EPiC Series in Computing

Volume 57, 2018, Pages 290–306

LPAR-22. 22nd International Conference on Logic for Programming, Artificial Intelligence and Reasoning

Experiments in Verification of Linear Model Predictive Control: Automatic Generation and Formal Verification of an Interior Point Method Algorithm*

Guillaume Davy[12], Eric Feron[3], Pierre-Loic Garoche[1], and Didier Henrion[2]

[1] Onera - The French Aerospace Lab, Toulouse, FRANCE
[2] CNRS LAAS, Toulouse, FRANCE
[3] Georgia Institute of Technology, Atlanta GA, USA

Abstract

Classical control of cyber-physical systems used to rely on basic linear controllers. These controllers provided a safe and robust behavior but lack the ability to perform more complex controls such as aggressive maneuvering or performing fuel-efficient controls. Another approach called optimal control is capable of computing such difficult trajectories but lacks the ability to adapt to dynamic changes in the environment. In both cases, the control was designed offline, relying on more or less complex algorithms to find the appropriate parameters. More recent kinds of approaches such as Linear Model-Predictive Control (MPC) rely on the online use of convex optimization to compute the best control at each sample time. In these settings optimization algorithms are specialized for the specific control problem and embed on the device.

This paper proposes to revisit the code generation of an interior point method (IPM) algorithm, an efficient family of convex optimization, focusing on the proof of its implementation at code level. Our approach relies on the code specialization phase to produce additional annotations formalizing the intended specification of the algorithm. Deductive methods are then used to prove automatically the validity of these assertions. Since the algorithm is complex, additional lemmas are also produced, allowing the complete proof to be checked by SMT solvers only.

This work is the first to address the effective formal proof of an IPM algorithm. The approach could also be generalized more systematically to code generation frameworks, producing proof certificate along the code, for numerical intensive software.

1 Model Predictive Control and Verification Challenges

When one wants to control the behavior of a physical device, one could rely on the use of a feedback controller, executed on a computer, to perform the necessary adjustements to the device to maintain its state or reach a given target. Classical means of this *control theory* amount to express the device behavior as a linear ordinary differencial equation (ODE) and

*This work was partially supported by ANR FEANICSES project.

define the feedback controller as a linear system; eg. a PID controller. The design phase searches for proper *gains*, ie. parametrization, of the controller to achieve the desired behavior.

While this approach has been used for years with great success, eg. in aircraft control, some more challenging behaviors or complex devices need more sophisticated controllers. Assuming that the device behavior is known, one can predict its future states. A first approach, *Optimal Control* with indirect-method, search for optimal solutions solving a complex mathematical problem, the Pontryagin Maximal Principle. This is typical used to compute rocket or satellite trajectories. However this approach, while theoretically optimal, requires complex computation and cannot yet be performed online, in real-time. A second approach, *Linear Model Predictive Control* or direct method, amounts to solve online a convex optimization problem describing the current state, the target and the problem constraints, ie. limits on the thrust. A pregnant example of such trajectory computation is the landing of SpaceX orbital rockets [3].

The use of a convex encoding of the problem guarantees the absence of saddle points (local minimums) and could be resolved efficiently with polynomial-time algorithms. Convex optimization covers a large set of convex-conic problems, from linear programming (LP: linear constraints, linear cost) to quadratic programming (QP: linear contraints, quadratic cost) or semi-definite programming (SDP: linear constraints over matrices, linear cost). While the famous Simplex algorithm can efficiently address LP ones, the Interior Point Method (IPM) is the state-of-the art one when it comes to more advanced cones (eg. QP, SDP).

Linear Model Predictive control can then be expressed as a bounded model-checking problem with an additional cost to find the best solution, using convex optimization:

Definition 1 (Example: LP encoding of MPC)**.** *Let $U \subseteq \mathbb{R}^u$ and $X \subseteq \mathbb{R}^s$ be constrained convex set for inputs and states of the controller. Let $(u_k \in U)_{0 \leq k < N}$ be an N-bounded sequence of control inputs for a linear system $x_{k+1} = Ax_k + Bu_k$ with $x_k \in X$ a vector of state variables, $A \in \mathbb{R}^{s \times s}$ and $B \in \mathbb{R}^{s \times u}$. Let X_0 the initial state and X_N the target one. The objective is to compute an optimal trajectory, for example minimizing the required inputs $\Sigma_{i=1}^{N} |u_i|$ to reach the target point X_N. Let us define the following LP problem:*

$$\min \Sigma_{i=1}^{N} |u_i| \ \ s.t. \ \begin{cases} X_0 = x_0, X_N = x_n, \\ x_1 = A \cdot x_0 + B \cdot u_0, \\ \vdots \\ x_N = A \cdot x_{N-1} + B \cdot u_{N-1}. \end{cases} \tag{1}$$

Let us remark that Eq. (1) relies on a linear and discrete description (through matrices A and B) of the original device behavior, typically a non linear ODE. This LP problem has $(N-2) \times s + N \times u$ variables since x_0, x_N, A and B are known. Without loss of generality, one can express Eq. (1) over fresh definitions of $A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^m$ and $c \in \mathbb{R}^n$ as the following LP:

$$\begin{array}{rl} \min & c^\intercal \cdot x \\ \text{s.t.} & A \cdot x \leq b \end{array} \tag{2}$$

with m constraints, and where $x \in \mathbb{R}^n$ denotes here a larger state. Equality constraints $x = e$ are defined as $x \leq e \wedge -x \leq -e$.

This computation is embedded on the device, and, depending on the starting point X_0, computes the sequence of N inputs to reach the final point X_N. Since most of the parameters are known *a priori* the *usual approach is to instanciate an optimization algorithm to the specific problem* rather than embed on the device a generic solver. For example SpaceX rockets rely on CVX [12] to produce custom code, compatible with embedded devices (eg. no dynamic allocation, reduced number of computations).

While classical linear control only amounts to the computation of a linear update, these MPC approaches rely on more involved online computation: the convex optimization has to return a sound solution in finite time.

The objective of this paper is to address the verification of these IPM implementations to ensure the good behavior of MPC-based controllers. A possible approach could have been to specify the algorithm in a proof assistant and extract a correct-by-construction code. Unfortunately code extraction from these proof assistants generates source code that hardly resembles classical critical embedded source code. We rather chose to synthesize the final code while producing automatically code contracts and additional annotations and lemmas to support the automatic validation of the code. Our contributions are then following:

1. We revisit the custom code generation, instanciating an IPM algorithm to the provided problem, producing both the code, its formal specification [14, 10] and proof artifacts.
2. We then rely on Frama-C [8] to prove the validity of generated Hoare triples using deductive methods [9].
3. Our proof process is automatic: instead of proving a generic version of the algorithm, which would require strong assumptions on the problem parameters, we perform an automatic instance-specific proof, achieving a complete validation without any user input: all proofs are performed, at code level, using SMT-solvers only.
4. The approach has been evaluated on a set of generated instances of various sizes, evaluating the scalability of the proof process. The generated code can then be embedded in actual devices.

The considered setting is a primal IPM solving a LP problem. In the presented work, we focus only on the algorithmic part while dealing with the actual implementation, therefore numerical issues such as floating point computation are here neglected.

This work is the first approach addressing the formal verification of an IPM convex algorithm, as used in MPC-based controllers.

This paper is structured as follow. We introduce the reader to convex optimization and IPM in Sec. 2 and we describe the code and specification generation in Sec. 3. Sec. 4 focuses on the proof process supporting the automatic proof of generated code specification. Sec. 5 provides a feedback on our approach evaluation. Related works are then presented in Sec. 6.

2 Convex Optimization with the Interior Point Method

Let us outline the key definition and principles behind the primal IPM we analyze. As mentioned above, these notions easily extend to more sophisticated cones. We chose a primal algorithm for its simplicity compared to primal/dual algorithms and we followed Nesterov's proof [17].

Definition 2 (LP solution)**.** *Let us consider the problem of Eq.* (2) *and assume that an optimal point x^* exists and is unique. We have then the following definitions:*

$$E_f = \{x \in \mathbb{R}^n \mid Ax < b\} \qquad \textit{(feasible set of } P\textit{)} \tag{3}$$

$$f(x) = \langle c, x \rangle = C^\intercal \cdot x \qquad \textit{(cost function)} \tag{4}$$

$$x^* = \arg\min_{x \in E_f} f \qquad \textit{(optimal point)} \tag{5}$$

In order to ensure the existence of an optimal value, we assume the set E_f to be bounded.

Barrier function and analytic center. One can describe the interior of the feasible set using a penalty function $F : E_f \to \mathbb{R}$. Figure 1 depicts such a logarithmic barrier function encoding a set of linear constraints, it diverges when x approaches the border of E_f. By construction, it admits a unique minimum, called the *analytic center*, in the interior of the feasible set.

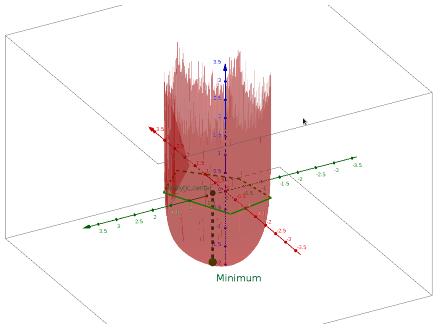


Figure 1: Barrier function.

Central path A minimum of a function is obtain by analyzing the zeros of its gradients; for convex function this minimum is unique. In case of a cost function with constraints, IPM represents these constraints within the cost function using the function $\tilde{f}(x,t)$. The variable t balances the impact of the barrier: when $t = 0$, $\tilde{f}(x,t)$ is independent from the objective while when $t \to +\infty$, the cost gains more weight.

Definition 3 (Adjusted cost $\tilde{f}$.)**.** *Let $\tilde{f}$ be a linear combination of the previous objective function f and the barrier function F.*

$$\tilde{f}(x,t) = t \times f(x) + F(x) \text{ with } t \in \mathbb{R} \tag{6}$$

Definition 4 (Central path: from analytic center to optimal solution.)**.** *The values of x minimizing $\tilde{f}$ when t varies from 0 to $+\infty$ characterize a path, the central path (cf. Fig. 2).*

$$\begin{aligned} x^* : \ \mathbb{R}^+ &\to E_f \\ t &\mapsto \arg\min_{x \in E_f} \tilde{f}(x,t) \end{aligned} \tag{7}$$

Note that $x^*(0)$ is the analytic center while $\lim_{t \to +\infty} x^*(t) = x^*$.

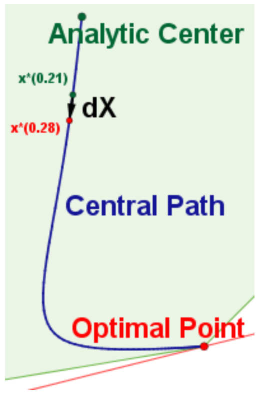


Figure 2: Central path.

IPM algorithm: a sequence of Newton steps following the central path. IPM algorithm performs a sequence of iterations, updating a point X that follows the central path and eventually reaches the optimal point. At the beginning of each loop iteration, there exists a real t such that $X = x^*(t)$. Then t is increased by $dt > 0$ and $x^*(t + dt)$ is the new point X. This translation dX is performed by a Newton step.

Computing dX using Newton step. We recall that the Newton's method computes an approximation of a root of a function $G : \mathbb{R}^k \to \mathbb{R}^l$. It is a first order method, ie. it relies on the gradient of the function and, from a point in the domain of the neighborhood of a root, performs a sequence of iterations, called Newton steps. A Newton step transforms a point Y_n into Y_{n+1} as follows:

$$Y_{n+1} - Y_n = -\big(G'(Y_n)\big)^{-1} G(Y_n) \tag{8}$$

We apply the Newton step to the gradient of $\tilde{f}$, computing its root which coincides with the minimum of $\tilde{f}$. We obtain:

$$dX = -\big(F''(X)\big)^{-1}((t+dt)c + F'(X)) \tag{9}$$

Computing dt: preserving the Approximate Centering Condition (ACC). The convergence of the Newton method is guaranteed only in the neighborhood to the function root. This neighborhood is called the region of quadratic convergence; this region evolves on each iteration since t varies. To guarantee that the iterate X remains in the region after each iteration, we require the barrier function to be self-concordant. Without providing the complete definition of self-condordance, let us focus on the implied properties: assuming that F is a self-concordant, then F'', its Hessian, is non-degenerate([17, Th4.1.3]) and we can define a local norm.

Definition 5 (Local-norm).

$$\|y\|_x^* = \sqrt{y^T \times F''(x)^{-1} \times y} \tag{10}$$

This local-norm allows one to define the Approximate Centering Condition (ACC), the crucial property which guarantees that X remains in the region of quadratic convergence:

Definition 6 (ACC). *Let $x \in E_f$ and $t \in \mathbb{R}^+$, $ACC(x,t,\beta)$ is a predicate defined by*

$$\|\tilde{f}'(x)\|_x^* = \|tc + F'(x)\|_x^* \leq \beta \tag{11}$$

In the following, as advised in [17], we choose a specific value for β, as defined in (12).

$$\beta < \frac{3-\sqrt{5}}{2} \tag{12}$$

The only step remaining is the computation of the largest dt such that X remains in the region of quadratic convergence around $x^*(t+dt)$, with γ a constant:

$$dt = \frac{\gamma}{\|c\|_x^*} \tag{13}$$

Theorem 1 (ACC preserved). *This choice maintains the ACC at each iteration([17, Th4.2.8]). When $ACC(X,t,\beta)$ and $\gamma \leq \frac{\sqrt{\beta}}{1+\sqrt{\beta}} - \beta$ then $ACC(X+dX, t+dt, \beta)$.*

Summary of IPM Thanks to an appropriate barrier function to describe the feasible set, IPM algorithm starts from the point $X = x^*(0)$, the analytic center, and $t = 0$. Then its updates both variables using Eqs. (9) and (13), until the required precision is reached.

Remark 1 (Choice of a barrier function.). *For this work, we use the classic self-concordant barrier for linear programs:*

$$F(x) = \sum_{i=0}^{m} -log(b_i - A_i \times x) \tag{14}$$

with A_1, A_m the columns of A.

Remark 2 (Computation of the analytic center.)**.** *x_F^* is required to initiate the algorithm. In case of offline use the value could be precomputed and validated. However in case of online use, its computation itself has to be proved. Fortunately this can be done by a similar algorithm with comparable proofs. In addition, in MPC-based controllers, the set of constraints can be fed with previously computed values, guaranteeing the existence of a non-empty interior and proving a feasible point to compute this new analytic center.*

3 Generating Code and Formal Specification

Typical uses of MPC do not rely on a generic solver implementing an IPM algorithm with a large set of parameters but rather instanciate the algorithm to the provided instance. As mentioned in the introduction, a large subset of the problem is known beforehand and therefore lots of computation can be hard-coded. As an example, the computation of the local norm relies mainly on the computation of the Hessian of the barrier function and can be, in some cases, precomputed.

The code generation is therefore very similar from one instance to the other. The main changes lie in the sizes of the various arrays representing the problem and the points along the central path.

Figure 3 sketches our fully automatic process which, when provided with a convex problem with some unknown values, generates the code, the associated annotation and prove it automatically. We are not going to present all the process in this paper but concentrate on how to write the embedded code, annotate it and automatize its proof. Early stages of the process reformulate the given instance in a canoncial form. Regarding the code generation, our approach is really similar to the CVXGEN or CVXOPT tools.

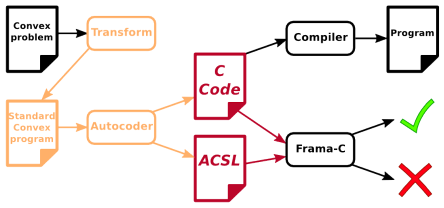


Figure 3: Autocoding toolchain: automatic generation of both C code and annotations.

3.1 Code structure: Newton steps in a For-Loop

The generated code follows strictly the algorithm presented in the previous section for each iteration steps. However, usual implementations perform the iterations in a while-loop until a condition $t_k > t_{stop}(\epsilon)$ is satisfied, where t_k represents the position on the central path at iteration k.

In order to guarantee termination, we rely on the convergence proof and complexity analysis of [17, Th 4.2.9] and compute, a priori, the required number of iterations $k_{last}(\epsilon)$ to reach a given optimality ϵ. The termination proof relies on the characterization of a a geometric progression $\textsc{Lower}(k)$ such that each increment dt is sufficient to achieve some progress towards termination: at k-th iteration, we have

$$t_k \geq \textsc{Lower}(k) \tag{15}$$

In terms of practical impact the code becomes a for loop with a provided number of iterations while the LOWER progression only appears in the formal specification part and is used for proof purposes.

```
axiomatic matrix
{
  type LMat;
        // Getters
  logic integer getM(LMat A);
  logic integer getN(LMat A);
  logic real mat_get(LMat A, integer i, integer j);
        // Constructors
  logic LMat MatVar(double* ar, integer m, integer n) reads
        ar[0..(m*n)];
  logic LMat MatCst_1_1(real x0);
  logic LMat MatCst_2_3(real x0, real x1, real x2, real x3,
        real x4, real x5);
        // Operators
  logic LMat mat_add(LMat A, LMat B);
  logic LMat mat_mult(LMat A, LMat B);
  logic LMat transpose(LMat A);
  logic LMat inv(LMat A);
  ...
        // Example of axioms
  axiom getM_add: \forall LMat A, B; getM(mat_add(A, B))
        ==getM(A);
  axiom mat_eq_def:
  \forall LMat A, B;
    (getM(A)==getM(B))==> (getN(A)==getN(B))==>
    (\forall integer i, j; 0<=i<getM(A) ==> 0<=j<getN(A) ==>
      mat_get(A,i,j)==(mat_get(B,i,j))==>
      A == B;
  ...
}
```

```
axiomatic Optim
{
  logic LMat hess(LMat A, LMat b, LMat X);
  logic LMat grad(LMat A, LMat b, LMat X);
  logic real sol(LMat A, LMat b, LMat c);
        logic real sol(LMat A, LMat b, LMat c);
          axiom sol_min: \forall LMat A, b, c;
    \forall LMat y; mat_gt(mat_mult(A, y), b) ==>
   dot(c, y) >= sol(A, b, c);
          axiom sol_greater: \forall LMat A, b, c;
    \forall Real y;
   (\forall LMat x; mat_gt(mat_mult(A, x), b) ==> dot(c, x)
          >= y) ==>
    sol(A, b, c) >= y;
  logic real norm(LMat A, LMat b, LMat x, LMat X) =
   \sqrt(mat_get(mat_mult(transpose(x), mat_mult(inv(hess(A,
          b, X)), x)), (0), (0)));
  logic boolean acc(LMat A, LMat b, LMat c, real t, LMat X,
        real beta) =
   ((norm(A, b, mat_add(grad(A, b, X), mat_scal(c, t)), X))
          <=(beta));
  ...
}
```

Figure 4: ACSL Axiomatics: Matrix and Optim

3.2 Domain-specific axiomatics

We rely on the tool Frama-C [8] to perform the proofs at code level and on ACSL [1], its annotation language, to formally express the specification. While extensible, ACSL does not provide high level constructs regarding linear algebra or optimzation related properties. We first defined new ACSL axiomatics: specific sets of types, operators and axioms to express these. Similar approaches were already proposed [13] for matrices but were too specific to ellipsoid problems. We give here an overview of the definitions of both the Matrix and Optim ACSL axiomatics, both presented in Fig. 4. Note that these definition are not instance-specific and are shared among all uses of our framework.

Axiomatic are defined as algebraic specifications. Logical opererators manipulating the local types can either be defined as functions or as uninterpreted functions fitted with axioms.

Matrix axiomatic. The new ACSL type `LMat`, ie. Logic Matrix, is introduced and fitted with getters and constructors. Constructors such as `MatVar` allow to bind fresh `LMat` value. This one reads a C array while others create constant values. Operators such as matrix addition or inverse are defined axiomatically: the operator is defined and fitted with axioms.

Optimization axiomatic. Besides generic matrix operators, we also need some operators specific to our algorithm. The Newton step defined in Eq. (9) as well as the local norm of Eq. (10) are both based on the gradient and the Hessian of the barrier function (Eq. (14)) encoding the feasible set. These functions are parametrized by the matrix A, the vector b and the local point X. However Hessian and gradient are hard to define without real analysis

```
/*@ ensures MatVar(dX, 2, 1) == \old(mat_scal(MatVar(cholesky, 2, 1), -1.0));
  @ assigns *(dX+(0..2)); */
void set_dX()
{
    dX[0] = -cholesky[0];      dX[1] = -cholesky[1];
}
```

Simple case of `dx %= -cholesky` encapsulated in a function for `dx` and `cholesky` of size 2×1. The first *ensures* statement specifies, using our own encoding of matrices in ACSL, that, after the execution of the function body, the relationship $dX = -1 * cholesky$ holds where both dX and $cholesky$ are interpreted as matrices. The second annotation *assigns* is used for the alias analysis and expresses that the only modified values are $dX[0]$, $dX[1]$. This property is also automatically generated and has to be proved by the solver. It is of great help to support the proof of relationships between arrays. Note that all variables are global here.

Figure 5: Example of ACSL-specified matrix operation.

which is well beyond the scope of this article. Therefore we decided to directly axiomatize some theorems relying on their definition like [17, Th4.1.14].

Another important notion is the optimal solution x^* of the convex problem. We can characterize axiomatically the equations (3) – (5) of Definition 2 using the uninterpreted function `sol`:

$$\forall y \in E_f, c^T y \geq \texttt{sol} \qquad \forall y \in \mathbb{R}, \forall x \in E_f, c^T x \geq y \implies \texttt{sol} \geq y \tag{16}$$

Last we defined some operators representing definitions 5, 6 and Lower(15).

3.3 Functions and contracts

Multiple functions to support proof and local contrats. Proving large functions is usually hard with SMT-based reasoning since the generated goals are too complex to be discharged automatically. A more efficient approach is to associate small pieces of code with local contracts; these intermediate annotations acting as cut-rules in the proof processes.

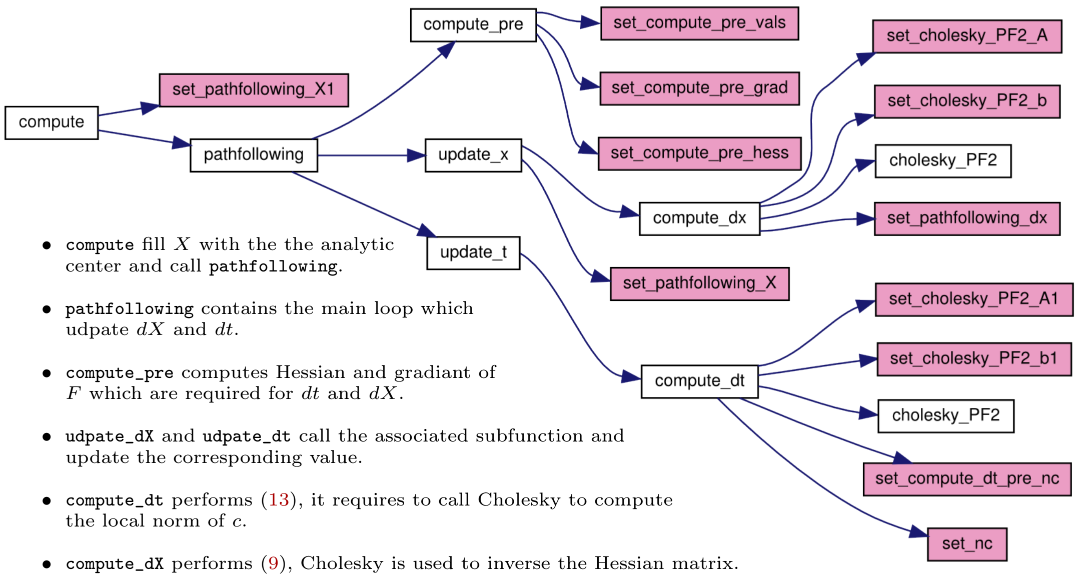
Let `A = B[C]` be a piece of code containing `C`. Replacing `C` by a call to `f() { C }` requires either to inline the call or to write a new contract $\{P\}$ `f()` $\{Q\}$, characterizing two smaller goals instead of a larger one. Specifically in the proof of a `B[f()]`, `C` has been replaced by P and Q which is simpler than an automatically computed weakest precondition.

Therefore instead of having one large function, our code is structured into several functions. As an example, each basic matrix operation is performed in a dedicated function. The associated contract provides a high level interpretation of the function behavior expressed over matrices abstract datatypes. This modular encoding supports the generation of simpler goals for the SMT solver, while keeping the code structure clearer.

Fig. 5 illustrates a very simple generated code and contract while Fig. 6 presents the hierarchy of function calls.

Specification of `pathfollowing`. A sound algorithm must produce a point in the feasible set such that its cost is ϵ-close to sol. This represents the functional specification that we expect from the code and is asserted by two post-conditions. In addition, two preconditions state that X is feasible and close enough to the analytic center.

Thanks to our two new theories `Matrix` and `Optim`, writing and reading this contract is straightforward and can be checked by anyone familiar with linear programming. Our contri-



- `compute` fill X with the the analytic center and call `pathfollowing`.
- `pathfollowing` contains the main loop which udpate dX and dt.
- `compute_pre` computes Hessian and gradiant of F which are required for dt and dX.
- `udpate_dX` and `udpate_dt` call the associated subfunction and update the corresponding value.
- `compute_dt` performs (13), it requires to call Cholesky to compute the local norm of c.
- `compute_dX` performs (9), Cholesky is used to inverse the Hessian matrix.

Figure 6: Call tree of the implementation (colored boxes are matrix computation)

```
/*@ requires mat_gt(mat_mult(A, MatVar(X, N, 1)), b);
  @ requires acc(A, b, c, 0, MatVar(X, N, 1), BETA);
  @ ensures mat_gt(mat_mult(A, MatVar(X, N, 1)), b);
  @ ensures dot(MatVar(X, 2, 1), c) - sol(A, b, c) < EPSILON
*/
void pathfollowing() {
  ...
  /*@ loop-invariant mat_gt(mat_mult(A, MatVar(X, N, 1)), b);
    @ loop-invariant acc(A, b, c, t, MatVar(X, N, 1), BETA);
    @ loop-invariant t > lower(l); */
  for (int l = 0; l < NBR;l++) { ... }
  ...
}
```

Figure 7: `Pathfollowing` ACSL contract and loop annotations.

bution focuses on the expression on intermediate annotations, spread over the code, to support the overall proof of specifications.

A loop needs to be annotated by an invariant to have its Weakest precondition computed. We need three invariants for our pathfollowing algorithm. The first one guarantees the feasibility of X while the second one states the conservation of the ACC (cf. Def. 6) The third invariant assert that t is increasing enough on each iteration, more specifically that it is greater than a geometric progression(15).

Proving the initialization is straightforward, thanks to the main preconditions. We wrote one ACSL lemma for each invariant. Whenever it is possible we try to follow [17], for example by translating [17, Th4.1.5] and Theorem 1. The last two loop invariants are combined to prove the second post-condition of `pathfollowing` thanks to Theorem 2. `NBR` is computed from k_{last}.

Theorem 2. *[17, Th4.2.7] Let $t \geq 0$, and X such that $ACC(X, t, \beta)$ then*

$$c^T X - c^T X^* < \frac{1}{t} \times (1 + \frac{(\beta+1)\beta}{1-\beta}) \tag{17}$$

Fig. 7 presents the specification in ACSL: the main function contract and the loop invariants.

Loop body. Each iteration of the IPM relies on three function calls: `update_pre` computing some common values, `update_t` and `update_x` (cf Fig. 6). Theorem 1 is then split into several properties and the corresponding post-conditions. For example, the contract of `update_t` is described in Fig. 8.

```
/*@ requires MatVar(hess, N, N)==hess(A, b, MatVar(X, N, 1));
  @ requires acc(A, b, c, t, MatVar(X, N, 1), BETA);
  @ ensures  acc(A, b, c, t, MatVar(X, N, 1), BETA + GAMMA);
  @ ensures t > \old(t)*(1 + GAMMA/(1 + BETA));*/
void update_t();
```

Figure 8: `update_t` ACSL contract

The first post-condition is an intermediary results stating that:

$$ACC(X, t + dt, \beta + \gamma) \tag{18}$$

This result is used as precondition for `update_x`. The second `ensures` corresponds to the product of t by the common ratio of the geometric progression LOWER, cf. Eq. (15) which will be used to prove the second invariant of the loop. The first precondition is a post-condition from `update_pre` and the second one is the first loop invariant.

4 Automatic Verification: Proof Refinement through Lemma Definitions and Local Contracts.

Our framework produces both the code and its annotations, as well as a set of axioms related to our newly defined axiomatics (cf. Fig. 4). However most of the contracts cannot be proved automatically by an SMT solver. After a first phase in which we tried to perform these proofs with Coq, we searched for more generic means to automatize the proof.

4.1 Refining the proofs.

We performed the following steps in order to identify the appropriate set of additional assertions, local contracts or lemma to be introduce:

- we took a fully defined custom code for a linear problem and study means to prove its validity: a feasible, optimal solution, while guarantying the convergence of the algorithm.
- these proofs were then manually generalized through the introduction of numerous intermediate lemmas as annotations in the code. These additional annotations enable the automatic proof of the algorithm using an off-the-shelf SMT solver Alt-Ergo [4].
- the custom code generation was extended to produce both the actual code and these annotations and function contracts, enabling both the generation of the embedded code and its proof of functional soundness.

Another approach is to refine the code into smaller components, as presented in Fig. 6. The encoding hides low-level operations to the rest of the code, leading to two kinds of goals:

```
assert getM(MatVar(dX,2,1)) == 2;
assert getN(MatVar(dX,2,1)) == 1;
assert getM(MatVar(cholesky,2,1)) == 2;
assert getN(MatVar(cholesky,2,1)) == 1;
assert mat_get(MatVar(dX,2,1),0,0) == mat_get(\old(mat_scal(MatVar(cholesky,2,1),-1.0)),0,0);
assert mat_get(MatVar(dX, 2, 1),1,0) == mat_get(\old(mat_scal(MatVar(cholesky,2,1),-1.0)),1,0);
```

Figure 9: Assertion appended to the function of Figure 5

- Low level operation (memory and basic matrix operation).
- High level operation (mathematics on matrices).

4.2 Simplifying the code: addressing memory-related issues.

One of the difficulties when analyzing C code are memory related issues. Two different pointers can *alias* and reference the same part of the stack. A fine and complex modeling of the memory in the predicate encoding, such as separation logic[18] could address these issues. Another more pragmatic approach amounts to substitute all local variables and function arguments with global static variables. Two static arrays cannot overlap since their memory is allocated at compile time.

Since we are targeting embedded system, static variables will also permit to compute and reduce memory footprints. However there are two major drawbacks: the code is less readable and variables are accessible from any function. These two points usually lead the programmer to mistakes but could be accepted in case of code generation. In order to support this we also tagged all variables with the function they belong to by prefixing each variable with its function name.

4.3 Low-level goals: Proving Matrix operations.

As explained in previous Section, the matrix computation are encapsulated into smaller functions. Their contract states the equality between the resulting matrix and the operation computed. The extensionality axiom (`mat_eq_def`) is required to prove this kind of contract. Extensionality means that if two objects have the same external properties then they are equal.

This axiom belong to the matrix axiomatic but is too general to be used therefore lemmas specific to the matrices size are added for each matrix affectation. This lemma can be proven with the previous axioms and therefore does not introduce more assumption.

The proof remains difficult or hardly automatic for SMT solvers therefore we append additional assertions, as sketched in Figure 9, at the end of function stating all the hypothesis of the extensionality lemma. Proving these post-conditions is straightforward and smaller goals need now to be proven.

Further split of functions into smaller functions is performed when functions become larger. As an example, for operations on matrices of size m^2, there is m^2 lines of C code manipulating arrays and logic. To avoid large goals involving array accesses for SMT solvers when m becomes greater than 10, each operation is encapsulated as a separate function annotated by the operation on a single element:

```
mat_get(MatVar(dX,2,1),0,0) == mat_get(\old(mat_scal(MatVar(cholesky,2,1),-1.0)),0,0);
```

These generated goals are small enough for SMT. The post-condition of the matrix operation remains large but the array accesses are abstracted away by all the function call and therefore becomes provable by SMT solvers.

4.4 High-level goals: IPM specific properties.

Proving sophisticated lemmas with non trivial proof is hardly feasible for an SMT solver. To support them, we introduced many intermediate lemmas. At each proof step (variable expansion, commutation, factorization, lemma instantiation, ...) a new lemma is automatically introduced. With this method, SMT solvers manage to handle each small step which eventually lead to the proof of the initial lemma.

For example, the post-condition (18) of `update_t` is not provable by Alt-Ergo. In order to prove it, we introduce the lemma (`update_t_ensures1`) where P_1 is $ACC(X, t, \beta)$, the precondition, and P_2 is $dt = \frac{\gamma}{\|c\|_x^*}$ the performed update.

$$\forall x, t, dt;\ P_1 \Rightarrow P_2 \Rightarrow ACC(X, t + dt, \beta + \gamma) \qquad (\texttt{update\_t\_ensures1})$$

Proving the post-condition with SMT solvers is straightforward given the previous lemma but again proving the lemma itself is beyond their capabilities. However with the additional lemma (`update_t_ensures1_l0`) the proof becomes obvious to the SMT solver. Indeed the lemma is just the expansion of the ACC definition.

$$\forall x, t, dt;\ P_1 \Rightarrow P_2 \Rightarrow \|F'(X) + c(t + dt)\|_x^* \leq \beta + \gamma \qquad (\texttt{update\_t\_ensures1\_l0})$$

Step by step, we introduce new lemmas to prove the previous ones: e.g. to prove the goal (`update_t_ensures1_l0`) we introduce the following three lemmas:

$$\forall x, t, dt;\ P_2 \Rightarrow \|c \times dt\|_x^* = \gamma \qquad (\texttt{update\_t\_ensures1\_l3})$$

$$\forall x, t, dt;\ P_1 \Rightarrow \|F'(X) + c \times t\|_x^* \leq \beta \qquad (\texttt{update\_t\_ensures1\_l2})$$

$$\forall x, t, dt;\ P_1 \Rightarrow P_2 \Rightarrow \|F'(X) + c(t + dt)\|_x^* \leq \|F'(X) + c \times t)\|_x^* + \|c \times dt\|_x^* \qquad (\texttt{update\_t\_ensures1\_l1})$$

Each lemma depends on other lemmas, and so do contracts, all this logic results shapes a proof tree. The one for the first ensures of `update_t` is presented in Fig. 10.

These small steps are usually bigger than the application of a tactic in a proof assistant. SMT solvers are also more resilient to a change in the source code. For these two reasons we decided not to write proofs directly within a proof assistant but to use SMT solvers. Moreover, all these intermediate lemmas are eventually automatically generated by the autocoding framework.

5 Experimentations

We implemented the presented approach: considering a provided LP problem, a source file is generated as well as a considerable amount of function contracts and additional lemmas – all expressed in ACSL [1], the annotation language of the Frama-C verification platform. The function contracts specify the functional requirements of this custom-code and instance-specific solver: it computes a feasible and optimal solution in a predictable – and known – number of iterations.

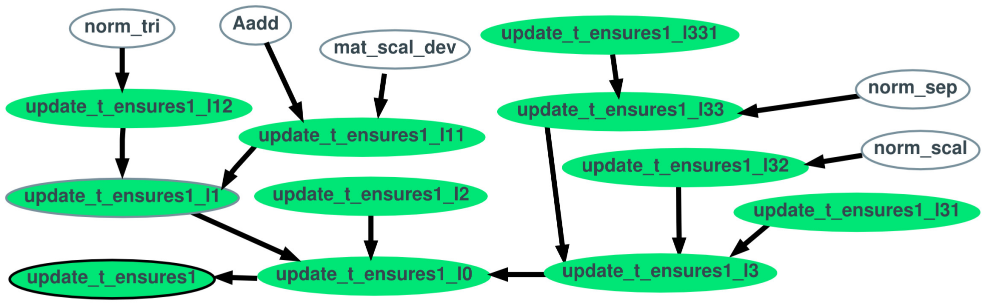


Figure 10: Proof tree for (18) (In green, proven goal, in white, axioms)

Now, this specific code has to be validated before being used on the embedded device. We rely on Frama-C to perform the formal verification process. Each generated annotation becomes a proof objective and is submitted to an SMT solver, Alt-Ergo [4], for verification. Thanks to our introduction of all these intermediate lemmas, each local lemma is proved as well as the global function contracts.

Note that the generated code is a direct implementation of the chosen primal IPM for LP. The main concern here is the proof. We present here our experimentations. First, we address issues with respect to the provers (Frama-C and Alt-Ergo) and how we tuned the code and proof generation to ease the proofs. Second, we present the results in terms of goals proved or not-yet-proved and the time required to perform the proofs.

5.1 Limitation of the approach: Asserted lemmas.

The verification is performed thanks to two sets of axioms we developed, supporting (1) the definition of matrices in ACSL or (2) properties of operators in linear programming. While the first set describing matrices commutation or norm positivity could be proved, it is an additional work outside of the scope of our contribution. The second set is more challenging to prove and corresponds to the axiomatization of some theorems from [17]. Their definition as axioms was carefully addressed and the number of such axioms minimized.

We deliberately choose to ignore:

- the proof of soundness of the Cholesky resolution;
- the validation of the gradient or Hessian computations;
- some vector and matrix related properties: eg. matrix multiplication associativity or norm positivity (scalar product);
- more involved optimization related theorems used within the proof of optimality, and developed in [16].

All these axioms are potential flaws for the verification. One solution could be to translate them within a Theorem prover such as Coq [2] and proving them instead of assuming them. Translating matrices to a theorem prover would also have the advantage to check the matrix axiomatic we wrote. This could be done by providing a bijection between the translation of our matrices and matrices defined in Coq libraries like SSReflect [11].

Last, the analytic center has to be provided, at least within the β-neighborhood, which is performed automatically through a dedicated algorithm in our experiments. While its existence cannot be guaranted in theory, its computation in MPC context can be eased (cf. Remark 2).

5.2 Results: Scalability of proof process

While the custom-code itself ought to be executed on an embedded system with limited computation power, the verification of its soundness can be performed on a regular desktop computer.

While all contracts and lemmas were proved, except the set of axioms we assumed (cf. previous Section), the time and therefore the scalability of the proof process largely depend on the problem size.

We provide here some figures regarding the total time required to perform the proofs. The verification has been performed on 2.6GHz computer running Linux with 4GB of RAM. Alt-Ergo does not parallelize computations.

Total computation time. The total proof time directly depends on the size of the input problem. This is expected since a $n \times m$ problem will manipulate arrays of that size and express properties over that many variables.

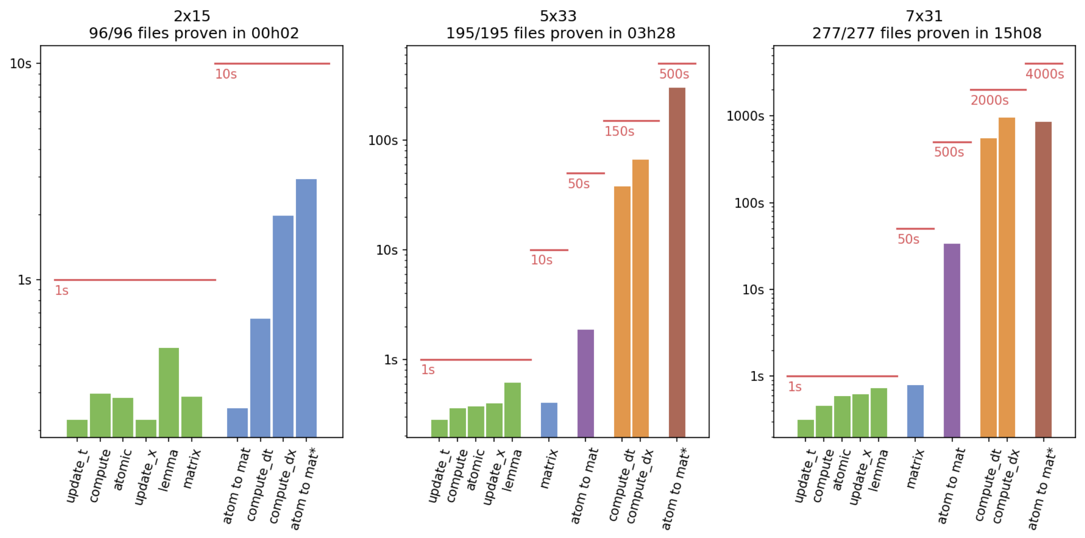
We used our framework on a large set of LP problems of various sizes. The boundedness constraint of the feasible set (cf. Sec. 2) has been artificially enforced by adding a hypercube constraint. Therefore problems over n variables have at least $2 \times n$ constraints. In practice, we generated about 100 random instances for each pair $(n, m) \in [2, 4] \times [3 \times n, 5 \times n[$ and perform the proofs with both a 1 second and 10 seconds timeouts, for each goal.

An interesting outcome is the relative independence of the proof time with respect to the actual numerical values of the constraints. Since we recorded the computation time for each instance of a (n, m) pair, we compare the experimental results. As an example, for problems of size $(3, 9)$, the total computation time ranges in $[124s, 127s]$. Note that sparse constraints will generate more optimized code, minimizing the number of computations, but are likely to generate harder problem for the solvers.

The number of generated goals is impressive: for a problem of size (n, m), we generate automatically $129 + 18 \times n + 3 \times m + 9 \times n^2$ goals. We can see that both the number of proofs is linear in m but quadratic in n. Let us now have a closer look at the proof costs depending on the category of proof.

Scalability of proofs. In order to precisely keep track of the proof time of each function, we stored them in separate files and recorded the associated proof time. Figure 11 presents such results. Files and their proof time have been packaged by clusters depending on their ability to scale, as identified by timeouts. For example one can identify the following clusters: *lemma* denoting the set of ACSL lemmas, *atomic* denoting local atomic manipulations of matrix elements, *matrix* denoting matrix level operation and *atom to mat* denoting an intermediary results between the last two. We isolate one of the file from the last cluster : *atom to mat** since it was longer to prove than the others. Other categories correspond to individual files implementing higher level functions of the IPM: *update_ dt*, *update_ dx*, *compute_ dt*, *compute_ dx* and *compute*.

As expected, some properties are difficult to prove automatically. These scalability issues mainly occur in *compute_ dt*, *compute_ dx* and the *set_* and *set_ in* clusters. However, the associated proofs are not mathematically challenging and the main limitation seems to be caused by the Frama-C encoding of goals as SMT proof objectives. The future versions of Frama-C



Each cluster is described by its average time and the timeout(red lines) setting of the SMT solver i.e. an upper bound on the maximum time for each goal. As an example, the analyzed instance of size 5×33 produces 195 goals, all proved. It requires less than one second for the goals of categories *lemma*, *atomic*, *update_ t*, *update_ x* and *compute*; less than 10 seconds for the category *matrix* and 50s for the category *atom to mat*. Both *compute_ dt* and *compute_ dx* are proved with a timeout of 150s. The isolated file *atom to mat** requires a higher timeout of 500s to achieve all proofs. Note that the average proof time of goals in cluster *compute_ dt* is about 40s so only a subset (2 out of 6) of the goals require 150s to be proved.

Figure 11: Proof-Time required per proof or file category.

may have better memory model, identifying unmodified variables in predicates, leading to better proof times of our generated instances.

An annotated code, automatically generated by our framework, is accessible at `https://github.com/davyg/proved_primal`, annotations can be found in the header `build_primalTest/code/primalTest.h` which itself includes `primalTest_contracts.h` the file containing all the function contracts.

6 Related work

Related works include first activities related to the validation of numerical intensive control algorithms. This article is an extension of Wang *et al* [21] which was presenting annotations for a convex optimization algorithm, namely IMP, but the process was both manual and theoretical: the code annotated within Matlab and without machine checked proofs. An other work from the same authors [13] presented a similar method than ours but limited to simple control algorithms, linear controllers. The required theories in ACSL were both different and less general than the ones we are proposing here.

Concerning soundness of convex optimization, Cimini and Bemporad [7] presents a termination proof for a quadratic program but without any concerns for the proof of the implementation itself. A similar criticism applies to Tøndel, Johansen and Bemporad [20] where another possible approach to online linear programming is proposed, moreover it is unclear how this could scale and how to extend it to other convex programs. Roux *et al* [19, 15] also presented a way to certify convex optimization algorithm, namely SDP and its sum-of-Squares (SOS) extension, but the certification is done a posteriori which is incompatible with online optimization.

A last set of works, e.g. the work of Boldo *et al* [5], concerns the formal proof of complex numerical algorithms, relying only on theorem provers. Although code can be extracted from the proof, the code is usually not directly suitable for embedded system: it is too slow and requires different compilation steps which should also be proven to have the same guarantee than our proposed method.

7 Conclusion and future works

This article focuses on the proof a Interior Point Method (IPM) convex optimization algorithm as used in state-of-the-art linear Model Predictive Control (MPC). The current setting is the simplest one: a primal IPM for Linear Programming problems.

In these MPC approaches convex optimzation engines are autocoded from an instance specification and embed on the cyber-physical system. Our approach relies on the autocoding fremwork to generate, along the code, the specification and the required proof artifacrs. Once both the code and these elements are generated, the proof is achieved automatically by Frama-C, using deductive methods (weakest precondition) and SMT solvers (Alt-Ergo).

This is the first proof, at code level, of an IPM algorithm. Still the proposed approach is a proof-of-concept and has some identified limitations. First, we worked with real variables to concentrate on runtime errors, termination and functional specification and left floating points errors for a future work. Then, some mathematical lemmas were asserted since our goal was to focus on the algorithm itself rather than proving that a norm ought to be positive. The set of asserted lemmas is still limited and reasonable. Last the setting is simple: a primal algorithm for LP. The proposed approach is a first step and can naturally be extended to more sophisticated Interior Point Methods (IPM): considering primal-dual algorithms, or quadratic constraints. The limitation is to restrict to IPM algorithms that guaranty the computation of a feasible solution. For example this does not include homogenized versions [6, §11, Bibliography] of IPM that do not compute iterates within the feasible set.

Another outcome is the general approach to deal with the proof of complex numerical algorithms which are autocoded from an instance description. Code generation is now widespread for embedded devices, especially in control, and could be fitted with formal specification and proof artifacts, to automatize the proof at code level. This would require further developments to axiomatize the associated mathematical theories.

References

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

[2] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development*. Springer Berlin Heidelberg, 2004.

[3] Lars Blackmore. Autonomous precision landing of space rockets. *National Academy of Engineering, Winter Bridge on Frontiers of Engineering*, 4(46), December 2016.

[4] François Bobot, Sylvain Conchon, Evelyne Contejean, and Stéphane Lescuyer. Implementing polymorphism in smt solvers. In *Proceedings of the Joint Workshops of the 6th International Workshop on Satisfiability Modulo Theories and 1st International Workshop on Bit-Precise Reasoning*, SMT '08/BPR '08, pages 1–5, New York, NY, USA, 2008. ACM.

[5] S. Boldo, F. Faissole, and A. Chapoutot. Round-off error analysis of explicit one-step numerical integration methods. In *2017 IEEE 24th Symposium on Computer Arithmetic (ARITH)*, pages 82–89, July 2017.

[6] Stephen Boyd and Lieven Vandenberghe. *Convex optimization*. Cambridge University Press, New York, NY, USA, 2004.

[7] G. Cimini and A. Bemporad. Exact complexity certification of active-set methods for quadratic programming. *IEEE Transactions on Automatic Control*, PP(99):1–1, 2017.

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

[9] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, 1975.

[10] Robert W. Floyd. Assigning meanings to programs. *Proceedings of Symposium on Applied Mathematics*, 19:19–32, 1967.

[11] Georges Gonthier, Assia Mahboubi, and Enrico Tassi. A Small Scale Reflection Extension for the Coq system. Research Report RR-6455, Inria Saclay Ile de France, 2016.

[12] Michael Grant and Stephen Boyd. CVX: Matlab software for disciplined convex programming, version 2.1. http://cvxr.com/cvx, March 2014.

[13] Heber Herencia-Zapana, Romain Jobredeaux, Sam Owre, Pierre-Loïc Garoche, Eric Feron, Gilberto Perez, and Pablo Ascariz. Pvs linear algebra libraries for verification of control software algorithms in c/acsl. In Alwyn Goodloe and Suzette Person, editors, *NASA Formal Methods - Forth International Symposium, NFM 2012, Norfolk, VA USA, April 3-5, 2012. Proceedings*, volume 7226 of *Lecture Notes in Computer Science*, pages 147–161. Springer, 2012.

[14] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[15] Érik Martin-Dorel and Pierre Roux. A reflexive tactic for polynomial positivity using numerical solvers and floating-point computations. In Yves Bertot and Viktor Vafeiadis, editors, *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs, CPP 2017, Paris, France, January 16-17, 2017*, pages 90–99. ACM, 2017.

[16] Yurii Nesterov. *Introductory lectures on convex optimization : a basic course*. Applied optimization. Kluwer Academic Publ., Boston, Dordrecht, London, 2004.

[17] Yurii Nesterov and Arkadi Nemirovski. *Interior-point Polynomial Algorithms in Convex Programming*, volume 13 of *Studies in Applied Mathematics*. Society for Industrial and Applied Mathematics, 1994.

[18] J. C. Reynolds. Separation logic: a logic for shared mutable data structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74, 2002.

[19] Pierre Roux. Formal proofs of rounding error bounds - with application to an automatic positive definiteness check. *J. Autom. Reasoning*, 57(2):135–156, 2016.

[20] Petter Tøndel, Tor Arne Johansen, and Alberto Bemporad. An algorithm for multi-parametric quadratic programming and explicit MPC solutions. *Automatica*, 39(3):489–497, 2003.

[21] Timothy Wang, Romain Jobredeaux, Marc Pantel, Pierre-Loic Garoche, Eric Feron, and Didier Henrion. Credible autocoding of convex optimization algorithms. *Optimization and Engineering*, 17(4):781–812, Dec 2016.