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

Sam Schoedel*, Khai Nguyen*, Elakhya Nedumaran, Brian Plancher, Zachary Manchester

Abstract— **Conic constraints appear in many important control applications like legged locomotion, robotic manipulation, and autonomous rocket landing. However, current solvers for conic optimization problems have relatively heavy computational demands in terms of both floating-point operations and memory footprint, making them impractical for use on small embedded devices. We extend TinyMPC, an open-source, high-speed solver targeting low-power embedded control applications, to handle second-order cone constraints. We also present code-generation software to enable deployment of TinyMPC on a variety of microcontrollers. We benchmark our generated code against state-of-the-art embedded QP and SOCP solvers, demonstrating a two-order-of-magnitude speed increase over ECOS while consuming less memory. Finally, we demonstrate TinyMPC's efficacy on the Crazyflie, a lightweight, resource-constrained quadrotor with fast dynamics.**

TinyMPC and its code-generation tools are publicly available at `https://tinympc.org`.

I. Introduction

Model-predictive control (MPC) is a powerful tool for controlling highly dynamic systems subject to complex constraints [1], [2], [3]. Second-order cones represent an important class of constraints that appear in many robotics and aerospace control problems when reasoning about friction, attitude, and thrust limits [4], [5], [6]. However, solving the resulting second-order cone programs (SOCPs) at real-time rates can be computationally challenging, especially on the low-power, resource-constrained microcontrollers found on embedded systems.

For deployment on microcontrollers, which often lack full hardware support for floating-point arithmetic, an ideal MPC solver should be division-free, only use static memory allocation, and support warm starting to take advantage of computation at previous time steps [7], [8], [9]. Compiled code should also have a low memory footprint and be easily verifiable through an interface to a high-level language like Python, MATLAB, or Julia. Table I compares existing solvers that are commonly used in control settings. We focus on SOCP solvers but also include two popular quadratic programming (QP) solvers. Because most 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 and thus use too much memory to deploy on most microcontrollers; are not written in a language that allows the solver to be easily ported to an embedded system; or some combination of these issues.

Furthermore, while several of these solvers, including ECOS [12] and SCS [15], support code generation for embedded devices, they still either use too much memory to fit on resource-constrained microcontrollers or are too computationally inefficient to solve MPC problems in real time. Specifically, ECOS's use of an interior-point method does not allow efficient warm starting, which is essential for good performance in MPC applications. Additionally, both ECOS and SCS return a certificate of infeasibility. While this is generally a useful feature, it requires extra computation that is unnecessary in an MPC setting, where an inability to find a solution will result in system failure regardless of whether the solver can detect infeasibility.

Prior work introduced TinyMPC [17], a QP solver for model-predictive control based on the alternating direction method of multipliers (ADMM) [18]. TinyMPC avoids divisions while reducing computational complexity and memory

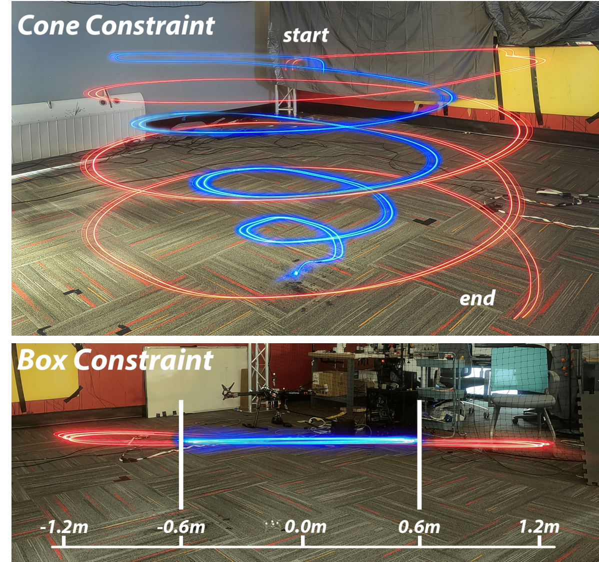


Fig. 1: TinyMPC allows real-time execution of convex model-predictive control on resource-constrained microcontrollers. We demonstrate TinyMPC's constraint handling capabilities on a 27 gram nano quadrotor, the Crazyflie. Top, we track a descending helical reference (red) with its position subject to a 45° second-order cone. 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).

∗These authors contributed equally.

Samuel Schoedel and Zachary Manchester are with the Robotics Institute, Carnegie Mellon University, 5000 Forbes Ave., Pittsburgh, PA 15213. {sschoede, zmanches}@andrew.cmu.edu

Khai Nguyen is with the Department of Mechanical Engineering, Carnegie Mellon University, 5000 Forbes Ave., Pittsburgh, PA 15213. xuankhan@andrew.cmu.edu

Elakhya Nedumaran is with the Department of Electrical and Computer Engineering, Carnegie Mellon University, 5000 Forbes Ave., Pittsburgh, PA 15213. enedumar@andrew.cmu.edu

Brian Plancher is with Barnard College, Columbia University, 3009 Broadway New York, NY 10027. bplancher@barnard.edu

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

Solver	SOC	Warm Starting	Embedded	Open Source	Quad. Obj.	MPC Tailored
Clarabel [10]	✓	✗	✗	✓	✓	✗
COSMO [11]	✓	✓	✗	✓	✓	✗
ECOS [12]	✓	✗	✓	✓	✗	✗
MOSEK [13]	✓	✗	✗	✗	✓	✗
OSQP [14]	✗	✓	✓	✓	✓	✗
SCS [15]	✓	✓	✓	✓	✓	✗
FORCES [16]	✗	✓	✓	✗	✓	✓
ALTRO-C [9]	✓	✓	✗	✓	✓	✓
TinyMPC (ours)	✓	✓	✓	✓	✓	✓

footprint by pre-computing and caching expensive matrix factorizations and only performing matrix-vector products online, achieving state-of-the-art performance.

In this work, we extend the TinyMPC solver [17] to support second-order cone constraints on states and inputs, enabling real-time second-order cone model-predictive control on resource constrained platforms. Importantly, we also develop an open-source code generation software package with interfaces to Python, MATLAB, and Julia to ease the deployment of TinyMPC on a wide range of microcontrollers. To the best of the authors' knowledge, TinyMPC is the first MPC solver intended for execution on resource-constrained microcontrollers that handles second-order cone constraints. Our specific contributions include:

- An open source implementation of TinyMPC with support for second-order cone constraints, making it the fastest and lowest-memory-footprint embedded SOCP solver.
- User-friendly interfaces for easy code generation and solution verification using TinyMPC with examples in Python, MATLAB, and Julia.
- Experimental validation on hardware for several benchmark control problems.

The remainder of this paper is organized as follows: Section II provides mathematical preliminaries on the linear-quadratic regulator and the alternating-direction method of multipliers. Section III introduces TinyMPC and its extension to handle conic constraints. Section IV gives an overview of the code-generation software and demonstrates the minimum code required to set up a problem with TinyMPC. Section V details benchmarks on ARM Cortex M4 and M7 microcontrollers, as well as hardware experiments on a Crazyflie quadrotor. Finally, Section VI summarizes our conclusions.

II. Background

A. The Linear-Quadratic Regulator

The linear-quadratic regulator (LQR) problem [19] minimizes a quadratic cost subject to linear (or affine) dynamics constraints:

$$\begin{aligned}\min_{x_{1:N}, u_{1:N-1}} \quad & J = \frac{1}{2} x_N^\intercal Q_N x_N + q_N^\intercal x_N + \\ & \sum_{k=1}^{N-1} \frac{1}{2} x_k^\intercal Q x_k + q_k^\intercal x_k + \frac{1}{2} u_k^\intercal R u_k + r_k^\intercal u_k \\ \text{subject to} \quad & x_{k+1} = A x_k + B u_k + c, \quad \forall k \in [1, N),\end{aligned} \tag{1}$$

where $x_k \in \mathbb{R}^n$, $u_k \in \mathbb{R}^m$ are the state and control input at time step k, N is the number of time steps (also referred to as the horizon), $A \in \mathbb{R}^{n \times n}$, $B \in \mathbb{R}^{n \times m}$, and $c \in \mathbb{R}^n$ define the system dynamics, $Q \succeq 0$, $R \succ 0$, and $Q_N \succeq 0$ are symmetric cost matrices and q and r are linear cost vectors.

Equation (1) has a closed-form solution in the form of an affine feedback controller [19]:

$$u_k^* = -K_k x_k - d_k. \tag{2}$$

The feedback and feedforward terms (K_k, d_k) are found by solving the discrete 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 [19]:

$$\begin{aligned}K_k &= (R + B^\intercal P_{k+1} B)^{-1} (B^\intercal P_{k+1} A), \\ d_k &= (R + B^\intercal P_{k+1} B)^{-1} (B^\intercal p_{k+1} + r_k + B^\intercal P_{k+1} c), \\ P_k &= Q + K_k^\intercal R K_k + (A - B K_k)^\intercal P_{k+1} (A - B K_k), \\ p_k &= q_k + (A - B K_k)^\intercal (p_{k+1} - P_{k+1} B d_k + P_{k+1} c) + \\ & \quad K_k^\intercal (R d_k - r_k).\end{aligned} \tag{3}$$

B. Convex Model-Predictive Control

Convex MPC extends the LQR formulation to admit additional convex constraints on the system states and control inputs:

$$\begin{aligned}\min_{x_{1:N}, u_{1:N-1}} \quad & J(x_{1:N}, u_{1:N-1}) \\ \text{subject to} \quad & x_{k+1} = A x_k + B u_k + c, \\ & x_k \in \mathcal{X}, u_k \in \mathcal{U},\end{aligned} \tag{4}$$

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 appli-

cations including autonomous rocket landings [20], legged locomotion [21], and autonomous driving [22].

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

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

C. The Alternating Direction Method of Multipliers

The alternating direction method of multipliers (ADMM) is a popular and efficient approach for solving convex optimization problems, including SOCPs like (5). We provide a very brief summary here and refer readers to [18] for more details.

Given a generic problem (with f and $\mathcal{C}$ convex):

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

we can define the indicator function for the set $\mathcal{C}$ as

$$I_{\mathcal{C}}(z) = \begin{cases} 0 & z \in \mathcal{C} \\ \infty & \text{otherwise}, \end{cases} \tag{7}$$

and then form the equivalent problem (introducing the slack variable z)

$$\begin{aligned} \min_{x} \quad & f(x) + I_{\mathcal{C}}(z) \\ \text{subject to} \quad & x = z. \end{aligned} \tag{8}$$

The augmented Lagrangian of the transformed problem (8) is (with Lagrange multiplier λ and scalar penalty weight ρ):

$$\mathcal{L}_A(x, z, \lambda) = f(x) + I_{\mathcal{C}}(z) + \lambda^\intercal (x - z) + \frac{\rho}{2} ||x - z||_2^2. \tag{9}$$

Thus, if we alternate minimization over x and z, we arrive at the three-step ADMM iteration,

$$\text{primal update} : x^+ = \arg\min_{x} \mathcal{L}_A(x, z, \lambda), \tag{10}$$

$$\text{slack update} : z^+ = \arg\min_{z} \mathcal{L}_A(x^+, z, \lambda), \tag{11}$$

$$\text{dual update} : \lambda^+ = \lambda + \rho(x^+ - z^+), \tag{12}$$

where the last step is a gradient-ascent update on the Lagrange multiplier [18]. 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, and the slack update is a linear or conic projection. ADMM-based QP and SOCP solvers such as OSQP [14] and SCS [15] have demonstrated state-of-the-art results.

III. The TinyMPC Solver

TinyMPC exploits properties of the MPC problem to efficiently solve the primal update in (10), prioritizing high speed and low memory footprint over generality. Specifically, we cache the solution to an infinite-horizon LQR problem to reduce memory and pre-compute expensive matrix inverses in (3) to reduce online computation.

A. Combining LQR and ADMM for MPC

We solve the following problem, introducing slack variables and indicator functions for state and input constraints as in (9), where z, w, λ, and μ are the state slack, input slack, state dual, and input dual variables over the entire horizon:

$$\begin{aligned} \min_{\substack{x_{1:N}, z_{1:N}, \\ \lambda_{1:N}, u_{1:N-1}, \\ w_{1:N-1}, \mu_{1:N-1}}} \quad & \mathcal{L}_A(\cdot) = J(x_{1:N}, u_{1:N-1}) + \\ & \qquad I_{\mathcal{X}}(z_{1:N}) + I_{\mathcal{U}}(w_{1:N-1}) + \\ & \sum_{k=1}^{N} \frac{\rho}{2} (x_k - z_k)^\intercal (x_k - z_k) + \lambda_k^\intercal (x_k - z_k) + \\ & \sum_{k=1}^{N-1} \frac{\rho}{2} (u_k - w_k)^\intercal (u_k - w_k) + \mu_k^\intercal (u_k - w_k) \\ \text{subject to:} \quad & x_{k+1} = A x_k + B u_k + c, \quad \forall k \in [1, N). \end{aligned} \tag{13}$$

State and input constraints are enforced through the indicator functions $I_{\mathcal{X}}$ and $I_{\mathcal{U}}$. We use the ADMM algorithm (10), (11), (12) to solve this optimal control problem. The primal update for (13) becomes an equality-constrained QP:

$$\begin{aligned} \min_{x_{1:N}, u_{1:N-1}} \quad & \frac{1}{2} x_N^\intercal \tilde{Q}_N x_N + \tilde{q}_N^\intercal x_N + \\ & \sum_{k=1}^{N-1} \frac{1}{2} x_k^\intercal \tilde{Q} x_k + \tilde{q}_k^\intercal x_k + \frac{1}{2} u_k^\intercal \tilde{R} x_k + \tilde{r}^\intercal u_k \\ \text{subject to} \quad & x_{k+1} = A x_k + B u_k + c, \end{aligned} \tag{14}$$

where

$$\begin{aligned} \tilde{Q}_N &= Q_N + \rho I, & \tilde{q}_N &= q_N + \lambda_N - \rho z_N, \\ \tilde{Q} &= Q + \rho I, & \tilde{q}_k &= q_k + \lambda_k - \rho z_k, \\ \tilde{R} &= R + \rho I, & \tilde{r}_k &= r_k + \mu_k - \rho w_k. \end{aligned} \tag{15}$$

We observe that, because (14) is the same form as (1), it can be solved efficiently with the Riccati recursion in (3). The primal update in (10) then becomes the solution to the Riccati recursion in the backward pass and an affine dynamics roll-out of the resulting LQR policy in the forward pass.

B. Infinite-Horizon LQR

The LQR Riccati solution has properties we can leverage to significantly reduce computation and memory usage. Given a long enough horizon, the Riccati recursion (3) converges to the solution of the infinite-horizon LQR problem [19]. Thus, we only cache a single gain matrix K_{inf} and cost-to-go Hessian P_{inf}, trading the versatility of the time-varying LQR solution to maintain a small memory footprint.

C. Precomputation

We rearrange the linear and feedforward terms in (3) to isolate the following static matrices, which are precomputed

and cached alongside K_{inf} and P_{inf}:

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

The ADMM backward pass can then be rewritten as

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

which only requires matrix-vector products to compute, drastically reducing online computation time. Compared to [17], additional terms are derived to handle the affine dynamics.

D. Second-Order Conic Projection

The slack update in (11) can be written as the operator Π that projects the slack variable onto the feasible space. For linear inequality constraints, the projection is onto a set of upper and lower bounds defined by the element-wise operator

$$\Pi(z) = \max(z_l, \min(z_u, z)), \tag{18}$$

where z corresponds to the state and control input slack variables. We now extend (1) to solve problems that include conic constraints defined by

$$\mathcal{K} = \left\{ z \in \mathbb{R}^n | z_n \geq \sqrt{z_1^2 + z_2^2 + \cdots + z_{n-1}^2} \right\}. \tag{19}$$

The structure of the ADMM algorithm inherently isolates the projection step, allowing us to replace the projection operator in the slack update (18) with the SOC projection

$$\Pi_{\mathcal{K}}(z) = \begin{cases} 0, & \|v\|_2 \leq -a, \\ z, & \|v\|_2 \leq a, \\ \frac{1}{2}(1 + \frac{a}{\|v\|_2}) \begin{bmatrix} v \\ \|v\|_2 \end{bmatrix}, & \|v\|_2 > |a|, \end{cases} \tag{20}$$

where $v = [z_1, \ldots, z_{n-1}]^\intercal$, $a = z_n$. Here, $z_i, i = 1, ..., n$ is any vector subset of the state or control slack variables.

IV. Code Generation with TinyMPC

We have developed a code-generation tool for TinyMPC with interfaces to Python, MATLAB, and Julia that produces dependency-free C++ code. Listing 1 is an example Python script that generates problem-specific TinyMPC code. The `setup` function is used to initialize the problem with specific data, which consists of the time horizon (N), system model (A, B, and c), cost weights (Q and R), linear and conic constraint parameters (`bounds` and `socs`), and solver settings. Users may opt to set primal and dual solution tolerances, the maximum number of iterations per solve, and whether to check termination conditions. Tuning the maximum number of iterations for a particular system is often critical for returning a usable solution within real-time limits. The `codegen` function is then used to generate the tailored code. Nearly identical scripts can be used to generate code from MATLAB or Julia.

The directory structure of the resulting 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 2).

```
import tinympc

# Create a TinyMPC object
tiny = tinympc.TinyMPC()

# Initialize the solver
tiny.setup(N, A, B, c, Q, R, bounds, socs,
    settings)

# Generate code
tiny.codegen(output_dir)
```

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

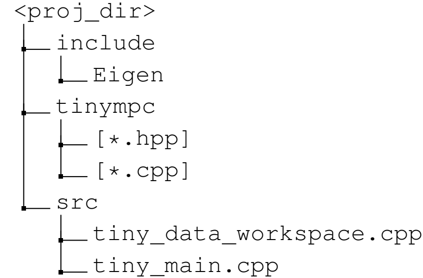


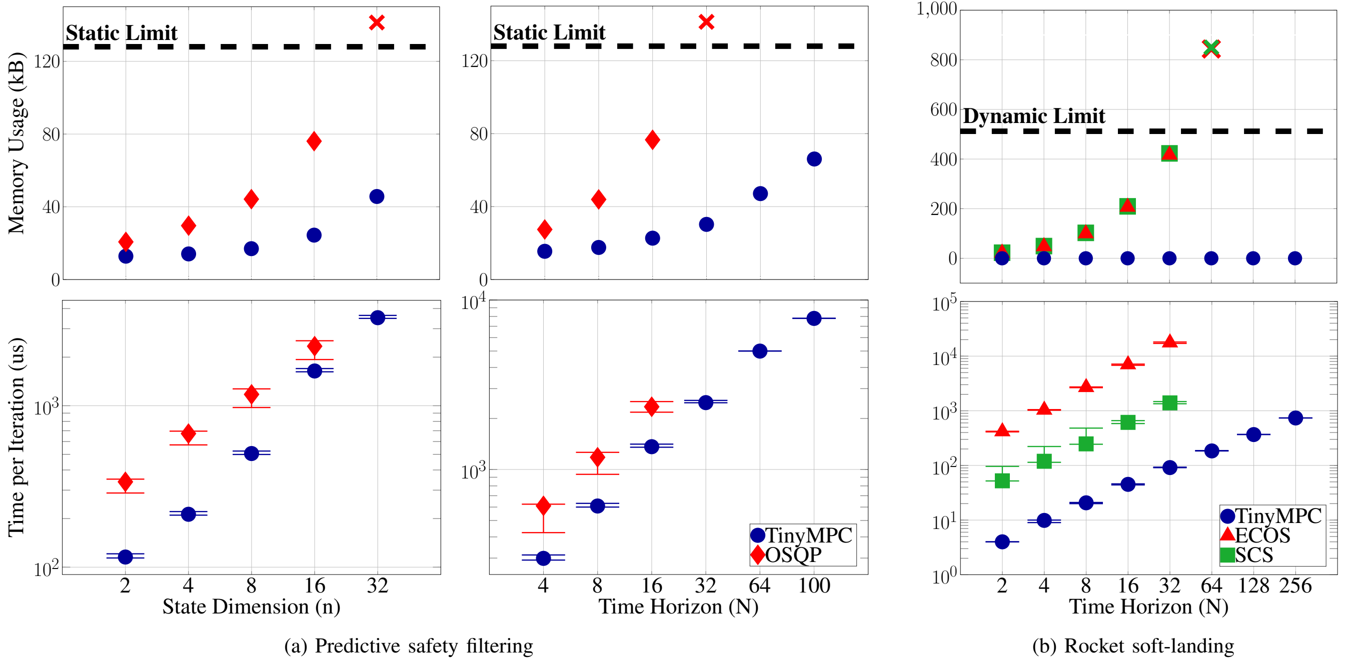
Fig. 2: The tree structure of the generated code. The main program is stored in `tiny_main.cpp`.

```
#include "tinympc.hpp"
#include "tiny_data_workspace.hpp"

int main(int argc, char **argv) {
   tiny_solve(&solver); // Solve the problem
   return 0;
}
```

Listing 2: A simple C++ program that loads the problem data from `tiny_data_workspace.hpp` and solves the problem.

```
import tinympc

# Create a TinyMPC object
tiny = tinympc.TinyMPC()

# Load the library
tiny.load_lib("path/to/shared/library.so")

# Solve the problem
tiny.solve()

# Get the solution
tiny.get_u(u)
```

Listing 3: An example Python script to run the generated code.



(a) Predictive safety filtering

(b) Rocket soft-landing

Fig. 3: Comparison of average iteration times (top) and memory usage (bottom) between different solvers. (a) compares TinyMPC to OSQP on a QP-based predictive safety filtering task and was performed on an STM32F405 Feather board (ARM Cortex-M4 operating at 168 MHz with 32-bit floating point support, 1 MB of Flash memory, and 128 kB of RAM). In the first column, the time horizon was kept constant at $N = 10$ while the state dimension n ranged from 2 to 32 and the input dimension was set to half of the state dimension. In the second column, the state and control input were held constant at $n = 10$ and $m = 5$ while N ranged from 4 to 100. (b) compares TinyMPC to ECOS and SCS on an SOCP-based rocket soft-landing using a Teensy 4.1 development board (ARM Cortex-M7 operating at 600 MHz with 32-bit floating point support, 7.75 MB of flash memory, and 512 kB of tightly coupled RAM). In this experiment $n = 6$ and $m = 3$ while N varied from 2 to 256. The error bars represent the maximum and minimum time taken per iteration for all MPC steps performed for a specific problem. The black dotted lines denote memory thresholds.

Users may choose to compile code for their host system either manually or through the TinyMPC interface for testing. Listing 3 shows an example Python script that loads the generated code library, solves the problem, then retrieves the solution. The reference trajectory and initial state may be set before solving using the `set_xref`, `set_uref`, and `set_x0` functions, and may be done on the microcontroller using the C++ equivalents. Additional wrapped functions exist for overwriting existing constraint parameters.

V. Experiments

We benchmark the performance of TinyMPC's generated code through two sets of experiments: first, we compare TinyMPC against state-of-the-art solvers on two common microcontrollers, demonstrating faster computation and decreased memory usage. Second, we solve various control tasks running onboard a 27 gram Crazyflie nano-quadrotor [23]. Our results show that TinyMPC's fast computation and low memory footprint enables robots to execute dynamic behaviors while respecting both linear and second-order cone constraints.

A. Microcontroller Benchmarks

We compare TinyMPC against state-of-the-art solvers for two problems while varying the number of states and the horizon length. The first is a predictive safety filtering problem with box constraints on the state and input. The second is a rocket soft-landing problem with a second-order cone constraint on the thrust vector. The safety filter QP is benchmarked against OSQP and the rocket soft-landing SOCP is benchmarked against ECOS and SCS. The microcontroller results are reported in Fig. 3.

1) Predictive Safety Filtering: We formulate a QP with box constraints on the state and input variables to act as a predictive safety filter for a nominal task policy [24], [25]. We compare the solution times and memory usage of TinyMPC and OSQP while varying state and horizon dimensions. We utilize TinyMPC's and OSQP's Python code generation interfaces to produce microcontroller firmware for each problem. In a previous study [17], QP-based MPC was evaluated on the Teensy 4.1 microcontroller which features a powerful ARM Cortex-M7 operating at 600 MHz with 7.75 MB of flash memory, and 512 kB of RAM. Here, we benchmark on a much less powerful microcontroller, the 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, to better understand the scalability limits of embedded QP solvers.

Fig. 3a shows the total program size and the average execution times per iteration, in which TinyMPC uses drastically less memory and exhibits a maximum speed-up of 2.5x over OSQP. This reduction in memory usage allows TinyMPC to solve real-time optimal control of complex systems with long time horizons. In particular, TinyMPC

TABLE II: Solver performance of different control step durations. 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.

Control Step (ms)	1000	20	10	2
A. Constraint Violation				
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**

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, TinyMPC demonstrated scalability to larger state dimensions up to 32, whereas OSQP encountered memory limitations beyond $n = 16$.

2) Rocket Soft-Landing: The soft-landing problem requires a rocket to land with small final velocity at a desired position. This is often decomposed into two control loops, where the translation dynamics are handled with MPC and the attitude dynamics are handled by a faster linear feedback controller [20]. In this scenario, we assume an ideal attitude controller and use a point-mass model with a second-order cone constraint on the thrust vector. We benchmark the performance of TinyMPC against ECOS and SCS, state-of-the-art SOCP solvers. C code for ECOS and SCS was generated using CVXPYgen [26]. C++ code for TinyMPC was produced using the Python code generation interface introduced in Section IV. All solver options were set to equivalent values wherever possible. All tolerances were set to 0.01 and code was executed on the Teensy 4.1 microcontroller. The Teensy allowed us to collect more data than on the less capable STM32F405, 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.

Fig. 3b shows the amount of dynamically allocated memory and the average execution times per iteration for varying time horizon. TinyMPC outperforms SCS and ECOS in execution time and memory, achieving an average speed-up of 13x over SCS and 137x over ECOS. TinyMPC performed no dynamic allocation while SCS and ECOS dynamically allocated the workspace at the beginning due to the use of the CVXPYgen interface. This caused SCS and ECOS to exceed the RAM of the Teensy during execution. Without using the CVXPYgen interface, the dynamically allocated workspace must instead be stored statically, still exceeding the memory limit of the Teensy. Overall, TinyMPC was able to solve problems with a horizon of 256, while SCS and ECOS failed at $N = 64$.

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 different control step durations. We solve the same problem as in V-A.2, except that each solver must return within the specified control step duration. 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 the solvers are given 20ms to solve the problem, the maximum number of solver iterations for ECOS, SCS, and TinyMPC were 3, 33, and 444, respectively. Two different metrics are reported: 1) the total control input violation on box and SOC constraints and 2) the landing error (defined as the norm of the deviation between the final and goal states). These metrics were evaluated at four different control step durations for each solver.

ECOS successfully solved to convergence only when given 1000ms for each control step, which is 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 TinyMPC were not able to solve the problem to full convergence at every iteration for shorter control steps, they were able to utilize warm starting to maintain low constraint violation and landing error. TinyMPC outperformed SCS for all control step durations and, critically, only appreciably violated constraints at the shortest duration of 2ms.

B. Hardware Experiments

We deployed our TinyMPC implementation onto a Crazyflie nano-quadrotor [23], which has an ARM Cortex-M4 (STM32F405) clocked at 168 MHz with 192 kB of SRAM and 1 MB of flash. The Crazyflie was subjected to three problems involving second-order cone or box constraints: predictive safety filtering, attitude/thrust vector regulating, and spiral landing. The Crazyflie's state was represented by a point mass with a thrust vector input. We used a cascaded control architecture for each of these tasks [20], running TinyMPC at 50 Hz and using the Crazyflie's built-in Brescinanini controller [27] to track the solution at 1 kHz. All experiments were done using an optical flow deck attached to the Crazyflie for fully onboard state estimation. An Optitrack motion capture system was used only for long exposure photos (Fig. 1).

1) Quadrotor Predictive Safety Filtering: We use a nominal PD controller and formulate a predictive safety filtering problem as a QP that can be efficiently solved using TinyMPC, similar to V-A. The Crazyflie was commanded to execute an unsafe sinusoidal path of amplitude 1.2 m, which was then tracked with the PD controller and filtered by TinyMPC using a horizon of 20 knot points and box constraints at ± 0.6 m. As illustrated in Fig. 1 bottom, the Crazyflie respects the safety limits (blue) despite the nominal PD-controlled trajectory being 1.2 m in magnitude (red). This experiment illustrates TinyMPC's ability to act as a safety layer for unsafe policies.

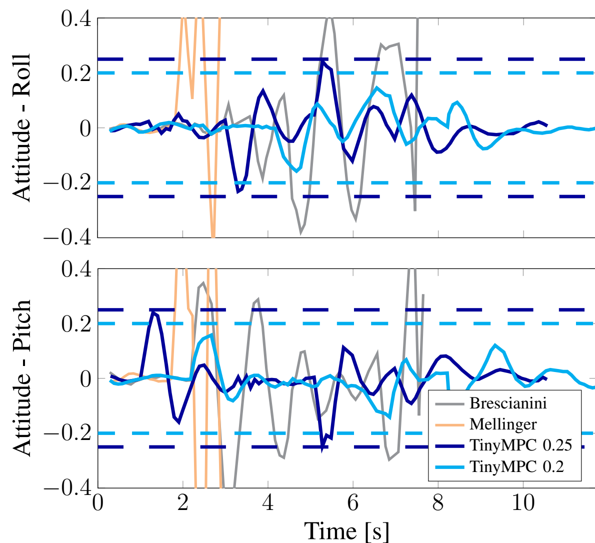


Fig. 4: Attitude/thrust vector regulating performance of different controllers on the Crazyflie. When performing aggressive maneuvers, TinyMPC was able to solve SOCP-based MPC to constrain the aircraft attitude within the bounds (dashed lines, 0.25 and 0.2 radians, respectively). Meanwhile, Brescianini and Mellinger exibited large attitude deviations, causing failures during the task.

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. We formulated an SOCP-based MPC problem for the Crazyflie drone that incorporates such a thrust-cone constraint, which implicitly constrains the drone's attitude. As depicted in Fig. 4, TinyMPC was able to successfully limit the Crazyflie's attitude to two different maximum values of 0.25 radians and 0.2 radians. Conversely, the built-in Brescianini and Mellinger controllers exhibited significant attitude deviations, resulting in failures. It is important to note that one can only reduce the attitude deviations of these reactive controllers through careful gain tuning, while TinyMPC allows them to be specified explicitly as constraints.

3) Conically Constrained Spiral Landing: Fig. 1 (top) illustrates TinyMPC's ability to handle second-order cone constraints on state variables. The reference trajectory is a descending cylindrical spiral (red). We formulated an SOCP-based MPC problem to restrict the Crazyflie's position to within a 45° cone originating from the center of the cylindrical reference trajectory, and solved it with TinyMPC. Our solver forces the Crazyflie to conform to the state-constraint cone, resulting in a spiral landing maneuver (blue).

VI. Conclusions

In this paper, we extend the model-predictive control solver TinyMPC to support second-order cone constraints and present a code-generation package with high-level interfaces in Python, MATLAB, and Julia that make it easy to generate and validate model-predictive control programs. We demonstrate TinyMPC's ability to support second-order cone constraints through a variety of new problems, including rocket soft-landing and attitude and thrust-vector regulation for quadrotors. TinyMPC solves these problems at least an order of magnitude faster than existing state-of-the-art SOCP solvers and with no dynamic memory allocation. These advancements help bridge the gap between computationally intensive convex model-predictive control and resource-constrained processing platforms.

For more information and to get started using TinyMPC, our open source solver, visit `https://tinympc.org`.

References

[1] S. Kuindersma, R. Deits, M. Fallon, A. Valenzuela, H. Dai, F. Permenter, T. Koolen, P. Marion, and R. Tedrake, "Optimization-based locomotion planning, estimation, and control design for the atlas humanoid robot," *Autonomous robots*, vol. 40, pp. 429–455, 2016.

[2] 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.

[3] A. Aydinoglu, A. Wei, and M. Posa, "Consensus complementarity control for multi-contact mpc," *arXiv preprint arXiv:2304.11259*, 2023.

[4] 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.

[5] 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.

[6] 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*, vol. 6, no. 1, pp. 73–85, 1990.

[7] 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.

[8] 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.

[9] B. E. Jackson, T. Punnoose, D. Neamati, K. Tracy, R. Jitosho, and Z. Manchester, "Altro-c: A fast solver for conic model-predictive control," in *2021 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2021, pp. 7357–7364.

[10] P. Goulart and Y. Chen. (2022) Clarabel. [Online]. Available: https://oxfordcontrol.github.io/ClarabelDocs/stable/

[11] M. Garstka, M. Cannon, and P. Goulart, "Cosmo: A conic operator splitting method for large convex problems," in *2019 18th European Control Conference (ECC)*. IEEE, 2019, pp. 1951–1956.

[12] A. Domahidi, E. Chu, and S. Boyd, "ECOS: An SOCP solver for embedded systems," in *European Control Conference (ECC)*, 2013, pp. 3071–3076.

[13] M. ApS, *Introducing the MOSEK Optimization Suite 10.1.28*, 2024. [Online]. Available: https://docs.mosek.com/latest/intro/index.html

[14] 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.

[15] 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. [Online]. Available: http://stanford.edu/~boyd/papers/scs.html

[16] E. AG, "Forcespro," 2014–2023. [Online]. Available: https://forces.embotech.com/

[17] 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.

[18] 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.

[19] F. L. Lewis, D. Vrabie, and V. Syrmos, "Optimal Control," 1 2012. [Online]. Available: https://doi.org/10.1002/9781118122631

[20] 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.

[21] J. Di Carlo, P. M. Wensing, B. Katz, G. Bledt, 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.

[22] 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 *2018 European Control Conference (ECC)*, 2018, pp. 641–648.

[23] Bitcraze, "Crazyflie 2.1," 2023. [Online]. Available: https://www.bitcraze.io/products/crazyflie-2-1/

[24] 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.

[25] F. P. Bejarano, L. Brunke, and A. P. Schoellig, "Multi-step model predictive safety filters: Reducing chattering by increasing the prediction horizon," in *2023 62nd IEEE Conference on Decision and Control (CDC)*. IEEE, 2023, pp. 4723–4730.

[26] 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.

[27] D. Brescianini, M. Hehn, and R. D'Andrea, "Nonlinear quadrocopter attitude control," 2013.