Model Predictive Control on Resource Constrained Microcontrollers

Samuel E. Schoedel

CMU-RI-TR-24-44

August 6st

The Robotics Institute
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA

Submitted in partial fulfillment of the requirements for the degree of Master of Science in Robotics.

To my friends. It's the people, not the place.

Abstract

Model predictive control is a powerful tool for controlling complex systems subject to complex constraints. However, it is computationally expensive and often requires large amounts of memory. Large, slow systems are capable of storing the necessary computational power onboard and thus do not require efficient, specialized algorithms to reduce time and space complexity. On the other hand, smaller systems typically have faster dynamics that require higher frequency controllers. In addition, these smaller systems are generally restricted to carrying smaller computers with less memory, and so they can not fit large problems that allow for intelligent control of the system. Existing methods for tackling model predictive control at small scales are generally inefficient and inaccessible. This thesis aims to tackle both problems by introducing a lightweight conically-constrained convex model predictive control solver, TinyMPC, and its associated software packages which facilitate easy adoption through the use of high-level interfaces and code generation. We benchmark TinyMPC against state-of-the-art quadratic and conic programming solvers on different microcontrollers, showing almost an order of magnitude speedup and memory reduction. Finally, we demonstrate TinyMPC's real world efficacy by deploying it on the Crazyflie 2.1, a nano-quadrotor with fast dynamics, and performing tasks with various convex constraints.

Acknowledgments

Thank you to Oma and Opa, to whom I owe my education.

I would be nothing without my parents, Tom and Kendall. Thank you for the long phone calls and long drives to visit. I can not ever repay you for the advice, support, and memories you have given me through my master's degree and through life. I love you both.

I was lucky to find Zac Manchester, and I am grateful for his willingness to spend one on one time explaining concepts, ability to share his passion, and desire to see his students excel. Zac honed my desire to build interesting robots into a specific skill set, one that I had been searching for during the years prior to joining CMU. I was equally lucky to then meet Brian Plancher, to whom I want to thank for his generosity with time, knowledge, mentorship, and friendship. I would not be the researcher I am today without him.

Life in Pittsburgh was made meaningful by the friendship of the incredible people I met both in and out of the lab. There are too many people to thank and experiences to describe here, but I can at least name a few. I want to thank Conner Pulling for being the first person I knew in Pittsburgh and who is a genuinely good roboticist, philosopher, travel buddy, and friend. Thank you for making me laugh. This thesis would look very different without Khai and Anoushka, to whom I am grateful for staying up with me through late nights in the lab and rides home listening to Price Tag and passing morning bikers. I have nothing but respect for you both. Finally, I want to thank Sofia Kwok for her unconditional support and encouragement, for lifting me out of ruts, and for all of the adventures, thoughts, and laughter we shared. I am a better person because of you. To those who joined me in listening to live music and loud music, trying new food, exploring new cities, paddling kayaks and paddle boarding, boating, eating crabs, watching movies, baking, throwing discs, scaling rock walls, hitting tennis balls, watching astronomical phenomena, skiing, playing chess, camping, talking, and building robots, thank you.

Contents

When this document is viewed as a PDF, the page header is a link to this Table of Contents.

List of Figures

List of Tables

Chapter 1

Introduction

Since its conception, model predictive control (MPC) has been used to control a wide variety of systems subject to complex constraints. MPC was first used in the 1970s to control chemical processes, which generally had slow dynamics and thus didn't require frequent solutions to the MPC problem [63]. As faster computers were developed, the technique was adopted by biomedical scientists [66], HVAC engineers [61], financial advisors [67], power system engineers [7], and roboticists [9, 36], among many others.

The concept is straightforward: repeatedly optimize a trajectory at every time step to overcome error in the model and noise in the system. For systems with fast dynamics, the optimization problem must be solved at higher frequencies while larger systems with slower dynamics don't require new solutions as often. For autonomous, mobile robotic systems, the computational power a robot must carry to do onboard computation becomes a function of scale, where the compute a robot can physically hold is directly proportional to its size, the controller frequency required to stabilize the robot is inversely proportional to its size, and the size of the computer is directly proportional to the minimum required controller frequency. As a result, larger robots with slower dynamics require smaller computers than smaller robots with faster dynamics, many of which can only carry microcontrollers.

Reducing the size limit of computers for small robots creates two issues: low throughput and low memory. Not solving the model predictive control problem in the expected window of time will cause the controller to lag behind the actual system and might result in catastrophic behavior, and running out of memory means no work can be done on the problem in the first place. A solver tailored for tiny robots must address these issues, namely compressing the problem to reduce memory footprint and accelerating the problem to accommodate the higher frequency dynamics of smaller systems.

Reductions in memory footprint have been approached by creating solvers tailored for embedded systems [35, 56, 65] and by approximating the model predictive control problem using lightweight neural networks [6, 54, 60, 69]. Solvers such as ECOS [23] and HPIPM [26], among others, were built with the memory constraints of embedded systems in mind. ECOS and HPIPM, however, rely on interior-point methods that are difficult to warm start, which eliminates their potential for use in for model predictive control. Those that are capable of warm starting, such as OSQP [65] and SCS [57], are either too memory intensive, slow, or some combination due to their goal of solving generic quadratic programs rather than model predictive control problems specifically. Learned approaches to controlling tiny robots have been developed as well, and organizations such as TinyML [54] have shown that it is possible to condense these problems such that they fit on memory constrained and computationally constrained systems. However, learning approaches are inherently an approximation to the actual optimization problem and thus do not provide any constraint satisfaction guarantees without additional computation [40, 72].

1.1 Contributions

The goal of this work was to explore different optimization techniques that lend themselves to compressing and accelerating the model predictive control problem and to ultimately show that it is feasible and useful to solve MPC problems on tiny robots. The overall contributions are a model predictive control solver that compresses and accelerates the optimal control problem to a degree over existing state-of-the-art techniques that allows for controlling tiny robots, hardware demonstrations of the solver running onboard a Crazyflie 2.1 nano-quadrotor [13], an open-source software package with code generation capabilities, and an extension of the original solver that allows for handling second-order cone constraints with demonstrations on the same nano-quadrotor.

1.2 Thesis Structure

Chapter 2: Background

This chapter introduces concepts related to model predictive control and classical control that are helpful in understanding the techniques and algorithms described in subsequent chapters. It begins with a holistic view of optimization then narrows scope to the math relevant to this thesis.

Chapter 3: TinyMPC: Model-Predictive Control on Resource-Constrained Microcontrollers

Running model predictive controllers on tiny robots requires a suitably tiny solver. In this chapter we derive and describe the inner-workings of TinyMPC, a model predictive control solver tailored for microcontrollers. We discuss the advantages of using TinyMPC over more generic solvers and the sacrifices made to achieve gains in speed and reductions in memory footprint. We defend algorithmic choices and demonstrate the solver's real-world efficacy by controlling a nano-quadrotor subject to various convex constraints.

Chapter 4: Code Generation and Conic Model-Predictive Control with TinyMPC

A large section of optimization problems can be modeled using convex constraints. In this chapter we demonstrate TinyMPC's ability to easily include additional convex constraints. An example is shown using second-order cones. After, we describe the code generation abilities of TinyMPC and example usage from the Python interface. The addition of second-order cone constraints allows TinyMPC to handle an entirely new and large section of optimization problems, giving it the ability to reason about friction cones and land rockets, among many other scenarios. The addition of a code generation module allows problems solved with TinyMPC to be easily distributed and verified across different microcontrollers through the use of a high level interface.

Chapter 5: Practical Implementation Tips and Tricks

This chapter contains an unordered list of the implementation details used during the course of this research important to running TinyMPC on hardware. Information ranges from how to choose the best horizon length and selecting hyperparameter values for better convergence to tips and trick for performing obstacle avoidance.

Chapter 6: Conclusions

There are many directions this work may be taken and many who could benefit from its adoption. Chapter 5 discusses some of these directions along with various ways the existing software could be used, some of the tradeoffs that come with its use, and when to use TinyMPC in the first place.

Chapter 2

Background

This chapter describes optimal control and numerical optimization techniques fundamental to understanding the remainder of this thesis. We discuss the linear quadratic regulator and its generic solution via Riccati recursion, introduce a generic model predictive control formulation, and end with an explanation of the alternating direction method of multipliers (ADMM).

2.1 The Linear-Quadratic Regulator

The linear-quadratic regulator (LQR) [41] is a widely used approach for solving robotic control problems. LQR optimizes a quadratic cost function subject to a set of linear dynamics constraints:

$$\begin{aligned} \min_{x_{1:N}, u_{1:N-1}} \quad & J = \frac{1}{2} x_N^\intercal Q_f x_N + q_f^\intercal x_N + \\ & \sum_{k=1}^{N-1} \frac{1}{2} x_k^\intercal Q x_k + q_k^\intercal x_k + \frac{1}{2} u_k^\intercal R u_k + r_k^\intercal u_k \\ \text{subject to} \quad & x_{k+1} = A x_k + B u_k \quad \forall k \in [1, N), \end{aligned} \tag{2.1}$$

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), $A \in \mathbb{R}^{n \times n}$ and $B \in \mathbb{R}^{n \times m}$ define the system dynamics, $Q \succeq 0$, $R \succ 0$, and $Q_f \succeq 0$ are symmetric cost matrices and q and r are linear cost vectors.

Equation (2.1) has a closed-form solution in the form of an affine feedback controller [41]:

$$u_k^* = -K_k x_k - d_k. \tag{2.2}$$

The feedback gain K_k and feedforward d_k are found by solving the discrete Riccati equation backward in time, starting from $P_N = Q_f, p_N = q_f$, where P_k, p_k are the quadratic and linear terms of the cost-to-go (or value) function [41]:

$$\begin{aligned} K_k &= (R + B^\intercal P_{k+1} B)^{-1}(B^\intercal P_{k+1} A) \\ d_k &= (R + B^\intercal P_{k+1} B)^{-1}(B^\intercal p_{k+1} + r_k) \\ P_k &= Q + K_k^\intercal R K_k + (A - BK_k)^\intercal P_{k+1}(A - BK_k) \\ p_k &= q_k + (A - BK_k)^\intercal (p_{k+1} - P_{k+1} B d_k) + \\ &\qquad K_k^\intercal (R d_k - r_k). \end{aligned} \tag{2.3}$$

2.2 Convex Model Predictive Control

Convex MPC extends the LQR formulation to admit additional convex constraints on the system states and control inputs, such as joint and torque limits, hyperplanes for obstacle avoidance, and contact constraints:

$$\begin{aligned} \min_{x_{1:N}, u_{1:N-1}} \quad & J(x_{1:N}, u_{1:N-1}) \\ \text{subject to} \quad & x_{k+1} = A x_k + B u_k, \\ & x_k \in \mathcal{X}, u_k \in \mathcal{U}, \end{aligned} \tag{2.4}$$

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 the landing of rockets [9], legged locomotion [22], and autonomous driving [10].

When $\mathcal{X}$ and $\mathcal{U}$ can be expressed as linear constraints, (2.4) is a QP, and can be put into the standard form:

$$\begin{aligned} \min_{x \in \mathbb{R}^n} \quad & \frac{1}{2} x^\intercal P x + q^\intercal x \\ \text{subject to} \quad & Ax \leq b, \\ & Cx = d. \end{aligned} \tag{2.5}$$

Further analysis, including theoretical guarantees regarding feasibility and stability, can be found in [14] and [52].

2.3 The Alternating Direction Method of Multipliers

The alternating direction method of multipliers (ADMM) [17, 27, 29] is a popular and efficient approach for solving convex optimization problems, including QPs like (2.5). We provide a very brief summary here and refer readers to [15] for more details.

Given a generic problem:

$$\begin{aligned} \min_{x} \quad & f(x) \\ \text{subject to} \quad & x \in \mathcal{C}, \end{aligned} \tag{2.6}$$

with f and $\mathcal{C}$ convex, we define the indicator function for the set $\mathcal{C}$:

$$I_{\mathcal{C}}(z) = \begin{cases} 0 & z \in \mathcal{C} \\ \infty & \text{otherwise.} \end{cases} \tag{2.7}$$

We can now form the following equivalent problem by introducing the slack variable z:

$$\begin{aligned} \min_{x} \quad & f(x) + I_{\mathcal{C}}(z) \\ \text{subject to} \quad & x = z. \end{aligned} \tag{2.8}$$

The augmented Lagrangian of the transformed problem (2.8) is as follows, where λ is a Lagrange multiplier and ρ is a scalar penalty weight:

$$\mathcal{L}_A(x, z, \lambda) = f(x) + I_{\mathcal{C}}(z) + \lambda^\intercal (x - z) + \frac{\rho}{2} ||x - z||_2^2. \tag{2.9}$$

If we alternate minimization over x and z, rather than simultaneously minimizing over both, we arrive at the three-step ADMM iteration,

$$\begin{aligned} \text{primal update} : x^+ &= \arg\min_x \mathcal{L}_A(x, z, \lambda), && (2.10) \\ \text{slack update} : z^+ &= \arg\min_z \mathcal{L}_A(x^+, z, \lambda), && (2.11) \\ \text{dual update} : \lambda^+ &= \lambda + \rho(x^+ - z^+), && (2.12) \end{aligned}$$

the last step of which is a gradient-ascent update on the Lagrange multiplier [17]. These steps can be iterated until a desired convergence tolerance is achieved.

In the special case of a QP, 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 projection. ADMM-based QP solvers, like OSQP [65], have demonstrated state-of-the-art results.

Chapter 3

TinyMPC: Model Predictive Control on Resource Constrained Microcontrollers

Model predictive control (MPC) is a powerful tool for controlling highly dynamic robotic systems subject to complex constraints. However, MPC is computationally demanding, and is often impractical to implement on small, resource-constrained robotic platforms. We present TinyMPC, a high-speed MPC solver with a low memory footprint targeting the microcontrollers common on small robots. Our approach is based on the alternating direction method of multipliers (ADMM) and leverages the structure of the MPC problem for efficiency. We demonstrate TinyMPC's effectiveness by benchmarking against the state-of-the-art solver OSQP, achieving nearly an order of magnitude speed increase, as well as through hardware experiments on a 27 gram quadrotor, demonstrating high-speed trajectory tracking and dynamic obstacle avoidance.

3.1 Introduction

Model predictive control (MPC) enables reactive and dynamic online control for robots while respecting complex control and state constraints such as those encountered during dynamic obstacle avoidance and contact events [21, 38, 47, 70]. However, despite MPC's many successes, its practical application is often hindered by computational limitations, which can necessitate algorithmic simplifications [53, 59]. This challenge is amplified when dealing with systems that have fast or unstable open-loop dynamics, where high control rates are needed for safe and effective operation.

At the same time, there has been an explosion of interest in tiny, low-cost robots that can operate in confined spaces, making them a promising solution for applications ranging from emergency search and rescue [50] to routine monitoring and maintenance of infrastructure and equipment [20, 24]. These robots are limited to low-power, resource-constrained microcontrollers (MCUs) for their computation [25, 58]. As shown in Figure 3.2, these microcontrollers feature orders of magnitude less RAM, flash memory, and processor speed compared to the CPUs and GPUs available on larger robots and historically were not able to support the real-time execution of computationally or memory-intensive algorithms [54, 73]. Consequently, many examples in the literature of intelligent robot behaviors executed on these tiny platforms rely on off-board computers [4, 19, 39, 44, 68, 69, 71].

Several efficient optimization solvers and techniques suitable for embedded MPC have emerged in recent years [35, 56]. Notable software packages among these are OSQP [65] and CVXGEN [49]. Both of these solvers have code-generation tools that enable users to create dependency-free C code to solve quadratic programs (QPs) on embedded computers. However, they do not take full advantage of the unique structure of the MPC problem, generally making them too memory intensive and too computationally demanding to run within the resource constraints of many microcontrollers.

On the other hand, the recent success of "TinyML" has enabled the deployment of neural networks on microcontrollers [54]. Motivated by these results, we developed TinyMPC, an MCU-optimized implementation of convex MPC using the alternating direction method of multipliers (ADMM) algorithm. At its core, our solver is designed to *accelerate and compress* the ADMM algorithm by *exploiting the structure* of the MPC problem.

In particular, we precompute and cache expensive matrix factorizations, allowing TinyMPC to completely avoid online division and matrix inversion. This approach enables rapid computation with a very small memory footprint, enabling deployment

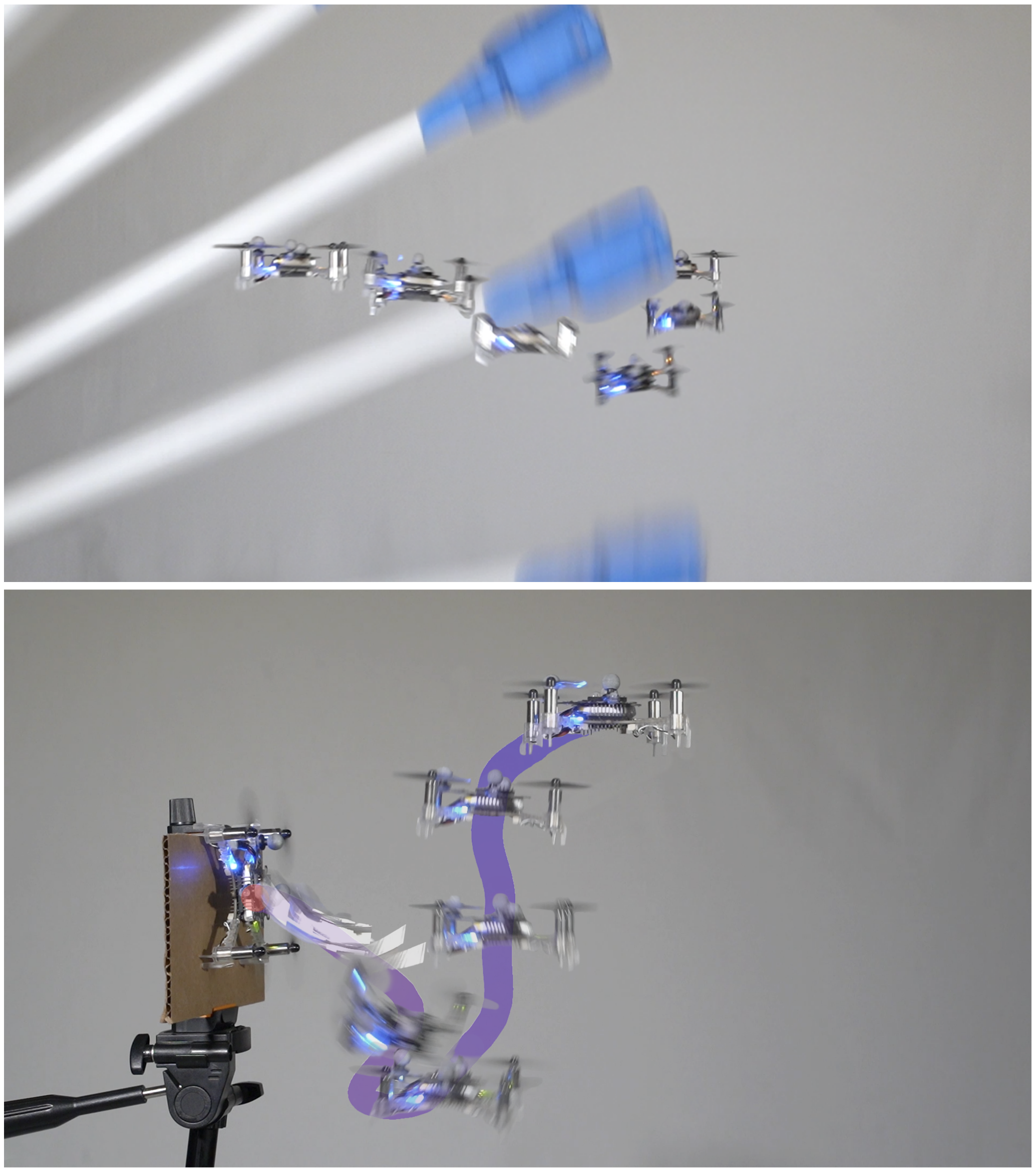
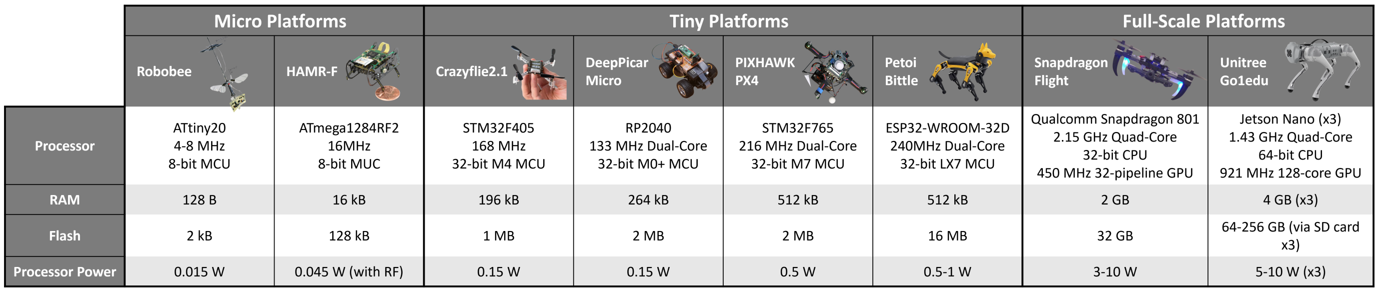


Figure 3.1: TinyMPC is a fast convex model predictive control solver that enables real-time optimal control on resource-constrained microcontrollers. We demonstrate its efficacy in dynamic obstacle avoidance (top) and recovery from 90° attitude errors (bottom) on a 27 gram Crazyflie quadrotor.

on resource-constrained MCUs. To the best of the authors' knowledge, TinyMPC is the first MPC solver tailored for execution on these MCUs that has been demonstrated onboard a highly dynamic, compute-limited robotic system. Our contributions include:

- A novel quadratic-programming algorithm that is optimized for MPC, is matrix-inversion free, and achieves high efficiency and a very low memory footprint.



	Micro Platforms		Tiny Platforms				Full-Scale Platforms	
	Robobee	HAMR-F	Crazyflie2.1	DeepPicar Micro	PIXHAWK PX4	Petoi Bittle	Snapdragon Flight	Unitree Go1edu
Processor	ATtiny20 4-8 MHz 8-bit MCU	ATmega1284RF2 16MHz 8-bit MUC	STM32F405 168 MHz 32-bit M4 MCU	RP2040 133 MHz Dual-Core 32-bit M0+ MCU	STM32F765 216 MHz Dual-Core 32-bit M7 MCU	ESP32-WROOM-32D 240MHz Dual-Core 32-bit LX7 MCU	Qualcomm Snapdragon 801 2.15 GHz Quad-Core 32-bit CPU 450 MHz 32-pipeline GPU	Jetson Nano (x3) 1.43 GHz Quad-Core 64-bit CPU 921 MHz 128-core GPU
RAM	128 B	16 kB	196 kB	264 kB	512 kB	512 kB	2 GB	4 GB (x3)
Flash	2 kB	128 kB	1 MB	2 MB	2 MB	16 MB	32 GB	64-256 GB (via SD card x3)
Processor Power	0.015 W	0.045 W (with RF)	0.15 W	0.15 W	0.5 W	0.5-1 W	3-10 W	5-10 W (x3)

Figure 3.2: A comparison of micro, tiny, and full-scale robot platforms and their associated computational hardware.

This combination makes it suitable for deployment on resource-constrained microcontrollers.

- An open-source implementation of TinyMPC in C++ that delivers state-of-the-art real-time performance for convex MPC problems on microcontrollers.
- Experimental demonstrations on a small, agile, resource-constrained quadrotor platform.

This paper proceeds as follows: Section 4.2 reviews linear-quadratic optimal control, convex optimization, and ADMM. Section 3.2 then derives the core TinyMPC solver algorithm. Benchmarking results and hardware experiments on a Crazyflie quadrotor are presented in Section 4.5. Finally, we summarize our results and conclusions in Section 3.4.

3.2 The TinyMPC Solver

TinyMPC trades generality for speed by exploiting the special structure of the MPC problem. Specifically, we leverage the closed-form Riccati solution to the LQR problem to compute the primal update in (2.10). Pre-computing and caching this solution allows us to avoid online matrix factorizations and enables very fast performance while maintaining a small memory footprint.

3.2.1 Combining LQR and ADMM for MPC

We solve the following problem, introducing slack variables as in (2.9) and transforming (2.4) into the following:

$$\begin{aligned}
\min_{\substack{x_{1:N}, z_{1:N},\\ \lambda_{1:N}, u_{1:N-1},\\ w_{1:N-1}, \mu_{1:N-1}}} \quad & \mathcal{L}_A(\cdot) = J(x_{1:N}, u_{1:N-1}) + \\
& \qquad I_{\mathcal{X}}(z_{1:N}) + I_{\mathcal{U}}(w_{1:N-1}) + \\
& \sum_{k=1}^{N} \frac{\rho}{2}(x_k - z_k)^\intercal (x_k - z_k) + \lambda_k^\intercal (x_k - z_k) + \\
& \sum_{k=1}^{N-1} \frac{\rho}{2}(u_k - w_k)^\intercal (u_k - w_k) + \mu_k^\intercal (u_k - w_k) \\
\text{subject to:} \quad & x_{k+1} = Ax_k + Bu_k,
\end{aligned} \tag{3.1}$$

where z, w, λ, and μ are the state slack, input slack, state dual, and input dual variables over the entire horizon. State and input constraints are enforced through the indicator functions $I_{\mathcal{X}}$ and $I_{\mathcal{U}}$. We use the ADMM algorithm (2.10), (2.11), (2.12) to solve this optimal control problem. The primal update for (3.1) becomes an equality-constrained QP:

$$\begin{aligned}
\min_{x_{1:N}, u_{1:N-1}} \quad & \frac{1}{2} x_N^\intercal \tilde{Q}_f x_N + \tilde{q}_f^\intercal x_N + \\
& \sum_{k=1}^{N-1} \frac{1}{2} x_k^\intercal \tilde{Q} x_k + \tilde{q}_k^\intercal x_k + \frac{1}{2} u_k^\intercal \tilde{R} u_k + \tilde{r}^\intercal u_k \\
\text{subject to} \quad & x_{k+1} = Ax_k + Bu_k,
\end{aligned} \tag{3.2}$$

where

$$\begin{aligned}
\tilde{Q}_f &= Q_f + \rho I, & \tilde{q}_f &= q_f + \lambda_N - \rho z_N, \\
\tilde{Q} &= Q + \rho I, & \tilde{q}_k &= q_k + \lambda_k - \rho z_k, \\
\tilde{R} &= R + \rho I, & \tilde{r}_k &= r_k + \mu_k - \rho w_k.
\end{aligned} \tag{3.3}$$

We reformulate (3.3) and introduce the scaled dual variables y and g for convenience [17]:

$$\begin{aligned}\tilde{q}_f &= q_f + \rho(\lambda_N/\rho - z_N) = q_f + \rho(y_N - z_N),\\ \tilde{q}_k &= q_k + \rho(\lambda_k/\rho - z_k) = q_k + \rho(y_k - z_k),\\ \tilde{r}_k &= r_k + \rho(\mu_k/\rho - w_k) = r_k + \rho(g_k - w_k).\end{aligned} \tag{3.4}$$

We observe that, because (3.2) exhibits the same LQR problem structure as in (2.1), it can be solved efficiently with the Riccati recursion in (2.3). The slack update for (3.1) becomes a simple linear projection onto the feasible set:

$$\begin{aligned}z_k^+ &= \mathrm{proj}_{\mathcal{X}}(x_k^+ + y_k),\\ w_k^+ &= \mathrm{proj}_{\mathcal{U}}(u_k^+ + g_k),\end{aligned} \tag{3.5}$$

where the superscript denotes the variable at the subsequent ADMM iteration. The dual update for (3.1) becomes:

$$\begin{aligned}y_k^+ &= y_k + x_k^+ - z_k^+,\\ g_k^+ &= g_k + u_k^+ - w_k^+.\end{aligned} \tag{3.6}$$

Finally, the algorithm terminates when the primal and dual residuals are within a set tolerance.

3.2.2 Pre-Computation

Solving the linear system in each primal update is the most expensive step in each ADMM iteration. In our case, this is the solution to the Riccati equation, which has properties we can leverage to significantly reduce computation and memory usage. Given a long enough horizon, the Riccati recursion (2.3) converges to the constant solution of the infinite-horizon LQR problem [41]. Thus, we pre-compute a single LQR gain matrix K_{inf} and cost-to-go Hessian P_{inf}. We then cache the following matrices from (2.3):

$$\begin{aligned} C_1 &= (R + B^\intercal P_{\text{inf}} B)^{-1}, \\ C_2 &= (A - BK_{\text{inf}})^\intercal. \end{aligned} \tag{3.7}$$

A careful analysis of the Riccati equation then reveals that only the linear terms need to be updated as part of the ADMM iteration:

$$\begin{aligned} d_k &= C_1(B^\intercal p_{k+1} + r_k), \\ p_k &= q_k + C_2 p_{k+1} - K_{\text{inf}}^\intercal r_k. \end{aligned} \tag{3.8}$$

As a result, we completely avoid online matrix factorization and only compute matrix-vector products. We also dramatically reduce memory footprint by only storing a few vectors at each time step.

3.2.3 Penalty Scaling

ADMM is sensitive to the value of the penalty term ρ in (2.9). Solvers like OSQP [65] overcome this issue by adaptively scaling ρ. However, this requires performing additional matrix factorizations. To avoid this, we pre-compute and cache a set of matrices corresponding to several values of ρ. Online, we switch between these cached matrices based on the values of the primal and dual residual values in a scheme adapted from OSQP. The resulting TinyMPC algorithm is summarized in Algorithm 1.

Algorithm 1 TinyMPC

```
  function TINY_SOLVE(input)
      while not converged do
          //Primal update
          $p_{1:N-1}, d_{1:N-1} \leftarrow$ Backward pass via (3.8)
          $x_{1:N}, u_{1:N-1} \leftarrow$ Forward pass via (2.2)
          //Slack update
          $z_{1:N}, w_{1:N-1} \leftarrow$ Project to feasible set (3.5)
          //Dual update
          $y_{1:N}, g_{1:N-1} \leftarrow$ Gradient ascent (3.6)
          $q_{1:N}, r_{1:N-1}, p_N \leftarrow$ Update linear cost terms
      return $x_{1:N}, u_{1:N-1}$
```

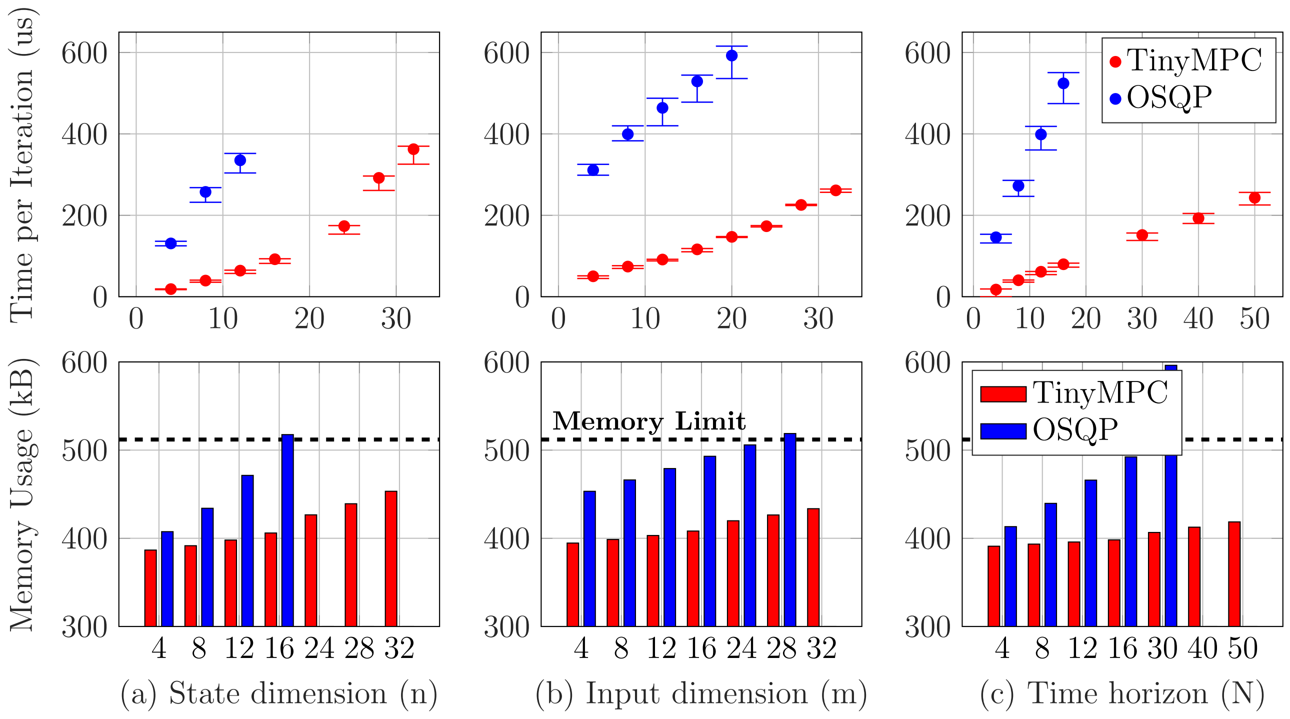


Figure 3.3: Comparison of average iteration times (top) and memory usage (bottom) for OSQP and TinyMPC on randomly generated trajectory tracking problems on a Teensy 4.1 development board.

3.3 Experiments

We evaluate TinyMPC through two sets of experiments: first, we benchmark our solver against the state-of-the-art OSQP [65] solver on a representative microcontroller,

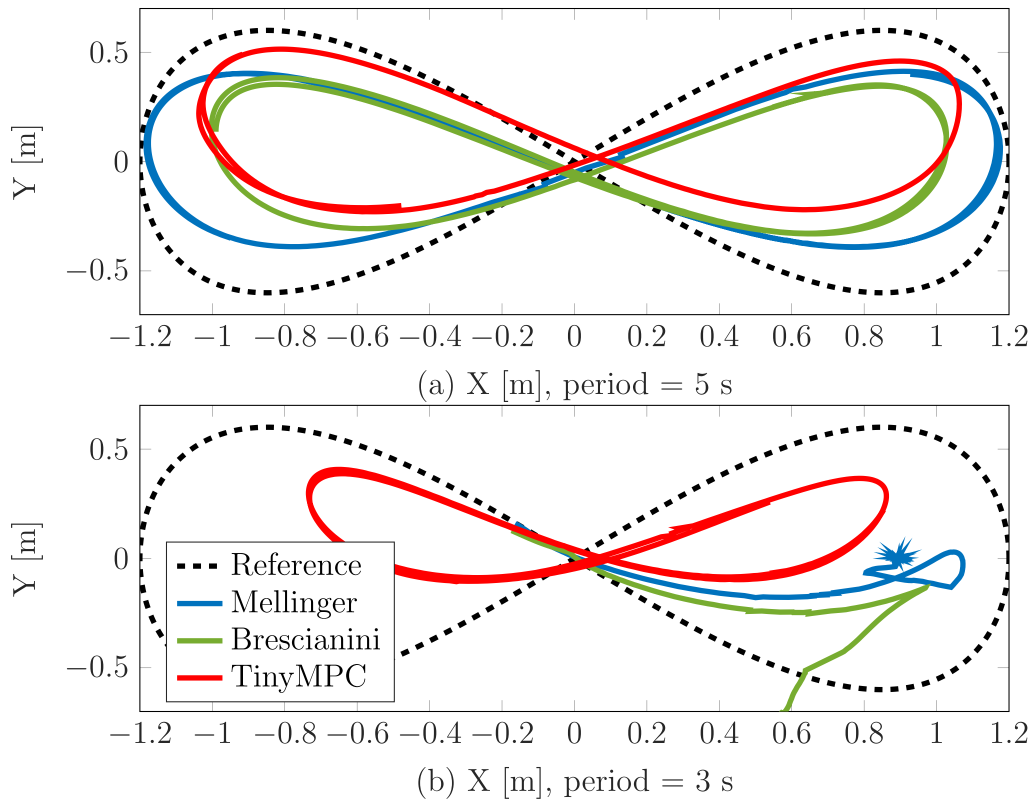


Figure 3.4: Figure-eight tracking at low speed (top) and high speed (bottom) comparing TinyMPC with the two most performant controllers available on the Crazyflie.

demonstrating improved computational speed and reduced memory footprint. We then test the efficacy of our solver on a resource-constrained nano-quadrotor platform, the Crazyflie 2.1. We show that TinyMPC enables the Crazyflie to track aggressive reference trajectories while satisfying control limits and time-varying state constraints.

3.3.1 Microcontroller Benchmarks

As shown in Fig. 3.3, we first compare TinyMPC and OSQP on random linear MPC problems while varying the state and input dimensions, as well as the horizon length.

Methodology

Experiments were performed on a Teensy 4.1 [2] development board, which has an ARM Cortex-M7 microcontroller operating at 600 MHz, 7.75 MB of flash memory,

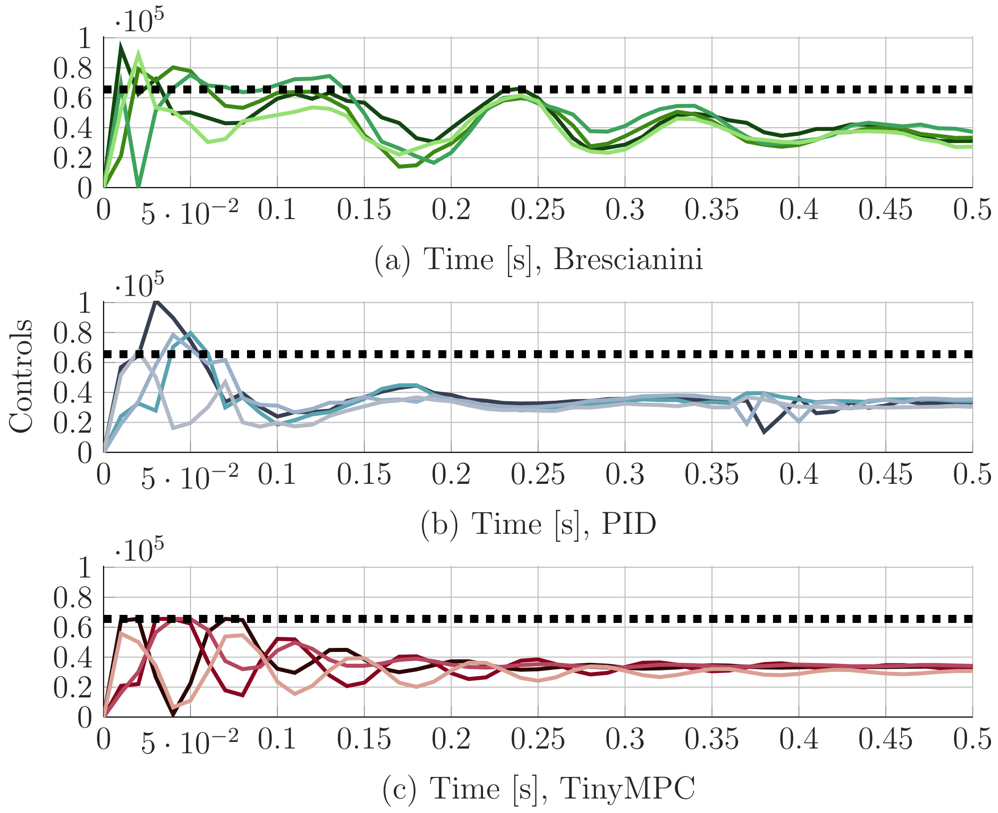


Figure 3.5: Control trajectories during the Extreme Initial Poses experiment.

and 512 kB of RAM. TinyMPC is implemented in C++ using the Eigen matrix library [31]. We used OSQP's code-generation feature to generate a C implementation of each problem to run on the microcontroller. Objective tolerances were set to 10^{-3} and constraint tolerances to 10^{-4}. The maximum number of iterations for both solvers was set to 4000, and both utilized warm starting. OSQP's solution polishing was disabled to decrease solve time. Other parameters were set to equivalent values wherever possible.

Dynamics models A and B were randomly generated and checked to ensure controllability for all values of state dimension n, input dimension m, and time horizon N. The control input was constrained within fixed bounds over the entire horizon. During the microcontroller tests, noise was added to mimic imperfect state estimation. The largest problem instance involved 696 decision variables, 490 linear equality constraints, and 392 linear inequality constraints.

Evaluation

Fig. 3.3 shows the average execution times for both solvers, in which TinyMPC exhibits a maximum speed-up of 8.85x over OSQP. This speed-up allows TinyMPC to perform real-time trajectory tracking while handling input constraints. OSQP also quickly exceeded the memory limitations of the MCU, while TinyMPC was able to scale to much larger problem sizes. For example, for a fixed input dimension of $m = 4$ and time horizon of $N = 10$ (Fig. 3.3a), OSQP exceeded the 512 kB memory limit of the Teensy at a state dimension of only $n = 16$, while TinyMPC only used around 400 kB at a state dimension of $n = 32$.

3.3.2 Hardware Experiments

We demonstrate the efficacy of our solver for real-time execution of dynamic control tasks on a resource-constrained Crazyflie 2.1 quadrotor. We present three experiments: 1) figure-eight trajectory tracking at slow and fast speeds, 2) recovery from extreme initial attitudes, and 3) dynamic obstacle avoidance through online updating of state constraints.

Methodology

The Crazyflie 2.1 is a 27 gram quadrotor. Its primary MCU is an ARM Cortex-M4 (STM32F405) clocked at 168 MHz with 192 kB of SRAM and 1 MB of flash. OSQP could not fit within the memory available on this MCU, making it impossible to be used as an MPC baseline. Instead, we compare against the four controllers included with the Crazyflie firmware: Cascaded PID [1], Mellinger [51], INDI [64], and Brescianini [18]. These are reactive controllers that often clip the control input to meet hardware constraints.

All experiments shown were performed in an OptiTrack motion-capture environment sending pose data to the Crazyflie at 100 Hz. TinyMPC ran at 500 Hz with a horizon length of $N = 15$ for the figure-eight tracking task and the attitude-recovery task. For the obstacle-avoidance task, we sent the location of the end of a stick to the Crazyflie using the onboard radio. Additionally, we reduced the MPC frequency to 100 Hz and increased N to 20.

In all experiments, we linearized the quadrotor's 6-DOF dynamics about a hover and represented its attitude with a quaternion using the formulation in [34]. This problem has state dimension $n = 12$ and $m = 4$ representing the quadrotor's full state and PWM motor commands. The largest problem was in the dynamic obstacle avoidance scenario, which was solved onboard at high frequency and consisted of 316 decision variables, 248 linear equality constraints, and 172 linear inequality constraints.

Evaluation—Figure-Eight Trajectory Tracking

We compare the tracking performance of TinyMPC and other controllers with a figure-eight trajectory, as shown in Fig. 3.4. For the faster trajectory, the maximum velocity and attitude deviation reached 1.5 m/s and 20°, respectively. Only TinyMPC could track the entire reference while respecting actuator limits, while the Mellinger and Brescianini controllers crashed almost immediately. TinyMPC converged at all steps within a maximum of 7 iterations and under the allotted 2 ms solve time defined by the 500 Hz control frequency.

Evaluation—Extreme Initial Poses

Fig. 3.1 (bottom) shows the performance of the Crazyflie when initialized with a 90° attitude error. TinyMPC displayed the best recovery performance with a maximum position error of 23 cm while respecting the input limits. The PID and Brescianini controllers achieved maximum errors of 40 cm and 65 cm, respectively, while violating input limits (Fig. 3.5). The other controllers, INDI and Mellinger, failed to stabilize the quadrotor, causing it to crash.

Evaluation—Dynamic Obstacle Avoidance

We demonstrate TinyMPC's ability to handle time-varying state constraints by avoiding a moving stick (Fig. 3.1 top). These experiments are more challenging because the constraints arbitrarily switch between inactive and active, requiring far more iterations to solve to convergence. The obstacle sphere was re-linearized about its updated position at each MPC step, allowing the drone to avoid the unplanned movements of the swinging stick. As illustrated, the quadrotor could move freely in space to avoid the dynamic obstacle and come back safely to the hovering position. As an additional challenge, we added a constraint such that the quadrotor must stay within a vertical plane defined by $x = 0$. The Crazyflie deviated a maximum of approximately 5 cm from this constraint plane while successfully avoiding the dynamic obstacle.

3.4 Conclusions

We introduce TinyMPC, a model predictive control solver for resource-constrained embedded systems. TinyMPC uses ADMM to handle state and input constraints while leveraging the structure of the MPC problem and insights from LQR to reduce memory footprint and speed up online execution compared to existing state-of-the-art solvers like OSQP. We demonstrate TinyMPC's practical performance on a Crazyflie nano-quadrotor performing highly dynamic tasks with input and obstacle constraints.

Chapter 4

Conic Constraints and Code Generation with TinyMPC

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 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 through multiple hardware experiments on the Crazyflie 2.1.

4.1 Introduction

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 [37, 42, 43]. 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.

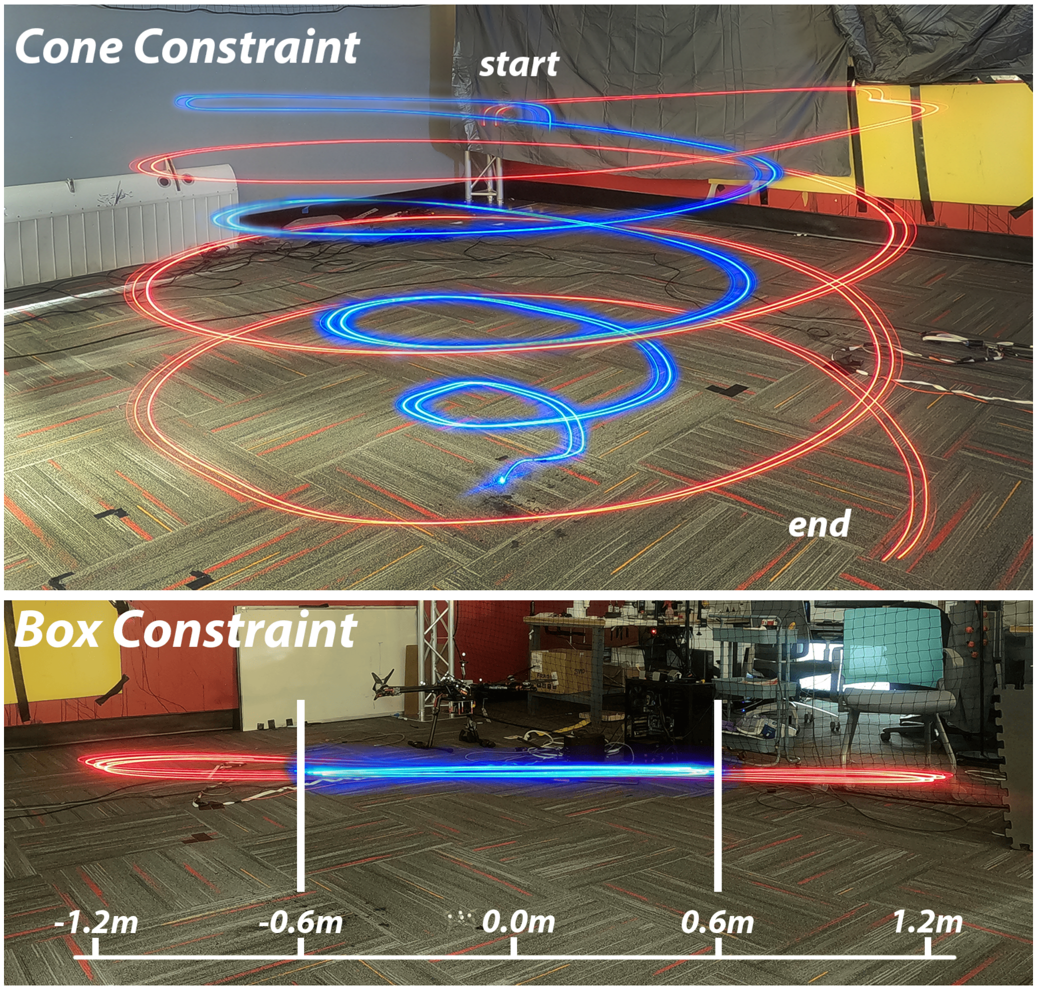


Figure 4.1: Top, we track a descending helical reference (red) with its position subject to a 45° second-order cone glideslope. This requires the aircraft to perform a spiral landing maneuver (blue). Bottom, we design a predictive safety filter to guarantee safe maneuvers within a box-shaped space (blue) regardless of the nominal controller behavior (red).

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 [3, 12, 33, 48]. Compiled code should also have a low memory footprint and be easily verifiable through an interface to a high-level language like Python. Table 4.1 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

Table 4.1: Comparison of general-purpose and model-predictive control solvers.

Solver	**SOC**	**Warm Starting**	**Embedded**	**Open Source**	**Quad. Obj.**	**MPC Tailored**
Clarabel [30]	✓	✗	✗	✓	✓	✗
COSMO [28]	✓	✓	✗	✓	✓	✗
ECOS [23]	✓	✗	✓	✓	✗	✗
MOSEK [8]	✓	✗	✗	✗	✓	✗
OSQP [65]	✗	✓	✓	✓	✓	✗
SCS [57]	✓	✓	✓	✓	✓	✗
FORCES [5]	✗	✓	✓	✗	✓	✓
ALTRO-C [33]	✓	✓	✗	✓	✓	✓
TinyMPC (ours)	✓	✓	✓	✓	✓	✓

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 [23] and SCS [57], 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 [55], a QP solver for model-predictive control based on the alternating direction method of multipliers (ADMM) [16]. TinyMPC avoids divisions while reducing computational complexity and memory 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 [55] 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 a Python interface 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 and C++.
- Experimental validation on hardware for several benchmark control problems.

The remainder of this paper is organized as follows: Section 4.2 provides mathematical preliminaries on the linear-quadratic regulator and the alternating-direction method of multipliers. Section 4.3 introduces TinyMPC and its extension to handle conic constraints. Section 4.4 gives an overview of the code generation software and demonstrates the minimum code required to set up a problem with TinyMPC. Section 4.5 details benchmarks on ARM Cortex M4 and M7 microcontrollers, as well as hardware experiments on a Crazyflie quadrotor. Finally, Section **??** summarizes our conclusions.

4.2 Background

4.2.1 Conic Model-Predictive Control

As opposed to 2.5, when the convex constraint sets $\mathcal{X}$ and $\mathcal{U}$ can be expressed as linear and second-order cone constraints, (2.4) is an SOCP, and can be put into the standard form (where $\mathcal{K}$ is a cone):

$$\begin{aligned} \min_{x \in \mathbb{R}^n} \quad & \frac{1}{2} x^\intercal P x + q^\intercal x \\ \text{subject to} \quad & x \in \mathcal{K}, \\ & Gx \leq h. \end{aligned} \tag{4.1}$$

4.2.2 The Alternating Direction Method of Multipliers

ADMM takes the same augmented Lagrangian as in 2.9 and three-step-update form as described in chapter 2. ADMM splits the problem into separate steps to divide the work done on the cost function and on satisfying the constraints. Because of this, it is straightforward to modify the slack update step to add additional convex constraints such as second-order cones. ADMM-based SOCP solvers such as SCS [57] have demonstrated state-of-the-art results.

4.3 Second-Order Conic Projection

TinyMPC exploits properties of the MPC problem to efficiently solve the primal update in (2.10), 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 (2.3) to reduce online computation as in chapter 3.

To add arbitrary convex constraints, we can write the slack update in (2.11) 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

$$\Pi(z) = \max(z_l, \min(z_u, z)), \tag{4.2}$$

where z corresponds to the state and control input slack variables. We now extend (2.1) to solve problems that include conic constraints defined by

$$\mathcal{K} = \left\{ z \in \mathbb{R}^n | z_n \geq \sqrt{z_1^2 + z_2^2 + \cdots + z_{n-1}^2} \right\}. \tag{4.3}$$

The structure of the ADMM algorithm inherently isolates the projection step, allowing us to replace the projection operator in the slack update (4.2) with the SOC projection

$$\Pi_{\mathcal{K}}(z) = \begin{cases} 0, & \|v\|_2 \leq -a, \\ z, & \|v\|_2 \leq a, \\ \frac{1}{2}(1 + \frac{a}{\|v\|_2}) \begin{bmatrix} v \\ \|v\|_2 \end{bmatrix}, & \|v\|_2 > |a|, \end{cases} \tag{4.4}$$

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. This projection can be used in addition to projections for other types of convex constraints, allowing ADMM to solve any problem with constraints defined by disciplined convex programming (DCP) [45].

4.4 Code Generation with TinyMPC

We have developed a code generation tool for TinyMPC with a Python interface that produces dependency-free C++ code. Listing 4.1 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.

The directory structure of the resulting code is shown in Fig. 4.2. 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 4.2).

Users may choose to compile code for their host system either manually or through the TinyMPC interface for testing. Listing 4.3 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`,

```python
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 4.1: A minimal Python script to generate the code for an MPC problem.

```
<proj_dir>
├── include
│   └── Eigen
├── tinympc
│   ├── [*.hpp]
│   └── [*.cpp]
└── src
    ├── tiny_data_workspace.cpp
    └── tiny_main.cpp
```

Figure 4.2: The tree structure of the generated code. The main program is stored in `tiny_main.cpp`.

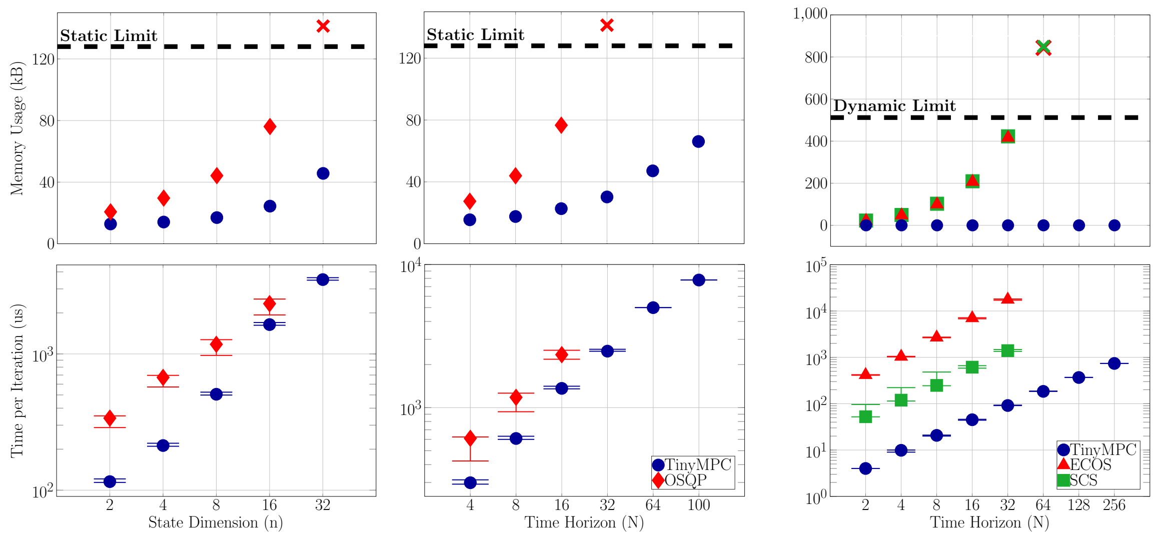
```cpp
#include "tinympc.hpp"
#include "tiny_data_workspace.hpp"

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

Listing 4.2: A simple C++ program that loads the problem data from `tiny_data_workspace.hpp` and solves the problem.

`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.

```python
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 4.3: An example Python script to run the generated code.



(a) Predictive safety filtering

(b) Rocket soft-landing

Figure 4.3: 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. (b) compares TinyMPC to ECOS and SCS on an SOCP-based rocket soft-landing using a Teensy 4.1 development board.

4.5 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 [13]. 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.

4.5.1 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. 4.3.

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 [11, 32]. 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 [55], 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. 4.3a 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 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 $n = 16$.

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 [9]. 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 [62]. C++ code for TinyMPC was produced using the Python code generation interface introduced in Section 4.4. 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. 4.3b 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 $N = 64$.

Table 4.2: 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.

A. Constraint Violation

Control Step (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

Control Step (ms)	**1000**	**20**	**10**	**2**
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**

Early Termination

High-rate real-time control requires a solver to return a solution within a strict time window. Table 4.2 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 4.5.1, 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. 4.3b). 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.

4.5.2 Hardware Experiments

We deployed our TinyMPC implementation onto a Crazyflie nano-quadrotor [13], 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 [9], running TinyMPC at 50 Hz and using the Crazyflie's built-in Brescinanini controller [18] 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. 4.1).

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 4.5.1. 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. 4.1 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.

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 [46]. 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. 4.4, 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

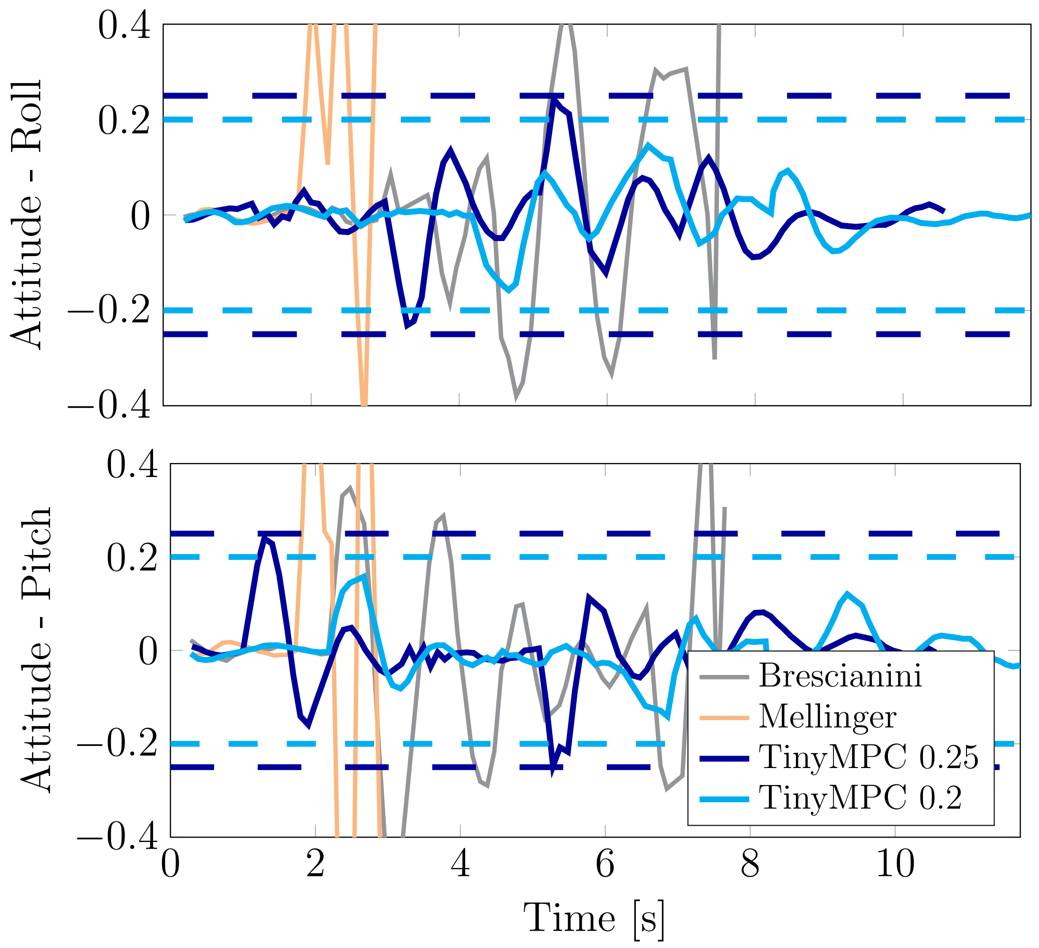


Figure 4.4: Comparison of attitude/thrust vector constraint violations by TinyMPC and different built-in controllers on the Crazyflie.

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.

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 the surroundings [46]. Fig. 4.1 demonstrates the ability of TinyMPC to handle the planetary landing glideslope constraint of spacecraft. 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 within a safe glideslope (blue).

Chapter 5

Practical Implementation Tips and Tricks

5.1 Solver Frequency vs Horizon Length

Horizon length is one of the most important parameters to choose when initializing TinyMPC. A longer horizon will almost always produce more dynamic behaviors, but only given that the solver will converge or come close to convergence at every time step. In a real-time scenario, there is a computational budget that restricts how many iterations may be performed at every time step. Longer horizons reduce the number of iterations that may be computed and potentially compromise the stability of the system. In a real implementation, stability analysis may be performed on a particular system to determine the minimum controller frequency. Horizon length may then be extended or shortened depending on the complexity of the task. Unfortunately, more complex tasks generally require more iterations to converge but also longer horizons to fully reason about the task. This is an unavoidable truth that limits the number of solver iterations that can be performed at every time step.

A general rule of thumb is to reduce the horizon length and increase the solver frequency for highly dynamic tasks that don't include constraints dependent on data external to the system's state, such as obstacles, and to increase horizon length as much as possible for tasks such as obstacle avoidance that require large deviations to the reference trajectory. For tasks that require long horizon lengths that result in an optimal control problem too complex to compute in real-time, sometimes it is necessary to add a hierarchial control scheme where TinyMPC computes a new trajectory at a slower rate than it would normally be used to control the system directly, and then track the output with a higher frequency controller, commonly implemented as a PID or LQR controller. This is sometimes necessary for systems with very high frequency dynamics, and while the low level controller might violate the constraints satisfies by TinyMPC, the hierarchial control scheme will still allow the system to reason about dynamically changing environments. It is also possible to add safety margins on TinyMPC's constraints when using a hierarchial controller such that the low level controller is less likely to violate the actual system limits.

5.2 Solution Tolerance

Solution tolerance determines how accurate the solver's answer must be before finishing. A tighter tolerance takes more time to reach, and different solvers have different solution tolerance time curves. ADMM, for example, is capable of solving to coarse tolerances very quickly but takes much longer to solve to tight tolerances. Solvers sometimes include a fine-tuning step that is specifically built for converging to tight tolerances. In our case, however, solving to tight tolerances is detrimental. When solving as part of a feedback loop, the responsiveness of the loop is often limited by the speed of the solver, and the longer the solver takes the more error will accumulate between the most recent state estimate and the actual state of the system. One must strike a balance between fast solves and accurate solves. Solutions with very tight tolerances take longer to find and thus reduce the overall accuracy of the system, and very coarse tolerances can result in solutions that haven't converged enough to be useful.

5.2.1 Continuous Iteration

For a problem where we care about knowing that the solver has converged at each time step, the best solution tolerance to use is often found through trial and error. Another approach is to simply not check whether we have converged. This has the benefit of not spending additional CPU cycles computing solution tolerance, which can be more expensive than another solver iteration.

For all of the demonstrations presented in this thesis, we did not check for convergence and instead assumed we would endlessly solve the MPC problem. New state estimation data was fed to the system at every control time step, which was 500 Hz for most tasks, and each control time step consisted of only a few solver iterations. This scheme of high control frequency and low solver iterations makes sense in real-time scenarios where the system is constantly evolving and solving the optimal control problem with old data has rapidly diminishing returns. This setup only works because of warm starting, which ensures that work done on solving the MPC problem in previous time steps is carried over into future time steps. One can imagine that when the system is at equilibrium and the state is not changing rapidly, the solution error will become very small. When the system is undergoing more change, as when tracking an aggressive reference trajectory, the solution error will increase. This is because the initial state of the system is constantly changing in these scenarios and not enough iterations are performed per control step to converge to a tighter tolerance.

5.3 Penalty Parameters

Most ADMM-based solvers compute a new KKT system online when they determine that the penalty parameter ρ needs to be updated. OSQP, for example, uses a heuristic that compares the speed of convergence for the constraints and cost function, and updates ρ when that ratio is beyond a certain threshold. Since TinyMPC relies on offline computation of expensive matrices, cached values that correspond to different values of ρ are all computed offline. A similar check to the one used by OSQP is performed online and the precomputed values corresponding to the closest ρ, known as a cache level in TinyMPC, is chosen for use in the next iteration. With the current implementation, each cache level must be precomputed offline. The values for ρ are chosen by trial and error but are generally scaled geometrically between a few orders of magnitude. For example, ρ might be between 1e-3 to 1e3, but for specific tasks that push the limits of the microcontroller in terms of required solution frequency, it is possible to test different precomputed cache levels in simulation to determine which ones result in the fastest convergence.

It may also be important to switch between different cache levels based on external circumstances. For example, in the obstacle avoidance task, we set two different values of ρ and overwrote the penalty switching heuristic to force a change whenever the drone violated the obstacle constraints. Similar results may be achieved with the penalty update heuristic. We chose to explicitly change the penalty parameter using problem specific data to increase responsiveness.

5.4 Obstacle Avoidance

The controller must run alongside additional necessary processes such as the state estimator, and for most microcontrollers the computational budget for each time step is taken up by these two tasks. However, TinyMPC is fast enough that additional computation may be performed without compromising these necessary processes. In the dynamic obstacle avoidance example, we stream the location of the end of the stick to the Crazyflie at every time step. We then perform a number of preprocessing steps before solving the actual MPC problem. We assume an inflated sphere surrounds the end of the stick we are trying to avoid and, given the most recent state of the Crazyflie and the sphere, compute a plane tangent to the sphere at the point closest to the drone. This plane defines a halfspace constraint that requires the drone stay outside of the sphere. We do this for each time step, so there are as many halfspace constraints as there are knot points in the horizon. We use the velocity of the sphere to predict its location at each time step before linearizing each halfspace constraint.

Chapter 6

Conclusions

6.1 Summary

In this work we present TinyMPC, a solver for convex model predictive control on resource constrained robots. The solver outperforms the state-of-the-art QP solver OSQP both in terms of memory footprint and solution speed. We then extend TinyMPC to handle second-order cone constraints and show that it outperforms the state-of-the-art conically-constrained QP solvers SCS and ECOS in the same metrics.

We demonstrate TinyMPC's efficacy on the Crazyflie 2.1, a 27 gram nano-quadrotor with fast dynamics. TinyMPC is able to control the Crazyflie to maneuver around dynamic obstacles, reasoning about constantly changing linear inequality constraints in real time. Although TinyMPC reasons about only a single linearization of the full nonlinear system dynamics, it can still successfully command aggressive maneuvers. This is shown by stabilizing the Crazyflie after taking off at a 90 degree angle. We show that reasoning about thrust limits results in a smoother recovery than obtained with the built-in controllers, which do not reason about thrust limits. We also demonstrate TinyMPC's second-order cone constraint handling capabilities by following a helical descent path subject to a conic position constraint. Finally, we perform thrust vector regulation by constraining the attitude of the Crazyflie to stay within a second-order cone. We show that this additional conic constraint allows the Crazyflie to successfully track an aggressive maneuver while built-in controllers fail to do so.

The code used to perform these experiments has been open-sourced on GitHub at `https://github.com/TinyMPC`. Additional information is available at the associated website, `https://tinympc.org`. We developed a code generation tool to streamline the process of integrating TinyMPC with various microcontrollers and a Python package to make the solver more accessible. The Python package is able to generate microcontroller-flashable code and is also capable of hooking into generated Python bindings for the newly generated C++ code for validation in a higher level language. This allows anyone with an existing pipeline with simulation, estimation, and visualization in either C++ or Python to iterate and interact with the TinyMPC solver.

6.2 Future Work

TinyMPC's primary limitation is its ability to reason about only one set of linear dynamics. As shown in this work, a single linearization is often enough to perform a large number of maneuvers. However, for more extreme agility, it is sometimes necessary to be able to reason about the entire system. In these cases, it could be possible to extend TinyMPC by linearizing about multiple robot configurations and switching between them in real-time. Doing this would require additional memory and offline computation time proportional to the number of linearizations, but would allow the system to approximate the manifold of the true nonlinear dynamics.

Supporting numerous high level languages and deploying the solver on a wide range of microcontrollers to validate usability is an important part of making a solver accessible to a larger community. Future work might include developing packages for additional languages that bind to the existing C++ API.

Because the alternating direction method of multipliers separates constraint handling from making progress on the cost function, it is simple to add additional convex constraints to the problem. Thus, future work could include refactoring the existing code base to allow for arbitrary convex constraint inputs using a disciplined convex programming approach.

6.3 Impact

Model predictive control is generally considered less approachable than learning methods, and this is largely due to a lack of tooling. TinyMPC aims to introduce new techniques for compressing the model predictive control problem to a size suitable for tiny robots, but is also meant to be an accessible tool that can easily fit into existing software stacks. Integrating TinyMPC as an available default controller in software such as Bitcraze's Crazyflie firmware or PX4 autopilot would make model predictive control more familiar to a larger audience.

TinyMPC's speedups and memory footprint reductions over state-of-the-art solvers allows it to solve long horizon model predictive control problems in real time on resource constrained hardware. This enables more intelligent control on small robots and less power consuming control on large robots. Ultimately, these advancements bridge the gap between computationally intensive convex model-predictive control and resource-constrained processing platforms.

Bibliography

[1] Controllers in the Crazyflie — Bitcraze. URL https://www.bitcraze.io/documentation/repository/crazyflie-firmware/master/functional-areas/sensor-to-control/controllers/.

[2] Teensy® 4.1. URL https://www.pjrc.com/store/teensy41.html.

[3] Emre Adabag, Miloni Atal, William Gerard, and Brian 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.

[4] Vivek Adajania, Siqi Zhou, Singh Arun, and Angela Schoellig. Amswarm: An alternating minimization approach for safe motion planning of quadrotor swarms in cluttered environments. In *2023 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1421–1427, 2023.

[5] Embotech AG. Forcespro, 2014–2023. URL https://forces.embotech.com/.

[6] Kwangjun Ahn, Zakaria Mhammedi, Horia Mania, Zhang-Wei Hong, and Ali Jadbabaie. Model predictive control via on-policy imitation learning, 2022.

[7] Lars Imsland Anne Mai Ersdal, Davide Fabozzi and Nina F. Thornhill. Model predictive control for power system frequency control taking into account imbalance uncertainty. *SEAMS-UGM International Conference on Mathematics and its Applications*, 2014. doi: https://doi.org/10.3182/20140824-6-ZA-1003.01631. URL https://www.sciencedirect.com/science/article/pii/S1474667016417428.

[8] MOSEK ApS. *Introducing the MOSEK Optimization Suite 10.1.28*, 2024. URL https://docs.mosek.com/latest/intro/index.html.

[9] Behçet Açıkmeşe, John M. Carson, and Lars Blackmore. Lossless convexification of nonconvex control bound and pointing constraints of the soft landing optimal control problem. *IEEE Transactions on Control Systems Technology*, 21(6): 2104–2113, 2013. doi: 10.1109/TCST.2012.2237346.

[10] Mithun Babu, Yash Oza, Arun Kumar Singh, K. Madhava Krishna, and Shanti

Medasani. Model predictive control for autonomous driving based on time scaled collision cone. In *2018 European Control Conference (ECC)*, pages 641–648, 2018. doi: 10.23919/ECC.2018.8550510.

[11] Federico Pizarro Bejarano, Lukas Brunke, and Angela 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)*, pages 4723–4730. IEEE, 2023.

[12] Arun L. Bishop, John Z. Zhang, Swaminathan Gurumurthy, Kevin Tracy, and Zachary 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.

[13] Bitcraze. Crazyflie 2.1, 2023. URL https://www.bitcraze.io/products/crazyflie-2-1/.

[14] Andrea Boccia, Lars Grüne, and Karl Worthmann. Stability and feasibility of state constrained mpc without stabilizing terminal constraints. *Systems and Control Letters*, 72:14–21, 2014. ISSN 0167-6911. doi: https://doi.org/10.1016/j.sysconle.2014.08.002. URL https://www.sciencedirect.com/science/article/pii/S0167691114001595.

[15] Stephen Boyd, Neal Parikh, Eric Chu, Borja Peleato, and Jonathan Eckstein. 2011.

[16] Stephen Boyd, Neal Parikh, Eric Chu, Borja Peleato, Jonathan Eckstein, et al. Distributed optimization and statistical learning via the alternating direction method of multipliers. *Foundations and Trends® in Machine learning*, 3(1): 1–122, 2011.

[17] Stephen Boyd, Neal Parikh, Eric Chu, Borja Peleato, Jonathan Eckstein, et al. Distributed optimization and statistical learning via the alternating direction method of multipliers. *Foundations and Trends® in Machine learning*, 3(1): 1–122, 2011.

[18] Dario Brescianini, Markus Hehn, and Raffaello D'Andrea. Nonlinear quadrocopter attitude control. 2013.

[19] Kong Yao Chee, Tom Z Jiahao, and M Ani Hsieh. Knode-mpc: A knowledge-based data-driven predictive control framework for aerial robots. *IEEE Robotics and Automation Letters*, 7(2):2819–2826, 2022.

[20] Sébastien D De Rivaz, Benjamin Goldberg, Neel Doshi, Kaushik Jayaram, Jack Zhou, and Robert J Wood. Inverted and vertical climbing of a quadrupedal microrobot using electroadhesion. *Science Robotics*, 3(25):eaau3038, 2018.

[21] Jared Di Carlo. *Software and control design for the MIT Cheetah quadruped*

robots. PhD thesis, Massachusetts Institute of Technology, 2020.

[22] Jared Di Carlo, Patrick M. Wensing, Benjamin Katz, Gerardo Bledt, and Sangbae 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)*, pages 1–9, 2018. doi: 10.1109/IROS.2018.8594448.

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

[24] Bardienus P Duisterhof, Shushuai Li, Javier Burgués, Vijay Janapa Reddi, and Guido CHE de Croon. Sniffy bug: A fully autonomous swarm of gas-seeking nano quadcopters in cluttered environments. In *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 9099–9106. IEEE, 2021.

[25] 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`.

[26] Gianluca Frison and Moritz Diehl. Hpipm: a high-performance quadratic programming framework for model predictive control. *IFAC-PapersOnLine*, 53(2): 6563–6569, 2020.

[27] Daniel Gabay and Bertrand Mercier. A dual algorithm for the solution of nonlinear variational problems via finite element approximation. *Computers & mathematics with applications*, 2(1):17–40, 1976.

[28] Michael Garstka, Mark Cannon, and Paul Goulart. Cosmo: A conic operator splitting method for large convex problems. In *2019 18th European Control Conference (ECC)*, pages 1951–1956. IEEE, 2019.

[29] Roland Glowinski and Americo Marroco. Sur l'approximation, par éléments finis d'ordre un, et la résolution, par pénalisation-dualité d'une classe de problèmes de dirichlet non linéaires. *Revue française d'automatique, informatique, recherche opérationnelle. Analyse numérique*, 9(R2):41–76, 1975.

[30] Paul Goulart and Yuwen Chen. Clarabel, 2022. URL `https://oxfordcontrol.github.io/ClarabelDocs/stable/`.

[31] Gaël Guennebaud, Benoît Jacob, et al. Eigen v3. http://eigen.tuxfamily.org, 2010.

[32] Kai-Chieh Hsu, Haimin Hu, and Jaime F Fisac. The safety filter: A unified view of safety-critical control in autonomous systems. *Annual Review of Control, Robotics, and Autonomous Systems*, 7, 2023.

[33] Brian E Jackson, Tarun Punnoose, Daniel Neamati, Kevin Tracy, Rianna Jitosho, and Zachary Manchester. Altro-c: A fast solver for conic model-predictive control.

In *2021 IEEE International Conference on Robotics and Automation (ICRA)*, pages 7357–7364. IEEE, 2021.

[34] Brian E. Jackson, Kevin Tracy, and Zachary Manchester. Planning with attitude. *IEEE Robotics and Automation Letters*, 6(3):5658–5664, 2021. doi: 10.1109/LRA.2021.3052431.

[35] Juan L. Jerez, Paul J. Goulart, Stefan Richter, George A. Constantinides, Eric C. Kerrigan, and Manfred Morari. Embedded online optimization for model predictive control at megahertz rates. *IEEE Transactions on Automatic Control*, 59(12):3238–3251, 2014. doi: 10.1109/TAC.2014.2351991.

[36] Donghyun Kim, Jared Di Carlo, Benjamin Katz, Gerardo Bledt, and Sangbae Kim. Highly dynamic quadruped locomotion via whole-body impulse control and model predictive control. *arXiv preprint arXiv:1909.06586*, 2019.

[37] Charles A Klein and Sakon Kittivatcharapong. Optimal force distribution for the legs of a walking machine with friction cone constraints. *IEEE Transactions on Robotics and Automation*, 6(1):73–85, 1990.

[38] Scott Kuindersma. Taskable agility: Making useful dynamic behavior easier to create. Princeton Robotics Seminar, 4 2023.

[39] Nathan O Lambert, Daniel S Drew, Joseph Yaconelli, Sergey Levine, Roberto Calandra, and Kristofer SJ Pister. Low-level control of a quadrotor with deep model-based reinforcement learning. *IEEE Robotics and Automation Letters*, 4 (4):4224–4230, 2019.

[40] Keuntaek Lee, Kamil Saigol, and Evangelos A Theodorou. Safe end-to-end imitation learning for model predictive control. *IEEE International Conference on Intelligent Robots and Systems*, 2019.

[41] Frank L. Lewis, Draguna Vrabie, and V.L. Syrmos. Optimal Control, 1 2012. URL https://doi.org/10.1002/9781118122631.

[42] Xinfu Liu, Zuojun Shen, and Ping Lu. Entry trajectory optimization by second-order cone programming. *Journal of Guidance, Control, and Dynamics*, 39(2): 227–241, 2016.

[43] Miguel Sousa Lobo, Lieven Vandenberghe, Stephen Boyd, and Hervé Lebret. Applications of second-order cone programming. *Linear algebra and its applications*, 284(1-3):193–228, 1998.

[44] Carlos E Luis, Marijan Vukosavljev, and Angela P Schoellig. Online trajectory generation with distributed model predictive control for multi-robot motion planning. *IEEE Robotics and Automation Letters*, 5(2):604–611, 2020.

[45] S. Boyd M. Grant and Y. Ye. *Disciplined Convex Programming*. Global Optimization: From Theory to Implementation. Springer.

[46] Danylo Malyuta, Taylor P. Reynolds, Michael Szmuk, Thomas Lew, Riccardo Bonalli, Marco Pavone, and Behçet Açıkmeşe. Convex optimization for trajectory generation: A tutorial on generating dynamically feasible trajectories reliably and efficiently. *IEEE Control Systems Magazine*, 42(5):40–113, 2022. doi: 10.1109/MCS.2022.3187542.

[47] Zachary Manchester, Neel Doshi, Robert J Wood, and Scott Kuindersma. Contact-implicit trajectory optimization using variational integrators. *The International Journal of Robotics Research*, 38(12-13):1463–1476, 2019.

[48] Tobia Marcucci and Russ Tedrake. Warm start of mixed-integer programs for model predictive control of hybrid systems. *IEEE Transactions on Automatic Control*, 66(6):2433–2448, 2020.

[49] Jacob Mattingley and Stephen Boyd. CVXGEN: A code generator for embedded convex optimization. In *Optimization Engineering*, pages 1–27.

[50] KN McGuire, Christophe De Wagter, Karl Tuyls, HJ Kappen, and Guido CHE de Croon. Minimal navigation solution for a swarm of tiny flying robots to explore an unknown environment. *Science Robotics*, 4(35):eaaw9710, 2019.

[51] Daniel Mellinger and Vijay Kumar. Minimum snap trajectory generation and control for quadrotors. In *2011 IEEE International Conference on Robotics and Automation*, pages 2520–2525, 2011. doi: 10.1109/ICRA.2011.5980409.

[52] Arash Mohammadi, Houshyar Asadi, Shady Mohamed, Kyle Nelson, and Saeid Nahavandi. Optimizing model predictive control horizons using genetic algorithm for motion cueing algorithm. *Expert Systems with Applications*, 92:73–81, 2018. ISSN 0957-4174. doi: https://doi.org/10.1016/j.eswa.2017.09.004. URL https://www.sciencedirect.com/science/article/pii/S0957417417306000.

[53] Sabrina M. Neuman, Brian Plancher, Thomas Bourgeat, Thierry Tambe, Srinivas Devadas, and Vijay Janapa Reddi. Robomorphic computing: A design methodology for domain-specific accelerators parameterized by robot morphology. ASPLOS 2021, page 674–686, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383172. doi: 10.1145/3445814.3446746. URL https://doi-org.ezp-prod1.hul.harvard.edu/10.1145/3445814.3446746.

[54] Sabrina M Neuman, Brian Plancher, Bardienus P Duisterhof, Srivatsan Krishnan, Colby Banbury, Mark Mazumder, Shvetank Prakash, Jason Jabbour, Aleksandra Faust, Guido CHE de Croon, et al. Tiny robot learning: challenges and directions for machine learning in resource-constrained robots. In *2022 IEEE 4th International Conference on Artificial Intelligence Circuits and Systems (AICAS)*, pages 296–299. IEEE, 2022.

[55] Khai Nguyen, Sam Schoedel, Anoushka Alavilli, Brian Plancher, and Zachary Manchester. Tinympc: Model-predictive control on resource-constrained mi-

crocontrollers. In *IEEE International Conference on Robotics and Automation (ICRA)*, Yokohama, Japan, May. 2024.

[56] Brendan O'Donoghue, Georgios Stathopoulos, and Stephen Boyd. A splitting method for optimal control. *Control Systems Technology, IEEE Transactions on*, 21:2432–2442, 11 2013. doi: 10.1109/TCST.2012.2231960.

[57] Brendan O'Donoghue, Eric Chu, Neal Parikh, and Stephen Boyd. Conic optimization via operator splitting and homogeneous self-dual embedding. *Journal of Optimization Theory and Applications*, 169(3):1042–1068, June 2016. URL http://stanford.edu/~boyd/papers/scs.html.

[58] Petoi. Open source, programmable robot dog bittle. Available at https://www.petoi.com/pages/bittle-open-source-bionic-robot-dog (5.9.2023).

[59] Brian Plancher and Scott Kuindersma. A performance analysis of parallel differential dynamic programming on a gpu. In *Algorithmic Foundations of Robotics XIII: Proceedings of the 13th Workshop on the Algorithmic Foundations of Robotics 13*, pages 656–672. Springer, 2020.

[60] Alexander Reske, Jan Carius, Yuntao Ma, Farbod Farshidian, and Marco Hutter. Imitation learning from MPC for quadrupedal multi-gait control. In *2021 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, may 2021. doi: 10.1109/icra48506.2021.9561444. URL https://doi.org/10.1109%2Ficra48506.2021.9561444.

[61] Paniz Hosseini Saman Taheri and Ali Razban. Model predictive control of heating, ventilation, and air conditioning (hvac) systems: A state-of-the-art review. *Journal of Building Engineering*, 2022. doi: https://doi.org/10.1016/j.jobe.2022.105067. URL https://www.sciencedirect.com/science/article/abs/pii/S2352710222010750.

[62] Maximilian Schaller, Goran Banjac, Steven Diamond, Akshay Agrawal, Bartolomeo Stellato, and Stephen Boyd. Embedded code generation with cvxpy. *IEEE Control Systems Letters*, 6:2653–2658, 2022. doi: 10.1109/LCSYS.2022.3173209.

[63] Dale E Seborg, Duncan A Mellichamp, and Thomas F Edgar. *Process Dynamics and Control*. World Scientific Publishing.

[64] Ewoud J. J. Smeur, Qiping Chu, and Guido C. H. E. de Croon. Adaptive incremental nonlinear dynamic inversion for attitude control of micro air vehicles. *Journal of Guidance, Control, and Dynamics*, 39(3):450–461, 2016. doi: 10.2514/1.G001490.

[65] Bartolomeo Stellato, Goran Banjac, Paul Goulart, Alberto Bemporad, and Stephen Boyd. Osqp: An operator splitting solver for quadratic programs. *Mathematical Programming Computation*, 12(4):637–672, 2020.

[66] Ravi Sekhar Sushma Parihar, Pritesh Shah and Jui Lagoo. Model predictive control and its role in biomedical therapeutic automation: A brief review. *Applied System Innovation*, 2022. doi: https://doi.org/10.3390/asi5060118. URL https://www.mdpi.com/2571-5577/5/6/118.

[67] Wawan Hafid Syaifudin and Endah R. M. Putri. Model predictive control of heating, ventilation, and air conditioning (hvac) systems: A state-of-the-art review. *SEAMS-UGM International Conference on Mathematics and its Applications*, 2019. doi: https://doi.org/10.1063/1.5139166. URL https://pubs.aip.org/aip/acp/article/2192/1/060020/756115/The-application-of-model-predictive-control-on.

[68] Guillem Torrente, Elia Kaufmann, Philipp Föhn, and Davide Scaramuzza. Data-driven mpc for quadrotors. *IEEE Robotics and Automation Letters*, 6(2):3769–3776, 2021.

[69] Pratyush Varshney, Gajendra Nagar, and Indranil Saha. Deepcontrol: Energy-efficient control of a quadrotor using a deep neural network. In *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 43–50, 2019. doi: 10.1109/IROS40897.2019.8968236.

[70] Patrick M Wensing, Michael Posa, Yue Hu, Adrien Escande, Nicolas Mansard, and Andrea Del Prete. Optimization-based control for dynamic legged robots. *arXiv preprint arXiv:2211.11644*, 2022.

[71] Lele Xi, Xinyi Wang, Lei Jiao, Shupeng Lai, Zhihong Peng, and Ben M Chen. Gto-mpc-based target chasing using a quadrotor in cluttered environments. *IEEE Transactions on Industrial Electronics*, 69(6):6026–6035, 2021.

[72] He Yin, Peter Seiler, Ming Jin, and Murat Arcak. Imitation learning with stability and safety guarantees. *IEEE Control Systems Letters*, 6(2):409–414, 2021.

[73] Zhengdong Zhang, Amr AbdulZahir Suleiman, Luca Carlone, Vivienne Sze, and Sertac Karaman. Visual-inertial odometry on chip: An algorithm-and-hardware co-design approach. 2017.