

Code Generation and Conic Constraints for Model-Predictive Control on Microcontrollers with Conic-TinyMPC

Ishaan Mahajan^{1*}, Khai Nguyen^{2,3*}, Sam Schoedel^{2*}, Elakhya Nedumaran², Moises Mata¹,
Brian Plancher^{4,5}, and Zachary Manchester^{2,3}

Abstract—Model-predictive control (MPC) is a state-of-the-art control method for constrained robotic systems, yet deployment on resource-limited hardware remains difficult. This challenge is magnified by expressive conic constraints, which offer greater modeling power but require significantly more computation than linear alternatives. To address this challenge, we extend recent work developing fast, structure-exploiting, cached solvers for embedded applications based on the Alternating Direction Method of Multipliers (ADMM) to provide support for second-order cones, as well as C++ code generation from Python, MATLAB, and Julia. Microcontroller benchmarks show that our solver provides up to a two-order-of-magnitude speedup, ranging from 10.6x to 142.7x, over state-of-the-art embedded solvers on QP and SOCP problems, and enables us to fit order-of-magnitude larger problems in memory. We validate our solver’s deployed performance through simulation and hardware experiments, including trajectory tracking with conic constraints on a 27g Crazyflie quadrotor. Our open-source code is available at <https://tinympc.org>.

I. INTRODUCTION

Model Predictive Control (MPC) is an algorithmic approach that enables highly dynamic online control for robots subject to actuator and state constraints [1]–[4]. However, while MPC has been deployed quite successfully in both academia and industry, its application is often hindered by computational limitations. This challenge is amplified when dealing with tiny, low-cost, low-power robots, as their onboard microcontroller units (MCUs) feature orders-of-magnitude less RAM, flash memory, and processor speed compared to the CPUs and GPUs available on larger robots [5]–[7]. Consequently, many examples of intelligent behaviors executed on these tiny platforms rely on off-board compute [8]–[12]. For deployment on such limited computational platforms, which also often lack full hardware support for floating-point arithmetic, an ideal MPC solver should be division-free, use only static memory allocation, and support warm starting to take advantage of computation at previous time steps [13]–[16]. Compiled code should also have a low memory footprint and be easily verifiable through an interface to a high-level language (e.g., Python).

This material is based upon work supported by the National Science Foundation (under Award 2411369). Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect those of the funding organizations.

¹ School of Engineering and Applied Science, Columbia University

² Carnegie Mellon University

³ Massachusetts Institute of Technology

^{4,5} Barnard College, Columbia University and Dartmouth College

*Equal Contribution. Correspondence to: plancher@dartmouth.edu

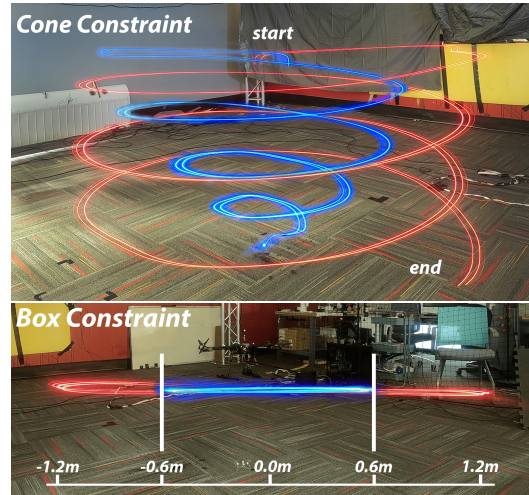


Fig. 1: We demonstrate our solver using a 27 gram nano quadrotor, the Crazyflie. **Top:** we track a descending helical reference (red) with its position subject to a 45° second-order cone glideslope. This requires the aircraft to perform a spiral landing maneuver (blue). **Bottom:** we design a predictive safety filter to guarantee safe maneuvers within a box-shaped space (blue) regardless of the nominal controller behavior (red).

Moreover, while many embedded solvers today focus solely on quadratic programming (QP), second-order cone programs (SOCPs) represent a significantly richer class of tractable convex optimization problems, strictly generalizing linear and quadratic ones while remaining efficiently solvable due to their symmetric, self-dual structure [17], [18]. Many problems that appear nonconvex admit exact or lifted SOCP reformulations [19], [20]. Such problems also arise naturally in robotic and aerospace control problems involving friction, attitude, and thrust limits [21]–[23]. As such, an ideal embedded MPC solver should natively support SOCPs while efficiently addressing their computational demands.

Table I compares commonly used SOCP and QP solvers to illustrate how well they align with these design criteria. Several efficient optimization solvers and techniques suitable for embedded MPC have emerged in recent years [33], [34], with notable software packages including OSQP [28], CVXGEN [35], ECOS [26], and SCS [29]. However, because many of these are not purpose-built for MPC, they either do not easily support warm starting, don’t take advantage of problem or sparsity structure, are not designed to easily enable embedded deployment, or some combination of these issues. In contrast, while TinyMPC [5] is the first MPC solver tailored for dynamic tiny robot control on MCUs, it (as well as OSQP and CVXGEN) only supports QPs. TinyMPC also lacked a convenient high-level programming interface.

TABLE I: Comparison of general-purpose and model predictive control solvers.

Solver	SOC	Warm Starting	Embedded	Open Source	Quad. Obj.	MPC Tailored
Clarabel [24]	✓	✗	✗	✓	✓	✗
COSMO [25]	✓	✓	✗	✓	✓	✗
ECOS [26]	✓	✗	✓	✓	✗	✗
MOSEK [27]	✓	✗	✗	✗	✓	✗
OSQP [28]	✗	✓	✓	✓	✓	✗
SCS [29]	✓	✓	✓	✓	✓	✗
FORCESPRO [30]	✗	✓	✓	✗	✓	✓
ALTRO-C [15]	✓	✓	✗	✓	✓	✓
acados [31]	✗	✓	✗	✓	✓	✓
HPIPM [32]	✗	✓	✗	✓	✓	✓
TinyMPC [5]	✗	✓	✓	✓	✓	✓
Conic-TinyMPC(ours)	✓	✓	✓	✓	✓	✓

To address these shortcomings, in this work, we develop Conic-TinyMPC. Our contributions include: 1) *support for conic constraints*, focusing on SOCPs (Section III), a critical need for many real-world robotics applications; and 2) an *open-source code generation software package* with Python, MATLAB, and Julia interfaces to both ease the deployment of embedded MPC, as well as provide code generation and solution verification examples (Section IV).

We present microcontroller benchmarks (Section V-A) demonstrating up to a two-order-of-magnitude speedup and one-order-of-magnitude reduction in memory usage over state-of-the-art embedded QP and SOCP solvers. We also validate our solver's deployed performance through hardware experiments on a 27g Crazyflie quadrotor (Section V-B), including trajectory tracking with conic constraints. Our open-source code is available at <https://tinympc.org>.

II. BACKGROUND

A. The Linear-Quadratic Regulator

The linear-quadratic regulator (LQR) problem [36] is an optimal control problem in which a quadratic cost function is minimized subject to linear (or affine) dynamics constraints:

$$\min_{x_{1:N}, u_{1:N-1}} J = \frac{1}{2} x_N^\top Q_N x_N + q_N^\top x_N + \sum_{k=1}^{N-1} \left(\frac{1}{2} x_k^\top Q_k x_k + q_k^\top x_k + \frac{1}{2} u_k^\top R_k u_k + r_k^\top u_k \right) \quad (1)$$

$$\text{subject to } x_{k+1} = A_k x_k + B_k u_k + c_k, \quad \forall k \in [1, N),$$

where $x_k \in \mathbb{R}^n$, $u_k \in \mathbb{R}^m$ are the state and control at time step k , N is the number of time steps, $A_k \in \mathbb{R}^{n \times n}$, $B_k \in \mathbb{R}^{n \times m}$, and $c_k \in \mathbb{R}^n$ define the system dynamics, $Q_k \succeq 0$, $R_k \succ 0$, and $Q_N \succeq 0$ are symmetric cost-weighting matrices and q_k and r_k are linear cost vectors. Equation (1) is a classical problem in the field of optimal control whose solution is an affine feedback controller [36]:

$$u_k^* = -K_k x_k - d_k. \quad (2)$$

Feedback and feedback terms (K_k , d_k) are found by solving the discrete-time Riccati equation backward in time, starting with $P_N = Q_N$ and $p_N = q_N$, where P_k and p_k are the quadratic and linear terms of the cost-to-go function [36]:

$$\begin{aligned} K_k &= (R_k + B_k^\top P_{k+1} B_k)^{-1} (B_k^\top P_{k+1} A_k) \\ d_k &= (R_k + B_k^\top P_{k+1} B_k)^{-1} (B_k^\top p_{k+1} + r_k + B_k^\top P_{k+1} c_k) \\ P_k &= Q_k + K_k^\top R_k K_k + (A_k - B_k K_k)^\top P_{k+1} (A_k - B_k K_k) \\ p_k &= q_k + (A_k - B_k K_k)^\top (p_{k+1} - P_{k+1} B_k d_k + P_{k+1} c_k) \\ &\quad + K_k^\top (R_k d_k - r_k). \end{aligned} \quad (3)$$

B. Convex Model-Predictive Control

Convex MPC extends this to admit additional convex constraints on the states and controls (as shown in blue):

$$\begin{aligned} \min_{x_{1:N}, u_{1:N-1}} & J(x_{1:N}, u_{1:N-1}) \\ \text{subject to} & x_{k+1} = A_k x_k + B_k u_k + c_k, \quad \forall k \in [1, N) \\ & x_k \in \mathcal{X}, u_k \in \mathcal{U}, \quad \forall k \in [1, N), \end{aligned} \quad (4)$$

where \mathcal{X} and \mathcal{U} are convex sets. The convexity of this problem means that it can be solved efficiently and reliably, enabling real-time deployment in a variety of control applications, including autonomous rocket landings [37], legged locomotion [38], and autonomous driving [39].

When \mathcal{X} and \mathcal{U} can be expressed as linear constraints, (4) is a QP. When \mathcal{X} and \mathcal{U} can be expressed as both linear and second-order cone constraints, (4) is an SOCP, and can be put into the standard form (where \mathcal{K} is a cone):

$$\begin{aligned} \min_{x \in \mathbb{R}^n} & \frac{1}{2} x^\top P x + q^\top x \\ \text{subject to} & G x \leq h, \quad x \in \mathcal{K}. \end{aligned} \quad (5)$$

The addition of the final constraints in blue separate the SOCP from the QP. Further analysis, including feasibility and stability guarantees can be found in [18], [29].

C. Alternating Direction Method of Multipliers (ADMM)

We provide a very brief summary of ADMM here and refer readers to [40] for more details. Given a generic optimization problem (with f and \mathcal{C} convex):

$$\begin{aligned} \min_x & f(x) \\ \text{subject to} & x \in \mathcal{C}, \end{aligned} \quad (6)$$

we can form the equivalent problem, introducing slack z , and indicator function $I_{\mathcal{C}}$:

$$\begin{aligned} \min_{x, z} & f(x) + I_{\mathcal{C}}(z) \\ \text{subject to} & x = z \end{aligned} \quad I_{\mathcal{C}}(z) = \begin{cases} 0 & z \in \mathcal{C} \\ \infty & \text{otherwise.} \end{cases} \quad (7)$$

The augmented Lagrangian of the transformed problem (7) is (with Lagrange multiplier λ and scalar penalty weight ρ):

$$\mathcal{L}_A(x, z, \lambda) = f(x) + I_C(z) + \lambda^\top(x - z) + \frac{\rho}{2}\|x - z\|_2^2. \quad (8)$$

If we perform alternating minimization of (8) with respect to x and z , we arrive at the three-step ADMM iteration,

$$\text{primal update : } x^+ = \arg \min_x \mathcal{L}_A(x, z, \lambda), \quad (9)$$

$$\text{slack update : } z^+ = \arg \min_z \mathcal{L}_A(x^+, z, \lambda), \quad (10)$$

$$\text{dual update : } \lambda^+ = \lambda + \rho(x^+ - z^+), \quad (11)$$

where the last step is a gradient-ascent update on the Lagrange multiplier [40]. These steps can be iterated until a desired convergence tolerance is achieved.

In the special cases of QPs and SOCPs, each step of the ADMM algorithm becomes very simple to compute: the primal update is the solution to a linear system, the slack update is a linear or conic projection, and the dual update is simply scaled vector addition. As such, the computational complexity of the three steps for QPs and SOCPs is:

- $\mathcal{O}(n^3)$ for the primal update (9),
- $\mathcal{O}(n^2)$ for the slack update (10),
- and $\mathcal{O}(n)$ for the dual update (11).

Due to this simplicity, ADMM-based QP and SOCP solvers have demonstrated state-of-the-art results [28], [29].

D. TinyMPC

TinyMPC [5], exploits properties of the MPC problem through pre-computation and caching with an ADMM framework to efficiently solve this problem via three assumptions:

- 1) The dynamical system can be modeled as linear time invariant, with fixed $A, B, c \forall k \in [0, N]$;
- 2) The quadratic cost can be modeled with fixed Hessians, $Q, R \forall k \in [0, N)$, Q_N ; and
- 3) The finite horizon LQR feedback gain and cost-to-go Hessian, K_k, P_k , can be effectively approximated by the solution to the infinite-horizon LQR solution, $K_{\text{inf}}, P_{\text{inf}} \forall k \in [0, N]$.

We provide a brief summary of the approach and refer to [5] for more details. TinyMPC [5] splits the standard LQR problem from all additional state and input constraints via ADMM. The primal update, (9), becomes:

$$\begin{aligned} \min_{x_{1:N}, u_{1:N-1}} \quad & \frac{1}{2}x_N^\top \tilde{Q}_N x_N + \tilde{q}_N^\top x_N + \\ & \sum_{k=1}^{N-1} \frac{1}{2}x_k^\top \tilde{Q}_k x_k + \tilde{q}_k^\top x_k + \frac{1}{2}u_k^\top \tilde{R} u_k + \tilde{r}^\top u_k \end{aligned} \quad (12)$$

$$\text{subject to } x_{k+1} = Ax_k + Bu_k + c,$$

where we use scaled dual variables y and g and for convenience [40] and the following are defined:

$$\begin{aligned} \tilde{Q}_N &= Q_N + \rho I, & \tilde{Q} &= Q + \rho I, & \tilde{R} &= R + \rho I, \\ \tilde{q}_N &= q_N + \rho(\lambda_N/\rho - z_N) = q_N + \rho(y_N - z_N), \\ \tilde{q}_k &= q_k + \rho(\lambda_k/\rho - z_k) = q_k + \rho(y_k - z_k), \\ \tilde{r}_k &= r_k + \rho(\mu_k/\rho - w_k) = r_k + \rho(g_k - w_k). \end{aligned} \quad (13)$$

This enables a slight simplification to (11), where we substitute the dual variables with their scaled forms and eliminate ρ . As a result, the scaled dual variables y_k and g_k are always equal to the difference between the primal and slack variables, which is a convergence criteria that now does not need to be recalculated during convergence checks.

As (12) has the same form as (1), it can be solved efficiently through a backwards Riccati recursion, (3), followed by an affine dynamics roll-out of the resulting policy, (2). The slack update remains a projection onto the feasible set:

$$\begin{aligned} z_k^+ &= \text{proj}_{\mathcal{X}}(x_k^+ + y_k), \\ w_k^+ &= \text{proj}_{\mathcal{U}}(u_k^+ + g_k), \end{aligned} \quad (14)$$

where the superscript denotes the variable at the subsequent ADMM iteration, and the dual update becomes:

$$\begin{aligned} y_k^+ &= y_k + x_k^+ - z_k^+, \\ g_k^+ &= g_k + u_k^+ - w_k^+. \end{aligned} \quad (15)$$

Given a long enough horizon, the Riccati recursion (3) converges to the solution of the infinite-horizon LQR problem [36]. [5] exploits this property and assumes that the single infinite horizon gain, K_{inf} , and cost-to-go Hessian, P_{inf} , sufficiently approximate the time-varying values, K_k, P_k . Combining this approximation with our assumption of fixed A, B, c, Q, Q_N, R matrices enables us to drastically simplify the Riccati recursion, (3), not only easing its computational complexity, but also greatly reducing its memory footprint as we only need to cache $A, B, c, Q, Q_N, R, K_{\text{inf}}, P_{\text{inf}}$, along with a handful of other precomputed and cached constants:

$$\begin{aligned} C_1 &= (R + B^\top P_{\text{inf}} B)^{-1}, \\ C_2 &= (A - BK_{\text{inf}})^\top, \\ C_3 &= C_1 B^\top P_{\text{inf}} c, \\ C_4 &= C_2 P_{\text{inf}} c. \end{aligned} \quad (16)$$

Using these terms, the LQR backward pass simplifies to:

$$\begin{aligned} d_k &= C_1(B^\top p_{k+1} + r_k) + C_3, \\ p_k &= q_k + C_2 p_{k+1} - K_{\text{inf}}^\top r_k + C_4, \end{aligned} \quad (17)$$

which only requires matrix-vector products to compute, reducing computational complexity of the primal update from $\mathcal{O}(n^3)$ to $\mathcal{O}(n^2)$, drastically reducing online computation time, and avoiding online division entirely. We note that C_3 and C_4 are derived in addition to C_1 and C_2 from [5] to support dynamics with the additional constant term c .

Finally, we note that ADMM solvers like OSQP [28] adaptively scale the penalty term ρ in (8) for performance. However, this requires performing additional matrix factorizations. To avoid this, [5], [16] pre-compute and cache sets of matrices corresponding to several values of ρ , which we refer to as the set $[\rho]$. Online, the solver switches between these values of ρ , and their respective cached matrices, based on the values of the primal and dual residuals using heuristics adapted from OSQP [28].¹

¹We also note that recent work [41] proposes additional schemes to adapt ρ with finer-grained updates. We will integrate this advance into our open-source conic framework in future work.

III. THE CONIC-TINYMPC SOLVER

As noted in [5], the slack update in (10) can be expressed as the operator Π , which projects the slack variable onto its feasible set. More generally, this projection step can be defined for any convex set. Because the ADMM algorithm naturally isolates the projection subproblem, any convex set with a computationally efficient projection operator can be seamlessly incorporated into our framework. Conveniently, many standard convex cones admit simple closed-form projection operators [40]. We demonstrate this by example in the remainder of this section.

We first note that projection onto a linear inequality constraint, or equivalently, projection of a point z to a hyperplane $\mathcal{H} = \{x : \langle x, a \rangle = b\}$, can be written as follows:

$$\Pi_{\mathcal{H}}(z) = \text{proj}_{\mathcal{H}}(z) = z - \frac{\langle z, a \rangle - b}{\|a\|^2} a. \quad (18)$$

For constant bounds on variables, such as the case of position, velocity, or control limits, (l, u) , this can be reduced to a projection onto a set of upper and lower bounds:

$$\Pi_{l,u}(z) = \max(l, \min(u, z)). \quad (19)$$

This projection approach extends to conic problems in the same manner. We can, for example, define the second-order cone (“ice-cream cone”) [18] as follows:

$$\mathcal{K} = \left\{ z \in \mathbb{R}^n \mid z_n \geq \sqrt{z_1^2 + z_2^2 + \dots + z_{n-1}^2} \right\}, \quad (20)$$

The second-order cone also admits a closed-form and compact projection operator:

$$\Pi_{\mathcal{K}}(z) = \begin{cases} 0, & \|v\|_2 \leq -a, \\ z, & \|v\|_2 \leq a, \\ \frac{1}{2} \left(1 + \frac{a}{\|v\|_2} \right) \begin{bmatrix} v \\ \|v\|_2 \end{bmatrix}, & \|v\|_2 > |a|, \end{cases} \quad (21)$$

where $v = [z_1, \dots, z_{n-1}]^T$ and $a = z_n$. Here, $z_i, i = 1, \dots, n$ is any vector subset of the state or control slack variables. In principle, other cones can also be implemented, e.g. the cone of $n \times n$ positive semi-definite matrices (“semi-definite cone”) [18]. Algorithm 1 summarizes the overall algorithm.

IV. CODE GENERATION

To enable the community to more easily leverage Conic-TinyMPC, we have developed a code-generation tool with Python, MATLAB, and Julia interfaces that produces dependency-free C++ code for easy deployment. We hope that through such interfaces, and our additional examples, available alongside our open-source code, the community can quickly prototype and deploy our solver onto their tiny robot systems. In the remainder of this section, we describe our code-generation interfaces through examples and code listings using our Python interface and note that the process is nearly identical in MATLAB and Julia.

Listing 1 shows how to generate problem-specific code. The `setup` function initializes the problem with specific data, namely: time horizon (N), system model (A, B , and c),

Algorithm 1 Conic-TinyMPC

```
function OFFLINE_PRECOMPUTE(input)
for  $\rho \in [\varrho]$  form cache via (13),  $K_{inf}, P_{inf}$ , (16)
function ONLINE_SOLVE(input)
  Select  $\rho \in [\varrho]$  and associated cached terms
  while not converged do
    //Primal Update
     $p_{1:N-1}, d_{1:N-1} \leftarrow$  Backward pass via (17)
     $x_{1:N}, u_{1:N-1} \leftarrow$  Forward pass via (2)
    //Slack and Dual Updates
     $z_{1:N}, w_{1:N-1} \leftarrow$  Projection via (18), (19), or (21)
     $y_{1:N}, g_{1:N-1} \leftarrow$  Gradient ascent (15)
     $q_{1:N}, r_{1:N-1}, p_N \leftarrow$  Update linear cost terms
  return  $x_{1:N}, u_{1:N-1}$ 
```

```
import tinympc
# Create the solver object
solver = tinympc.TinyMPC()
# Initialize the solver
solver.setup(N, A, B, c, Q, R, bnds, socs, options)
# Generate code
solver.codegen(output_dir)
```

Listing 1: A minimal Python script to generate MPC problem code.

```
import numpy as np
import tinympcgen
# Set initial state
tinympcgen.set_x0(np.array([0.5, 0, 0, 0]))
# Solve the problem
solution = tinympcgen.solve()
# Get the solution
controls = solution["controls"]
```

Listing 2: An example Python script to run the generated code.

```
#include "tinympc.hpp"
#include "tiny_data_workspace.hpp"
int main(int argc, char **argv) {
  tiny_solve(&solver); // Solve the problem
  return 0;
}
```

Listing 3: A simple C++ program that loads the problem data from `tiny_data_workspace.hpp` and solves the problem.

```
// Update initial/feedback state
tiny_set_x0(&solver, x0_new);
// Update trajectory reference
tiny_set_x_ref(&solver, xref_new);
// Update Bounds
tiny_set_bound_constraints(&solver, xmin_new,
  xmax_new, umin_new, umax_new);
```

Listing 4: Directly updating parameters of the MPC problem in C++.

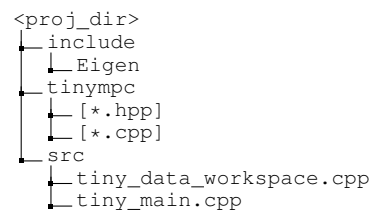


Fig. 2: The tree structure of the generated code. The main program is stored in `tiny_main.cpp`.

cost weights (Q and R), linear and conic constraint parameters (`bnds` and `socs`), and solver options. For example, users may set primal and dual tolerances, or the maximum number of ADMM iterations. This kind of parameter tuning is often critical for returning a usable solution within real-time limits for particular systems of interest. The `codegen` function is then used to generate the custom-tailored code.

Users may choose to compile code for their host system either manually or through our interface for ease of testing when cumbersome to build in C++. Listing 2 shows an example script that loads the generated code library, solves the problem, then retrieves the solution. The reference trajectory and initial state may be set using `set_x_ref`, `set_u_ref`, and `set_x0`, and may be done on the microcontroller using the C++ equivalents. Additional wrapped functions exist for overwriting constraint parameters.

The directory structure of the resulting generated C++ code is shown in Fig. 2. The solver’s source code and associated headers are in the `tinympc` subdirectory. The generated code is compact and does not rely on dynamic memory allocation, making it particularly suitable for embedded use cases. An example program is located in `tiny_main.cpp`. This program imports workspace data from the `tiny_data_workspace.hpp` header and then solves the given problem (Listing 3).

We also offer functions to update the initial state, reference trajectories, and constraints on the states and inputs using wrapper functions, which are essential in MPC settings (see Listing 4 for a number of examples).

We note that the Python interface not only allows users to generate C++ code that may be run on a microcontroller, but it also allows the user to run TinyMPC functions directly in Python. This enables users to investigate the solver in a desktop environment before switching to a microcontroller. In future work, we also hope to build on these Python interfaces to enable us to build a complete MicroPython library, for even easier use on microcontrollers. Finally, we remind the reader that similar features, functions, and interfaces exist through our MATLAB and Julia interfaces.

V. EXPERIMENTS

We benchmark the performance of the generated code from Conic-TinyMPC through a sets of microcontroller benchmarks and control tasks running onboard a 27 gram Crazyflie quadrotor [42] to demonstrate TinyMPC’s effectiveness in real-world deployed conditions. All experiments are available with our open-source code for reproducibility.

A. Microcontroller Benchmarks

1) *Predictive Safety Filtering*: We first formulate a QP with box constraints on states and controls to act as a predictive safety filter for a nominal task policy as in [43], [44]. We compare the solution times and memory usage of Conic-TinyMPC against the state-of-the-art OSQP [28] QP solver, utilizing both solvers’ Python code generation interfaces, while varying the state and horizon dimensions. We benchmark on a STM32F405 Adafruit Feather board,

which has an ARM Cortex-M4 operating at 168 MHz with 1 MB of flash memory and 128 kB of RAM, very similar to computational hardware on the Crazyflie 2.1 used for our later hardware experiments in Section V-B.

Fig. 3a shows the total program size and the average execution times per iteration. Conic-TinyMPC uses drastically less memory and exhibits significant speed-ups over OSQP. For varying states, Conic-TinyMPC achieves up to 20.4× faster execution, while for varying time horizons, it achieves up to 7.2× faster execution. Moreover, the reduction in memory usage allows Conic-TinyMPC to solve real-time optimal control of complex systems with long time horizons. In particular, Conic-TinyMPC was able to handle time horizons of up to 100 knot points, whereas OSQP surpassed the 128 kB memory capacity of the STM32 at a time horizon of only $N = 32$. Additionally, Conic-TinyMPC demonstrated scalability to larger state dimensions up to $n = 32$, whereas OSQP encountered memory limitations beyond $n = 28$.

2) *Rocket Soft-Landing*: The second benchmark is a rocket soft-landing problem which requires a rocket to land with small final velocity at a desired position, resulting in a conic glide-scope constraint. We benchmark the performance of Conic-TinyMPC again via it’s Python code generation against ECOS [26] and SCS [29], state-of-the-art SOCP solvers, using CVXPYgen’s [45] code generation interface. All solver options were set to equivalent values wherever possible and all tolerances were set to 0.01.

Here we benchmark on a Teensy 4.1 [46] development board, which has an ARM Cortex-M7 microcontroller operating at 600 MHz, with 7.75 MB of flash memory, 512 kB of tightly coupled static RAM, and an additional 512 kB of tightly coupled dynamic RAM. The increased compute and memory capacity of the Teensy was particularly important to enable us to benchmark against ECOS and SCS, and enabled us to collect more overall data as the largest SOCP problem involved 2301 decision variables as well as 1530 linear equality constraints, 1530 linear inequality constraints, and 255 second-order cone constraints. However, we note that Conic-TinyMPC, even for this larger problem, could still fit on the more constrained Adafruit Feather used in the prior benchmark, as well as on the constrained MCU found on the Crazyflie 2.1, which we demonstrate via our hardware experiments in Section V-B.

Fig. 3b shows the amount of statically and dynamically allocated memory and the average execution times per iteration for varying time horizon. Conic-TinyMPC outperforms SCS and ECOS in execution time and memory, achieving an average speed-up of 13.8x over SCS and 142.7x over ECOS. Conic-TinyMPC performed no dynamic allocation while SCS and ECOS dynamically allocated the workspace at the beginning due to the use of the CVXPYgen interface, causing them to exceed the total available RAM during execution. Without using the CVXPYgen interface, the dynamically allocated workspace must instead be stored statically, far exceeding the static memory limit. This severely limited SCS and ECOS, with both solvers exceeding total memory limits at $N = 64$, while Conic-TinyMPC can scale to $N = 256$.

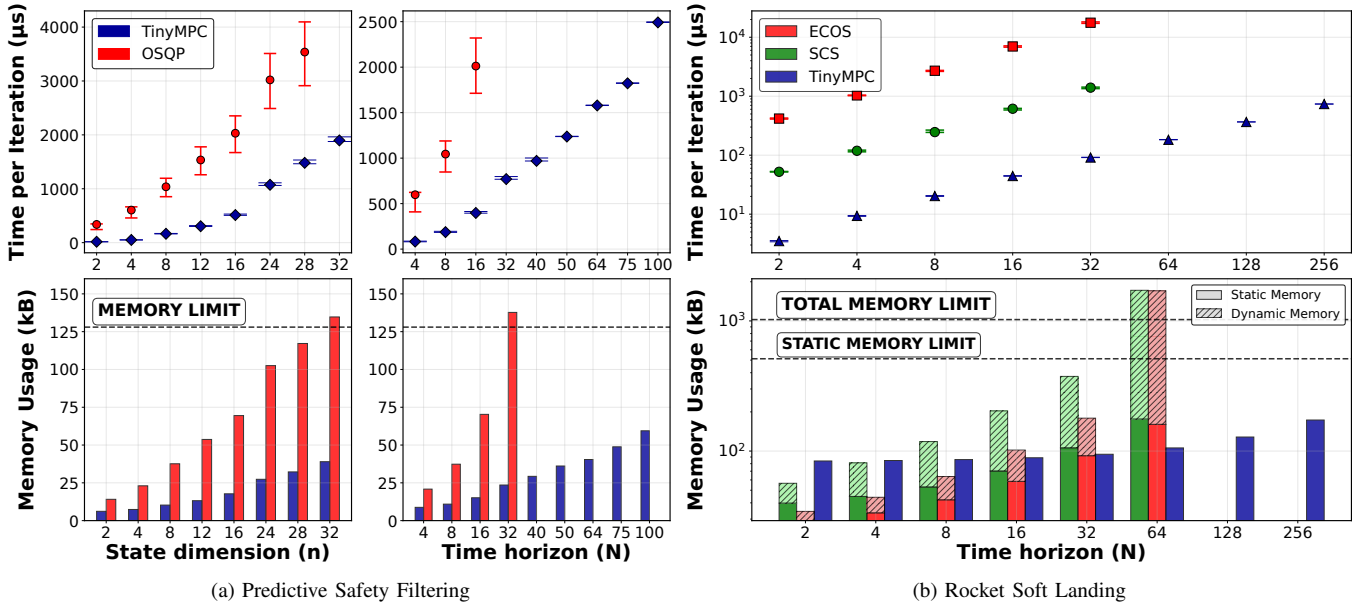


Fig. 3: (a) Predictive safety filtering performance comparison between Conic-TinyMPC and OSQP on an STM32F405 Feather board. Top row shows average iteration times, bottom row shows memory usage. Left column: time horizon kept constant at $N = 10$ while state dimension n ranged from 2 to 32 and input dimension was set to half of the state dimension. Right column: state and control input held constant at $n = 10$ and $m = 5$ while N ranged from 4 to 100. Error bars represent maximum and minimum time taken per iteration for all MPC steps. Black dotted lines denote memory thresholds. (b) Rocket soft-landing performance comparison between Conic-TinyMPC, ECOS, and SCS using a Teensy 4.1 development board. Top plot shows memory usage, bottom plot shows average iteration times. In this SOCP-based experiment $n = 6$ and $m = 3$ while N varied from 2 to 256. Error bars represent maximum and minimum time taken per iteration for all MPC steps performed. Black dotted lines denote memory thresholds.

3) *Early Termination*: High-rate real-time control requires a solver to return a solution within a strict time window. Table II shows the trajectory-tracking performance of each solver on the rocket soft-landing problem with four different control step durations, resulting in four different time budgets. We solve the same problem as in V-A.2, except that each solver must return within the specified time budget. The maximum number of iterations for each solver was determined based on the average time per iteration for each solver with $N = 16$ (Fig. 3b). For example, when given 20ms to solve the problem, the maximum number of solver iterations for ECOS, SCS, and Conic-TinyMPC were 3, 33, and 444, respectively. This represents a factor of 11x to 148x more solver iterations for Conic-TinyMPC. Table II reports two different metrics: A) the total control input violation on box and SOC constraints and B) the landing error (defined as the norm of the deviation between the final and goal states).

ECOS successfully solved to convergence only when given 1000ms, impractical for most real-time control tasks. It failed in subsequent cases due to its limited speed and inability to warm start, with zero iterations completed within 2ms. On the other hand, even though SCS and Conic-TinyMPC were both unable to solve the problem to full convergence at every iteration for shorter time budgets, Conic-TinyMPC was able to utilize its increased number of iterations and warm starting to maintain low constraint violation and landing error. This resulted in Conic-TinyMPC outperforming SCS for all scenarios with a 1.6x to 2.4x reduction in landing error and, most critically, while SCS violated constraints across all time budgets, Conic-TinyMPC only appreciably did so for the shortest 2ms time budget.

TABLE II: Solver performance comparison with different solution-time budgets. Within 20ms ($N = 16$), the maximum number of solver iterations for ECOS, SCS, and TinyMPC are 3, 33, and 444, respectively. ECOS was not able to complete a single optimization iteration within 2ms.

A. Constraint Violation				
Time Budget (ms)	1000	20	10	2
ECOS	0.00	132.63	1757.00	–
SCS	2.04	5.48	10.59	22.84
TinyMPC	0.01	0.01	0.01	4.43
B. Landing Error				
ECOS	1.33	629.02	939.26	–
SCS	1.35	1.36	1.37	2.11
TinyMPC	0.87	0.87	0.87	0.87

B. Robot Hardware Experiments

Next, we demonstrate the efficacy of our solver for real-time execution of dynamic control tasks on a Crazyflie 2.1, a 27 gram quadrotor with an ARM Cortex-M4 (STM32F405) clocked at 168 MHz with 192 kB of SRAM and 1 MB of flash. We present three experiments detailing the high performance of Conic-TinyMPC for dynamic control tasks requiring the online solution to QPs and SOCPs: 1) predictive safety filtering to enable safe control of fundamentally unsafe policies, 2) attitude/thrust vector regulation with thrust-cone constraints, and 3) tracking a spiral landing trajectory with conic constraints and a constraint-violating helical reference.

We note that for the problem sizes required for these experiments, OSQP, SCS, and ECOS all could not fit within the memory available on this MCU and, as such, cannot be used as baselines. Instead, we compare against the Bresciani [47] and Mellinger [48] reactive controllers included

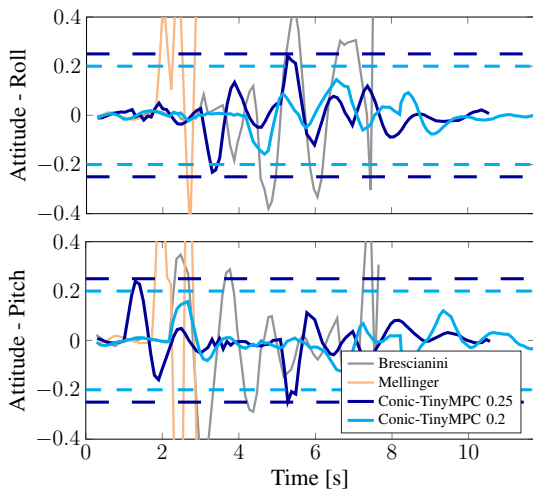


Fig. 4: Attitude/thrust vector regulating performance of different controllers on the Crazyflie. While Conic-TinyMPC was able to constrain the aircraft attitude within the bounds (dashed lines for 0.25 and 0.2 radians), Brescianini and Mellinger exhibited large attitude deviations, causing failures.

with the Crazyflie firmware. These controllers often clip the control input to meet hardware constraints. For all experiments, we ran all controllers with their default parameters and attached an optical flow deck to the Crazyflie to perform state estimation fully onboard the robot.

In all experiments, we linearized the quadrotor’s 6-DOF dynamics about a hover, representing the quadrotor as a point mass with a thrust vector input, and representing its attitude with a quaternion using the formulation in [49]. This problem has state dimension $n = 12$ and $m = 4$, representing the quadrotor’s full state and PWM motor commands. It is worth noting that the Crazyflie platform offers a great chance to test the controller’s robustness due to its high model uncertainty and rapidly depleting battery power (only 5-15 minutes of flight). Under the restricted budget, our Conic-TinyMPC ran at 50 Hz with at most 20 ADMM iterations per call, using a fast reactive controller to track the predicted next state.

1) *Predictive Safety Filtering*: We use a nominal PD controller and formulate a predictive safety filtering problem as a QP, similar to V-A. The Crazyflie was commanded to follow a sinusoidal path along a single axis with an amplitude of 1.2 m (Fig. 1 bottom), which was then tracked with both a nominal PD controller (red) and by Conic-TinyMPC (blue) using a horizon of 20 knot points and box constraints at ± 0.6 m. The box constraints represent safety limits on the quadrotor’s operating space. Conic-TinyMPC is able to successfully respect the safety limits, handling them by slowing to a stop and hovering at the boundaries of the constraints until the reference trajectory comes back around and sends the Crazyflie to the other side of the boundary. This experiment demonstrates Conic-TinyMPC’s ability to act as a safety layer for unsafe policies.

2) *Attitude and Thrust-Vector Regulation*: In many controllers for vertical take-off and landing (VTOL) aircraft, the thrust vector is constrained to lie within a cone [50]. We formulated an SOCP-based MPC problem for the Crazyflie that incorporates such a constraint, implicitly constraining

the drone’s attitude. We used the Brescianini, Mellinger, and Conic-TinyMPC controllers to track an aggressive maneuver (drawing a circle in the air very quickly) to determine if the cone constraint was limiting the Crazyflie’s attitude. As depicted in Fig. 4, Conic-TinyMPC was able to successfully limit the Crazyflie’s attitude to two different maximum values (0.2 and 0.25 radians). Conversely, the baselines exhibited significant attitude deviations, resulting in failures. It is important to note that one can only reduce the attitude deviations of these myopic baselines through careful gain tuning, without any guarantees, while Conic-TinyMPC allows them to be specified explicitly as constraints.²

3) *Conically Constrained Spiral Landing*: Planetary landing problems typically include a glideslope constraint to ensure sufficient elevation during approach and to prevent the spacecraft from crashing into terrain [50]. Fig. 1 top demonstrates the ability of Conic-TinyMPC to handle the planetary landing glideslope constraint of a spacecraft. The reference trajectory is a descending cylindrical spiral (red) which we tracked with Conic-TinyMPC and no position constraints. We then added a conic constraint to restrict the Crazyflie’s position to within a 45° cone originating from the center of the cylindrical reference trajectory. Conic-TinyMPC restricts the Crazyflie from leaving the glideslope constraint, resulting in a spiral landing maneuver (blue).

VI. CONCLUSIONS AND FUTURE WORK

In this paper, we develop Conic-TinyMPC, an open-source, high-speed, structure-exploiting, alternating direction method of multipliers (ADMM) solver targeting low-power embedded conic control applications. We also present a code-generation framework with high level Python, MATLAB, and Julia interfaces that makes it easy to use our solver. We demonstrate the performance of Conic-TinyMPC through a series of experiments including a number of microcontroller benchmarks, and hardware deployments using a 27 gram Crazyflie quadrotor [42].

There are several directions for future work. One of particular note is that our approach, like that of [5], relies on fixed (set of) linearizations, which may not capture all robotic systems well. To address this, we plan to explore recent work [51] that models the nonlinear-to-linear gap as an antagonistic disturbance using reachability analysis, enabling us to more safely support nonlinear systems.

REFERENCES

- [1] P. M. Wensing, M. Posa, Y. Hu, A. Escande, N. Mansard, and A. D. Prete, “Optimization-based control for dynamic legged robots,” *IEEE Transactions on Robotics*, 2024.
- [2] A. Aydinoglu, A. Wei, W.-C. Huang, and M. Posa, “Consensus complementarity control for multicontact mpc,” *IEEE Transactions on Robotics*, vol. 40, pp. 3879–3896, 2024.

²We note that thrust-cone constraints are particularly valuable for Conic-TinyMPC on quadrotors, as the solver relies on linearized dynamics with small-angle approximations, which are only valid within a fixed region of the state space. As such, enforcing a thrust-cone constraint helps ensure that the system remains within this valid operating region, which is essential for maintaining stability during control tasks.

- [3] S. Le Cleac'h, T. A. Howell, S. Yang, C.-Y. Lee, J. Zhang, A. Bishop, M. Schwager, and Z. Manchester, "Fast contact-implicit model predictive control," *IEEE Transactions on Robotics*, 2024.
- [4] S. Gurumurthy, K. Nguyen, A. L. Bishop, J. Z. Kolter, and Z. Manchester, "Deq-mpc: Deep equilibrium model predictive control," in *9th Annual Conference on Robot Learning*, 2025.
- [5] K. Nguyen, S. Schoedel, A. Alavilli, B. Plancher, and Z. Manchester, "Tinympc: Model-predictive control on resource-constrained microcontrollers," in *IEEE International Conference on Robotics and Automation (ICRA)*, Yokohama, Japan, May. 2024.
- [6] S. M. Neuman, B. Plancher, B. P. Duisterhof, S. Krishnan, C. Banbury, M. Mazumder, S. Prakash, J. Jabbour, A. Faust, G. C. de Croon, and V. Janapa Reddi, "Tiny robot learning: challenges and directions for machine learning in resource-constrained robots," in *IEEE International Conference on Artificial Intelligence Circuits and Systems (AICAS)*, 2022.
- [7] Z. Zhang, A. A. Suleiman, L. Carlone, V. Sze, and S. Karaman, "Visual-inertial odometry on chip: An algorithm-and-hardware co-design approach," 2017.
- [8] N. O. Lambert, D. S. Drew, J. Yaconelli, S. Levine, R. Calandra, and K. S. Pister, "Low-level control of a quadrotor with deep model-based reinforcement learning," *IEEE Robotics and Automation Letters*, 2019.
- [9] C. E. Luis, M. Vukosavljev, and A. P. Schoellig, "Online trajectory generation with distributed model predictive control for multi-robot motion planning," *IEEE Robotics and Automation Letters*, 2020.
- [10] G. Torrente, E. Kaufmann, P. Föhn, and D. Scaramuzza, "Data-driven mpc for quadrotors," *IEEE Robotics and Automation Letters*, 2021.
- [11] L. Xi, X. Wang, L. Jiao, S. Lai, Z. Peng, and B. M. Chen, "Gto-mpc-based target chasing using a quadrotor in cluttered environments," *IEEE Transactions on Industrial Electronics*, vol. 69, no. 6, pp. 6026–6035, 2021.
- [12] V. Adajania, S. Zhou, S. Arun, and A. Schoellig, "Amswarm: An alternating minimization approach for safe motion planning of quadrotor swarms in cluttered environments," in *IEEE International Conference on Robotics and Automation (ICRA)*, 2023.
- [13] T. Marcucci and R. Tedrake, "Warm start of mixed-integer programs for model predictive control of hybrid systems," *IEEE Transactions on Automatic Control*, vol. 66, no. 6, pp. 2433–2448, 2020.
- [14] E. Adabag, M. Atal, W. Gerard, and B. Plancher, "Mpcgpu: Real-time nonlinear model predictive control through preconditioned conjugate gradient on the gpu," in *IEEE International Conference on Robotics and Automation (ICRA)*, Yokohama, Japan, May. 2024.
- [15] B. E. Jackson, T. Punnoose, D. Neamati, K. Tracy, R. Jitsho, and Z. Manchester, "Altro-c: A fast solver for conic model-predictive control," in *IEEE International Conference on Robotics and Automation (ICRA)*, Xi'an, China, May 2021.
- [16] A. L. Bishop, J. Z. Zhang, S. Gurumurthy, K. Tracy, and Z. Manchester, "Relu-qp: A gpu-accelerated quadratic programming solver for model-predictive control," in *IEEE International Conference on Robotics and Automation (ICRA)*, Yokohama, Japan, May. 2024.
- [17] Y. E. Nesterov and M. J. Todd, *Self-scaled cones and interior-point methods in nonlinear programming*. Université Catholique de Louvain. Center for Operations Research and ..., 1994.
- [18] S. Boyd and L. Vandenberghe, *Convex Optimization*. Cambridge University Press.
- [19] F. Alizadeh and D. Goldfarb, "Second-order cone programming," *Mathematical programming*, vol. 95, no. 1, pp. 3–51, 2003.
- [20] A. Ben-Tal, A. Nemirovski, and L. El Ghaoui, "Robust optimization," 2009.
- [21] M. S. Lobo, L. Vandenberghe, S. Boyd, and H. Lebret, "Applications of second-order cone programming," *Linear algebra and its applications*, vol. 284, no. 1-3, pp. 193–228, 1998.
- [22] X. Liu, Z. Shen, and P. Lu, "Entry trajectory optimization by second-order cone programming," *Journal of Guidance, Control, and Dynamics*, vol. 39, no. 2, pp. 227–241, 2016.
- [23] C. A. Klein and S. Kittivatcharapong, "Optimal force distribution for the legs of a walking machine with friction cone constraints," *IEEE Transactions on Robotics and Automation*, 1990.
- [24] P. Goulart and Y. Chen. (2022) Clarabel. [Online]. Available: <https://oxfordcontrol.github.io/ClarabelDocs/stable/>
- [25] M. Garstka, M. Cannon, and P. Goulart, "Cosmo: A conic operator splitting method for large convex problems," in *IEEE European Control Conference (ECC)*, 2019.
- [26] A. Domahidi, E. Chu, and S. Boyd, "ECOS: An SOCP solver for embedded systems," in *IEEE European Control Conference (ECC)*, 2013.
- [27] M. ApS, *Introducing the MOSEK Optimization Suite 10.1.28*, 2024. [Online]. Available: <https://docs.mosek.com/latest/intro/index.html>
- [28] B. Stellato, G. Banjac, P. Goulart, A. Bemporad, and S. Boyd, "Osqp: An operator splitting solver for quadratic programs," *Mathematical Programming Computation*, vol. 12, no. 4, pp. 637–672, 2020.
- [29] B. O'Donoghue, E. Chu, N. Parikh, and S. Boyd, "Conic optimization via operator splitting and homogeneous self-dual embedding," *Journal of Optimization Theory and Applications*, vol. 169, no. 3, pp. 1042–1068, June 2016.
- [30] E. AG, "Forcespro," 2014–2023. [Online]. Available: <https://forces.embotech.com/>
- [31] R. Verschuere, G. Frison, D. Kouzoupis, J. Frey, N. van Duijkeren, A. Zanelli, B. Novoselnik, T. Albin, R. Quirynen, and M. Diehl, "acados – a modular open-source framework for fast embedded optimal control," *Mathematical Programming Computation*, 2021.
- [32] G. Frison and M. Diehl, "Hpipm: a high-performance quadratic programming framework for model predictive control," *IFAC-PapersOnLine*, vol. 53, no. 2, pp. 6563–6569, 2020.
- [33] J. L. Jerez, P. J. Goulart, S. Richter, G. A. Constantinides, E. C. Kerrigan, and M. Morari, "Embedded online optimization for model predictive control at megahertz rates," *IEEE Transactions on Automatic Control*, vol. 59, no. 12, pp. 3238–3251, 2014.
- [34] B. O'Donoghue, G. Stathopoulos, and S. Boyd, "A splitting method for optimal control," *IEEE Transactions on Control Systems Technology*, vol. 21, pp. 2432–2442, 11 2013.
- [35] J. Mattingley and S. Boyd, "CVXGEN: A code generator for embedded convex optimization," in *Optimization Engineering*, pp. 1–27.
- [36] F. L. Lewis, D. Vrabie, and V. Szymos, "Optimal Control," 1 2012.
- [37] B. Açıkmeşe, J. M. Carson, and L. Blackmore, "Lossless convexification of nonconvex control bound and pointing constraints of the soft landing optimal control problem," *IEEE Transactions on Control Systems Technology*, vol. 21, no. 6, pp. 2104–2113, 2013.
- [38] J. Di Carlo, P. M. Wensing, B. Katz, G. Bleedt, and S. Kim, "Dynamic locomotion in the mit cheetah 3 through convex model-predictive control," in *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2018, pp. 1–9.
- [39] M. Babu, Y. Oza, A. K. Singh, K. M. Krishna, and S. Medasani, "Model predictive control for autonomous driving based on time scaled collision cone," in *IEEE European Control Conference (ECC)*, 2018.
- [40] S. Boyd, N. Parikh, E. Chu, B. Peleato, J. Eckstein, et al., "Distributed optimization and statistical learning via the alternating direction method of multipliers," *Foundations and Trends® in Machine Learning*, vol. 3, no. 1, pp. 1–122, 2011.
- [41] I. Mahajan and B. Plancher, "Robust and efficient embedded convex optimization through first-order adaptive caching," in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2025.
- [42] Bitcraze, "Crazyflie 2.1," 2023. [Online]. Available: <https://www.bitcraze.io/products/crazyflie-2-1/>
- [43] K.-C. Hsu, H. Hu, and J. F. Fisac, "The safety filter: A unified view of safety-critical control in autonomous systems," *Annual Review of Control, Robotics, and Autonomous Systems*, vol. 7, 2023.
- [44] F. P. Bejarano, L. Brunke, and A. P. Schoellig, "Multi-step model predictive safety filters: Reducing chattering by increasing the prediction horizon," in *IEEE Conference on Decision and Control (CDC)*, 2023.
- [45] M. Schaller, G. Banjac, S. Diamond, A. Agrawal, B. Stellato, and S. Boyd, "Embedded code generation with cvxpy," *IEEE Control Systems Letters*, vol. 6, pp. 2653–2658, 2022.
- [46] "Teensy@ 4.1." [Online]. Available: <https://www.pjrc.com/store/teensy41.html>
- [47] D. Brescianini, M. Hehn, and R. D'Andrea, "Nonlinear quadcopter attitude control," 2013.
- [48] D. Mellinger and V. Kumar, "Minimum snap trajectory generation and control for quadrotors," in *IEEE International Conference on Robotics and Automation (ICRA)*, 2011.
- [49] B. E. Jackson, K. Tracy, and Z. Manchester, "Planning with attitude," *IEEE Robotics and Automation Letters*, 2021.
- [50] D. Maljuta, T. P. Reynolds, M. Szmuk, T. Lew, R. Bonalli, M. Pavone, and B. Açıkmeşe, "Convex optimization for trajectory generation," *IEEE Control Systems Magazine*, vol. 42, no. 5, pp. 40–113, 2022.
- [51] W. Sharpless, Y. T. Chow, and S. Herbert, "State-augmented linear games with antagonistic error for high-dimensional, nonlinear hamilton-jacobi reachability," in *IEEE Conference on Decision and Control (CDC)*, 2024.