FORMAL VERIFICATION AND VALIDATION OF CONVEX OPTIMIZATION ALGORITHMS FOR MODEL PREDICTIVE CONTROL

A Thesis
Presented to
The Academic Faculty

by

Raphael P. Cohen

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Aerospace Engineering

Georgia Institute of Technology
May 2019

FORMAL VERIFICATION AND VALIDATION OF CONVEX OPTIMIZATION ALGORITHMS FOR MODEL PREDICTIVE CONTROL

Approved by:

Professor Eric Feron, Advisor
School of Aerospace Engineering
Georgia Institute of Technology

Dr. Pierre-Loïc Garoche, Co-Advisor
DTIS
Onera – The French Aerospace Lab

Professor Marcus Holzinger
School of Aerospace Engineering
Georgia Institute of Technology

Professor Panagiotis Tsiotras
School of Aerospace Engineering
Georgia Institute of Technology

Dr. Tim Wang
Systems Department
United Technology Research Center

Dr. César A. Muñoz
Safety Critical Avionics Systems Branch
NASA Langley Research Center

Date Approved: December 3, 2018

To my beloved grandmother, Daisy ז״ל

ACKNOWLEDGEMENTS

I would like to thank everyone who made this thesis possible and an unforgettable adventure for me. First, I would like to thank my committee members, Professor Holzinger, Professor Tsiotras, Dr. Hasnaa Zidani, Dr. Cesar Munoz and Dr. Tim Wang for accepting to be part of this PhD committee. The work presented in this thesis is at the border of multiple different areas, including optimal control, optimization theory and formal methods. Therefore having such experts on Control Systems, Optimal Control, Space Systems and Computer Science on my committee is a honor and I fully realize the chance I have.

My advisor, Dr. Eric Feron, thank you for believing in me and for offering me to work with you as your PhD student. Working next to you brought me a lot, both on a professional and a personal point of view. Your extremely sharp knowledge and your experience in research gave me, I believe, a certain maturity that I would not have acquired elsewhere and I feel privileged for this. Thank you also for all the opportunities you bring me everyday, none of this could have been possible without you. Because a complete and sound PhD cannot be performed without a little taste of teaching, I would like to thank you as well for having me as your teaching assistant. As you already know, I really enjoyed this part of the job and I look forward to do it again.

To my co-advisor Dr. Pierre-Loïc Garoche, I seem redundant but I would like to thank you as well, for believing in me and offering me to work with you at Onera in Toulouse. From the minute Professor Feron told me about this co-thesis opportunity with you, I got extremely excited. Your guidance brought me a lot and I wanted to express to you all my gratitude for this. Working with you as your PhD student was

an exciting and very resourceful journey.

This PhD thesis being performed in the frame of a cotutelle between Georgia Institute of Technology and Institut Supérieur de l'Aéronautique et de l'Espace (ISAE) in Toulouse, France, I would like to express my gratitude to all the employees from both institutions who made this Cotutelle possible.
I apologize in advance for everyone that I am going to forget but I wanted to thank all my friends at Georgia Tech, Tom , Helene, Gabriel, Elaud, Louis, Phillipe, Trey, Thunde, Corbin, Mark, Kevin, Harold, Jose, David, Giuseppe, Ayush, ...
As well, I wanted to sincerely thank everyone at Onera who welcomed me when I arrived in Toulouse and more particularly Guillaume, Hugo, Matthieu, Emanuele, Kevin, Jeanne, Oscar and many more...

Guillaume, thank you again for welcoming me in Toulouse when I arrived and for your help throughout this thesis, specially with the use of formal methods. As well, having you next to me suffering as much as me and fighting against Frama-C gave me some comfort and made me avoid a serious breakdown. I would also like to thank Pierre Roux and Arnaud Dieumegard for their help and for being always available for me.

As usual, none of this would have been possible without funding, therefore I want to thank the sponsor, the National Science Foundation (NSF) and the French Aerospace Laboratory (ONERA).

Mom, dad, Rebecca, Eva, and my dear brother in law, nephews and niece, Stéphane, Aaron, Noam and Tahel. Thank you all for the love and support you gave me. Doing this PhD thesis far away from you was hard and painful and I was missing you everyday.

Mamy Daisy, I will always remember you as this wonderful woman and loving grandmother that you were. I am sad you had to go so brutally and that you could not see any of this.

Tamara, my fiancé, my last grateful words will be for you. Your love means the world to me. Thank you for all the support you give me everyday.

Contents

List of Tables

List of Figures

LIST OF SYMBOLS

Symbol	**Description**
$\mathbb{R}$	The set of all real numbers.
$\mathbb{R}^+$	The set of all positive real numbers.
$\mathbb{R}_+^*$	The set of all stricly positive real numbers.
$\mathbb{R}^n$	The set of real vectors of length n.
$\mathbb{R}^{m\times n}$	The set of real matrices of size $m \times n$.
$\|A\|_F$	Frobenius norm of a matrix A.
$\|A\|$	Two norm of a matrix A.
$\|x\|$	Two norm of a vector x.
B_n	n-dimensionnal unit Eucliean ball. $B_n = \{z \in \mathbb{R}^n : \|z\| \leq 1\}$.
$B_r(x)$	Ball of radius r centered on x. $B_r(x) = \{z \in \mathbb{R}^n : \|z - x\| \leq r\}$.
$\mathrm{Ell}(B, c)$	Ellipsoid set defined by: $\mathrm{Ell}(B, c) = \{Bu + c : u \in B_n\}$.
Vol()	Volume of a given set.
fl()	Floating-point rounding to nearest of a given real number.
$\sigma_{max}(A)$	Largest singular value of a matrix A.
$\sigma_{min}(A)$	Smallest singular value of a matrix A.
$k(A)$	Condition number of a matrix A.
x	Plant State Vector.
N	Model Predictive Control Horizon.
$\mathbf{x}$	Collection of state vectors to horizon: $\mathbf{x} = [x_1 \dots x_N]$.
u	Plant Input.
$\mathbf{u}$	Collection of input vectors to horizon: $\mathbf{u} = [u_1 \dots u_{N-1}]$.

X	Original Decision Vector for an Optimization problem.
Z	Projected Decision Vector for an Optimization problem.
X_f	Feasible set of an optimization problem.
X_ϵ	Epsilon optimal set of an optimization problem.
	ACSL
requires	Introduces a precondition in a local contract or a function contract.
ensures	Introduces a postcondition in a local contract or a function contract.
assigns	Listing the memory being assigned in a function contract.
\result	Referring to the output of a C function.
loop invariant	Introduces a loop invariant, a property true at each iteration of the loop.
loop variant	Introduces a strictly positive and decreasing quantity in a denumerable set, usual $\mathbb{N}$. Needed to prove termination.
loop assigns	Listing the memory being assigned in a loop.
axiomatic	Defining a new ACSL Theory. Containings new types, functions, axioms, lemmas, theorems.
lemma	Defining a new lemma. Need to be proven.
theorem	Defining a new theorem. Need to be proven.
axiom	Defining a new axiom. Assumed to be true.
logic	Defining a new ACSL function.
predicate	Defining and naming a given ACSL property.

genemo Files

Constants	Introduces the section where constants can be defined.
Variables	Defines the decision variable for the optimization problem.
Input	Optional Section defining the MPC input.
Output	Optional Section defining the MPC output.
Minimize	Introduces the cost to be minimized.
SubjectTo	Introduces the constraints of the optimization problem.
Information	Introduces the scalars needed for the verification.

Acronyms

ACSL	ANSI/C Specification Language.
AST	Abstract Syntax Tree.
CPS	Cyber Physical System.
DOF	Degrees of Freedom.
HLR	High-Level Requirements
LP	Linear Programming.
LTI	Linear Time Invariant.
LLR	Low-Level Requirements.
MPC	Model Predictive Control.
QCQP	Quadratically Constrained Quadratic Program.
QP	Quadratic Programming.
RHC	Receding Horizon Control.
SDP	Semi-Definite Programming.
SMT	Satisfiability Modulo Theories.
SOCP	Second-Order Cone Programming.

SUMMARY

The efficiency of modern optimization methods, coupled with increasing computational resources, has led to the possibility of real-time optimization algorithms acting in safety critical roles. However, this cannot happen without addressing proper attention to the soundness of these algorithms.

This PhD thesis discusses the formal verification of convex optimization algorithms with a particular emphasis on receding-horizon controllers. Additionally, we demonstrate how theoretical proofs of real-time optimization algorithms can be used to describe functional properties at the code level, thereby making it accessible for the formal methods community. In seeking zero-bug software, we use the Credible Autocoding scheme. In this framework, with the use of a "Credible Autocoder", we are able to automatically generate C code implementation of Receding Horizon controllers along with its proof of soundness at code level.

We focused our attention on the Ellipsoid Method solving Second-Order Cone Programs (SOCP). Also, we present a modified version of the original Ellipsoid Method, in order to take into account and control its numerical error. Following this, a floating-point analysis of the algorithm and a framework is presented to numerically validate the method.

Chapter I

INTRODUCTION

Cyber-physical systems (CPS) regroup all the mechanical systems that interact with computer-based algorithms. A cyber-physical system can be controlled by a human being or completely autonomous. Among those systems, we call safety-critical the ones for which a failure could cause human death. Such systems include planes, trains, nuclear power plants, human spaceflight vehicles, robotic surgery machines and many more.

Designing the embedded software carried by a safety-critical cyber-physical systems is a meticulous task and should be performed with caution. In most applications, safety-critical CPS are required to follow a given certification and requirements. In the case of Airborne Systems, the DO-178C, Software Considerations in Airborne Systems and Equipment Certification, give technical guidelines and requirements for developing avionics software systems. We discuss in more detail how formal methods fit into DO-178C and the scope of this PhD thesis in section 1.3.3.

The efficiency of modern optimization methods, coupled with increasing computational resources, has led to the possibility of real-time optimization algorithms acting in safety-critical roles. However, this cannot happen without addressing proper attention to the soundness of these algorithms. This PhD thesis discusses the formal verification of convex optimization algorithms with a particular emphasis on receding-horizon controllers. We demonstrate how theoretical proofs of real-time optimization algorithms can be used to describe functional properties at the code level, thereby making it accessible for the formal methods community.

The need for enhanced safety and better performance is currently pushing for the introduction of advanced numerical methods into next generations of cyber-physical systems. While most of the algorithms described in this paper have been established for a long time, their online use within embedded systems is relatively new and opens issues that have to be addressed. Among these methods, we are concerned with numerical optimization algorithms.

In this chapter, we recall preliminaries about formal verification, semantics of programs, and the Credible Autocoding framework. Finally, we develop the state of the art and explain how this thesis fits into the previous work. In Chapter 2, we give background about convex optimization, dynamical systems and model predictive control (MPC). Chapter 3 focuses on the axiomatization of a second-order cone program and the formal verification of the ellipsoid algorithm. A floating-point analysis of a modified version of the ellipsoid method is presented in Chapter 4, while Chapter 5 gives details about closed-loop management. This chapter presents how this framework can be automated and applied to a system as well. Chapter 6 concludes this thesis.

1.1 Credible Autocoding Framework

Credible autocoding, is a process by which an implementation of a certain input model in a given programming language is being generated along with formally verifiable evidence that the output source code correctly implements the input model. Given that the mathematical proofs of high-level functional properties of convex optimization algorithms do exist, we want to translate, generate and carry them at code level. This is done by adding comments, which does not perturb the code compilation and execution. Furthermore, those comments are expressed in a formal specification language in order to be read and analyzed by other software. An illustration of this framework is given in Figure 1. In this thesis, we focus on automatically generating

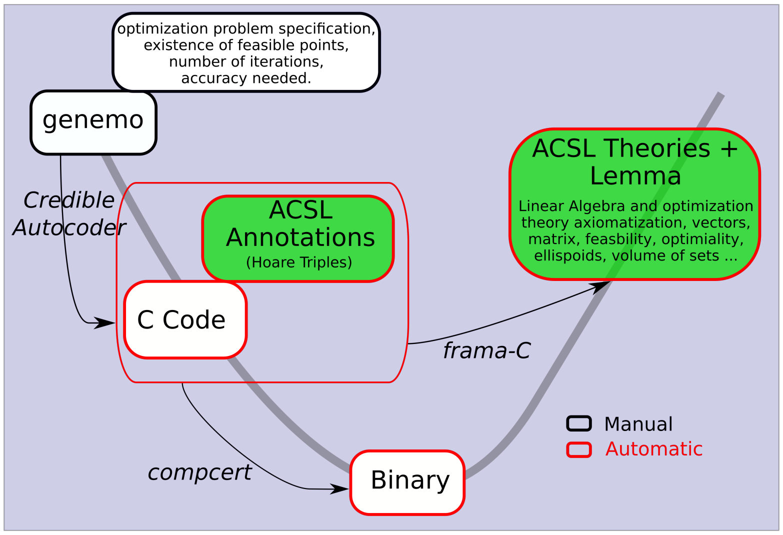

optimization problem specification,
existence of feasible points,
number of iterations,
accuracy needed.
genemo
Credible
Autocoder
ACSL
Annotations
(Hoare Triples)
C Code
ACSL Theories +
Lemma
Linear Algebra and optimization theory axiomatization, vectors, matrix, feasbility, optimiality, ellispoids, volume of sets ...
frama-C
compcert
Binary
Manual
Automatic

Figure 1: V & V Cycle for Credible Autocoding

certifiable convex optimization algorithms to implement receding horizon controllers (see Sections 2.2 and 2.3). In other word, we instantiate a given algorithm on a model predictive control (MPC) problem. This framework is similar to [32].

For that, we build an Autocoder (see Chapter 5) that will automatically generate C code implementation of Receding Horizon Controllers from a text file containing the high-level MPC formulation. Along with generating the C code, this autocoder, that we call Credible Autocoder, also generates the semantics of the corresponding algorithms and its proof of soundness at code level. From a MPC formulation specified using a given programming language (detailed in Section 5.3) written by the user, the Credible Autocoder will generate C code algorithms implementing the original MPC formulation. As it was said before, along with generating those algorithms, it also generates the proof of soundness of those algorithms at code level. Once the semantics annotated C code generated, it is checked using a software analyzer and if it is proven correct, compiled then embedded at the heart of a system's feedback.

Figure 2 gives an illustration of the corresponding toolchain.

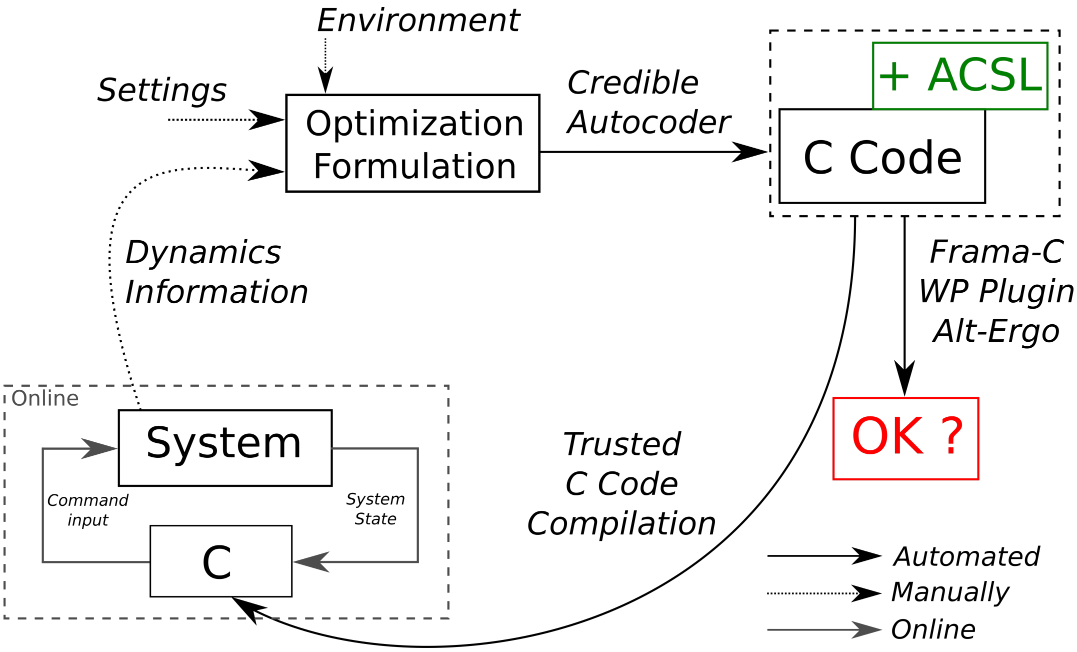


Figure 2: MPC Integration of Credible Autocoded Algorithms

1.2 Formal Verification

1.2.1 Semantics of Software

Semantics of programs express their behavior. It gives a rigorous mathematical description of the meaning of a given program. For the same program, different means can be used to specify its semantics:

- a denotational semantics, expressing the program as a mathematical function,
- an operational semantics, expressing it as a sequence of basic computations, or
- an axiomatic semantics, as a set of observations.

In the latter case, the semantics can be defined in an incomplete way, as a set of projective statements, ie. observations. This idea was formalized by [20] and then [25] as a way to specify the expected behavior of a program through pre- and post-condition, or assume-guarantee contracts.

1.2.2 Hoare Logic

A piece of code C is axiomatically described by a pair of formulas (P, Q) (see Figures 3) that if P holds before executing C, then Q should be valid after its execution. This pair acts as a contract for the function and (P, C, Q) is called a Hoare triple. In most uses P and Q are expressed as first order formulas over the variables of the program. Depending on the level of precision of these annotations, the behavior can be fully or partially specified. In our case we are interested in specifying, at code level, algorithm specific properties such as the convergence of the analysis or preservation of feasibility for intermediate iterates. Software frameworks, such as the Frama-C platform [16], provide means to annotate a source code with these contracts, and tools to reason about these formal specifications. For the C language, ACSL [4], (ANSI C Specification language) can be used as source comments to specify function contracts, or local annotations such as loop invariants. Local statements annotations act as cuts in proofs and are typically required when analyzing loops. Figure 4 shows an example of a Hoare triple. In this example, if the triple is indeed valid, we can conclude that assuming $0 \leq x \leq N$ is true, then after executing the command $x := x + 1$ we know that the property $1 \leq x \leq N + 1$ will be true. Figure 5 displays another example of a Hoare triple. The two examples presented show correct Hoare triples and have been proven using Frama-C. ACSL also give means to introduce function contract. Those contracts consist in listing all the properties that we assume to be true before the execution of the function and all the properties that will be true after its execution. For this, we use the keywords **requires** and **ensures**. For every function contract, we will then try to prove that, after the execution of a function, all the properties

$$\{\mathrm{P}\}\ \mathrm{C}\ \{\mathrm{Q}\}$$

Figure 3: Hoare Triple

ACSL + C

```
//@ assert 0 <= x <= N;
x = x + 1;
//@ assert 1 <= x <= N + 1;
```

Figure 4: Hoare Triple Expressed in ACSL Example 1

ACSL + C

```
//@ assert -2 <= x <= 2;
y = x * x;
//@ assert 0 <= y <= 4;
```

Figure 5: Hoare Triple Expressed in ACSL Example 2

listed following a **ensures** keyword will be true assuming that all the properties listed within a **requires** were true before its execution. The part of the memory assigned by a function can also be specified using the keyword **assigns**. Checking the memory assignment is very important and could lead to checking errors if it is omitted.

Let us take a look at a simple and specific example. In Figure 6, we show an implementation of a function, “addInt”, that takes two integers as inputs and store the addition of those two integers at a given memory address, referenced as an input pointer. Also, this function returns the corresponding addition. Let us assume the user only specifies that the output corresponds to the sum of the input integers. Then, while proving the second property at line 16, the SMT solvers would assume the value of *a* did not change and is correct. Therefore, missing an assigns clause could lead to logic issues and checking errors. Although Frama-C rises a flag reminding the user that a assign clause is missing, it still lacks the level of confidence we would like to achieve. In order to get around this, we make sure we write an assigns clause for every function contract.

Let us say that the user did write an assigns clause but a non correct one. Could this lead to logical issues? The answer is no. Indeed, the solvers might proved correctness for properties that are not correct but they will not be able to prove correctness for

ACSL + C

```
 1 #include <stdio.h>
 2
 3 /*@
 4   @ ensures \result == a + b;
 5 */
 6 int addInt(int *x, int a, int b){
 7     int add = a + b;
 8     *x = add;
 9     return add;
10 }
11
12 int main(){
13   int a = 0;
14   /*@ assert a == 0 ; */
15   addInt(&a, 5, 5);
16   /*@ assert a == 0 ; */
17   return 0;
18 }
```

Figure 6: Memory Assignment Issue Example in ACSL

the original assigns statement. Because of this, the user will therefore understand that there is something wrong in the implementation or ACSL annotations. The corrected contract is presented in Figure 7. In this case, the property at line 18 cannot be proved correct as expected. A second example is shown in Figure 8 where we present a function contract expressed in ACSL for a C code function implementing the square function. In this case, the corresponding function does not modify any global variable stored in memory.

ACSL + C

```
 1 #include <stdio.h>
 2
 3 /*@
 4   @ ensures \result == a + b;
 5   @ ensures *x == a + b;
 6   @ assigns *x;
 7 */
 8 int addInt(int *x, int a, int b){
 9     int add = a + b;
10     *x = add;
11     return add;
12 }
13
14 int main(){
15   int a = 0;
16   /*@ assert a == 0 ; */
17   addInt(&a, 5, 5);
18   /*@ assert a == 0 ; */
19   return 0;
20 }
```

Figure 7: Corrected Function Contract For Memory Assignment Checking

ACSL + C

```
/*@
  @ requires -2 <= x <= 2;
  @ ensures \result == x*x;
  @ ensures 0 <= \result <= 4;
  @ assigns \nothing ;
*/
double square (double x){
  return x*x;
}
```

Figure 8: ACSL Function Contract Example

1.2.3 Weakest Precondition

In this section we give details about how this process can be automated at code level. Meaning that, how can the correctness of a Hoare triple be checked automatically. For this, we present how to prove a Hoare triple by weakest precondition. Let Q be a assertion and C a command. We define the weakest precondition of the couple (C, Q) as the weakest assertion P such that the triple $\{P\}\ C\ \{Q\}$ is valid. For two assertions P_1 and P_2, P_1 is weaker than P_2 means that $P_2 \implies P_1$. The advantage of proving Hoare triple by weakest precondition lies in the fact that this latter step is easily automated. Indeed, the weakest precondition of a couple (C, Q), depending of the nature of the command C, follows a simple rule. Once this weakest precondition $wp(C, Q)$ has been computed, we then check if the property, $P \implies wp(C, Q)$, is true. If the last property is indeed true, using Theorem 1 we conclude that the triple $\{P\}\ C\ \{Q\}$ is valid. Theorem 1 expresses the fact that $wp(C, Q)$ represents the

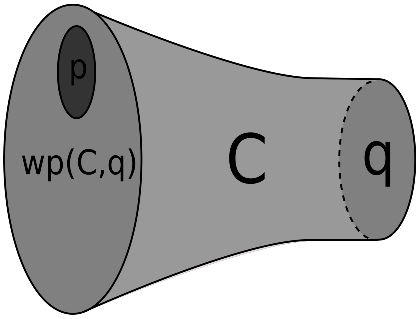


Figure 9: Weakest Precondition Illustration

weakest acceptable precondition, such that $\{wp(C, Q)\}\ C\ \{Q\}$ is valid. An illustration of this theorem is shown in Figure 9.

Theorem 1 *If the assertion below is true:*

$$P \implies wp(C, Q),$$

then, we can conclude that the triple $\{P\}\ C\ \{Q\}$ *is valid.*

1.3 State of the Art and Scope of the Thesis

1.3.1 Real-time Optimization Based Control

Formal verification of convex optimization algorithms used online within control systems is the sole focus of this research. Recently, such algorithms have been used online with great success for the guidance of systems within safety-critical applications, including, autonomous cars [18, 41, 28] and reusable rockets [3, 6]. The latter case has resulted in spectacular experiments, including landings of SpaceX's Falcon 9 and BlueOrigin's New Shepard. In order to perform the landing of the Falcon 9, the spaceX's teams used convex optimization technique [5]. The landing problem was formulated as a Quadratic Programming (see Section 2) and ran during the descent phase of the Falcon 9 rocket. In addition, automatic code generation has been used for numerical optimization, with the use of the code generator for convex optimization CVXGEN [32].

Thus, powerful algorithms solving optimization problems are already used online, have been embedded on board, and yet still lack the level of qualification required by civil aircraft or manned rocket flight. Automatic code generation for solving convex optimization problems has already been done [32, 33]. In these articles, the authors present a new code generator for embedded convex optimization. The generated code is said to be library-free, auto sufficient and without failure. Those algorithms are very complex and arguing that they do not contains any bug could be delicate, especially for safety-critical conditions. Also, this work does not include the use of formal methods.

Likewise, work within the field of model predictive control already exists where numerical properties of algorithms are being evaluated [39]. This paper, [39], proposes a dual gradient projection (DGP) algorithm implemented on fixed-point hardware. A convergence analysis with taking into account round-off errors due to fixed-point arithmetic is presented. Following this, the minimum number of integer bits that guarantee convergence to a solution can be computed, minimizing the computational power. Nevertheless, this work is only valid for Quadratic Programming and using fixed-point numbers. Additionally, no formal verification was performed.

Similarly, as it is presented at [27], MPC controllers have been successively implemented on FPGAs, letting the possibility to ran some problems at megahertz rate. In this article, the authors focus on the alternating direction method of multipliers (ADMM) and perform a numerical analysis in fixed-pint arithmetic.

1.3.2 Formal Verification of Control Systems Software

Since the 60s, computer scientists were studying ways of analyzing programs. For this, different techniques were proposed based on mathematical supports. Those methods, developed for analyzing software, were based on mathematical descriptions of the programs behaviors (i.e. its semantics): a formal description. formal methods represents an advantage over testing because they guarantee the correction of properties and soundness of programs for a wide range of inputs and under certain known hypothesis. When testing programs, only certain test cases are executed, giving satisfactions when no run-time errors are detected. Unfortunately, the absence of run-time errors does not imply the correction of the program for other test cases, as close as they can be.

We shortly present in the following paragraphs, the different formal methods developed and successfully used to analyze embedded safety-critical software.

Abstract interpretation is one the most successful method developed towards the static analysis of programs. It was first proposed in the 70s. In practice, it is mainly used in order to compute numerical invariants over programs. For this, abstract interpretation uses results from set theory to compute over approximations of program behaviors. Abstract domains are used and depending on the programs nature, different shapes and geometries are considered, representing trade offs between accuracy and performance. Definitions and the use of the main abstract domains are detailed in [14, 15, 34, 35]. In 2010, a remarkable advanced was made towards formal verification of safety-critical control system using abstract interpretation by successfully proving the absence of runtime error of the flight control system of the Airbus A380 [29].

On the other hand, another technique, called Satisfiability Modulo Theories (SMT), is growing in the area of formal verification for control system. SMT solvers are decision problems for logical formulas with respect to combinations of background theories expressed in first-order logic such as linear real/integer arithmetic. Roughly speaking, those SMT solvers are deciding procedures for the satisfiability of conjunctions of items, where an item is an atomic formula. SMT solvers handle sub-formulas by performing case analysis, which is the technique used in most automated deduction tools. SMT-solvers are used as back-end reasoning engines in a wide range of formal verification applications, such as model checking, test case generation, static analysis, etc. In this PhD thesis, we focus on the SMT solver Alt-Ergo connected to the software analyzer back-end Frama-C.

Contributions for higher-level properties have been made concerning formal verification of control systems [19, 13, 24, 44]. Those articles mainly focus on formal verification and code generation for linear control system and typical feedback control techniques. The authors show how theoretical system level properties (closed-loop Lyapunov stability) can be carried out from a high level programming language like Simulink all the way to code through automatic code generation. Following this, research has also been made toward the verification of numerical optimization algorithms [48], yet it remains purely theoretical and no proof was actually performed with the use of formal methods.

1.3.3 Formal Verification And Software Certification

In this section, we explain the differences between formal verification, testing and certification. The same way it is for Cyber-Physical System, we qualify a software of safety-critical if its failure can lead to catastrophic consequences. Therefore the correctness is a crucial issue in the design process of safety-critical software. For those systems, a higher authority is usually present and act as a supervisor checking software quality. For airborne systems many authority exists such as the FAA or EASA. Certification is the legal recognition by a certification authority that a product fulfills specific requirements.

Software verification regroups all the computer science techniques that aim to show that a piece of software complies with predefined properties and that does not perform unexpected behavior. Until recently, the high majority of software verification performed by companies while developing software consisted of reviewing and testing. As it is recalled in [21], 30% to 50% of the software costs are dedicated to testing.

Unfortunately, detecting bugs by reviewing and testing is very limited, especially for safety-critical applications. Testing can only cover certain scenarios, and bugs can still be present even when a software passes all the test cases. Also, reviewing does not represent the perfect option because the reviewers might embrace the same logic as the programmer while reading the code and fall into the same mistakes. The reviewer's mind gets corrupted by adopting the programmer's point of view. For airborne software, the the legal authorities such as the FAA, EASA, follow the guidelines detailed in the document "DO-178C, Software Considerations in Airborne Systems and Equipment Certification" [37].

DO-178C Software Requirements: In the DO178C, it is explained that each part of the software is assigned a specific software level related to the criticality of the software. Software levels are assigned from levels A to E, A corresponding to the most critical piece of software. Thus, a level A software failure can lead to tremendous catastrophes. One on the requirement listed in the DO-178C requires that each line of code must be directly traced to a requirement. Also it specifies that this requirement needs to be traced to a corresponding test case.

Roughly speaking, the DO-178C decomposes the software design phase into three processes:

- the software planning,
- the software development,
- and the integral.

The software planning defines and manages all the software development-related activities. During the software development process, High-Level Requirements (HLR), software architecture and Low-Level Requirements (LLR) are defined. The software design, coding and integration are being performed during this process as well. The integral process addresses the verification. It aims to guarantee the correctness, control, and confidence in the safety and reliability of all previous lifecycle processes. More details on DO-178B and DO178C can be found at [21, 1, 37]

For each processes DO-178C gives requirements and guidelines on how formal methods can be incorporated. Also it specifies that formal methods cannot be used as a replacement for any existing requirements. The use of formal methods for verification purposes should be explained and fixed during the software planning process. All the assumption made during the formal modeling and verification should be specified.The properties aiming to be verified, the formal verification work plan and objectives should be detailed and listed beforehand. DO-178C specifies the repeatability characteristic that the formal verification should fulfill.

Contribution: This PhD thesis discusses the formal verification of convex optimization algorithms with a particular emphasis on receding-horizon controllers. We demonstrate how theoretical proofs of real-time optimization algorithms can be used to describe functional properties at the code level, thereby making it accessible for the formal methods community.
This PhD thesis only dedicated work towards the generation of C code and checking its correctness. Indeed, the compiling process and its verification are not part of this PhD thesis scope. Work has previously been done towards those issues [31]. In this PhD dissertation, we will present the following scientific contributions:

- the axiomatization of optimization problems using the specification language ACSL (mathematical properties of linear algebra, set theory and optimization theory were defined),

- the formalization of the algorithm's proof using this same language at code level, where ACSL annotations and Lemmas were formalized and proved,

- a modification of the original Ellipsoid Algorithm in order to account for numerical errors along with its complete numerical analysis.
- the development of a high-level optimization parser using a specific and predefined language (detailed in section 5).
- the development of an Autocoder (detailed in section 5) generating C code implementation of Convex Optimization Algorithms along with ACSL annotations

Chapter II

OPTIMIZATION BASED CONTROL OF LTI SYSTEMS

Optimization based control deals with the problem of finding the control law for a given system such that a certain optimality criterion is achieved. In order to achieve this goal, two methods can be used: direct and indirect. In the latter case, we seek an analytical and closed form solution for the control input with the use of mathematical principles such as the Pontryagin maximum principle (PMP). The most famous result of this theory being the Riccatti equation, giving a closed form solution to a controller minimizing a quadratic cost on the inputs and states for a linear plant. Further information about indirect methods can be found at [8, 47]. In this thesis, we focus on direct methods, where we usually work with discretized systems and express the optimal behavior of a system by formulating an optimization problem. This optimization problem is then solved using numerical solvers and the optimal input sequence is found. In this chapter we recall first the setting of LTI theory and some stability results of LTI systems. Then, more details about convex programming and receding horizon controllers are given.

2.1 Linear Time Invariant (LTI) Systems

2.1.1 LTI Theory and State Space Realization

Given a system, a state-space representation is a mathematical model of a physical system as a set of inputs, outputs and state variables related by first-order differential equations.

In a general setting, a dynamical system can be represented by the system of equations 1. We write the state vector x, the input vector u and the output vector y.

$$\begin{cases} \dot{x} = f(x, u, t) \\ y = h(x, u, t) \end{cases} \tag{1}$$

Among dynamical systems, autonomous time-invariant systems are very important and represent a good trade-off between being simple enough to analyze but complicated enough to capture a wide variety of systems. For those systems, many stability results exist. In a general setting, an autonomous time-invariant system can be represented by the system of equations 2.

$$\begin{cases} \dot{x} = f(x) \\ y = g(x) \end{cases} \tag{2}$$

For LTI systems, the functions f and h are required to be linear in the input and state and to be independent of the time. Thus, LTI systems can be represented by the system of equations 3.

$$\begin{cases} \dot{x} = Ax + Bu \\ y = Cx + Du \end{cases} \tag{3}$$

Consequently, a LTI system that is additionally autonomous can be represented by equation 4.

$$\dot{x} = Ax \tag{4}$$

2.1.2 Stability of LTI Systems

When analyzing dynamical systems or designing controllers, studying stability is a crucial characteristic. In most cases, we study the stability of a closed-loop system. Meaning that the controller has already been designed and the loop closed (see Figure 10). That way, the system in consideration is now autonomous and do not depend on any input. Let us recall the definition of an equilibrium point. A point x_e is an

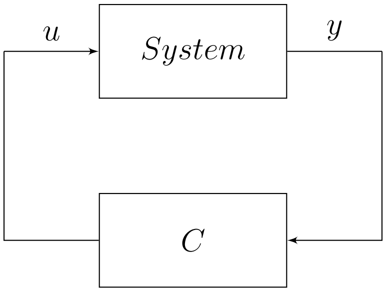


Figure 10: Closed-Loop System

equilibrium for the autonomous system 2 if $\dot{x_e} = f(x_e) = 0$. For autonomous LTI systems, the point $x_e = 0$ is always an equilibrium. In term of stability, each equilibrium can be characterized as stable, asymptotically stable or unstable.

Definition 1 (Equilibrium Stability) *The equilibrium point x_e of the system 2 is:*

- *stable if,*

$$\forall \epsilon > 0, \exists\ \delta > 0 \ \ \textit{such that:} \ \ \|x(0) - x_e\| \leq \delta \implies \|x(t) - x_e\| \leq \epsilon \quad \forall t \geq 0,$$

- *unstable, if it is not stable,*

- *asymptotically stable if,*

$$\forall \epsilon > 0, \exists\ \delta > 0 \ \ \textit{such that:} \ \ \|x(0) - x_e\| \leq \delta \implies \|x(t) - x_e\| \leq \epsilon \quad \forall t \geq 0,$$

$$\textit{and} \ \ \lim_{t \to \infty} x(t) = x_e.$$

We state now the Lyapunov Stability theorem which gives means to conclude on system stability under existence of a continuously differentiable function that we call a “Lyapunov function”.

Theorem 2 (Lyapunov Stability) *Let x_e be an equilibrium point for the system 2 and $D \in \mathbb{R}^n$ be a domain containing x_e. Let $V : D \to \mathbb{R}$ be a continuously differentiable function such that:*

$$V(x_e) = 0 \ \ , \ \ V(x) > 0 \ \forall x \in D \backslash \{x_e\},$$

$$\textit{and} \;\; \dot{V}(x) \leq 0 \;\; \forall x \in D$$

Then x_e is stable. Moreover, if

$$\dot{V}(x) < 0 \;\; \forall x \in D \backslash \{x_e\},$$

Then x_e is asymptotically stable.

For autonomous LTI systems, the results we have are even stronger than this last theorem. We know that we only have to look for quadratic Lyapunov functions. Furthermore, we know that the solution of a LMI could give us a proof of stability.

Theorem 3 *Suppose there exists a symmetric matrix $P = P^T \in \mathbb{R}^{n \times n}$ such that*

$$P > 0$$

$$A^T P + PA < 0$$

Then the system 4 is asymptotically stable.

In this section, we recalled definitions and results known about a subclass of dynamical systems. As it was said earlier, when designing a controller and closing the loop, studying the stability of the resulting system, is the first and most crucial job of the controls engineer. Further details and proof about control feedback system can be found at [38, 12]. In our case, because we focus on model predictive control (MPC), the controller carries an optimization algorithm and is therefore a complicated entity. Nevertheless, having closed-loop stability guarantees cannot be avoided and details regarding this manner are discussed in Section 2.3.1. Additionally, we clarify the technical differences between MPC and Path-planning and explain why confusing those two techniques could be an issue for our application.

2.2 Convex Optimization Problems

Optimization algorithms solve a constrained optimization problem, defined by an objective function, the cost function, and a set of constraints to be satisfied:

$$\begin{aligned} \min \quad & f_o(x) \\ \text{s.t.} \quad & f_i(x) \leq b_i \text{ for } i \in [1, m] \end{aligned} \tag{5}$$

This problem searches for $x \in \mathbb{R}^n$, the optimization variable, minimizing $f_o \in \mathbb{R}^n \to \mathbb{R}$, the objective function, while satisfying constraints $f_i \in \mathbb{R}^n \to \mathbb{R}$, with associated bound b_i. An element of $\mathbb{R}^n$ is feasible when it satisfies all the constraints f_i. An optimal point is defined by the element having the smallest cost value among all feasible points. An optimization algorithm computes an exact or approximated estimate of the optimal cost value, together with one or more feasible points achieving this value. A subclass of these problems can be efficiently solved: convex problems. In these cases, the functions f_o and f_i are required to be convex [10]. When optimization algorithms are used offline, the soundness of their implementation and the feasibility of the computed optimizers is not as critical and solutions could be validated a posteriori [43].

Here, we only present a specific subset of convex optimization problems: Second-Order Cone Programs. For $x \in \mathbb{R}^n$, a SOCP in standard form can be written as:

$$\begin{aligned} \min \quad & f^T x \\ \text{s.t.} \quad & \|A_i x + b_i\|_2 \ \leq \ c_i^T x + d_i \text{ for } i \in [1\ ,\ m] \end{aligned} \tag{6}$$

$$\text{With: } f \in \mathbb{R}^n,\ A_i \in \mathbb{R}^{n_i \times n},\ b_i \in \mathbb{R}^{n_i},\ c_i \in \mathbb{R}^n,\ d_i \in \mathbb{R}.$$

The focus of this PhD thesis is the online use of Convex Optimization algorithms. Model predictive control or real-time based control is more general and the convex aspect is not necessary in general. When optimization problems that are not convex are being solved, in the general case, there is no guarantee that the global minimizer

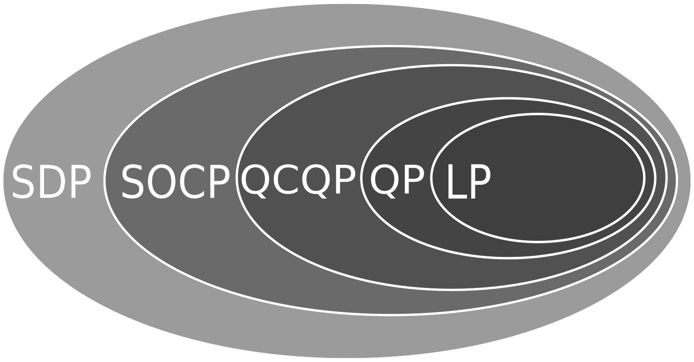


Figure 11: Classification of Some Convex Optimization Problems

is indeed being computed. For this reason, and the fact that we are targeting safety-critical applications, we chose to focus on the online use of convex optimization problems. A classification of the most famous convex optimization problems is presented in Figure 11. A Linear Program (LP) is a convex optimization problem for which both the cost and the constraints functions are linear functions. A convex optimization problem that has linear constraints but a quadratic cost is called a Quadratic Problem (QP). When both the cost and constraints functions are quadratic, we call the corresponding optimization problem a quadratically constrained quadratic program (QCQP). Second order cone program SOCP can be defined as the minimization of a linear objective function over the second-order cone and Semidefinite programming (SDP) is concerned with the minimization of a linear objective function over the cone of positive semidefinite matrices. Frequently, optimization problems that are used online for control systems can be formulated as a SOCP. Therefore, only dealing with SOCP is legitimately not constraining for the application we are targeting. In control systems, SDP's are mostly used off-line, a priory checking system's stability [9].

2.3 Real-time Convex Optimization Based Control and MPC

Model Predictive Control (also known as receding horizon control) is an optimal control strategy based on numerical optimization. In this technique, a dynamical model of the plant is being used to predict potential future trajectories. Also, a cost function J, that depends on the potential future control input and state, is being considered over the receding prediction horizon N and the objective here is to minimize this cost J. At each time t, a convex optimization problem where J has to be minimized, is being solved. From the solution of this problem, an optimal trajectory starting at $x(t)$ is being calculated and the control input sent to the plant corresponds to the first input of this optimal trajectory. A time step later, at $t + \Delta t$, the exact same process occurs and is repeated until final time. As an example, we present in problem 7 an example of a MPC formulation and an illustration of this process in Figure 13.

$$\begin{array}{ll} \underset{\mathbf{x},\mathbf{u}}{\text{minimize}} & \sum_{k=1}^{N} J(x_k, u_k) \\ \text{subject to} & x_{k+1} = Ax_k + Bu_k, \quad \forall k \in [1, N] \\ & x_k \in \mathcal{X}, u_k \in \mathcal{U}, \quad \forall k \in [1, N] \\ & x_1 = x(t) \end{array} \tag{7}$$

At each iteration, we aim to solve problem 7. For a given step k, we write

$$\begin{bmatrix} \hat{u_{1,k}} & \hat{u_{2,k}} & \dots & \hat{u_{N-1,k}} \end{bmatrix}$$

the optimal input sequence found by the algorithm. The resulting input sent to the plant is therefore $\hat{u_k} = \hat{u_1}$ (Figure 12).

When using online convex optimization techniques for the guidance of system, we usually account for the plants dynamics by constraining the trajectories. It is done with the help of equality constraints and therefore, only linear dynamics can be captured. For this reason, throughout this PhD thesis, we consider only linear dynamics.

Spring-Mass System As an example, we develop a MPC controller to control a spring-mass system, shown in Figure 14. The state vector x regroups the position z

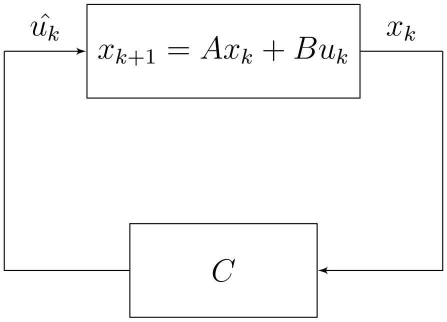


Figure 12: MPC Closed-Loop System

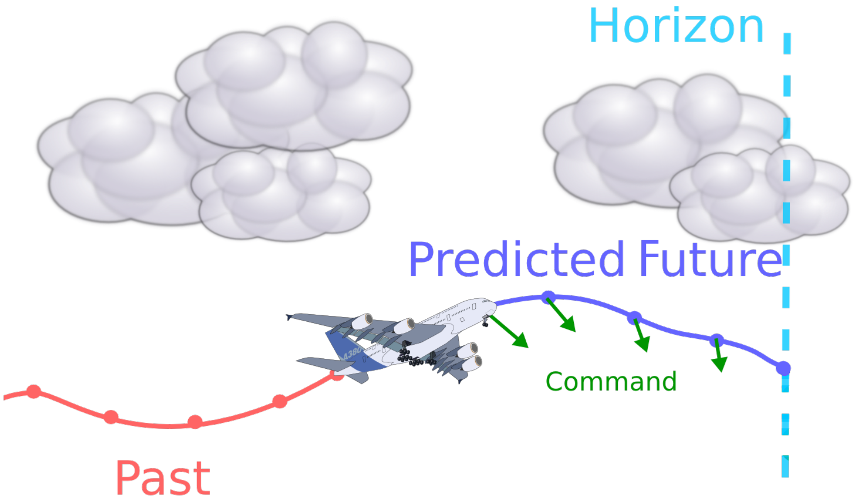


Figure 13: MPC Technique Illustration

and the velocity $\dot{z}$ of the system. The input u corresponds to a force applied on the system. A state space realization of this system is presented in equations 8.

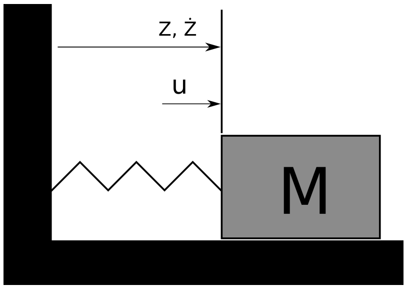


Figure 14: Spring-Mass System

$$\begin{cases} \dot{x} = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} x + \begin{bmatrix} 0 \\ 1 \end{bmatrix} u \\ y = \begin{bmatrix} 1 & 0 \end{bmatrix} x \end{cases} \tag{8}$$

We want to apply the MPC controller detailed in problem 9. The initial point of the trajectory being the point $x_o = \begin{bmatrix} 2 & -1 \end{bmatrix}^T$.

$$\begin{aligned} \underset{\mathbf{x},\mathbf{u}}{\text{minimize}} \quad & \sum_{k=1}^{N} \|Q \cdot x_k\| \\ \text{subject to} \quad & x_{k+1} = Ax_k + Bu_k, \quad \forall k \in [1, N-1] \\ & \|u_k\| \leq 5 \quad \forall k \in [1, N-1] \\ & x_1 = x(t) \end{aligned} \tag{9}$$

After discretizing the system at $T = 0.01$ sec (10 Hz) and for a MPC horizon of $N = 10$ we performed a simulation of the closed-loop system. Figures 15, 16 and 17 show the resulting position, velocity and input force. We used the constants below:

$$Q = \begin{bmatrix} 5 & 0 \\ 0 & 1 \end{bmatrix} \;\; ; \;\; A = \begin{bmatrix} 1 & 0.01 \\ -0.01 & 1 \end{bmatrix} \;\; ; \;\; B = \begin{bmatrix} 0 \\ 0.01 \end{bmatrix}.$$

The matrix Q represents the weight associated with each components of the state vector. The main inconvenient of using indirect methods is the fact that a closed-loop solution is mathematically complex to found, particularly when having constraints on the control and the state. When using Receding Horizon Control one can use arbitrary cost function and constraints in the MPC formulation. When the cost and the constraints are convex, convex optimization solvers are available and a solution (when there is one) can be found rapidly, letting the possibility of using this technique online.

2.3.1 Difference between MPC and Path Planning

Model predictive control is a feedback technique used to control systems based on numerical optimization algorithms. Therefore, those controllers are meant to be used

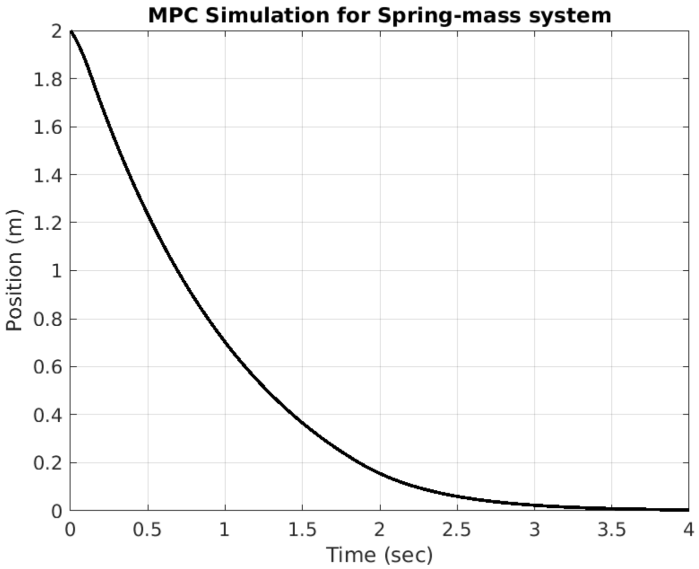


Figure 15: Closed-Loop Position Versus Time

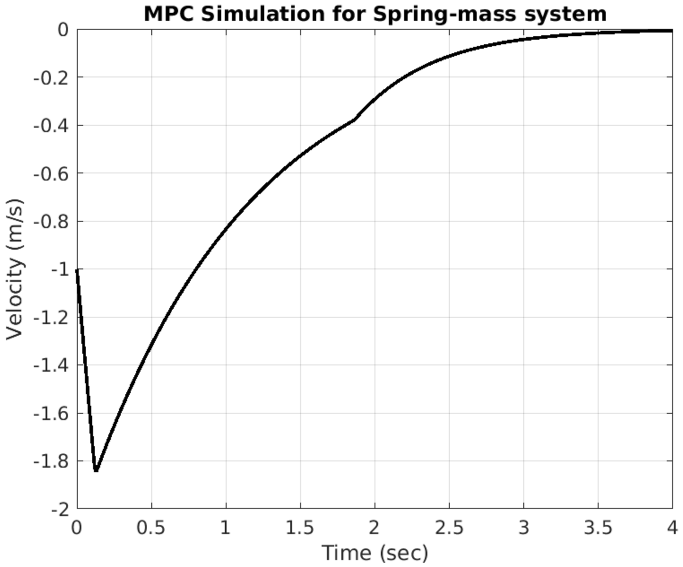


Figure 16: Closed-Loop Velocity Versus Time

online, the same way we used typical linear controllers.

Similar to this technique, one can use convex optimization technique to generate upfront nominal trajectories. Following this, with the help of online lower level controllers, we make the system follow this trajectory. This last technique being called “path-planning”. Although, those two techniques might look very similar, one should

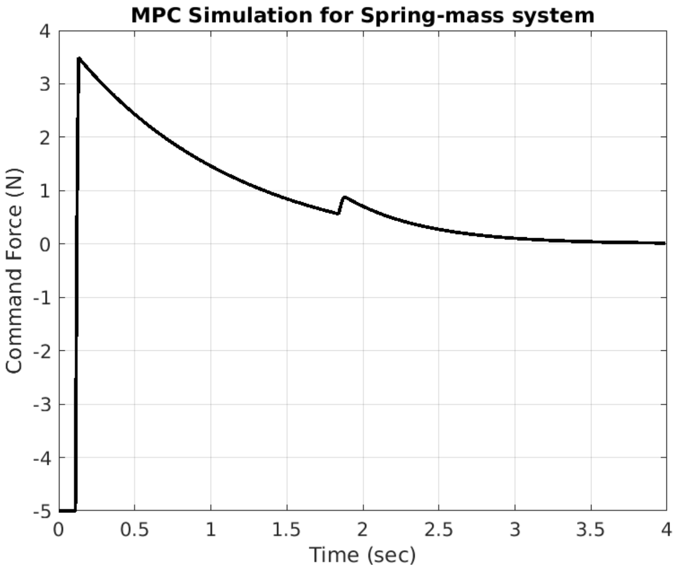


Figure 17: Input Force Versus Time

be very careful and understand that they imply different guarantees.

When implementing a path-planning scheme, the stability of the closed-loop system lies in the stability of the lower-level controllers and the correctness of the nominal trajectory generated beforehand. For receding horizon controllers, system stability is not that obvious. Using optimization techniques online, the closed-loop trajectory is not necessary close to the future potential trajectories computed at each iterations. Because of this, we need more powerful mathematical tools in order to prove stability. As it was explained in [26], Lyapunov functions could be used to prove stability of MPC controllers. In that paper, the authors show how end-point penalty can be used as Lyapunov functions and therefore prove closed-loop stability.

Chapter III

THE ELLIPSOID METHOD AND ITS SEMANTICS

In this chapter, we recall the last advances and steps in optimization theory and show its consequences in science and engineering. After this, we give details about the algorithm of interest in this thesis.

3.1 A Brief Recent History of Optimization Theory

Optimization has been a on going research area of mathematics for centuries. Many famous mathematicians were interested in it including Lagrange, Euler, Newton and many others. More recently, a revolutionary discovery was made in 1947 by George Dantzig with the elaboration of the Simplex Method solving Linear Programs. This method was a huge step forward in the area of linear optimization and represented at the time, an extremely efficient method although having poor theoretical complexity (exponential).

In 1970 the mathematicians Shor, Judin, and Nemirovski published a highly important result, the discovery of an algorithm, the Ellipsoid Method, solving in polynomial time any convex programs. Couple of years latter, in 1979 soviet mathematician Leonid Khachiyan applied this last method to the more specific setting of Linear Programs, showing for the first time the polynomial solvability of Linear Programs. At this point, although polynomial time algorithm for LP was found using the Ellipsoid Method, the Simplex Method was still widely used because of better running time. Indeed, theoretically very slow, the Simplex Method remains in practice extremely fast.

Following this, in 1984, Indian mathematician Narendra Karmarkar published the first interior-point method solving linear programs in polynomial time. This drastically increased the interest for interior-point methods to solve linear optimization problems, which was until there only applied for non-linear optimization. This discovery was quite a revolution, unifying linear and nonlinear optimization techniques. Up until 1984, linear programs and non-linear programs where seen as two different problems that should be treated with different types of methods. With Karmarkar's first interior-point method solving LP in linear time, the community realized that linear and non-linear optimization could be treated the same way and shared interesting properties. This last algorithm being also efficient on both complexity and practice point of view.

Following the increasing interest for interior-point methods, mathematician Nemirowski, in the 90s came up with a marvelous result, the extension of interior-point methods to solve semi-definite programs (SDP). Indeed, Arkadi Nemirovski showed and explained how interior-point methods could be expanded to solve semi-definite optimization problems. This result is quite remarkable, allowing the community to expand the known results of Linear Programs to semi-definite optimization problems.

3.2 The Ellipsoid Method

As it was recalled in the introduction on this chapter, the Ellipsoid Method was first published by the mathematicians Shor, Judin, and Nemirovski in 1970. This algorithm was said to solve convex problems in polynomial time. One of the most famous consequence of this discovery is the elaboration of the Khachiyan method, which is named after its author Leonid Khachiyan and consist of an application of this method to linear programs.

Despite its relative efficiency with respect to interior point methods, the Ellipsoid Method benefits from concrete proof elements and could be considered a viable option for critical embedded systems where safety is more important than performance. For the Ellipsoid Method, several algorithms that are mathematically equivalent exist. All algorithms implement the same successive ellipsoids and present a mathematically equivalent update. Because there are several ways of encoding and defining an Ellipsoid, several algorithms exist. The point here being the fact that those algorithms are not numerically equivalent and we need to focus on the most accurate one. An Ellipsoid can be defined by a positive-definite matrix $P > 0$. Equivalently, because for all positive-definite matrix $P > 0$, there exists a square-root matrix B such that $B^2 = P$, one can choose to implement the same algorithm but propagating the successive square root matrices B. The algorithm presented in [11] show a implementation of the Ellipsoid Method propagating the positive-definite matrices P and the one developed in [36] (The first one published) show an implementation propagating the matrices B. The same way it is for the Kalman Filter, we experienced that the algorithm implementing the square-root matrices is more numerically accurate. For that reason, we chose to focus ourselves on this implementation. This chapter presents a way to annotate a C code implementation of the Ellipsoid Method to ensure that the code implements the method and therefore shares its properties (convergence, soundness). Before recalling the main steps of the algorithm, the needed elements will be presented.

Ellipsoids in $\mathbb{R}^n$. An ellipsoid can be characterized as an affine transformation of an Euclidean Ball. Before defining an Ellipsoid set, we first recall the definition for an Euclidean ball.

Definition 2 (Euclidean balls) *Let $n \in \mathbb{N}$ we denote B_n the unit Euclidean ball in*

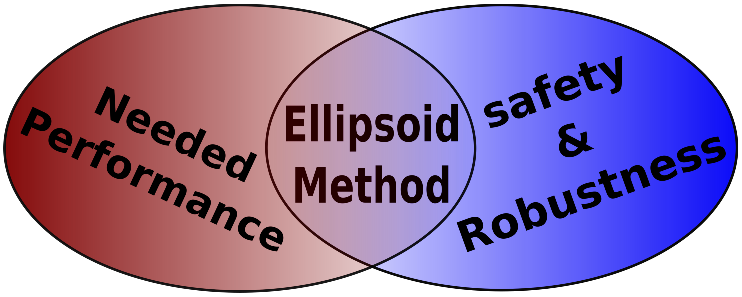


Figure 18: Ellispoid Method Trade off

$\mathbb{R}^n$. *For $n \in \mathbb{R}^n$, we define the Euclidean ball by:*

$$B_n = \{z \in \mathbb{R}^n \ : \ \|z\| \leq 1\}$$

Also, $\mathrm{Vol}(B_n)$ *denotes its volume. Also, we define $B_r(x)$ as the ball of radius r centered on x* $\Big(\ i.e \ \{z \in \mathbb{R}^n : \|z - x\| \leq r\} \ \Big)$.

Definition 3 (Ellipsoid Sets) *Let $c \in \mathbb{R}^n$ and $B \in \mathbb{R}^{n \times n}$ a non-singular matrix ($det(B) \neq 0$). The Ellipsoid* $\mathrm{Ell}(B, c)$ *is the set :*

$$\mathrm{Ell}(B, c) = \{Bu + c : u^T u \leq 1\} \tag{10}$$

Also, because the convergence of the method is a consequence of a volume decreasing property, we define below the volume of an Ellipsoid.

Definition 4 (Volume of Ellipsoids) *Let* $\mathrm{Ell}(B, c)$ *be an ellipsoid set in $\mathbb{R}^n$. We denote by $Vol(\mathrm{Ell}(B, c))$ its volume defined as :*

$$\mathrm{Vol}(\mathrm{Ell}(B, c)) = |det(B)| \cdot \mathrm{Vol}(V_n) \tag{11}$$

Algorithm. Let us now recall the main steps of the algorithm detailed in [7, 36, 11]. In the following, we denote $E_k = \mathrm{Ell}(B_k, c_k)$, the ellipsoid computed by the algorithm at the $k - th$ iteration.

Ellipsoid cut. We start the algorithm with an ellipsoid containing the feasible set X, and therefore the optimal point x^*. We iterate by transforming the current ellipsoid E_k into a smaller volume ellipsoid E_{k+1} that also contains x^*. Given an ellipsoid E_k of center c_k, we find a hyperplane containing c_k that cuts E_k in half, such that one half is known not to contain x^*. Finding such a hyperplane is called the *oracle separation* step, *cf.* [36]. In our SOCP setting, this cutting hyperplane is obtained by taking the gradient of either a violated constraint or the cost function. Then, we define the ellipsoid E_{k+1} by the minimal volume ellipsoid containing the half ellipsoid $\hat{E}_k$ that is known to contain x^*. The Figures 19 and 20 illustrate such ellipsoids cuts.

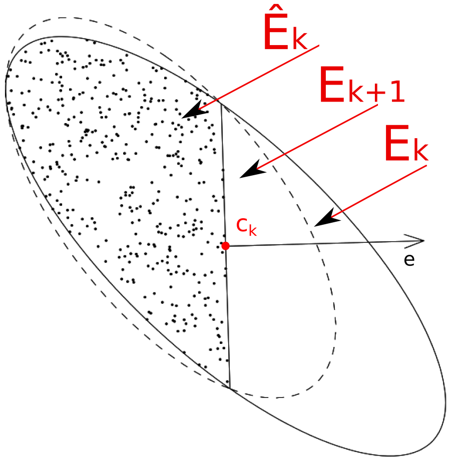


Figure 19: Ellipsoid Cut

Ellipsoid transformation. From the oracle separation step, a separating hyperplane, e, that cuts E_k in half with the guarantee that x^* is localized in $\hat{E}_k$ has been computed. The following step is the *Ellipsoid transformation*. Using this hyperplane e, one can update the ellipsoid E_k to its next iterate E_{k+1} according to equations (12) and (13). In addition to that, we know an upper bound, γ, of the ratio of Vol(E_{k+1}) to Vol(E_k) (see Property 1).

$$c_{k+1} = c_k - \frac{1}{(n+1)} \cdot B_k p \quad , \tag{12}$$

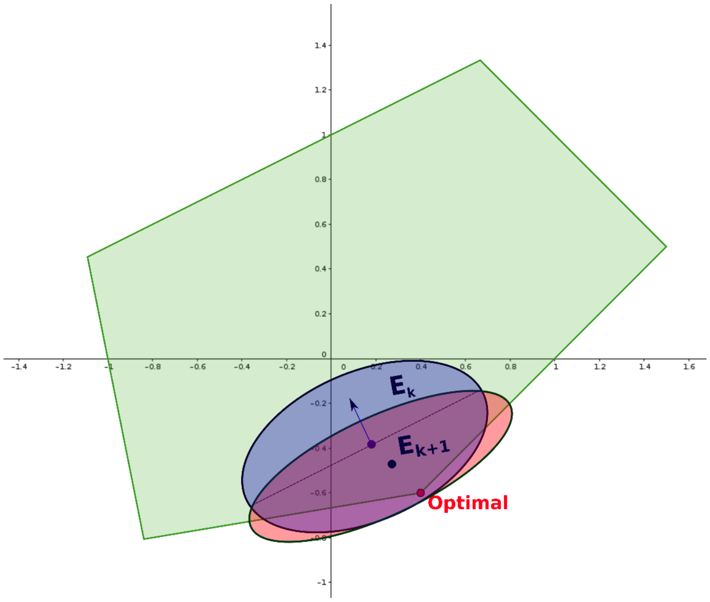


Figure 20: Ellipsoid Cut In an LP Settings

$$B_{k+1} = \frac{n}{\sqrt{n^2-1}} B_k + \left(\frac{n}{n+1} - \frac{n}{\sqrt{n^2-1}} \right) (B_k p) p^T \tag{13}$$

with:

$$p = \frac{B_k^T e}{\sqrt{e^T B_k B_k^T e}}. \tag{14}$$

Termination. The search points are the successive centers of the ellipsoids. Throughout the execution of the algorithm, we keep track of the best point so far, $\hat{x}$. A point x is better than a point y if it is feasible and have a smaller cost. When the program reaches the number of iterations needed, the best point so far, $\hat{x}$, which is known to be feasible and ϵ-optimal, is returned by the algorithm. We state a volume related property, at the origin of the algorithm convergence, then state the main theorem of the method.

Property 1 *[Reduction ratio.] Let $k \geq 0$, by construction:*

$$\text{Vol}(E_{k+1}) \leq \exp\left(\frac{-1}{2 \cdot (n+1)} \right) \cdot \text{Vol}(E_k) \tag{15}$$

Proof *We give a proof to show that the successive ellipsoids computed by the method are actually decreasing by a ratio γ. First, let us put the update formula 13 into the form:*

$$B_{k+1} = \alpha B_k + \beta (B_k p) p^T$$

with:

$$\alpha = \frac{n}{\sqrt{n^2 - 1}} \quad \textit{and} \quad \beta = \frac{n}{n+1} - \frac{n}{\sqrt{n^2-1}}.$$

Let us now take the determinant of both sides.

$$\begin{aligned}
\det(B_{k+1}) &= \det\Big(\alpha B_k + \beta (B_k p) p^T\Big) \\
&= \det\Big(B_k \cdot \big(\alpha I_n + \beta p p^T\big)\Big) \\
&= \det\big(B_k\big) \det\Big(\alpha I_n + \beta p p^T\Big) \\
&= \det\big(B_k\big) \alpha^n \det\Big(I_n + \frac{\beta}{\alpha} p p^T\Big)
\end{aligned}$$

Using Sylvester's determinant identity:

$$\det(I_n + AB) = \det(I_m + BA) \qquad \forall A \in \mathbb{R}^{n \times m}, B \in \mathbb{R}^{m \times n}$$

the determinant on the right side of the equality can be express as:

$$\det(B_{k+1}) = \alpha^n \det\big(B_k\big) \cdot \Big(1 + \frac{\beta}{\alpha} \|p\|\Big)$$

But, From equation 14, we can see that $\|p\| = 1$. Therefore,

$$\frac{\det(B_{k+1})}{\det(B_k)} = \alpha^n \cdot \Big(1 + \frac{\beta}{\alpha}\Big) \leq \exp\left(\frac{-1}{2(n+1)}\right)$$

□

Hypotheses. In order to characterize the number of steps required for the algorithm to return an ϵ-optimal solution, three scalars and a point $x_c \in \mathbb{R}^n$ are needed:

- a radius R such that:

$$X \subset B_R(x_c) \tag{16}$$

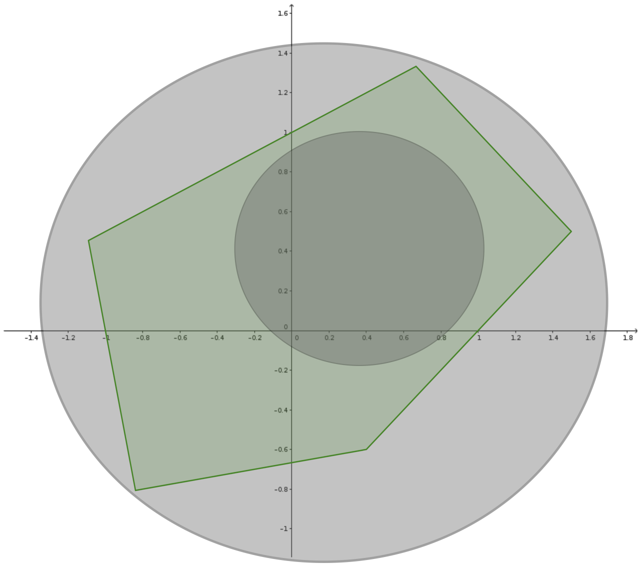


Figure 21: Included and Including Balls

- a scalar r such that:

$$B_r(x_c) \subset X \tag{17}$$

- and another scalar V such that:

$$\max_{x \in X} \; f_o - \min_{x \in X} \; f_o \leq V. \tag{18}$$

Those hypotheses are illustrated in Figure 21. This example corresponds to a Linear Program setting where the feasible set is a bounded and not flat polyhedral set. The different balls are shown in shades of gray. More generally, the existence of those assumptions implies that the feasible set needs to be both bounded and not flat. Unfortunately, when implementing MPC controllers, equality constraints are present, implying that the feasible set is flat on some dimensions. Therefore, we performed an initial equality constraint elimination (see Chapter 5.1.1).

The main result can be stated as:

Theorem 4 *Let us assume that X is bounded, not empty and such that R, r and V are known. Then, for all $\epsilon \in \mathbb{R}_+^*$, the algorithm, using N iterations, will return $\hat{x}$, satisfying:*

$$f_o(\hat{x}) \leq f_o(x^*) + \epsilon \textit{ and } \hat{x} \in X \ (\epsilon\textit{-solution})$$

Furthermore, if we write n the dimension of the optimization problem, the number of steps, N, is polynomial in n and is of the form:

$$N = 2n \cdot (n+1) \log\left(\frac{RV}{r\epsilon}\right).$$

This result, when applied to LP, is historically at the origin of the proof of the polynomial solvability of linear programs. Its proof can be found at [30, 36]. In order to produce formally verifiable code, we want to generate annotated code with Hoare triples, including function contracts expressing pre and post conditions. To do so, we present the development of ACSL theories related to optimization problems.

3.3 Building ACSL Theory Related to the Ellipsoid Method

3.3.1 Linear Algebra Axiomatization

In this section we give details about the ACSL theories we had to build in order to prove mathematical properties at code level. Indeed, the software analyzer takes as an input the annotated C code augmented with ACSL theories that define new abstract types, functions but also axioms, lemmas and theorems. The lemmas and theorems need to be proven but the axioms are always assumed to be true. For the SMT solver, properties at code level are usually harder to prove than lemmas within ACSL theories. Thus, our approach here was to develop the needed ACSL theories enough to be able to express and prove the main results used by the algorithm within the

ACSL

```
/*@ axiomatic LinAlg {
  type vector;
  type matrix;
  logic vector vec_of_16_scalar(double * x)  reads x[0..15];
  logic vector vec_of_36_scalar(double * x)  reads x[0..35];
  ...
  logic vector vector_add(vector A, vector B);
  axiom vector_add_length:
    \forall vector x, y;
      vector_length(x) == vector_length(y) ==>
      vector_length(vector_add(x,y)) == vector_length(x);
  axiom vector_add_select:
    \forall vector x, y, integer i;
      vector_length(x) == vector_length(y) ==>
      0 <= i < vector_length(x) ==>
      vector_select(vector_add(x,y),i)==vector_select(x,i)+vector_select(y,i);
  ...
}
*/
```

Figure 22: ACSL Linear Algebra Theory

ACSL theories. That way, the Hoare triples at code level will only be an instantiation of those lemmas and be relatively simple to prove for the SMT solvers.

Linear Algebra Based ACSL Theory. In this ACSL theory, we defined new abstract types for vectors and matrices. We also defined functions that allow us to create a vector and a matrix from a C code pointer. Additionally, all the very well known operations have also been axiomatized (give a mathematical description to it) such as matrix multiplication, matrix addition, vector-scalar multiplication, scalar product, norm.

This ACSL theory is automatically generated during the autocoding process of the project, thus, all the sizes of the vectors and matrices are known. Within this theory, we only defined functions that will create, from a C code pointer, objects of appropriate sizes (as illustrated in figure 22). ACSL code is printed in green and its keywords in red. The C code keywords are printed in blue and the actual C code is printed in black. Figure 22 presents the definition of two abstract types matrix and vector, the ACSL constructors for those types and the axiomatization of vectors addition. Figure 22 represents an extract from the autocoded ACSL linear algebra theory (which can be found in appendix A).

Ellipsoid Method Based ACSL Theory. In addition to defining new types for optimization problem, we also axiomatize the calculation of the vector constraint, feasibility, epsilon optimality, etc, ... Also, Ellipsoids and related properties are axiomatized, as presented in Figure 23. Within this theory, all the axioms and lemmas required for the main proof will be autocoded and proved. Before stating the main result, some preliminary results needs to be obtained. For instance, volume related properties are defined and proved (Figure 24). The proof of the main lemma for the method is stated in Figure 25. The first assumption listed on this lemma is the fact that the reals r, ϵ, V are strictly positive. It also assumes that the variable V satisfies the property (18). The lines 13 and 14 express the fact that we are assuming the ellipsoid $\text{Ell}(P, x)$ to be a minimum localizer (meaning that any point outside of this ellipsoid cannot be better than x_best, in term of cost or feasibility. Lines 15 to 17 specify the assumption for the scalar r to fulfill, corresponding to equation (17). Finally, at line 18, we express the fact the the ellipsoid $\text{Ell}(P, x)$ has a volume less than $(\epsilon * r/V)^n$. The conclusion of the lemma being the fact that x_{best} represents an ϵ-solution to the optimization problem. In Figure 24, we show a axiomatization of a set. We wrote two axioms. One specifying that every set have a positive volume, and another one specifying that if a set B contains all the elements of a set A then the volume of B is at least equal to the volume of A. Then, we state a lemma expressing

ACSL

```
#include "axiom_linalg.h"
/*@ axiomatic Ellipsoid {
  ...
  type ellipsoid;
  logic ellipsoid Ell(matrix P, vector x);
  logic boolean inEllipsoid(ellipsoid E, vector z);
  ...
}
*/
```

Figure 23: Ellipsoid Type Definition

ACSL

```
#include "axiom_linalg.h"
/*@ axiomatic Optim {
  ...
  type myset;
  logic boolean in(myset A, vector x);
  logic real volume(myset A);
  axiom PositiveVolume:
    \forall myset A; volume(A) >= 0;
  axiom greatherVolume:
    \forall myset A, B;
      (\forall vector x; in(A,x) ==> in(B,x)) ==>volume(B) >= volume(A);
  lemma lemmaExitsElement:
    \forall myset A, B;
      (volume(A) < volume(B)) ==> \exists vector x; in(B,x) && !in(A,x);
}
*/
```

Figure 24: Existence vector ACSL Lemma

that for two sets A and B, if the volume of A is strictly less than the volume of B then there exists a element x that belong to the set B but does not belong to the set A.

Both lemmas have been successfully proved using the software analyzer Frama-C and the SMT solver Alt-Ergo.

ACSL Lemma

```
/*@
lemma epsilon_solution_lemma_BIS:
  \forall optim OPT, real r,V,epsilon, matrix P, vector x, x_best;
    (0 < epsilon/V < 1) ==>
    0 < r ==>
    0 < V ==>
    0 < epsilon ==>
    size_n(OPT) > 0 ==>
    ( \forall vector x1, x2;
      isFeasible(OPT, x1) ==>
      isFeasible(OPT, x2) ==>
      cost(OPT,x1) - cost(OPT,x2) <= V  ) ==>
    ( \forall vector z;
       !inEllipsoid(Ell(P,x), z) ==> isBetter(OPT, z, x_best) ) ==>
    ( \exists vector x;
       include(tomyset(Ell(mat_mult_scalar(ident(size_n(OPT)),r), x)) ,
           feasible_set(OPT)) ) ==>
    volume(tomyset(Ell(P,x))) < pow(epsilon/V*r, size_n(OPT)) ==>
    isEpsilonSolution(OPT, x_best, epsilon);  */
```

Figure 25: Ellipsoid Method ACSL Lemma

3.3.2 Optimization Theory Axiomatization

To axiomatize an optimization problem, we intend to see it, independently of the method used to solve it, as a pure mathematical object. Our goal is to axiomatize it with enough properties, allowing us to state all the needed optimization-level properties at code level for the proof. Let us consider the second-order cone program, described in Eq. 6.

Encoding an SOCP. In order to fully describe an SOCP, we use the variables:

$$f \in \mathbb{R}^n, \quad A = \begin{bmatrix} A_1 \\ \vdots \\ A_m \end{bmatrix}, \quad b = \begin{bmatrix} b_1 \\ \vdots \\ b_m \end{bmatrix}, \quad C = \begin{bmatrix} c_1^T \\ \vdots \\ c_m^T \end{bmatrix}, \quad d = \begin{bmatrix} d_1 \\ \vdots \\ d_m \end{bmatrix}$$

And also the vector $m = \begin{bmatrix} n_1 & \dots & n_m \end{bmatrix}$ collecting the sizes of the vectors $A_i \cdot x + b_i$. Furthermore, if $na = \sum_{i=1}^{m} n_i = 0$, then, the SOCP we are considering is actually an LP. Using ACSL, we define a new type and a high level function, providing the possibility to create objects of the type "*optim*" (Figure 26).

When applying a method to solve an optimization problem, many concepts are important. The work here is to highlight those concepts and write a library translating those concepts into a formal specification language. That way, formally verifiable code could be produced, independently of the method and implementation used. The concepts of feasibility and optimality are being axiomatized. For this, given a second-order cone program, we gave an axiomatic definition for the vector constraint, the gradient of a constraint, the cost, optimal point (making the assumption that it exists and is unique), etc. For instance, Figure 27 illustrates the axiomatization of a constraint calculation and the feasibility predicate definition. When instantiating a object of type vector or ma-

ACSL

```
/*@
axiomatic OptimSOCP {
type optim;
logic optim socp_of_size_2_6_0(
   matrix A,vector b,matrix C,
   vector d, vector f, int* m)
   reads m[0..5];
logic real constraint(optim OPT,
   vector x,integer i);
logic vector constraints(optim OPT,
   vector x);
*/
```

Figure 26: ACSL Optim Type Definition

ACSL

```
/*@
  axiom constraint_linear_axiom:
    \forall optim OPT, vector x, integer i;
      getm(OPT)[i] == 0 ==>
        constraint(OPT, x, i) ==
          -scalarProduct(getci(OPT,i),x,size_n(OPT))-getdi(OPT,i);
  axiom constraint_socp_axiom:
    \forall optim OPT, vector x, integer i;
      getm(OPT)[i] != 0 ==>
        constraint(OPT, x, i) ==
          twoNorm(vector_affine(getAi(OPT,i),x,getbi(OPT,i))) -
          scalarProduct(getci(OPT,i),x,size_n(OPT))-getdi(OPT,i);
  ...
  predicate
    isFeasible(optim OPT,vector x) = isNegative(constraints(OPT,x));
*/
```

Figure 27: ACSL Feasible Predicate Definition

trix, the size of the considered object needs to be know since it is hard-coded in the ACSL axiomatization. This does not represent an issue at this time since we already know all the sizes of the variable used (from the autocoder, see Section 5). Also, working with predefined and hard-coded size objects will help the analyzers proving the goals. The work presented here is generic and the code can be generated for any size of matrices and vectors.

3.4 Annotating a C code implementation of the Ellipsoid Method

We now give details about how we annotated the C code and the type of Hoare triples present in the code. For this, we adopted a specific technique. Every C code function will be implemented in a separated file. That way for every function, a corresponding C code body (.c) file and header file (.h) will be automatically generated. The body file contains the implementation of the function along with annotations and loop invariants. The header file contains the declaration of the function with its ACSL contract.

The first kind of Hoare triples and function contract we added to the code was to check the basic mathematical operations. For instance, Figures 29 and 28 present the

C code body and header files of the function computing the two norm of a vector of size two. Thanks to this contract, we can prove that the value returned by the function is indeed the two norm of the vector of size two associated with the input pointer. Furthermore, we proved that the result is always positive or null, and that, assuming the corresponding vector is not equal to zero, the output is necessary strictly greater than zero. This last property being interesting when proving there are no division by zero (normalizing vectors). Then, once all the functions implementing elementary mathematical operations have been annotated and proven, we annotate the higher-level C functions such as constraint calculations, gradient calculations, matrix and vector update, ... The Figures 30 and 31 show the annotated C function and contract for the function "getp" that computes the vector p as described in equation (14), needed to perform the ellipsoid update.

C Code + ACSL

```
#ifndef getNorm_2_lib
#define getNorm_2_lib
#include "axiom_linalg.h"
#include "my_sqrt.h"
#include "scalarProduct_2.h"
/*@
  @ requires \valid(Ain+(0..1));
  @ ensures \result == twoNorm(vec_of_2_scalar(Ain));
  @ ensures \result >= 0;
  @ assigns \nothing;
  @ behavior Ain_non_null:
    @ assumes nonnull(vec_of_2_scalar(Ain));
    @ ensures \result > 0;
  @ behavior Ain_null:
    @ assumes !nonnull(vec_of_2_scalar(Ain));
    @ ensures \result == 0;
  @ complete behaviors Ain_non_null, Ain_null;
  @ disjoint behaviors Ain_non_null, Ain_null;
*/
double getNorm_2(double *Ain);
#endif
```

Figure 28: getNorm_2 Header C Code File

The function contract shown on figure 31, extracts the fact that no variable corresponding to the optimization problem (A,b,C,d,f and m) are getting assigned and that their values after the execution of the function are the same as before. This contract specifies also the same property for the variables $grad$, P_minus and x_minus.

C Code + ACSL

```c
#include "getNorm_2.h"
double getNorm_2(double *Ain) {
  double sum;
  sum = scalarProduct_2(Ain, Ain);
  return my_sqrt(sum);
}
```

Figure 29: getNorm_2 Body C Code File

C Code + ACSL

```c
#include "getp.h"
void getp() {
  double norm;
  double norm_inv;
  getTranspose();
  /*@ assert mat_of_2x2_scalar(&temp_matrix[0]) ==
      transpose( mat_of_2x2_scalar(&P_minus[0]) );
  */
  changeAxis();
  /*@ assert vec_of_2_scalar(&temp2[0]) ==
    mat_mult_vector( mat_of_2x2_scalar(&temp_matrix[0]),
             vec_of_2_scalar(&grad[0])) ;
  */
  /*@ assert vec_of_2_scalar(&temp2[0]) ==
        mat_mult_vector(transpose( mat_of_2x2_scalar(&P_minus[0]) ),
                vec_of_2_scalar(&grad[0])) ;
  */
  norm = getNorm_2(temp2);
  /*@ assert 1/norm ==
      1/twoNorm( mat_mult_vector(transpose( mat_of_2x2_scalar(&P_minus[0]) ),
                         vec_of_2_scalar(&grad[0]))  )  ;
  */
  /*@ assert vec_of_2_scalar(&temp2[0]) ==
          mat_mult_vector(transpose( mat_of_2x2_scalar(&P_minus[0]) ),
                  vec_of_2_scalar(&grad[0])) ;
  */
  norm_inv = 1.0 / (norm);
  scaleAxis(norm_inv);
  /*@ assert 1/norm ==
    1/twoNorm( mat_mult_vector( transpose( mat_of_2x2_scalar(&P_minus[0]) ),
                  vec_of_2_scalar(&grad[0]))  ) ;
  */
  /*@ assert vec_of_2_scalar(&temp2[0]) ==
      mat_mult_vector(transpose( mat_of_2x2_scalar(&P_minus[0]) ),
                     vec_of_2_scalar(&grad[0])) ;
  */
  /*@ assert vec_of_2_scalar(&p[0]) ==
      vec_mult_scalar(vec_of_2_scalar(&temp2[0]), 1/norm);
  */
}
```

Figure 30: getp.c Body C Code File

The requires clause at line 17, expresses the fact that we are assuming the matrix of size $(2, 2)$ defined by the pointer P_minus to be invertible. Thanks to the assigns clause, we point out that this function only affects the variables p, $temp2$ and $temp\_matrix$. Finally, this triple lists three post execution properties that are supposed to be true. One expresses that the two norm of the vector of size two defined by

C Code + ACSL

```c
#ifndef getp_lib
#define getp_lib

#include "axiom_def_lin_alg.h"
#include "socp/sizes.h"
#include "getNorm_2.h"
#include "changeAxis.h"
#include "scaleAxis.h"
#include "getTranspose.h"

extern double grad[N];
extern double P_minus[N*N];
extern double p[N];
extern double temp2[N];

/*@
  @ requires invertible(mat_of_2x2_scalar(&P_minus[0])) == 1;
  @ ensures A_unchanged: mat_of_0x2_scalar((double *) A) ==
            mat_of_0x2_scalar((double *) A);
  @ ensures b_unchanged: vec_of_0_scalar((double *) b) ==
              vec_of_0_scalar((double *) b);
  @ ensures C_unchanged: mat_of_6x2_scalar{Here}((double *) C) ==
            mat_of_6x2_scalar{Pre}((double *) C);
  @ ensures d_unchanged: vec_of_6_scalar{Here}((double *) d) ==
            vec_of_6_scalar{Pre}((double *) d);
  @ ensures f_unchanged: vec_of_2_scalar{Here}((double *) f) ==
            vec_of_2_scalar{Pre}((double *) f);
  @ ensures m_unchanged: \forall integer l; 0 <= l < 6 ==>
              \at(m[l], Here) ==  \at(m[l], Pre);
  @ ensures grad_unchanged: vec_of_2_scalar{Here}((double *) grad) ==
            vec_of_2_scalar{Old}((double *) grad);
  @ ensures P_minus_unchanged: mat_of_2x2_scalar{Here}((double *) P_minus) ==
            mat_of_2x2_scalar{Old}((double *) P_minus);
  @ ensures x_minus_unchanged: vec_of_2_scalar{Here}((double *) x_minus) ==
            vec_of_2_scalar{Old}((double *) x_minus);
  @ ensures vec_of_2_scalar(&p[0]) ==
          vec_mult_scalar(mat_mult_vector(transpose( mat_of_2x2_scalar(&P_minus[0]) ),
                                vec_of_2_scalar(&grad[0])),
                  1/twoNorm(mat_mult_vector(transpose(mat_of_2x2_scalar(&P_minus[0])),
                                  vec_of_2_scalar(&grad[0]))));
  @ ensures twoNorm(vec_of_2_scalar(&p[0])) == 1;
  @ ensures invertible(mat_of_2x2_scalar(&P_minus[0])) == 1;
  @ assigns p[0..1], temp2[0..1], temp_matrix[0..3];
*/
void getp();
#endif
```

Figure 31: getp.h Header File

the pointer p is equal to one. Another specifies that the matrix of size $(2, 2)$ defined by the pointer *P_minus* is invertible. The last expresses that the vector of size two defined by the pointer p is equal to the normalized multiplication of the transpose of the matrix of size $(2, 2)$ defined by the pointer *P_minus* times the vector of size two defined by the pointer *grad*. This latter property expresses equation (14). In Figure 30, we show the implementation of the corresponding C code function and all the annotations required for the proof. For this, we decomposed the function into elementary mathematical operations performed via function called. That way, the code is traceable and easier to prove for the SMT solvers. Both function contracts have been successfully proved. One can wonder why we have the specification for the matrix P_minus to be invertible after the execution of the function. Indeed, we assumed this same property to be true beforehand and specify that the variable it applies for would not change. These two properties making it trivial. In order to accelerate the verification process, SMT solvers need to be guided. Therefore, every needed property that is required somewhere else in the project should be listed, even if, logically speaking, it does not give any additional information. Thus, in order to guide the SMT solvers and to accelerate the verification, we added the ensures property at line 42.

Chapter IV

FLOATING POINTS ANALYSIS

So far we presented how one can formalize existing results about the Ellipsoid Method and prove them at code level. We show now how the original algorithm can be modified in order to manage numerical errors. In this chapter we present a way of analyzing and controlling the numerical errors of a modified version of the Ellipsoid Algorithm. Numerical errors of embedded systems can lead to tremendous catastrophes, therefore analyzing the numerical property of an algorithm that is meant to be embedded on a safety-critical CPS is an inevitable task. After recalling some past system failures we present in this chapter the work that has been done towards those issues.

4.1 Past System Failures and Motivation

4.1.1 US Patriot Missile

During the Gulf war in Saudi Arabia, 1991, a US patriot missile failed to intercept an incoming Iraqi Scud missile. It ended up killing 28 US soldiers and injuring around 100 other people.

It turned out the issue was coming from internal variable used by the system to compute the current time in *sec*. The time from the internal clock of the system was stored in a variable t_{clock}, encoded on a 24 bit fixed-point number and represented the time since system boot in decasecond. When current time in *sec* was needed, the program was performing equation (19), multiply as expected the internal variable by 0.1.

$$t\ (sec) = t_{clock} \times \mathrm{fl}(0.1) \tag{19}$$

Unfortunately, the constant 0.1 cannot be represented using 24 bit fixed-point numbers (it is also not the case for floating-point numbers). Therefore, this constant was chopped and induced an error on the time computation that was not expected by the engineers. The small rounding error, when multiplied by the large number giving the time in tenths of a second, led to a significant error. After 100 hours the resulting time error was approximately of 0.34 *sec* (see equation (19)). Equivalently, because a Scud travels at about $3,750$ *mph*, it corresponds to an error of more than $1,600$ ft.

Figure 32: US Patriot Missile

4.1.2 Ariane 5 Rocket

This error is known as one of the most expensive floating-point error. It caused a damage worth half a billion dollars. The horizontal velocity of the rocket encoded on a 64 bit floating-point number was converted to a 16 bit signed integer. The problem here being the fact that this number was larger than the largest integer representable on a 16 bit signed integer. The conversion failed and the software ended up triggering a SAFETY mode, switching to a backup computer. Unfortunately, the same error happened and it was misinterpreted as a scenario where aggressive control input from the motor was needed. Only 40 seconds after ignition, at an altitude of about 12000 ft, the rocket went out of its nominal trajectory, broke up and exploded.

Figure 33: European Ariane 5 Rocket

4.1.3 Motivations

As seen from the past sections, it is undeniable that floating-points errors could lead to system failure. We now argue that, in addition to that, the nature of the algorithm we are developing are particularly sensitive to numerical errors and thus analyzing its numerical properties is even more important. A large majority of optimization algorithms are iterative algorithms. Meaning that within its structure, lies a main loop in which the $n-th$ approximation is derived from the previous ones. Each iterate have given properties, which are necessary for the algorithm convergence. If we consider the operations performed by the computer to have errors, those errors could potentially be added-up at each iteration and could lead to a software error. Therefore, studying numerical stability is indeed a need and would give guarantees that are highly appreciated.

4.2 Controlling the Condition Number

In this section we explain how the condition number of the successive Ellipsoids of the algorithm can be bounded through the execution of the code and why it is important. We recall below the definition of the condition number of a non-singular matrix.

Definition 5 (Condition Number of a Matrix) *Let* $A \in \mathbb{R}^{n\times n}$ *a non-singular* *matrix. We define the condition number of the matrix A the scalar $k(A)$ such that:*

$$k(A) = \|A\| \cdot \left\|A^{-1}\right\|$$

By extension, we talk about the condition number of an Ellipsoid Ell(B, c) by taking the condition number of the matrix B, $k(B)$.

Bounding the condition number of the matrix B is fundamental and represents the main argument of the algorithm numerical stability. Unfortunately, for the original algorithm, no reasonable bound on $k(B)$ can be found. Therefore, we slightly modified the ellipsoid algorithm to make it able to correct the current ellipsoid E_i in the case where its condition number had become too high (ellipsoid too flat). That way we can control the condition number of B. Additionally, we made sure that this correcting step, when it occurs, does not break the convergence of the algorithm and its semantics described in Section 3.2. In this section, we give details about the modification performed on the original algorithm.

4.2.1 Bounding the Singular Values

When updating the matrix B_i by the usual formulas of the Ellipsoid Algorithm, B_i evolves according to

$$B_{i+1} = B_i \cdot D_i, \tag{20}$$

where $n-1$ singular values of D_i are $n/\sqrt{n^2-1}$, and one singular value is $n/(n+1)$. It follows that at a single step the largest and the smallest singular values of B_i can change by a factor from $[1/2, 2]$.

Proof *From the update equation* (20)*, we have the property:*

$$\sigma_{max}(B_{i+1}) = \sigma_{max}(B_i \cdot D_i) = \|B_i \cdot D_i\|_2 \leq \|B_i\|_2 \cdot \|D_i\|_2$$

because the two norm is a consistent norm and

$$\|A\|_2 = \sigma_{max}(A) \quad \forall A \in \mathbb{R}^{n \times n}$$

Therefore,

$$\sigma_{max}(B_{i+1}) \leq \sigma_{max}(B_i) \cdot \sigma_{max}(D_i) = \sigma_{max}(B_i) \cdot \frac{n}{\sqrt{n^2-1}}.$$

Which implies that:

$$\sigma_{max}(B_{i+1}) \leq 2 \cdot \sigma_{max}(B_i) \quad \forall n \in \mathbb{N} \ , \ n \geq 2$$

For σ_{min}, let us take first the inverse of the update equation (20) *(we know that all the matrices are indeed invertible). We have: $B_{i+1}^{-1} = D_i^{-1} \cdot B_i^{-1}$. Thus,*

$$\sigma_{max}(B_{i+1}^{-1}) = \sigma_{max}(D_i^{-1} \cdot B_i^{-1}) \leq \sigma_{max}(D_i^{-1}) \cdot \sigma_{max}(B_i^{-1})$$

Using the fact that for an invertible matrix A, we have: $\sigma_{max}(A^{-1}) = \frac{1}{\sigma_{min}(A)}$ we conclude that:

$$\frac{1}{\sigma_{min}(B_{i+1})} \leq \frac{1}{\sigma_{min}(D_i)} \cdot \frac{1}{\sigma_{min}(B_i)}$$

Rearranging this last equation, we end up with:

$$\sigma_{min}(B_{i+1}) \geq \sigma_{min}(D_i) \cdot \sigma_{min}(B_i) = \sigma_{min}(B_i) \cdot \frac{n}{n+1}.$$

Again, if we consider all possible sizes, we end up with:

$$\sigma_{min}(B_{i+1}) \geq \frac{1}{2} \cdot \sigma_{min}(B_i) \quad \forall n \in \mathbb{N} \ , \ n \geq 2$$

□

Let us argue now that one can bound the singular values of the matrix B_i throughout the execution of the program.

Minimum Half Axis: First, we claim that if $\sigma_{min}(B)$ is less than $r\epsilon/V$ then the algorithm has already found an ϵ-solution. The scalar ϵ being the wanted precision and the scalars r and V being defined in Section 3.2.

Let us assume $\sigma_{min}(B) < r\epsilon/V$. In this case, E_i is contained in the stripe between two parallel hyperplanes, the width of the stripe being strictly less than $2 \cdot r\epsilon/V$

and consequently E_i does not contain X_θ (defined in equation (21)), where x_* is the minimizer of f_o and $\theta = \epsilon/V$. This argument being a consequence of the fact that X_θ contains a ball of radius $r\epsilon/V$ (by definition).

$$X_\theta = \theta X + (1-\theta)x_* = \{\theta z + (1-\theta)x_* \ , \quad z \in X\} \tag{21}$$

Consequently, there exists $z \in X$ such that $y = \theta z + (1-\theta)x_* \in X_\epsilon$ but $y \notin E_i$, implying by the standard argument that the best value f^+ of f processed so far for feasible solutions satisfies $f^+ \leq f(y) \leq f(x_*) + \theta f(z - x_*)$ which implies that $f^+ \leq f^* + \epsilon$. We can thus stop the algorithm and return the current best point found (feasible and smallest cost).

Maximum Half Axis: We argue in the section that one can modify the original ellipsoid algorithm in order to bound the value of the maximum singular value of B_i. When the largest singular value of B_i is less than, say, $2R\sqrt{n+1}$, we carry out a step as in the basic ellipsoid method. When this singular value is greater than $2R\sqrt{n+1}$, we take some time to "correct "B_i , namely, to pass from $E_i = B_i X$ to $E_i^+ = B_i^+ X$ in such a way that E_i^+ is a localizer along with E_i, meaning that $E_i \cap X \subset E_i^+ \cap X$. In addition to that, we have the following properties:

(a) The volume of E_i^+ is at most γ times the volume of E_i ;

(b) The largest singular value of B_i^+ is at most $2R\sqrt{n+1}$.

For this, let us define $\sigma = \sigma_{max}(B_i) > 2R\sqrt{n+1}$ and let e_o being corresponding direction and index. We then consider the matrix G such that:

$$G = \text{diag}\left(\sqrt{n/(n+1)},\ \sqrt{n+1}/\sigma,\ \ldots,\ \sqrt{n+1}/\sigma\right)$$

We conclude then this case by performing the below update on B_i and c_i:

$$B_{i+1} = B_i \cdot G \quad \text{and} \quad c_{i+1} = c_i - (e_o^T c_i) \cdot e_o \tag{22}$$

Figure 34 shows an illustration of such a correction. The unit ball being the feasible

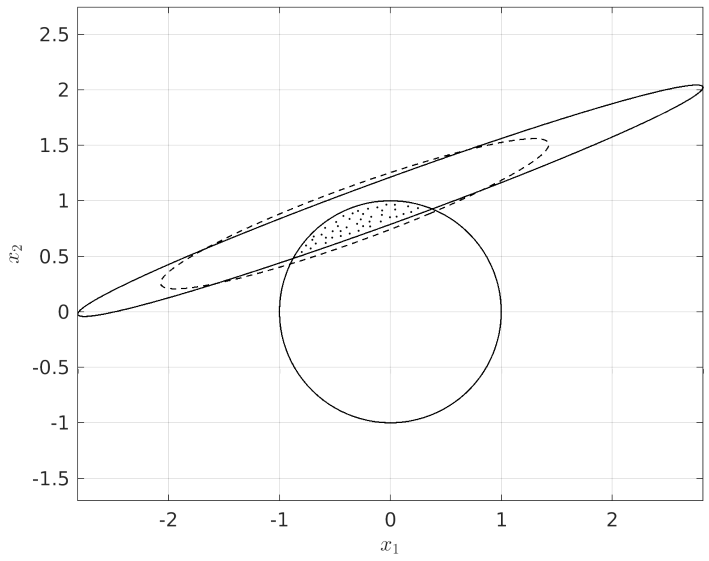


Figure 34: Corrected Ellipsoid

set. Hence, we conclude from this that, throughout the execution of the code we have:

$$\sigma_{min}(B_i) \geq \frac{1}{2}\frac{r\epsilon}{V} = \frac{r\epsilon}{2V} \quad \text{and} \quad \sigma_{max}(B_i) \leq 2 \times 2R\sqrt{n+1} = 4R\sqrt{n+1}$$

We have the following element of proof. First, let us compute the volume of the ellipsoid obtained using this correction step.

$$\text{Vol}(E_i^+) = \det(G) \cdot \text{Vol}(E_i)$$

So,

$$\text{Vol}(E_i^+) = (1 + 1/n)^{(n-1)/2}\sqrt{n+1}/\sigma \cdot \text{Vol}(E_i) \leq \frac{\exp(1/2)}{2R} \cdot \text{Vol}(E_i).$$

Usually, R being quite large, we have $\exp(1/2)/(2R) \leq \gamma$. If we look at this equation more carefully, we can see that having $R \geq \exp(1/2) = 1.65$ implies that $\exp(1/2)/(2R) \leq \gamma$ for all dimension n (we recall that γ depends on n) greater than 2. Hence, as R is usually a substantial scalar (radius of a ball including the feasible set), it will most likely be true. If it is not the case, then we could just make R bigger until we have the wanted property.

4.2.2 Corresponding Condition Number

Let us now see how this process impact the condition number of B. First, from the definition of the condition number we have:

$$k(B) = \|B\| \cdot \left\|B^{-1}\right\| = \frac{\sigma_{max}(B)}{\sigma_{min}(B)}$$

Because of the very well known property of the two norm: $\|A\| = \sigma_{max}(A)$. and for A non-singular:

$$\left\|A^{-1}\right\| = \sigma_{max}(A^{-1}) = \frac{1}{\sigma_{min}(A)}.$$

Thus, by bounding the singular values of B, we concluded on a bound on the condition number of the matrix B.

$$k(B) \leq \left(\frac{2}{1/2} \cdot \frac{2R\sqrt{n+1}}{r\epsilon/V} \right) = \left(\frac{8R\sqrt{n+1}}{r\epsilon/V} \right)$$

and

$$\|B\| = \sigma_{max}(B) \leq 4R\sqrt{n+1}$$

4.2.3 Corresponding norm on c

At each iteration we know that we have:

$$x^* \in \text{Ell}(B, c)$$

Thus,

$$\|x^* - c\| \ = \ \|Bu\| \ \leq \ \|B\| \cdot \|u\| \ \leq \ \|B\|, \qquad \text{for some } u \in B_1(0)$$

Finally,

$$\|c\| \ \leq \ R + \|c\| + \|B\|$$

4.2.4 Consequences on Code

In this section, we explain how to implement this correcting step and give the tools to verify it. In order to detect ellipsoids with large semi-major axes, we need to compute the largest singular value σ_{max} of the current matrix B_k. However, performing a singular value decomposition would be way too expensive and slow (this decomposition being performed at each iteration). On the other hand, because $\sigma_{max}(P)$ is equal to the two norm, we compute an over approximation of the two norm, the Frobenius norm. The latter one being an over approximation of $\sigma_{max}(P)$ and extremely fast to compute, it represents the best option. Because in practice, the successive ellipsoids stay within acceptable bounds, no semi-major axis should be detected. Thus, if we know that the Frobenius norm of the matrix B_i is less than the maximum value acceptable for semi-major axis, we can conclude that the ellipsoid is well conditioned. The equations (23), (24) and (25) show more details about matrix norm equivalence and semi-major axis. We have the following very well known properties corresponding to matrix norms:

$$\|A\|_2 \leq \|A\|_F \leq \sqrt{n}\,\|A\|_2 \quad \forall A \in \mathbb{R}^{n\times n} \tag{23}$$

with:

$$\sigma_{max}(A) = \|A\|_2 \quad \forall A \in \mathbb{R}^{n\times n}. \tag{24}$$

Thus, we have the property needed below:

$$\sigma_{max}(A) \leq \|A\|_F \quad \forall A \in \mathbb{R}^{n\times n}. \tag{25}$$

The mathematical definition of the Frobenius norm of a matrix is presented in definition 6.

Definition 6 *The Frobenius norm of a matrix $A \in \mathbb{R}^{n\times n}$ is:*

$$\|A\|_F \;=\; \sqrt{\sum_{i=1}^{n}\sum_{j=1}^{n} a_{i,j}^2}$$

We axiomatized the Frobenius norm of a vector by being equal to the vector two norm of the “vectorized” matrix. We “vectorize” a matrix by concatenating all its rows in

C Code + ACSL

```c
#include "axiom_def_lin_alg.h"
#include "getNorm_256.h"
#include "socp/sizes.h"

extern double P_minus[N*N];

/*@
  @ ensures \result == normFrobenius(mat_of_16x16_scalar(&P_minus[0]));
  @ assigns \nothing;
*/
double getFrobeniusNorm() {
  return getNorm_256(P_minus);
}
```

Figure 35: getFrobeniusNorm Annotated C Function

a single vector. As an example, we present the ACSL function contract of the C code function computing the Frobenius norm of a matrix in Figure 35. In the case where a too large semi-major axis is detected, The direction e in which this axis lies is also needed. Therefore, we perform a power's iterative algorithm in order to compute this information. Other algorithms could be used in order to get the full decomposition of the current matrix, but does not represent useful information for our use. Work already have been made concerning the formal verification of matrix decomposition algorithms [42].

4.3 Propagating the rounding errors through the algorithm

Let $\mathbb{F}$ denotes the set of all floating-point numbers and $\mathbb{R}$ the set of reals. We use standard notation for rounding error analysis [43, 45, 46], $\mathrm{fl}(\cdot)$ being the result of the expression within the parenthesis computed in rounding to nearest. We write the relative rounding error unit **u** and the underflow unit **eta**. For IEEE 754 double precision (binary64) we have **u**$= 2^{-53}$ and **eta**$= 2^{-1074}$.

We present in this section an analysis targeting the numerical properties of the ellipsoid algorithm. Contributions already have been made concerning finite-precision calculations within the ellipsoid method [30]. However, this work only shows that it is possible to compute approximate solutions without giving exact bounds; it remains very theoretical and only applied to Linear Programming (LP). Also, the analysis performed considers abstract finite-precision numbers and floating-points are not mentioned. Thanks to the analysis performed in this section, using the IEEE standard for floating-point arithmetic and knowing exactly how the errors are being propagated, we would be able to check a posteriori the correctness of the analysis using static analyzers [22, 23, 40].

4.3.1 Preliminaries

Within this algorithm, we focus our attention on the update formulas (12), (13) and (14), allowing us to update the current ellipsoid into the next one. This program dealing with ellipsoids, when investigating its numerical property, the condition number is a decisive information.

Theorem 5 *[Matrix perturbations and Inverse] Let A be a non-singular matrix of $\mathbb{R}^{n\times n}$ and ΔA a small perturbation of A. Then, from [17], we know that,*

$$\frac{\|(A+\Delta A)^{-1} - A^{-1}\|}{\|A^{-1}\|} \leq k(A)\frac{\|\Delta A\|}{\|A\|} \tag{26}$$

4.3.2 Norms and Bounds

To successfully perform the numerical analysis of the algorithm, we need to know how "big" the variables can grow within the execution of the algorithm. Indeed, for a given instruction, the errors due to floating-point arithmetic are usually proportional to the value of the variables.

Bound on variables c, B, $k(B)$ and p.

For the variable p, we have:

$$\|p\| = \frac{\left\|B^T e\right\|}{\sqrt{e^T B B^T e}} = 1 \tag{27}$$

After modifying the original algorithm in the way described in Section 4.2 we have the following results:

$$\|B\| \leq 4R\sqrt{n+1} \tag{28}$$

$$k(B) \leq \frac{8RV\sqrt{n+1}}{r\epsilon} \tag{29}$$

$$\|c\| \leq R + \|x_c\| + \|B\| \tag{30}$$

Where n, R, r, V, x_c and ϵ are the variables described in Section 3.2.

4.3.3 Floating-Point Rounding of Elementary Transformations

In this section, we express the floating-point errors taking place when performing the update formulas (12) and (13). For this, we present first the error analysis for basic operations appearing in the algorithm.

Rounding of a Real. Let $z \in \mathbb{R}$

$$\tilde{z} = \mathrm{fl}(z) = z + \delta + \eta \qquad \text{with } |\delta| < \mathbf{u} \text{ and } |\eta| < \mathbf{eta}/2$$

Product and Addition of Floating-Points. Let $a, b \in \mathbb{F}$.

$$\begin{aligned} \mathrm{fl}(a \times b) &= (a \times b)(1 + \epsilon_2) + \eta_2 \\ \mathrm{fl}(a + b) &= (a + b)(1 + \epsilon_1) \end{aligned}$$

with: $|\epsilon_1| < \mathbf{u}$, $|\epsilon_2| < \mathbf{u}$, $|\eta_2| < \mathbf{eta}$ and $\epsilon_2\eta_2 = 0$

Reals-Floats Product. Let $z \in \mathbb{R}$ and $a \in \mathbb{F}$,

$$|\mathrm{fl}\big(\mathrm{fl}(z) \cdot a\big) - z \cdot a| \leq |z||a| \cdot \mathbf{u} + |a| \cdot 2\mathbf{u}(1 + \mathbf{u})$$

Scalar Product. Let $a, b \in \mathbb{F}^n$. We define,

$\langle a, b \rangle = \sum_{i=1}^{n} a_i b_i$ and $|a, b| = \sum_{i=1}^{n} |a_i b_i|$. We have then:

$$|\mathrm{fl}\langle a, b \rangle - \langle a, b \rangle| \leq A_n |a, b| + \Gamma_n$$

With:

$$A_n = \frac{n \cdot \mathbf{u}}{1 - n \cdot \mathbf{u}} \quad \text{and} \quad \Gamma_n = A_{2n} \frac{\mathbf{eta}}{\mathbf{u}} = \frac{2n \cdot u}{1 - 2n \cdot u} \frac{\mathbf{eta}}{\mathbf{u}}$$

Multiplication of Reals and Floating Scalar Product.

Let $z \in \mathbb{R}$ and $a, b \in \mathbb{F}^n$. We have the following property:

$$|\text{fl}\Big(\text{fl}(z) \cdot \text{fl}\langle a, b\rangle\Big) - z \times \langle a, b\rangle| \leq |a, b| \cdot |z| \cdot 2\mathbf{u}(1+n) + |a, b| \cdot 4\mathbf{u} \tag{31}$$

Now that the propagation of the numerical errors through the elementary transformations have been presented. In the same fashion, we identify the kind of operations performed to update the current ellipsoid. For this, we define Δ_B, $\Delta_{B^{-1}}$ and Δ_c representing the floating-point errors, such that:

$$\Delta_c = \text{fl}(c^+) - c^+ \tag{32}$$

$$\Delta_B = \text{fl}(B^+) - B^+ \tag{33}$$

$$\Delta_{B^{-1}} = \big(\text{fl}(B^+)\big)^{-1} - \big(B^+\big)^{-1} \tag{34}$$

and assume that after performing the floating-point analysis we found $\mathcal{E}_B$ and $\mathcal{E}_c$ such that:

$$|(\Delta_B)_{i,j}| \leq \mathcal{E}_B \quad \forall i, j \in [1, n] \tag{35}$$

and

$$|(\Delta_c)_i| \leq \mathcal{E}_c \quad \forall i \in [1, n]. \tag{36}$$

We dedicated Section 4.3.3 to the computation of $\mathcal{E}_c$ and $\mathcal{E}_B$.

Rounding Error on c^+. Knowing how the errors are being propagated through elementary transformations, we want to compute the error for a transformation similar to what happen for the vector c update. For each component of c, we have:

$$c_i^+ = c_i - 1/(n+1) \cdot \langle \text{Row}_i(B), p\rangle.$$

Therefore, the operation performed, in floating-point arithmetic is:

$$\text{fl}\big(c + \text{fl}\big(\text{fl}(z) \cdot \text{fl}\langle a, b\rangle\big)\big).$$

With: $a, b \in \mathbb{F}^n$, $c \in \mathbb{F}$ and $z \in \mathbb{R}$.

Using the type of floating-point transformation (37) and neglecting all terms in **eta** and powers of $\mathbf{u}$ greater than two we get:

$$\mathcal{E}_c \leq u \cdot \Big(\big(16n^2 + 16n + 3\big) \cdot \|B\| + \|c\| \Big). \tag{37}$$

Error on B^+. Similarly, For each component of B, we have:

$$B^+_{i,j} = \alpha \cdot B_{i,j} + \beta \cdot \langle \mathrm{Row}_i(B), p \rangle \cdot p_j$$

which is of the form, in floating-point arithmetic:

$$\mathrm{fl}\Big(\mathrm{fl}\Big(\mathrm{fl}(z_1) \cdot d \Big) + \mathrm{fl}\Big(\mathrm{fl}(z_2) \cdot \mathrm{fl}\big(\mathrm{fl}\langle a, b \rangle \cdot c \big) \Big).$$

Hence, we found:

$$\mathcal{E}_B \leq u \cdot \|B\| \cdot \Big(\big(n^2/(1 - nu) + 2\big)|\beta| + n + 2|\alpha| + 1 \Big). \tag{38}$$

4.4 Necessary conditions for numerical stability

4.4.1 Problem Formulation

In order to take into account the uncertainties on the variables due to floating-point rounding, we want to modify the algorithm to make it more robust. For this, we choose to evaluate those uncertainties and conclude on a coefficient λ that represents by how much we are going to widen the ellipsoid E_k at each iteration (see Figure 36). Let us assume we have $B \in \mathbb{F}^{n \times n}$, $p \in \mathbb{F}^n$, $c \in \mathbb{F}^n$. We want to find $\lambda \geq 1 \in \mathbb{R}$ such that:

$$\mathrm{Ell}\Big(B^+, c^+ \Big) \subset \mathrm{Ell}\Big(\lambda \cdot \mathrm{fl}(B^+),\ \mathrm{fl}(c^+) \Big). \tag{39}$$

4.4.2 Equivalent and Sufficient Conditions for Covering

In this section, we state two lemmas that give an equivalent and a sufficient condition for the ellipsoid $\mathrm{Ell}\Big(\lambda \cdot \mathrm{fl}(B^+),\ \mathrm{fl}(c^+) \Big)$ to include the ellipsoid $\mathrm{Ell}\Big(B^+, c^+ \Big)$.

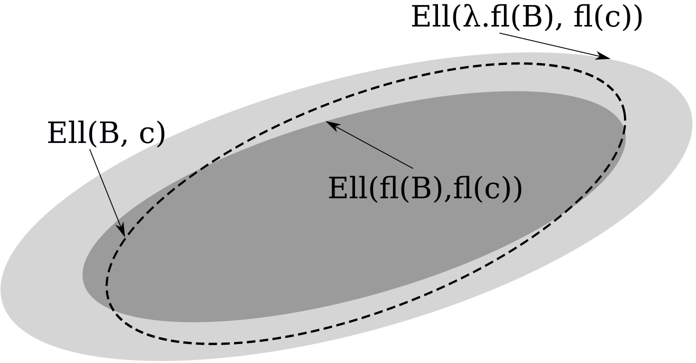


Figure 36: Ellipsoid Widening

Lemma 1 *[Widening - Equivalent Condition]*

$$\mathrm{Ell}\Big(B^+, c^+\Big) \subset \mathrm{Ell}\Big(\lambda \cdot \mathrm{fl}(B^+),\ \mathrm{fl}(c^+)\Big) \iff$$

$$\Big\|\mathrm{fl}\big(B^+\big)^{-1} \cdot \Big(B^+u + c^+ - \mathrm{fl}(c^+)\Big)\Big\| \leq \lambda \quad \forall u, \|u\| \leq 1$$

Proof *Let us denote $B_1(0)$ the unit Euclidean ball. (i.e. $B_1(0) = \{x \in \mathbb{R}^n : x^T x \leq 1\}$). Using definition (10), we have the following equivalent statement:*

$$\forall\ u_1 \in B_1(0)\ ,\ z = B^+u_1 + c^+ \rightarrow \exists\ u_2 \in B_1(0)\ ,\ z = \lambda \cdot \mathrm{fl}(B^+)u_2 + \mathrm{fl}(c^+)$$

Let us now reformulate the second part of the statement.

$$\forall\ u_1 \in B_1(0)\ ,\ z = B^+u_1 + c^+ \rightarrow \exists\ u_2 \in B_1(0)\ ,\ \Big(\lambda \cdot \mathrm{fl}(B^+)\Big)^{-1} \cdot \Big(z - \mathrm{fl}(c^+)\Big) = u_2$$

which is equivalent to:

$$\forall\ u_1 \in B_1(0)\ ,\ z = B^+u_1 + c^+ \rightarrow \Big\|\Big(\lambda \cdot \mathrm{fl}(B^+)\Big)^{-1} \cdot \Big(z - \mathrm{fl}(c^+)\Big)\Big\| \leq 1$$

Let us now replace z by its formula in the second part of the statement.

$$\forall\ u_1 \in B_1(0)\ ,\ \Big\|\Big(\lambda \cdot \mathrm{fl}(B^+)\Big)^{-1} \cdot \Big(B^+u_1 + c^+ - \mathrm{fl}(c^+)\Big)\Big\| \leq 1$$

Putting on the other side of the inequality λ, we end up with the wanted property.

$$\forall\ u \in B_1(0)\ ,\ \Big\|\Big(\mathrm{fl}(B^+)\Big)^{-1} \cdot \Big(B^+u + c^+ - \mathrm{fl}(c^+)\Big)\Big\| \leq \lambda$$

□

We now state lemma 2, which gives a sufficient condition for the coefficient λ to have $\mathrm{Ell}\big(\lambda \cdot \mathrm{fl}(B), \mathrm{fl}(c)\big)$ including $\mathrm{Ell}(B, c)$. If the calculations of B^+ and c^+ were perfect, using lemma 2, we would have $\lambda = 1$ working; no correction is indeed necessary.

Lemma 2 *[Widening - Sufficient Condition]*

$$\left\|\mathrm{fl}\big(B^+\big)^{-1}B^+\right\| + \left\|\mathrm{fl}\big(B^+\big)^{-1}\right\| \cdot \left\|c^+ - \mathrm{fl}(c^+)\right\| \leq \lambda \implies$$
$$\mathrm{Ell}\Big(B^+, c^+\Big) \subset \mathrm{Ell}\Big(\lambda \cdot \mathrm{fl}(B^+),\ \mathrm{fl}(c^+)\Big).$$

Proof *let us assume that*

$$\left\|\mathrm{fl}\big(B^+\big)^{-1}B^+\right\| + \left\|\mathrm{fl}\big(B^+\big)^{-1}\right\| \cdot \left\|c^+ - \mathrm{fl}(c^+)\right\| \leq \lambda \tag{40}$$

The goal here is to end up this the inclusion property (39). *Using the fact that the two norm is a consistent norm, we have:*

$$\forall\ u \in B_1(0)\ ,\ \left\|\mathrm{fl}\big(B^+\big)^{-1}B^+u\right\| \leq \left\|\mathrm{fl}\big(B^+\big)^{-1}B^+\right\| \tag{41}$$

and:

$$\left\|\mathrm{fl}\big(B^+\big)^{-1}\Big(c^+ - \mathrm{fl}(c^+)\Big)\right\| \leq \left\|\mathrm{fl}\big(B^+\big)^{-1}\right\| \cdot \left\|c^+ - \mathrm{fl}(c^+)\right\| \tag{42}$$

Therefore using equations (41), (42) *and* (40) *we have:*

$$\forall\ u \in B_1(0)\ ,\ \left\|\mathrm{fl}\big(B^+\big)^{-1}B^+u\right\| + \left\|\mathrm{fl}\big(B^+\big)^{-1} \cdot \Big(c^+ - \mathrm{fl}(c^+)\Big)\right\| \leq \lambda$$

Then, using the triangle Inequality ($\|x+y\| \leq \|x\| + \|y\|$), we finally have that:

$$\forall\ u \in B_1(0)\ ,\ \left\|\mathrm{fl}\big(B^+\big)^{-1}B^+u + \mathrm{fl}\big(B^+\big)^{-1} \cdot \Big(c^+ - \mathrm{fl}(c^+)\Big)\right\| \leq \lambda$$

yields:

$$\forall\ u \in B_1(0)\ ,\ \left\|\mathrm{fl}\big(B^+\big)^{-1} \cdot \Big(B^+u + c^+ - \mathrm{fl}(c^+)\Big)\right\| \leq \lambda$$

So, using lemma 1 we have:

$$\mathrm{Ell}\Big(B^+, c^+\Big) \subset \mathrm{Ell}\Big(\lambda \cdot \mathrm{fl}(B^+),\ \mathrm{fl}(c^+)\Big)$$

□

4.4.3 Analytical Sufficient Conditions for Covering

From Lemma 2, one can see that the calculation of a widening coefficient λ highly depends on the accuracy of the matrix $\mathrm{fl}(B^+)^{-1}$. Therefore, we also need to compute a number $\mathcal{E}_{B^{-1}}$ such that:

$$|(\Delta_{B^{-1}})_{i,j}| \leq \mathcal{E}_{B^{-1}} \quad \forall i, j \in [1, n]. \tag{43}$$

The quantity $(B^+)^{-1}$ is not used explicitly in the algorithm and its floating-point error could not be evaluated by numerically analyzing the method. Instead, we will use perturbation matrix theory [17] and Theorem 5. This will give us an upper bound on $\mathcal{E}_{B^{-1}}$ given $\mathcal{E}_B$, the norm of B and its condition number. The result is stated in the following lemma.

Lemma 3 *[Widening - Analytical Sufficient Condition]*

$$\lambda \geq 1 + \frac{k(B)}{\|B\|}\sqrt{n} \cdot \left(\sqrt{n} \cdot k(B)\mathcal{E}_B + \mathcal{E}_c + \frac{k(B)}{\|B\|} n\mathcal{E}_B\mathcal{E}_c\right) \implies$$
$$\mathrm{Ell}\Big(B^+, c^+\Big) \subset \mathrm{Ell}\Big(\lambda \cdot \mathrm{fl}(B^+),\ \mathrm{fl}(c^+)\Big).$$

Proof *First, in order to prove this theorem, we need to evaluate* $\|\Delta_{B^{-1}}\|$*. Using equation* (26)*,*

$$\|\Delta_{B^{-1}}\| \leq k(B)\frac{\|B^{-1}\|}{\|B\|}\|\Delta_B\| = \frac{k^2(B)}{\|B\|^2}\|\Delta_B\|$$

But,

$$\|\Delta_B\| \leq \|\Delta_B\|_F \leq n\mathcal{E}_B$$

So, we have:

$$\|\Delta_{B^{-1}}\| \leq \frac{k^2(B)}{\|B\|^2} n\mathcal{E}_B$$

let us now define the three constants below:

$$I = \left\|\mathrm{fl}(B^+)^{-1}B^+\right\|,$$

$$J = \left\| \mathrm{fl}\left(B^{+}\right)^{-1} \right\|$$

and

$$K = \left\| c^{+} - \mathrm{fl}(c^{+}) \right\|.$$

Using equation (57)*:*

$$I = \left\| I_n + \Delta_{B^{-1}} B^{+} \right\| \implies I \leq \|I_n\| + \|\Delta_{B^{-1}}\| \left\| B^{+} \right\| = 1 + \frac{k^2(B)}{\|B\|} n\mathcal{E}_B$$

$$J \leq \left\| (B^{+})^{-1} \right\| + \|\Delta_{B^{-1}}\| = \frac{k(B)}{\|B\|} \cdot \left(1 + \frac{k(B)}{\|B\|} n\mathcal{E}_B\right)$$

and

$$K \leq \sqrt{n}\mathcal{E}_c$$

So, if:

$$1 + \frac{k(B)}{\|B\|}\sqrt{n} \cdot \left(\sqrt{n} \cdot k(B)\mathcal{E}_B + \mathcal{E}_c + \frac{k(B)}{\|B\|} n\mathcal{E}_B\mathcal{E}_c\right) \leq \lambda \implies I + J \cdot K \leq \lambda$$

implying that:

$$\left\| \mathrm{fl}\left(B^{+}\right)^{-1} B^{+} \right\| + \left\| \mathrm{fl}\left(B^{+}\right)^{-1} \right\| \cdot \left\| c^{+} - \mathrm{fl}(c^{+}) \right\| \leq \lambda$$

Using lemma 2 we conclude then on the inclusion property (39). □

Thus, due to this lemma, following the floating-point analysis of the algorithm, we are now able to compute a coefficient λ such that equation (39) is valid. After founding such a λ, we would like to know whether the algorithm is still converging. On the other hand, because the method's proof lies in the fact that the final ellipsoid has a small enough volume, this correction will have an impact of the number of iterations. Lemma 4 addresses those issues.

Lemma 4 *[Convergent Widening Coefficient] Let $n \in \mathbb{N}, n \geq 2$.*

The algorithm implementing the widened ellipsoids, with coefficient λ, will converge if:

$$\lambda < \exp\left(\frac{1}{n(n+1)}\right) \tag{44}$$

In that case, if N denotes the original number of iteration needed, the algorithm implementing the widened ellipsoids will require:

$$N_\lambda = \frac{N}{1 - n(n+1)\log(\lambda)} \text{ iterations} \tag{45}$$

Proof *We recall that the algorithm converges if and only if the volumes of the successive ellipsoids are decreasing.*

$$\text{Vol}\big(\text{Ell}(B_{k+1}, c_{k+1})\big) < \text{Vol}\big(\text{Ell}(B_k, c_k)\big)$$

But we know when using the original update process of the Ellipsoid Method we have:

$$\text{Vol}\big(\text{Ell}(B_{k+1}, c_{k+1})\big) \leq \gamma \cdot \text{Vol}\big(\text{Ell}(B_k, c_k)\big)$$

With $\gamma = \exp(-1/(2(n+1)))$. Also, thanks to equation (11), we know that

$$\text{Vol}\big(\text{Ell}(\lambda \cdot B_k, c_k)\big) = \lambda^{n/2} \cdot \text{Vol}\big(\text{Ell}(B_k, c_k)\big)$$

Therefore, the corrected algorithm, with widening coefficient λ will converge if and only if:

$$\lambda^{n/2} \cdot \gamma < 1$$

Which is equivalent to:

$$\lambda < \exp\left(\frac{1}{n(n+1)}\right)$$

Therefore using equation (11),

$$\text{Vol}(E'_{k+1}) = \lambda^{n/2} \cdot \text{Vol}(E_{k+1})$$

So, we have:

$$\gamma_\lambda = \frac{\text{Vol}(E'_{k+1})}{\text{Vol}(E_k)} = \frac{\text{Vol}(E'_{k+1})}{\text{Vol}(E_{k+1})} \frac{\text{Vol}(E_{k+1})}{\text{Vol}(E_k)} = \lambda^{n/2} \cdot \gamma$$

In order to end up at the final step with an ellipsoid of the same volume, we need:

$$\text{Vol}(E_o) \cdot \gamma_\lambda^{N_\lambda} = \text{Vol}(E_o) \cdot \gamma^N$$

Which implies:

$$N_\lambda \cdot \Big(\log(\gamma) + \frac{n}{2}\log(\lambda)\Big) = N \cdot \log(\gamma)$$

Replacing γ by its value, using property 1:

$$N_\lambda \cdot \Big(\frac{-1}{2(n+1)} + \frac{n}{2}\log(\lambda)\Big) = N \cdot \frac{-1}{2(n+1)}$$

And thus, finding the wanted formula:

$$N_\lambda = \frac{N}{1 - n(n+1)\log(\lambda)}.$$

□

An application of this framework is presented in Section 5.4.

Chapter V

AUTOCODER AND CLOSED LOOP MANAGEMENT

In this Chapter, we give details about the closed-loop management of optimization algorithms. We show how to extract convergence guarantees for parameterized optimization problems. Following this, we present how to use the autocoder GENEMO that we built and how to modify the output generated code. Finally, real-time simulations using generated code for different systems are presented.

5.1 Closed Loop Management – Sequential Optimization Problems

In this section, we are interested in extracting convergence guarantees for a class of optimization problems that are going to be used online. For this, considering the MPC controller input as a parameter, we end up with an optimization problem with parameterized constraints and cost. The work presented in this section is about studying how this parameter affects the constraints and the cost at each iteration and trying to find hypothesis for this parameter to fulfill in order to conclude on the convergence of the algorithm for every points along the trajectory.

First, we focus on the special case where only linear constraints are present. Following this, another section will be dedicated to the more general setting of SOCP constraints.

5.1.1 Parameterized Linear Constraints

Let us write the optimization problem we are aiming to solve in real-time. The vector $X \in \mathbb{R}^{n_x}$ denotes the decision vector.

$$\begin{aligned} \underset{X}{\text{minimize}} \quad & f_o(X) \\ \text{subject to} \quad & AX \leq b \end{aligned} \tag{46}$$

Let us assume the initialization of this optimization problem is being done such as equation (47), where S is a full rank matrix. Usually, S represents a selector matrix and is of the form: $[I_p \; O_{p\times(n_x-p)}]$. In that case it is obviously full rank. x_o denotes the input of the controller. We write $\hat{x_o}$ to account for the fact that x_o will change from one optimization problem to another.

$$S \cdot X = \hat{x_o} \tag{47}$$

We decompose and separate the equality and inequality constraints hidden behind the original matrix A and vector b. That way, Problem 46 can be written as:

$$\begin{aligned} \underset{X}{\text{minimize}} \quad & f_o(X) \\ \text{subject to} \quad & A_{eq} \cdot X = b_{eq} \\ & A_{ineq} \cdot X \leq b_{ineq} \\ & S \cdot X = \hat{x_o} \end{aligned} \tag{48}$$

The idea here is to project all the equality constraints in order to eliminate it while keeping track of the variable parameter, $\hat{x_o}$. We know there exist matrices M, A_1 and A_2 such that for all vector $X \in \mathbb{R}^{n_x}$ satisfying the equality constraints of Problem 48, there exists a vector $Z \in \mathbb{R}^{n_z}$ such that:

$$X = A_1 \cdot b_{eq} + A_2 \cdot \hat{x_o} + M \cdot Z \tag{49}$$

M being a matrix formed by an orthonormal basis of the null space of the matrix $\begin{bmatrix} A_{eq} \\ S \end{bmatrix}$. d denotes the total number of equality constraints (number of rows of A_{eq}

+ number of rows of S). Then $Z \in \mathbb{R}^{n_z}$ with $n_z = n_x - d$. Thus, the original optimization problem 48 is equivalent to the below projected problem.

$$\begin{aligned} \underset{Z}{\text{minimize}} \quad & f_o(A_1 \cdot b_{eq} + A_2 \cdot \hat{x_o} + M \cdot Z) \\ & A_f \cdot Z \leq b_{f,o} + A_{f,l} \cdot \hat{x_o} \end{aligned} \tag{50}$$

With:

$$A_f = A_{ineq} \cdot M \quad \text{and} \quad b_{f,o} = b_{ineq} - A_{ineq} \cdot A_1 \cdot b_{eq} \quad \text{and} \quad A_{f,l} = -A_{ineq} \cdot A_2 \tag{51}$$

More details and references about equality constraints elimination can be found at [10]. In order to solve this optimization problem with the Ellipsoid Method and have convergence guarantees for every possible x_o, we need to compute geometric characteristics on the feasible set of this parametric optimization problem. For that reason, for now, let us assume: $\|x_o\|_2 \leq r_o$.
We define the parameterized polyhedral set:

$$P_{\hat{x_o}} = \{z \in \mathbb{R}^{n_z} : A_f \cdot z \leq b_{f,o} + A_{f,l} \cdot \hat{x_o}\} \tag{52}$$

Having this collection of polyhedral sets, we want to compute ball radii that will tell us about the volume of the feasible set that would be true for every initialization point x_o. Beforehand, let us define the operator $\phi(.)$ that returns for a matrix A the vector $\phi(A)$ whose coordinates are the two norm of the rows of A. Let us now consider the two extreme polyhedral sets below:

$$P_l = \{z \in \mathbb{R}^{n_z} : A_f \cdot z \leq b_{f,o} - r_o \cdot \phi(A_{f,l})\} \tag{53}$$

$$P_u = \{z \in \mathbb{R}^{n_z} : A_f \cdot z \leq b_{f,o} + r_o \cdot \phi(A_{f,l})\} \tag{54}$$

In order to give an example of this concept, we show in Figure 37 an illustration of such polyhedral sets (the illustrated sets have no physical meaning and do not represent any MPC problem. We used the values below:

$$A_f = \begin{bmatrix} -1 & 1 \\ 1 & 1 \\ 1 & -0.5 \\ 0 & 1 \\ -1 & 0 \\ 0 & -1 \end{bmatrix} \quad ; \phi(A_{f,l}) = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} \quad ; b_{f,o} = \begin{bmatrix} 1 \\ 2 \\ 1 \\ 1.5 \\ 0.5 \\ 0.5 \end{bmatrix} \quad ; r_o = 0.5.$$

We state an important Lemma, illustrating the fact that for every point along the trajectory, the feasible set of the current optimization problem is bounded by the two extreme polyhedral sets defined previously. We also provide the proof for it.

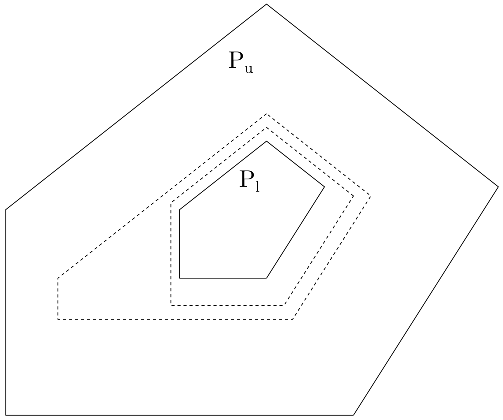


Figure 37: P_u and P_l Polyhedral Sets

Fact 1 *[Extreme Polyhedral Sets]*

$$\forall x_o \in \mathbb{R}^n \; s.t. \; \|x_o\|_2 \leq r_o \quad , \quad P_l \subset P_{x_o} \subset P_u$$

Proof *Let us take x_o such that $\|x_o\|_2 \leq r_o$.*

First, let us establish a very simple inequality using Cauchy-Schwarz inequality:

$$\begin{aligned} |(A_{f,l} \cdot x_o)(i)| &= |row(A_{f,l}, i)^T \cdot x_o| \\ &\leq \|row(A_{f,l}, i)\|_2 \cdot \|x_o\|_2 \\ &\leq \phi(A_{f,l})(i) \cdot r_o \quad \forall i \end{aligned} \tag{55}$$

Therefore,

$$-\phi(A_{f,l}) \cdot r_o \leq A_{f,l} \cdot x_o \leq \phi(A_{f,l}) \cdot r_o$$

Then, if $x \in P_l$ we have $A_f \cdot x \leq b_{f,o} - r_o \cdot \phi(A_{f,l})$. Using the inequality below, it is clear that it implies $x \in P_{x_o}$. The same way, assuming that $x \in P_{x_o}$ and using the inequality below, it is clear that we have $x \in P_u$. □

The work of the control engineer consists in finding three scalars r, R and V such that:

$$\exists\ \bar{z_1} \ \text{ such that } \ B(\bar{z_1}, r) \subset P_l \quad , \tag{56}$$

$$\exists\ \bar{z_2} \ \text{ such that } \ P_u \subset B(\bar{z_2}, R) \quad , \tag{57}$$

$$V \geq \max_{z \in P_{\hat{x_o}}} \ f_o(x_{sol} + Mz) - \min_{z \in P_{\hat{x_o}}} \ f_o(x_{sol} + Mz) \quad , \ \forall \quad \|\hat{x_o}\| \leq r_o. \tag{58}$$

For the first scalar, r, one can compute a numerical value by running an off-line optimization problem finding the largest ball inside P_l. If no solution can be found, the value of r_o need to be decreased, and we repeat the process until finding an acceptable r_o and radius r. Further information about finding the largest ball in a polytope can be found at [10].

As a consequence of equality constraint elimination, and recalled in Eq. 49, we have the relation below between the original decision vector X and the projected one Z:

$$X = A_{proj} \cdot \begin{bmatrix} b_{eq} \\ \hat{x_o} \end{bmatrix} + MZ \tag{59}$$

We decompose now the decision vector X into two parts, $\mathbf{x}$ and $\mathbf{u}$. The part $\mathbf{u}$ being bounded due to constraints in the original optimization problem. If no constraints on $\mathbf{u}$ were originally present, one can add some making the problem bounded and simpler to analyze. The point here being that from bounded variables within the vector X, one can conclude on bounds on the projected vector Z. There is no need to have original bounds particularly on the collection of future inputs $\mathbf{u}$. Rewriting equation 59 yield:

$$\begin{bmatrix} \mathbf{x} \\ \mathbf{u} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} b_{eq} \\ \hat{x_o} \end{bmatrix} + \begin{bmatrix} M_1 \\ M_2 \end{bmatrix} Z \tag{60}$$

Following this, we would have:

$$Z = M_2^{-1} \cdot \big(\mathbf{u} - A_{21} b_{eq} - A_{22} \hat{x_o} \big) \tag{61}$$

and therefore assuming again that $\|\hat{x_o}\| \leq r_o$, one can compute a value of R such that:

$$\|Z\| \leq \left\| M_2^{-1} \right\| \cdot \big(\|\mathbf{u}\| + \|A_{21} b_{eq}\| + \|A_{22}\| \, r_o \big) = R \tag{62}$$

On the other hand, from the physical meaning of the variables and the constraints of the optimization problem, one can conclude on bounds in which the variables should live in, and therefore find a lower bound for V.

Additionally, when the original problem is too hard to analyze one can add constraints in order to make the analysis easier. Adding constraints in order to find acceptable values for the scalars R and V more easily is a very efficient technique but can have a negative effect on the value of r. When adding constraint to a given optimization problem, its feasible set get smaller and it is harder to find a inscribed ball that lies inside.

We now state the general lemma, giving an upper bound on the number of iteration throughout a trajectory.

Fact 2 *[MPC Ellipsoid Method Convergence]*

Let us assume we want to run the problem 48 on-line in order to implement a receding horizon control.

That way, using the Ellipsoid Method and initializing the first Ellipsoid by $B(\bar{z}_2, R)$, the method will find an ϵ-solution using N_{it} iteration for all x_o such that $\|x_o\| \leq r_o$, *with:*

$$N_{it} = 2 \cdot n_z(n_z+1) \cdot \log\Big(\frac{R}{r}\frac{V}{\epsilon}\Big) \qquad and \qquad n_z = dim\Bigg(Null\bigg(\Big[A_{eq} \quad S\Big]^T\bigg)\Bigg)$$

Proof *Let us assume, we are running this problem online and we are currently trying to solve this problem for a given point. We assume x_o such that $\|x_o\| \leq r_o$. Let us write X_f to denote the feasible set of the current optimization problem to solve. that way, we know that we have Eq. 56 Thus, by definition of* r*, we know that:*

$$\exists \bar{z_1} \ \ such\ that \ \ B(\bar{z_1}, r) \subset P_l \subset X_f$$

The same way, because we assume R satisfies Eq. 57, we also have:

$$\exists \bar{z_2} \ \ such\ that \ \ X_f \subset B(\bar{z_1}, r)$$

We also have the scalar V meeting the wanted requirement. Finally, we see that all the needed hypothesis are fulfilled and we can conclude that the returned point will indeed be ϵ-optimal using N_{it} iteration thanks to Theorem 4. □

Linear Programs: In the case of Linear Programs, the method developed is identical and having a linear cost, the scalar V is easily found by running the two optimization problems below.

$$\begin{array}{ll} \underset{\hat{x_o},z}{\text{minimize}} & c^T \cdot (A_1 b_{eq} + A_2 \hat{x_o} + Mz) \\ & A_f \cdot z \leq b_{f,o} + A_{f,l} \cdot \hat{x_o} \\ & \|\hat{x_o}\|_2 \ \leq \ r_o \end{array} \qquad \begin{array}{ll} \underset{x_o,z}{\text{maximize}} & c^T \cdot (A_1 b_{eq} + A_2 x_o + Mz) \\ & A_f \cdot z \leq b_{f,o} + A_{f,l} \cdot x_o \\ & \|x_o\|_2 \ \leq \ r_o \end{array}$$

Having $f_o(x) = c^T x$.

5.1.2 Second-Order Conic Constraints

In the previous section, we used the fact that we had linear constraints to construct extreme polyhedral sets and conclude on geometric characteristics on feasible set. In this section we aim at giving ways of computing the same needed constants for the general setting of second-order constraints. Let us now consider we want to solve a model predictive control optimization problem of the form:

$$\begin{aligned} \underset{X}{\text{minimize}} \quad & f_o(X) \\ & \|A_i \cdot X + b_i\|_2 \ \leq \ c_i^T X + d_i \ \ , \ i = 1 \ldots m \\ & A_{eq} X = b_{eq} \\ & S \cdot X = \hat{x_o} \end{aligned} \tag{63}$$

First, let us assume that no equality constraint are hidden in the second-order constraints (if it is not the case, a very simple analysis will confirm it and one can extract those equality constraints and put it into the couple (A_{eq}, b_{eq})). Unfortunately, even after eliminating the equality constraints of problem 63, in order to find a value for the scalar r, we need to find the largest balls inside a second-order cone, which is not an easy task. To get around this, we use the equivalence of norm in finite dimensions (see Eq. 64, 65 and Figure 38), noting that $\|\cdot\|_1$ and $\|\cdot\|_\infty$ can be expressed with linear constraints. We perform a linear relaxation, by replacing the second-order constraints with stronger constraints that can be expressed with linear constraints. That way, the feasible set becomes a polyhedron and the previous tools could be applied. For instance, problem 66 represents a linear relaxation of the original problem 63.

$$\|x\|_\infty \leq \|x\|_2 \leq \sqrt{n} \cdot \|x\|_\infty \tag{64}$$

$$n^{-1/2} \cdot \|x\|_1 \leq \|x\|_2 \leq \|x\|_1 \tag{65}$$

$$\begin{aligned} \underset{X}{\text{minimize}} \quad & f_o(X) \\ & \sqrt{n} \cdot \|A_i \cdot X + b_i\|_\infty \ \leq \ c_i^T X + d_i \ \ , \ i = 1 \ldots m \\ & A_{eq} X = b_{eq} \\ & S \cdot X = \hat{x_o} \end{aligned} \tag{66}$$

Figure 39 shows an illustration of a second-order linear relaxation, along with the largest ball inside the relaxed polyhedral set.

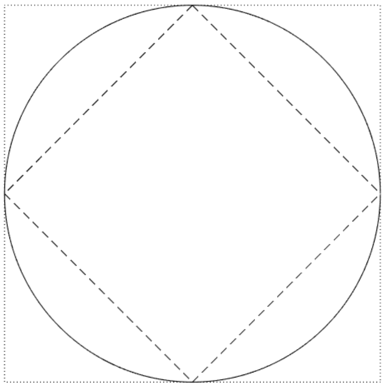


Figure 38: Unit balls of $\mathbb{R}^2$

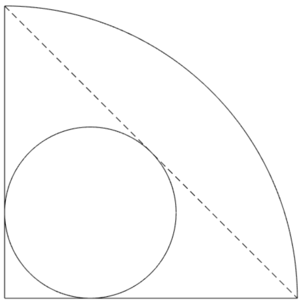


Figure 39: Linear Relaxation of a Second Order Cone

5.2 Running Time Evaluation

In this chapter, we analyze the online use of optimization algorithms in order to extract convergence guarantees for every point along the trajectory. In Section 5.1, we showed how one can decomposed the online optimization problem as a parametrized one and compute an a priori upper bound on the number of iterations needed to be performed for every points along the trajectory. Those algorithms are run online and have very strict time constraint on their execution time. Therefore, the key information here is the running time of those algorithms, which could be directly evaluated from the number of iterations. From the floating-point operations per second (flops) relative to each processor, one can therefore evaluate this execution time for a given problem size and number of iterations. In this thesis, we only highlight that it is a needed work in order to predict feasibility of the real-time use of those algorithms but do not present such analysis.

5.3 GENEMO Programming Language Syntax

5.3.1 Using Genemo and Autocode Credible Implementation of Optimization Algorithms

The input text file used to formulate an optimization problem or a MPC controller consists in several different sections. Each sections are introduced by a given keywords. A section, introduced by the keyword "Constants" allows the user to introduce constants that could be used throughout the GENEMO file. The defined constants could be scalars, vectors or matrices. The user can give information about the objective function using the keywords "Minimize". Because most MPC controllers are trying to minimize a sum of quadratic form on the state or input along the trajectory, the sum function has been implemented in the autocoder. Thus encoding of this type of cost function becomes really necessary. The sum function implemented in the autocoder, follows the syntax described in equation 67.

GENEMO

```
Constants
N  = 10;
Ts = 0.01;
A = [1 Ts;-Ts 1];
B = [0;Ts];
uMax = 5;
positionMax = 10;
speedMax = 10;
M = N-1;
Q = [5 0;0 1];
xinit = [2;-1];
```

Figure 40: Constants Section For The Spring-Mass System

$$\text{sum(f(k), k=1..N)} = \sum_{k=1}^{N} f(k) \tag{67}$$

The user can access the norm of a vector x by writing $||x||$. On the other hand, if x is a matrix, $x(:,k)$ denotes the k-th column of the matrix x. Figure 41 presents an example of a objective function written in GENEMO. Constraints are encoded in with the use of the keyword “subjectTo”. A constraint is of the form:

GENEMO

```
Minimize
sum ( || Q * x (: , k ) || , k =1.. N )
```

Figure 41: Objective Section For The Spring-Mass System

“constraintj: ... ; ”.

The same way we used a receding index variable to implement the sum function, we also used such index to define sliding constraints. Figure 42 presents an example of constraints written in GENEMO. Also, with the use of the keyword “Information”, one can incorporate the a priori information needed described in Chapter 3 (see Figure 43). The next and last section that will be discussed deals with the fact that the GENEMO file could implement an MPC controller but also a single optimization problem. In the latter case, the goal being to solve and give the solution of a given single optimization problem and the sections described below (Input/Output sections)

GENEMO

```
SubjectTo
constraint1: x (: ,1) = xinit;
constraint2: x (: , k +1) = A * x (: , k ) + B * u (: , k ) , k =1..N-1;
constraint3: || u (: , k )) || <= uMax, k =1..N-1;
constraint5: x (1 , k ) <= positionMax, k =1..N;
```

Figure 42: Constraint Section For The Spring-Mass System

GENEMO

```
Information
r = 4.05;
R = 26.47;
V = 282.84;
eps = 1e-1;
lambda = 1.000554155008;
```

Figure 43: Information Section For The Spring-Mass System

should be ignored. On the other hand, if the purpose of using this autocoder was to automatically generate C code implementation of MPC controller, the user has to complete these sections in order to give information about the Input/Output aspect of the MPC controller. Figure 44 presents the input and output sections completed for the spring-mass example. Comments can be added along the code thanks to the sign #. Only single line comments are managed.

GENEMO

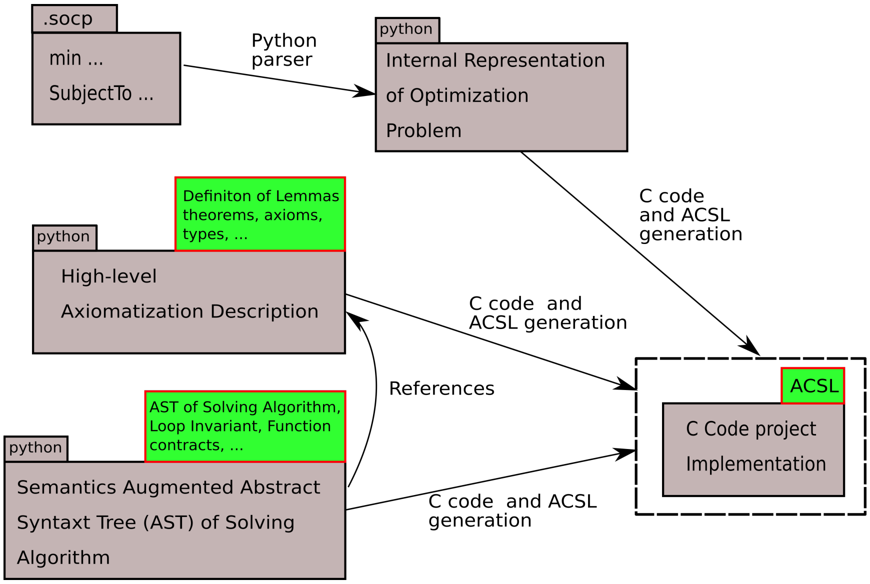
```
Input
xo(2)
Output
u(:,1)
#This is a comment
```

Figure 44: Input/Output Sections For The Spring-Mass System

Several examples of GENEMO files are presented in Appendix C. A single optimization problem formulated in GENEMO is presented in Figure 66. GENEMO MPC controllers formulation are shown in Figures 67 and 68.

5.3.2 Internal Aspect of The Autocoder

In this section we give details about the internal aspect of the autocoder. The user first has to formalize the MPC controller desired and write the corresponding input file detailed in Section 5.3. In addition to this, the user has to specify the implementation of the method used to solve optimization problems. By default, this autocoder has been created in order to generate C code implementation of the Ellipsoid Algorithm. Nevertheless, the C code generator is generic and could be applied to any algorithms. In order to generate C code implementation of another solving method, the user has to give and formalize the augmented AST (Abstract Syntax Tree) using a python library wrote beforehand. This library implements all the necessary properties relative to trees, AST and C code generation. Additionally, this library has been written in a way to facilitate the augmentation for semantics and Hoare Triples to those AST. By default, the usual control structures will be autocoded along with annotations. For instance, when creating a node relative to a for loop using the AST library, the autocoder will by default generate the C code implementation of the for loop structure along with trivial loop invariants, loop variant and assigns clauses. That way, a considerable amount of annotations, required for the proof, are handled directly by the autocoder which makes the development process faster.



.socp
min ...
SubjectTo ...
Python parser
python
Internal Representation of Optimization Problem
C code and ACSL generation
Definiton of Lemmas theorems, axioms, types, ...
python
High-level Axiomatization Description
C code and ACSL generation
References
ACSL
C Code project Implementation
AST of Solving Algorithm, Loop Invariant, Function contracts, ...
python
Semantics Augmented Abstract Syntaxt Tree (AST) of Solving Algorithm
C code and ACSL generation

Figure 45: Internal Aspect of The Autocoder

5.4 Real-Time Simulations and Examples

5.4.1 Spring-Mass System

We present in this section an application of this framework for the spring-mass system described in Section 2.3, Figure 14. For this, we use again a discretization period of $T = 0.1$ *sec* and an horizon of $N = 10$.

We recall the discrete state-space realization matrices A and B of this system below:

$$A = \begin{bmatrix} 1 & T \\ -T & 1 \end{bmatrix} \quad ; \quad B = \begin{bmatrix} 0 \\ T \end{bmatrix}.$$

Let us assume the initial point at $t = 0$ is:

$$x_o = \begin{bmatrix} 2 & -1 \end{bmatrix}^T ,$$

and the matrix Q equal to:

$$Q = \begin{bmatrix} 10 & 0 \\ 0 & 1 \end{bmatrix} .$$

We want to generate C code in order to perform the below MPC for this system. The goal is to stabilize the origin as fast as possible (minimizing the norm of the state along the future trajectories).

$$\begin{aligned} \underset{X=[\mathbf{x},\mathbf{u}]}{\text{minimize}} \quad & \sum_{k=1}^{N} \quad \|Q \cdot x_k\| \\ & x_{k+1} = Ax_k + Bu_k \ , \ k = 1..N-1 \\ & -5 \le u_k(1) \le 5 \ , \ k = 1..N-1 \\ & -10 \le x_k(1) \le 10 \ , \ k = 1..N \\ & -10 \le x_k(2) \le 10 \ , \ k = 1..N \\ & x_1 = x_i \end{aligned} \tag{68}$$

We use an accuracy of $\epsilon = 0.1$ and using the method developed in the previous sections, we found:

$$r = 4.55, \quad R = 18.19, \quad V = 141.42.$$

From that, we conclude that the program will need $N_{it} = 1556$ iterations in order to compute an ϵ-solution. Using double precision floating points, we can also conclude on a widening coefficient $\lambda = 1 + 5.2 \cdot 10^{-5}$ to apply in order to account for the rounding errors. As stated in Section 4.3, a new number of iterations of $N_\lambda = 1563$. is now needed. In order to automatically generate the C code used for the control, we used to GENEMO input file presented in Appendix C.3. We repeated the same analysis for different horizon values. The results are collected in Tables 1 and 2.

N	r	R	V	n_z	N_{it}	$f^*(Ell)$	RunTime(s)	$f^*(CVX)$
5	4.79	12.24	70.71	4	300	5.1565	0.001	5.15648
7	4.69	14.90	98.99	6	677	6.9293	0.003	6.92928
10	4.54	18.19	141.42	9	1556	9.3116	0.010	9.31154
15	4.30	22.67	212.13	14	3916	12.6910	0.050	12.6910
20	4.05	26.47	282.84	19	7466	15.6975	0.169	15.6974

Table 1: Performances For Different Problem Sizes

N	N_{it}	λ $(double)$	$N_{\lambda\ (double)}$
5	300	$1 + 5.9e^{-7}$	301
7	677	$1 + 5.2e^{-6}$	678
10	1556	$1 + 5.2e^{-5}$	1563
15	3916	1.000709	4548
20	7466	1.004669	Not Converging

Table 2: Floating Point Consideration

5.4.2 The 3 DOF Helicopter

A picture of the 3 DOF helicopter is presented in Figure 46. We used this system in order to perform a running example for the framework developed in this thesis. The vector state of the system is $x = [\lambda\ \psi\ \phi\ \dot{\lambda}\ \dot{\psi}\ \dot{\phi}]^T$ and the system's inputs are the front and back DC motors voltages. λ denotes the elevation angle, ψ the pitch angle and ϕ

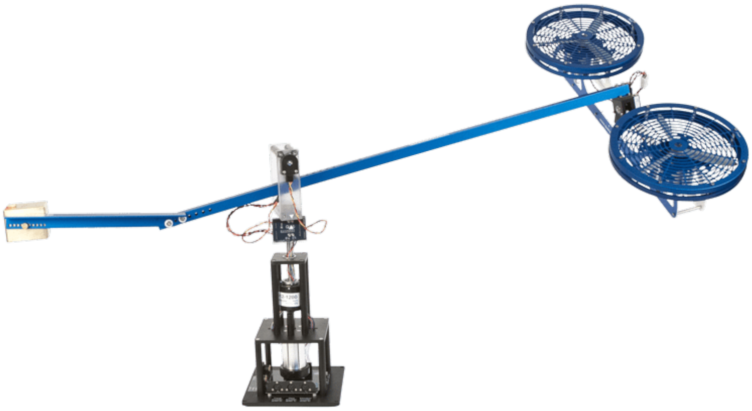


Figure 46: 3 DOF Helicopter

the travel angle. The different axis are presented in figure 47. We use a closed-loop

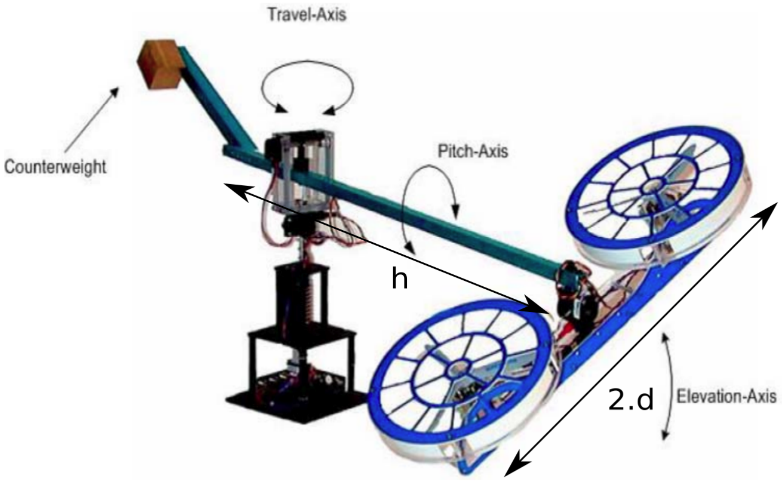


Figure 47: 3 DOF Helicopter Axis and Dimensions

system with a inner feedback controller, and a discretization step of $T = 0.5$ *sec*. The resulting system is a stable linear system that we control using MPC. The problem we are trying to solve is supposed to model a landing of the 3 DOF helicopter. We initialize the system with an angle of 25 *deg* in elevation and 15 *deg* travel and we want to make the system go back and land at the origin while avoiding the ground. The ground is the area below $\lambda = 0$. Thus, the constraint in order to avoid the ground is a combination of the elevation and pitch angle and can be formulated as:

$$h \sin(\lambda) \pm d \sin(\phi) >= 0$$

We linearized those constraints, assuming small angles perturbations, and naturally end up with linear inequalities of the form: $A_{obs} \cdot x \leq b_{obs}$. The MPC controller that we want to implement is formalized in problem 69.

$$\begin{aligned}
\underset{X=[\mathbf{x},\mathbf{u}]}{\text{minimize}} \quad & \sum_{k=1}^{N} \quad \|x_k\| \\
& x_{k+1} = Ax_k + Bu_k \ , \ k = 1..N-1 \\
& \|u_k\| \leq 50 \ , \ k = 1..N-1 \\
& \|x_k\| \leq 200 \ , \ k = 2..N \\
& A_{obs} \cdot x_k \leq b_{obs} \ , \ k = 2..N \\
& -40 \leq x_k(2) \leq 40 \ , \ k = 1..N-1 \\
& x_1 = \hat{x_o}
\end{aligned} \tag{69}$$

Assuming that $\|x_o\| \leq 30$, we found, using the method developed in section 5.1 a radius 8.0612 (running an off-line optimization problem that finds the largest ball inside P_l). For R, using the method described in 5.1 and equation 62 we found $R = 322$. From the problem formulation, one can see that if the problem is feasible, the sum of the norm of the successive x is supposed to be minimized. That way, the worst case is when x_o had the largest norm, and when the system stay at this point throughout the whole trajectory (or keep the same norm at least). That way, we should have a V of:

$$V = N \cdot \|x_o\| \ \leq \ N \cdot 27 = 162 \tag{70}$$

For all those cases, we can also conclude on a widening coefficient lambda to apply in order to control floating-point errors. The method we use to control the floating points error is affecting the volume ratio of the method. Therefore, in order to account for it, we need to do more iteration depending on the value of the widening coefficient λ. Using double precision floating points and a accuracy of $\epsilon = 0.25$ we have the following results:

$$\lambda = 1.0063428$$

An original number of iteration equal to $N = 5528$ and therefore an updated number of steps of: $N_\lambda = 6817$. Doing the simulation on a Intel Core i5-3450 CPU @ 3.10GHz × 4 processor we have a running time of approximately 0.2 *sec* for a single point and therefore it took the computer 4 *sec* to simulate 10 *sec* (because we discretized the system at $2Hz$). The results of the simulation are presented in Figures 49 and 50. We used the text file presented in Figure 68 in appendix C to generate the C code in order to performed the simulation. The whole simulation was performed using the generated C code and the Simulink model presented in Figure 48. The block “MPC CONTROLLER”, with the help of the Matlab coder, compiles and calls the generated and annotated C code.

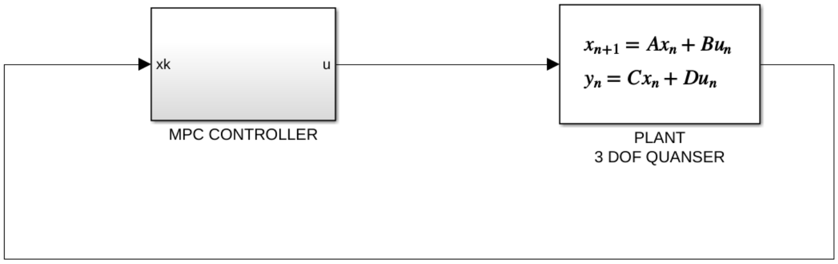


Figure 48: Simulink File Used for the Quanser Simulation

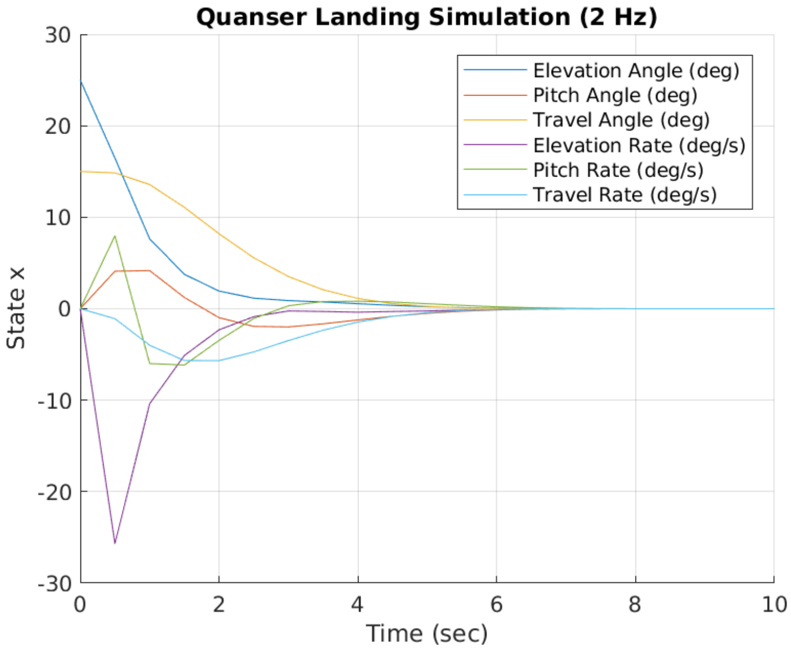


Figure 49: Simulation of State Vector versus time

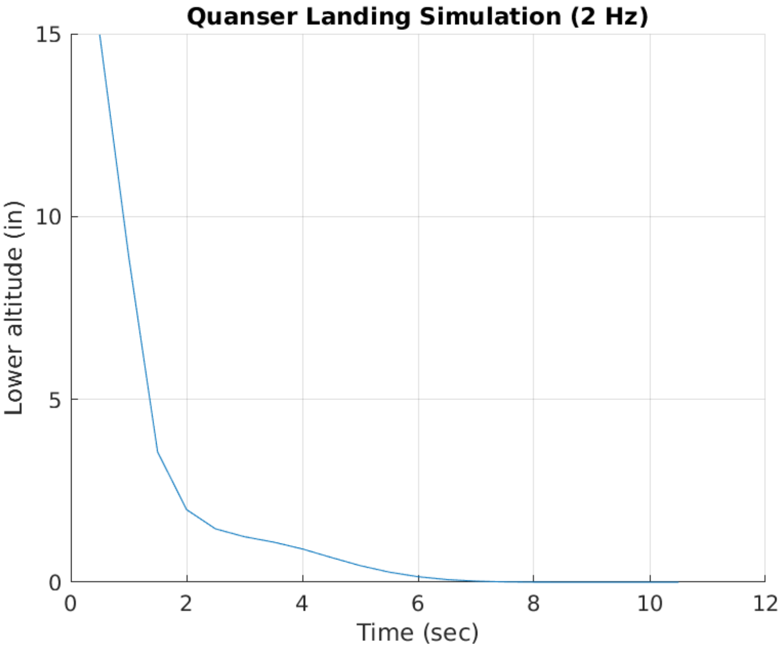


Figure 50: Simulation of Lowest Altitude versus time

5.4.3 Quadcopter Drones

We present in this section an application of this framework for Quadcopter Drones. For this we used a linear discrete-time representation of the drone of the form:

$$x_{k+1} = Ax_k + B_1u_k + B_2u_{k+1} + fd. \tag{71}$$

The vector input u collects the vector thrust and the vector x regroups the position and velocities and the drone along each axes. The matrix A, B_1 and B_2 are described below. The mass of the system is written m , T_s denotes the sampling period and g, the gravity constant.

In this section, we implement a path-planner based on the autocoder we built. For this, we generate beforehand a nominal trajectory using convex optimization techniques. We formulate an optimization problem, to compute an optimal trajectory for the drone to go from a point A to a point B, while satisfying constraints.

One of the constraints to respect is the fact that the input of this system, the thrust

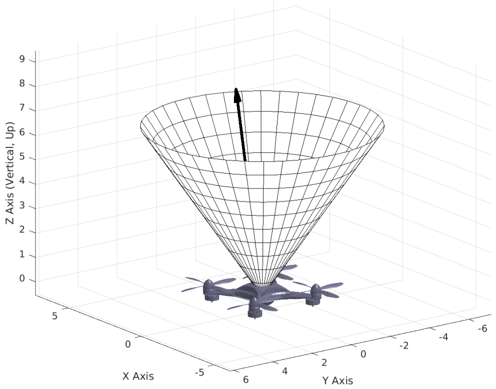


Figure 51: Cone constraint on drone

vector u, needs to stay within a cone from the vertical. This last constraint models the fact that we do not want the drone to rotate too much because non-linearities phenomenon will rise and it will be difficult to control it. One of the problem, was the fact that this constraint is not convex. For that, we perform a lossless convexification step adding slack variables in order to encode this constraint in a convex manner. More details about lossless convexification can be found at [2]. We end up with the optimization problem presented in equation 72. Following the method described in Section 5.1, we found the values for the variable r, R and V listed in Table 3. A plot

N	r	V	R	N_{it}
4	0.4648	13.9245	13.9245	297
5	0.1454	15.5680	64.9628	1474
6	0.2796	17.0539	144.4199	3267
7	0.3359	18.4204	242.6074	5920
8	0.3457	19.6922	359.5805	9504
9	0.3535	20.8867	496.0807	14007
10	0.3617	22.0165	652.8481	19449
11	0.3692	23.0911	830.4812	25860

Table 3: Performances For different Horizon – Drone

of the MPC simulation applied for the quadricopter using the autocoded C code is shown in figure 52.

$$A = \begin{bmatrix} 1 & T_s & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & T_s & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & T_s \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \quad ; \quad fd = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0.5 \cdot T_s \cdot T_s \cdot g \\ T_s \cdot g \end{bmatrix}$$

$$B_1 = \begin{bmatrix} (1/3) \cdot T_s \cdot T_s/m & 0 & 0 \\ (1/2) \cdot T_s/m & 0 & 0 \\ 0 & (1/3) \cdot T_s \cdot T_s/m & 0 \\ 0 & (1/2) \cdot T_s/m & 0 \\ 0 & 0 & (1/3) \cdot T_s \cdot T_s/m \\ 0 & 0 & (1/2) \cdot T_s/m \end{bmatrix}$$

$$B_2 = \begin{bmatrix} (1/6) \cdot T_s \cdot T_s/m & 0 & 0 \\ (1/2) \cdot T_s/m & 0 & 0 \\ 0 & (1/6) \cdot T_s \cdot T_s/m & 0 \\ 0 & (1/2) \cdot T_s/m & 0 \\ 0 & 0 & (1/6) \cdot T_s \cdot T_s/m \\ 0 & 0 & (1/2) \cdot T_s/m \end{bmatrix}$$

$$\begin{aligned} \underset{X=[\mathbf{x},s,\mathbf{u}]}{\text{minimize}} \quad & \|s\| \\ & x_{k+1} = Ax_k + B_1 u_k + B_2 u_{k+1} + fd. \ , \ k = 1..N-1 \\ & \|u_k\| \leq s(k) \ , \ k = 1..N \\ & T_{min} \leq s(k) \leq T_{max} \ , \ k = 1..N \\ & -u_k(3) \geq s(k) \cdot \cos(\phi_{max}) \\ & x_k(5) \geq 0 \\ & x_1 = x_i \ ; \ u_1 = u_i \ ; \ x_N = x_f \ ; \ u_N = u_f \end{aligned} \tag{72}$$

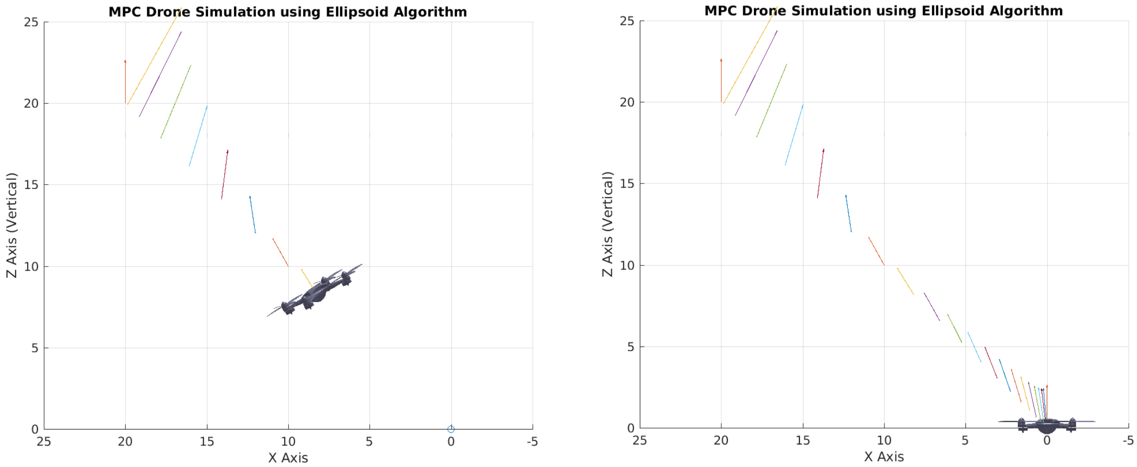


Figure 52: MPC Simulation for a Quadcopter

Chapter VI

CONCLUSION

One of the goals of this research was to fill the gap between computer science, optimization and control theory. We used fundamental results in those areas with the hope of advancing V&V techniques.

In Chapter 1, we introduced the subject, gave details about the needed elements and presented a literature review. Following this, we recalled some important results of control theory for understanding the application we are targeting in this research in Chapter 2. Chapter 3 presents an axiomatization of second-order cone programs, a subset of convex optimization problem, using the specification language ACSL. Additionally, annotations for numerical algorithms solving these problems were proposed. We focused our attention on the Ellipsoid Method. A numerical analysis of the method has been presented in Chapter 4. We first show how to modify the original Ellipsoid Algorithm to properly bound the programs variables. Then the propagation of numerical errors due to floating-point calculations through the operations performed by the program is presented. Finally, Chapter 5 gives mathematical evidence of the online use of these optimization algorithms. Closed-loop system is handled and receding-horizon controllers can be autocoded with soundness guarantees.

As it was recalled earlier, one of the purposes of this research was to improve V&V techniques while developing critical software. Nevertheless, this work should not be seen as a replacement for unit or integration testing. Instead, we intend to see it as a process that takes places throughout the development of the embedded software and a way to force control engineers and programmers to keep in mind that algorithms should follow mathematical evidence and need to be implemented with caution and rigor.

Distributed systems and architectures are growing in interest within Aerospace Industries. It is especially the case for engine manufacturers. Formal verification or certification of an engine controller running on a multi-core architecture is very challenging. The work developed in this thesis naturally fits the use of multi-core architecture in the sense that it is extremely promising in terms of computational resources and power, which are crucial features when using real-time optimization based control. Work towards the parallelization of optimization algorithms and its formal verification would therefore be interesting and might need to be addressed.

Appendix A

ACSL LINEAR ALGEBRA THEORY

ACSL

```
#ifndef LIB_LINALG
#define LIB_LINALG


/*@ axiomatic LinAlg {

  //NEW TYPES DEFINITION
  type vector;
  type matrix;


  logic vector vec_of_0_scalar(double * x);
  logic vector vec_of_3_scalar(double * x)
        reads x[0..2];
  logic vector vec_of_9_scalar(double * x)
        reads x[0..2];
  logic vector vec_of_2_scalar(double * x)
        reads x[0..1];
  logic matrix mat_of_3x3_scalar(double * x)
        reads x[0..8];
  logic matrix mat_of_2x3_scalar(double * x)
        reads x[0..5];
  logic real mat_select(matrix A, integer i, integer j);
  logic integer mat_row(matrix A);
  logic integer mat_col(matrix A);
  logic matrix mat_add(matrix A, matrix B);
  logic matrix mat_mult_scalar(matrix A, real z);
  logic matrix mat_mult(matrix A, matrix B);
  logic vector getRow(matrix A, integer i);
  logic vector getCol(matrix A, integer i);
  logic matrix getRows(matrix A, integer i, integer ni);
  logic matrix inverse(matrix A);
  logic integer invertible(matrix A);
  logic matrix ident(integer n);
  logic vector delta(integer n, integer i);
  logic real det(matrix A);
  logic vector vector_add(vector A, vector B);
  logic vector vector_minus(vector A, vector B);
  logic real vector_select(vector x, integer i);
  logic integer vector_length(vector x);
  logic real scalarProduct(vector x, vector y, integer n);
  logic real twoNorm(vector x);
  logic real normFrobenius(matrix A);
  logic vector vec(matrix A);
  logic vector subArray(vector x, integer i, integer ni);
  logic vector vec_mult_scalar(vector x, real z);
  logic integer sum(int * m, integer i);
  logic vector mat_mult_vector(matrix A, vector x);
  logic vector vector_affine(vector x, real alpha, vector y);
  logic matrix transpose(matrix A);
  logic real absolutevalue(real a);
  logic matrix ketbra(vector x);
  logic real pow(real a, integer i);
*/
```

Figure 53: Linear Algebra ACSL Theory

ACSL

```
/*@
  axiom pow_zero:
    \forall real a;
    pow(a, 0) == 1.0;
  axiom pow_n:
    \forall real a, integer n;
    n > 0 ==> pow(a,n) == a * pow(a,n-1);
  axiom pow_mult:
    \forall real a,b, integer n;
    pow(a, n) * pow(b, n) == pow(a*b, n);
  axiom pow_r_eps_N:
    \forall real epsilon, V, r;
      r == 0.2 ==>
      epsilon == 0.1 ==>
      V == 110 ==>
      pow(epsilon/V * r, 3) >= 6.0105004e-12;
  axiom non_singular_mat_mult_scalar:
    \forall matrix A, real a;
    invertible(A) == 1 ==>
    a != 0 ==>
    invertible(mat_mult_scalar(A,a)) == 1;
  axiom non_singulatI_AB:
    \forall vector x, real a;
    twoNorm(x) == 1 ==>
    a != -1 ==>
    invertible(mat_add(ident(3),mat_mult_scalar(ketbra(x),a))) == 1;
  axiom det_non_null:
    \forall matrix A;
      invertible(A) ==  1 ==> det(A) != 0;
  axiom det_matrix_mult:
    \forall matrix A, B;
      det(mat_mult(A,B)) == det(A)*det(B);
  axiom det_matrix_mult_scalar:
    \forall matrix A, real a;
    mat_col(A) == 3 ==>
    mat_row(A) == 3 ==>
    det(mat_mult_scalar(A,a)) == pow(a,3)*det(A);
  axiom det_I_AB:
    \forall vector x, real a;
    twoNorm(x) == 1.0 ==>
      det(mat_add(ident(3), mat_mult_scalar(ketbra(x),a))) == 1.0 + a;

  axiom normalize_vector:
    \forall vector x;
      twoNorm(vec_mult_scalar(x,1/twoNorm(x))) == 1;
  axiom ketbra_axiom_select:
    \forall vector p, integer i,j;
    0 <= i < 3 ==>
    0 <= j < 3 ==>
      mat_select(ketbra(p),i,j) == vector_select(p,i) * vector_select(p,j);
  axiom ketbra_axiom_size:
    \forall vector p;
      mat_row(ketbra(p)) == vector_length(p) &&
      mat_col(ketbra(p)) == vector_length(p);
  axiom absolutevalue_positivity:
    \forall real a; absolutevalue(a) >= 0;
  axiom absolutevalue_positive:
    \forall real a; a >= 0 ==> absolutevalue(a) == a;
  axiom absolutevalue_negative:
    \forall real a; a <= 0 ==> absolutevalue(a) == -a;
  axiom transpose_select:
    \forall matrix A, integer i, j;
      0 <= i < mat_col(A) ==>
      0 <= j < mat_row(A) ==>
      mat_select(transpose(A), i, j) == mat_select(A, j, i);
  axiom transpose_row:
    \forall matrix A;
      mat_row(transpose(A)) == mat_col(A);
  axiom transpose_col:
    \forall matrix A;
      mat_col(transpose(A)) == mat_row(A);
  axiom vec_mult_scalar_length:
    \forall vector A, real x;
      vector_length(vec_mult_scalar(A,x)) == vector_length(A);
```

Figure 54: Linear Algebra ACSL Theory (Part 2)

ACSL

```
/*@
  axiom vec_mult_scalar_select:
    \forall vector A, real x, integer i;
      0 <= i < vector_length(A) ==>
      vector_select(vec_mult_scalar(A,x), i) == vector_select(A, i) * x;
  axiom invertible_same_matrix:
    \forall matrix A, B;
      A == B ==> invertible(A) <==> invertible(B);
  axiom invertible_matrix:
    \forall matrix A;
      mat_col(A) == mat_row(A) ==>
      invertible(A) == 1 <==> det(A) != 0;
  axiom invertible_identity:
    \forall real a;
      a > 0 ==> invertible(mat_mult_scalar(ident(3),a)) == 1;
  axiom inverse_col:
    \forall matrix A;
      mat_col(A) == mat_row(A) ==>
      mat_col(inverse(A)) == mat_col(A);
  axiom inverse_row:
    \forall matrix A;
      mat_col(A) == mat_row(A) ==>
      mat_row(inverse(A)) == mat_row(A);
  axiom inverse_select:
    \forall matrix A;
      mat_col(A) == mat_row(A) ==>
      mat_mult(A,inverse(A)) == ident(mat_col(A));
  axiom ident_col:
    mat_col(ident(3)) == 3;
  axiom ident_row:
    mat_row(ident(3)) == 3;
  axiom ident_select_diff:
    \forall integer i, j;
      0 <= i < 3 ==>
      0 <= j < 3 ==>
      i != j ==> mat_select(ident(3), i, j) == 0;
  axiom ident_select_diag:
    \forall integer i, j;
      0 <= i < 3 ==>
      0 <= j < 3 ==>
      i == j ==> mat_select(ident(3), i, j) == 1;
  axiom getRows_select:
    \forall matrix A, integer i, ni, k, l;
      0 <= i < mat_row(A) ==>
      0 <= i+ni-1 <  mat_row(A) ==>
      k < ni ==>
      l < mat_col(A) ==>
      mat_select(getRows(A, i, ni), k, l) == mat_select(A, k+i, l);
  axiom getRows_row:
    \forall matrix A, integer i, ni;
      mat_row(getRows(A, i, ni)) == ni;
  axiom getRows_col:
    \forall matrix A, integer i, ni;
      mat_col(getRows(A, i, ni)) == mat_col(A);
  axiom subArray_select:
    \forall vector x, integer i, ni, k;
      vector_select(subArray(x,i,ni), k) == vector_select(x, i+k);
  axiom subArray_length:
    \forall vector x, integer i, ni;
      vector_length(subArray(x,i,ni)) == ni;
  axiom vector_minus_length:
    \forall vector x, y;
      vector_length(x) == vector_length(y) ==>
      vector_length(vector_minus(x, y)) == vector_length(x);
  axiom vector_minus_select:
    \forall vector x, y, integer i;
      vector_length(x) == vector_length(y) ==>
      0 <= i < vector_length(x) ==>
      vector_select(vector_minus(x, y), i) == vector_select(x, i) - vector_select(y, i);
*/
```

Figure 55: Linear Algebra ACSL Theory (Part 3)

ACSL

```
/*@
  axiom vector_add_length:
    \forall vector x, y;
      vector_length(x) == vector_length(y) ==>
      vector_length(vector_add(x, y)) == vector_length(x);
  axiom vector_add_select:
    \forall vector x, y, integer i;
    vector_length(x) == vector_length(y) ==>
    0 <= i < vector_length(x) ==>
    vector_select(vector_add(x, y), i) ==
    vector_select(x, i) + vector_select(y, i);
  axiom vector_affine_length:
    \forall real alpha, vector x,y;
    vector_length(x) == vector_length(x) ==>
    vector_length(vector_affine(x, alpha, y)) == vector_length(x);
  axiom vector_affine_select:
    \forall real alpha, vector x,y, integer i;
      vector_length(x) == vector_length(x) ==>
      vector_select(vector_affine(x, alpha, y), i) ==
        vector_select(x, i) + alpha*vector_select(y, i);
  axiom mat_mult_vector_length:
    \forall matrix A, vector x;
      mat_col(A) == vector_length(x) ==>
      vector_length(mat_mult_vector(A, x)) == mat_row(A);
  axiom mat_mult_vector_select:
    \forall matrix A, vector x, integer i;
      mat_col(A) == vector_length(x) ==>
      0 <= i < mat_row(A) ==>
      vector_select(mat_mult_vector(A, x), i) == scalarProduct(getRow(A,i), x, mat_col(A));
  axiom sum_init:
    \forall int *x;
      sum(x, 0) == 0;
  axiom sum_next:
    \forall int *x, integer i;
      sum(x, i+1) == *(x+i) + sum(x, i);
  axiom vec_of_3_scalar_select:
    \forall double *x, integer i;
      0 <= i < 3 ==>
      vector_select(vec_of_3_scalar(x), i) == x[i];
  axiom vec_of_3_scalar_length:
    \forall double *x;
      vector_length(vec_of_3_scalar(x)) == 3;
  axiom vec_of_9_scalar_select:
    \forall double *x, integer i;
      0 <= i < 9 ==>
      vector_select(vec_of_9_scalar(x), i) == x[i];
  axiom vec_of_9_scalar_length:
    \forall double *x;
      vector_length(vec_of_9_scalar(x)) == 9;
  axiom mat_of_3x3_scalar_select:
    \forall double *x, integer i, j;
      0 <= i < 3 ==>
      0 <= j < 3 ==>
      mat_select(mat_of_3x3_scalar(x), i, j) == *(x + i*3+j);
  axiom mat_of_3x3_scalar_row:
    \forall double *x;
      mat_row(mat_of_3x3_scalar(x)) == 3;
  axiom mat_of_3x3_scalar_col:
    \forall double *x;
      mat_col(mat_of_3x3_scalar(x)) == 3;
  axiom vec_of_2_scalar_select:
    \forall double *x, integer i;
      0 <= i < 2 ==>
      vector_select(vec_of_2_scalar(x), i) == x[i];
  axiom vec_of_2_scalar_length:
    \forall double *x;
      vector_length(vec_of_2_scalar(x)) == 2;
  axiom mat_of_2x3_scalar_select:
    \forall double *x, integer i, j;
      0 <= i < 2 ==>
      0 <= j < 3 ==>
    mat_select(mat_of_2x3_scalar(x), i, j) == *(x + i*3+j);
  axiom mat_of_2x3_scalar_row:
    \forall double *x;
      mat_row(mat_of_2x3_scalar(x)) == 2;
*/
```

Figure 56: Linear Algebra ACSL Theory (Part 4)

ACSL

```
/*@
  axiom mat_of_2x3_scalar_col:
    \forall double *x;
      mat_col(mat_of_2x3_scalar(x)) == 3;
  axiom mat_add_select:
    \forall matrix A, B;
      mat_row(A) == mat_row(B) ==>
      mat_col(A) == mat_col(B) ==>
      \forall integer i, j;
        0 <= i < mat_row(A) ==>
        0 <= j < mat_col(A) ==>
        mat_select(mat_add(A,B),i,j) == mat_select(A, i, j) + mat_select(B, i, j);
  axiom mat_add_row:
    \forall matrix A, B;
      mat_row(A) == mat_row(B) ==>
      mat_col(A) == mat_col(B) ==>
      mat_row ( mat_add(A, B)) == mat_row (A);
  axiom mat_add_col:
    \forall matrix A, B;
      mat_row(A) == mat_row(B) ==>
      mat_col(A) == mat_col(B) ==>
      mat_col ( mat_add(A, B)) == mat_col (A);
  axiom mat_mult_scalar_select:
    \forall matrix A, real z;
      \forall integer i, j;
        0 <= i < mat_row(A) ==>
        0 <= j < mat_col(A) ==>
          mat_select(mat_mult_scalar(A, z),i,j) ==mat_select(A, i, j)*z;
  axiom mat_mult_scalar_row:
    \forall matrix A, real z;
      mat_row(mat_mult_scalar(A, z)) == mat_row(A);

  axiom mat_mult_scalar_col:
    \forall matrix A, real z;
      mat_col(mat_mult_scalar(A, z)) == mat_col(A);
  predicate
    nonnull(vector x) =
      \exists integer i;
        0 <= i < vector_length(x) &&
        vector_select(x, i) != 0;
  predicate
    isZeroVector(vector A) =
      \forall integer i;
        0 <= i < vector_length(A) ==>
        vector_select(A, i) == 0.0;
  predicate
    isZeroMatrix(matrix A) =
      \forall integer i,j;
        0 <= i < mat_row(A) ==>
        0 <= j < mat_col(A) ==>
        mat_select(A, i, j) == 0.0;
  predicate
    isIdentMatrix(matrix A) =
      mat_row(A) == mat_col(A) &&
      \forall integer i,j;
        0 <= i < mat_row(A) ==>
        0 <= j < mat_col(A) ==>
        ((i == j ==> mat_select(A, i, j) == 1.0) &&
        (i != j ==> mat_select(A, i, j) == 0.0));
  predicate
    isNegatif(vector x) =
      \forall integer m;
        0 <= m < vector_length(x) ==>
        vector_select(x, m) <= 0;
  predicate
    isNotNegatif(vector x) =
      \exists integer m;
        0 <= m < vector_length(x) &&
        vector_select(x, m) > 0;
  axiom twoNorm_main:
    \forall vector x;
      twoNorm(x) == \sqrt(scalarProduct(x,x,vector_length(x)));
  axiom twoNorm_mult_scalar:
    \forall vector x, real a;
      twoNorm(vec_mult_scalar(x,a)) == absolutevalue(a) * twoNorm(x);
*/
```

Figure 57: Linear Algebra ACSL Theory (Part 5)

ACSL

```
/*@
  axiom getRow_length:
    \forall matrix A, integer i;
      0 <= i < mat_row(A) ==>
      vector_length(getRow(A, i)) == mat_col(A);
  axiom getRow_select:
    \forall matrix A, integer i,j;
      0 <= i < mat_row(A) ==>
      0 <= j < mat_col(A) ==>
      vector_select(getRow(A, i),j) == mat_select(A, i, j);
  axiom getCol_length:
    \forall matrix A, integer i;
      0 <= i < mat_col(A) ==>
      vector_length(getCol(A, i)) == mat_row(A);
  axiom getCol_select:
    \forall matrix A, integer i,j;
      0 <= i < mat_col(A) ==>
      0 <= j < mat_row(A) ==>
      vector_select(getCol(A, i),j) == mat_select(A, j, i);
  axiom scalarProduct_init:
    \forall vector x, y, integer n;
      n <= 0 ==>
      scalarProduct(x, y, n) == 0;
  axiom scalarProduct_induction:
    \forall vector x, y, integer n;
      0 <= n < vector_length(x) ==>
      0 <= n < vector_length(y) ==>
      scalarProduct(x, y, n+1) == scalarProduct(x, y, n) +
        vector_select(x,n)*vector_select(y,n);
  axiom Forbenius_Norm:
    \forall matrix A;
      normFrobenius(A) == twoNorm(vec(A));
  axiom Vectorization_mat_3_3:
    \forall double * x;
      vec(mat_of_3x3_scalar(x)) == vec_of_9_scalar(x);
  axiom mat_mult_row:
    \forall matrix A,B;
      mat_col(A) == mat_row(B) ==>
      mat_row(mat_mult(A,B)) == mat_row(A);
  axiom mat_mult_col:
    \forall matrix A,B;
      mat_col(A) == mat_row(B) ==>
      mat_col(mat_mult(A,B)) == mat_col(B);
  axiom mat_mult_select:
    \forall matrix A,B, integer i,j;
      mat_col(A) == mat_row(B) ==>
      0 <= i < mat_row(A) ==>
      0 <= j < mat_col(B) ==>
      mat_select(mat_mult(A,B), i, j) ==
        scalarProduct(getRow(A,i), getCol(B,j), mat_col(A));
  axiom equalityVec:
    \forall vector x, y;
      (vector_length(x) == vector_length(y) &&
      \forall integer i;
        0 <= i < vector_length(x) ==>
        vector_select(x, i) == vector_select(y, i)) ==> x == y;
  axiom equalityMat:
    \forall matrix A, B;
      (mat_row(A) == mat_row(B) &&
      mat_col(A) == mat_col(B) &&
      (\forall integer i, j;
        0 <= i < mat_row(A) ==>
        0 <= j < mat_col(A) ==>
        mat_select(A, i, j) == mat_select(B, i, j))) ==> A == B;
  axiom axiom_delta_length:
    \forall integer n,i;
      n > 0 ==>
      0 <= i < n ==>
      vector_length(delta(n,i)) == n;
  axiom axiom_delta_select_diff:
    \forall integer n, i, j;
      n > 0 ==>
      0 <= i < n ==>
      0 <= j < n ==>
      i != j ==>
      vector_select(delta(n,i), j) == 0;
*/
```

Figure 58: Linear Algebra ACSL Theory (Part 6)

ACSL

```
/*@
  axiom axiom_delta_select_egal:
    \forall integer n, i, j;
      n > 0 ==>
      0 <= i < n ==>
      i == j ==>
      vector_select(delta(n,i), j) == 1;
  predicate
    equalVec(vector x, vector y) =
      vector_length(x) == vector_length(y) &&
      \forall integer i;
        0 <= i < vector_length(x) ==>
        vector_select(x, i) == vector_select(y, i);
  predicate
    equalMat(matrix A, matrix B) =
      mat_row(A) == mat_row(B) &&
      mat_col(A) == mat_col(B) &&
      (\forall integer i, j;
        0 <= i < mat_row(A) ==>
        0 <= j < mat_col(A) ==>
        mat_select(A, i, j) == mat_select(B, i, j));
  axiom equalityVecFromPre:
    \forall vector x,y;
      equalVec(x, y) ==> x == y;
  axiom equalityMatFromPre:
    \forall matrix A, B;
      equalMat(A, B) ==> A == B;
  axiom negative_vec_mult_scalar:
    \forall vector x,y, real a, real b;
      isNegatif(x) ==>
      isNegatif(y) ==>
      a >= 0 ==>
      b >= 0 ==>
      isNegatif(vector_add(vec_mult_scalar(x,a),vec_mult_scalar(y,b)));
}
*/
#endif
```

Figure 59: Linear Algebra ACSL Theory (Part 7)

Appendix B

ACSL OPTIMIZATION AND ELLIPSOID THEORY

ACSL

```
/*@
#include "axiom_def_lin_alg.h"
#ifndef LIB_OPTIM
#define LIB_OPTIM
/*@ axiomatic Optim {
  //NEW TYPES DEFINITION
  type optim;
  type myset;
  logic myset feasible_set(optim OPT);
  logic myset epsilon_optimal_set(optim OPT, real epsilon, real V);
  logic real volume(myset A);
  logic myset tomyset(ellipsoid E);
  logic boolean in(myset A, vector x);
  logic optim socp_of_size_3_2_2(matrix A, vector b,
      matrix C, vector d, vector f, int* m)
      reads m[0..1];
  logic integer   size_n(optim OPT);
  logic integer   size_m(optim OPT);
  logic integer   size_na(optim OPT);
  logic matrix  getA(optim OPT);
  logic vector  getb(optim OPT);
  logic matrix  getC(optim OPT);
  logic vector  getd(optim OPT);
  logic vector  getf(optim OPT);
  logic int*    getm(optim OPT);
  logic matrix  getAi(optim OPT, integer i);
  logic vector  getbi(optim OPT, integer i);
  logic vector  getci(optim OPT, integer i);
  logic real    getdi(optim OPT, integer i);
  logic real    constraint(optim OPT, vector x, integer i);
  logic vector  constraints(optim OPT, vector x);
  logic real    minimum(optim OPT);
  logic vector  Optimal(optim OPT);
  logic real    cost(optim OPT, vector x);
  logic boolean   isBetter(optim OPT, vector x, vector y); //y >= x
  logic real    shrinkCoef(optim OPT);
  axiom getAi_axiom:
    \forall optim OPT, integer i;
      0 <= i <= vector_length(getd(OPT)) ==>
        getAi(OPT,i) == getRows(getA(OPT), sum(getm(OPT), i), *(getm(OPT)+i));
  axiom getbi_axiom:
    \forall optim OPT, integer i;
      getbi(OPT,i) == subArray(getb(OPT), sum(getm(OPT), i), *(getm(OPT)+i));
  axiom getci_axiom:
    \forall optim OPT, integer i;
      getci(OPT,i) == getRow(getC(OPT), i);
  axiom getdi_axiom:
    \forall optim OPT, integer i;
      getdi(OPT,i) == vector_select(getd(OPT), i);
  axiom optim_getA:
    \forall matrix A, C, vector b, d, f, int* m;
      getA(socp_of_size_3_2_2(A,b,C,d,f,m)) == A;
*/
```

Figure 60: Optimization ACSL Theory

ACSL

```
/*@
  axiom optim_getb:
    \forall matrix A, C, vector b, d, f, int* m;
      getb(socp_of_size_3_2_2(A,b,C,d,f,m)) == b;
  axiom optim_getC:
    \forall matrix A, C, vector b, d, f, int* m;
      getC(socp_of_size_3_2_2(A,b,C,d,f,m)) == C;
  axiom optim_getd:
    \forall matrix A, C, vector b, d, f, int* m;
      getd(socp_of_size_3_2_2(A,b,C,d,f,m)) == d;
  axiom optim_getf:
    \forall matrix A, C, vector b, d, f, int* m;
      getf(socp_of_size_3_2_2(A,b,C,d,f,m)) == f;
  axiom optim_getm:
    \forall matrix A, C, vector b, d, f, int* m;
      getm(socp_of_size_3_2_2(A,b,C,d,f,m)) == m;
  axiom cost:
    \forall matrix A, C, vector b, d, f, x, int* m;
      cost(socp_of_size_3_2_2(A,b,C,d,f,m), x) == scalarProduct(f, x, 3);
  axiom optim_n:
    \forall matrix A, C, vector b, d, f, int* m;
      size_n(socp_of_size_3_2_2(A,b,C,d,f,m)) == 3;
  axiom optim_m:
    \forall matrix A, C, vector b, d, f, int* m;
      size_m(socp_of_size_3_2_2(A,b,C,d,f,m)) == 2;
  axiom optim_na:
    \forall matrix A, C, vector b, d, f, int* m;
      size_na(socp_of_size_3_2_2(A,b,C,d,f,m)) == 2;
  axiom equalityOpt:
    \forall matrix A1,A2, C1,C2, vector b1,b2,d1,d2, f1,f2, int *m1,*m2;
      A1 == A2 ==>
      b1 == b2 ==>
      C1 == C2 ==>
      d1 == d2 ==>
      f1 == f2 ==>
      \forall integer l; 0 <= l < 2 ==> m1[l] == m2[l] ==>
      socp_of_size_3_2_2(A1,b1,C1,d1,f1,m1) == socp_of_size_3_2_2(A2,b2,C2,d2,f2,m2);
  predicate
    isFeasible(optim OPT, vector x) =
      isNegatif(constraints(OPT,x));
  predicate
    include(myset A, myset B) =
      \forall vector x; in(A,x) ==> in(B,x) ;
  predicate
    isEpsilonSolution(optim OPT, vector x, real epsilon) =
      isFeasible(OPT, x) && cost(OPT, x)  <= epsilon + minimum(OPT);
  predicate
    ValueV(optim OPT, real V, real epsilon) =
      \forall vector x1, x2; isFeasible(OPT, x1) ==> isFeasible(OPT, x2) ==>
        epsilon/V*(cost(OPT,x1) - cost(OPT,x2)) <= epsilon ;
  axiom Stay_in_Ellipsoid:
    \forall matrix P, P_plus, vector x, x_plus,p, grad, optim OPT, real alpha, beta, gamma;
      inEllipsoid(Ell(P,x),Optimal(OPT)) ==>
      p == vec_mult_scalar(mat_mult_vector(transpose(P), grad),
          1/twoNorm( mat_mult_vector(transpose(P),grad))) ==>
      x_plus == vector_add(x, vec_mult_scalar(mat_mult_vector(P, p),beta)) ==>
      P_plus == mat_mult(P,mat_mult_scalar(mat_add(ident(size_n(OPT)),
            mat_mult_scalar(ketbra(p),gamma)),alpha)) ==>
      inEllipsoid(Ell(P_plus, x_plus), Optimal(OPT));
  axiom volumeEllipsoid:
    \forall matrix P, vector x;
      volume(tomyset(Ell(P,x))) == absolutevalue(det(P));
  axiom constraints_select:
    \forall optim OPT, vector x, integer i;
      0 <= i < size_m(OPT) ==>
      vector_select(constraints(OPT, x), i) == constraint(OPT, x, i);
  axiom constraints_length:
    \forall optim OPT, vector x;
      vector_length(constraints(OPT, x)) == size_m(OPT);
  axiom constraint_linear_axiom:
    \forall optim OPT, vector x, integer i;
      getm(OPT)[i] == 0 ==>
      constraint(OPT, x, i) == -scalarProduct(getci(OPT,i), x, size_n(OPT))-getdi(OPT,i);
*/
```

Figure 61: Optimization ACSL Theory (part 2)

ACSL

```
/*@
  axiom constraint_socp_axiom:
    \forall optim OPT, vector x, integer i;
      getm(OPT)[i] != 0 ==>
        constraint(OPT, x, i) ==
          twoNorm(vector_add( mat_mult_vector(getAi(OPT,i),x),getbi(OPT,i))) -
          scalarProduct(getci(OPT,i), x, size_n(OPT)) -
          getdi(OPT, i);
  axiom isBetter_case1:
    \forall optim OPT, vector x, vector x_best;
      (!isFeasible(OPT, x) ) ==>
        isBetter(OPT, x_best, x) == \false;
  axiom isBetter_case2:
    \forall optim OPT, vector x, vector x_best;
      (isFeasible(OPT, x)  &&
      !isFeasible(OPT, x_best) ) ==>
      isBetter(OPT, x_best, x) == \true;
  axiom isBetter_case3:
    \forall optim OPT, vector x, vector x_best;
      (isFeasible(OPT, x)  &&
      isFeasible(OPT, x_best)  &&
      cost(OPT, x_best) <= cost(OPT, x)) ==>
      isBetter(OPT, x_best, x) == \false;
  axiom isBetter_case4:
    \forall optim OPT, vector x, vector x_best;
      (isFeasible(OPT, x) &&
      isFeasible(OPT, x_best)  &&
      cost(OPT, x_best) > cost(OPT, x)) ==>
      isBetter(OPT, x_best, x) == \true;
  axiom minimum_feasible:
    \forall optim OPT, vector x;
      isFeasible(OPT,x) ==> cost(OPT,x) >= minimum(OPT);
  axiom optimal_point_feasible:
    \forall optim OPT;isFeasible(OPT,Optimal(OPT));
  axiom optimal_point_cost:
    \forall optim OPT;
      cost(OPT,Optimal(OPT)) == minimum(OPT);
  axiom convex_cost:
    \forall optim OPT, vector x,y, real a;
      0 <= a <= 1 ==>
      cost(OPT,vector_add(vec_mult_scalar(x, a),vec_mult_scalar(y, 1-a))) <=
        a * cost(OPT,x) + (1-a)*cost(OPT,y);
  axiom convex_constraint:
    \forall optim OPT, vector x,y,z, real a;
      0 <= a <= 1 ==>
      x == vector_add(vec_mult_scalar(y, a),vec_mult_scalar(z, 1-a)) ==>
      isNegatif(constraints(OPT,y)) ==>
      isNegatif(constraints(OPT,z)) ==>
      isNegatif(constraints(OPT,x));
  axiom epsilon_optimal_set_axiom:
    \forall optim OPT, vector x, real epsilon, V;
      in(epsilon_optimal_set(OPT,epsilon,V),x) <==>
      (\exists vector y ;
      isFeasible(OPT,y) &&
        x == vector_add(vec_mult_scalar(y, epsilon/V),
        vec_mult_scalar(Optimal(OPT), 1-epsilon/V)));
  axiom feasible_set_axiom:
    \forall optim OPT, vector x;
      in(feasible_set(OPT),x) <==> isFeasible(OPT, x);
  axiom PositiveVolume:
    \forall myset A; volume(A) >= 0;
  axiom greatherVolume:
    \forall myset A, B;
      (\forall vector x; in(A,x) ==> in(B,x) ) ==>  volume(B) >= volume(A);
  lemma lemmaExitsElement:
    \forall myset A, B;
      (volume(A) < volume(B))==> \exists vector x; in(B,x)  && !in(A,x) ;
  lemma lemmaInclude:
    \forall myset A,B;
      include(A,B) ==> volume(A) <= volume(B);
  axiom tomySet_ell:
    \forall matrix P, vector x, y;
      in(tomyset(Ell(P,x)),y) == inEllipsoid(Ell(P,x),y);
  lemma lemmatest1:
    \forall myset A, matrix P, vector x;
      volume(tomyset(Ell(P,x))) < volume(A) ==>
      \exists vector y; in(A,y)  && !in(tomyset(Ell(P,x)),y);
*/
```

Figure 62: Optimization ACSL Theory (part 3)

ACSL

```
/*@
  lemma lemmatest2:
    \forall myset A, matrix P, vector x;
      volume(tomyset(Ell(P,x))) < volume(A) ==>
      \exists vector y; in(A,y)  && !inEllipsoid(Ell(P,x), y);
  lemma epsilon_optimal_set_cost_1:
    \forall optim OPT, vector x, real epsilon, V;
      0 < epsilon/V <= 1  ==>
      in(epsilon_optimal_set(OPT,epsilon,V),x) ==>
      (\exists vector y ; cost(OPT,x)  <=
        epsilon/V*cost(OPT,y) + (1-epsilon/V)*cost(OPT,Optimal(OPT)));
  lemma epsilon_optimal_set_cost_2:
    \forall optim OPT, vector x, real epsilon, V;
      0 < epsilon/V <= 1  ==>
      in(epsilon_optimal_set(OPT,epsilon,V),x) ==>
      (\exists vector y ; cost(OPT,x)  <=
        epsilon/V*(cost(OPT,y)-cost(OPT,Optimal(OPT)))
        + cost(OPT,Optimal(OPT)));
  lemma epsilon_optimal_set_cost_3:
    \forall optim OPT, vector x, real epsilon, V;
      0 < epsilon/V <= 1  ==>
      in(epsilon_optimal_set(OPT,epsilon,V),x) ==>
      (\exists vector y ; isFeasible(OPT, y) &&
        cost(OPT,x)  <=  epsilon/V*(cost(OPT,y)-cost(OPT,Optimal(OPT)))+ minimum(OPT));
  lemma epsilon_optimal_set_cost:
    \forall optim OPT, vector x, real epsilon, V;
      V > 0 ==>
      0 < epsilon/V <= 1  ==>
      ValueV(OPT, V, epsilon) ==>
      in(epsilon_optimal_set(OPT,epsilon,V),x) ==>
      (\exists vector y ; isFeasible(OPT, y) &&
        cost(OPT,x)  <= epsilon/V*(cost(OPT,y)-cost(OPT,Optimal(OPT)))+ minimum(OPT) &&
        cost(OPT,x)  <= epsilon + minimum(OPT));
  lemma epsilon_optimal_set_cost5:
    \forall optim OPT, vector x, real epsilon, V;
      V > 0 ==>
      0 < epsilon/V < 1  ==>
      ValueV(OPT, V, epsilon) ==>
      in(epsilon_optimal_set(OPT,epsilon,V),x) ==>
      cost(OPT,x)  <= epsilon + minimum(OPT);
  lemma epsilon_optimal_set_constraints:
    \forall optim OPT, vector x, real epsilon, V;
      0 < epsilon/V < 1 ==>
      V > 0 ==>
      in(epsilon_optimal_set(OPT,epsilon,V),x) ==>
      (\exists vector y ;
      isNegatif(constraints(OPT,y)) &&
      isNegatif(constraints(OPT,Optimal(OPT))) &&
      x == vector_add(  vec_mult_scalar(y, epsilon/V),
                vec_mult_scalar(Optimal(OPT), 1-epsilon/V)) &&
      isNegatif(constraints(OPT,x))  );
  lemma epsilon_optimal_set_constraints2:
    \forall optim OPT, vector x, real epsilon, V;
      0 < epsilon/V < 1 ==>
      V > 0 ==>
      in(epsilon_optimal_set(OPT,epsilon,V),x) ==>
      isFeasible(OPT,x);
  lemma epsilon_optimal_set:
    \forall optim OPT, vector x, real epsilon, V;
      0 < epsilon/V < 1 ==>
      0 < V ==>
      0 < epsilon ==>
      ValueV(OPT, V, epsilon) ==>
      in(epsilon_optimal_set(OPT,epsilon,V),x) ==>
      isEpsilonSolution(OPT, x, epsilon);
  lemma better_epsilon_optimal_set:
    \forall optim OPT, vector y, x, real epsilon, V;
      0 < epsilon/V < 1 ==>
      0 < V ==>
      0 < epsilon ==>
      ValueV(OPT, V, epsilon) ==>
      in(epsilon_optimal_set(OPT,epsilon,V),x) ==>
      isBetter(OPT,x,y) ==>
      isEpsilonSolution(OPT, y, epsilon);
*/
```

Figure 63: Optimization ACSL Theory (part 4)

ACSL

```
/*@
  lemma better_epsilon_optimal_set_exists:
    \forall optim OPT, vector x, real epsilon, V;
      0 < epsilon/V < 1 ==>
      0 < V ==>
      0 < epsilon ==>
      ValueV(OPT, V, epsilon) ==>
      (\exists vector y; in(epsilon_optimal_set(OPT,epsilon,V),y)
        && isBetter(OPT,y,x) ) ==>
      isEpsilonSolution(OPT, x, epsilon);
  axiom volume_ball_set:
    \forall real r, integer n;
      n > 0 ==>
      det(mat_mult_scalar(ident(n),r)) == pow(r, n);
  axiom positive_pow:
    \forall real r, integer n;
      r > 0 ==>
      pow(r,n) > 0;
  axiom axiom_algebra:
    \forall real a,b,c;
      a <= b ==>
      c >= 0 ==>
      c*a <= c*b;
  axiom axiom_algebra2:
    \forall real a, b, c,d;
      a >= 0 ==>
      b >= 0 ==>
      c >= 0 ==>
      d >= 0 ==>
      (d <= c &&  a == b*c ) ==>
      a <= b*d;
  lemma epsilon_solution_lemma_test1:
    \forall optim OPT, real V,epsilon, matrix P, vector x, x_best;
      volume(tomyset(Ell(P,x))) <
        volume(epsilon_optimal_set(OPT,epsilon,V)) ==>
      \exists vector y; in(epsilon_optimal_set(OPT,epsilon,V),y)
        && !inEllipsoid(Ell(P,x), y);
  lemma epsilon_solution_lemma:
    \forall optim OPT, real V,epsilon, matrix P, vector x, x_best;
      (0 < epsilon/V < 1) ==>
      0 < V ==>
      0 < epsilon ==>
      ValueV(OPT, V, epsilon) ==>
      (\forall vector z; !inEllipsoid(Ell(P,x), z) ==>
        isBetter(OPT, z, x_best) ) ==>
      volume(tomyset(Ell(P,x))) <
        volume(epsilon_optimal_set(OPT,epsilon,V)) ==>
      isEpsilonSolution(OPT, x_best, epsilon);
  axiom volume_epsilon_optimal_set_1:
    \forall optim OPT, real epsilon, real V;
      volume(epsilon_optimal_set(OPT, epsilon, V)) ==
        pow(epsilon/V, size_n(OPT))*volume(feasible_set(OPT));
  lemma volume_epsilon_optimal_set_2:
    \forall optim OPT, real r, epsilon, V, vector x;
      r > 0 ==>
      include(tomyset(Ell(mat_mult_scalar(ident(size_n(OPT)),r), x)) ,
            feasible_set(OPT))  ==>
      volume(tomyset(Ell(mat_mult_scalar(ident(size_n(OPT)),r), x))) <=
        volume(feasible_set(OPT));
  lemma volume_epsilon_optimal_set_3:
    \forall optim OPT, real r, epsilon, V, vector x;
      volume(tomyset(Ell(mat_mult_scalar(ident(size_n(OPT)),r), x))) ==
          absolutevalue(det( mat_mult_scalar(ident(size_n(OPT)),r) ));
  lemma volume_epsilon_optimal_set_4:
    \forall optim OPT, real r, epsilon, V;
      r > 0 ==>
      ( \exists vector x; include(tomyset(Ell(mat_mult_scalar(
            ident(size_n(OPT)),r), x)) , feasible_set(OPT)) ) ==>
      absolutevalue(det(mat_mult_scalar(ident(size_n(OPT)),r))) <=
        volume(feasible_set(OPT));
*/
```

Figure 64: Optimization ACSL Theory (part 5)

ACSL

```
/*@
  lemma volume_epsilon_optimal_set_5:
    \forall optim OPT, real r, epsilon, V;
      r > 0 ==>
      size_n(OPT) > 0 ==>
      ( \exists vector x; include(tomyset(Ell(mat_mult_scalar(
          ident(size_n(OPT)),r), x)) , feasible_set(OPT)) ) ==>
      absolutevalue( pow(r, size_n(OPT) ) ) <= volume(feasible_set(OPT));
  lemma volume_epsilon_optimal_set_6:
    \forall optim OPT, real r, epsilon, V;
      r > 0 ==>
      size_n(OPT) > 0 ==>
      ( \exists vector x; include(tomyset(Ell(mat_mult_scalar(ident(size_n(OPT)),r), x)) ,
                      feasible_set(OPT)) ) ==>
      pow(r, size_n(OPT)) <= volume(feasible_set(OPT));
  lemma volume_epsilon_optimal_set_7:
    \forall optim OPT, real r, epsilon, V;
      r > 0 ==>
      size_n(OPT) > 0 ==>
      ( \exists vector x; include(tomyset(Ell(mat_mult_scalar(ident(size_n(OPT)),r), x)) ,
              feasible_set(OPT)) ) ==>
      pow(r, size_n(OPT)) <= volume(feasible_set(OPT)) &&
      volume(epsilon_optimal_set(OPT, epsilon, V)) ==
          pow(epsilon/V, size_n(OPT))*volume(feasible_set(OPT));
  lemma volume_epsilon_optimal_set_8:
    \forall optim OPT, real r, epsilon, V;
      r > 0 ==>
      size_n(OPT) > 0 ==>
      ( \exists vector x; include(tomyset(Ell(mat_mult_scalar(ident(size_n(OPT)),r), x)) ,
              feasible_set(OPT)) ) ==>
      volume(epsilon_optimal_set(OPT, epsilon, V)) >=
            pow(epsilon/V, size_n(OPT))*pow(r, size_n(OPT));
  lemma volume_epsilon_optimal_set:
    \forall optim OPT, real r, epsilon, V;
      r > 0 ==>
      size_n(OPT) > 0 ==>
      ( \exists vector x; include(tomyset(Ell(mat_mult_scalar(ident(size_n(OPT)),r), x)) ,
              feasible_set(OPT)) ) ==>
      volume(epsilon_optimal_set(OPT, epsilon, V)) >= pow(epsilon/V*r, size_n(OPT));
  lemma epsilon_solution_lemma_BIS:
    \forall optim OPT, real r,V,epsilon, matrix P, vector x, x_best;
      (0 < epsilon/V < 1) ==>
      0 < r ==>
      0 < V ==>
      0 < epsilon ==>
      size_n(OPT) > 0 ==>
      ValueV(OPT, V, epsilon) ==>
      ( \forall vector z; !inEllipsoid(Ell(P,x), z) ==> isBetter(OPT, z, x_best) ) ==>
      ( \exists vector x; include(tomyset(Ell(mat_mult_scalar(ident(size_n(OPT)),r), x)) ,
              feasible_set(OPT)) ) ==>
      volume(tomyset(Ell(P,x))) < pow(epsilon/V*r, size_n(OPT)) ==>
      isEpsilonSolution(OPT, x_best, epsilon);
}
*/
#endif
```

Figure 65: Optimization ACSL Theory (part 6)

Appendix C

INPUT FILE USED TO GENERATE C CODE

C.1 Single Optimization Problem Example

Input File to the Autocoder

```
#GENEMO format example
Constants
# No constants defined
Variables
a b c
Minimize
5*a + 6*b + 5*c
SubjectTo
constraint1: a + b >= 11 ;
constraint2: a - b <= 5 ;
constraint3: c - a - b = 0 ;
constraint4: 7*a >= 35 - 12*b ;
constraint5: a >= 0 ;
constraint6: b >= 0 ;
constraint7: c >= 0 ;
Information
r = 0.1;
R = 40.0;
eps = 0.02;
V = 2;
#Solution: x =  [8.00000 ; 3.00000 ; 11.00000]
```

Figure 66: Single Point Optimization Text File

C.2 Spring Mass System

Input File to the Autocoder

```
Input
xinit(2)
Output
u(:,1)
Constants
N  = 10;
Ts = 0.01;
A = [1 Ts;-Ts 1];
B = [0;Ts];
uMax = 5;
positionMax = 10;
speedMax = 10;
M = N-1;
Q = [5 0;0 1];
Variables
x(2,N) u(1,M)
Minimize
sum( ||x(:,k) || , k=1..N)
SubjectTo
constraint1: x(:,1) = xinit;
constraint2: x(:,k+1) = A*x(:,k) + B*u(:,k) , k=1..N-1;
constraint3: u(:,k)  <= uMax                , k=1..N-1;
constraint4: -1*u(:,k)  <= uMax             , k=1..N-1;
constraint5: x(1,k)  <= positionMax         , k=1..N;
constraint6: -1*x(1,k)  <= positionMax      , k=1..N;
constraint7: x(2,k)  <= speedMax            , k=1..N;
constraint8: -1*x(2,k)  <= speedMax         , k=1..N;
Information
r = 1e-5;
R = 1.25*1e5;
V = 1e3;
eps = 0.1;
```

Figure 67: Spring Mass Autocoder Input File

C.3 3 DOF Helicopter

Input File to the Autocoder

```
Input
xo(6)
Output
u(:,1)
Constants
N = 6;
M = N-1;
l = 90;
r = 40;
A = [0.7101    0.0000   -0.0000    0.2331    0.0000    0.0000;
     0.0000    0.2105    0.4023    0.0000    0.0977    0.7390;
    -0.0000   -0.1272    0.9846   -0.0000   -0.0134    0.4733;
    -0.8721    0.0000   -0.0000    0.0724    0.0000    0.0000;
    -0.0000   -2.0777    0.7830    0.0000   -0.2674    1.6711;
    -0.0000   -0.4224   -0.1072   -0.0000   -0.0618    0.8109];
B = [0.2899    0.0000;    -0.0000   -0.4023; 0.0000    0.0154;
     0.8721    0.0000;     0.0000   -0.7830; 0.0000    0.1072];
Aobs = [-1   -r     0     0     0     0;
        -1    r     0     0     0     0];
bosbt = [0;0];
Variables
x(6,N) u(2,M)
Minimize
sum( || x(:,k) || , k = 1..N )
SubjectTo
constraint1: x(:,1) = xo;
constraint2: x(:,k+1) = A*x(:,k) + B*u(:,k)  ,k=1..N-1;
constraint3: -30 <= u(1,k)                   ,k=1..N-1;
constraint4: u(1,k) <= 30                    ,k=1..N-1;
constraint5: -30 <= u(2,k)                   ,k=1..N-1;
constraint6: u(2,k) <= 30                    ,k=1..N-1;
constraint8:  0 <= x(1,k)                    ,k=2..N;
constraint9: -40 <= x(2,k)                   ,k=2..N;
constraint10: x(2,k) <= 40                   ,k=2..N;
constraint11: Aobs*x(:, k) <= bosbt          ,k=2..N;
Information
r = 8.06;
R = 322;
V = 162;
eps = 0.25;
lambda = 1.000695409372118;
```

Figure 68: 3 DOF Helicopter Landing Problem: Autocoder Input File

REFERENCES

[1] "DO-178B: Software Considerations in Airborne Systems and Equipment Certification," 1982.

[2] Açikmeşe, B. and Blackmore, L., "Lossless convexification of a class of optimal control problems with non-convex control constraints," *Automatica*, vol. 47, no. 2, pp. 341–347, 2011.

[3] Açikmese, B., III, J. M. C., and Blackmore, L., "Lossless convexification of nonconvex control bound and pointing constraints of the soft landing optimal control problem," *IEEE Trans. Contr. Sys. Techn.*, vol. 21, no. 6, pp. 2104–2113, 2013.

[4] Baudin, P., Filliâtre, J.-C., Marché, C., Monate, B., Moy, Y., and Prevosto, V., "ACSL: ANSI/ISO C Specification Language. version 1.11.." `http://frama-c.com/download/acsl.pdf`, 2016.

[5] Blackmore, L., "Autonomous precision landing of space rockets," *The Bridge*, vol. 4, no. 46, pp. 15–20, 2016.

[6] Blackmore, L., Açikmese, B., and III, J. M. C., "Lossless convexification of control constraints for a class of nonlinear optimal control problems," *Systems & Control Letters*, vol. 61, no. 8, pp. 863–870, 2012.

[7] Bland, R. G., Goldfarb, D., and Todd, M. J., "The ellipsoid method: A survey," *Operations research*, vol. 29, no. 6, pp. 1039–1091, 1981.

[8] Bonnans, F., Martinon, P., and Trélat, E., "Singular arcs in the generalized goddard's problem," *Journal of optimization theory and applications*, vol. 139, no. 2, pp. 439–461, 2008.

[9] Boyd, S., El Ghaoui, L., Feron, E., and Balakrishnan, V., *Linear matrix inequalities in system and control theory.* SIAM, 1994.

[10] Boyd, S. and Vandenberghe, L., *Convex optimization.* New York, NY, USA: Cambridge University Press, 2004.

[11] Boyd, S. P. and Barratt, C. H., *Linear controller design: limits of performance.* Prentice Hall Englewood Cliffs, NJ, 1991.

[12] Brogan, W. L., *Modern control theory.* Pearson education india, 1982.

[13] Champion, A., Delmas, R., Dierkes, M., Garoche, P.-L., Jobredeaux, R., and Roux, P., "Formal methods for the analysis of critical control systems models: Combining non-linear and linear analyses," in *International Workshop on Formal Methods for Industrial Critical Systems*, pp. 1–16, Springer, 2013.

[14] Cousot, P. and Cousot, R., "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pp. 238–252, ACM, 1977.

[15] Cousot, P. and Halbwachs, N., "Automatic discovery of linear restraints among variables of a program," in *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pp. 84–96, ACM, 1978.

[16] Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., and Yakobowski, B., "Frama-c: a software analysis perspective," SEFM'12, pp. 233–247, Springer, 2012.

[17] El Ghaoui, L., "Inversion error, condition number, and approximate inverse of structured matrices," *Linear Algebra and its Applications*, vol. 342, Feb. 2002.

[18] Falcone, P., Borrelli, F., Asgari, J., Tseng, H. E., and Hrovat, D., "Predictive active steering control for autonomous vehicle systems," *IEEE Transactions on control systems technology*, vol. 15, no. 3, pp. 566–580, 2007.

[19] Feron, E., "From control systems to control software," *Control Systems, IEEE*, vol. 30, pp. 50 –71, December 2010.

[20] Floyd, R. W., "Assigning meanings to programs," *Proceedings of Symposium on Applied Mathematics*, vol. 19, pp. 19–32, 1967.

[21] Gigante, G. and Pascarella, D., "Formal methods in avionic software certification: The do-178c perspective," in *Leveraging Applications of Formal Methods, Verification and Validation. Applications and Case Studies* (Margaria, T. and Steffen, B., eds.), (Berlin, Heidelberg), pp. 205–215, Springer Berlin Heidelberg, 2012.

[22] Goubault, E., *Static Analyses of the Precision of Floating-Point Operations*, pp. 234–259. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001.

[23] Goubault, E. and Putot, S., *Static Analysis of Finite Precision Computations*, pp. 232–247. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011.

[24] Herencia-Zapana, H., Jobredeaux, R., Owre, S., Garoche, P.-L., Feron, E., Perez, G., and Ascariz, P., "Pvs linear algebra libraries for verification of control software algorithms in c/acsl," in *NASA Formal Methods Symposium*, pp. 147–161, Springer, 2012.

[25] Hoare, C. A. R., "An axiomatic basis for computer programming," *Commun. ACM*, vol. 12, pp. 576–580, October 1969.

[26] Jadbabaie, A., Yu, J., and Hauser, J., "Stabilizing receding horizon control of nonlinear systems: a control lyapunov function approach," in *American Control Conference - 1999 - Proceedings of the 1999*, pp. 1535–1539, 1999.

[27] Jerez, J. L., Goulart, P. J., Richter, S., Constantinides, G. A., Kerrigan, E. C., and Morari, M., "Embedded online optimization for model predictive control at megahertz rates," *IEEE Transactions on Automatic Control*, vol. 59, no. 12, pp. 3238–3251, 2014.

[28] Jerez, J. L., Goulart, P. J., Richter, S., Constantinides, G. A., Kerrigan, E. C., and Morari, M., "Embedded online optimization for model predictive control at megahertz rates," *IEEE Trans. Automat. Contr.*, vol. 59, no. 12, pp. 3238–3251, 2014.

[29] Kästner, D., Wilhelm, S., Nenova, S., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Rival, X., and others, "Astrée: Proving the absence of runtime errors," *Proc. of Embedded Real Time Software and Systems (ERTS2 2010)*, p. 9, 2010.

[30] Khachiyan, L. G., "Polynomial algorithms in linear programming," *USSR Computational Mathematics and Mathematical Physics*, vol. 20, no. 1, pp. 53–72, 1980.

[31] Leroy, X., "Formal verification of a realistic compiler," *Communications of the ACM*, vol. 52, no. 7, pp. 107–115, 2009.

[32] Mattingley, J. and Boyd, S., "Cvxgen: A code generator for embedded convex optimization," *Optimization and Engineering*, vol. 13, no. 1, pp. 1–27, 2012.

[33] Mattingley, J., Wang, Y., and Boyd, S., "Code generation for receding horizon control," in *Computer-Aided Control System Design (CACSD), 2010 IEEE International Symposium on*, pp. 985–992, IEEE, 2010.

[34] Miné, A., "A new numerical abstract domain based on difference-bound matrices," in *Programs as Data Objects*, pp. 155–172, Springer, 2001.

[35] Miné, A., "The octagon abstract domain," *Higher-order and symbolic computation*, vol. 19, no. 1, pp. 31–100, 2006.

[36] Nemirovski, A., *Introduction to Linear Optimization.* Lecture notes, Georgia Institute of Technology, 2012.

[37] of RTCA, S. C., "DO-178C, software considerations in airborne systems and equipment certification," 2011.

[38] Ogata, K., *System dynamics*, vol. 3. Prentice Hall Upper Saddle River, NJ, 1998.

[39] Patrinos, P., Guiggiani, A., and Bemporad, A., "A dual gradient-projection algorithm for model predictive control in fixed-point arithmetic," *Automatica*, vol. 55, pp. 226–235, 2015.

[40] Putot, S., Goubault, E., and Martel, M., *Static Analysis-Based Validation of Floating-Point Computations*, pp. 306–313. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004.

[41] Raffo, G. V., Gomes, G. K., Normey-Rico, J. E., Kelber, C. R., and Becker, L. B., "A predictive controller for autonomous vehicle path tracking," *IEEE transactions on intelligent transportation systems*, vol. 10, no. 1, pp. 92–102, 2009.

[42] Roux, P., "Formal proofs of rounding error bounds," *Journal of Automated Reasoning*, vol. 57, no. 2, pp. 135–156, 2016.

[43] Roux, P., Jobredeaux, R., and Garoche, P., "Closed loop analysis of control command software," in *Proceedings of the 18th International Conference on Hybrid Systems: Computation and Control, HSCC'15, Seattle, WA, USA, April 14-16, 2015*, pp. 108–117, 2015.

[44] Roux, P., Jobredeaux, R., Garoche, P.-L., and Féron, É., "A generic ellipsoid abstract domain for linear time invariant systems," in *Proceedings of the 15th ACM international conference on Hybrid Systems: Computation and Control*, pp. 105–114, ACM, 2012.

[45] Rump, S. M., "Verification of positive definiteness," *BIT Numerical Mathematics*, vol. 46, no. 2, pp. 433–452, 2006.

[46] Rump, S. M., "Error estimation of floating-point summation and dot product," *BIT Numerical Mathematics*, vol. 52, no. 1, pp. 201–220, 2012.

[47] Tsiotras, P. and Kelley, H. J., "Goddard problem with constrained time of flight," *Journal of guidance, control, and dynamics*, vol. 15, no. 2, pp. 289–296, 1992.

[48] Wang, T., Jobredeaux, R., Pantel, M., Garoche, P.-L., Feron, E., and Henrion, D., "Credible autocoding of convex optimization algorithms," *Optimization and Engineering*, vol. 17, no. 4, pp. 781–812, 2016.

VITA

Raphael Cohen was born in Blanc-Mesnil, France. His interest of research includes control systems, optimization and formal verification. Raphael spent some time at Onera (Toulouse, France) and University of Washington (Seattle) as a visiting researcher.

This thesis being performed as a Cotutelle between Georgia Tech and ISAE, following the defense of the thesis, Raphael Cohen is receiving both the diploma of Doctor from ISAE and the diploma of Doctor of Philosophy from Georgia Tech.

Prior to this Ph.D., Raphael Cohen graduated from ISAE-ENSMA located in Poitiers, France with a Master of Aerospace Engineering.