (G_k u_k - g_{0k})^\intercal I_\rho G_k \\ &= (Q_{uu} + \rho H_k^\intercal H_k + G_k^\intercal I_\rho G_k)u_k + Q_{ux}x_k \\ &\quad + [Q_u + H_k^\intercal(\lambda_k - \rho h_{0k}) + G_k^\intercal(\mu_k - I_\rho g_{0k})] \end{aligned} \quad (28)$$

The modification to the backward pass is as follows:

$$\begin{aligned} Q_u &= r_k + B_k^\intercal(P_{k+1}f_k + p_{k+1}) + H_k^\intercal(\lambda_k - \rho h_{0k}) + \\ &\quad G_k^\intercal(\mu_k - I_\rho g_{0k}) \\ Q_{uu} &= R_k + B_k^\intercal P_{k+1} B_k + \rho H_k^\intercal H_k + G_k^\intercal I_\rho G_k \end{aligned} \quad (29)$$

Note that the same modification would apply for state constraint Q_x, Q_{xx} and terminal state P_N, p_N. We may have mixed state-input constraints as well.

If the constraints are nonlinear, we can linearize them about the nominal trajectory and obtain the forms (22) and (25) which fits (27).

B. Dual Updates

Eq. (29) suggests the dual updates as:

$$\lambda_k \leftarrow \lambda_k - \rho h_{0k}, \quad (30)$$

$$\mu_k \leftarrow \max(0, \mu_k - I_\rho g_{0k}) \quad (31)$$

Note the subtle difference between AL here and in iterative LQR (iLQR). In iLQR, one will naturally approximate the cost with second-order Taylor expansion about the current trajectory, then group the cost in gradient and Hessian terms, not precisely like linear and quadratic terms like ours. Third-rank tensors are ignored. Moreover, one will solve the backward pass and forward pass iteratively until convergence so that any mismatch due to nonlinearity can be eliminated. In (27), constraints have to be in linear form so they can be substituted and grouped into corresponding terms.

C. Conic Constraints

There is a mathematical background in generalized inequality which starts with the conic combination. Typical inequality like (23) can be seen as a type of cone called non-positive orthant. In fact, the steps of dual update (31) can be interpreted as a projection of the new value into the dual cone (non-positive orthant). Most of the cones we care about are self-dual. This projection operator helps drive the duals back to the constraint manifold.

IV. The TinyMPC Framework

A. Jacobian Code Generation

Functions to compute state and control matrices were generated using `Symbolics.jl` in Julia by symbolically compute the Jacobians of the given robot's dynamics function with respect to the state control variables. These symbolic functions were converted to C functions. Reference trajectories were generated using ALTRO and similarly copied into C code format for use on a microcontroller.

B. Linear Algebra Library

We use the SLAP (Simple Linear Algebra Protocols) library developed by Dr. Brian Jackson for matrix computations. SLAP was chosen because it is tested and lightweight and because of our architectural decision to have as few dependencies as possible. We modified SLAP, as discussed in V-A.1, to replace the backend of some of the most commonly used functions with existing implementations that are much faster and can be precompiled for use with SLAP.

V. Results

After having verified the correctness of our algorithm by comparing it with the solution from ALTRO in Julia, we chose to implement our algorithm on three common and accessible microcontrollers as a demonstration of the range of embedded systems on which our solution can run. We chose to implement our algorithm on a Teensy 4.1, an STM NucleoF401RE, and an STMF405 on a Crazyflie 2.1 drone, shown in Fig. 1. The Crazyflie drone is a common research platform for autonomous and distributed swarm algorithm research. We chose to use it because it is small and thus has fast attitude dynamics that we want to show can be stabilized by TinyMPC.

A. Speed Increases

1) Matrix Library Backend: After profiling our C implementation, we found the algorithm was spending around 77% of its time in the SLAP, the library we used as our matrix processing backend. Specifically, the algorithm was in the matrix add and multiply function around 75% of

Fig. 1. Crazyflie drone used for testing.

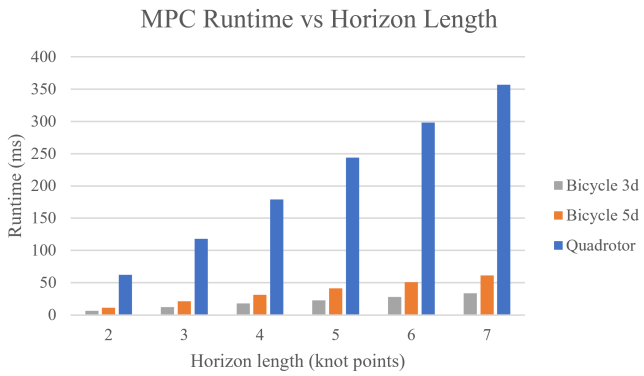

Fig. 2. MPC runtime vs. horizon length using Eigen as the backend to the SLAP linear algebra library.

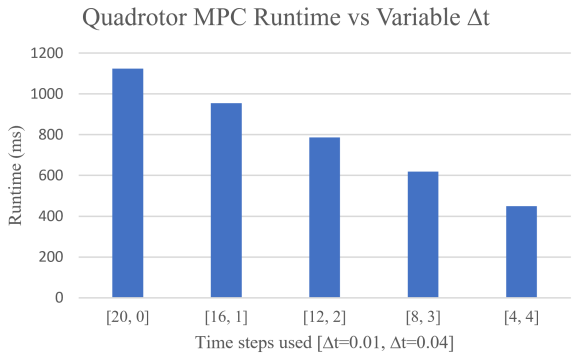

Fig. 3. MPC runtime vs. horizon length using variable sampling times. The format of the x-axis is [number of steps at dt=0.01, number of steps at dt=0.04] The final horizon times are all equal to 0.2 seconds but each are achieved with a different number of knot points.

the time. This function multiplies two matrices, multiplies them by a scalar, then sums the result with a third matrix multiplied by a second scalar to produce a final result. To speed up our implementation, we replaced the original SLAP implementation with Eigen 3.1 matrices. This adds a new dependency to our implementation, but since it is added as a backend to the SLAP library, users can simply download a precompiled version of SLAP and not need to worry about downloading and compiling Eigen themselves. Replacing the SLAP backend with Eigen halved the runtime of our MPC function, allowing us to run the dynamic bicycle model in real time (faster than 50Hz) with a horizon length of 4 knot points. As shown in Fig. 2, this speed increase is still not enough to run the quadrotor at 50Hz, even with only two knot points in the horizon.

2) Variable Sampling Time: We take advantage of the constantly updating nature of MPC and the fact that knot points farther into the future do not have to be predicted as accurately as those closer to the current state of the robot by increasing the time step for later knot points. This allows the MPC algorithm to cover the same amount of horizon time using fewer knot points, decreasing runtime while achieving similar performance. This works primarily because the fast dynamics of the system need to be stabilized immediately while obstacle avoidance, which relies more on computing the position of the robot, generally has slower dynamics and thus can be computed using a larger time step. On the quadrotor, this was implemented by linearizing two dynamics functions about hover, one with the initial fast time step and one with the slow time step used for the remainder of the horizon time. Each of the datapoints in Fig. 3 correspond to combinations of Δt that equal 0.2 seconds. As can be seen in the graph, it takes 1.1 seconds to compute one MPC step with a horizon time of 0.2 seconds and a time step of 0.01 seconds. Replacing the last four time steps with a single time step of 0.04 seconds, we see that it now takes only 950 milliseconds to compute one MPC step. This trend continues linearly down to the extreme case of four initial steps at $\Delta t = 0.01$ and four more steps at $\Delta t = 0.04$, which takes around 450 milliseconds to solve. The percentage computation speed increase can be computed as $n_0/(n_0 - n_2(\Delta t_2/\Delta t_1 - 1))100\%$, where n_0 is the number of steps required to reach the desired horizon time using only Δt_1, and n_2 is the number of steps using Δt_2. Similarly, n_1 is the number of steps using Δt_1, but is not required to compute the relative speed-up from using more steps at Δt_2.

B. Teensy 4.1

The results in Fig. 2 and 3 show results from running on the STM NucleoF401RE. The F401RE uses a Cortex M4 processor running at a maximum clock frequency of 16 MHz. The function runtimes from this processor are far too slow to be run on the Crazyflie quadrotor. The exact same code was run on a Teensy 4.1 in a fraction of the time, as shown in Fig. 4. The Teensy 4.1 uses a Cortex M7 chip that, in our tests, was run at a clock frequency of 600 MHz. This is the highest natively supported clock frequency that does not require active cooling. The Teensy ran our MPC code in under 1 millisecond for each horizon length shown in the figure. With 130 knot points for the horizon length, the Teensy ran the function in 20 milliseconds, which is an MPC runtime frequency of 50Hz. This is the same frequency used to control the drone using LQR, and is promising given we're able to run the Cortex M4 on the Crazyflie at the speed of the Teensy 4.1.

C. Crazyflie 2.1

Because our MPC algorithm does not yet run fast enough to control a quadrotor, we opted to solve for the infinite horizon LQR gains for the model we obtained from Bitcraze, the developers of the Crazyflie, and implement our own controller on their software stack. Directly copying the optimal gain matrix obtained from infinite horizon LQR caused the drone to exhibit a stable, sinusoidal wobbling behavior during flight. This wobbling was removed by hand-tuning the optimal gain matrix. We then generated position and velocity knot points for a figure-8 trajectory we wanted the drone to follow. Although the drone did not follow the trajectory properly, it did stay in the air for the duration of the trajectory.

VI. Conclusion

The hardware results above highlight the computational complexities of running an MPC algorithm in real-time on an underpowered micro controller. We have demonstrated that our algorithm may run in real-time on devices like the Teensy 4.1 which runs on a Cortex M7 processor running at up to 1GHz. There is still work to do to demonstrate these results on power and resource constrained devices such as the Crazyflie 2.1 running a Cortex M4 chip.

We have not yet reduced the runtime of our algorithm to the point where it may be run on the Crazyflie's processor in real-time, but there are a few things we can implement to speed up our algorithm.

Our future work is proposed as below:

1) Algorithm: We plan to create a new version of the solver using ADMM rather than AL to help reduce the required online computation. ADMM will require fewer matrix-matrix multiplications (from matrix factorizations), which is primarily what is slowing down our code.

2) Conic Constraints: We have attempted to incorporate conic constraint handling, but this is still a work in progress.

3) Model Hierarchy Predictive Control: An additional method implemented to increase performance is a hierarchical dynamics model scheme [8], where the first few time steps use the full model dynamics and the remainder use increasingly simplified versions of the robot's dynamics. This can be done for similar reasons as variable sampling time, discussed in V-A.2. Doing this reduces the computational complexity of the Jacobians for later time steps which decreases overall runtime. Time steps closer to the first horizon knot point require higher-accuracy dynamics to correctly determine the robot's future state, but later time steps only need to look at low frequency dynamics since the higher frequency dynamics will likely be incorrectly estimated anyway. These low frequency dynamics tend to be attributed to much simpler models, such as point mass models, and can be used to determine a subset of the robot's state farther into the future, such as its center of mass. Often, center of mass is all that is required when trying to avoid obstacles farther into the future.

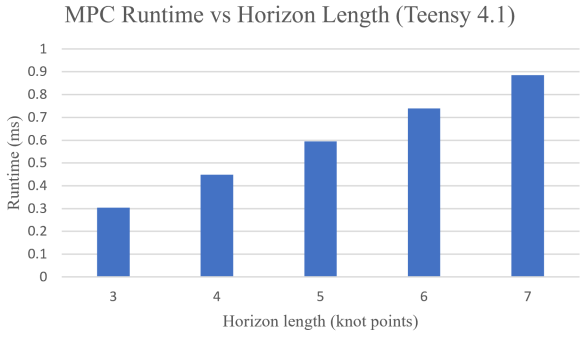

Fig. 4. MPC runtime vs. horizon length on a Teensy 4.1 running at 600 MHz.

4) Code Generation: Following applications like OSQP, Simulink, and SLinGen, we would like to be able to generate C code from a higher level programming language. The overhead of extra work required to do this pays off in the form of a much simpler user interface and being able to optimize for a wide variety of specific platforms using a single program.

TinyMPC is available at `https://github.com/RoboticExplorationLab/TinyMPC`.

References

[1] B. Jackson et al. *ALTRO-C: A Fast Solver for Conic Model-Predictive Control*. URL: `https://ieeexplore.ieee.org/document/9561438`. (accessed: 5.1.2023).

[2] Laura Smith et al. *Learning and Adapting Agile Locomotion Skills by Transferring Experience*. URL: `https://doi.org/10.48550/arXiv.2304.09834`. (accessed: 5.9.2023).

[3] Tuomas Haarnoja et al. *Learning Agile Soccer Skills for a Bipedal Robot with Deep Reinforcement Learning*. URL: `https://doi.org/10.48550/arXiv.2304.13653`. (accessed: 5.9.2023).

[4] Wojciech Giernacki et al. *Crazyflie 2.0 Quadrotor as a Platform for Research and Education in Robotics and Control Engineering*. URL: `https://www.bitcraze.io/papers/giernacki_draft_crazyflie2.0.pdf`. (accessed: 5.9.2023).

[5] Xiaoyu Huang et al. *Creating a Dynamic Quadrupedal Robotic Goalkeeper with Reinforcement Learning*. URL: `https://doi.org/10.48550/arXiv.2210.04435`. (accessed: 5.9.2023).

[6] Faouzi Bouani Amira Kheriji Abbes and Mekki Ksouri. *A Microcontroller Implementation of Constrained Model Predictive Control*. URL: `https://www.idc-online.com/technical_references/pdfs/electrical_engineering/A%20Microcontroller.pdf`. (accessed: 5.4.2023).

[7] John M. Carson III Behçet Açıkmese and Lars Blackmore. *Lossless Convexification of Nonconvex Control Bound and Pointing Constraints of the Soft Landing Optimal Control Problem*. URL: `http://www.larsblackmore.com/iee_tcst13.pdf`. (accessed: 5.9.2023).

[8] Robert J. Frei He Li and Patrick M. Wensing. *Model Hierarchy Predictive Control of Robotic Systems*. URL: `https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9361258`. (accessed: 5.5.2023).

[9] Petoi. *Open Source, Programmable Robot Dog Bittle*. Available at `https://www.petoi.com/pages/bittle-open-source-bionic-robot-dog` (5.9.2023).

[10] David Artz Richard Berger and Paul Kapcio. *RAD750TM Radiation Hardened PowerPCTM Microprocessor*. URL: https://caxapa.ru/thumbs/440955/download.pdf. (accessed: 5.9.2023).

[11] B. Jackson T. Howell and Z. Manchester. *ALTRO: A Fast Solver for Constrained Trajectory Optimization*. URL: https://ieeexplore.ieee.org/document/8967788. (accessed: 5.1.2023).

[12] Trieu Minh Vu. *Model Predictive Control for Autonomous Driving Vehicles*. URL: https://www.mdpi.com/2079-9292/10/21/2593. (accessed: 5.9.2023).