

Stochastic Optimization for Autonomous Navigation, Leveraging Parallel Computation

Joshua Spisak
CMU-RI-TR-23-56
August 2023



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

Thesis Committee:

Dr. Matthew Travers, *chair*
Dr. Michael Kaess
Chao Cao

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

Copyright © 2023 Joshua Spisak. All rights reserved.

Abstract

Stochastic Optimal Control (SOC) is a framework that allows disturbances and uncertainty in system models to be accounted for in its optimization framework. Despite accounting for this uncertainty, many first and second order methods for solving SOC problems are subject to local minima and are most appropriate for solving convex problems. Another class of SOC solvers, sample-based optimal controllers, are able to sample widely over the control space and thus see into various convex regions of the cost landscape, avoiding local minima. However, these methods suffer from issues of computational tractability and the curse of dimensionality. A common method to mitigate this issue is to use importance sampling to ensure the samples are focused in regions that most inform the solver. As such, much of the research in this area seeks to understand performance as a function of this sampling - and how to modify the sampling distribution to achieve better performance. To deepen our understanding of these methods we instead explore their computational aspects.

Specifically, we implement the Model Predictive Path Integral (MPPI), and Cross-Entropy-Method (CEM) algorithms in an extensible controls framework and benchmark them on a navigation task. We then evaluate common navigation performance metrics as a function of the number of particles. We also evaluate how quickly the algorithms are able to run using sequential evaluation, a threadpool parallelization backend, and a GPU parallelization backend.

Additionally, we propose a framework for interoperable parallelization in C++. Allowing the same C++ code to be parallelized on a CPU or on an NVIDIA GPU device with clean high-level interfaces and data structures C++ programmers are accustomed to. We discuss the utility of this framework in developing parallelized sampling-based algorithms in C++, and compare this work with another modern multiprocessing framework - thrust.

Acknowledgments

There are a few key people I would like to acknowledge. Firstly those on my committee- Matthew Travers, Michael Kaess, and Chao Cao, who supported me tremendously through all of this work.

Secondly I would like to thank my parents Joseph and Lisa Spisak, without whose support I would never have reached this point. I thank God daily for you.

I believe I owe a debt of gratitude to the team that I first joined at the Robotics Institute: Red Whittaker, Chuck Whittaker, James Teza, Siri O'Malley, David Kohanbash, Heather Jones, Andrew Zhang, Jordan Ford, Nikhil Jog, Ralph Boirum, and others. From the very first day you saw enough worth in me to keep me around and invest in my learning. I am forever grateful, especially to Red. Rest in peace Jim, I strive to be as capable in my field as you were in yours.

To the RACER crew, no words can describe my feelings. Ryan Darnley, Steve Willits, we've been through it and made it out the other end.

Thanks to the friends who helped give ideas, review sections, and improve this work and this document- Adam Johnson, Bhaskar Vundurthy, Charles Noren, I owe much of my sanity to you. To the friends who supported me emotionally through this period, thank you. To mention a few- Nayana Suvarna, we've been friends for so long, it's hard to imagine who I'd be without your support. Sam Speer. Ananya Rao. Ian Higgins. Adam Johnson. Jay Maier. Emma Benjaminson. Nate Sho-Tre. Katie Ford. Sydney Maier. There are too many to list but my deep appreciation to you all.

To whoever might read this, without you these are just words on a page. Cheers.

Contents

1	Introduction	1
2	Stochastic Optimization	5
2.1	Stochastic Optimal Control	5
2.1.1	Forms of Model Uncertainty	6
2.1.2	Methods to Solve SOC Problems	8
2.2	Cross-Entropy-Method (CEM)	10
2.3	Model-Predictive Path-Integral (MPPI)	10
2.3.1	Tuning MPPI	10
2.4	Importance Sampling And Disturbances	12
3	Multi-Processing and Parallel Programming	23
3.1	Moore’s Law is Dead	23
3.2	GPU Computation	24
3.2.1	Memory Model	26
3.2.2	NVCC and Code Compilation	29
3.2.3	The Future of Interopable (Heterogeneous) Computing	30
3.3	Shared-Memory Model	31
3.4	JUMP	32
3.4.1	Omissions	35
4	Controls Methodology	37
4.1	Kinematic Bicycle Model	38
4.2	Cost Functions	41
4.2.1	Traversal Cost	41
4.2.2	Waypoint Tracking	44
4.3	Costmap Generation	46
4.4	Simulation	46
4.4.1	Costmap Simulator	48
4.4.2	Model Simulator	49
4.5	Navigation as Optimal Control	49
4.6	Navigation as Benchmark	49
4.6.1	Literature Review	51

5	Benchmarking Analysis	53
5.1	Benchmark Execution Rate Results	53
5.2	Navigation Performance Results	58
5.3	Future Work	59
6	Conclusions	63
A	Disturbance Experiments	65
B	Temperature Sweep Experiments	69
C	How Stochastic is Stochastic?	73
	Bibliography	79

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

List of Figures

2.1	Open-loop rollouts for a ground vehicle modelling using a kinematic bicycle model, starting from an initial state $x_0 = (0, 0, 0, 8, 0)$ (moving straight at $8m/s$), we plot the system state every $0.5s$ with a zero control signal (continue moving straight with no change in speed), this produces the blue path. We then sample an additive control noise from $\epsilon = \mathcal{N}([00], [0.050.2])$, this produces deviations from the trajectory that are plotted in red. The full trajectories over the 5 second horizon are plotted on the left, the first second of the trajectories is focused on in the right. As time increases the noise causes red rollouts to deviate further and further from the blue rollout without noise.	7
2.2	Open-loop rollouts for a ground vehicle modelling using a kinematic bicycle model, starting from an initial state $x_0 = (0, 0, 0, 8, 0)$ (moving straight at $8m/s$), we plot the system state every $0.5s$ with a zero control signal (continue moving straight with no change in speed), this produces the blue path. We then sample an additive control noise from $\epsilon = \mathcal{N}([00], [0.05 \ 00 \ 0.2])$ as well as an additive state noise $w = \mathcal{N}(0, 0.01I)$ this produces deviations from the trajectory that are plotted in red. The full trajectories over the 5 second horizon are plotted on the left, the first second of the trajectories is focused on in the right. As time increases the noise causes red rollouts to deviate further and further from the blue rollout without noise.	8
2.3	How the temperature (λ) parameter affects sample weighting. As the λ parameter decreases, the weighting on non-zero costs converges to zero. As the λ parameter increases the weighting for non-zero costs	11
2.4	How the temperature (λ) parameter affects the MPPI solution on a navigation task. In the plots above, MPPI has converged to a smooth trajectory navigating around a lethal obstacle directly ahead of it. The seed (blue) is directly above the best rollout. In the upper plots ($\lambda = 0.01, 1.0$), the optimized result from MPPI (green) is very closely overlaid on the blue rollout as well. In the lower plots ($\lambda = 5.0, 20.0$), the optimized result from MPPI (green) begins moving further away from the obstacle, as a result of the changes in weighting.	16

2.5	How the temperature (λ) parameter effects how far the MPPI solution is from lethal obstacles. As the temperature (λ) parameter increases, MPPI gives a higher buffer to the lethal obstacle (evaluated by taking the center of the lethal obstacle at (25,0) and computing the closest point on the optimized trajectory to it, for a given temperature parameter value).	17
2.6	The affect of a model disturbance on MPPI importance sampling. This shows a mobile platform at (0,0 attempting to navigate to a waypoint at (100,0), it has 100 samples which are plotted above, colored by weighting to the final control update (light rollouts are high weight, low cost; dark rollouts are low weight, high cost). The blue rollout is the seed given to MPPI, the green is the optimized solution from MPPI. In the above image at t=0 we can see that the vehicle has primarily light rollouts indicating most of the rollouts do not intersect with the black lethal obstacle, and the optimized result is very close to the seed driving straight. In the below image we see that there has been a disturbance to the vehicle and it has drifted 2m to the right placing the majority of the rollouts in intersection with the black lethal obstacle. The optimized result now shoots to the left of the obstacle far away from the seed.	18
2.7	The affect of a model disturbance on MPPI importance sampling, after several iterations of MPPI. After the initial model disturbance, at the same timestep nine iterations of MPPI are called successively to refine the solution and adjust the importance sampling. The Optimized result now smoothly avoids the obstacle and continues straight. . . .	19
2.8	The affect of an environmental disturbance on MPPI importance sampling. This shows a mobile platform at (0,0 attempting to navigate to a waypoint at (100,0), it has 100 samples which are plotted above, colored by weighting to the final control update (light rollouts are high weight, low cost; dark rollouts are low weight, high cost). The blue rollout is the seed given to MPPI, the green is the optimized solution from MPPI. In the above image at t=0 we can see that the vehicle has primarily light rollouts indicating there are no lethal obstacles observed by MPPI, and the optimized result is very close to the seed driving straight. In the below image we see that there has been a major environmental disturbance with a large lethal obstacle appearing several meters in front of the vehicle. This makes the majority of the rollouts in intersect with the black lethal obstacle. The optimized result now shoots to the left of the obstacle far away from the seed. . .	20

2.9	The affect of an environmental disturbance on MPPI importance sampling, after several iterations of MPPI. After the initial environmental disturbance, at the same timestep nine iterations of MPPI are called successively to refine the solution and adjust the importance sampling. The Optimized result now smoothly avoids the obstacle and continues straight.	21
3.1	All desktop, client, server, and tablet processor released by Intel over the last 13 years, plotted by speed and time. Each point shows a core's speed (GHz) over it's release date. We fit a line to all of these points to show the overall trend. We also fit a line to the maximum speed available each year. This shows that the common trend is for the maximum available processor speed to double roughly every 11.2 years.	25
3.2	All desktop, client, server, and tablet processor released by Intel over the last 13 years, plotted by core count and time. Each point shows the processors number of cores over it's release date. We fit a line to all of these points to show the overall trend. We also fit a line to the maximum number of cores available each year. This shows that the common trend is for the maximum available number of cores to double roughly every 0.8 years.	26
3.3	NVIDIA Memory Model for a GPU in a common desktop computer. GPU memory can only be accessed by the GPU. Conversely CPU memory can only be accessed by the GPU. Memory copys (also referred to as transfers) can be made between host and device memory.	27
3.4	NVIDIA Unified Memory Model for a GPU in a common desktop computer. GPU memory can only be accessed by the GPU. Conversely CPU memory can only be accessed by the GPU. When allocated in unified memory, the same memory addresses are valid for both CPU and GPU memory, and the nvidia driver will automatically sync data as necessary.	28
3.5	NVIDIA Memory Model for a Jetson device (we use the Orin). The CPU and GPU share the same physical memory, however, separate allocations can still be made requiring memory copies to be performed by the application.	28
3.6	NVIDIA Unified Memory Model for a Jetson device (we use the Orin). The CPU and GPU share the same physical memory. Since a unified memory call is made and it is the same physical memory, there are no memory copies that need to be performed.	29

3.7	A map of what function calls are valid from which code given host and device tags on various functions. Since global functions can be called from device code we consider it interoperable, however it means something different than a normal function call.	30
3.8	The normal shared memory model. The main program dispatches some number of threads to perform operations in parallel. Each of these threads is performing operations from and on the same memory pool	32
3.9	A target aware shared memory model. Before dispatching the main program must perform memory copies if dispatching to a GPU, and must also copy data back from the GPU after computation. Otherwise, it is the same as the normal shared memory model because all threads still read and write to the same memory.	33
3.10	Comparison of the ease-of-use of jump and thrust to parallelize a functor over an array using different parallelization backends. We consider the yellow code examples to be less convenient or portable. Specifically, to use a parallelization backend different than sequential evaluation on the GPU or parallel evaluation on the GPU, the code must be recompiled in the case of using thrust with threadpool based parallelization. Additionally, to use thrust with GPU based parallelization instead of sequential CPU evaluation, a specific data structure must be used. This means that thrust code is inherently specific to host or device evaluation. Whereas in all of these cases JUMP requires only a small parameter switch that can be done at run-time.	34
3.11	Comparison of the ease-of-use of jump and thrust to parallelize a functor over an array using different parallelization backends. We consider the yellow code examples to be less convenient or portable. JUMP is easily able to handle this case, again reducing to a single call and parameter change to switch backends. Where-as thrust has additional calls to facilitate the index processing, requires re-compilation for threadpool based parallelization, and is unable to parallelize this example using the GPU.	35
3.12	The calling structure of nested data types being parallelized over. The foreach call triggers a recursive to_device() call that transfers each child member to device memory as necessary.	36
4.1	How a kinematic bicycle model can be used to represent the movement of a four-wheeled vehicle.	39

4.2	The kinematic bicycle model’s motion through space. The bicycle has a primary reference point at the center of the rear wheel. It is assumed that wheel is driving the vehicle forward along at some velocity V . It is rotated θ relative to the x-axis. The bicycle has a wheel-base with length L . Given some steering angle δ of the front wheel, it is rotating around some point which is R distance from the rear wheel, with an instantaneous center of rotation at the intersection of the lines orthogonal to the wheels of the bicycle.	39
4.3	An example of how obstacles in an environment can be converted to a costmap parameterized over x-y space. In the camera image, there are two clusters of boulders, in the costmap on the right the black areas correspond to lethal areas in the costmap that the vehicle must avoid. The golden start to the upper left of the costmap is the waypoint the vehicle might be attempting to reach as part of the navigation task.	42
4.4	The vehicle footprint that encompasses the entire base of the vehicle is discretized to sample some number of points over the costmap. It is assumed that at minimum the four corners of the vehicle will be sampled, and some number of longitudinal and lateral samples are added to produce a total of $(n_{\text{lateral samples}} + 2) * (n_{\text{longitudinal samples}} + 2)$ samples over the footprint of the vehicle. These samples are distributed evenly over the footprint of the vehicle.	43
4.5	The vehicle footprint is shown over a costmap, which is then converted to discrete points that can be sampled over the costmap.	44
4.6	A plot of the bound cost function, $y = v_{\text{max}} \frac{\lambda x}{\lambda x + 1}$, with $v_{\text{max}} = 5$, $\lambda = 0.1$	45
4.7	Costmap simulation of sensor range and line-of-sight, to the left is how the costmap simulator is processing lethal cells as either out-of-range, in-range but obscured, or visible. To the right is the final costmap that gets passed to the optimizer to perform the navigation task.	48
5.1	MPPI Timing Profile using GPU parallelization on Computer A in table 4.5. This computer has an Intel i9-13900K with 32 threads and an NVIDIA RTX 4090. While a 50Hz real-time rate is only maintained using GPU parallelization, with 8 threads or more the solver maintains 10Hz easily with 500 samples.	54
5.2	CEM Timing Profile using GPU parallelization on Computer A in table 4.5. This computer has an Intel i9-13900K with 32 threads and an NVIDIA RTX 4090. While a 50Hz real-time rate is only maintained using GPU parallelization, with 8 threads or more the solver maintains 10Hz easily with 500 samples. There are interesting spikes in the solve times below 500 samples.	54

5.3	MPPI Timing Profile using GPU parallelization on Computer B in table 4.5. This computer has an Intel i7-8700K with 12 threads and an NVIDIA RTX 2070. We see that with GPU parallelization, MPPI runs just below 50Hz at 3000 particles, however at 2500 samples and less it is able to maintain 50Hz real-time performance. With approximately 400 samples, it is able to achieve 10Hz with 8 or 12 threads.	55
5.4	CEM Timing Profile using GPU parallelization on Computer B in table 4.5. This computer has an Intel i7-8700K with 12 threads and an NVIDIA RTX 2070. This algorithm is able to run at 50Hz up to 3000 particles, however with threadpool parallelization it struggles to achieve 10Hz at sample counts greater than 200. We see another interesting non-linear relationship between sample count and execution rate at less than 500 particles.	55
5.5	MPPI Timing Profile using GPU parallelization on Computer C in table 4.5. This computer has an AMD Ryzen 7 3700X with 16 threads and an NVIDIA RTX 2070 Super. We see that with GPU parallelization, MPPI runs just below 50Hz at 3000 particles, however at 2500 samples and less it is able to maintain 50Hz real-time performance. With approximately 500 samples, it is able to achieve 10Hz with 8 or 12 threads.	56
5.6	CEM Timing Profile using GPU parallelization on Computer C in table 4.5. This computer has an AMD Ryzen 7 3700X with 16 threads and an NVIDIA RTX 2070 Super. This algorithm is able to run at 50Hz up to 3000 particles. While we observe a non-linear relationship between sample count and execution rate below 500 particles, it seems to meet 10Hz for 500 particles and less.	56
5.7	MPPI Timing Profile using GPU parallelization on Computer D in table 4.5. This computer has a 12 core arm CPU, and a 2048 core NVIDIA GPU. It is unable to maintain 10Hz at even 500 particles on a GPU parallelization backend.	57
5.8	CEM Timing Profile using GPU parallelization on Computer D in table 4.5. This computer has a 12 core arm CPU, and a 2048 core nvidia GPU. It is unable to maintain 10Hz at even 500 particles on a GPU parallelization backend.	57
5.9	Metrics for course 0 in table 4.4. While CEM performs better for low particle counts on distance taken to reach goal metrics, generally MPPI takes a shorter amount of time to reach the goal.	58
5.10	Metrics for course 2 in table 4.4. We see that CEM takes a shorter path to the goal, however MPPI maintains higher speed and reaches the goal in less time.	59

5.11	Metrics for course 3 in table 4.4. We see that while CEM occasionally takes a shorter path to the goal, MPPI reliably reaches the goal in less time.	59
5.12	Metrics averaged over all courses in table 4.4. There is some minor variation in distance taken to goal at high and low sample counts. On average MPPI reliably finds a faster way to reach the goal.	60
5.13	Navigation of a four obstacle course with 100 samples and MPPI random number generator seed set to 1 (top left), 200 (top right), 500 (bottom left), 800 (bottom right). The paths take different homotopy classes through the obstacles in each case.	61
5.14	Navigation of a four obstacle course with 3000 samples and MPPI random number generator seed set to 1 (top left), 200 (top right), 500 (bottom left), 800 (bottom right). The same path through the obstacle is taken with seeds set to 1 and 500, however the paths are still noticeably different. Every other test takes a different path.	62

List of Tables

4.1	State Variables, their state vector notation, and a short description of these variables.	40
4.2	Control Variables, their control vector notation, and a short description of the variables.	40
4.3	Course configurations and costmaps generated randomly from them. .	47
4.4	Benchmark courses and example solutions.	50
4.5	Compute Platforms we benchmark on.	51

Chapter 1

Introduction

In this thesis we aim to discuss the applications of multi-processing to the field of stochastic optimal control. Specifically, we find that most modern frameworks that provide run-time flexibility are designed for python and thus restrict researchers to using python for their algorithms. To alleviate this, we implement a multi-processing library enabling novel workflows for C++ developers utilizing an interoperable computation paradigm. We then apply that library to the implementation of an optimal control framework which we benchmark on an uncrewed ground vehicle (UGV) navigation task. Finally, we discuss the future of computation within the robotics and research domain as we observe trends in processor design, software design, and active frontiers in optimal control and differentiable algorithms research.

Parallel Processing with either large-scale multi-threading or graphics processing units (GPUs) are a resource that can accelerate algorithms by orders of magnitude. However, the complexity of the hardware and software frameworks to use these resources often makes it difficult for researchers in domains not focused on computer science and software development to utilize them. One key domain in which parallelization is leveraged heavily is machine learning, and many libraries like PyTorch and JAX provide easy-to-use interfaces for acceleration and differentiation. A key feature of these libraries is the ability to parallelize with different computing backends without significant code restructure. The same neural net can be accelerated using multi-threading or GPU parallelization with a small change in parameters and no extra burden on the developer. In C++, some libraries exist enabling the same capa-

1. Introduction

bilities, however they are often limited by difficult to use interfaces or backend-specific peccadillo's. The same generalized easy-to-use capabilities that exist in libraries in python do not exist in C++.

We consider this gap to be caused by the nuances of the C++ language and limitations on run-time flexibility. While there are certain limitations that cannot be remedied, we argue that some of this gap can be bridged with creative usage of modern language features and compiler tooling, allowing C++ developers to obtain some of the same flexibility at run-time as python users while retaining the benefits of static analysis, code efficiency, industry standards compliance, and compatibility with large legacy code-bases that many organizations require.

To bridge this gap we propose the Just Multi-Processing (JUMP) C++ library which implements modern interoperable data structures and an easy to use shared-memory model of computation with run-time selection of parallelization backends. We compare this framework with the modern multi-processing library thrust.

We also consider the applications of parallel processing to optimal control and robotic navigation, specifically in the fields of stochastic optimization and stochastic optimal control. As modern computers trend towards becoming increasingly parallel, many modern control algorithms seek to leverage this capability. Within the field of stochastic optimal control we consider two sampling based optimal control algorithms - Model Predictive Path Integral control and the Cross Entropy Method. Both algorithms rely on parallelization to achieve run-time performance. We explore the computational requirements of these algorithms as well as their ability to function with a reduced set of resources.

To achieve this we implement Just Controls (JCTL) a C++ library implementing a number of optimal control algorithms utilizing the jump library as a backend. We also implement a simulation framework for an uncrewed ground vehicle (UGV) which allows us to also benchmark the performance of this library on a navigation task through a randomly generated landscape. We discuss the applications of this library for real-time navigation on a UGV and the validity and applicability of the navigation task as a benchmark.

We believe the key contribution of this work to be in extending the heterogeneous and interoperable computing paradigms with the JUMP library. Considering computational trends, and works such as [24], [31], [20] and many other optimization and

planning techniques that leverage parallelization; we believe that parallelization will become increasingly importance. As such being able to implement these algorithms in a way that is efficient, easy, and extensible is critical. We believe this work could contribute heavily to this future.

While much of the rest of this work is in essence a re-implementation of existing methods using this framework, we still believe it to be of use as a reference for existing computational constraints and as a reference for algorithm comparison of CEM and MPPI. Additionally, we believe the stochastic controls framework implemented as part of this work will be helpful in testing new optimal control methods on the navigation task and creating new parallelized and stochastic optimal control schemes.

1. Introduction

Chapter 2

Stochastic Optimization

The field of stochastic optimization concerns the usage of randomization in the optimization process, or uncertainty in the optimization objective itself. Similarly, the field of stochastic optimal control concerns the control of systems with some form of model or observational uncertainty. There are stochastic optimization methods that can be used for stochastic optimal control, however there are also stochastic optimization methods that rely on randomization to solve deterministic control problems.

2.1 Stochastic Optimal Control

Stochastic optimal control concerns many models and types of uncertainty. Let us consider the optimal control problem phrased in 2.1. The objective is to select some control sequence $U^* = (u_0, \dots, u_T)$ that minimizes an objective function c over a control horizon T . Since the control sequence is defined over the time horizon T , there is also a terminating cost c_f that is a function of the final state. This is subject to some model F . This definition encapsulates many control tasks, as c is generally defined as a function of state and control. Notably, this phrasing does not contain any hard constraints except for the box constraints on each control u in the control sequence U .

$$U^* = \operatorname{argmin}_U \left[c_f(x_{T+1}) + \sum_{t=0}^T c(x_t, u_t) \right] \quad (2.1)$$

s.t.

$$x_{t+1} = F(x_t, u_t) \quad (2.2)$$

$$u_{\min} \leq u \leq u_{\max} \forall u \in U = (u_0, \dots, u_T) \quad (2.3)$$

Supposing the model F is deterministic, this does not constitute a stochastic optimal control problem. However if F can be structured to contain some form of uncertainty or stochasticity it then constitutes stochastic optimal control.

2.1.1 Forms of Model Uncertainty

However, F can be structured to incorporate uncertainty such as in equation 2.4 which adds a Gaussian noise term w_t sampled from a normal distribution to some underlying model f .

$$x_{t+1} = F(x_t, u_t) = f(x_t, u_t) + w_t \quad (2.4)$$

$$w_t \sim \mathcal{N}(0, \Sigma) \quad (2.5)$$

Another example is in equation 2.6, which adds a Gaussian noise term ϵ_t to the control signal u_t .

$$x_{t+1} = F(x_t, u_t) = f(x_t, u_t + \epsilon_t) \quad (2.6)$$

$$\epsilon_t \sim \mathcal{N}(0, \Sigma) \quad (2.7)$$

To demonstrate the affect of this noise, let us consider a mobile platform that can be described using the model in 4.1. We can see in Figure 2.1 that additive noise to an open-loop control signal can cause significant deviation from the deterministic model.

Let us also consider the stochastic model in equation 2.8, which has both additive noise terms, one to the state and another to the control signal.

$$x_{t+1} = F(x_t, u_t) = f(x_t, u_t + \epsilon_t) + w_t \quad (2.8)$$

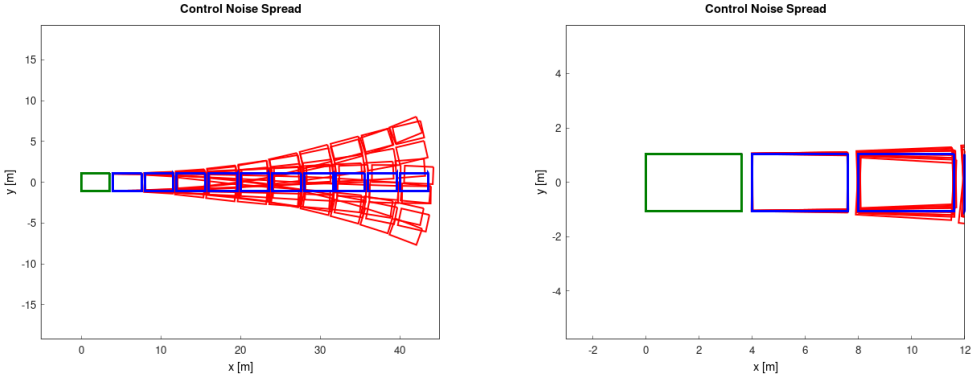


Figure 2.1: Open-loop rollouts for a ground vehicle modelling using a kinematic bicycle model, starting from an initial state $x_0 = (0, 0, 0, 8, 0)$ (moving straight at $8m/s$), we plot the system state every $0.5s$ with a zero control signal (continue moving straight with no change in speed), this produces the blue path. We then sample an additive control noise from $\epsilon = \mathcal{N}\left(\begin{bmatrix} 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0.05 \\ 0.2 \end{bmatrix}\right)$, this produces deviations from the trajectory that are plotted in red. The full trajectories over the 5 second horizon are plotted on the left, the first second of the trajectories is focused on in the right. As time increases the noise causes red rollouts to deviate further and further from the blue rollout without noise.

$$\epsilon_t \sim \mathcal{N}(0, \Sigma) \tag{2.9}$$

$$w_t \sim \mathcal{N}(0, \Sigma) \tag{2.10}$$

We can see the affect of this noise in figure 2.2, which appears to be very similar qualitatively to the control noise from figure 2.1, however the addition of the state additive noise produces larger deviations.

For a ground vehicle we consider all of the above noise profiles to be relevant. In the case of a control additive noise this could capture latency or error in actuation which always exists. The state additive noise is also of interest to capture large random disturbances like the ones mentioned above, which is especially of interested with a dynamically capable vehicle which may switch between gripping and slipping conditions or may experience large disturbances from changes in the terrain. Often the practical consideration of this noise is to provide more buffer for the vehicle around lethal areas in the state space to prevent disturbances from pushing the system into those areas. Another way to deal with these uncertainties is to tie the

2. Stochastic Optimization

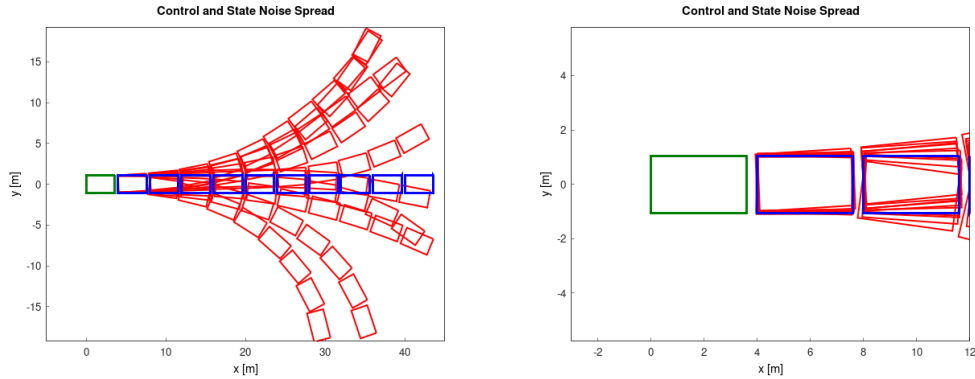


Figure 2.2: Open-loop rollouts for a ground vehicle modelling using a kinematic bicycle model, starting from an initial state $x_0 = (0, 0, 0, 8, 0)$ (moving straight at $8m/s$), we plot the system state every $0.5s$ with a zero control signal (continue moving straight with no change in speed), this produces the blue path. We then sample an additive control noise from $\epsilon = \mathcal{N}\left(\begin{bmatrix} 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0.05 & 0 \\ 0 & 0.2 \end{bmatrix}\right)$ as well as an additive state noise $w = \mathcal{N}(0, 0.01I)$ this produces deviations from the trajectory that are plotted in red. The full trajectories over the 5 second horizon are plotted on the left, the first second of the trajectories is focused on in the right. As time increases the noise causes red rollouts to deviate further and further from the blue rollout without noise.

noise to a particular component of the state - for example velocity. Such that by moving more slow the controller can be more certain and deliberate in the movement of the system in high-risk areas.

While all of these forms are of interest, the focus of this document will be on the system described by equation 2.6, since this is the form that is considered by the Model Predictive Path Integral (MPPI) control algorithm described in section 2.3. In addition to providing intuition for what practical considerations may correspond to this noise profile, figure 2.1 and provides intuition for how MPPI explores the state space, since it samples from a similar distribution to inform it's control update rule.

2.1.2 Methods to Solve SOC Problems

Within the field of stochastic optimal control there are two primary approaches for solving these problems. The first involves analytically solving the problem, often using a first or second order method that treats it as a convex problem, such as

[8][9][21]. Another common approach is referred to as scenario-based optimization, as referred to in [6][5][7], where-in some number of relevant scenarios or noise profiles are sampled and optimized over.

Methods like the Cross Entropy Method (CEM) and Information Theoretic MPC (IT-MPC, also known as MPPI), are often put into the category of scenario based optimization since they use a finite number of samples to represent the distribution formed by the uncertainty in the dynamics. However, the samples or ‘scenarios’ are utilized in a different way for these sampling-based methods. In scenario-based optimization schemes the scenarios are usually combined into a convex optimization problem that must consider the enumerated scenarios [7][6][5][18][17]. Methods like CEM and MPPI have no such final convex optimization problem, they can operate using only the zeroth order information provided by the samples.

Some variants of CEM and MPPI incorporate first order information into the optimization process, however the relationship between the samples and the first-order optimization is different than the relationship between the scenarios and the convex optimization in scenario-based optimization. For example, in CEM-GD a gradient descent optimizer refines the result from a CEM iteration. In Shield-MPPI, gradient descent is used to enforce a control barrier function on the result of MPPI. In both of these the first order information is used to refine or improve a result. If the naive result from CEM or MPPI respectively is of sufficient quality then the first-order optimizer is redundant, and the samples in and of themselves are sufficient to produce a result. Where-as in scenario-based optimization the convex optimization is what is used to optimize over the scenarios, without-which there is no solution. In CC-MPPI the authors describe scenario-based methods as those relying on ”randomization to solve optimization problems”, then note how Covariance-Controlled MPPI (CC-MPPI) does not fit within this or the analytical approach to stochastic optimal control. We therefore consider sample-based optimal controllers to be a unique family of methods within stochastic optimal control encompassing CEM, MPPI, CEM-GD, Shield-MPPI, CC-MPPI and all of it’s many variants.

The CEM and MPPI methods will be the primary focus of these documents, as we seek to understand the computational aspects of these algorithms, how to leverage parallelism to achieve run-time performance, and how many samples are required to achieve good performance on a navigation task. We describe the CEM and MPPI

algorithms, as well as some considerations when tuning and using them below.

2.2 Cross-Entropy-Method (CEM)

The cross-entropy-method (CEM) as introduced in [25] is an optimization method that allows iterative optimization of some objective function using some number of samples. We implement the cross-entropy-method using a gaussian distribution. This algorithm is detailed in 1. In our implementation we heavily parallelize the perturbations, rollouts, cost computations, and cost summations.

2.3 Model-Predictive Path-Integral (MPPI)

Model Predictive Path Integral (MPPI) is a very similar algorithm to CEM, however it does account for noise of the form of equation 2.6. Our implementation is mostly based on [31], however we utilize some of the numerical stability tricks from [32]. For our implementation see algorithm 2.

2.3.1 Tuning MPPI

We can observe from the control update rule in Algorithm 2 line 39, that each control perturbation put into a weighted average with the coefficient determined by the cost of that control sequence $weight = exp(-\frac{1}{\lambda}S)$, where S is the final cost for a given sampled control sequence (all costs have the smallest subtracted from them such that the lowest cost sample has $S = 0$). The λ parameter (often referred to as the temperature) can influence how leniently the algorithm weights higher-cost rollouts. To get some intuition for this we can plot an inverse exponential mapping for a few different λ values. As is shown in figure 2.3, as the lambda parameter approaches zero, the slope of the weighting appears sharper and the algorithm converges to weighting the lowest cost sample 1.0, and the rest 0.0.

With an extremely low temperature parameter MPPI can behave similarly to CEM, and simply return the lowest cost sample as the optimized result. We can also observe how this effects the result from MPPI on a navigation task in figure 2.4, as the temperature parameter increases, MPPI seems to give a larger buffer of space between

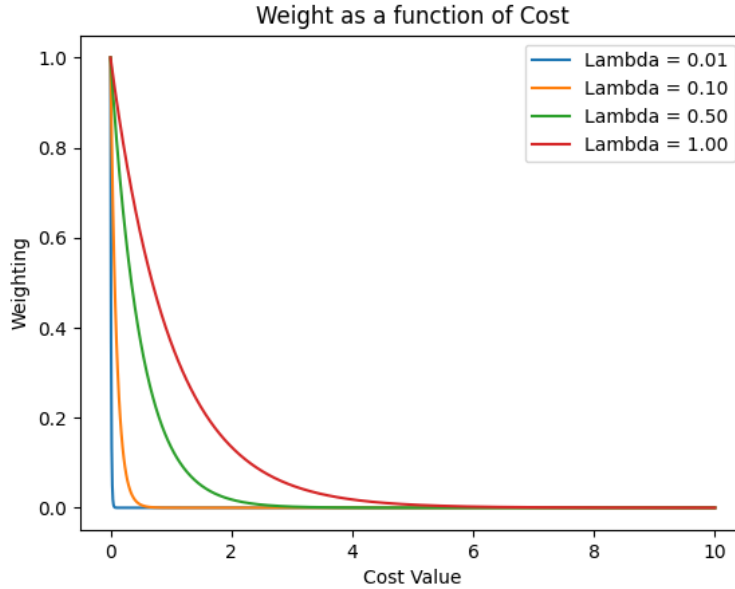


Figure 2.3: How the temperature (λ) parameter affects sample weighting. As the λ parameter decreases, the weighting on non-zero costs converges to zero. As the λ parameter increases the weighting for non-zero costs

the optimized trajectory and the lethal obstacle. This can be quantified in figure 2.5, which shows that tuning the temperature parameter can add up to $0.4m$ additional safety buffer. It is interesting that intuitively it would make sense for the noise to determine how much buffer must be provided to minimize the expectation of a lethal collision, however the temperature parameter seems to provide this functionality.

Often it is easy to intuit MPPI as an algorithm that performs similarly to CEM, taking some number of samples and returning the best result. Indeed, MPPI can be tuned to perform in that way. However, the importance sampling and averaging mechanisms are fundamentally a different operation. Consequently, we consider introspection of the costing, weighting, and sampling to be of the utmost importance when using and tuning this algorithm.

2.4 Importance Sampling And Disturbances

Many of the MPPI variants mentioned above are at a frontier of research that seeks to better handle large disturbances that can be better phrased as an additive disturbance to some model, similar to the form of equation 2.4. In [12] they assume a model that takes the form of equation 2.11, where ω_t is not Gaussian but is some unmodeled, but bounded disturbance.

$$x_{t+1} = f(x_t, u_t) + \omega_t \quad (2.11)$$

This disturbance could be many things, often it is assumed to be some environment interaction that causes a system to deviate from it's model. For example, an ATV running over some rocks that cause it to slide, or side-slip down a steep hill. For sampling-based algorithms the impact is that importance-sampling assumptions made for tractability break. Specifically, the assumption is that samples are focused on a region that is important to the task at hand, that they are near the optimum and are therefore providing information to the controller that allows it to make the best available choice in it's action.

This assumption often holds when the optimizers are being used in a model predictive control task, which would dictate that at each point in time the first action in an optimized sequence is being executed. At the next point in time the controlled system should then be roughly where the model would dictate it should. By stepping forward the control sequence and appending a random or zero value at the end, it should then be safe to assume that unless the cost landscape has drastically changed, that control sequence should be close to the optimum. However, when the system deviates from the state predicted by the model then the surrounding cost-landscape can also be drastically different. Consider an uncrewed ground vehicle (UGV) navigating through a boulder field. The boulders can each be considered lethal obstacles, and the task of the UGV is to navigate around these boulders to reach some goal on the other side.

We can directly observe the affect of large state disturbances on a task like this in figure 2.6. Since the vehicle has drifted to the right between timesteps 0 and 1, most of the samples that were focused around a path that takes the vehicle straight now intersect with the obstacle. With little information on what actions avoid the lethal

obstacle and take the platform towards the goal, the optimized result is a sharp turn to the left. However, we can see what a better converged importance sampling looks like in figure 2.7 which shows the importance sampling after MPPI has had several iterations to converge.

We can also consider the case of a large environmental disturbance and the impact it might have on importance sampling. In figure 2.8 we can see that a large lethal obstacle appearing in the environment can produce the same result as a model disturbance, invalidating importance sampling assumptions and producing a sub-optimal result from this sample based optimizer. We can also similarly observe that after several iterations of MPPI that the result has converged to a smooth avoiding trajectory that then proceeds straight to the goal.

This demonstrates that in some cases, if a solver is robust to large model disturbances it might be robust to large environmental disturbances as well, motivating that branch of research for mapless autonomy, where the stack is building an understanding of the environment as it moves through it and a large lethal obstacle might only be discovered and placed in the map once close.

Algorithm 1 The Cross-Entropy-Method.

```

1: procedure CEM( $x_0, U_{\text{init}}$ )
2:   Given some model  $f$  s.t.  $x_{t+1} = f(x_t, u_t)$ 
3:   Given some cost function  $c$  s.t.  $\text{scalar\_cost} = c(x_t, u_t)$ ,  $c_F$  s.t.  $\text{scalar\_cost} = c_F(x_t)$ 
4:   Given control limits  $u_{\text{max}}, u_{\text{min}}$ 
5:   Given params  $\text{max\_iters}, \text{num\_samples}, \text{num\_elite}, \Sigma_{\text{init}}$ 
6:    $U = U_{\text{init}}$ 
7:    $U_{\text{best}} = U_{\text{init}}$ 
8:    $\Sigma_u = (\Sigma_{\text{init}}, \dots, \Sigma_{\text{init}})$   $\triangleright$  Repeat the  $\Sigma_u$  sizeof( $U$ ) times.
9:    $\delta U = \text{Array of size}(\text{num\_particles}, \text{sizeof}(U))$   $\triangleright$  Controls for each sample
10:   $X = \text{Array of size}(\text{num\_particles}, \text{sizeof}(U) + 1)$   $\triangleright$  States for each sample
11:   $C = \text{Array of size}(\text{num\_particles}, \text{sizeof}(U) + 1)$   $\triangleright$  Cost for each sample / timestep
12:   $C_{\text{sums}} = \text{Array of size}(\text{num\_particles})$   $\triangleright$  Cost for each sample
13:  for  $t = 1$  to  $\text{max\_iters}$  do
14:     $\delta U = \text{SamplePerturbations}(U, \Sigma_u)$ 
15:    for  $k = 1$  to  $\text{num\_samples}$  do
16:      for  $t = 1$  to  $\text{sizeof}(U)$  do
17:         $\delta U^{k,t} = \text{clamp}(\delta U^{k,t}, u_{\text{min}}, u_{\text{max}})$ 
18:      end for
19:    end for
20:    for  $k = 1$  to  $\text{num\_samples}$  do
21:      for  $t = 1$  to  $\text{sizeof}(U)$  do
22:         $X^{k,t+1} = f(X^{k,t}, \delta U^t)$ 
23:      end for
24:    end for
25:    for  $k = 1$  to  $\text{num\_samples}$  do
26:      for  $t = 1$  to  $\text{sizeof}(U) + 1$  do
27:        if  $t == \text{sizeof}(U) + 1$  then
28:           $C^{k,t} = c_F(X^{k,t})$ 
29:        else
30:           $C^{k,t} = c(X^{k,t}, \delta U^t)$ 
31:        end if
32:      end for
33:       $C_{\text{sums}}^k = 0$ 
34:      for  $t = 1$  to  $\text{sizeof}(U) + 1$  do
35:         $C_{\text{sums}}^k += C^{k,t}$ 
36:      end for
37:    end for
38:     $a = ((C_{\text{sums}}^0, 0), \dots, (C_{\text{sums}}^k, k))$   $\triangleright$  Associate each rollout with its cost
39:     $\text{Sort}(a)$   $\triangleright$  Sort by cost.
40:     $U = \text{mean}(\delta U^{a(0,1)}, \dots, \delta U^{a(\text{num\_elite}, 1)})$   $\triangleright$  Compute mean of top samples.
41:     $\Sigma_u = \text{variance}(\delta U^{a(0,1)}, \dots, \delta U^{a(\text{num\_elite}, 1)})$   $\triangleright$  Same for variance.
42:     $U_{\text{best}} = \delta U^{a(0,1)}$ 
43:  end for
44:  return  $U_{\text{best}}$ 
45: end procedure

```

Algorithm 2 The Model-Predictive-Path-Integral Algorithm.

```

1: procedure CEM( $x_0, U_{\text{init}}$ )
2:   Given some model  $f$  s.t.  $x_{t+1} = f(x_t, u_t)$ 
3:   Given some cost function  $c(x_t, u_t), c_F(x_t)$ 
4:   Given params num_samples, num_elite,  $\Sigma, \lambda, u_{\text{max}}, u_{\text{min}}$ 
5:    $U_{\text{new}} = U_{\text{init}}$ 
6:    $\delta U = \text{Array of size}(\text{num\_particles}, \text{sizeof}(U))$   $\triangleright$  Controls for each sample
7:    $X = \text{Array of size}(\text{num\_particles}, \text{sizeof}(U) + 1)$   $\triangleright$  States for each sample
8:    $C = \text{Array of size}(\text{num\_particles}, \text{sizeof}(U) + 1)$   $\triangleright$  Costs per sample/timestep
9:    $C_{\text{sums}} = \text{Array of size}(\text{num\_particles})$   $\triangleright$  Cost for each sample
10:   $\delta U = \text{SamplePerturbations}(U, \Sigma)$ 
11:  for  $k = 1$  to num_samples do
12:    for  $t = 1$  to  $\text{sizeof}(U)$  do
13:       $\delta U^{k,t} = \text{clamp}(\delta U^{k,t}, u_{\text{min}}, u_{\text{max}})$ 
14:    end for
15:  end for
16:  for  $k = 1$  to num_samples do
17:    for  $t = 1$  to  $\text{sizeof}(U)$  do
18:       $X^{k,t+1} = f(X^{k,t}, \delta U^t)$ 
19:    end for
20:  end for
21:  for  $k = 1$  to num_samples do
22:    for  $t = 1$  to  $\text{sizeof}(U) + 1$  do
23:      if  $t == \text{sizeof}(U) + 1$  then
24:         $C^{k,t} = c_F(X^{k,t})$ 
25:      else
26:         $C^{k,t} = c(X^{k,t}, \delta U^t)$ 
27:      end if
28:    end for
29:     $C_{\text{sums}}^k = 0$ 
30:    for  $t = 1$  to  $\text{sizeof}(U) + 1$  do
31:       $C_{\text{sums}}^k += C^{k,t}$ 
32:    end for
33:  end for
34:   $C_{\text{min}} = \text{min}(C_{\text{sums}})$ 
35:  for  $t = 1$  to  $\text{sizeof}(U)$  do
36:     $E = 0$ 
37:    for  $k = 1$  to num_samples do
38:       $E+ = \exp(-(C_{\text{sums}}^k - C_{\text{min}})/\lambda)$ 
39:       $U_{\text{new}}^t += (\delta U_{\text{init}}^{k,t} - U^t) \exp(-(C_{\text{sums}}^k - C_{\text{min}})/\lambda)$ 
40:    end for
41:     $U_{\text{new}}^t / = E$ 
42:  end for
43:  return  $U_{\text{new}}$ 
44: end procedure

```

2. Stochastic Optimization

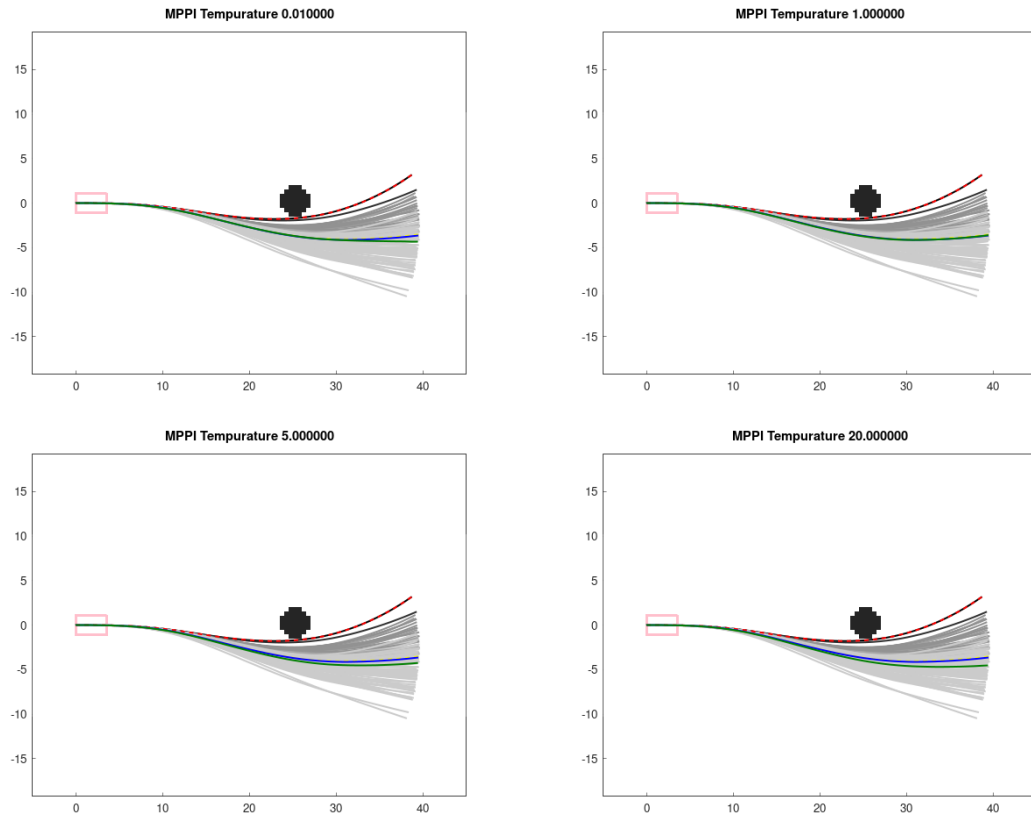


Figure 2.4: How the temperature (λ) parameter affects the MPPI solution on a navigation task. In the plots above, MPPI has converged to a smooth trajectory navigating around a lethal obstacle directly ahead of it. The seed (blue) is directly above the best rollout. In the upper plots ($\lambda = 0.01, 1.0$), the optimized result from MPPI (green) is very closely overlaid on the blue rollout as well. In the lower plots ($\lambda = 5.0, 20.0$), the optimized result from MPPI (green) begins moving further away from the obstacle, as a result of the changes in weighting.

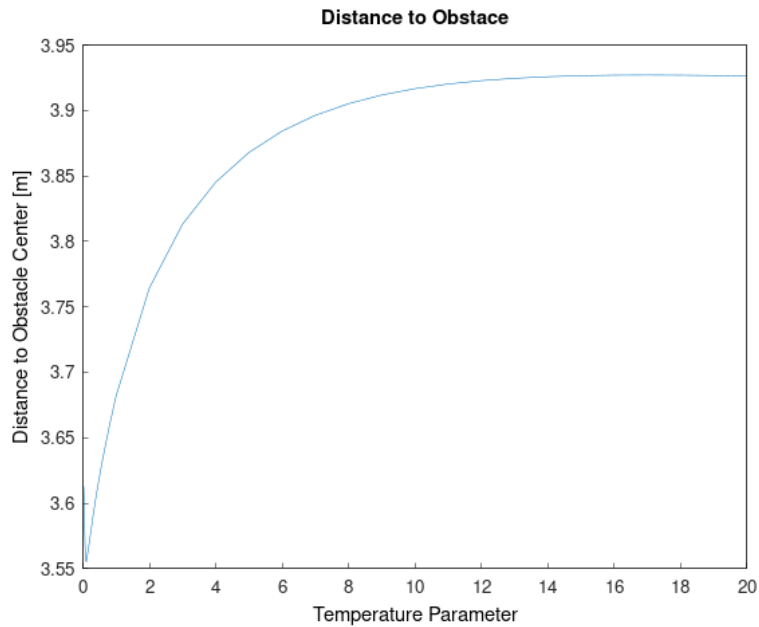


Figure 2.5: How the temperature (λ) parameter effects how far the MPPI solution is from lethal obstacles. As the temperature (λ) parameter increases, MPPI gives a higher buffer to the lethal obstacle (evaluated by taking the center of the lethal obstacle at $(25, 0)$ and computing the closest point on the optimized trajectory to it, for a given temperature parameter value).

2. Stochastic Optimization

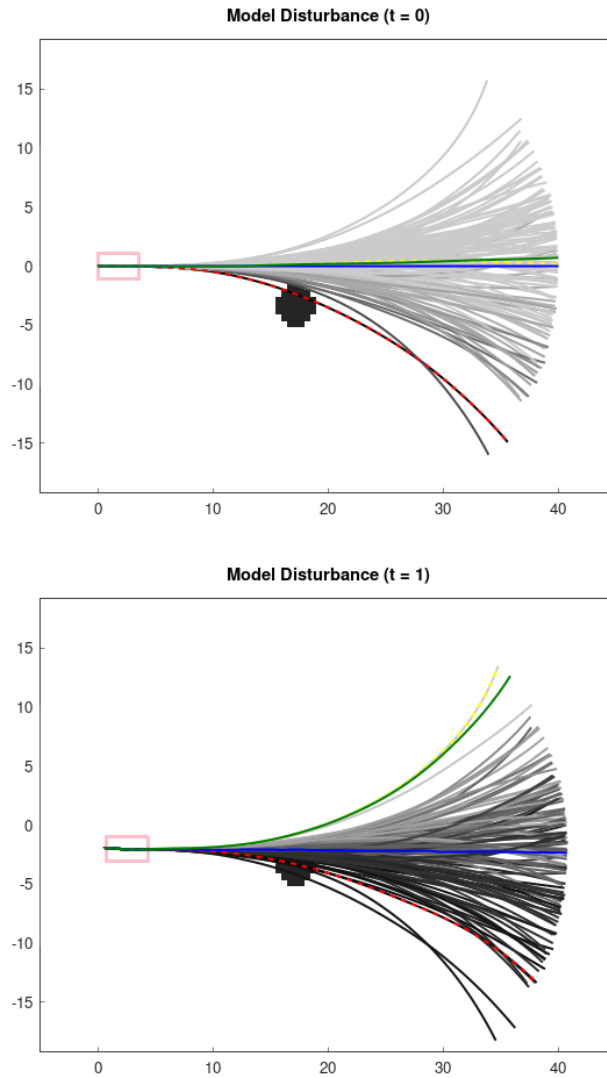


Figure 2.6: The affect of a model disturbance on MPPI importance sampling. This shows a mobile platform at $(0, 0)$ attempting to navigate to a waypoint at $(100, 0)$, it has 100 samples which are plotted above, colorized by weighting to the final control update (light rollouts are high weight, low cost; dark rollouts are low weight, high cost). The blue rollout is the seed given to MPPI, the green is the optimized solution from MPPI. In the above image at $t=0$ we can see that the vehicle has primarily light rollouts indicating most of the rollouts do not intersect with the black lethal obstacle, and the optimized result is very close to the seed driving straight. In the below image we see that there has been a disturbance to the vehicle and it has drifted $2m$ to the right placing the majority of the rollouts in intersection with the black lethal obstacle. The optimized result now shoots to the left of the obstacle far away from the seed.

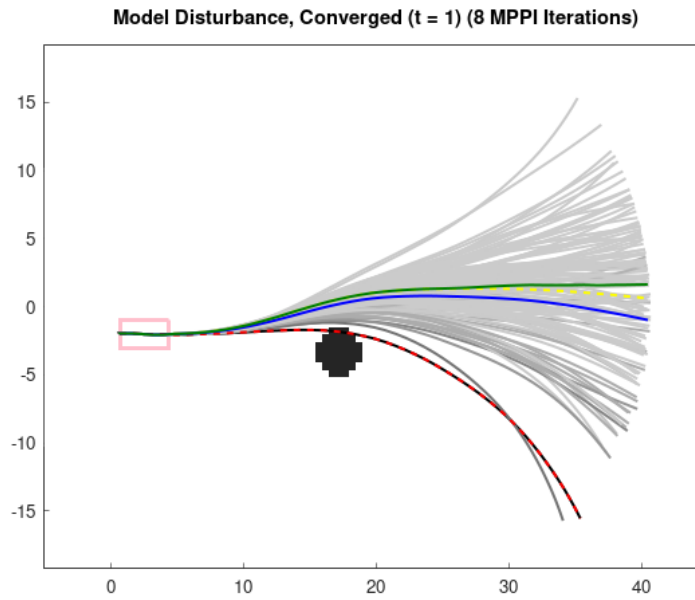


Figure 2.7: The affect of a model disturbance on MPPI importance sampling, after several iterations of MPPI. After the initial model disturbance, at the same timestep nine iterations of MPPI are called successively to refine the solution and adjust the importance sampling. The Optimized result now smoothly avoids the obstacle and continues straight.

2. Stochastic Optimization

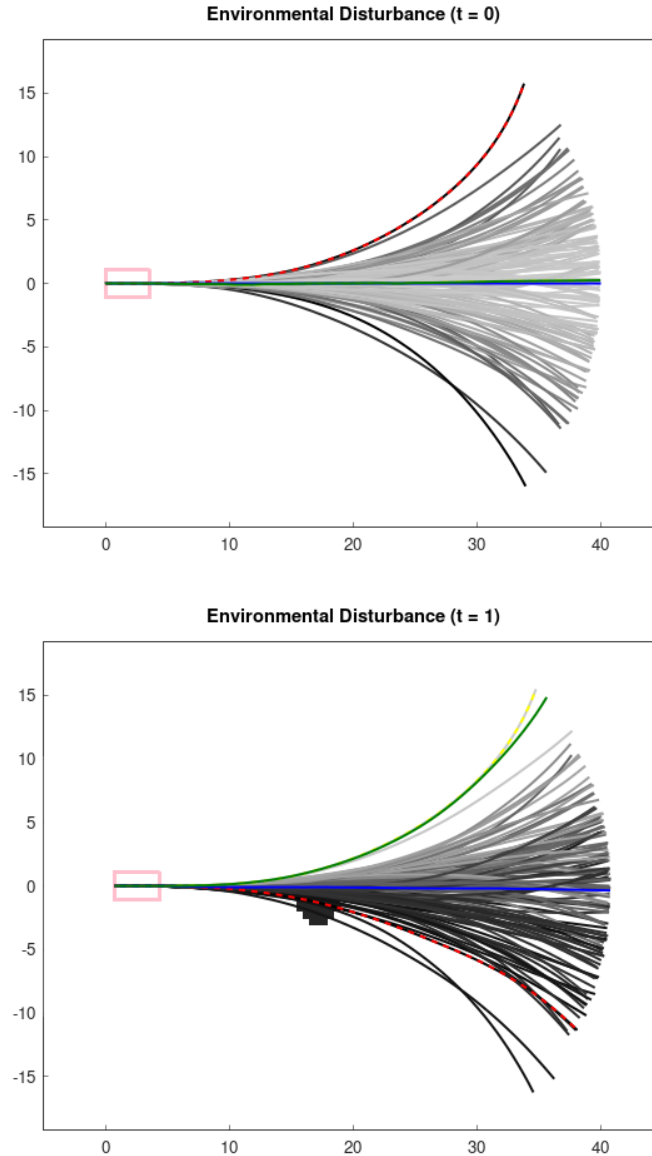


Figure 2.8: The affect of an environmental disturbance on MPPI importance sampling. This shows a mobile platform at $(0, 0)$ attempting to navigate to a waypoint at $(100, 0)$, it has 100 samples which are plotted above, colorized by weighting to the final control update (light rollouts are high weight, low cost; dark rollouts are low weight, high cost). The blue rollout is the seed given to MPPI, the green is the optimized solution from MPPI. In the above image at $t=0$ we can see that the vehicle has primarily light rollouts indicating there are no lethal obstacles observed by MPPI, and the optimized result is very close to the seed driving straight. In the below image we see that there has been a major environmental disturbance with a large lethal obstacle appearing several meters in front of the vehicle. This makes the majority of the rollouts intersect with the black lethal obstacle. The optimized result now shoots to the left of the obstacle far away from the seed.

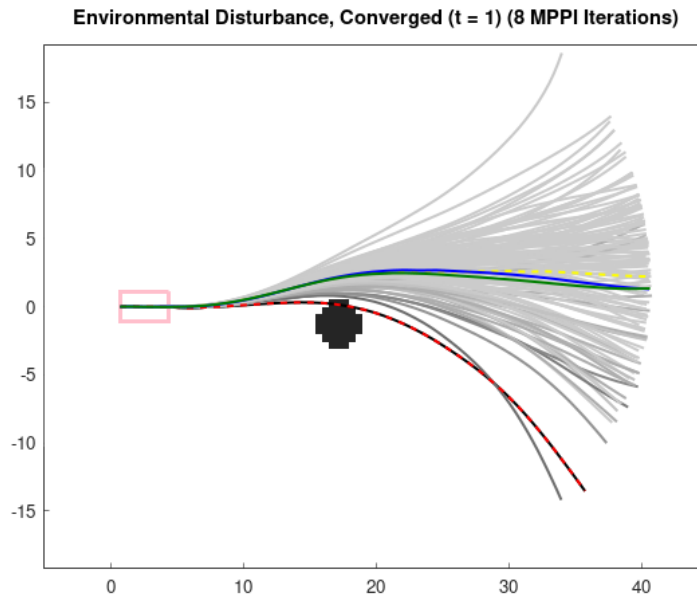


Figure 2.9: The affect of an environmental disturbance on MPPI importance sampling, after several iterations of MPPI. After the initial environmental disturbance, at the same timestep nine iterations of MPPI are called successively to refine the solution and adjust the importance sampling. The Optimized result now smoothly avoids the obstacle and continues straight.

2. Stochastic Optimization

Chapter 3

Multi-Processing and Parallel Programming

A common point of discussion in [31] is the reliance on a modern GPU to parallelize different components of the algorithm to make the algorithm itself tractable in real-time. This can make implementation of these algorithms itself challenging, as programming parallel processing often involves using target-specific languages and interfacing with low-level mechanisms for memory allocation and management. Many frameworks and libraries have been developed to make this more accessible to the everyday researcher and developer, with most libraries focusing on ease-of-development in python. In this section we discuss approaches to parallel programming, and introduce a library for parallel computation in C++ that improves the accessibility of these paradigms to developers who do not specialize in these domains.

3.1 Moore's Law is Dead

Moore's law is a "law" taken from an observation by a former CEO of Intel, Gordon Moore, that the number of transistors on an integrated circuit would double every two years. Along with this increase in transistor count came a drastic increase in core speed and capability[26]. For many years this "law" held, and developers were able to count on computer capability eventually catching up to the computational demands of their intensive algorithms. That is, an algorithm written assuming serial

computation (for example, a long chain of computation and math to compute an optimal control sequence such as differential dynamic programming), would continue to increase speed as the number of transistors increased and the speed of processors double every few years. However, since 2010 there has been a sharp decline in the rate at which processor speed increases, we can directly observe this change by looking at processor speeds in the last thirteen years. Figure 3.1 shows a plot of processor speeds for each desktop, client, server, and tablet core released by Intel over the last 13 years [1]. We can see that given these trends we can only expect processor speed to double every 11.2 years, looking only at the maximum processor speeds available each year.

However, we can also observe a trend in core count in Figure 3.2, showing core count for the same set of processors released by Intel in the last thirteen years, and we can observe that the maximum available core count is on a trend to double at least every year, with massive increases in core count over the last few years! This trend would indicate that while cores may not get much faster, we can expect a higher number of fast cores to continue to become available so long as the market demands it.

The natural conclusion to this data is that Moore's law is dead, however there are new horizons in software design utilizing parallel processing. From a software perspective this implies that there will be less emphasis on serial algorithms running fast, but rather on parallel algorithms running many operations at the same time. For some algorithms this can be as simple as turning a for loop into a parallel operation. For other applications to see any gains would require dealing with other hardware accelerators such as the massively parallel GPU's.

3.2 GPU Computation

Graphical Processing Units (GPU's) are familiar to many people even outside technical domains. As suggested by it's name, GPU's were first used to accelerate graphics processing with applications to general computing and video games, to improve render fidelity and increase frames per second. However, rather than performing computations faster, they work by performing more computations in parallel at the same time. This work will primarily focus on the NVIDIA GPU's architecture, with

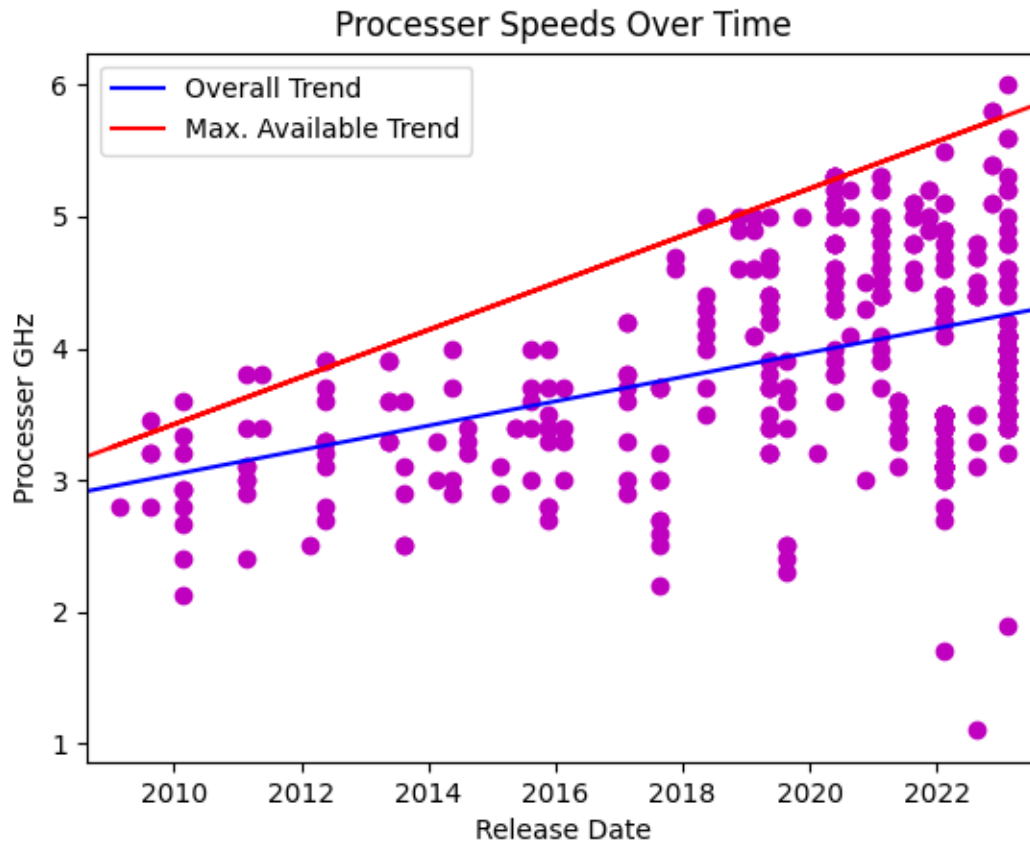


Figure 3.1: All desktop, client, server, and tablet processor released by Intel over the last 13 years, plotted by speed and time. Each point shows a core’s speed (GHz) over it’s release date. We fit a line to all of these points to show the overall trend. We also fit a line to the maximum speed available each year. This shows that the common trend is for the maximum available processor speed to double roughly every 11.2 years.

benchmarking performed on the computers listed later in table 4.5.

We provide an overview of the NVIDIA memory model and C++ syntax, however this should not be considered an extensive tutorial or introduction for a complete beginner. For more information we recommend referencing [14] and [13], or any of the excellent presentations, documentation, or resources provided by nvidia online.

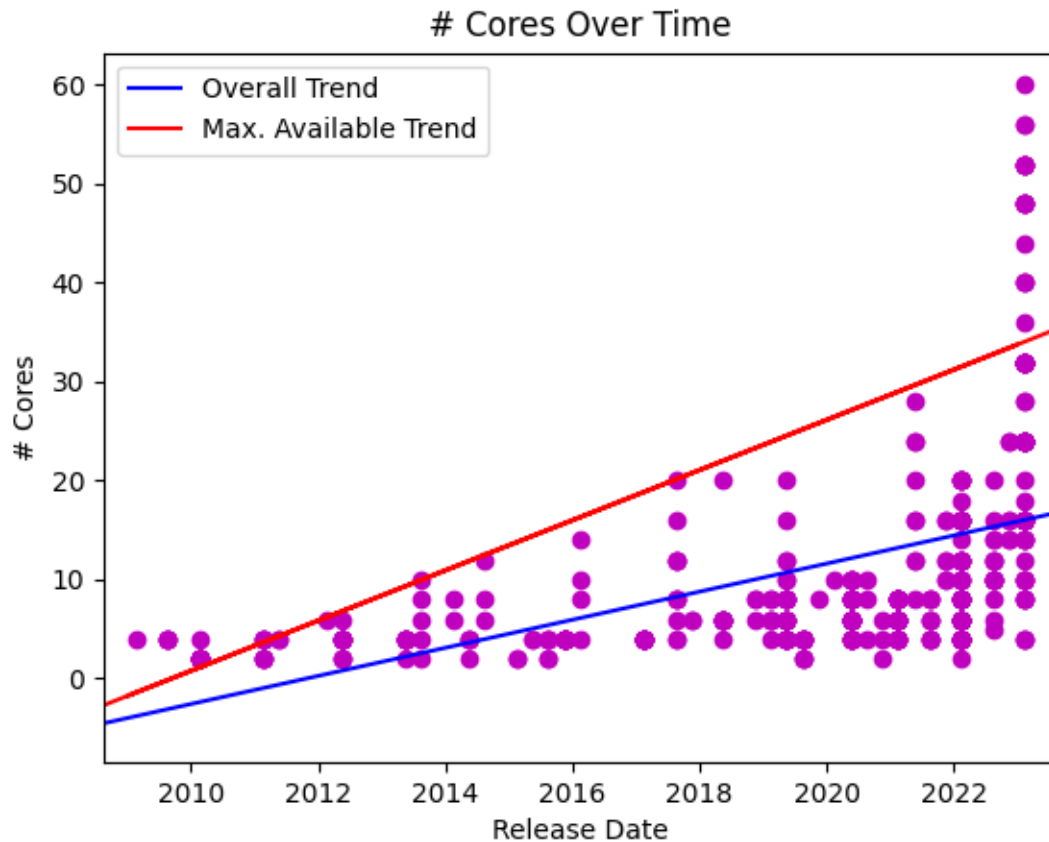


Figure 3.2: All desktop, client, server, and tablet processor released by Intel over the last 13 years, plotted by core count and time. Each point shows the processors number of cores over it's release date. We fit a line to all of these points to show the overall trend. We also fit a line to the maximum number of cores available each year. This shows that the common trend is for the maximum available number of cores to double roughly every 0.8 years.

3.2.1 Memory Model

NVIDIA GPU's are architected with various types of memory. GPU cores are put into groups called warps, and blocks, which have memory pools that are shared between them and can only be accessed between those particular threads. To conform to the shared-memory model we discuss below, we rely on global and unified memory. Normal global memory is allocated separately on the host (CPU) and the device

(GPU). Once allocated, code on the CPU can make memory copy calls between them. Often, this involves a memory copy to the device before performing GPU computation, then transferring the memory (modified by GPU computation) back to the host. This relation is illustrated in figure 3.3.

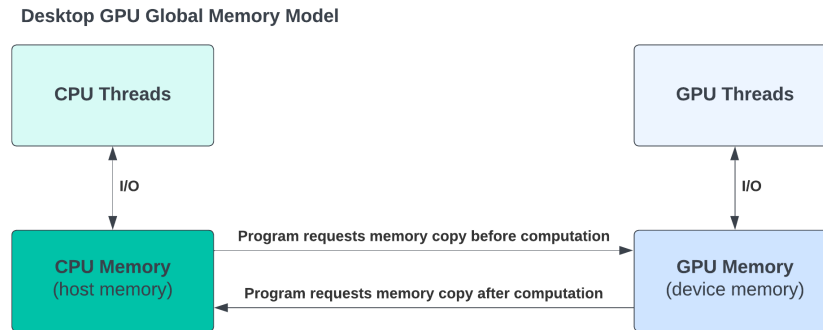


Figure 3.3: NVIDIA Memory Model for a GPU in a common desktop computer. GPU memory can only be accessed by the GPU. Conversely CPU memory can only be accessed by the CPU. Memory copies (also referred to as transfers) can be made between host and device memory.

There are a few other memory models and architectures that can be used while assuming a shared memory model of computation on a GPU. Since allocating and managing memory copies between device and host memory buffers can be cumbersome for a developer, and can take up valuable processing time waiting for copies to complete. NVIDIA designed an unified memory architecture, where a different allocation call is used which gives a memory address that can be used on both the host and device, with memory copies happening implicitly in the background handled by the NVIDIA driver. This relation is illustrated in figure 3.4.

Besides desktop computers, we also perform benchmarking on the NVIDIA Jetson Orin computer, this shares the same physical memory between the CPU and GPU. This allows us to leverage unified memory to our advantage, which gives us a pointer that can be used from both the host and device. Since it is the same physical memory there are no memory copies that need to be performed, as shown in figure 3.6. However, the normal global memory calls can be used which allocated separate memory buffers for both the host and device. Consequently, even though it is the same physical memory, memory copies of the data there-in still needs to be performed,

as shown in figure 3.5.

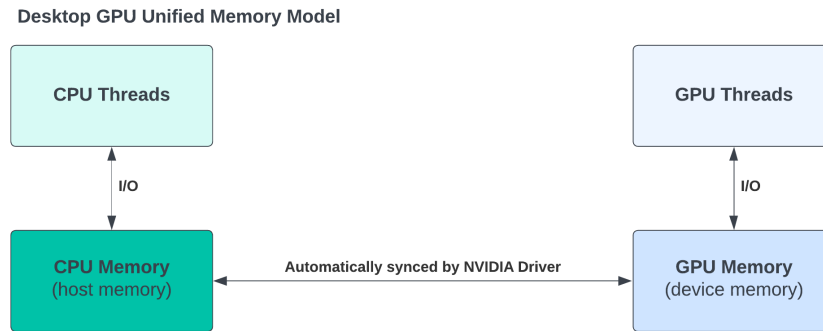


Figure 3.4: NVIDIA Unified Memory Model for a GPU in a common desktop computer. GPU memory can only be accessed by the GPU. Conversely CPU memory can only be accessed by the CPU. When allocated in unified memory, the same memory addresses are valid for both CPU and GPU memory, and the nvidia driver will automatically sync data as necessary.

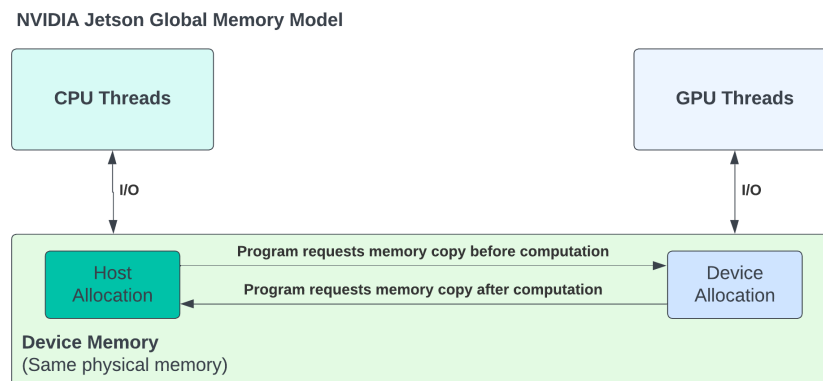


Figure 3.5: NVIDIA Memory Model for a Jetson device (we use the Orin). The CPU and GPU share the same physical memory, however, separate allocations can still be made requiring memory copies to be performed by the application.

Often, parallelization frameworks consider these memory models as different memory entities entirely. For example the thrust library [4] uses different classes to represent host, device, and unified memory. It's interface restricts kernels to be functions of the contained data type, and not the data containers as a whole. Additionally, the execution policy is tied to the container type - a device vector cannot be used in a threadpool and vice-versa.

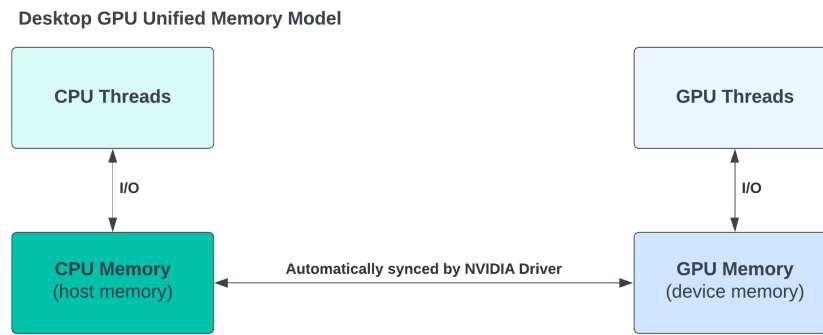


Figure 3.6: NVIDIA Unified Memory Model for a Jetson device (we use the Orin). The CPU and GPU share the same physical memory. Since a unified memory call is made and it is the same physical memory, there are no memory copies that need to be performed.

3.2.2 NVCC and Code Compilation

Many C++ developers are familiar with the `gcc`, `clang`, `mingw`, or other common C++ compilers. These compilers take in C++ code and emit binaries that are able to execute on a variety of target platforms, eg: `arm`, `x86`, etc. GPU's are a unique target platform that has it's own language. NVIDIA also has it's own syntax for calling GPU functions through C++. To support developers they have provided a compiler toolkit called `nvcc`, which is able to compile for the GPU target platform as well as handle the special calling syntax. Interestingly, the LLVM project has done the legwork to support compiling the same code using `clang`.

In addition to memory separation, there is also code separation where code has `__host__` and `__device__` tags that prefix functions. Once a `__global__` function has been called (these serve as the entry-point for GPU computation, and are often called kernels), functions that have the host tag (the default when not specified), cannot be called from these functions, or any device functions. Functions that have the device tag are compiled solely for use on the GPU and can be called from a global function, or any other device function. However, it is possible to provide both tags to a function so that it can be called from any function anywhere in code. We refer to this as being interoperable, since the host and device code are able to work together seamlessly. Figure 2.9 shows a map of what function calls are valid or invalid with different tags. Another library from NVIDIA, `libcudpp` [2], utilizes this mode of

3. Multi-Processing and Parallel Programming

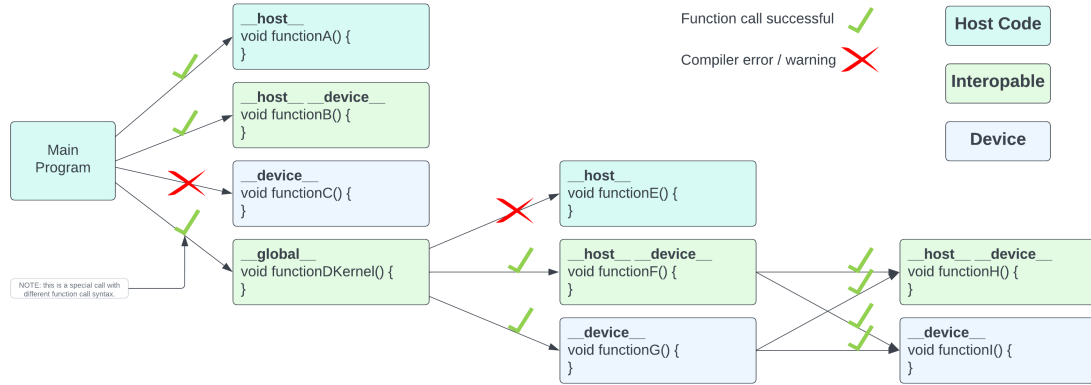


Figure 3.7: A map of what function calls are valid from which code given host and device tags on various functions. Since global functions can be called from device code we consider it interopable, however it means something different than a normal function call.

computation, referring to it as heterogeneous code. However, they do not provide vector and dynamic memory storage containers that are heterogenous.

3.2.3 The Future of Interopable (Heterogeneous) Computing

Often, the classes and objects that are shared between host and device are small: primitive data types, an atomic container, time primitives, complex numbers. These objects are units, and the kernels and code that utilize them are rarely deep (where deep here is in reference to the call stack, and levels of redirection through pointers). In some sense, this is driven by a desire to optimize code. GPU cores are many, but they are slower than CPU cores. Additionally, the memory available is often more limited in scope because it must be shared across so many cores. It is also likely this is derived from the cuda API itself which does not leverage object oriented programming or many modern C++ features (which is likely why there are so many higher-level abstractions wrapping the CUDA API).

However, since the nvcc compiler allows full use of C++ features (so long as only device functions are called from GPU code), developers are able to design more complex heterogeneous data types and make deeper and more expressive object oriented code.

Naturally, while it is advantageous from a developers perspective to use higher-level abstractions and modern language features and containers, this does not necessarily produce the most optimal code. Similarly, our reliance on global memory rather than using warp level memory operations for highly optimized code with produce easier to understand code with performance loss. We consider this loss worth the gains in readability, extensibility, and portability.

Generally, we believe it is the job of the compiler to consider platform specific nuances and optimize the code to perform the task at hand with the highest level of performance given the target platform. Of course, there are many hardware specific nuances to GPU programming, gremlins in memory handling, alignment, simple entrypoint calling, and much more. It is difficult for a programmer to design a language that abstracts all of these details in a way that doesn't require any syntax or calls from programmers, and then implement a compiler that is able to parse and compile that language for a variety of platforms. We do not consider this task to be trivial, however, we do consider it to be worthwhile.

Especially as we consider languages like python (frameworks like jax, pytorch, etc.) and julia that have advanced beyond C++ in run-time flexibility, JIT compiling, and in some cases even performance and optimality! Of course, we do not feel these languages or frameworks provide a panacea. We would assert that picking out the best features of all these languages, frameworks, and capabilities is achievable, it will just take some leg-work and effort.

3.3 Shared-Memory Model

The shared memory model assumes that each thread or parallel line of computation has full access to the same shared memory space. For sequential and threadpool based parallel evaluation this is implicit in a modern CPU's architecture, this architecture is show in figure 3.8. Using global or unified memory allows all threads performing parallel computation to read and write from the same memory, however additional steps must be added to copy data to and from device memory before and after computation. This target aware shared memory model is show in figure 3.9.

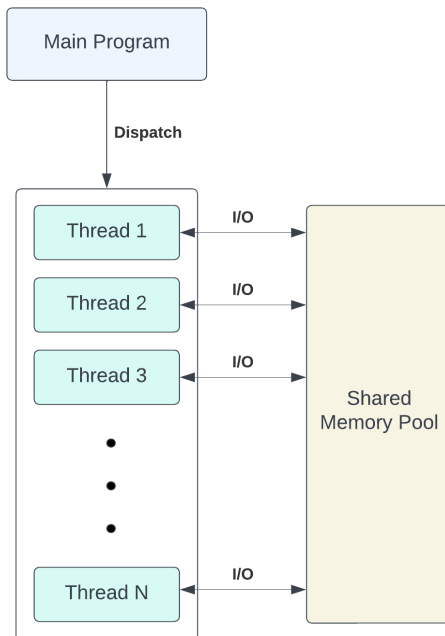


Figure 3.8: The normal shared memory model. The main program dispatches some number of threads to perform operations in parallel. Each of these threads is performing operations from and on the same memory pool

3.4 JUMP

To advance a vision of interoperable or heterogeneous computation, we implement a multiprocessing library in C++ called "Just Multi-Processing" or JUMP. This library implements interoperable random number generators, and interoperable memory buffer, an interoperable array (has some STL vector functionality), an interoperable multi-dimensional array, an interoperable shared pointer, and threadpool and GPU based parallelization.

We can demonstrate convenience of the JUMP library by comparing the same code to perform a foreach call on an array. We see in figure 3.10, JUMP is able to switch between backends with nothing more than a parameter switch. However, thrust requires specific data structures to be used for GPU based parallelization, additionally thrust requires compile-time flags to be set and a re-compile to use threadpool based parallelization. Consequently a binary built using thrust with a sequential or GPU parallelization backend CANNOT also use a threadpool based

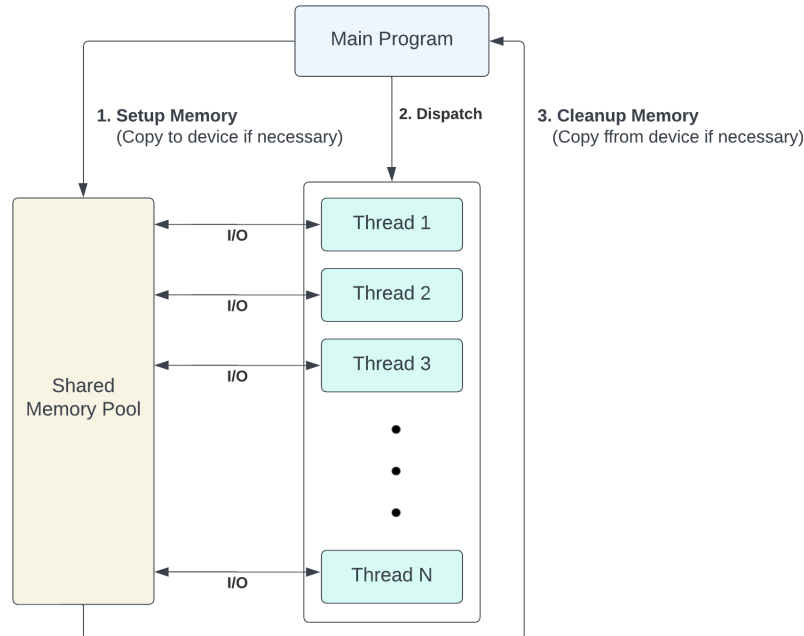


Figure 3.9: A target aware shared memory model. Before dispatching the main program must perform memory copies if dispatching to a GPU, and must also copy data back from the GPU after computation. Otherwise, it is the same as the normal shared memory model because all threads still read and write to the same memory.

parallelization backend, it must be re-compiled to enable that feature. This severely impacts the portability of the code.

Let us consider a more complex case, suppose the functor needs to read from multiple elements in the array in order to update the single entry it is responsible for. This can be difficult using thrust. We see an example in figure 3.11 which demonstrates the limitations of thrust in handling this scenario.

We also find that JUMP is able to handle more complex calling structures and program depth. This is a function of better abstractions like the "jump::iterate" call which can trivially parallelize 1, 2 and 3 layers deep for loops. It is also a function of better data structures, namely the "jump::multi_array" which provides the abilities to trivially reason and parallelize over multi-dimensional arrays, the "jump::shared_ptr" which allows object ownership to be tracked, and the owned object to be shared and used on GPU trivially. Generally we believe this work provides a basis for algorithms to be implemented interoperably, such that they can be written using heterogeneous data structures and then be used in parallelized function calls on CPU or GPU.

3. Multi-Processing and Parallel Programming

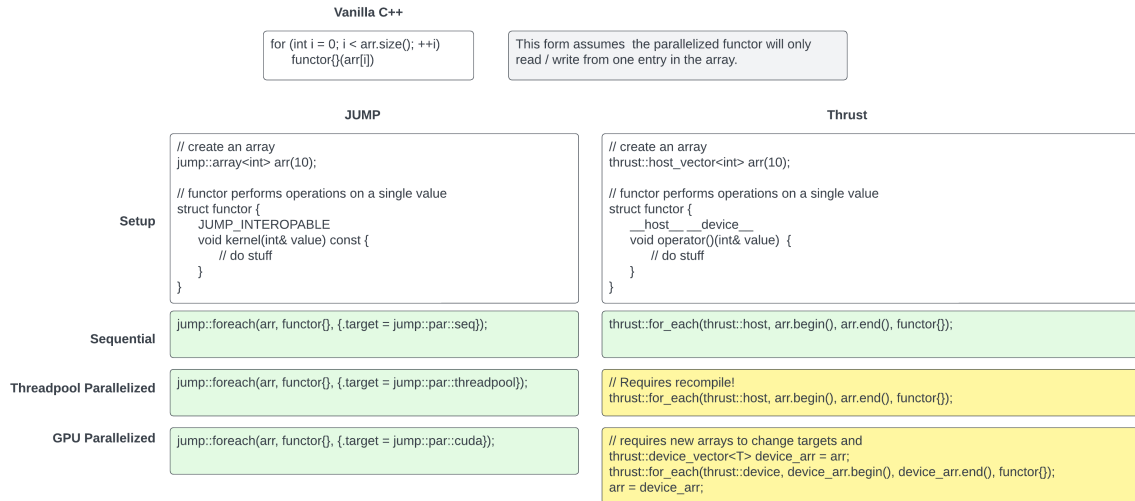


Figure 3.10: Comparison of the ease-of-use of jump and thrust to parallelize a functor over an array using different parallelization backends. We consider the yellow code examples to be less convenient or portable. Specifically, to use a parallelization backend different than sequential evaluation on the GPU or parallel evaluation on the GPU, the code must be recompiled in the case of using thrust with threadpool based parallelization. Additionally, to use thrust with GPU based parallelization instead of sequential CPU evaluation, a specific data structure must be used. This means that thrust code is inherently specific to host or device evaluation. Whereas in all of these cases JUMP requires only a small parameter switch that can be done at run-time.

While JUMP provides a high-level abstraction from GPU memory details, it is understood that those details must still be handled. The mechanism for this is the `to_device()` function. We provide an example of how the call structure looks in figure 3.12. A `jump::array` data structure uses compile-time introspection to determine if the contained data structure has the `to_device()` function, if it does it assumes that this function will in turn copy any data to the GPU in it's child members. For flat data types (no pointers, arrays, dynamic memory) copying the data to the GPU is trivial and no `to_device()` method is necessary. However, if one of those complex data types is contained then `to_device()` should be implemented to handle memory copying. If the complex type is one of the jump data structures, all the details are already handled within it's own `to_device()` function and it becomes a matter of recursively calling `to_device()` in child members until all complex jump data structures have their `to_device()` function that actually handles memory management is called.

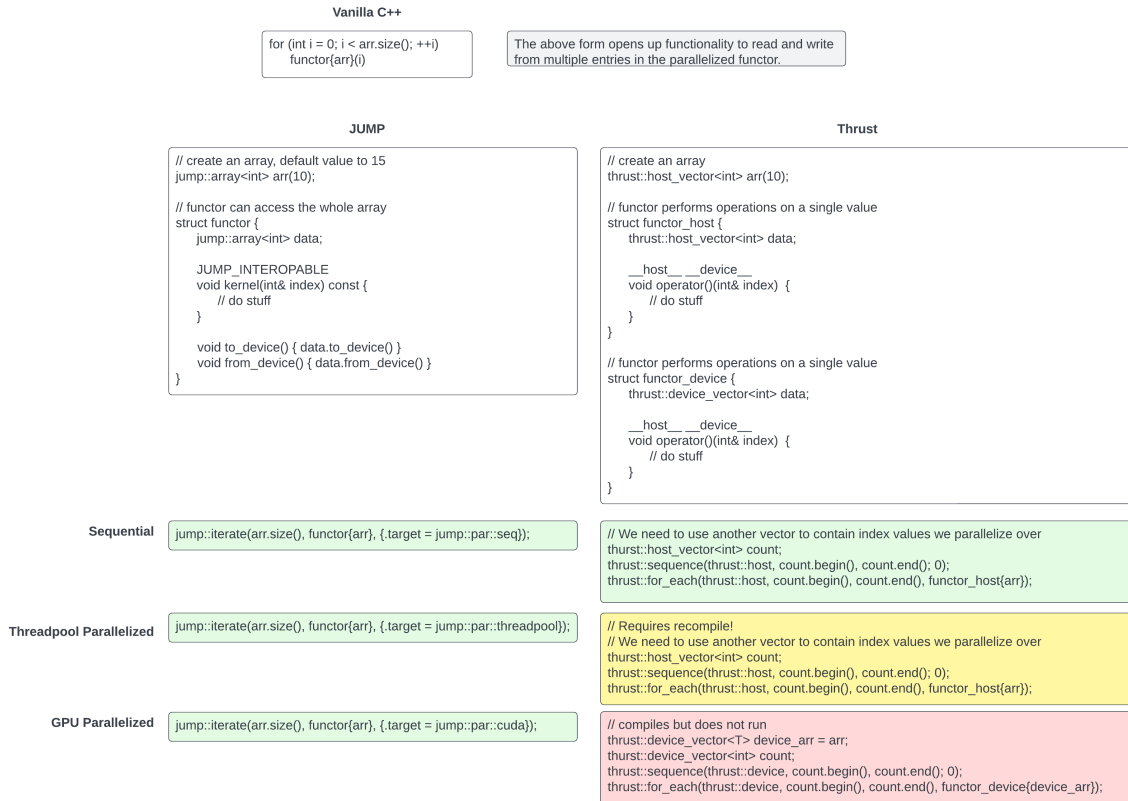


Figure 3.11: Comparison of the ease-of-use of jump and thrust to parallelize a functor over an array using different parallelization backends. We consider the yellow code examples to be less convenient or portable. JUMP is easily able to handle this case, again reducing to a single call and parameter change to switch backends. Whereas thrust has additional calls to facilitate the index processing, requires re-compilation for threadpool based parallelization, and is unable to parallelize this example using the GPU.

3.4.1 Omissions

While JUMP advances an interoperable and heterogeneous computation paradigm, there is still some functionality it omits. The most important feature that it's missing is parallelized reductions, which thrust has implemented very well. This is a rather glaring omission as it is something that could improve the runtime performance of MPPI by using a reduction for the control update (which takes the most time out of all the MPPI parallelized operations).

3. Multi-Processing and Parallel Programming

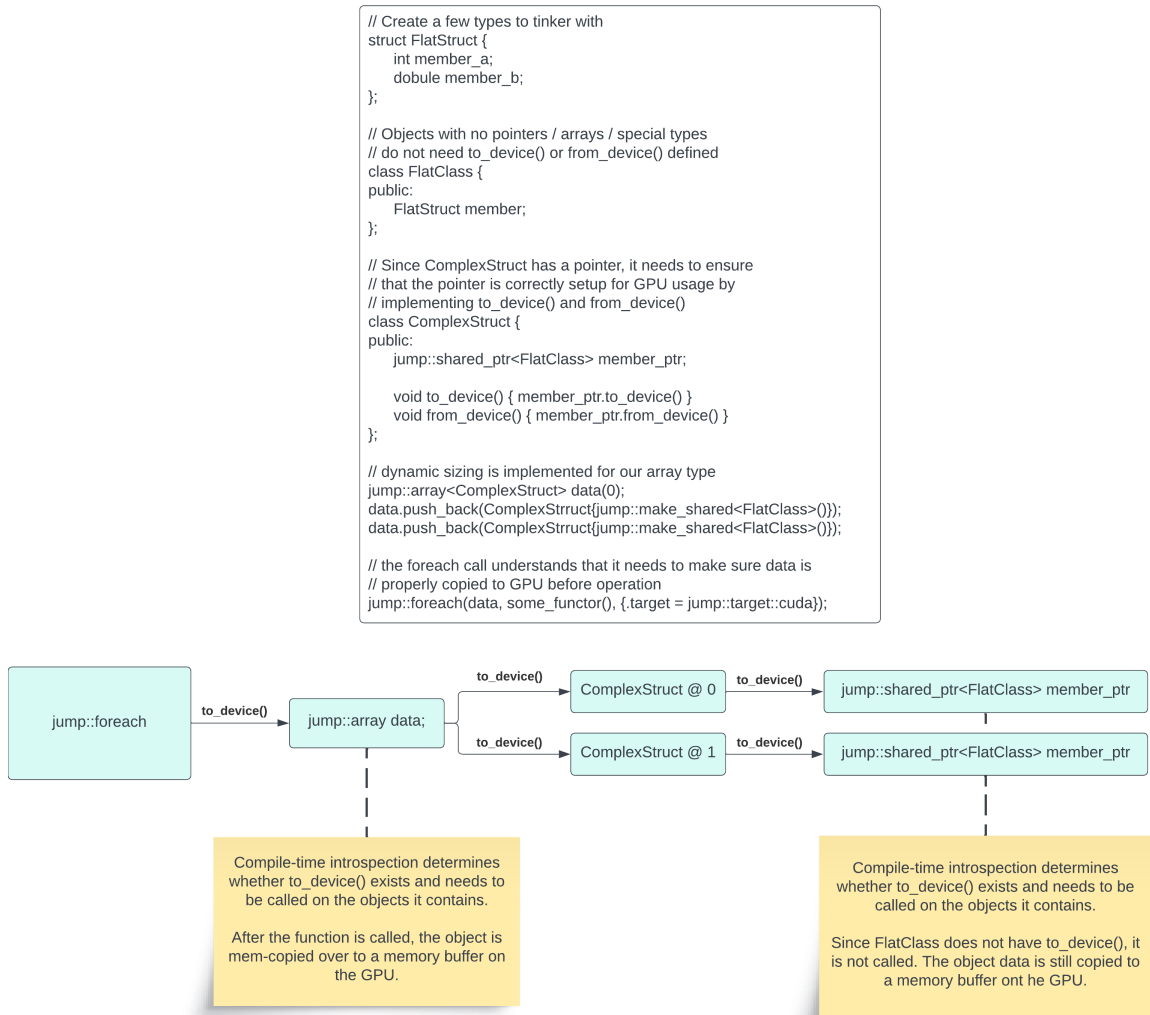


Figure 3.12: The calling structure of nested data types being parallelized over. The `foreach` call triggers a recursive `to_device()` call that transfers each child member to device memory as necessary.

Chapter 4

Controls Methodology

In this section we describe the modeling and cost-functions used to phrase navigation as an optimal control problem. This is implemented in an optimal control framework called "Just Control" or jctl which leverages the jump multi-processing framework to parallelize the optimization process. While the jctl framework has many utilities, cost functions, and infrastructure that can be useful for problems beside navigation, we describe in detail only the parts specific to the navigation task.

Generally, The jctl framework considers the optimization problem phrased in equation 4.1. We believe this structure describes a large family of problems, including many constrained optimization problems that can be represented in this format by using exponential barrier functions augmenting the cost functions $c(x_t, u_t)$ and $c(x_T)$.

$$u^* = \operatorname{argmin}_u c_F(x_T) + \sum_{t=0}^T c(x_t, u_t) \quad (4.1)$$

$$\text{s.t. } x_{t+1} = f(x_t, u_t) \quad (4.2)$$

$$(4.3)$$

It is important to note that the problem described in equation 4.1 does not include any assumptions of stochasticity. This is because model stochasticity and the forms that can be accounted for are intimately related to the solver that is used. The Cross-Entropy Method (CEM) implementation uses a form of stochastic optimization,

however it does not account for any stochasticity in the model. Whereas the Model Predictive Path Integral (MPPI) method does consider some model stochasticity, as such the specific form of that stochasticity and uncertainty is specific to that optimizer and parameterized in that solver and not the general control framework and the models implemented there-in.

In section 4.1 we describe the primary model that is used by this framework, the kinematic bicycle model. We then discuss the cost functions that are used in the navigation task in section 4.2. Lastly in section 4.5 we describe how the model and cost functions culminate into a fully-fleshed optimal control problem that addresses the navigation challenge and discuss the utility of navigation as a benchmark we use.

4.1 Kinematic Bicycle Model

Vehicle modelling is in itself a challenging task with a large body of research and active frontiers. For tractability, we limit the scope of this work and assume a kinematic bicycle model is sufficient to represent the motion of our vehicle when used in a feedback-control setting in an MPC controller. The kinematic bicycle model describes a bicycle moving in two-dimensional space. This model is unable to describe the complex terrain interaction, slip, or any forces or dynamics acting on a four-wheeled all-terrain vehicle. However, it is able to provide some concept of how the Ackerman steering of a four-wheeled vehicle constrains the movement of the vehicle. Figure 4.1 shows a visual representation of this, with a bicycle overlaid on the four-wheeled vehicle body. The key assumption here is that the movement of the bicycle with some steering angle is roughly the same as a four-wheeled vehicle with the same steering angle and wheel-base length.

Accepting this assumption, we first derive a continuous-time motion model for the vehicle. Figure 4.2 shows how the bicycle model can be represented moving through a two dimensional plane. Our primary reference point is the center of the rear-axle (the rear-wheel), which gives us the state variables x and y , the location of the center of the rear-axle in the global frame shown in figure 4.2. Assuming we input the steering angle δ and the velocity V , we must derive a model for the following: $(\dot{x}, \dot{y}, \dot{\theta})$. The \dot{x} and \dot{y} equations can be obtained by projecting the velocity V onto the x and y axis using simple trigonometry. The yaw rate can be derived by calculating the

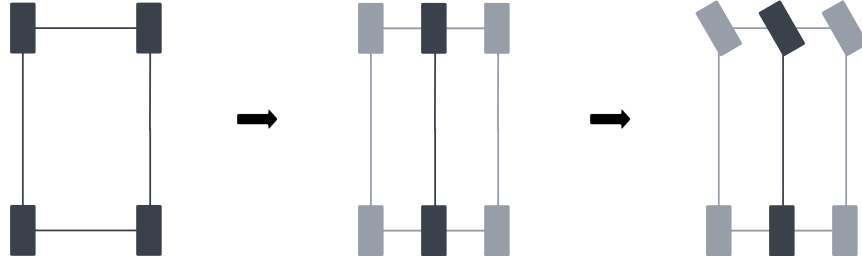


Figure 4.1: How a kinematic bicycle model can be used to represent the movement of a four-wheeled vehicle.

angular velocity $\dot{\theta} = \omega = \frac{V}{R}$, and using trivial trigonometry to calculate $R = \frac{L}{\tan(\delta)}$, $\dot{\theta} = V \frac{\tan(\delta)}{L}$.

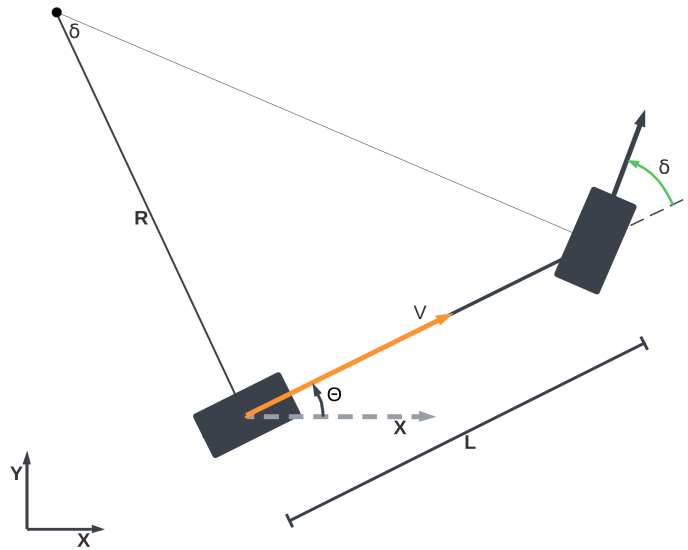


Figure 4.2: The kinematic bicycle model's motion through space. The bicycle has a primary reference point at the center of the rear wheel. It is assumed that wheel is driving the vehicle forward along at some velocity V . It is rotated θ relative to the x -axis. The bicycle has a wheel-base with length L . Given some steering angle δ of the front wheel, it is rotating around some point which is R distance from the rear wheel, with an instantaneous center of rotation at the intersection of the lines orthogonal to the wheels of the bicycle.

This allows us to phrase the motion model as a continuous-time motion model of the form $\dot{x}_t = f(x_t, u_t)$. Where $x_t \in \mathbb{R}^3 = \begin{bmatrix} x & y & \theta \end{bmatrix}$ and $u_t \in \mathbb{R}^2 = \begin{bmatrix} V & \delta \end{bmatrix}$. To

4. Controls Methodology

clarify notation, when we use the variable x , we are referring to the x position of the center of the rear axle and not some state vector notated as x_t .

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = f \left(\begin{bmatrix} x \\ y \\ \theta \end{bmatrix}, \begin{bmatrix} V \\ \delta \end{bmatrix} \right) = \begin{bmatrix} V \cos(\theta) \\ V \sin(\theta) \\ V \frac{\tan(\delta)}{L} \end{bmatrix} \quad (4.4)$$

We then restructure the model in two ways, firstly we convert the state to $x_t \in \mathbb{R}^5 = [x \ y \ \theta \ V \ \delta]$, $u_t \in \mathbb{R}^2 = [a \ \dot{\delta}]$. Where $a = \dot{V}$ = acceleration. Then using Euler integration we discretize the model to some time resolution Δt . For clarity, the state and control spaces of the model in equation 4.5 are described in Tables 4.1 and 4.2.

$$\begin{bmatrix} x \\ y \\ \theta \\ V \\ \delta \end{bmatrix}_{t+1} = f \left(\begin{bmatrix} x \\ y \\ \theta \\ V \\ \delta \end{bmatrix}_t, \begin{bmatrix} a \\ \dot{\delta} \end{bmatrix}_t \right) = \begin{bmatrix} x \\ y \\ \theta \\ V \\ \delta \end{bmatrix}_t + \Delta t \begin{bmatrix} V_t \cos(\theta_t) \\ V_t \sin(\theta_t) \\ V_t \frac{\tan(\delta_t)}{L} \\ a_t \\ \dot{\delta}_T = t \end{bmatrix} \quad (4.5)$$

Variable	Notated	Description
x	x_t^0	The x position of the center of the rear-axle in the global frame.
y	x_t^1	The y position of the center of the rear-axle in the global frame.
θ	x_t^2	The yaw of the vehicle in the global frame relative to the x-axis.
V	x_t^3	The velocity of the vehicle, aligned with the body of the vehicle.
δ	x_t^4	Steering angle of the vehicle, relative to the body of the vehicle.

Table 4.1: State Variables, their state vector notation, and a short description of these variables.

Variable	Notated	Description
a	u_t^0	The acceleration (\dot{V}) to command.
$\dot{\delta}$	u_t^1	The change in the steering angle with respect to time to command.

Table 4.2: Control Variables, their control vector notation, and a short description of the variables.

4.2 Cost Functions

The main objective of the optimization phrased in equation 4.1 is to minimize some cost function $c(x_t, u_t)$ over the control horizon T , in addition to a terminal cost $c_F(x_T)$. This form is able to express many common cost functions such as traversal cost, target state tracking, quadratic tracking costs and many more. The task of autonomous navigation can be phrased many different ways as an optimization problem, often utilizing hierarchical approaches that reduce the controls optimization problem to a path or waypoint tracking problem. We structure it as a waypoint tracking problem with some environmental awareness provided through a traversal cost over a costmap.

While the high-level optimization problem phrased in equation 4.1 reasons about a single cost function for each time step and terminal cost, we understand that often cost functions are the composition of several objectives. We assume for simplicity that unless otherwise noted, the cost functions described in section 4.2.2 and 4.2.1 are linearly combined as in equation 4.6. Since the cost functions described below need only the state x_t , $c(x_t, u_t) = c(x_t)$, further simplifying the cost functions.

$$c_{\text{final}} = c_{\text{traversal cost}} + c_{\text{waypoint tracking}} \quad (4.6)$$

In the below cost functions we use the notation x_t to mean some state along the trajectory being optimized over in equation 4.1, we then index into a vector $x_t = \begin{bmatrix} x & y & \theta & V & \delta \end{bmatrix}$ so that x_t^0 is the x position of the vehicle in the global reference frame as shown in figure 4.2, x_t^1 is the y position of the vehicle in the global reference frame as show in figure 4.2, and so forth as delineated in Table 4.1.

4.2.1 Traversal Cost

Often in robot navigation there is a concept of traversal cost that is used to constrain the optimization problem and produce more expressive behaviors. This concept of traversal cost can be derived from hand-tuned heuristics such as obstacle height as in figure 4.3, a boulder might break the vehicle if traversed at speed so it can be assigned lethal cost. Semantic information can also be considered, for example the bushes in the same figure pose little risk to the vehicle and so do not have any associated cost in that example, despite being a certain height. The costmap could also include a cost

4. Controls Methodology

for hitting those bushes that is much less than the boulder to incentive avoiding them, but not at the cost of hitting a truly lethal obstacle such as the boulders. Tuning these costmaps is an extensive area of research, for example it can be derived from a learned cost-space such as in [30], or from a complex concept of risk as in [11]. In this

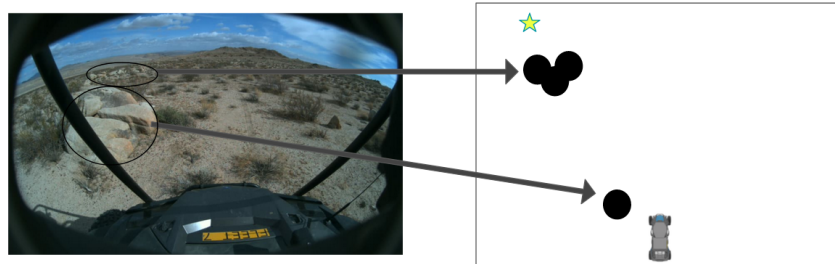


Figure 4.3: An example of how obstacles in an environment can be converted to a costmap parameterized over x-y space. In the camera image, there are two clusters of boulders, in the costmap on the right the black areas correspond to lethal areas in the costmap that the vehicle must avoid. The golden star to the upper left of the costmap is the waypoint the vehicle might be attempting to reach as part of the navigation task.

work we primarily reason about lethal and non-lethal costs, however the traversal cost function we construct is able to reason about costs of continuous or discrete values and is extensible to many of these abstract or theoretically rigorous concepts of cost and risk.

To construct the traversal cost we reason about the footprint of the vehicle as it traverses over a discretized costmap. This discretization assumes that the four corners of the vehicle will be sampled, with an additional number of longitudinal ($n_{\text{longitudinal samples}}$) and lateral ($(n_{\text{lateral samples}})$) samples, with $n_{\text{samples}} = (n_{\text{longitudinal samples}} + 2) * (n_{\text{lateral samples}} + 2)$, as shown in figure 4.4. Since the reference point for our model as described in section 4.1 is the rear axle we assume two functions exist. The first converts from a state x_t to a sample position s_k provided a sample number $k \in [1, n_{\text{samples}}]$, this function can trivially be constructed using two dimensional coordinate transforms and takes the form: $s_k = f_{\text{sample}}(x_t, k)$. The second function that is assumed to exist is the function that takes a sample position s_k and generates a cost from a discretized costmap, this takes the form: $\text{cost} = c_{\text{costmap}}(s_k)$. Using these functions we construct the traversal cost function in equation 4.7 which takes the average over all sampled points.

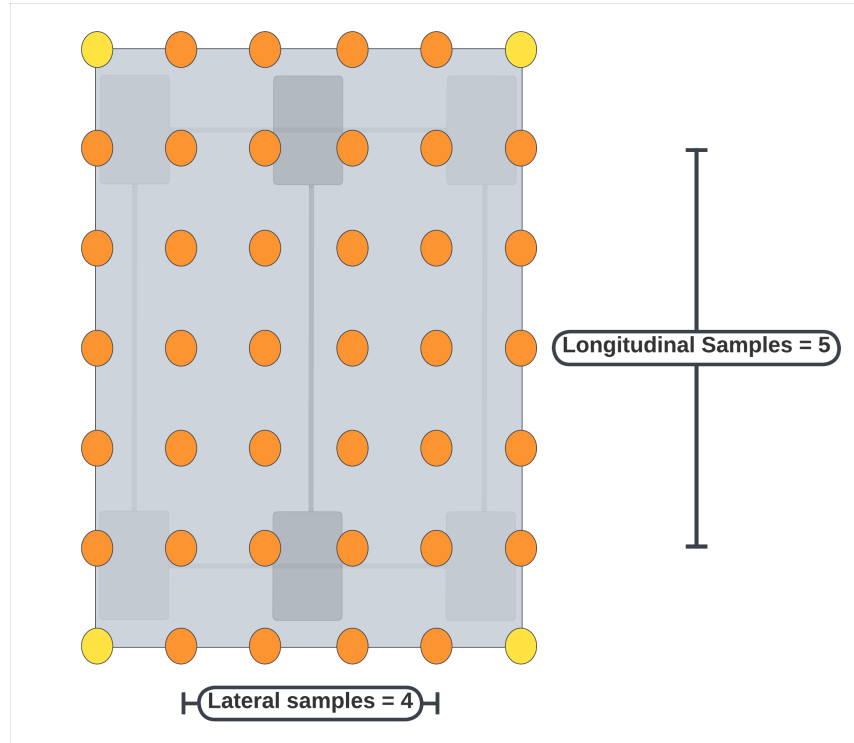


Figure 4.4: The vehicle footprint that encompasses the entire base of the vehicle is discretized to sample some number of points over the costmap. It is assumed that at minimum the four corners of the vehicle will be sampled, and some number of longitudinal and lateral samples are added to produce a total of $(n_{\text{lateral samples}} + 2) * (n_{\text{longitudinal samples}} + 2)$ samples over the footprint of the vehicle. These samples are distributed evenly over the footprint of the vehicle.

$$c_{\text{traversal cost}}(x_t) = \frac{1}{n_{\text{samples}}} \sum_{k=0}^{n_{\text{samples}}} c_{\text{costmap}}(f_{\text{sample}}(x_t, k)) \quad (4.7)$$

The function c_{costmap} can be trivial to implement, in it's simplest form it takes a point in space and goes to the discretized cell in the costmap that contains that point and returns the value in that cell. The jctl implementation can perform that function, but also has options to perform bilinear interpolation over the costmap values. Another key feature of the jctl implementation is the guarantee that if any of the samples are in a lethal cost cell, the final value is scaled such that the final averaged cost is greater than or equal to a lethal cost.

This is important since for the navigation problem interacting with a lethal obstacle

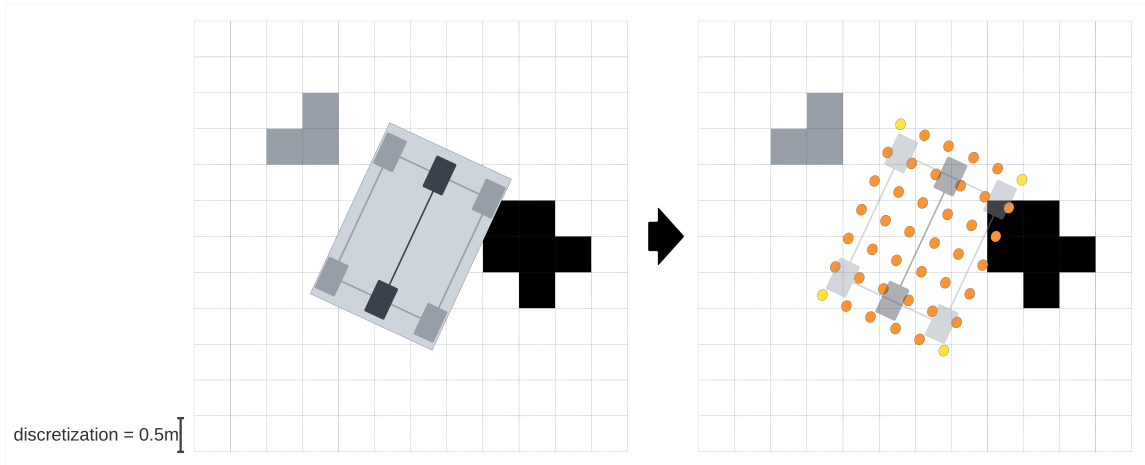


Figure 4.5: The vehicle footprint is shown over a costmap, which is then converted to discrete points that can be sampled over the costmap.

(accruing a lethal traversal cost) is thought of as a constraint rather than simply a cost. As discussed in section 4, we assume that constraints can be incorporated into the cost function as an exponential barrier function, but the costmap is implemented with a finite (however arbitrarily high) lethal cost for lethal traversals. By guaranteeing that any lethal traversal result in a lethal traversal cost after any interpolation or averaging, we can instead limit any other costs to a finite value lower than the lethal traversal cost, guaranteeing that unless there is no safe solution found the optimum will never intersect with a lethal obstacle. Additionally, unless costmap interpolation is enabled, it is also guaranteed that the final traversal cost will be less than the lethal cost if there is no sample within a lethal cost cell. This motivates the cost bounding described in section 4.2.2. Another potential solution to guarantee lethal obstacles are not considered is to explicitly remove rollouts with lethal interactions from the weighted average (explicitly zero the cost). While this approach has merit, this additional processing breaks some of the parallelization and would add additional overhead to the run-time, so we consider our approach to be reasonable.

4.2.2 Waypoint Tracking

For the off-road navigation task, we want the vehicle to reach a waypoint or some goal within some radius. To facilitate this behavior, we reason about two cost components.

The first is simply x-y offset from the waypoint, termed waypoint distance, calculated simply using an l2-norm as in equation 4.8. The second component reasons about our velocity and direction, with the key idea being that if the vehicle has a non-zero velocity and is vectored towards the waypoint, it will eventually reach it. We therefore construct the function in equation 4.9, which has a target waypoint in addition to a target velocity.

$$c_{\text{waypoint distance}}(x_t) = \left\| \begin{bmatrix} x_t^0 \\ x_t^1 \end{bmatrix} - \begin{bmatrix} x_{\text{waypoint}} \\ y_{\text{waypoint}} \end{bmatrix} \right\|_2 \quad (4.8)$$

$$c_{\text{waypoint direction}}(x_t) = \alpha_{\text{direction}} \sqrt{\left(x_t^2 - \arctan \left(\frac{x_t^1 - y_{\text{waypoint}}}{x_t^0 - x_{\text{waypoint}}} \right) \right)^2} + \alpha_{\text{velocity}} \sqrt{(x_t^3 - v_{\text{target}})} \quad (4.9)$$

To ensure that the waypoint tracking objective never dominates the lethal traversal cost, we bound the output range of the waypoint tracking cost. To do this we use the following function: $c_{\text{bound}}(c_{\text{raw}}, v_{\text{max}}, \lambda) = v_{\text{max}} \frac{\lambda c_{\text{raw}}}{\lambda c_{\text{raw}} + 1}$, the general shape of this cost function is shown in figure 4.6.

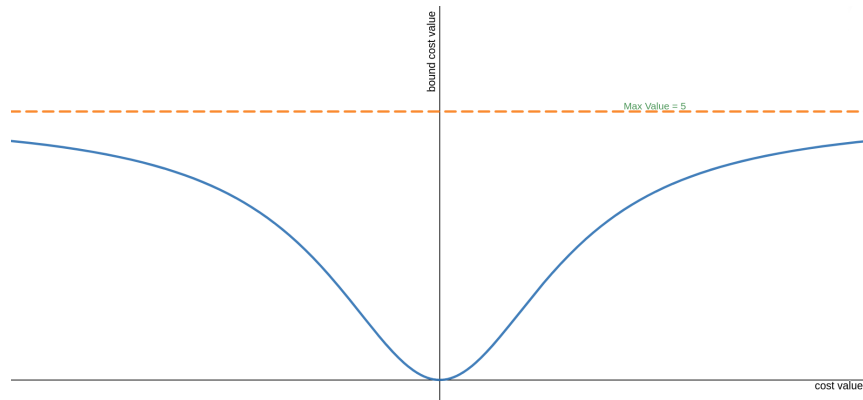


Figure 4.6: A plot of the bound cost function, $y = v_{\text{max}} \frac{\lambda x}{\lambda x + 1}$, with $v_{\text{max}} = 5$, $\lambda = 0.1$.

We bound $c_{\text{waypoint distance}}$ and $c_{\text{waypoint direction}}$ separately then linearly combine them, producing the waypoint tracking cost function in equation 4.10.

$$\begin{aligned}
c_{\text{waypoint tracking}}(x_t) = & c_{\text{bound}}(c_{\text{waypoint distance}}(x_t), v_{\text{wpt. distance max}}, \lambda_{\text{wpt. distance max}}) \\
& + c_{\text{bound}}(c_{\text{waypoint direction}}(x_t), v_{\text{wpt. direction max}}, \lambda_{\text{wpt. direction max}})
\end{aligned}
\tag{4.10}$$

4.3 Costmap Generation

We show a conceptual example in figure 4.3 of how costmaps can be generated from terrain. The benchmarking we describe in section 4.5 is performed solely in simulation and relies on procedural generation of costmaps from a configuration file. This procedural costmap generation considers two shapes that can be used to generate obstacles of some cost into the costmap: circles, and rectangles. To generate a costmap a configuration file specifies a positional and size range for some number of each object type.

Table 4.3 demonstrates three different course configurations and the corresponding costmaps that are generated from them. For each shape that needs generated the range provided for each attribute is sampled from uniformly to determine size and placement of the shape in the costmap. For this work, each shape is assumed to correspond to a lethal obstacle, and hence is a lethal cost in the costmap; however, this too is configurable for future work.

4.4 Simulation

A common challenge in robotics is the sim-to-real gap, in order for this work to be meaningful the problems that this work solves must correlate to real-world problems. For example, in section 4.3 we discuss how the costmaps can be conceptually generated from obstacles near the vehicle to suggest that the randomly generated costmaps might correspond costmaps generated by a real perception system in a real-world environment. The simulation that runs a navigation test has other mechanism intended to minimize the sim-to-real gap and ensure this work can be directly applied to a real-life full-scale autonomous vehicle, discussed below.

There are two components to the navigation test simulation, a costmap simulation

Table 4.3: Course configurations and costmaps generated randomly from them.

Configuration	Generated Course (Seed: 0)
<p>(a)</p> <hr/> Type: Circle Count: 20 X Range [m]: [10, 80] Y Range [m]: [-30, 30] Radius Range [m]: [1, 2] <hr/>	<p>20 Circles:0</p>
<p>(b)</p> <hr/> Type: Rectangle Count: 20 X Range [m]: [10, 80] Y Range [m]: [-30, 30] Length Range [m]: [1, 3] Width Range [m]: [1, 3] Yaw Range [rad]: [-1.7, 1.7] <hr/>	<p>20 Rectangles:0</p>
<p>(c)</p> <hr/> Type: Rectangle Count: 20 X Range [m]: [10, 80] Y Range [m]: [-30, 30] Length Range [m]: [1, 3] Width Range [m]: [1, 3] Yaw Range [rad]: [-1.7, 1.7] <hr/> Type: Circle Count: 20 X Range [m]: [10, 80] Y Range [m]: [-30, 30] Radius Range [m]: [1, 2] <hr/>	<p>20 Circles, 20 Rectangles:0</p>

that consumes a ground truth costmap and simulates sensor limitations such as sensor range and line of sight, as well as a model simulator that simulates a kinematic bicycle

with added noise.

4.4.1 Costmap Simulator

The costmap simulator consumes the current location of the vehicle x_{vehicle} and y_{vehicle} and first processes sensor range, then from cells that are within range it processes if the cells are obscured by any other lethal cells blocking line-of-sight to the cell being processed. This logic and the resulting simulated costmap are shown in figure 4.7.

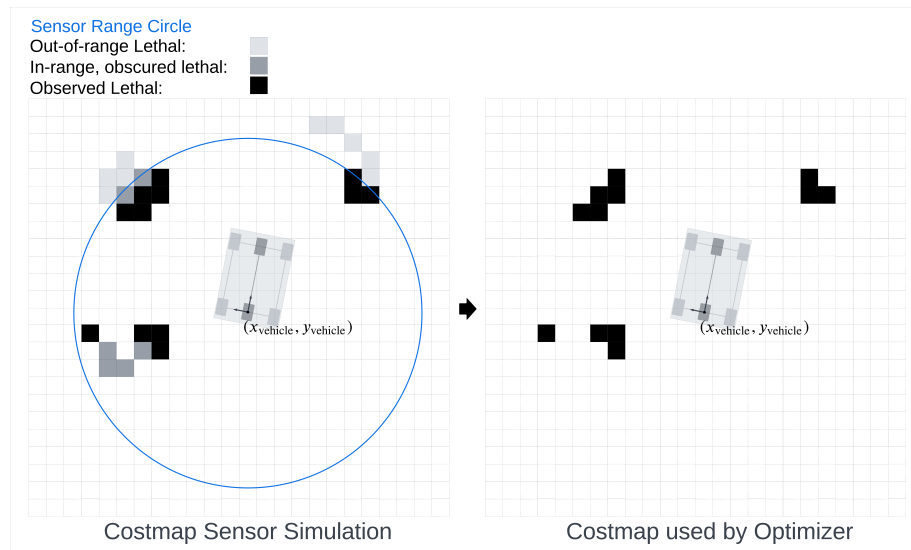


Figure 4.7: Costmap simulation of sensor range and line-of-sight, to the left is how the costmap simulator is processing lethal cells as either out-of-range, in-range but obscured, or visible. To the right is the final costmap that gets passed to the optimizer to perform the navigation task.

This reduces the sim-to-real gap by mimicking the effect of real sensor limitations on the perception system that would generate the costmap used by an optimal control scheme to perform navigation. Specifically, when a vehicle is navigating in an enclosed space the LiDARs and cameras are unable to see around the corner or behind the immediate foliage, and the area beyond is unknown until the vehicle gets through. This simulates a similar effect allowing it to be observed what happens when the optimizer is "surprised" and how it handles situations like those discussed in section 2.4.

4.4.2 Model Simulator

We discuss in section 2.1.1 how disturbances and model uncertainty is often used to capture error in the model. Naturally, in a real world environment there are model deviations as well as uncertainty in system state. To capture this, we inject noise into the control input to the system as in equation 2.6. Additionally, the state that is observed by the controller has noise injected. While there are certainly more expressive disturbances that can be injected into the model update to test controller robustness, together these capture some modeling error as well as error and uncertainty in the state estimate that would be seen from a deployed SLAM system.

4.5 Navigation as Optimal Control

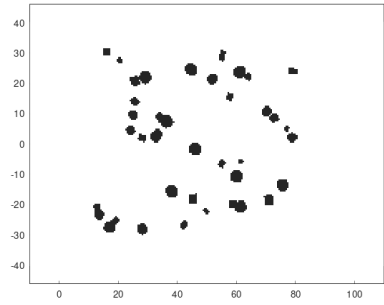
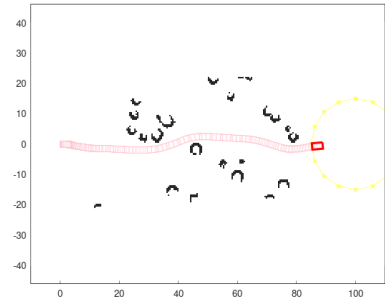
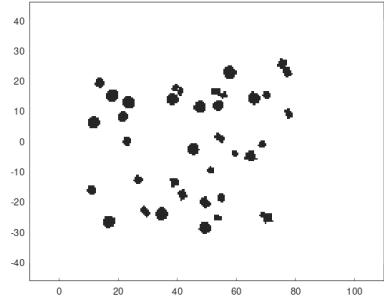
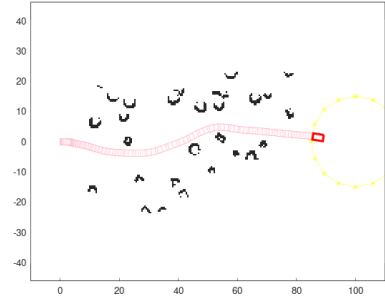
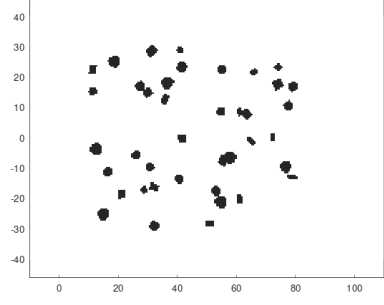
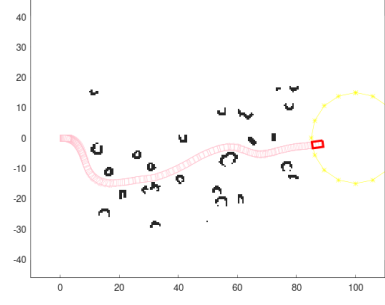
Using the cost functions and models described above we can form a fully-fleshed optimization problem that fits the form of equation 4.1. Where $c_F(x_t) = c_{\text{waypoint tracking}}(x_t) + c_{\text{traversal cost}}(x_t)$ and $c(x_t, u_t) = c_F(x_t)$. For each test we place a waypoint at $(100, 0)$ so that for $c_{\text{waypoint tracking}}$, $x_{\text{waypoint}} = 100$ and $y_{\text{waypoint}} = 0$. We generate three costmaps from the configuration defined in Table 4.3(c) using the random number generator seeds $(0, 2, 3)$, the courses and example navigation's are in Table 4.4. Note that we originally intended to benchmark with the course seed 1, however that data was corrupted and we did not have time to regenerate it.

4.6 Navigation as Benchmark

This simulation environment allows us to perform a benchmarking of the optimization algorithms we implement within the jetl framework, specifically MPPI and CEM. Additionally, since this is implemented using the jump parallelization backend we can profile how quickly they run with computational constraints, giving insight to real-time performance as well as what performance can be expected with fewer samples. We believe this is useful since many years have passed since a computational evaluation has been published in [31].

We first benchmark the computational efficiency by measuring how quickly the algorithm runs on different platforms, with varying numbers of samples, with different

Table 4.4: Benchmark courses and example solutions.

Seed	Ground Truth Costmap	Simulated Costmap and Solution
0		
2		
3		

parallelization backends. Specifically we have the computers listed in table 4.5, which range from a small SOC computer to a powerful ML workstation. This provides insight into what computational performance can be expected with constrained resources.

Additionally, we analyse how the controllers perform in the navigation task, analyzing time-to-waypoint as a function of sample count. This provides insight into how the algorithms perform with a reduced amount of information and whether high particle counts are needed to ensure high-quality performance.

Computer	CPU	Threads	GPU
A Desktop	Intel i9-13900K	32	NVIDIA RTX 4090
B Desktop	Intel i7-8700K	12	NVIDIA RTX 2070
C Desktop	AMD Ryzen 7 3700X	16	NVIDIA RTX 2070 Super
D NVIDIA Orin	Arm Cortex-A78AE	12	2048-core NVIDIA GPU

Table 4.5: Compute Platforms we benchmark on.

4.6.1 Literature Review

Much of this work is oriented around Navigation as a task, we treat this primarily as a surrogate task for benchmarking and not necessarily the end-goal in itself considering the primary contributions of this paper are the parallelization framework. However, we still believe it worthwhile to discuss the applications of stochastic optimization for navigation.

The task we define as navigation is autonomously navigating a vehicle at high-speeds from its current position to a goal, while avoiding any damage to the vehicle. There are many approaches for this task, and often this navigation task is embedded in a larger system that considers a higher object that can be achieved through navigation. For example, we consider the approach in [28], which uses a motion primitive library to plan then passes the path down to a lower-level controller that uses a PI controller to track it. However research has shown that model-aware approaches are able to track more accurately as in [29]. Consequently, research has trended towards more integrated approaches, two years after the grand challenge we saw a more integrated approach with [3] that searched over a trajectory space to directly command a curvature, curvature rate, velocity and acceleration. MPPI and CEM as approaches work very similarly, however they sample in continuous space rather than discrete intervals, this allows MPPI and CEM to refine the solution more accurately than discrete motion primitive libraries.

Other common approaches for navigation using randomization include [10] and [16] which both use RRT's to plan. These approaches are only used in hierarchical schemes, where the result of the RRT search provides a path for a lower-level controller to use. This is a fine approach, however these methods only consider a vehicle moving at slower speeds around $2m/s$, at higher speeds often the disconnect between the

4. Controls Methodology

higher-level planner and what the vehicle can achieve become more evident. Another approach using randomization is [15] which uses an algorithm that samples directly in the control space, allowing it to reason about the model in higher-detail. However, these approaches have fewer real-time guarantees and have fewer guarantees of informative samples than the MPPI work in [32].

There are more complex approaches as in [11] uses a hierarchical approach in addition to a high-fidelity model predictive controller. However, there are many approaches that rely solely on a sample based optimizer for navigation, such as [19], and many of the MPPI works we cite in section 2.1.2. However, considering the focus of this work is on the algorithms themselves we accept that our approach to navigation may not be the most advanced, but is still relevant to the community.

Chapter 5

Benchmarking Analysis

Using the parallelization framework we discuss in section 3.4, we implement the optimal control navigation problem discussed in chapter 4. We then use the stochastic optimization methods discussed in chapter 2 to solve the navigation problem. We use these to benchmark execution rate as well as performance metrics of the algorithms. In section 5.1 we discuss the execution rate benchmarking and what components of the algorithm consume the most time.

5.1 Benchmark Execution Rate Results

We benchmark execution rate using GPU, and threadpool (with 3, 8, 12, 24, 32 threads) backends on four computers. These computers are listed in table 4.5. For each backend on each computer we profile the algorithm with the following number of particles: (25, 50, 100, 200, 500, 1000, 1500, 2000, 2500, 3000).

Computer A in table 4.5 is a very capable computer with Intel i9-13900K with 32 threads and an NVIDIA RTX 4090. We can see the results in figures 5.1 and 5.2. As expected, we see that these algorithms are extremely parallelizable and maintain a real-time rate less than 50Hz for high particle counts up to 3000 using the GPU. Using threadpool parallelization we can see that using 8 threads or more allows the algorithms to maintain a real-time rate of 10Hz with 500 samples which might be sufficient for some online tasks.

Computer B in table 4.5 is a less capable but still higher-end computer with an

5. Benchmarking Analysis

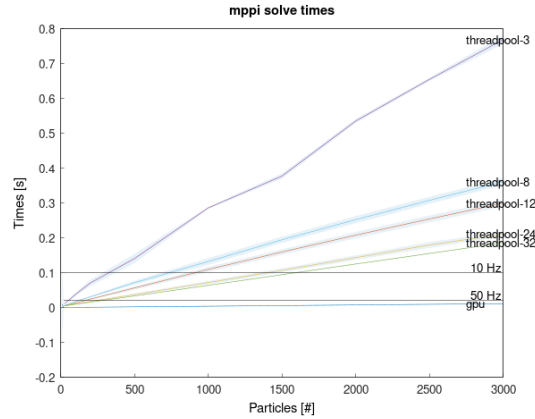


Figure 5.1: MPPI Timing Profile using GPU parallelization on Computer A in table 4.5. This computer has an Intel i9-13900K with 32 threads and an NVIDIA RTX 4090. While a 50Hz real-time rate is only maintained using GPU parallelization, with 8 threads or more the solver maintains 10Hz easily with 500 samples.

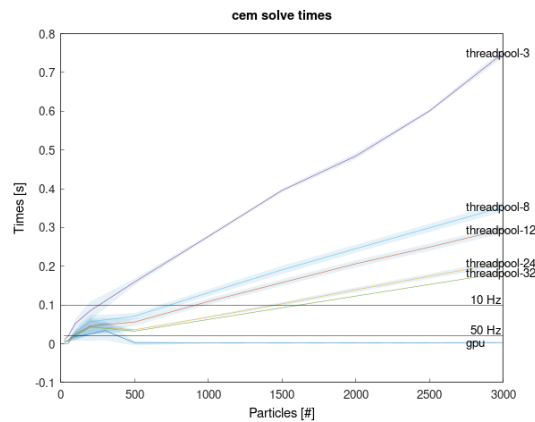


Figure 5.2: CEM Timing Profile using GPU parallelization on Computer A in table 4.5. This computer has an Intel i9-13900K with 32 threads and an NVIDIA RTX 4090. While a 50Hz real-time rate is only maintained using GPU parallelization, with 8 threads or more the solver maintains 10Hz easily with 500 samples. There are interesting spikes in the solve times below 500 samples.

Intel i7-8700K with 12 threads and an NVIDIA RTX 2070. We can see the results for this computer in figures 5.3 and 5.4. On this computer MPPI dips above a 50Hz execution rate at 3000 particles. This CPU is also evidently slower as the algorithms do not run at 10Hz with 500 samples.

Computer C in table 4.5 is another higher-end computer with an AMD Ryzen

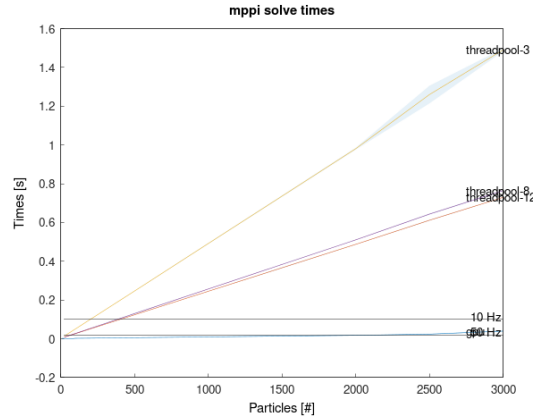


Figure 5.3: MPPI Timing Profile using GPU parallelization on Computer B in table 4.5. This computer has an Intel i7-8700K with 12 threads and an NVIDIA RTX 2070. We see that with GPU parallelization, MPPI runs just below 50Hz at 3000 particles, however at 2500 samples and less it is able to maintain 50Hz real-time performance. With approximately 400 samples, it is able to achieve 10Hz with 8 or 12 threads.

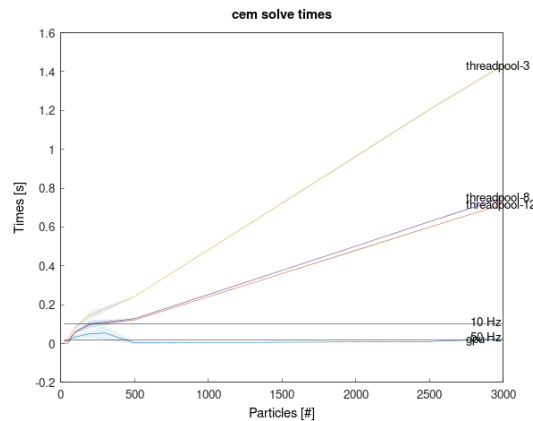


Figure 5.4: CEM Timing Profile using GPU parallelization on Computer B in table 4.5. This computer has an Intel i7-8700K with 12 threads and an NVIDIA RTX 2070. This algorithm is able to run at 50Hz up to 3000 particles, however with threadpool parallelization it struggles to achieve 10Hz at sample counts greater than 200. We see another interesting non-linear relationship between sample count and execution rate at less than 500 particles.

7 3700X with 16 threads and an NVIDIA RTX 2070 Super. We can see the results for this computer in figures 5.5 and 5.6. On this computer MPPI dips above a 50Hz execution rate at 3000 particles. However, it is otherwise able to maintain 50Hz. It

5. Benchmarking Analysis

seems for both CEM and MPPI it is able to achieve 10Hz with 500 samples and less.

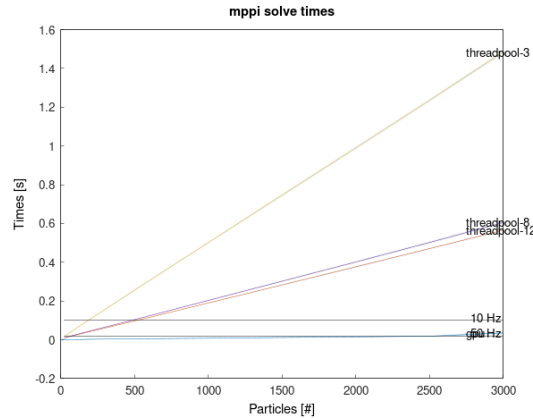


Figure 5.5: MPPI Timing Profile using GPU parallelization on Computer C in table 4.5. This computer has an AMD Ryzen 7 3700X with 16 threads and an NVIDIA RTX 2070 Super. We see that with GPU parallelization, MPPI runs just below 50Hz at 3000 particles, however at 2500 samples and less it is able to maintain 50Hz real-time performance. With approximately 500 samples, it is able to achieve 10Hz with 8 or 12 threads.

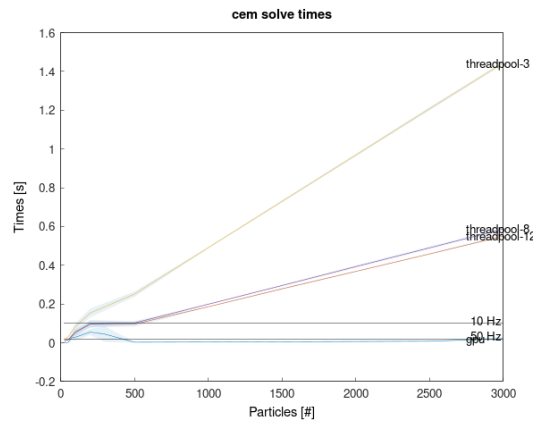


Figure 5.6: CEM Timing Profile using GPU parallelization on Computer C in table 4.5. This computer has an AMD Ryzen 7 3700X with 16 threads and an NVIDIA RTX 2070 Super. This algorithm is able to run at 50Hz up to 3000 particles. While we observe a non-linear relationship between sample count and execution rate below 500 particles, it seems to meet 10Hz for 500 particles and less.

Computer D in table 4.5 is a System on Chip (also SOC) computer that has a 12 core arm CPU, and a 2048 core nvidia GPU. This computer is significantly limited

in comparison to the other desktops, and it shows. We can see the results for this computer in figures 5.7 and 5.8. We see that this computer is unable to maintain 10Hz even at 500 particles on a GPU parallelization backend.

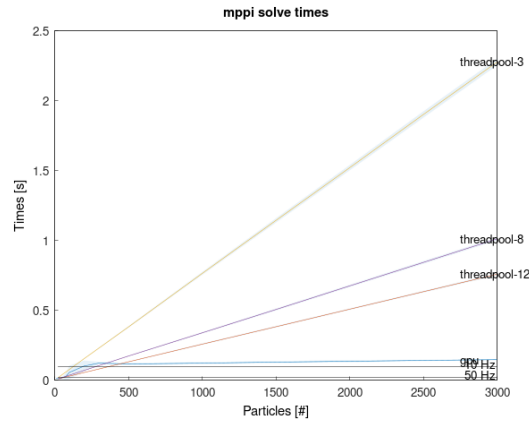


Figure 5.7: MPPI Timing Profile using GPU parallelization on Computer D in table 4.5. This computer has a 12 core arm CPU, and a 2048 core NVIDIA GPU. It is unable to maintain 10Hz at even 500 particles on a GPU parallelization backend.

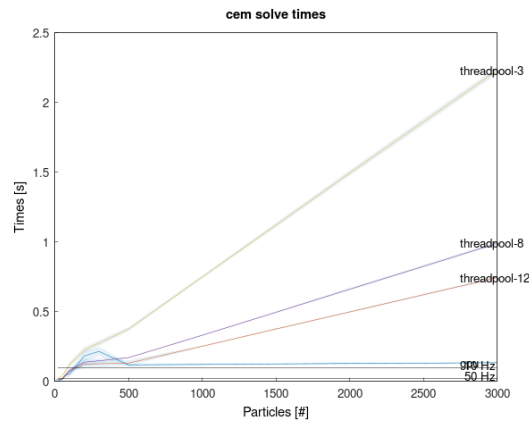


Figure 5.8: CEM Timing Profile using GPU parallelization on Computer D in table 4.5. This computer has a 12 core arm CPU, and a 2048 core nvidia GPU. It is unable to maintain 10Hz at even 500 particles on a GPU parallelization backend.

5.2 Navigation Performance Results

We simulate a navigation task using the costmap and model simulation described in section 4. We simulate with a time discretization of $\delta t = 0.1$. While this reduces the fidelity of the sim we believe this still yields information about how the algorithms perform the navigation task. We leave higher-fidelity simulations with smaller time discretization for future work.

To perform these benchmarks we run the simulation tests with both algorithms with the particle counts (75, 100, 300, 500, 1000, 1500, 2000, 2500, 3000), we found that below that the algorithms are unable to reliably navigate the courses without obstacle collision (lethal cost). The results are detailed in figures 5.9, 5.10, 5.11. All courses are combined into a final average in figure 5.12.

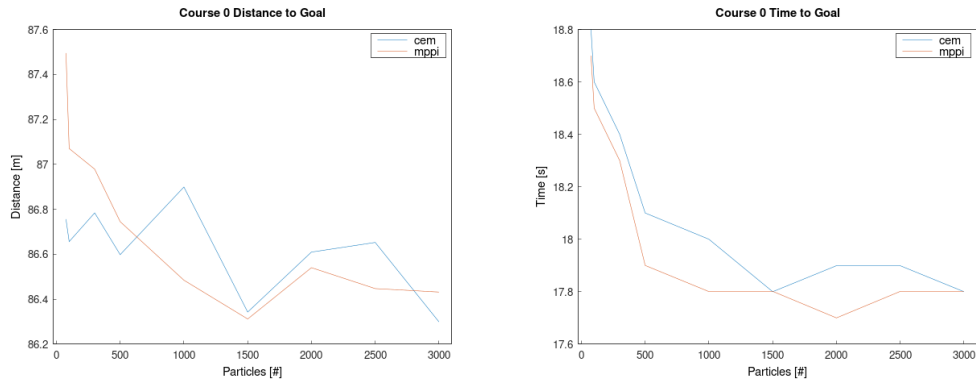


Figure 5.9: Metrics for course 0 in table 4.4. While CEM performs better for low particle counts on distance taken to reach goal metrics, generally MPPI takes a shorter amount of time to reach the goal.

Overall, it appears that while CEM can sometimes find shorter routes to reach the waypoint, MPPI often takes the course at higher speeds thus reaching the waypoint in less time. There is no clear reason for this, the optimal control problem, model, cost functions, and even the ground truth costmap itself that are used by the solvers are identical. For some reason inherent to the solver itself, MPPI takes the course at a higher speed.

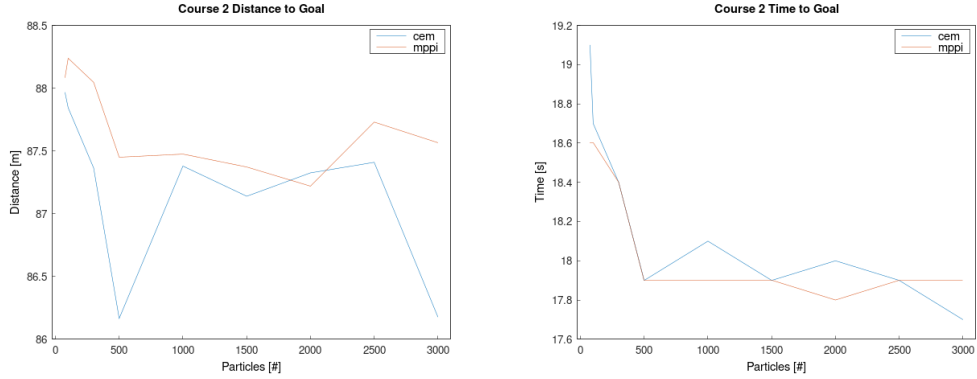


Figure 5.10: Metrics for course 2 in table 4.4. We see that CEM takes a shorter path to the goal, however MPPI maintains higher speed and reaches the goal in less time.

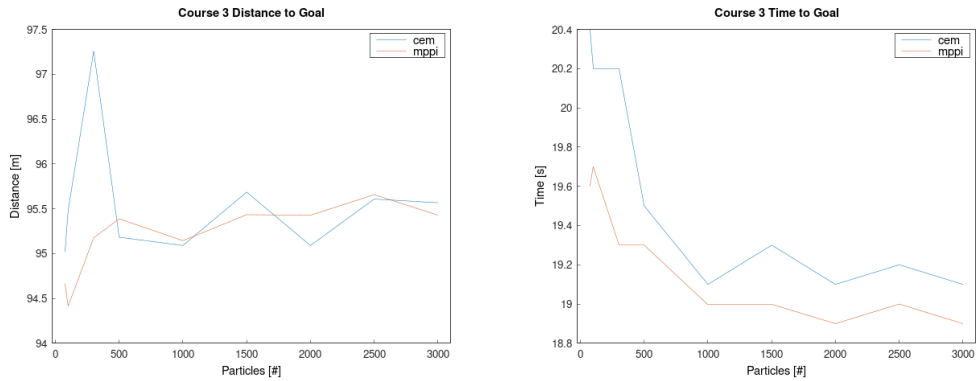


Figure 5.11: Metrics for course 3 in table 4.4. We see that while CEM occasionally takes a shorter path to the goal, MPPI reliably reaches the goal in less time.

5.3 Future Work

This section presents only a preliminary analysis of the results of this benchmark. These results seem to suggest that MPPI is generally a better algorithm for the navigation task. However, understanding why MPPI often moves at faster speeds than CEM to reach the goal in less time requires deeper analysis.

Additionally, we believe an analysis of how execution rates effect performance to be of value. It may be that MPPI running at 100 Hz with 300 samples may be able to outperform MPPI running at 10 Hz with 1000 samples.

Generally, we believe further benchmarking of the same tests with different

5. Benchmarking Analysis

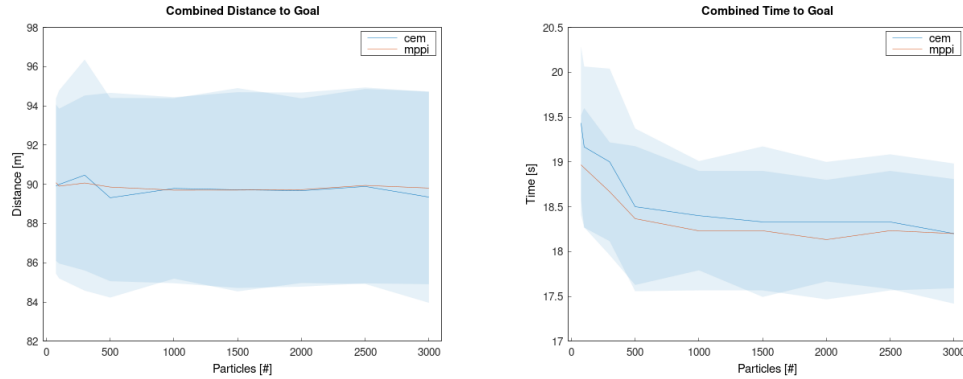


Figure 5.12: Metrics averaged over all courses in table 4.4. There is some minor variation in distance taken to goal at high and low sample counts. On average MPPI reliably finds a faster way to reach the goal.

courses and denser obstacle to be of value. Since stochastic optimization relies on randomization, the sampling can have dramatic effects on the choices the navigating algorithm makes and therefore on performance itself. To demonstrate this we construct a small course with four obstacles, start the vehicle to the left side of the obstacles and set the target waypoint to the right.

We can see the results letting CEM navigate the course with 100 samples in figure 5.13. In each test, the vehicle takes a different path to the waypoint, solely as a result of the pseudo-random number generator having a different seed and everything else in the test remaining the same.

Naturally, one might think this is because the number of samples is so low. We can repeat this test with 3000 samples. The results for this are in figure 5.14 While the paths with the seed set to 1 and 500 take the same path through, still the path itself is noticeably different and every other test the vehicle takes a different path through the course.

One might suspect that something is implemented incorrectly with the pseudo-random number generator implemented in JUMP, however this test is repeatable with all three pseudo-random number generator algorithms implemented in JUMP (split-mix [27], xorshift [23], PCG [22]). It is also repeatable with MPPI up to 3000 particles.

This result is interesting because it was the last one we explore, yet it calls into question the comparative navigation statistics. If the navigation choices made by

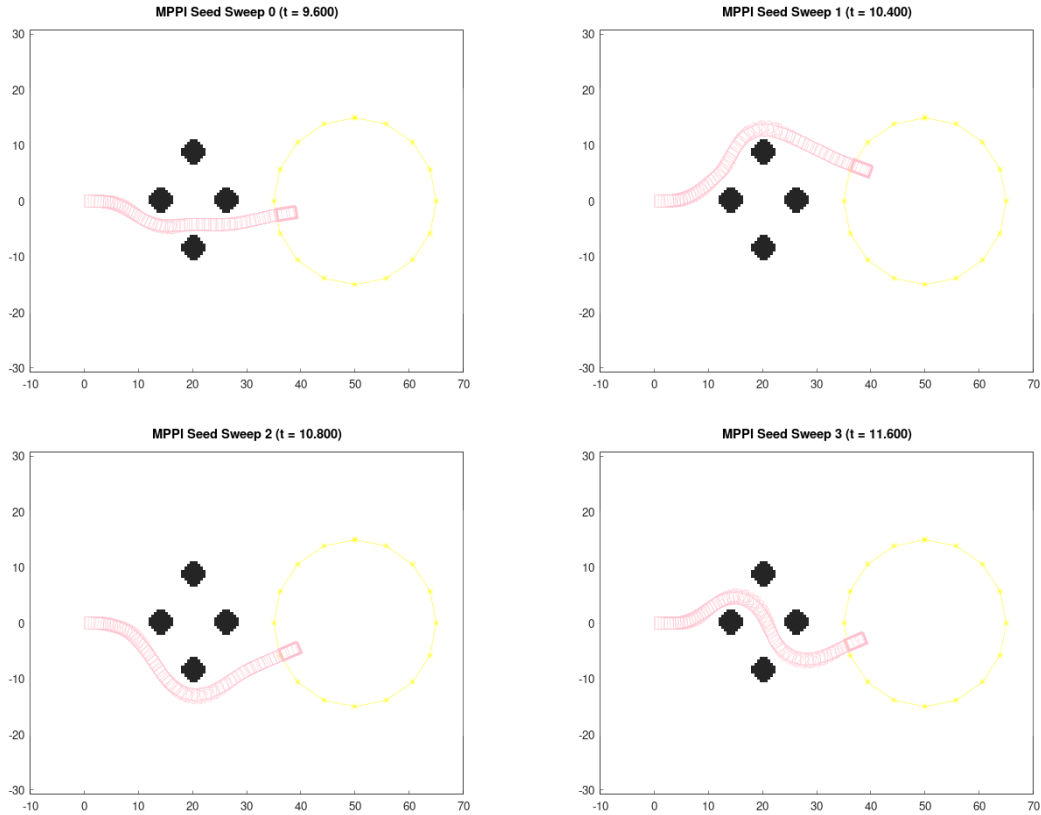


Figure 5.13: Navigation of a four obstacle course with 100 samples and MPPI random number generator seed set to 1 (top left), 200 (top right), 500 (bottom left), 800 (bottom right). The paths take different homotopy classes through the obstacles in each case.

these algorithms is so subject to chance, can they be counted on to reliably produce repeatable and safe behavior. We believe this question must be further explored, especially considering that many implementations are not as computationally efficient as this and are limited to a low number of particles and thus might be subject to greater stochasticity.

5. Benchmarking Analysis

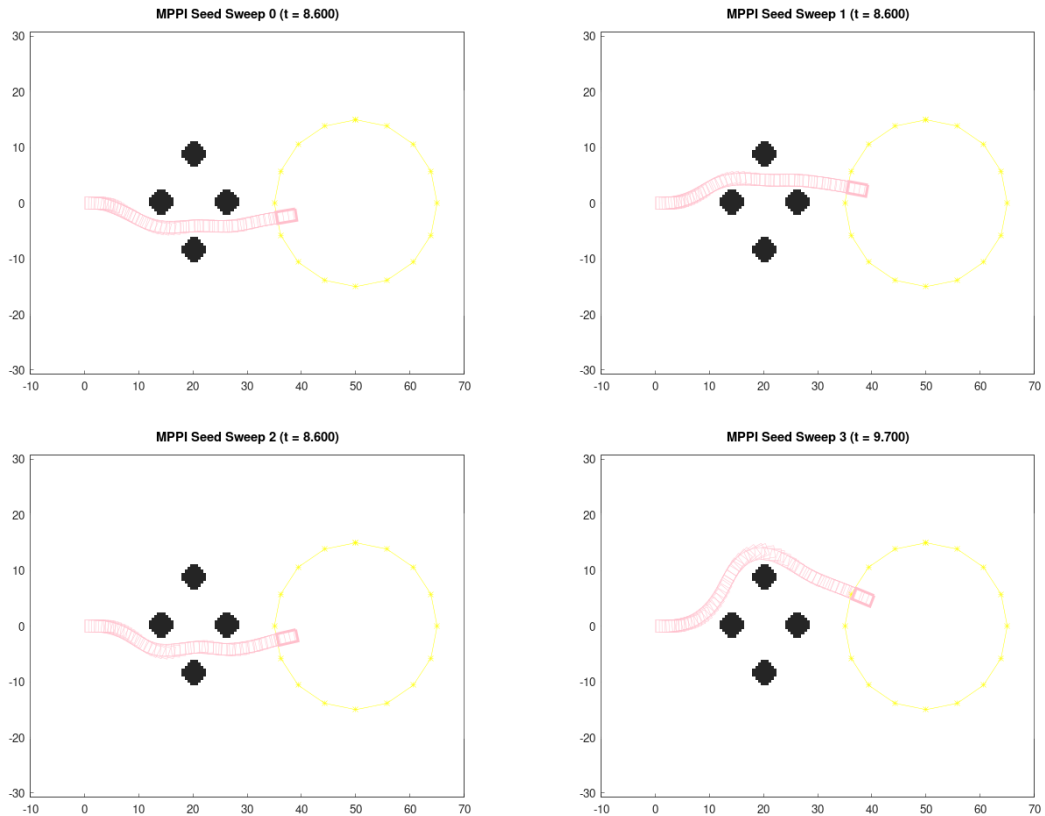


Figure 5.14: Navigation of a four obstacle course with 3000 samples and MPPI random number generator seed set to 1 (top left), 200 (top right), 500 (bottom left), 800 (bottom right). The same path through the obstacle is taken with seeds set to 1 and 500, however the paths are still noticeably different. Every other test takes a different path.

Chapter 6

Conclusions

In this document we discuss stochastic optimization in chapter 2, we discuss two algorithms that are trivially parallelized. We then present a new C++ parallelization framework that expands the heterogeneous and interoperable computing paradigms, providing new interfaces and capabilities for parallelization and ease of implementation in C++. We discuss optimal control methodology and cost functions and models that can be used for autonomous navigation in unknown environments as well as how to simulate them, we then implement these algorithms and methods using the parallelization framework we present.

Using this framework we benchmark various aspects of stochastic optimal control, benchmarking execution rate with different backends and comparing the very similar CEM and MPPI algorithms. We hope this is insightful to those choosing which optimal control algorithm to use in practice, or those implementing optimal control schemes like this for the first time.

6. *Conclusions*

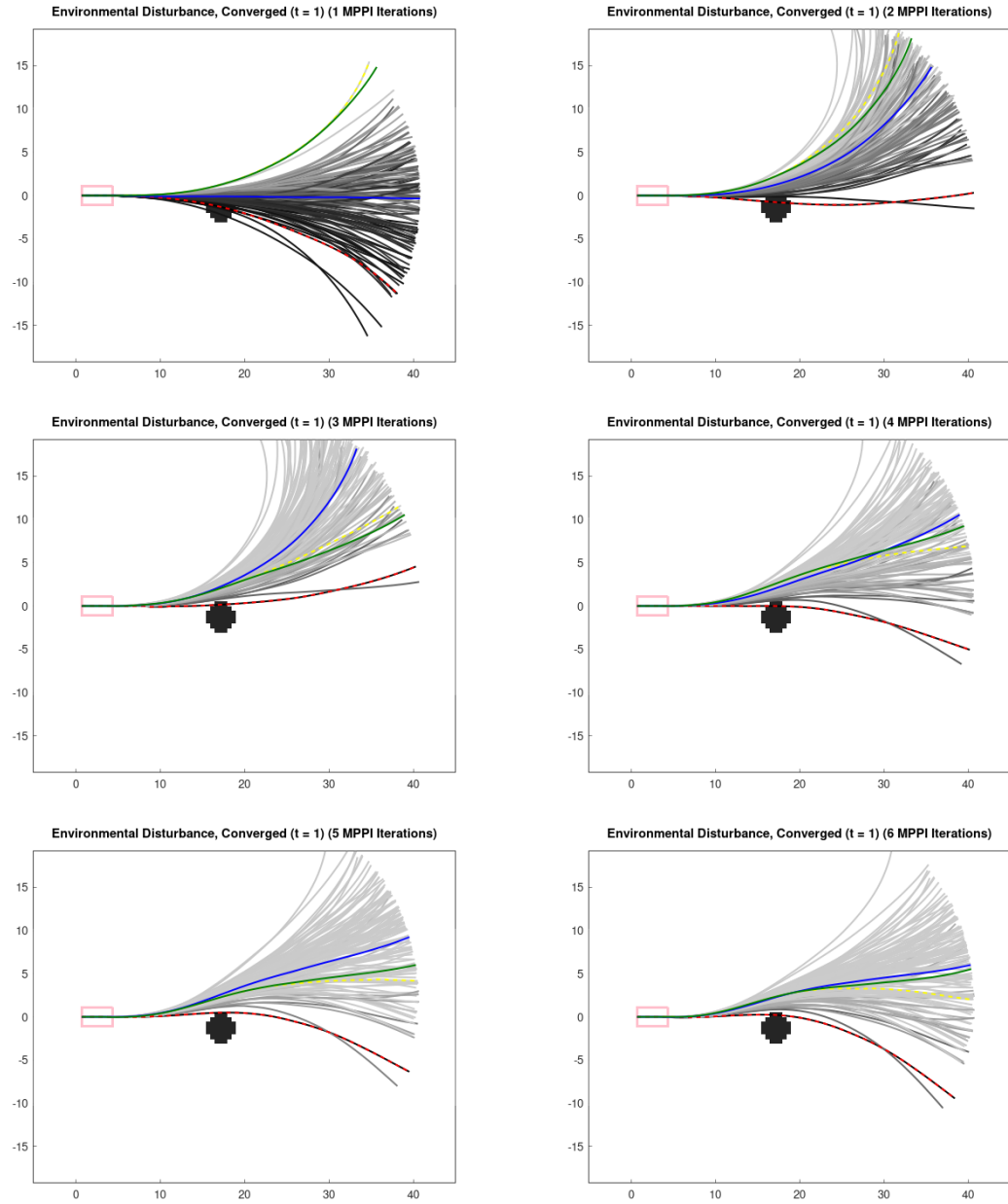
Appendix A

Disturbance Experiments

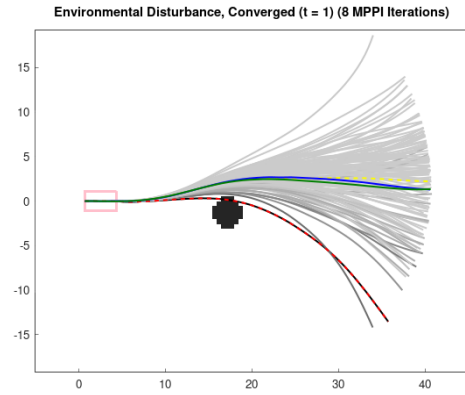
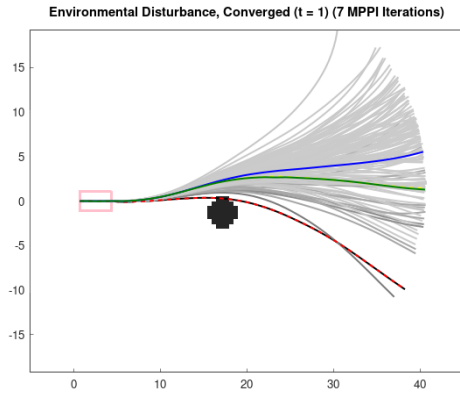
In section 2.4 we demonstrate the effects of difference disturbance types, then show the sampling after several iterations of MPPI so that the sampling converges to a better distribution. We also plot the distribution for the intermediate iterations of MPPI, shown below. This is informative to show how the MPPI distribution shifts over time, and is a positive aspect of running the algorithm at high rates - it allows the algorithm to adjust to changes in the cost landscape more rapidly. It is also interesting to observe the differences between the environmental and model disturbances converged solutions. The experiments are specifically constructed to have the obstacle be positioned in the same place relative to the vehicle at $t = 0$, however the waypoint in both tests is positioned at $(200, 0)$. If the costmap and waypoint were both positioned exactly the same relative to the vehicle, with the same initial seed we would expect the solutions to be the same between the experiments. Since the waypoint is a few meters off, we see them converge to different solutions.

Environmental Disturbance

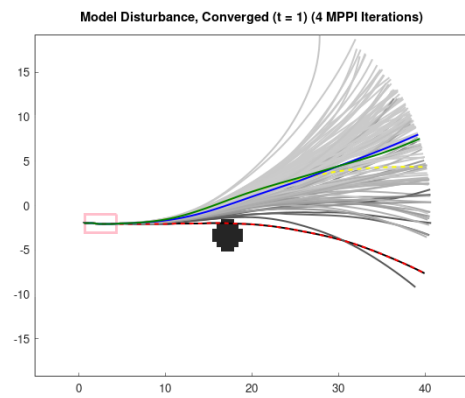
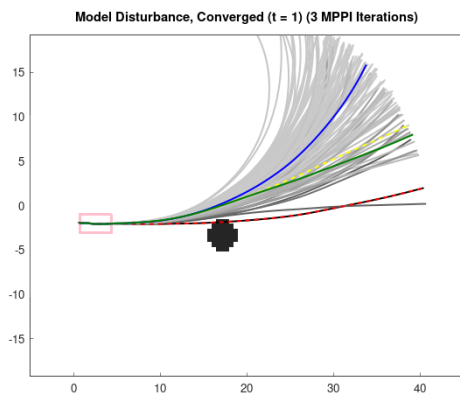
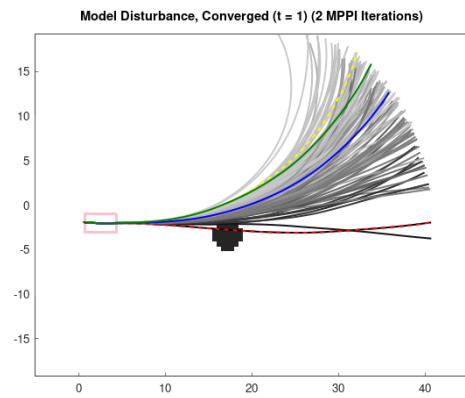
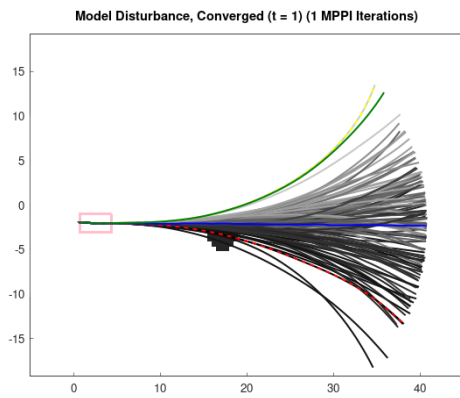
A. Disturbance Experiments



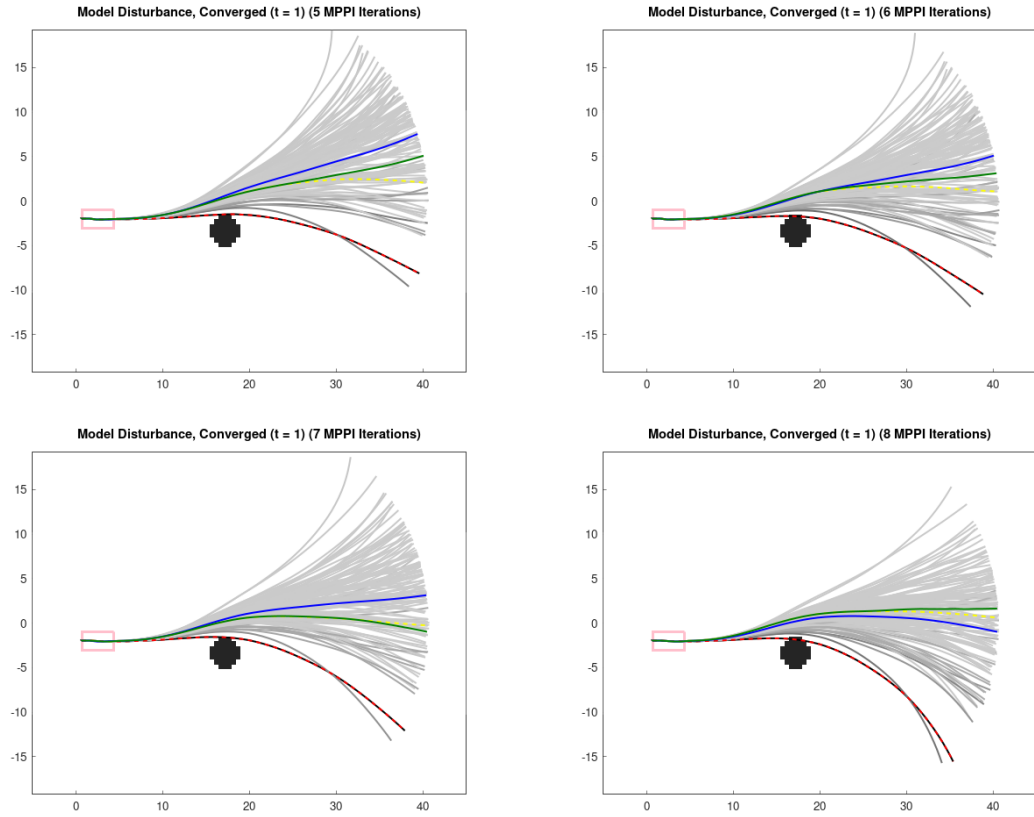
A. Disturbance Experiments



Model Disturbance



A. Disturbance Experiments

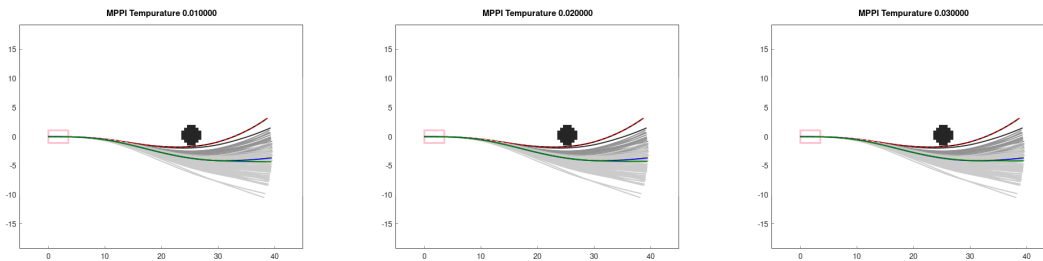


Appendix B

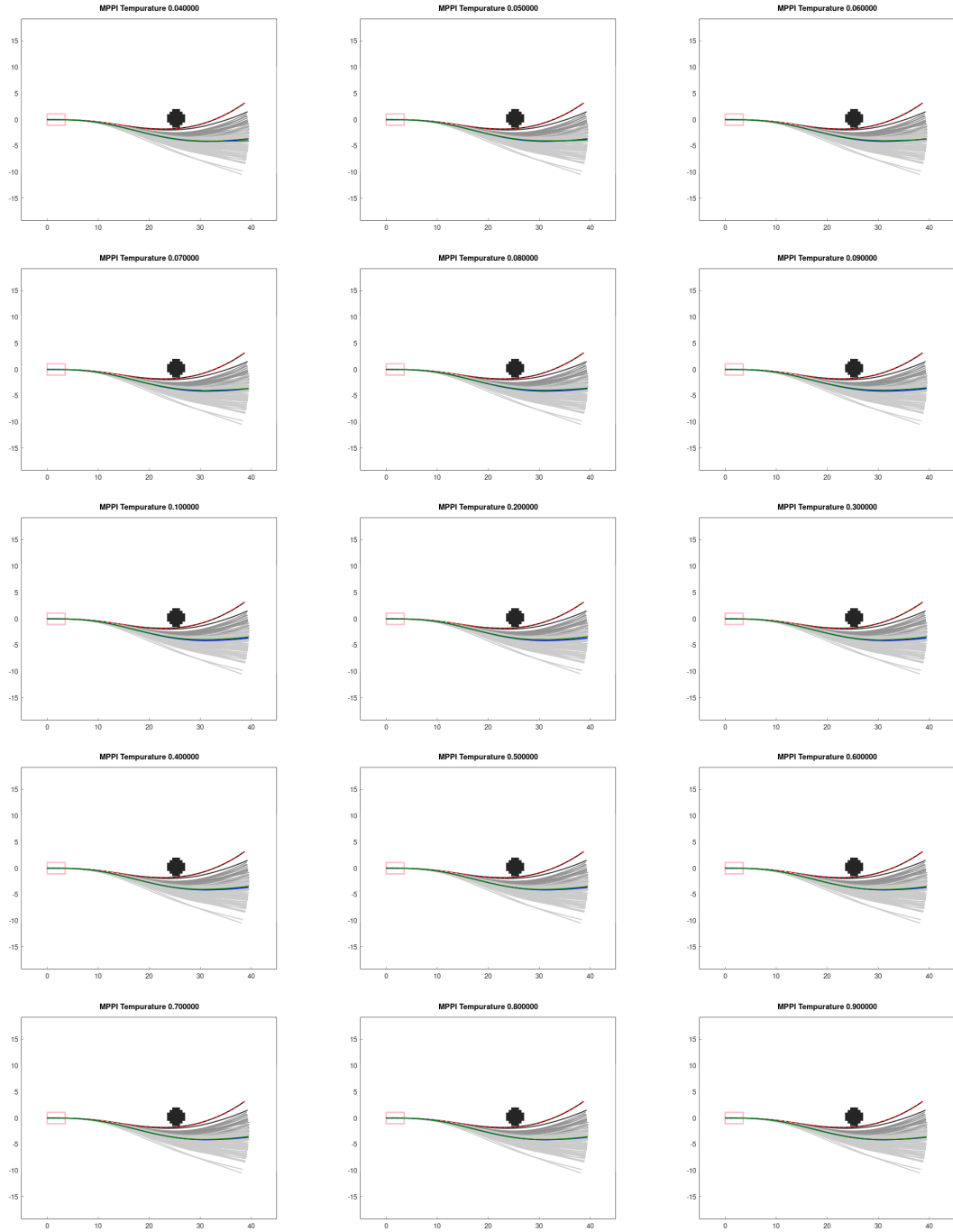
Temperature Sweep Experiments

In section 2.3.1 we show an experiment doing a sweep over the temperature (λ) parameter of MPPI to see how it effects the result. The experiment was setup with the vehicle at an initial state $x_0 = (0, 0, 0, 8, 0)$ which means it was essentially driving straight at $8m/s$. To make the comparison easier to reason through, the solver was converged with several cycles of MPPI with the same x_0 , just refining the seed with $\lambda = 0.01$. Then the temperature parameter was modified to the value in the title of each plot and an iteration of MPPI was run to generate the new result. Constructing the experiment in this way allowed the sampling to be controlled such that for each individual test the sole difference was the λ parameter, everything else (initial conditions, costmap, samples) were the same.

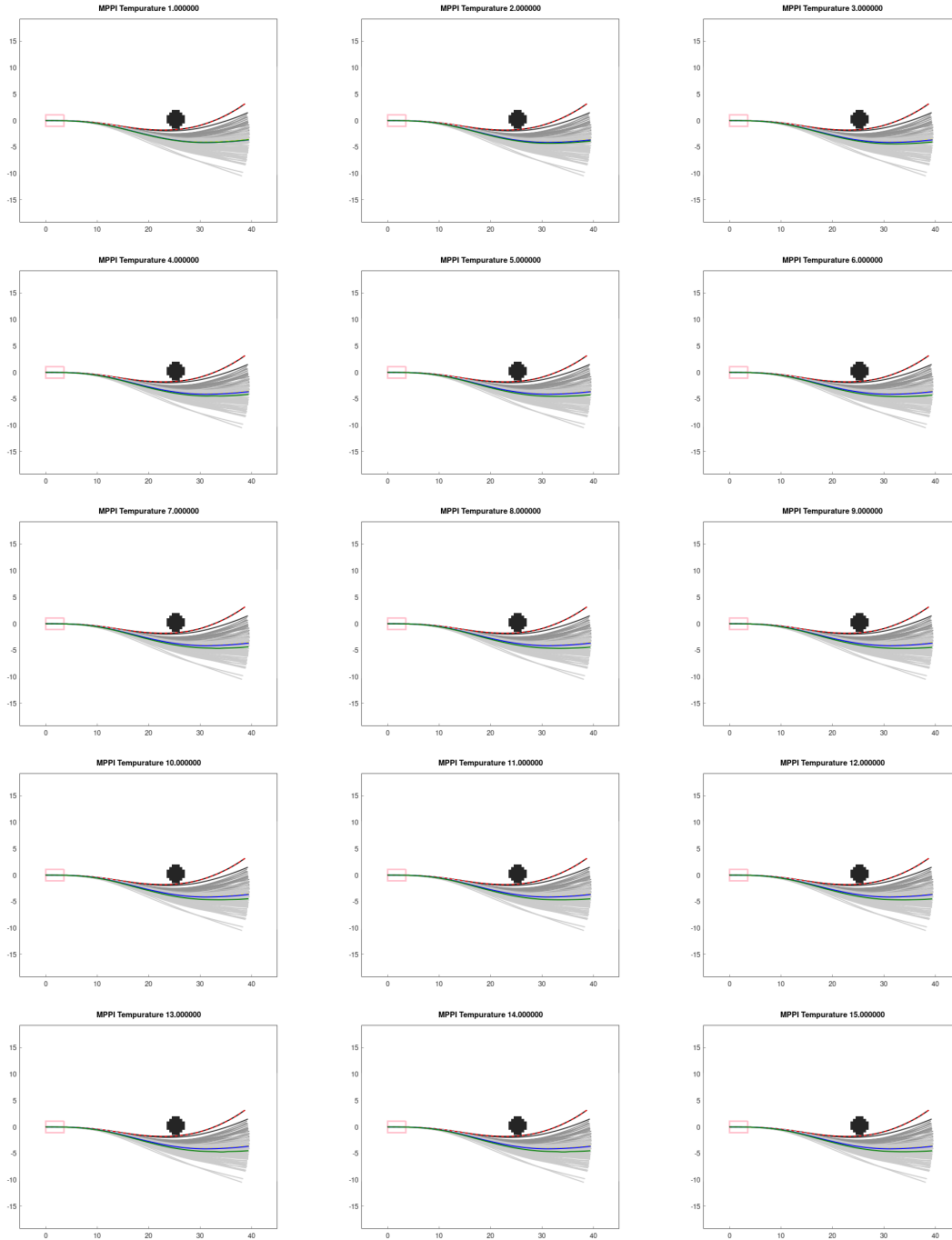
It is interesting to observe that the optimized path only begins to diverge from the lowest-cost sample when $\lambda > 1.0$.



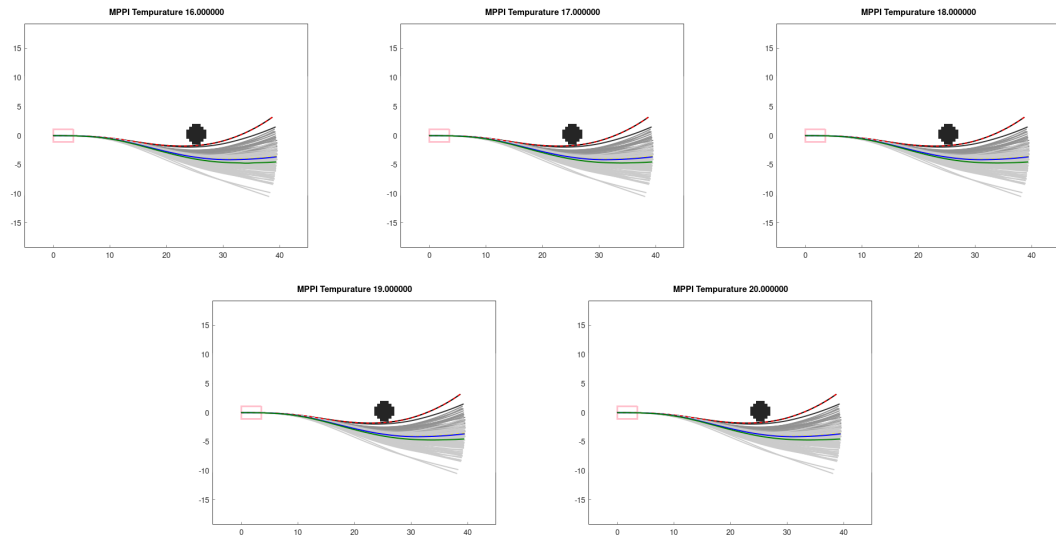
B. Temperature Sweep Experiments



B. Temperature Sweep Experiments



B. Temperature Sweep Experiments



Appendix C

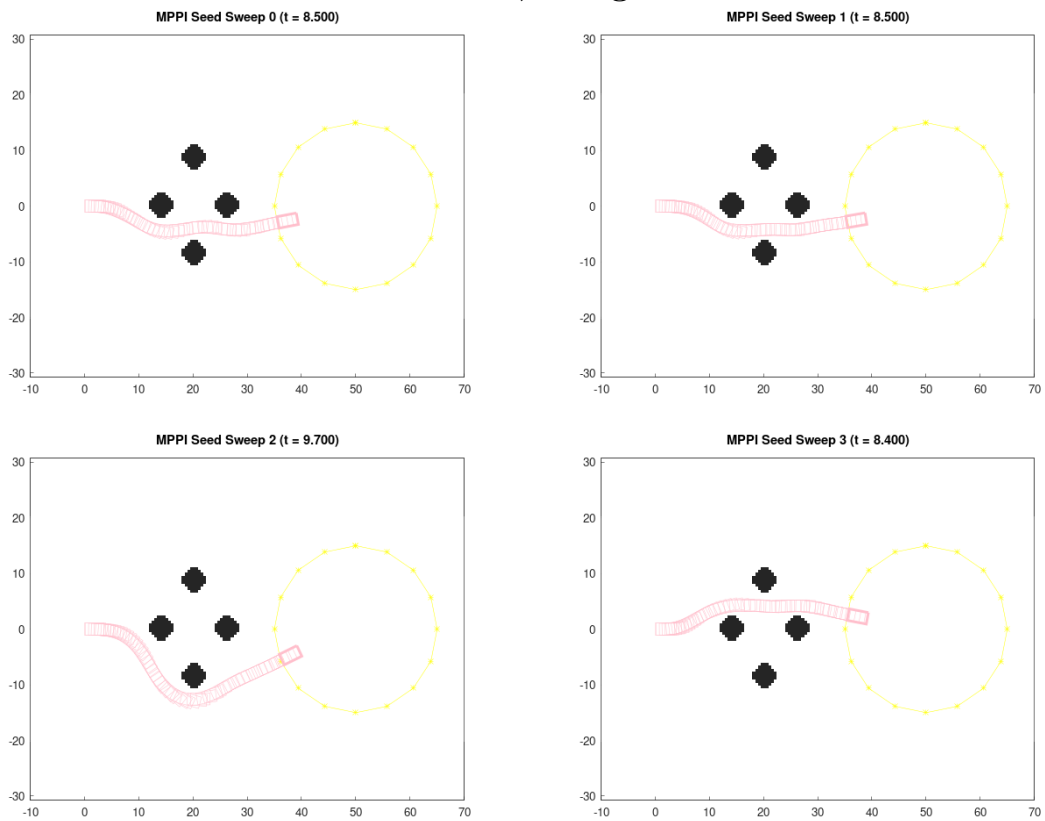
How Stochastic is Stochastic?

In section [5.3](#) we discuss how changing the seed can drastically effect the results of a navigation. We include more results from those experiments here. In each group of four photos below, the seeds 1 (top-left), 200 (top-right), 500 (bottom-left), 800 (bottom right) are used.

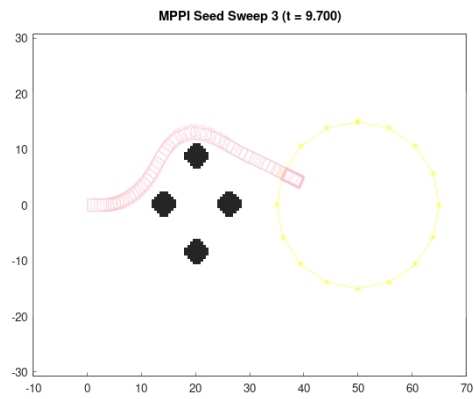
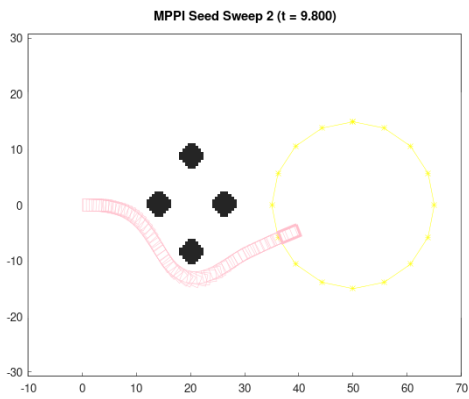
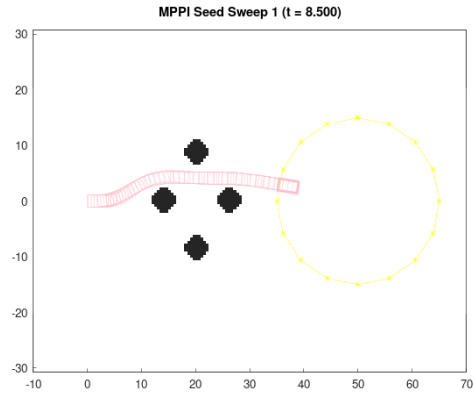
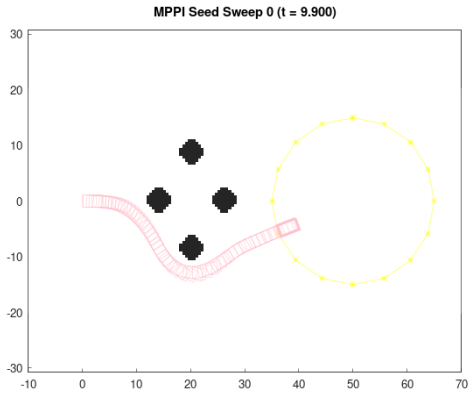
This is easily the most interesting result of this work. While this example was designed to have a few solutions with similar costs, we did not expect there to be this much variance in the results with so many particles. Suppose this example had traps or surprise obstacles hiding around corners, whether or not the solver fell into the trap could be a function of random chance rather than reliable behavior from a robust system.

C. How Stochastic is Stochastic?

MPPI With 3000 Particles, Using XORSHIFT PRNG

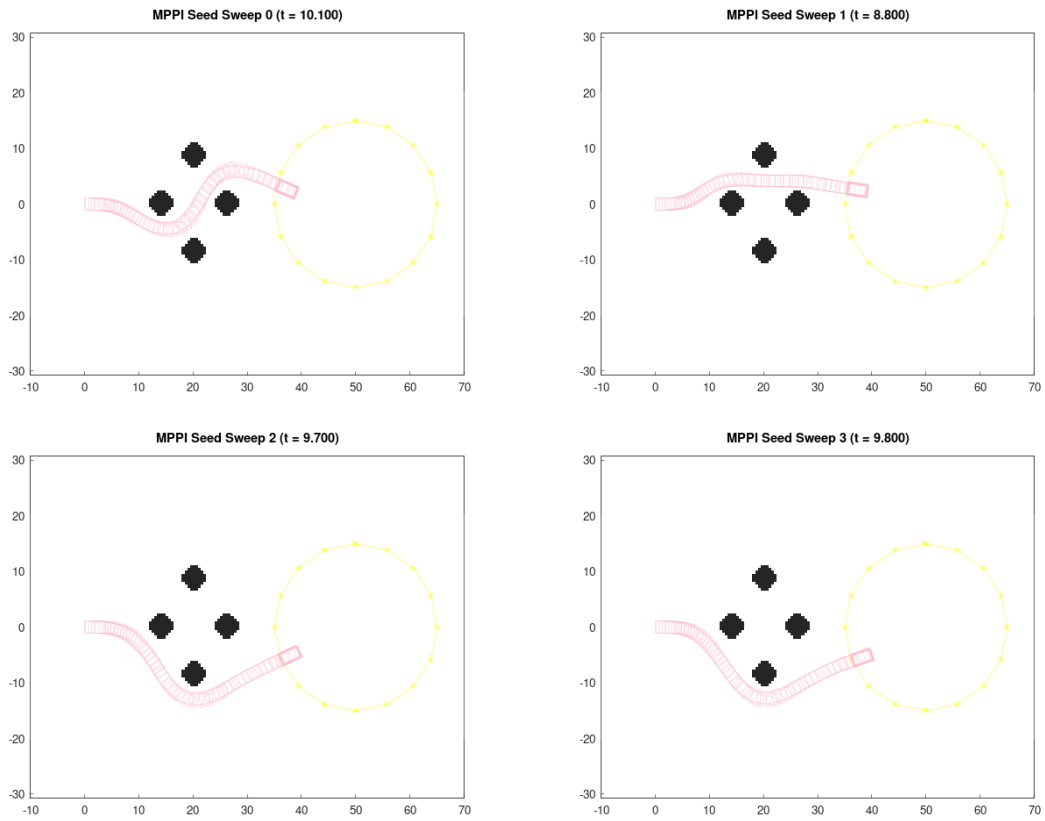


CEM With 2000 Particles, Using SPLITMIX PRNG

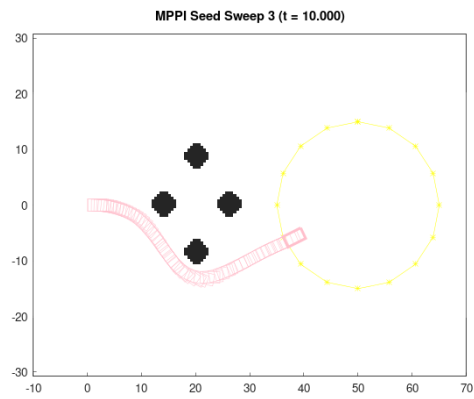
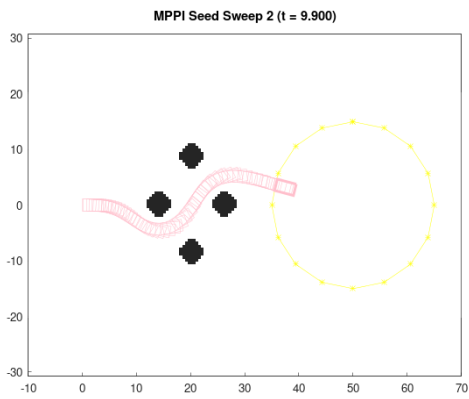
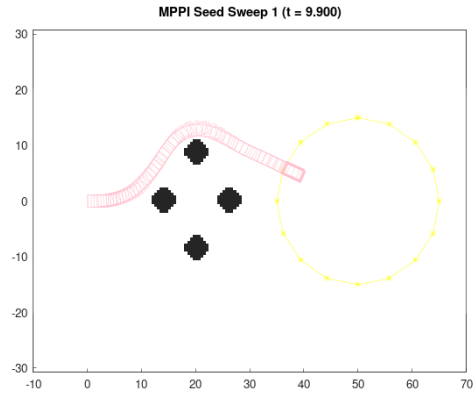
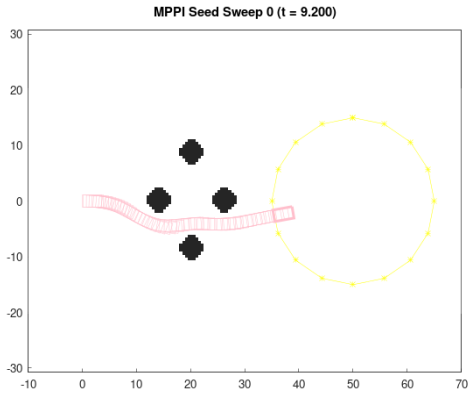


C. How Stochastic is Stochastic?

CEM With 1000 Particles, Using SPLITMIX PRNG



CEM With 500 Particles, Using SPLITMIX PRNG



C. How Stochastic is Stochastic?

Bibliography

- [1] Intel product specifications, . URL <https://www.intel.com/content/www/us/en/ark/search/featurefilter.html>. 3.1
- [2] libcu++: The C++ Standard Library for Your Entire System, . URL <https://nvidia.github.io/libcudacxx/>. 3.2.2
- [3] Andrew Bacha, Cheryl Bauman, Ruel Faruque, Michael Fleming, Chris Terwelp, Charles Reinholtz, Dennis Hong, Al Wicks, Thomas Alberi, David Anderson, Stephen Cacciola, Patrick Currier, Aaron Dalton, Jesse Farmer, Jesse Hurdus, Shawn Kimmel, Peter King, Andrew Taylor, David Van Covern, and Mike Webster. Odin: Team VictorTango’s entry in the DARPA Urban Challenge. *Journal of Field Robotics*, 25(8):467–492, 2008. ISSN 1556-4967. doi: 10.1002/rob.20248. URL <https://onlinelibrary.wiley.com/doi/abs/10.1002/rob.20248>. eprint: <https://onlinelibrary.wiley.com/doi/pdf/10.1002/rob.20248>. 4.6.1
- [4] Nathan Bell, Jared Hoberock, and Chris Rodrigues. THRUST: a productivity-oriented library for CUDA. In *Programming Massively Parallel Processors*, pages 475–491. Elsevier, 2017. ISBN 978-0-12-811986-0. doi: 10.1016/B978-0-12-811986-0.00033-9. URL <https://linkinghub.elsevier.com/retrieve/pii/B9780128119860000339>. 3.2.1
- [5] Alberto Bemporad, Tommaso Gabbriellini, Laura Puglia, and Leonardo Bellucci. Scenario-based stochastic model predictive control for dynamic option hedging. In *49th IEEE Conference on Decision and Control (CDC)*, pages 6089–6094, December 2010. doi: 10.1109/CDC.2010.5717004. ISSN: 0191-2216. 2.1.2
- [6] Daniele Bernardini and Alberto Bemporad. Scenario-based model predictive control of stochastic constrained linear systems. In *Proceedings of the 48th IEEE Conference on Decision and Control (CDC) held jointly with 2009 28th Chinese Control Conference*, pages 6333–6338, December 2009. doi: 10.1109/CDC.2009.5399917. ISSN: 0191-2216. 2.1.2
- [7] Giuseppe C. Calafiore and Lorenzo Fagiano. Robust Model Predictive Control via Scenario Optimization. *IEEE Transactions on Automatic Control*, 58(1):219–224, January 2013. ISSN 0018-9286, 1558-2523. doi: 10.1109/TAC.2012.2203054.

- URL <http://arxiv.org/abs/1206.0038>. arXiv:1206.0038 [cs, math]. 2.1.2
- [8] Mark Cannon, Basil Kouvaritakis, and Desmond Ng. Probabilistic tubes in linear stochastic model predictive control. *Systems & Control Letters*, 58(10-11):747–753, October 2009. ISSN 01676911. doi: 10.1016/j.sysconle.2009.08.004. URL <https://linkinghub.elsevier.com/retrieve/pii/S0167691109001078>. 2.1.2
- [9] Mark Cannon, Basil Kouvaritakis, and Xingjian Wu. Probabilistic Constrained MPC for Multiplicative and Additive Stochastic Uncertainty. *IEEE Transactions on Automatic Control*, 54(7):1626–1632, July 2009. ISSN 1558-2523. doi: 10.1109/TAC.2009.2017970. Conference Name: IEEE Transactions on Automatic Control. 2.1.2
- [10] Yiqun Dong, Efe Camci, and Erdal Kayacan. Faster RRT-based Nonholonomic Path Planning in 2D Building Environments Using Skeleton-constrained Path Biasing. *Journal of Intelligent & Robotic Systems*, 89(3-4):387–401, March 2018. ISSN 0921-0296, 1573-0409. doi: 10.1007/s10846-017-0567-9. URL <http://link.springer.com/10.1007/s10846-017-0567-9>. 4.6.1
- [11] David D. Fan, Kyohei Otsu, Yuki Kubo, Anushri Dixit, Joel Burdick, and Ali-Akbar Agha-Mohammadi. STEP: Stochastic Traversability Evaluation and Planning for Risk-Aware Off-road Navigation, June 2021. URL <http://arxiv.org/abs/2103.02828>. arXiv:2103.02828 [cs, eess]. 4.2.1, 4.6.1
- [12] Manan Gandhi, Bogdan Vlahov, Jason Gibson, Grady Williams, and Evangelos A. Theodorou. Robust Model Predictive Path Integral Control: Analysis and Performance Guarantees. *IEEE Robotics and Automation Letters*, 6(2):1423–1430, April 2021. ISSN 2377-3766, 2377-3774. doi: 10.1109/LRA.2021.3057563. URL <http://arxiv.org/abs/2102.09027>. arXiv:2102.09027 [cs, eess, math]. 2.4
- [13] Michael Garland. CUDA parallel programming model. In *2008 IEEE Hot Chips 20 Symposium (HCS)*, pages 1–29, Stanford, CA, USA, August 2008. IEEE. ISBN 978-1-4673-8871-9. doi: 10.1109/HOTCHIPS.2008.7476519. URL <http://ieeexplore.ieee.org/document/7476519/>. 3.2
- [14] B Kerbl, M Kenzel, M Winter, and M Steinberger. CUDA and Applications to Task-based Programming. 2022. 3.2
- [15] Sang Uk Lee, Ramon Gonzalez, and Karl Iagnemma. Robust sampling-based motion planning for autonomous tracked vehicles in deformable high slip terrain. In *2016 IEEE International Conference on Robotics and Automation (ICRA)*, pages 2569–2574, May 2016. doi: 10.1109/ICRA.2016.7487413. 4.6.1
- [16] Yijing Li. An RRT-Based Path Planning Strategy in a Dynamic Environment. In *2021 7th International Conference on Automation, Robotics and Applications*

- (*ICARA*), pages 1–5, February 2021. doi: 10.1109/ICARA51699.2021.9376472. 4.6.1
- [17] Matthias Lorenzen, Frank Allgöwer, Fabrizio Dabbene, and Roberto Tempo. Scenario-based Stochastic MPC with guaranteed recursive feasibility. In *2015 54th IEEE Conference on Decision and Control (CDC)*, pages 4958–4963, December 2015. doi: 10.1109/CDC.2015.7402994. 2.1.2
- [18] Francesco Micheli and John Lygeros. Scenario-based Stochastic MPC for systems with uncertain dynamics, July 2022. URL <http://arxiv.org/abs/2207.12517>. arXiv:2207.12517 [math]. 2.1.2
- [19] Ihab S. Mohamed, Kai Yin, and Lantao Liu. Autonomous Navigation of AGVs in Unknown Cluttered Environments: log-MPPI Control Strategy, July 2022. URL <http://arxiv.org/abs/2203.16599>. arXiv:2203.16599 [cs, eess]. 4.6.1
- [20] Shohin Mukherjee, Sandip Aine, and Maxim Likhachev. MPLP: Massively Parallelized Lazy Planning. *IEEE Robotics and Automation Letters*, 7(3):6067–6074, July 2022. ISSN 2377-3766, 2377-3774. doi: 10.1109/LRA.2022.3157544. URL <http://arxiv.org/abs/2107.02826>. arXiv:2107.02826 [cs]. 1
- [21] Kazuhide Okamoto and Panagiotis Tsiotras. Stochastic Model Predictive Control for Constrained Linear Systems Using Optimal Covariance Steering, November 2019. URL <http://arxiv.org/abs/1905.13296>. arXiv:1905.13296 [math]. 2.1.2
- [22] Melissa E O’Neill and Harvey Mudd College. PCG: A Family of Simple Fast Space-Efficient Statistically Good Algorithms for Random Number Generation. *ACM Transactions on Mathematical Software*. 5.3
- [23] François Panneton and Pierre L’Ecuyer. On the xorshift random number generators. *ACM Transactions on Modeling and Computer Simulation*, 15(4):346–361, October 2005. ISSN 1049-3301, 1558-1195. doi: 10.1145/1113316.1113319. URL <https://dl.acm.org/doi/10.1145/1113316.1113319>. 5.3
- [24] Brian Plancher and Scott Kuindersma. A Performance Analysis of Parallel Differential Dynamic Programming on a GPU. In Marco Morales, Lydia Tapia, Gildardo Sánchez-Ante, and Seth Hutchinson, editors, *Algorithmic Foundations of Robotics XIII*, volume 14, pages 656–672. Springer International Publishing, Cham, 2020. ISBN 978-3-030-44050-3 978-3-030-44051-0. doi: 10.1007/978-3-030-44051-0_38. URL http://link.springer.com/10.1007/978-3-030-44051-0_38. Series Title: Springer Proceedings in Advanced Robotics. 1
- [25] Reuven Rubinstein. The Cross-Entropy Method for Combinatorial and Continuous Optimization. *Methodology And Computing In Applied Probability*, 1(2):127–190, September 1999. ISSN 1573-7713. doi: 10.1023/A:1010091220143. URL <https://doi.org/10.1023/A:1010091220143>. 2.2

- [26] John Shalf. The future of computing beyond Moore’s Law. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 378(2166):20190061, January 2020. doi: 10.1098/rsta.2019.0061. URL <https://royalsocietypublishing.org/doi/full/10.1098/rsta.2019.0061>. Publisher: Royal Society. 3.1
- [27] Guy L. Steele, Doug Lea, and Christine H. Flood. Fast splittable pseudorandom number generators. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, pages 453–472, Portland Oregon USA, October 2014. ACM. ISBN 978-1-4503-2585-1. doi: 10.1145/2660193.2660195. URL <https://dl.acm.org/doi/10.1145/2660193.2660195>. 5.3
- [28] Sebastian Thrun, Mike Montemerlo, Hendrik Dahlkamp, David Stavens, Andrei Aron, James Diebel, Philip Fong, John Gale, Morgan Halpenny, Gabriel Hoffmann, Kenny Lau, Celia Oakley, Mark Palatucci, Vaughan Pratt, Pascal Stang, Sven Strohband, Cedric Dupont, Lars-Erik Jendrossek, Christian Koelen, Charles Markey, Carlo Rummel, Joe Van Niekerk, Eric Jensen, Philippe Alessandrini, Gary Bradski, Bob Davies, Scott Ettinger, Adrian Kaehler, Ara Nefian, and Pamela Mahoney. Stanley: The robot that won the DARPA Grand Challenge. *Journal of Field Robotics*, 23(9):661–692, September 2006. ISSN 15564959, 15564967. doi: 10.1002/rob.20147. URL <https://onlinelibrary.wiley.com/doi/10.1002/rob.20147>. 4.6.1
- [29] Bhaskar Varma, Nitin Swamy, and Sujoy Mukherjee. Trajectory Tracking of Autonomous Vehicles using Different Control Techniques(PID vs LQR vs MPC). In *2020 International Conference on Smart Technologies in Computing, Electrical and Electronics (ICSTCEE)*, pages 84–89, October 2020. doi: 10.1109/ICSTCEE49637.2020.9276986. 4.6.1
- [30] Kasi Viswanath, P B Sujit, and Srikanth Saripalli. CAMEL: Learning Cost-maps Made Easy for Off-road Driving. 4.2.1
- [31] Grady Williams, Andrew Aldrich, and Evangelos A. Theodorou. Model Predictive Path Integral Control: From Theory to Parallel Computation. *Journal of Guidance, Control, and Dynamics*, 40(2):344–357, February 2017. ISSN 0731-5090, 1533-3884. doi: 10.2514/1.G001921. URL <https://arc.aiaa.org/doi/10.2514/1.G001921>. 1, 2.3, 3, 4.6
- [32] Grady Williams, Paul Drews, Brian Goldfain, James M. Rehg, and Evangelos A. Theodorou. Information Theoretic Model Predictive Control: Theory and Applications to Autonomous Driving, July 2017. URL <http://arxiv.org/abs/1707.02342>. arXiv:1707.02342 [cs]. 2.3, 4.6.1