Credible Autocoding of Convex Optimization Algorithms

Timothy Wang[1], Romain Jobredeaux[1], Marc Pantel[4], Pierre-Loic Garoche[2], Eric Feron[1], and Didier Henrion[3]

[1] Georgia Institute of Technology, Atlanta, Georgia, USA
[2] ONERA – The French Aerospace Lab, Toulouse, FRANCE
[3] CNRS-LAAS – Laboratory for Analysis and Architecture of Systems, Toulouse, FRANCE
[4] ENSEEIHT, Toulouse, FRANCE

Abstract. 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. There is a considerable body of mathematical proofs on on-line optimization programs which can be leveraged to assist in the development and verification of their implementation. In this paper, we demonstrate how theoretical proofs of real-time optimization algorithms can be used to describe functional properties at the level of the code, thereby making it accessible for the formal methods community. The running example used in this paper is a generic semi-definite programming (SDP) solver. Semidefinite programs can encode a wide variety of optimization problems and can be solved in polynomial time at a given accuracy. We describe a top-down approach that transforms a high-level analysis of the algorithm into useful code annotations. We formulate some general remarks about how such a task can be incorporated into a convex programming autocoder. We then take a first step towards the automatic verification of the optimization program by identifying key issues to be adressed in future work.

Keywords: Control Theory, Autocoding, Lyapunov proofs, Formal Verification, Optimization, Interior-point Method, PVS, frama-C

1 Introduction

The applications of optimization algorithms are not limited to large scale, off-line problems on the desktop. They also can perform in a real-time setting as part of safety-critical systems in control, guidance and navigation. For example, modern aircrafts often have redundant control surface actuation, which allows for reconfiguration and recovery in case of emergency. The precise re-allocation of the actuation resources can be posed, in the simplest case, as a linear optimization problem that needs to be solved in real-time.

In contrast to off-line desktop optimization applications, real-time embedded optimization code needs to satisfy a higher standard of quality, if it is to be used

within a safety-critical system. Some important criteria in judging the quality of an embedded code include the predictability of its behaviors and whether or not its worst case computational time can be bounded. Several authors including Richter [21], Feron and McGovern [14][13] have worked on the certification problem for on-line optimization algorithms used in control, in particular on worst-case execution time issues. In those cases, the authors have chosen to tackle the problem at a high-level of abstraction. For example, McGovern re-examined the proofs of computational bounds on interior point methods for semi-definite programming; however he stopped short of using the proofs to analyze the implementations of interior point methods. In this paper, we extend McGovern's work further by demonstrating the expression of the proofs at the level of the code for the certification of on-line optimization code. The utility of such demonstration is twofold. First, we are considering the reality that the verification of safety-critical systems is almost always done at the source code level. Second, this effort provides an example output that is much closer to being in an accessible form for the formal methods community.

The most recent regulatory documents such as DO-178C [24] and, in particular, its addendum DO-333 [25], advocate for the use of formal methods in the verification and validation of safety critical software. However, complex computational cores in domain specific software such as control or optimization software make their automatic analysis difficult in the absence of input from domain experts. It is the authors' belief that communication between the communities of formal software analysis and domain-specific communities, such as the optimization community, are key to successfully express the semantics of these complex algorithms in a language compatible with the application of formal methods.

The main contribution of this paper is to present the expression, formalization, and translation of high-level functional properties of a convex optimization algorithm along with their proofs down to the code level for the purpose of formal program verification. Due to the complexity of the proofs, we cannot yet reason about them soundly on the implementation itself. Instead we choose an intermediate level of abstraction of the implementation where floating-point operations are replaced by real number algebra.

The algorithm chosen for this paper is based on a class of optimization methods known collectively as interior point methods. The theoretical foundation behind modern interior point methods can be found in the works of Nemrovskii et. al [17][18]. One key result is the self-concordance of certain barrier functions that guarantees the convergence of a Newton iteration to an ϵ-optimal solution in polynomial time. For more details on polynomial-time interior point methods, readers can refer to [19].

Interior-point algorithms vary in the Newton search direction used, the step length, the initialization process, and whether or not the algorithm can return infeasible answers in the intermediate iterations. Some example search directions are the Alizadeh-Haeberly-Overton (AHO) direction [1], the Monteiro-Zhang (MZ) directions [16], the Nesterov-Todd (NT) direction [20], and the Helmberg-Kojima-Monteiro (HKM) direction [8]. It was later determined in [15] that all

of these search directions can be captured by a particular scaling matrix in the linear transformation introduced by Sturm and Zhang in [28]. An accessible introduction to semi-definite programming using interior-point method can be found in the works of Boyd and Vandenberghe [3].

Autocoding is the computerized process of translating the specifications of an algorithm, initially expressed in a high-level modeling language such as Simulink, into source code that can be transformed further into an embedded executable binary. An example of an autocoder for optimization programs can be found in the work of Boyd [12]. One of the main ideas behind this paper is that, by combining the efficiency of the autocoding process with the rigorous proofs obtained from a formal analysis of the optimization algorithm, we can create a credible autocoding process [27] that can rapidly generate formally verifiable optimization code.

In this paper, the running example is an interior point algorithm with the Monteiro-Zhang (MZ) Newton search direction. The step length is fixed to be 1 and the input problem is a generic SDP problem obtained from system and control. The paper is organized as follows: first we introduce the basics of program verification and semi-definite programming. We then introduce the example interior point algorithm and discuss its known properties. We then give a manual example of a code implementation annotated with the semantics of the optimization algorithm using the Floyd-Hoare method [9]. Afterwards, we discuss an approach for automating the generation of the optimization semantics and the verification of the generated semantics with respect to the code. Finally, we discuss some future directions of research.

2   Credible Autocoding: General Principles

In this paper, we lay the foundations of a credible autocoding framework for convex optimization algorithms. Credible autocoding, analogous to credible compilation from [22], is a process by which the autocoding process generates formally verifiable evidence that the output source code correctly implements the input model. An overall view of a credible autocoding framework is given in figure 1. Existing work already provides for the automatic generation of embedded convex optimization code [12]. Given that proofs of high-level functional properties of interior point algorithms do exist, we want to generate the same proof that is sound for the implementation, and expressed in a formal specification language embedded in the code as comments. One of the key ingredients that made credible autocoding applicable for control systems [26] is that the ellipsoid sets generated by synthesizing quadratic Lyapunov functions are relatively easy to reason about even at the level of the code. The semantics of interior point algorithms, however, do not rely on simple quadratic invariants. The invariant obtained from the proof of good behavior of interior point algorithms is generated by a logarithmic function. This same logarithmic function can also be used in showing the optimization algorithm terminates in within a specified time. This function, is not provided, is perhaps impossible to synthesize from using exist-
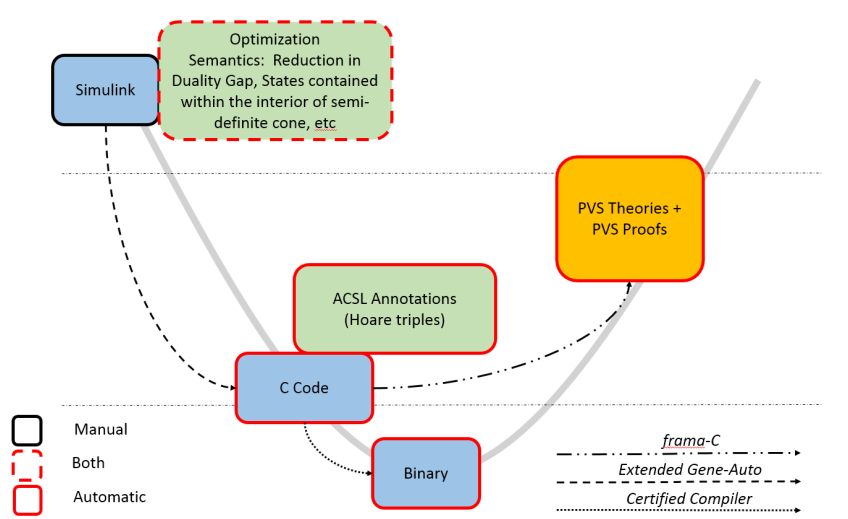
Fig. 1. Visualization of autocoding and verification process For Optimization Algorithms

ing code analysis techniques on the optimization source code. In fact nearly all existing code analyzers only handle linear properties with the notable exception of [23].

3   Program Verification

In this section, we introduce some concepts from program verification. Readers already familiar with Hoare logic and axiomatic semantics can skip ahead to the next section.

3.1   Axiomatic Semantics

One of the classic paradigms in formal verification of programs is the use of axiomatic semantics. In axiomatic semantics, the semantics or mathematical meaning of a program is based on the relations between logic predicates that hold true before and after a piece of code is executed. The program is said to be partially correct if the logic predicates holds throughout the execution of the program. For example, given the simple `while` loop program in figure 2, if we assume the value of variable x belongs to the set $[-1, 1]$ before the execution of the `while` loop, then the logic predicate `x*x<=1` holds before, during and after the execution of the `while` loop. A predicate that holds before the execution of a block of code is referred to as a pre-condition. A predicate that holds after the execution of a block of code is referred to as a post-condition. Whether a predicate is a pre or post-condition is contextual since it is dependent on the

```
1  while (x*x>0.5)
2       x=0.9*x;
3  end
```

Fig. 2. A `while` loop Program

block of code that it is mentioned in conjunction with. In other words, a pre-condition for one line of code can be the post-condition for the previous line of the code. A predicate that is preserved by a loop, i.e. holds before and after each of its multiple executions is called an *invariant*. For example, the predicate **x*x<=1** is an invariant for the `while` loop. However the predicate **x*x>=0.9** is not an invariant. Indeed it could be that it is true at the beginning of the execution of the loop (say $x = \sqrt{0.9}$) and false after executing it (we would have $x = 0.9\sqrt{0.9}$ which does not verify the invariant under consideration).

Invariants can be inserted into the code as comments. We refer to these comments as code specifications or annotations. For example, inserting the predicate **x*x<=1** into the program in figure 2 results in the annotated program in figure 3. The pseudo Matlab specification language used to express the annotations in figure 3 is modelled after the ANSI/ISO C Specification Language (ACSL [2]), which is an existing formal specification language for C programs. Invariants are denoted using the keyword *loop invariant*. Pre- and post-conditions are denoted respectively using ACSL keywords *requires* and *ensures*. The annotations are captured within comments denoted by the Matlab comment symbol %%. Throughout the rest of the paper, we use this pseudo Matlab specification lan-

```
1  %% loop invariant x*x<=1;
2  while (x*x>0.5)
3       x=0.9*x;
4  end
```

Fig. 3. Axiomatic Semantics for a `while` loop Program

guage in the annotations of the example convex optimization program. Other logic keywords from ACSL, such as *exists*, *forall* and *assumes* are also transferred over and have their usual mathematical meanings.

3.2 Hoare Logic

We now introduce a formal system of reasoning about the correctness of programs, which follows the axiomatic semantics paradigm, called Hoare Logic [9]. The main structure within Hoare logic is the Hoare triple. Let P be a pre-condition for a block of code C and let Q be a post-condition for C. The Hoare

triple denoted by $\{P\}\,C\,\{Q\}$ expresses the claim that, assuming P holds before the execution of C, then either Q will hold after its execution, or C will not terminate: indeed Hoare triples are used to show *partial correctness*, and as such have no regard for program termination in their original description.

Hoare logic includes a set of axioms and inference rules to reason about the correctness of Hoare triples for various program structures of a generic imperative programming language. Example program structures include loops, branches, jumps, etc. In this paper, we only consider `while` loops. The Hoare logic axiom for the `while` loop is

$$\frac{\{P \wedge B\}\,C;\{P\}}{\{P\}\,\text{while } B \text{ do } C \text{ done }\{\neg B \wedge P\}}. \tag{1}$$

Informally speaking, the axioms and inference rules should be interpreted as follows: the formula above the horizontal line implies the formula below that line. The rule in equation (1) expresses how, if predicate P is preserved by the execution of C (whenever the loop condition B holds), then P will be a valid loop invariant for the `while` loop that has C as its body. Such loop invariants are called *inductive invariants*. Thus, to show the correctness of the annotation in figure 3, we need to show that the predicate **x*x<=1** is preserved at every iteration of the loop (and holds initially). Axiomatic semantics also allow predicates that are essentially assumptions about the state of the program. This is especially useful in specifying properties about the inputs. For example, the variable **x** in figure 3 is assumed to have a value between -1 and 1. The validity of such a property cannot be proven since it is assumed. This type of property is referred to as an *assumption*. In our example, the assumption **x<=1 && x>=-1** is necessary to prove that **x*x<=1** is an inductive invariant of the loop.

For this paper, we use some basic inferences rules from Hoare logic, which are listed in table 1. The consequence rule in equation (2) is useful whenever a

$$\frac{\{P_1 \Rightarrow P_2\},\{P_2\}\,C\,\{Q_1\},\{Q_1 \Rightarrow Q_2\}}{\{P_1\}\,C\,\{Q_2\}} \qquad \frac{\{P\}\,C_1\,\{Q\}\,;\{Q\}\,C_2\,\{W\}}{\{P\}\,C_1;C_2\,\{W\}} \qquad (3)$$
$$(2)$$

$$\frac{}{\{P\}\,SKIP\,\{P\}} \qquad (4) \qquad \frac{}{\{P[e/x]\}\,x := expr\,\{P\}} \qquad (5)$$

$$\frac{}{\{P\}\,x := expr\,\{\exists x_0\,(x = expr\,[x_0/x]) \wedge P\,[x_0/x]\,\}}$$
$$(6)$$

Table 1. Axiomatic Semantics Inference Rules for a Imperative Language

stronger pre-condition or weaker post-condition is needed. By stronger, we mean the set defined by the predicate is smaller. By weaker, we mean precisely the opposite. The substitution rules in (5) and (6) are used when the piece of code under

analysis is an assignment statement. The weakest pre-condition $P[x/expr]$ in (5) means P with all instances of the expression $expr$ replaced by x. For example, given a line of code $\mathbf{y=x+1}$ and a known weakest pre-condition $\mathbf{x+1<=1}$, we can deduct that $\mathbf{y<=1}$ is a correct post-condition using the backward substitution rule. Equation (5) is usually used to compute the weakest pre-condition from a known post-condition. Alternatively the forward propagation rule in (6) is used to compute the strongest post-condition. Note that, when x does not appear in P in the substitution rule in equation (5), it simply becomes $\{P\}\, x := expr\, \{P\}$. This is useful to propagate properties over parts of the code that do not affect them.

3.3   Proof Checking

The utility of having the invariants in the code is that finding the invariants is in general more difficult than checking that given invariants are correct. By expressing and translating the high-level functional properties and their proofs onto the code level in the form of invariants, we can verify the correctness of the optimization program with respect to its high-level functional properties using a proof-checking procedure i.e. by verifying each use of a Hoare logic rule.

4   Semi-Definite Programming and the Interior Point Method

In this section, we give an overview of the Semi-Definite Programming (SDP) problem. The readers who are already familiar with interior point method and convex optimization can skip ahead to the next section. The notations used in this section are as follows: let $A = (a_{i,j})_{1 \le i,j \le n}, B \in \mathbb{R}^{n \times n}$ be two matrices and $a, b \in \mathbb{R}^n$ be two column vectors. $\mathrm{Tr}\,(A) = \sum_{i=1}^{n} a_{i,i}$ denotes the trace of matrix A. $\langle \cdot, \cdot \rangle$ denotes an inner product, defined in $\mathbb{R}^{n \times n} \times \mathbb{R}^{n \times n}$ as $\langle A, B \rangle := \mathrm{Tr}\,(A^{\mathrm{T}} B)$ and in $\mathbb{R}^n \times \mathbb{R}^n$ as $\langle a, b \rangle := a^{\mathrm{T}} b$. The Frobenius norm of A is defined as $\|A\|_F = \sqrt{\langle A, A \rangle}$. The symbol $\mathbb{S}^n$ denotes the space of symmetric matrices of size $n \times n$. The space of $n \times n$ symmetric positive-definite matrices is denoted as $\mathbb{S}^{n+} = \left\{ S \in \mathbb{S}^n | \forall x \in \mathbb{R}^n \setminus \{0\}, x^{\mathrm{T}} S x > 0 \right\}$. If A and B are symmetric, $A \prec B$ (respectively $A \succ B$) denotes the positive (respectively negative)-definiteness of matrix $B - A$. The symbol I denotes an identity matrix of appropriate dimension. For $X, Z \in \mathbb{S}^{n+}$, some basic properties of matrix derivative are $\dfrac{\partial \, \mathrm{Tr}\,(XZ)}{\partial X} = Z^{\mathrm{T}} = Z$ and $\dfrac{\partial \det (X)}{\partial X} = \det (X)^{-1} \left(X^{-1} \right)^{\mathrm{T}} = \det (X)^{-1} X^{-1}$.

4.1   SDP Problem

Let $n, m \in \mathbb{N}$, $F_0 \in \mathbb{S}^{n+}$, $F_1, F_2, \ldots, F_m \in \mathbb{S}^n$, and $b = \begin{bmatrix} b_1 \ b_2 \ \ldots \ b_m \end{bmatrix}^{\mathrm{T}} \in \mathbb{R}^m$. Consider a SDP problem of the form in equation (7). The linear objective function $\langle F_0, Z \rangle$ is to be maximized over the intersection of positive semi-definite cone

$\{Z \in \mathbb{S}^n | Z \succeq 0\}$ and a convex region defined by m affine equality constraints.

$$\begin{aligned}
\sup_{Z} \quad & \langle F_0, Z \rangle, \\
\text{subject to} \quad & \langle F_i, Z \rangle + b_i = 0, \quad i = 1, \ldots, m \\
& Z \succeq 0.
\end{aligned} \tag{7}$$

We denote the SDP problem in (7) as the *dual* form. Closely related to the dual form, is another SDP problem as shown in (8), called the *primal* form. In the primal formulation, the linear objective function $\langle b, p \rangle$ is minimized over all vectors $p = \begin{bmatrix} p_1 \ldots p_m \end{bmatrix}^{\mathrm{T}} \in \mathbb{R}^m$ under the semi-definite constraint $F_0 + \sum_{i=1}^{m} p_i F_i \preceq$ 0. Note the introduction in (8) of a variable $X = -F_0 - \sum_{i=1}^{m} p_i F_i$ such that $X \succeq 0$, which is not strictly needed to express the problem, but is used in later developments.

$$\begin{aligned}
\inf_{p, X} \quad & \langle b, p \rangle \\
\text{subject to} \quad & F_0 + \sum_{i=1}^{m} p_i F_i + X = 0 \\
& X \succeq 0.
\end{aligned} \tag{8}$$

We assume the primal and dual feasible sets defined as

$$\begin{aligned}
\mathcal{F}^p &= \left\{ X | X = -F_0 - \sum_{i=1}^{m} p_i F_i \succeq 0, p \in \mathbb{R}^m \right\}, \\
\mathcal{F}^d &= \{ Z | \langle F_i, Z \rangle + b_i = 0, Z \succeq 0 \}
\end{aligned} \tag{9}$$

are not empty. Under this condition, for any primal-dual pair (X, Z) that belongs to the feasible sets in (9), the primal cost $\langle b, p \rangle$ is always greater than or equal to the dual cost $\langle F_0, Z \rangle$. The difference between the primal and dual costs for a feasible pair (X, Z) is called the *duality gap*. The duality gap is a measure of the optimality of a primal-dual pair. The smaller the duality gap, the closer to optimal the solution pair (X, Z) is. For (8) and (7), the duality gap is the function

$$G(X, Z) = \mathrm{Tr}\,(XZ). \tag{10}$$

Indeed,

$$\begin{aligned}
\mathrm{Tr}\,(XZ) &= \mathrm{Tr}\left(\left(-F_0 - \sum_{i=1}^{m} p_i F_i \right) Z \right) = -\,\mathrm{Tr}\,(F_0 Z) - \sum_{i=1}^{m} p_i \,\mathrm{Tr}\,(F_i Z) \\
&= \langle b, p \rangle - \langle F_0, Z \rangle.
\end{aligned}$$

Finally, if we assume that both problems are strictly feasible i.e. the sets

$$\begin{aligned}
\mathcal{F}^{p'} &= \left\{ X | X = -F_0 - \sum_{i=1}^{m} p_i F_i \succ 0, p \in \mathbb{R}^m \right\}, \\
\mathcal{F}^{d'} &= \{ Z | \langle F_i, Z \rangle + b_i = 0, Z \succ 0 \}
\end{aligned} \tag{11}$$

are not empty, then there exists an optimal primal-dual pair (X^*, Z^*) such that

$$\text{Tr}\,(X^* Z^*) = 0. \tag{12}$$

Moreover, the primal and dual optimal costs are guaranteed to be finite. The condition in equation (12) implies that for strictly feasible problems, the primal and dual costs are equal at their respective optimal points X^* and Z^*. Note that in the strictly feasible problem, the semi-definite constraints become definite constraints.

The canonical way of dealing with constrained optimization is by first adding to the cost function a term that increases significantly if the constraints are not met, and then solve the unconstrained problem by minimizing the new cost function. This technique is commonly referred to as the *relaxation* of the constraints. For example, lets assume that the problems in (8) and (7) are strictly feasible. The positive-definite constraints $X \succ 0$ and $Z \succ 0$, which define the *interior* of a pair of semi-definite cones, can be relaxed using an indicator function $I(X, Z)$ such that

$$I : (X, Z) \to \begin{cases} 0, & X \succ 0, Z \succ 0 \\ +\infty, & \text{otherwise} \end{cases} \tag{13}$$

The intuition behind relaxation using an indicator function is as follows. If the primal-dual pair (X, Z) approaches the boundary of the interior region, then the indicator function $I(X, Z)$ approaches infinity, thus incurring a large penalty on the cost function.

The indicator function in (13) is not convenient for optimization because it is not differentiable. Instead, the indicator function can be replaced by a family of smooth, convex functions $B(X, Z)$ that not only approximate the behavior of the indicator function but are also *self-concordant*. We refer to these functions as *barrier* functions. A scalar function $F : \mathbb{R} \to \mathbb{R}$, is said to be self-concordant if it is at least three times differentiable and satisfies the inequality

$$|F'''(x)| \leq 2F''(x)^{\frac{3}{2}}. \tag{14}$$

The concept of self-concordance has been generalized to vector and matrix functions, thus we can also find such functions for positive-definite variables X and Z. Here we state, without proof, the key property of self-concordant functions.

Property 1. Functions that are self-concordant can be minimized in polynomial time to a given non-zero accuracy using a Newton type iteration [18].

Examples of self-concordant functions include linear functions, quadratic functions, and logarithmic functions. A valid barrier function for the semi-definite constraints from (8) and (7) is

$$B(X, Z) = -\log \det (X) - \log \det (Z). \tag{15}$$

5 An Interior Point Algorithm and Its Properties

We now describe an example primal-dual interior point algorithm. We focus on the key property of convergence. We show its usefulness in constructing the

inductive invariants to be applied towards documenting the software implementation. The algorithm is displayed in table 2 and is based on the work in [16].

5.1 Details of the Algorithm

The algorithm in table 2 consists of an initialization routine and a `while` loop. The operator **length** is used to compute the size of the input problem data. The operator $\hat{\ }^{-1}$ represents an algorithm such as QR decomposition that returns the inverse of the matrix. The operator $\hat{\ }^{0.5}$ represents an algorithm such as the Cholesky decomposition that computes the square root of the input matrix. The operator **lsqr** represents a least-square QR factorization algorithm that is used to solve linear systems of equation of the form $Ax = b$. Under the assumption of real algebra, all of these operators return exact solutions.

In the initialization part, the states X, Z and p are initialized to feasible values, and the input problem data are assigned to constants $F_i, i = 1, \ldots, m$. The term feasible here means that X, Z, and p satisfies the equality constraints of the primal and dual problems. The efficiency of the initialization process is discussed in later sections.

The `while` loop is a Newton iteration that computes the zero of the derivative of the potential function

$$\phi(X, Z) = \left(n + \nu\sqrt{n}\right) \log \operatorname{Tr}(XZ) - \log \det(XZ) - n \log n, \qquad (16)$$

in which ν is a positive weighting factor. Note that the potential function is a weighted sum of the primal-dual cost gap and the barrier function potential. The weighting factor ν is used in computing the duality gap reduction factor $\sigma \equiv \dfrac{n}{n + \nu\sqrt{n}}$. A larger ν implies a smaller σ, which then implies a shorter convergence time. For our algorithm, since we use a fix-step size of 1, a small enough σ combined with the newton step could result in a pair of X and Z that no longer belong to the interior of the positive-semidefinite cone. In the running example, we have $\nu = 0.4714$. While this choice of ν doubled the number of iterations of the running example compared to the typical choice of $\nu = 1$, it is critical in satisfying the invariants of the `while` loop that are introduced later this paper.

Let symbol $T = Z^{-0.5}$ and T_{inv} denotes the inverse of T. The `while` loop solves the set of matrix equations

$$\langle F_i, \Delta Z \rangle = 0$$

$$\sum_i^m \Delta p_i F_i + \Delta X = 0$$

$$\frac{1}{2}\left(T\left(Z\Delta X + \Delta Z X\right)T_{inv} + T_{inv}\left(\Delta X Z + X \Delta Z\right)T\right) = \sigma\mu I - T_{inv}XT_{inv}.$$

$$(17)$$

for the Newton-search directions ΔZ, ΔX and Δp. The first two equations in (17) are obtained from a Taylor expansion of the equality constraints from the

Algorithm 1. MZ Short-Path Primal-Dual Interior Point Algorithm

Input: $F_0 \succ 0$, $F_i \in \mathbb{S}^n, i = 1, \ldots, m$, $b \in \mathbb{R}^m$

   ϵ: Optimality desired

1. Initialize:

   Compute Z such that $\langle F_i, Z \rangle = -b_i, i = 1, \ldots, m$;

   Let $X \leftarrow \hat{X}$; // $\hat{X}$ *is some positive-definite matrix*

   Compute p such that $\displaystyle\sum_i^m p_i F_i = -X_0 - F_0$;

   Let $\mu \leftarrow \dfrac{\langle X, Z \rangle}{n}$;

   Let $\sigma \leftarrow 0.75$;

   Let $n \leftarrow \text{length } F_i$, $m \leftarrow \text{length } b_i$;

2. **while** $n\mu > \epsilon$ {

3.   Let $\phi_- \leftarrow \langle X, Z \rangle$;

4.   Let $T_{inv} \leftarrow Z^{0.5}$;

5.   Let $T \leftarrow T_{inv}^{-1}$;

6.   Compute $(\Delta Z, \Delta X, \Delta p)$ that satisfies (17);

7.   Let $Z \leftarrow Z + \Delta Z$, $X \leftarrow X + \Delta X$, $p \leftarrow p + \Delta p$;

8.   Let $\phi \leftarrow \langle X, Z \rangle$;

9.   Let $\mu \leftarrow \dfrac{\langle X, Z \rangle}{n}$;

10.   **if** $(\phi - \phi_- > 0)$ {

11.   return;

        }

    }

Table 2. Primal-Dual Short Path Interior Point Algorithm

primal and dual problems. These two constraints formulate the feasibility sets as defined in Eq (9). The last equation in (17) is obtained by setting the Taylor expansion of the derivative of (16) equal to 0, and then applying the symmetrizing transformation

$$H_T : M \to \frac{1}{2} \left(TMT^{-1} + \left(TMT^{-1} \right)^{\mathrm{T}} \right), T = Z^{-0.5} \tag{18}$$

to the result. To see this, note that derivative of (16) is $\left[XZ - \frac{n}{n + \nu\sqrt{n}} \frac{\mathrm{Tr}(XZ)}{n} I \right.$

$\left. ZX - \frac{n}{n + \nu\sqrt{n}} \frac{\mathrm{Tr}(XZ)}{n} I \right]$.

The transformation in (18) is necessary to guarantee that the solution ΔX is symmetric. The parameter σ, as mentioned before, can be interpreted as a duality gap reduction factor. To see this, note that the 3rd equation in (17) is the result of applying Newton iteration to solve the equation $XZ = \sigma\mu I$. With $\sigma \in (0, 1)$, the duality gap $\mathrm{Tr}(XZ) = n\sigma\mu$ is reduced after every iteration. The choice of T in (18) is taken from [15] and is called the Monteiro-Zhang (MZ) direction. Many of the Newton search directions from the interior-point method literature can be derived from an appropriate choice of T. The M-Z direction also guarantees a unique solution ΔX to (17). The `while` loop then updates the states X, Z, p with the computed search directions and computes the new normalized duality gap. The aforementioned steps are repeated until the duality gap $n\mu$ is less than the desired accuracy ϵ.

5.2   High-level Functional Property of the Algorithm

A key high-level functional property of the interior point algorithm in table (2) is an upper bound on its worst case computational time. This upper bound can be derived from a constant reduction in the potential function such as the one in (16). Since the convergence property of interior-point algorithms is well-established in the literature (for example see [10] and [20]), here we give a result without proof about the convergence property of our example algorithm in table 2. We have the following theorem, which gives us a tight upper bound on the convergence time of our example algorithm. Note that the main structure of the algorithm is a `while` loop that updates the values of the variables X, Z, and p during each iteration of the loop.

Theorem 1. *Let X, Z, and p denote their respective values at the ith iteration of the loop, and let X_-, Z_-, and p_- denote their respective values at the $i-1$th iteration. Given $\phi : (X, Z) \to \log(\mathrm{Tr}(XZ))$, then there exists a $\delta > 0$ such that*

$$\phi(X_-, Z_-) - \phi(X, Z) \geq \delta, \tag{19}$$

holds for every iteration of the loop.

For safety-critical applications, it is vital that the optimization code comes with a rigorous guarantee of convergence within a specified time. We assume that the

required precision ϵ, and the problem data size n are known beforehand. As a consequence of theorem 1, one can see that it takes at most $\mathcal{O}\left(\sqrt{n}\log\epsilon^{-1}\operatorname{Tr}(X_0 Z_0)\right)$ iterations until the duality gap $\operatorname{Tr}(XZ)$ converges to $\epsilon' < \epsilon$ i.e. the loop terminates.

For annotating the `while` loop of the implementation, we want to construct an inductive invariant of the form

$$0 \le \phi(X, Z) \le 1. \tag{20}$$

however the function $\phi(X, Z)$ specified in theorem 1 is not positive. Although theorem 1 also holds with $\phi(X, Z)$ from (16) [13], however that funciton is also not positive. To construct an invariant in the form of (20), we can just use the duality gap i.e.

$$\phi(X, Z) = \operatorname{Tr}(XZ). \tag{21}$$

An important implication of theorem 1 is that the duality gap $\operatorname{Tr}(XZ)$ converges to 0 linearly i.e. $\exists \kappa \in (0, 1)$ such that $\operatorname{Tr}(XZ) \le \kappa \operatorname{Tr}(X_- Z_-)$, in which X_- and Z_- are values of X and Z at the previous iteration. Using just $\operatorname{Tr}(XZ)$, we can construct one invariant of the form in (20) and a second invariant $\phi - \kappa\phi_- < 0$ to express the linear convergence property implied by theorem 1.

There are several other inductive invariants to be annotated for the `while` loop. For example, X and Z need to be positive-definite for the duality gap function to be positive. Not only do we need to check that X and Z are initialized to symmetric matrices within the interior of the semi-definite cone, we also need to show that they are guaranteed to remain in that cone throughout the execution of the `while` loop.

6 Annotating an Optimization Program for Deductive Verification

In this section, we discuss, line by line, an example of a fully-annotated optimization program that is based on the interior-point algorithm in figure 2. The fully-annotated program represents a claim of proof that the code also conforms to certain safety or liveness properties of the algorithm. The claim of proof needs to be verified by a proof-checking program, which we call the *backend*. The annotated program was written in the Matlab language and the Hoare logic style annotations were inserted manually. Matlab was chosen as the implementation language because of its compactness and readability. For the automation of the process i.e. the credible autocoding framework, a realistic target language such as C should be chosen, as it is more acceptable for formal analysis and verification.

The annotation process starts with the selection of a safety or liveness property of the interior-point algorithm from which the code originates. The chosen property is formalized and then translated into invariants, and then inserted into the code as Hoare logic pre- and post-conditions. For the example optimization program, the invariant that encode the property of bounded execution time of the program is $\phi - 0.76\phi_- < 0$. Various additional proof elements are inserted

into the code, also in the form of Hoare logic style pre- and post-conditions, to allow easy automation of the proof-checking process. The amount of additional proof elements needed is dependent on the capability of the backend. Lastly, by using the Hoare logic rules, the entire program is populated with pre- and post-conditions, thus resulting in a fully-annotated code.

The rest of this section is organized as follows. First we give the example Matlab implementation and a description of the non-standard Matlab functions used in the code. Second, we give a description of the annotation language used in expressing the invariants and assertions. Finally, we give a detailed description of all the Hoare triples in the fully-annotated example.

6.1 Matlab Implementation

The example implementation uses three non-standard Matlab operators **vecs**, **mats** and **krons**. These operators are used to transform the matrix equations in equation (17) into matrix vector equations of the form $Ax = b$. More details about these operator are given in the appendix. The symbol ' denotes Matlab's transpose function.

The program is divided into two parts. The first part, which we call the initialization part, assigns the data of the input optimization problem to memory and initializes the variables to be optimized. The second part, which is the `while` loop, executes the interior-point algorithm to solve the input optimization problem.

The input data is obtained from a generic optimization problem taken from systems and control. The details of the original problem are skipped here as they have no bearing on the main contribution of this article. Matrices $F_i, i = 0, \ldots, 3$ were computed from the original problem using the tool Yalmip [11].

6.2 Annotation Language

The fully-annotated example is displayed in section 11.2 of the appendix. The annotations are expressed using a pseudo Matlab specification language (MSL) that is similar, in features, to the ANSI/ISO C Specification Language (ACSL) for C programs [4]. Like in ACSL, the keywords **requires** and **ensures** denote a pre and post-condition statement respectively. A MSL contract, like the ACSL contract, is used to express a Hoare triple. For example, the MSL contract displayed in figure 5 can be parsed as the Hoare triple $\{P\} C; \{Q\}$.

A block of code can have more than one contract. For example, as shown in figure 6, the two MSL contracts translates to the Hoare triples

$$\{P_i\} C; \{Q_i\}, i = 1, 2. \tag{22}$$

A block of empty code can alsobe annotated by contract. For example, the formula $P \Rightarrow Q$ can be expressed as the Hoare triple $\{P\} empty; \{Q\}$, which is equivalent to the MSL contract in figure 7.

```
1   F0=[1, 0; 0, 0.1];
2   F1=[-0.750999 0.00499; 0.00499 0.0001];
3   F2=[0.03992 -0.999101; -0.999101 0.00002];
4   F3=[0.0016 0.00004; 0.00004 -0.999999];
5   b=[0.4; -0.2; 0.2];
6   n=length(F0);
7   m=length(b);
8   F=[vecs(F1); vecs(F2); vecs(F3)];
9   Ft=F';
10  Z=mats(lsqr(F,-b),n);
11  X=[0.3409 0.2407; 0.2407 0.9021];
12  epsilon=1e-8;
13  sigma=0.75;
14  phi=trace(X*Z);
15  phim=1/0.75*phi;
16  P=mats(lsqr(Ft,vecs(-X-F0)),n);
17  p=vecs(P);
18  while (phi>epsilon) {
19      Xm=X;
20      Zm=Z;
21      pm=p;
22      mu=trace(Xm*Zm)/n;
23      Zh=Zm^(0.5);
24      Zhi=Zh^(-1);
25      G=krons(Zhi,Zh'*Xm,n,m);
26      H=krons(Zhi*Zm,Zh',n,m);
27      r=sigma*mu*eye(n,n)-Zh*Xm*Zh;
28      dZm=lsqr(F,zeros(m,1));
29      dXm=lsqr(H, vecs(r)-G*dZm);
30      dpm=lsqr(Ft,-dXm);
31      p=pm+dpm
32      X=Xm+mats(dXm,n);
33      Z=Zm+mats(dZm,n);
34      phim=trace(Xm*Zm);
35      phi=trace(X*Z);
36      mu=trace(X*Z)/n;
37  }
```

Fig. 4. Optimization Program in Matlab

```
1   % requires P
2   % ensures  Q
3   {
4      C;
5   }
```

Fig. 5. An ACSL-like Contract

```
1  % requires P1
2  % ensures  Q1
3
4  % requires P2
5  % ensures  Q2
6  {
7     C;
8  }
```

Fig. 6. Multiple Contracts

```
1  % requires P
2  % ensures  Q
3  {
4     % empty;
5  }
```

Fig. 7. Contract for an empty code

Remark 1. Such empty code block annotations are used to decompose a proof to its most basic elements. Indeed, assume we are led to write a Hoare Triple:

$$\{P\}\,C;\{Q_2\} \tag{23}$$

where Q_2 does not naturally appear as a consequence of executing C when P holds. If there exists a more natural consequence Q_1 for the triple, and if $Q_1 \Rightarrow Q_2$, the triple can be expanded into two Hoare triples:

$$\{P\}\,C;\{Q_1\} \tag{24}$$

and

$$\{Q_1\}\,empty;\{Q_2\}\,. \tag{25}$$

The main reason to expand a Hoare triple into multiple ones is to simplify the automation of the proof-checking process. The choice in expanding is arbitrary as it is dependent on the capability of the backend. For example, in the case of (23), it is possible that the backend can verify (24) and (25) separately but not (23) without human input. Practically speaking, by reducing the complexity in the verification of an individual Hoare triple in exchange for an increase in the total number of Hoare triples, the proof-checking process usually become easier to automate.

A contract can have more than one pre-condition statement as shown in figure 8. The pre-condition statements are combined into a single pre-condition P conjunctively when the contract is parsed as a Hoare triple. For example, the contract in figure 8 expresses the Hoare triple $\{P_1 \wedge P_2\}\,empty;\{Q\}$.

```
1  % requires P1
2  % requires P2
3  % ensures   Q
4  {
5    C;
6  }
```

Fig. 8. Contract for an empty code

The logic comparison operator $>$ is overloaded to express $\succ$. The function **smat** is the inverse of the function **svec**. The function **svec** is similar to the symmetric vectorization function **vecs** but with a multiplication factor of 2 outside the diagonal. For example, the expression **smat([0.4;-0.2;0.2])** returns the matrix

$$\begin{bmatrix} 0.4 & -0.1 \\ -0.1 & 0.2 \end{bmatrix}. \tag{26}$$

6.3   Functions with Verified Contracts

Certain functions used in the example program come by default with MSL contracts to ensure the regularity of the inputs and outputs to the functions. These annotations are implicit since they are assumed to be correct.

For example, the square root function and the inverse function denoted by the symbols $^{0.5}$ and $^{-1}$ respectively, come by default with the properties that the input matrix is symmetric positive-definite and the output is also symmetric positive-definite.

```
1  % requires Zm>0;
2  % ensures   Zh>0;
3  Zh=Zm^(0.5);
4  % requires Zh>0;
5  % ensures   Zhi>0;
6  Zhi=Zh^(-1)
```

The function **lsqr** comes by default with the post-condition which stipulates that the matrix-vector product of the first argument of the function and the output of the function is equal to the second argument of the function.

```
1  % ensures F*y==-b;
2  y=lsqr(F,-b);
```

The function **vecs** by default comes with the pre-condition that the input argument to the function is a symmetric matrix.

```
1  % requires P==transpose(P);
2  p=vecs(P);
```

The function **mats**, as displayed below, requires that $x \in \mathbb{R}^{\frac{n(n+1)}{2}}$ and ensures that the output X is a symmetric matrix of size n.

```
1  % requires  n>=1;
2  % requires  type(x)==vector(n/2*(n+1));
3  % ensures   type(X)==symmetric_matrix(n);
4  X=mats(x,n);
```

A first effort in the automatic generation of contracts for the basic building blocks of high level languages can be found in [6].

6.4 Annotated Code

For the analysis and discussion of the annotations, the fully-annotated example is split into multiple parts and each part is displayed in a separate figure and discussed in a separate subsection. In these figures, every MSL contract is assigned a pair of numbers (n, m), which indicates the mth contract of the nth block of code of the figure. The Hoare triple expressed by the contract with a label of (n, m) is denoted as

$$\mathcal{H}_{n,m} := \{P_{n,m}\} \, C_n; \{Q_{n,m}\}. \tag{27}$$

For example, in figure 6.4, $\mathcal{H}_{1,1}$ is the Hoare triple $\{x > 0\} \, x = y; \{y > 0\}$, with $P_{1,1}$ being the pre-condition $x > z$, C_1 being the line of code **y=x-z;** and $Q_{1,1}$ being the post-condition $y > 0$. Also in figure 6.4, $\mathcal{H}_{1,2}$ is the Hoare triple

```
1  % (1,1) requires  x>z;
2  %       ensures   y>0;
3
4  % (1,2) requires  x<0;
5  %       requires  z>0;
6  %       ensures   y<0;
7  {
8      y=x-z;
9  }
```

$\{x < 0 \wedge z > 0\} \, y = x - z; \{y < 0\}$, with $P_{1,2} := x < 0 \wedge z > 0$, and $Q_{1,2} := y < 0$.

The variables from the program will be referenced using either their programmatic textual representation or their corresponding mathematical symbols. For example, the list of variables **phi**, **phim**, **dXm**, and **Xm** is the same as the list of symbols ϕ, ϕ_-, Δ_{X_-}, and X_- Likewise, the annotations are also referenced using both representations as well. For example, the expression **phim=trace(Xm*Zm)/n** is equivalent to $\phi_- := \dfrac{\mathrm{Tr}\,(X_-Z_-)}{n}$. Some expressions from the program such as **mats(dXm,n)** and **mats(dZm,n)** are referenced using the more compact symbols ΔX_- and ΔZ_- respectively.

6.5   Proof Checking the Annotations

We assume there exists an automated proof-checking program, which we refer to as the `backend`, that can be used to verify the Hoare triples in the annotated example. We will not go into much details about the backend other than occasionally discussing some of theories and formulas needed to verify certain annotations.

6.6   Initialization Part I

The first part of initialization code along with its annotations is displayed in figure 9. The annotations described in this subsection of the paper are from this figure unless explicitly stated otherwise.

In $C_i, i = 1, \ldots, 5$, the data parameters of the input optimization problem are assigned to the appropriate variables. The inserted post-conditions $P_{i,1}, i = 1, \ldots, 5$ represent regularity conditions on the data parameters of the input problem. The correctness of these post-conditions guarantee that the input optimization problem is well-posed. For example, the post-condition $P_{1,1}$ claims that the variable **F0** is positive-definite after the execution of C_1 in line 2. Another example is $Q_{2,1}$ which claims that after execution of C_2 in line 4, the variable **F1** is a symmetric matrix.

In C_6 and C_y, the problem sizes are computed. In our example, variable **m** denotes the number of equality constraints in the dual formulation and variable **n** denotes the dimensions of the optimization variables X and Z. The inserted post-conditions $Q_{6,1}$ and $Q_{7,1}$ asserts that the problem sizes are at least one.

6.7   Initialization Part II

The annotations described in this subsection of the paper are from figure 10 unless explicitly stated otherwise.

The blocks of code $C_i, i = 1, 2$ generate and then assign initial values to variables X and Z. The values of variables X and Z need to be positive-definite in order to satisfy the strict feasibility property of the input optimization problem. Furthermore, functions used in the latter part of the program such as $\char`\^{0.5}$ and $\char`\^{-1}$ implicitly require the input to be positive-definite. To check if X and Z are positive-definite after they are initialized, we insert the post-conditions $Q_{1,1}$ and $Q_{2,1}$. The correctness of $\mathcal{H}_{i,1}, i = 1, 2$ can be checked by a backend that can verify if a matrix is positive definite. For C_2, we insert a second contract to check if the quantity of the duality gap, defined as the inner product X and Z or **trace(X*Z)**, is bounded from above by 0.1.

Using the skip rule from table 1, the post-conditions $Q_{1,1}$ and $Q_{2,1}$ are propagated forward, and combined conjunctively to form $P_{3,1}$. Pre-condition $P_{3,1}$ is an invariant to be annotated for the `while` loop By the application of the theory

$$X \succ 0 \wedge Z \succ 0 \Rightarrow \mathrm{Tr}\,(XZ) > 0, \tag{28}$$

```
1  % (1,1) ensures F0>0;
2  {
3    F0=[1, 0; 0, 0.1];
4  }
5  % (2,1) ensures transpose(F1)==F1;
6  {
7    F1=[-0.750999 0.00499; 0.00499 0.0001];
8  }
9  % (3,1) ensures transpose(F2)==F2;
10 {
11   F2=[0.03992 -0.999101; -0.999101 0.00002];
12 }
13 % (4,1) ensures transpose(F3)==F3;
14 {
15   F3=[0.0016 0.00004; 0.00004 -0.999999];
16 }
17 % (5,1) ensures smat(b)>0;
18 {
19   b=[0.4; -0.2; 0.2];
20 }
21 F=[vecs(F1); vecs(F2); vecs(F3)];
22 % (6,1) ensures Ft==transpose(F);
23 {
24   Ft=F';
25 }
26 % (7,1) ensures n>=1;
27 {
28   n=length(F0);
29 }
30 % (8,1) ensures m>=1;
31 {
32   m=length(b);
33 }
```

Fig. 9. Initialization

```
1  % (1,1) ensures Z>0;
2  {
3      Z=mats(lsqr(F,-b),n);
4  }
5  % (2,1) ensures X>0
6
7  % (2,2) ensures trace(X*Z)<=0.1;
8  {
9      X=[0.3409 0.2407; 0.2407 0.9021];
10 }
11 % (3,1) requires Z>0 && X>0;
12 %        ensures   trace(X*Z)>0;
13 {
14     % empty code
15 }
16 % (4,1) ensures transpose(P)==P;
17 {
18     P=mats(lsqr(Ft,vecs(-X-F0)),n);
19     p=vecs(P);
20 }
21 % (5,1) ensures epsilon>0
22 {
23     epsilon=1e-8;
24 }
25 % (6,1) ensures sigma==0.75;
26 {
27     sigma=0.75;
28 }
29 % (7,1) requires trace(X*Z)<=0.1;
30 %        ensures   phi<=0.1;
31
32 % (7,2) requires trace(X*Z)>0;
33 %        ensures   phi>0;
34
35 % (7,3) ensures   phi==trace(X*Z);
36 {
37     phi=trace(X*Z);
38 }
39 % (8,1) requires phi>0;
40 %        ensures   phi-0.76/0.75*phi<0;
41 {
42     % empty code
43 }
44 % (9,1) requires phi-0.76/0.75*phi<0;
45 %        ensures   phi-0.76*phim<0;
46 {
47     phim=1/0.75*phi;
48 }
49 % (10,2) ensures   mu==trace(X*Z)/n;
50 {
51     mu=trace(X*Z)/n;
52 }
```

Fig. 10. Initialization Part II

we get the post-condition $Q_{3,1}$. Post-condition $Q_{3,1}$ is a claim that the duality gap is also bounded from below by 0. For the automatic checking of $\mathcal{H}_{3,1}$, the backend can use the same theory from (28).

Next in C_4, the initial value for p is computed using the matrix equality $F_0 + \sum_{i}^{m} p_i F_i + X = 0$, which is taken from the primal formulation. Here, we only insert the post-condition **transpose(P)==P** since it is one of the implicit pre-conditions of the function **vecs** in line 19.

We move on to C_5, which assigns the value 1×10^{-8} to the variable **epsilon**. This value is essentially a measure of the desired optimality and it is also an important factor in computing an upper bound on the number of iterations of the **while** loop. The proof of the bounded time termination of the loop requires the desired optimality to be greater than 0, hence the insertion of $Q_{5,1}$. The next block of code, which is C_6 assigns the value of 0.75 to the variable σ. The inserted post-condition, which is $P_{6,1}$ ensures that the variable σ is equal to the value 0.75. This trivial condition is to be used later in the annotation process. The verification of $Q_{i,1}, i = 5, 6$ should be automatic for any theorem prover.

Statement C_7 computes the duality gap $\text{Tr}(X * Z)$ and then assigns the result to the variable **phi**. There are three contracts for C_7. For $\mathcal{H}_{7,j}, j = 1, 2$, the pre-conditions $P_{7,1}$ and $P_{7,2}$, which correspond to **Trace(X*Z)>0** and **Trace(X*Z)<=0.1** respectively, are obtained by applying the skip rule to $Q_{2,2}$ and $Q_{3,1}$. The post-conditions $Q_{7,1}$ and $Q_{7,2}$ are generated using the backward substitution rule from 1. They are also invariants to be annotated for the **while** loop. The last contract of C_7 contains a trivially true post-condition **phi==trace(X*Z)** to be used later in the annotation process.

Now moving on to C_8, we see another inserted empty block of code (line 42) with the pre-condition **phi>0**. The pre-condition is simply $Q_{7,2}$ propagated forward. Before we discuss the post-condition in $\mathcal{H}_{8,1}$, we first jump ahead to next block of code which is C_9. For C_9, the inserted post-condition **phi-0.76*phim<0** is an invariant obtained by formalizing the property of finite termination of the interior point algorithm. This post-condition is an invariant to annotate the **while** loop. Applying the backwards substitution rule on $Q_{9,1}$, we get the pre-condition $P_{9,1}$ which is exactly the post-condition for C_8. Since C_8 is an empty block of code, the correctness of the Hoare triple $\mathcal{H}_{8,1}$ reduces to verifying the formula $P_{8,1} \Rightarrow Q_{8,1}$ which is equivalent to showing

$$c > 1 \wedge \phi > 0 \Rightarrow \phi - c\phi < 0 \wedge 0.76/0.75 > 1 \tag{29}$$

which can be discharged automatically by existing tools such as certain SMT solvers [5].

Finally, we insert a trivially true post-condition for the last block of code in figure 10. The post-condition simply ensures that the variable **mu** is equal to the expression **trace(X*Z)/n** after the latter has been assigned to the former.

In the next few sections, we show that $Q_{7,1} \wedge Q_{7,2} \wedge Q_{9,1}$ holds throughout execution of the **while** loop, thereby proving that optimization program terminates with an answer within a bounded time. We also show that $P_{3,1}$ holds as

well throughout the entire execution of the loop, thereby completing the proof of $Q_{7,1} \wedge Q_{7,2} \wedge Q_{9,1}$.

6.8   The `while` loop

The annotations and code discussed in this subsection are from figure 11 unless explicitly stated otherwise.

```
1  % (1,1) requires phi>0 && phi<=0.1;
2  %        ensures  phi>0 && phi<=0.1;
3
4  % (1,2) requires phi-0.76*phim<0;
5  %        ensures  phi-0.76*phim<0;
6
7
8  % (1,3) requires Z>0 && X>0;
9  %        ensures  Z>0 && X>0;
10 {
11     while (phi>epsilon) do
12         .
13         .
14         .
15     end
16 }
```

Fig. 11. The While Loop

The contracts on the `while` loop are constructed using the invariants $Q_{7,1} \wedge Q_{7,2}$, $Q_{9,1}$ and $P_{3,1}$, in which $P_{3,1}$, $Q_{7,1}$, $Q_{7,2}$ and $Q_{9,1}$ are from figure 9. By the application of the while axiom from (1), each invariant becomes the pre and post-condition of a contract. In the next few subsections of the paper, we discuss the line by line Hoare logic style proof of the correctness of $Q_{1,j}, j = 1, 3$. For partial correctness, we only need to show that $Q_{1,j}, j = 1, 3$ hold before the execution of the loop and after every execution of the loop body. The total correctness comes from the finite termination property encoded by the invariant **phi-0.76*phim<0**.

6.9   Proving $\phi - 0.76\phi_- < 0$ on the code: Part I

The annotations and code discussed in this subsection are from figure 12 unless stated otherwise.

The annotation of the loop body starts with the insertion of the block of empty code in line 4. Using the skip rule on the invariant $Q_{1,1}$ from figure 11 and $Q_{7,3}$ from figure 9, we get the pre-condition $P_{1,1}$. Apply the theory

$$P \wedge expr1 = expr2 \Rightarrow P[expr1/expr2] \vee P[expr2/expr1] \tag{30}$$

```
1  % (1,1)  requires  phi >0 && phi <=0.1 && phi==trace(X*Z);
2  %        ensures   trace(X*Z)>0 && trace(X*Z)<=0.1;
3  {
4    % empty code
5  }
6  % (2,1)  requires  trace(X*Z)>0 && trace(X*Z)<=0.1;
7  %        ensures   trace(Xm*Zm)>0 && trace(Xm*Zm)<=0.1;
8  {
9    Xm=X;
10   Zm=Z;
11 }
12   pm=p;
13 % (3,1)  requires  n>=1;
14 %        ensures   n*mu==trace(Xm*Zm);
15 {
16   mu=trace(Xm*Zm)/n;
17 }
```

Fig. 12. `while` Loop Body: Part I

to $P_{1,1}$, we get the post-condition $Q_{1,1}$. The formula in (30) is used for several more annotations in the loop body. In the case of going from $P_{1,1}$ to $Q_{1,1}$, we substituted all instances of the expression **phi** in $\phi > 0 \wedge \phi \leq 0.1$ with the expression **trace(X*Z)**. The verification of $\mathcal{H}_{1,1}$ is fairly straightforward in a theorem prover such as PVS.

The next block of code, which is C_2, assigns the value of X and Z to the variables X_- and Z_- respectively. The pre-condition for C_2 is $Q_{1,1}$ because the post-condition of a prior contract is also a pre-condition of the current contract. The post-condition $\mathrm{Tr}\,(X_- Z_-) > 0 \wedge \mathrm{Tr}\,(X_- Z_-) \leq 0.1$ is generated by the application of the backward substitution rule.

Finally, the last contract in figure 12 is for C_3, which assigns the expression **trace(Xm*Zm)/n** to the variable **mu**. The pre-condition $P_{3,1}$ is $Q_{7,1}$ from figure 9 propagated forward using the skip rule. The post-condition $n\mu = \mathrm{Tr}\,(X_- Z_-)$ is generated using the formula

$$n >= 1 \Rightarrow n \neq 0 \wedge n \neq 0 \wedge x = \frac{y}{n} \Rightarrow nx = y. \tag{31}$$

The correctness of equation (31) can be discharged automatically by a theorem prover such as PVS or most SMT solvers.

6.10 Proving $\phi - 0.76\phi_- < 0$ on the code: Part II

The annotations and code described in this subsection are from figure 13 unless stated otherwise.

We look at the first contract, which is for the inserted block of empty code in line 5. The first pre-condition statement **n*mu==trace(Xm*Zm)** is obtained by

```matlab
% (1,1) requires n*mu==trace(Xm*Zm);
%        requires trace(Xm*mats(lsqr(F,zeros(m,1)),n))+trace(
   mats(lsqr(krons((Zm^(0.5))^(-1)*Zm,(Zm^(0.5))',n,m),vecs(
   sigma*mu*eye(n,n)-(Zm^(0.5))*Xm*(Zm^(0.5)))-krons((Zm
   ^(0.5))^(-1),(Zm^(0.5))'*Xm,n,m)*lsqr(F,zeros(m,1))),n)*
   Zm)+trace(Xm*Zm)-sigma*n*mu==0;
%        ensures   trace(Xm*mats(lsqr(F,zeros(m,1)),n))+trace(
   mats(lsqr(krons((Zm^(0.5))^(-1)*Zm,(Zm^(0.5))',n,m),vecs(
   sigma*mu*eye(n,n)-(Zm^(0.5))*Xm*(Zm^(0.5)))-krons((Zm
   ^(0.5))^(-1),(Zm^(0.5))'*Xm,n,m)*lsqr(F,zeros(m,1))),n)*
   Zm)+trace(Xm*Zm)-sigma*trace(Xm*Zm)==0;
{
  % empty
}
% (2,1) requires trace(Xm*mats(lsqr(F,zeros(m,1)),n))+trace(
   mats(lsqr(krons((Zm^(0.5))^(-1)*Zm,(Zm^(0.5))',n,m),vecs(
   sigma*mu*eye(n,n)-(Zm^(0.5))*Xm*(Zm^(0.5)))-krons((Zm
   ^(0.5))^(-1),(Zm^(0.5))'*Xm,n,m)*lsqr(F,zeros(m,1))),n)*
   Zm)+trace(Xm*Zm)-sigma*trace(Xm*Zm)==0;
%        ensures   trace(Xm*mats(lsqr(F,zeros(m,1)),n))+trace(
   mats(lsqr(H,vecs(sigma*mu*eye(n,n)-Zh*Xm*Zh)-G*lsqr(F,
   zeros(m,1))),n)*Zm)+trace(Xm*Zm)-sigma*trace(Xm*Zm)==0;
{
  Zh=Zm^(0.5);
  Zhi=Zh^(-1);
  G=krons(Zhi,Zh'*Xm,n,m);
  H=krons(Zhi*Zm,Zh',n,m);
}
% (3,1) requires trace(Xm*mats(lsqr(F,zeros(m,1)),n))+trace(
   mats(lsqr(H,vecs(sigma*mu*eye(n,n)-Zh*Xm*Zh)-G*lsqr(F,
   zeros(m,1))),n)*Zm)+trace(Xm*Zm)-sigma*trace(Xm*Zm)==0;
%        ensures   trace(Xm*mats(dZm,n))+trace(mats(dXm,n)*Zm)+
   trace(Xm*Zm)-sigma*trace(Xm*Zm)==0;

% (3,2) ensures   dZm==lsqr(F,zeros(m,1));
{
  r=sigma*mu*eye(n,n)-Zh*Xm*Zh;
  dZm=lsqr(F,zeros(m,1));
  dXm=lsqr(H, vecs(r)-G*dZm);
}
% (4,1) requires sigma==0.75;
%        requires trace(Xm*mats(dZm,n))+trace(mats(dXm,n)*Zm)+
   trace(Xm*Zm)-sigma*trace(Xm*Zm)==0;
%        ensures   trace(Xm*mats(dZm,n))+trace(mats(dXm,n)*Zm)+
   trace(Xm*Zm)-0.75*trace(Xm*Zm)==0;
{
  % empty code
}
```

Fig. 13. `while` Loop Body: Part II

applying the skip rule to $Q_{3,1}$ from figure 12. The second pre-condition statement is the instantiation of the following true statement: given ΔX and ΔZ that satisfy the equations

$$\sum_{i=1}^{m} \langle F_i, \Delta Z \rangle = 0, \tag{32}$$

and

$$\begin{aligned}
&\frac{1}{2} \left(Z^{0.5} \left(\Delta X Z \right) Z^{-0.5} + Z^{-0.5} \left(Z \Delta X \right) Z^{0.5} \right) \\
&= \frac{-1}{2} \left(Z^{0.5} \left(X \Delta Z \right) Z^{-0.5} + Z^{-0.5} \left(\Delta Z X \right) Z^{0.5} \right) \\
&+ \sigma \mu I - Z^{0.5} X Z^{0.5},
\end{aligned} \tag{33}$$

then $\forall X, Z \in \mathbb{R}^{n \times n}$,

$$\mathrm{Tr} \left(X \Delta Z \right) + \mathrm{Tr} \left(\Delta X Z \right) + \mathrm{Tr} \left(X Z \right) - \sigma n \mu = 0. \tag{34}$$

The correctness of (34) can be seen by noting that

$$\begin{aligned}
\mathrm{Tr} \left(\Delta X Z \right) &= \mathrm{Tr} \left(\frac{1}{2} \left(Z^{0.5} \left(\Delta X Z \right) Z^{-0.5} + Z^{-0.5} \left(Z \Delta X \right) Z^{0.5} \right) \right) \\
&= \mathrm{Tr} \left(\frac{-1}{2} \left(Z^{0.5} \left(X \Delta Z \right) Z^{-0.5} + Z^{-0.5} \left(\Delta Z X \right) Z^{0.5} \right) \right) \\
&+ \mathrm{Tr} \left(\sigma \mu I_{n \times n} \right) - \mathrm{Tr} \left(Z^{0.5} X Z^{0.5} \right) \\
&= - \mathrm{Tr} \left(X \Delta Z \right) + \sigma n \mu - \mathrm{Tr} \left(X Z \right),
\end{aligned} \tag{35}$$

hence

$$\begin{aligned}
&\mathrm{Tr} \left(X \Delta Z \right) + \mathrm{Tr} \left(\Delta X Z \right) + \mathrm{Tr} \left(X Z \right) - \sigma n \mu \\
&= \mathrm{Tr} \left(X \Delta Z \right) - \mathrm{Tr} \left(X \Delta Z \right) + \sigma n \mu - \mathrm{Tr} \left(X Z \right) + \mathrm{Tr} \left(X Z \right) - \sigma n \mu \\
&= 0
\end{aligned} \tag{36}$$

Now we look at the pre-condition statement from line 2. First, note that the subexpression **mats(lsqr(F,zeros(m,1)))** from line 2 returns a ΔZ that satisfies equation (32). Second, note that the subexpression **mats(lsqr(krons((..., vecs(sigma...*Zm)** from line 2 returns a ΔX that satisfies, with X replaced by X_- and Z replaced by Z_-, the equation in (33). Finally, one can see that the pre-condition statement is generated by instantiating X and Z of equation(34) with X_- and Z_-.

The post-condition for $\mathcal{H}_{1,1}$ is generated using the formula from (30) i.e. by replacing all instances of the expression **n*mu** in the pre-condition statement from line 2 with the expression **trace(Xm*Zm)**.

The next contract consists of the pre-condition $P_{2,1}$, which is a duplication of $Q_{1,1}$, and the post-condition $Q_{2,1}$. The post-condition $Q_{2,1}$ is generated by four successive application of the backward substitution rule, one for each line of code in C_2. This process results in four successive Hoare triples, which are merged into $\mathcal{H}_{2,1}$ by the application of the composition rule from equation (3).

Hoare triple $\mathcal{H}_{3,1}$ is constructed in a similar fashion as $\mathcal{H}_{2,1}$. For C_3, we insert an additional contract ensuring that the variable **dZm** is equal to the expression **lsqr(F,zeros(m,1))**. This post-condition is correct because of the code in line 21 and will be used later in the annotation of the loop body.

The last contract from figure 13, which forms $\mathcal{H}_{4,1}$ has two pre-condition statements. The first pre-condition is $Q_{6,1}$ from figure 10. It is valid by repeated backward substitution (rule 5 in table 1). The second pre-condition statement is simply the post-condition of the prior contract. The post-condition $Q_{4,1}$ is generated using the formula in (30). In this case, the formula results in all instances of the expression **sigma** in $Q_{3,1}$ being replaced by the expression **0.75**.

6.11 Proving $\phi - 0.76\phi_- < 0$ on the code: Part III

The annotations and code described in this subsection are from figure 13 unless stated otherwise.

In figure 14, the annotations generated are used to prove the formula

$$Q_{4,1} \wedge Q_{5,1} \Rightarrow \mathrm{Tr}\left((X_- + \Delta X_-)(Z_- + \Delta Z_-)\right) - 0.75\,\mathrm{Tr}\left(X_- Z_-\right) = 0, \quad (37)$$

in which $Q_{4,1}$ and $Q_{3,2}$ are from figure 14. Each of the Hoare triple represents an individual step in the proof of (37).

We now look at the first contract from figure 14. The pre-condition $P_{1,1}$ is generated using the skip rule on $Q_{3,1}$ from figure 13. The post-condition is generated by noting that **F*lsqr(F,zeros(m,1))** is equal to $\{0\}^m$, which can be verified by checking the correctness of the default contract on the function **lsqr**.

In the next contract, the post-condition is generated by applying the transpose operator to the first pre-condition statement, which is precisely the post-condition of $\mathcal{H}_{1,1}$, and then followed by the application of the formula in (30) using the second pre-condition statement in line 7. The second pre-condition statement is obtained from the application of the skip rule on $Q_{6,1}$ from figure 9.

The third contract is a proof step that is correct because **mats** and **vecs** are unitary transformations. Although omitted from the discussion before, the properties of unitary transformation can be part of the default contract on both of these functions.

In the fourth contract, the first pre-condition statement is simply the post-condition of the prior contract. The second pre-condition statement is inserted by hand. Note that its correctness stems from the correctness of the **lsqr**. If the default contract on **lsqr** is satisfied, then

$$Ft\,\mathrm{lsqr}\left(Ft, -dXm\right) = -dXm \qquad (38)$$

is true for all Ft and dXm. By the application of (30), we get the post-condition in line 19.

In the last contract, $Q_{4,1}$ and $Q_{4,1}$ from figure 14 are the pre-condition statements. The post-condition is generated using a two step heuristics. First the pre-conditions are summed, and then followed by an algebraic transformation into the form in $Q_{5,1}$. The generation of $Q_{5,1}$ required far more ad hoc heuristics

```
1  % (1,1) requires   dZm==lsqr(F,zeros(m,1));
2  %        ensures    lsqr(Ft,-dXm)'*F*dZm==0;
3  {
4     % empty code
5  }
6  % (2,1) requires  lsqr(Ft,-dXm)*F*dZm==0;
7  %        requires  transpose(F)==Ft;
8  %        ensures   dot(Ft*lsqr(Ft,-dXm),dZm)==0;
9  {
10    % empty code
11 }
12 % (3,1) requires  dot(Ft*lsqr(Ft,-dXm),dZm)==0;
13 %        ensures   trace(mats(Ft*lsqr(Ft,-dXm),n)*mats(dZm,n))
       ==0;
14 {
15    % empty code
16 }
17 % (4,1) requires  trace(mats(Ft*lsqr(Ft,-dXm),n)*mats(dZm,n))
       ==0;
18 %        requires  Ft*lsqr(Ft,-dXm)==-dXm
19 %        ensures   trace(mats(dXm,n)*mats(dZm,n))==0
20 {
21    % empty code
22 }
23 % (5,1) requires  trace(mats(dXm,n)*mats(dZm,n))==0;
24 %        requires  trace(Xm*mats(dZm,n))+trace(mats(dXm,n)*Zm)+
       trace(Xm*Zm)-0.75*trace(Xm*Zm)==0;
25 %        ensures   trace((Xm+mats(dXm,n))*(Zm+mats(dZm,n)))
       -0.75*trace(Xm*Zm)==0;
26 {
27    % empty code
28 }
```

Fig. 14. `while` Loop Body: Part III

than the other contracts discussed so far. However, note that we can split the last contract into even smaller steps such as one for the summing followed by one using the algebraic transformation.

6.12 Proving $\phi - 0.76\phi_- < 0$ on the code: Part IV

The annotations and code described in this subsection are from figure 15 unless stated otherwise.

The block of code in line 2 and 3 has no contract. We skip ahead to the block of code in lines 8 and 9. For C_1, the pre-condition is a duplication of $Q_{5,1}$ from figure 14 and the post-condition $Q_{1,1}$ is generated by using the backward substitution rule twice i.e. once for each line of code in the block.

Next, we jump forward to last block of code in the loop body, which is C_5. For C_5, using the while loop axiom, we insert $Q_{1,1}$ and $Q_{1,2}$ from figure 11 as post-conditions. Applying the backward substitution rule on $Q_{5,1}$ and $Q_{5,2}$ results in $P_{5,1}$ and $P_{5,2}$ respectively. Additionally, we also have $Q_{5,3}$ in order to ensure consistency with the clause **phi==trace(X*Z)** in $P_{1,1}$ from figure 12.

We now jump backward to C_4, and we have the post-conditions $Q_{4,1}$ and $Q_{4,2}$ that are precisely $P_{5,1}$ and $P_{5,2}$. The post-condition $Q_{4,1}$ is not affected by the execution of C_4, so we cannot apply the Hoare logic rules. Instead we propagate the post-conditions $Q_{1,1}$ and $Q_{2,1}$ from figure 12 forward and then combined them conjunctively to form $P_{4,1}$. With the inserted $P_{4,1}$, verifying $\mathcal{H}_{4,1}$ is the same as checking if

$$P_{4,1} \Rightarrow Q_{4,1} \tag{39}$$

holds. Equation (39) can be discharged by a theorem prover for example in three steps: first we apply the following result from theory:

$$y - cx = 0 \Rightarrow y = cx \tag{40}$$

on $Q_{1,1}$ followed by the application of

$$c > 0 \wedge c < 1 \wedge x <= 0.1 \wedge x > 0 \Rightarrow cx <= 0.1 \wedge cx > 0, \tag{41}$$

on $Q_{2,1}$ from figure 12. Lastly, by applying the formula in (30) on the conclusions of (40) and (41), we get $y <= 0.1 \wedge y > 0$ which leads to $Q_{4,1}$.

The backward substitution rule is applied on $Q_{4,2}$ to get $P_{4,2}$. Looking back at C_1, we can see the post-condition $Q_{1,1}$ is not equivalent to $P_{4,2}$. Since there is no executable code between $Q_{1,1}$ and $P_{4,2}$, we would have a contradiction unless $Q_{1,1} \wedge \bigwedge_i Q_i \Rightarrow P_{4,2}$, in which Q_i belongs to the set of all post-conditions of C_1. We resolve this contradiction by inserting the Hoare triples $\mathcal{H}_{i,1}, i = 2, 3$ as a proof of

$$Q_{1,1} \wedge \mathrm{Tr}\left(X_- Z_-\right) > 0 \Rightarrow P_{4,2}. \tag{42}$$

The generation of $\mathcal{H}_{i,1}, i = 2, 3$, used ad hoc heuristics as in the annotations of figure 14. As done for equation (37), we first decided to split the proof of the formula in (42) into two steps. In the first step i.e. $\mathcal{H}_{2,1}$, the pre-condition

```
1  {
2     dpm=lsqr(Ft,-dXm);
3     p=pm+dpm
4  }
5  % (1,1) require trace((Xm+mats(dXm,n))*(Zm+mats(dZm,n)))
        -0.75*trace(Xm*Zm)==0;
6  %        ensures trace(X*Z)-0.75*trace(Xm*Zm)==0
7  {
8     X=Xm+mats(dXm,n);
9     Z=Zm+mats(dZm,n);
10 }
11 % (2,1) requires trace(Xm*Zm)>0;
12 %        ensures   0.01*trace(Xm*Zm)>0;
13 {
14    % empty code
15 }
16 % (3,1) requires trace(X*Z)-0.75*trace(Xm*Zm)==0
17 %        requires 0.01*trace(Xm*Zm)>0;
18 %        ensures   trace(X*Z)-0.76*trace(Xm*Zm)<0;
19 {
20    % empty code
21 }
22 % (4,1) requires trace(X*Z)-0.75*trace(Xm*Zm)==0
23 %        requires trace(Xm*Zm)>0 && trace(Xm*Zm)<=0.1;
24 %        ensures   trace(X*Z)>0 && trace(X*Z)<=0.1;
25
26 % (4,2) requires trace(X*Z)-0.76*trace(Xm*Zm)<0
27 %        ensures   trace(X*Z)-0.76*phim<0;
28 {
29       phim=trace(Xm*Zm);
30 }
31 % (5,1) requires trace(X*Z)>0 && trace(X*Z)<=0.1;
32 %        ensures   phi>0 && phi<=0.1;
33
34 % (5,2) requires trace(X*Z)-0.76*phim<0;
35 %        ensures   phi-0.76*phim<0;
36
37 % (5,3) ensures   phi==trace(X*Z);
38 {
39       phi=trace(X*Z);
40       mu=trace(X*Z);
41 }
```

Fig. 15. `while` Loop Body: Part IV

is **trace(Xm*Zm)>0**, which is true by conjunctive simplification of $Q_{2,1}$ from figure 12, and a post-condition $Q_{2,1}$ constructed using the following result from theory:

$$c > 0 \land x > 0 \Rightarrow cx > 0, \tag{43}$$

with a value of 0.01 being chosen purposefully for the positive constant c. In $\mathcal{H}_{3,1}$, we insert $Q_{1,1} \land Q_{2,1}$ as the pre-condition and $P_{4,2}$ as the post-condition. The correctness of $\mathcal{H}_{3,1}$ can be verified by applying the following result:

$$y = 0 \land cx > 0 \Rightarrow y - cx < 0. \tag{44}$$

By verifying $P_{4,2}$, we have now completed the discussion on the annotations to assist the mechanical proof of the correctness of the invariants **phi>0 && phi<=0.1** and **phi-0.76*phim<0**. In the next subsection, we discuss, albeit in a far less mechanical way, on the possible annotations to prove the correctness of the invariant **X>0 && Z>0**.

6.13   The loop invariant $X \succ 0 \land Z \succ 0$

In the following subsections, we describe some of the key annotations used to show that $Z \succ 0 \land X \succ 0$ hold as an invariant of the `while` loop. For the sake of brevity, some of the intermediate proof annotations are omitted.

First we use the central path property of the running example i.e. X and Z are assigned initial values such that

$$\|XZ - \mu I\|_F \leq 0.3105\mu \tag{45}$$

as an inductive invariant for the `while` loop.

Remark 2. The central path property in (45) is a constraint that guarantees that X and Z are initialized to values contained within a small neighborhood of the central path. This constraint is well-known in the interior point literature [7], and it is used to prove linear convergence of the duality gap in the general case. For some input optimization problems, a large deviation from the central path during a Newton step can result in an increase in the duality gap. Efficient methods exist in the interior point method literature (see [14]) to guarantee an initialization within a small neighborhood of the central path.

We now insert (45) and $X \succ 0 \land Z \succ 0$ as the invariants of the `while` loop.

6.14   Proving $X \succ 0 \land Z \succ 0$ on the code: Part I

All annotations discussed in this subsection are from figure 17 unless explicitly stated otherwise. The first pre-condition statement in $\mathcal{H}_{1,1}$ is an invariant from $Q_{1,1}$ of figure 16. The second pre-condition statement is obtained from $Q_{10,2}$ from figure 10. We apply equation (30) to get the post-condition

$$\left\| XZ - \frac{\text{Tr}\,(XZ)}{n}I \right\|_F \leq 0.3105 \frac{\text{Tr}\,(XZ)}{n}. \tag{46}$$

```
1  % (1,1) requires norm(X*Z−mu*eye(n,n))<=0.3105*mu && X>0 && Z
       >0;
2  %       ensures  norm(X*Z−mu*eye(n,n))<=0.3105*mu && X>0 && Z
       >0;
3  {
4    while (phi>epsilon) do
5        .
6        .
7        .
8    end
9  }
```

Fig. 16. Positive-Definiteness of X and Z as Invariants

```
1  % (1,1) requires norm(X*Z−mu*eye(n,n))<=0.3105*mu;
2  %       requires mu==trace(X*Z)/n;
3  %       ensures  norm(X*Z−trace(X*Z)/n*eye(n,n))<=0.3105*
       trace(X*Z)/n;
4  {
5    % empty
6  }
7  % (2,1) requires X>0 && Z>0;
8  %       ensures  Xm>0 && Zm>0;
9
10 % (2,2) requires norm(X*Z−trace(X*Z)/n*eye(n,n))<=0.3105*
       trace(X*Z)/n;
11 %       ensures  norm(Xm*Zm−trace(Xm*Zm)/n*eye(n,n))<=0.3105*
       trace(Xm*Zm)/n;
12 {
13   Xm=X
14   Zm=Z;
15   pm=p;
16 }
17 % (3,1) requires norm(Xm*Zm−trace(Xm*Zm)/n*eye(n,n))<=0.3105*
       trace(Xm*Zm)/n;
18 %       ensures  norm(Xm*Zm−mu*eye(n,n))<=0.3105*mu;
19 {
20   mu=trace(Xm*Zm)/n;
21 }
```

Fig. 17. Invariant $X \succ 0 \land Z \succ 0$ Part I

Next, for C_2, we insert two contracts. The first contract has the pre-condition $X \succ 0 \wedge Z \succ 0$, which is obtained from $Q_{1,1}$ of figure 16. The post-condition $Q_{2,1}$ is generated by the repeated applications of the backward substitution rule. For the second contract, the pre-condition is simply $Q_{1,1}$ and the post-condition is also generated by the repeated application of the backward substitution rule. We now move to C_3 which only has one contract. The pre-condition in $\mathcal{H}_{3,1}$ is simply $Q_{2,2}$ and the post-condition $Q_{3,1}$ is generated by another application of the backward substitution rule.

6.15   Proving $X \succ 0 \wedge Z \succ 0$ on the code: Part II

The annotations described in this subsection are from figure 18 unless explicitly stated otherwise. The post-condition $Q_{2,2}$ of figure 16 implies two properties that are vital to proving the correctness of $Z \succ 0$ and $X \succ 0$. The insertion of those properties result in the next two Hoare triples. Triple $\mathcal{H}_{1,1}$ is obtained from the formula

$$P_{1,1} \Rightarrow \|Z_-^{-0.5} \Delta Z Z_-^{-0.5}\|_F \leq 0.7, \tag{47}$$

with ΔZ that satisfies equation (32) and $P_{1,1}$ being $Q_{2,2}$ from figure 16. The value 0.7 in (47) is obtained from an over-approximation of the expression $\dfrac{\sqrt{n\left(1-\sigma\right)^2 + 0.3105^2}}{1 - 0.3105}$. The proof for this result is skipped here and can be found in [16]. We also have $\mathcal{H}_{1,2}$, which is obtained from the formula

$$P_{2,2} \Rightarrow \|Z_-^{-0.5} \Delta X \Delta Z Z_-^{0.5}\|_F \leq 0.3105\sigma\mu, \tag{48}$$

with ΔX and ΔZ that satisfy (32) and (33) and $P_{1,2}$ also being $Q_{2,2}$ from figure 16.

Next, the block of code C_2 computes the Newton directions ΔX_- and ΔZ_-. We duplicate $Q_{1,1}$ and $Q_{1,2}$ to form $P_{2,1}$ and $P_{2,2}$ respectively. Using the backward substitution rule on $Q_{2,1}$, we get the post-condition

$$\|Z_-^{-0.5} \Delta Z_- Z_-^{-0.5}\|_F \leq 0.7. \tag{49}$$

Applying the backward substitution rule again on $P_{2,2}$, we get the post-condition

$$\|Z_-^{-0.5} \Delta X_- \Delta Z_- Z_-^{0.5}\|_F \leq 0.3105\sigma\mu, \tag{50}$$

We move to the next Hoare triple which is $\mathcal{H}_{3,1}$. The pre-condition statement **mats(H*dXm)+mats(G*dZm)==sigma*mu*eye(n,n)-Zh*Xm*Zh** is correct by noting that **mats(H*dXm)** is equal to **mats(vecs(r)-G*dXm)** because of the assignment in line 21, and **mats(vecs(r)-G*dXm)** reduces to **r-mats(G*dXm)**. An equivalent formula to **mats(H*dXm)+mats(G*dZm)==sigma*mu*eye(n,n)-Zh*Xm*Zh** is

$$
\begin{aligned}
0.5\left(Z_-^{-0.5}\left(\Delta Z_- X_- + Z_- \Delta X_-\right)Z_-^{0.5} + Z_-^{0.5}\left(X_- \Delta Z_- + \Delta X_- Z_-\right)Z_-^{-0.5}\right) \\
= \sigma\mu I - Z_-^{0.5} X_- Z_-^{0.5}.
\end{aligned} \tag{51}
$$

```matlab
% (1,1) requires norm(Xm*Zm-mu*eye(n,n))<=0.3105*mu;
%        ensures   norm((Zm^(0.5))^(-1)*mats(lsqr(F,zeros(m,1))
       ,n)*Zm^(0.5))<=0.7;

% (1,2) requires norm(Xm*Zm-mu*eye(n,n))<=0.3105*mu;
%        requires norm(Zm^(0.5))^(-1)*mats(lsqr(krons((Zm
       ^(0.5))^(-1)*Zm,(Zm^(0.5))',n,m),vecs(sigma*mu*eye(n,n)-(
       Zm^(0.5))*Xm*(Zm^(0.5)))-krons((Zm^(0.5))^(-1),(Zm^(0.5))
       '*Xm,n,m)*lsqr(F,zeros(m,1))),n)*mats(lsqr(F,zeros(m,1)),
       n)*Zm^(0.5))<=0.3105*sigma*mu;
{
  % empty
}
% (2,1) requires norm((Zm^(0.5))^(-1)*mats(lsqr(F,zeros(m,1))
       ,n)*Zm^(0.5))<=0.7;
%        ensures   norm(Zhi*mats(dZm,n)*Zhi)<=0.7;

% (2,2) requires norm(Zm^(0.5))^(-1)*mats(lsqr(krons((Zm
       ^(0.5))^(-1)*Zm,(Zm^(0.5))',n,m),vecs(sigma*mu*eye(n,n)-(
       Zm^(0.5))*Xm*(Zm^(0.5)))-krons((Zm^(0.5))^(-1),(Zm^(0.5))
       '*Xm,n,m)*lsqr(F,zeros(m,1))),n)*mats(lsqr(F,zeros(m,1)),
       n)*Zm^(0.5))<=0.3105*sigma*mu;
%        ensures   norm(Zhi*mats(dXm,n)*mats(dXm,n)*Zhi)
       <=0.3105*sigma*mu;
{
  Zh=Zm^(0.5);
  Zhi=Zh^(-1);
  G=krons(Zhi,Zh'*Xm,n,m);
  H=krons(Zhi*Zm,Zh',n,m);
  r=sigma*mu*eye(n,n)-Zh*Xm*Zh;
  dZm=lsqr(F,zeros(m,1));
  dXm=lsqr(H, vecs(r)-G*dZm);
}
% (3,1) requires mats(H*dXm)+mats(G*dZm)==sigma*mu*eye(n,n)-
       Zh*Xm*Zh;
%        requires norm(Zhi*mats(dXm,n)*mats(dXm,n)*Zhi)
       <=0.3105*sigma*mu;
%        ensures   0.5*norm(Zhi*((Zm+mats(dZm,n)*(Xm+mats(dXm,n
       )-sigma*mu*eye(n,n))*Zh+Zh'*((Xm+mats(dXm,n)*(Zm+mats(dZm
       ,n)-sigma*mu*eye(n,n))*Zhi)<=0.3105*sigma*mu;
{
  % empty
}
  dpm=lsqr(Ft,-dXm);
  p=pm+dpm;
% (4,1) requires norm(Zhi*mats(dZm,n)*Zhi)<=0.7;
%        ensures   Zm+mats(dZm,n)>0;
{
  % empty
}
```

Fig. 18. Annotations for the Invariant $X \succ 0 \wedge Z \succ 0$: Part II

The second pre-condition statement of $\mathcal{H}_{3,1}$ is $Q_{2,2}$ unchanged. Given the pre-condition statements, we can deductively arrive at the post-condition $Q_{3,1}$, which is

$$0.5 \| Z_-^{-0.5} \left(\left(Z_- + \Delta Z_- \right) \left(X_- + \Delta X_- \right) - \sigma \mu I \right) Z_-^{0.5} +$$
$$Z_-^{0.5} \left(\left(X_- + \Delta X_- \right) \left(Z_- + \Delta Z_- \right) - \sigma \mu I \right) Z_-^{-0.5} \|_F \tag{52}$$
$$\leq 0.3105 \sigma \mu.$$

Remark 3. Note that in (52), we use the expression $Z_-^{0.5}$ to represent the variable **Zh** and the expression $Z_-^{-0.5}$ to represent **Zhi**. This can be done because of the assignments in lines 15 and 16. For the sake of brevity, we do not annotate the steps that lead to (52) as Hoare triples on the code, nonetheless here we give a sketch of the constructive proof. First note that

$$0.5 \left(Z_-^{-0.5} \left(\Delta Z_- X_- + Z_- \Delta X_- \right) Z_-^{0.5} + Z_-^{0.5} \left(X_- \Delta Z_- + \Delta X_- Z_- \right) Z_-^{-0.5} + \right.$$
$$Z_-^{0.5} X_- Z_-^{0.5} - \sigma \mu I + Z_-^{-0.5} \left(\Delta Z_- \Delta X_- \right) Z_-^{0.5} + Z_-^{0.5} \left(\Delta X_- \Delta Z_- \right) Z_-^{-0.5} \right)$$
$$= 0.5 \left(Z_-^{-0.5} \left(\left(Z_- + \Delta Z_- \right) \left(X_- + \Delta X_- \right) - \sigma \mu I \right) Z_-^{0.5} + \right.$$
$$\left. Z_-^{0.5} \left(\left(X_- + \Delta X_- \right) \left(Z_- + \Delta Z_- \right) - \sigma \mu I \right) Z_-^{-0.5} \right). \tag{53}$$

holds because the left hand side is an algebraic expansion of the right hand side. Second, apply the formula in (30) on the conjunction of (53) and (51), (53) is reduced to

$$Z_-^{-0.5} \left(\Delta Z_- \Delta X_- \right) Z_-^{0.5} + Z_-^{0.5} \left(\Delta X_- \Delta Z_- \right) Z_-^{-0.5}$$
$$= 0.5 \left(Z_-^{-0.5} \left(\left(Z_- + \Delta Z_- \right) \left(X_- + \Delta X_- \right) - \sigma \mu I \right) Z_-^{0.5} + \tag{54} \right.$$
$$\left. Z_-^{0.5} \left(\left(X_- + \Delta X_- \right) \left(Z_- + \Delta Z_- \right) - \sigma \mu I \right) Z_-^{-0.5} \right).$$

Finally, apply the transitive property of the comparison operators to $Q_{2,2}$ and the Frobenius norm of (54), we get the post-condition in (52).

Now we move ahead to the last Hoare triple of figure 18, which is $\mathcal{H}_{4,1}$. This Hoare triple is generated using the formula

$$Q_{2,1} \Rightarrow Z_- + \Delta Z_- \succ 0, \tag{55}$$

Remark 4. The formula in (55) is correct and to see that, note

$$Q_{2,1} \Rightarrow \| Z_-^{-0.5} \Delta Z_- Z_-^{-0.5} \|_F < 1 \Rightarrow I + Z_-^{-0.5} \Delta Z_- Z_-^{-0.5} \succ 0. \tag{56}$$

and that

$$I + Z_-^{-0.5} \Delta Z_- Z_-^{-0.5} = Z_-^{-0.5} \left(Z_- + \Delta Z_- \right) Z_-^{-0.5}. \tag{57}$$

```
1  % (1,1) requires 0.5*norm(Zhi*((Zm+mats(dZm,n)*(Xm+mats(dXm,n
       )-sigma*mu*eye(n,n))*Zh+Zh'*((Xm+mats(dXm,n)*(Zm+mats(dZm
       ,n)-sigma*mu*eye(n,n))*Zhi')<=0.3105*sigma*mu;
2  %        ensures   0.5*norm(Zhi*(X*Z-sigma*mu*eye(n,n))*Zh+Zh
       '*(Z*X-sigma*mu*eye(n,n))*Zhi')<=0.3105*sigma*mu;
3
4  % (1,2) requires Zm+mats(dZm,n)>0;
5  %        ensures   Z>0;
6  {
7    X=Xm+mats(dXm,n);
8    Z=Zm+mats(dZm,n);
9  }
10 % (2,1) requires Z>0 && Zm>0;
11 %        ensures   norm(Z^(0.5)*X*Z^(0.5)-sigma*mu*eye(n,n))
       <=0.5*norm(Zhi*(X*Z-sigma*mu*I)*Zh+Zh'*(Z*X-sigma*mu*I)*
       Zhi');
12 {
13   % empty
14 }
15   phim=trace(Xm*Zm);
16   phi=trace(X*Z);
17 % (3,1) requires trace(X*Z)-sigma*trace(Xm*Zm)==0;
18 %        requires trace(Xm*Zm)==n*mu;
19 %        ensures   trace(X*Z)-sigma*n*mu==0;
20 {
21   % empty
22 }
23 % (4,1) requires trace(X*Z)-sigma*n*mu==0;
24 %        requires norm(Z^(0.5)*X*Z^(0.5)-sigma*mu*eye(n,n))
       <=0.5*norm(Zhi*(X*Z-sigma*mu*I)*Zh+Zh'*(Z*X-sigma*mu*eye(
       n,n))*Zhi');
25 %        requires 0.5*norm(Zhi*(X*Z-sigma*mu*eye(n,n))*Zh+Zh
       '*(Z*X-sigma*mu*eye(n,n))*Zhi')<=0.3105*sigma*mu;
26 %        ensures   norm(Z^(0.5)*X*Z^(0.5)-trace(X*Z)/n*eye(n,n)
       )<=0.3105*trace(X*Z)/n;
27 {
28   % empty
29 }
30 % (5,1) requires norm(Z^(0.5)*X*Z^(0.5)-trace(X*Z)/n*eye(n,n)
       )<=0.3105*trace(X*Z)/n;
31 %        ensures   norm(Z^(0.5)*X*Z^(0.5)-mu*eye(n,n))<=0.3105*
       mu;
32 {
33   mu=trace(X*Z)/n;
34 }
35 % (6,1) requires norm(Z^(0.5)*X*Z^(0.5)-mu*eye(n,n))<=0.3105*
       mu;
36 %        ensures   norm(XZ-mu*eye(n,n))<=0.3105*mu;
37
38 % (6,2) requires Z>0;
39 %        requires norm(Z^(0.5)*X*Z^(0.5)-mu*eye(n,n))<=0.3105*
       mu;
40 %        ensures   X>0;
41 {
42   % empty
43 }
```

Fig. 19. Annotations for the Invariant $X \succ 0 \land Z \succ 0$: Part III

6.16   Proving $X \succ 0 \wedge Z \succ 0$ on the code: Part III

Now we move forward to the last part of the loop body. The annotations described in this subsection are from figure 19 unless explicitly stated otherwise. We insert two contracts for block of code C_1. The pre-conditions of the two contracts are respectively $Q_{3,1}$ and $Q_{4,1}$ of figure 18 propagated forward using the skip rule. By the application of the backward substitution, we obtain the post-conditions $P_{1,1}$ and $P_{1,2}$. The post-condition $Z \succ 0$ completes the proof for a part of the invariant $Q_{1,1}$ from figure 16. We still have to show that $X \succ 0$ and $\|XZ - \mu I\|_F \leq 0.3105\mu$ hold.

The next Hoare triple, which is $\mathcal{H}_{2,1}$, is generated using the formula, $\forall Z \succ 0, Z_- \succ 0$,

$$\|Z^{0.5}XZ^{0.5} - \sigma\mu I\|_F \leq \frac{1}{2}\|Z_-^{-0.5}\left(ZX - \sigma\mu I\right)Z_-^{0.5} + \tag{58}$$
$$Z_-^{0.5}\left(XZ - \sigma\mu I\right)Z_-^{-0.5}\|_F$$

holds. The post-condition of $\mathcal{H}_{2,1}$ is precisely the statement in (58) with **Zh** being $Z_-^{0.5}$ and **Zhi** being $Z_-^{-0.5}$.

Now we forward again to $\mathcal{H}_{3,1}$. Recall the post-condition $Q_{1,1}$ from figure 15, which is **trace(X*Z) - 0.75*trace(Xm*Zm)==0**. Also recall that the post-condition $Q_{1,1}$ from figure 10, which is **sigma==0.75**. We can apply the rule in (30) to **trace(X*Z) - 0.75*trace(Xm*Zm)==0 && sigma==0.75**, and obtain the first pre-condition statement **trace(X*Z)-sigma*trace(Xm*Zm)==0**. By propagating forward $Q_{1,1}$ from figure 12, we get the second pre-condition statement. Apply again the rule from (30) on $P_{3,1}$, we get the post-condition **trace(X*Z)-sigma*n*mu==0**.

In the next Hoare triple, which is $\mathcal{H}_{4,1}$, the pre-condition is formed by combining the post-conditions $Q_{i,1}, i = 1, 2, 3$. The post-condition $Q_{4,1}$ is generated by first noting that due to the transitivity of $<=$, the formula

$$Q_{2,1} \wedge Q_{1,1} \Rightarrow \|Z^{0.5}XZ^{0.5} - \sigma\mu I\|_F \leq 0.3105\mu \tag{59}$$

holds. Second, applying the rule in (30) to the conjunction of (59) and $Q_{3,1}$ precisely yields post-condition $Q_{4,1}$.

Finally we move forward to the block of code in line 33, in which the variable **mu** is updated with the expression $\dfrac{\text{Tr}\left(XZ\right)}{n}$. Apply the backward substitution rule on the pre-condition, which is simply $Q_{4,1}$ unaltered, we get a post-condition of

$$\|Z^{0.5}XZ^{0.5} - \mu I\|_F \leq 0.3105\mu. \tag{60}$$

The post-condition in (60) i.e. $Q_{5,1}$ is used to generate the last two Hoare triples of figure 19. The Hoare triple $\mathcal{H}_{6,1}$ is a result of the formula

$$\|Z^{0.5}XZ^{0.5} - \mu I\|_F \leq 0.3105\mu \Rightarrow \|XZ - \mu I\|_F \leq 0.3105\mu, \tag{61}$$

which we know is correct beforehand. For $\mathcal{H}_{6,2}$, we have the pre-condition $Z \succ 0 \wedge Q_{5,1}$ which implies $X \succ 0$, and thus completes the annotation process.

7 Autocoding of Convex Optimization Algorithms and Automatic Verification

In this section, we describe some general ideas towards the credible autocoding of convex optimization algorithms. In credible autocoding, the optimization semantics, such as those described in the manual process in the previous section, is generated automatically. The variations of the interior point method discussed in this paper is relatively simple with changes in one of the parameters such as the symmetrizing scaling matrix T, the step size α which is defaulted to 1 in the algorithm description, the duality gap reduction parameter σ, etc. The more complex interior point implementations such as those with heuristics in the predictor, can also be specified using a few additional parameters. Since many optimization programs differs from each other only in a finite set of parameter, and the input problem, we proppose an autocoding approach based on a set of standard parameterized mappings of interior-point algorithms to the output code. The same approach applies to the generation of the optimization semantics. We can construct a set of mappings from the type of interior point algorithm to a set of standard paramaterized optimizaiton semantics i.e. such as the ones described in the previous section. The autocoding process, roughly speaking, becomes a procedure to choose the mapping, followed by insertion of the pre-defined semantics into the generated code, and then subsituting in the values of the parameters and the input problem.

As we discussed briefly before, for domain-specific properties such as $Z \succ 0 \wedge X \succ 0$, the annotations are obtained usually from the steps of a complex proof that cannot always be discharge by a generic automated proof-checkers. The automatic verification of the annotated output most likely would require a theorem prover based tool. The theorem prover need to be adapted to handle the theories and formulas described in the previous section. The current theories available in the theorem prover PVS are not equipped to handle a lot of annotations discussed, especially in figures 18 and 19. The feasible approach would be to use the same method that we took for control system invariants i.e. to construct a few key theorems, which can be repeatedly applied to many autocoded optimization programs as they only differs from each other in the input data, step size or the Newton direction. Key theorems including properties on Frobenius norms, matrix operator theory, etc.

8 Future Work

In this paper, we introduce an approach to communicate high-level functional properties of convex optimization algorithms and their proofs down to the code level. This work open several research directions.

On a more theoretical front, one can explore the possibile existence of linear approximations to the potential function used in the construction of the invariant. Having linear approximations would possibly enable the construction of efficient automatic decision procedures to verify the annotations at code level.

On a more practical front, Matlab remains a high-level language that is never used as an embedded language. There is a need to carry out the work done in this paper on a lower-level embedded language such as C. In addition, the manual process outlined in this paper becomes all the more valuable in a safety-critical context if it is automated. Hence the need for the construction of a prototype tool capable of autocoding a variety of convex optimization programs along with their proofs down to the code level.

It is also crucial for a certification authority to be able to independently and automatically verify the validity of the annotations presented in this paper. To the authors' knowledge, no existing formal analysis tool supports the kind of properties used here. This entails, for example, the extension of existing proof assistants with high-level theorems from Optimization.

Finally, we also need to be able to reason about the invariants introduced in this paper in the presence of the numerical errors due to floating-point computations.

9   Conclusions

This paper proposes the transformation of high-level functional properties of interior point method algorithms down to implementation level for certification purpose. The approach is taken from a previous work done for control systems. We give an example of a primal-dual interior point algorithms and its convergence property. We show that the high-level proofs can be used as annotations for the verification of an online optimization program.

10   Acknowledgements

References

1. F. Alizadeh, J.-P. A. Haeberly, and M. L. Overton. Primal-dual interior-point methods for semidefinite programming: Convergence rates, stability and numerical results. *SIAM Journal on Optimization*, 5:13–51, 1994.
2. P. Baudin, J.-C. Filliâtre, C. Marché, B. Monate, Y. Moy, and V. Prevosto. *ACSL: ANSI/ISO C Specification Language*, 2008. http://frama-c.cea.fr/acsl.html.
3. S. Boyd and L. Vandenberghe. *Convex Optimization*. Cambridge University Press, New York, NY, USA, 2004.
4. P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-c: A software analysis perspective. In *Proceedings of the 10th International Conference on Software Engineering and Formal Methods*, SEFM'12, pages 233–247, Berlin, Heidelberg, 2012. Springer-Verlag.

5. L. de Moura and N. Bjørner. Z3: An efficient smt solver. In C. Ramakrishnan and J. Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer Berlin Heidelberg, 2008.

6. A. Dieumegard, A. Toom, and M. Pantel. Model-based formal specification of a dsl library for a qualified code generator. In *Proceedings of the 12th Workshop on OCL and Textual Modelling*, OCL '12, pages 61–62, New York, NY, USA, 2012. ACM.

7. J. Gondzio. Interior point methods 25 years later. *European Journal of Operational Research*, 2012.

8. C. Helmberg, F. Rendl, R. J. Vanderbei, and H. Wolkowicz. An interior-point method for semidefinite programming. *SIAM Journal on Optimization*, 6:342–361, 1996.

9. C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12:576–580, October 1969.

10. M. Kojima, S. Shindoh, and S. Hara. Interior-Point Methods for the Monotone Semidefinite Linear Complementarity Problem in Symmetric Matrices. *SIAM Journal on Optimization*, 7(1):86–125, Feb 1997.

11. J. Löfberg. Yalmip : A toolbox for modeling and optimization in MATLAB, 2004.

12. J. Mattingley and S. Boyd. Cvxgen: a code generator for embedded convex optimization. *Optimization and Engineering*, 13(1):1–27, 2012.

13. L. McGovern and E. Feron. Requirements and hard computational bounds for real-time optimization in safety-critical control systems. In *Decision and Control, 1998. Proceedings of the 37th IEEE Conference on*, volume 3, pages 3366–3371 vol.3, 1998.

14. L. K. McGovern. *Computational Analysis of Real-Time Convex Optimization for Control Systems*. PhD thesis, Massachussetts Institute of Technology, Boston, USA, May 2000.

15. R. D. Monteiro and Y. Zhang. A unified analysis for a class of long-step primal-dual path-following interior-point algorithms for semidefinite programming. *Math. Programming*, 81:281–299, 1998.

16. R. D. C. Monteiro. Primal–dual path-following algorithms for semidefinite programming. *SIAM J. on Optimization*, 7(3):663–678, March 1997.

17. Y. Nesterov and A. Nemirovskii. A general approach to the design of optimal methods for smooth convex functions minimization. *Ekonomika i Matem. Metody*, 24:509–517, 1988.

18. Y. Nesterov and A. Nemirovskii. *Self-Concordant functions and polynomial time methods in convex programming*. Materialy po matematicheskomu obespecheniiu EVM. USSR Academy of Sciences, Central Economic & Mathematic Institute, 1989.

19. Y. Nesterov and A. Nemirovskii. *Interior-point Polynomial Algorithms in Convex Programming*. Studies in Applied Mathematics. Society for Industrial and Applied Mathematics, 1994.

20. Y. Nesterov and M. J. Todd. Primal-dual interior-point methods for self-scaled cones. *SIAM Journal on Optimization*, 8:324–364, 1995.

21. S. Richter, C. N. Jones, and M. Morari. Certification aspects of the fast gradient method for solving the dual of parametric convex programs. *Mathematical Methods of Operations Research*, 77(3):305–321, 2013.

22. M. Rinard. Credible compilation. Technical report, In Proceedings of CC 2001: International Conference on Compiler Construction, 1999.

23. P. Roux, R. Jobredeaux, P.-L. Garoche, and E. Feron. A generic ellipsoid abstract domain for linear time invariant systems. In *HSCC*, pages 105–114, 2012.
24. S. C. RTCA. DO-178C, software considerations in airborne systems and equipment certification, 2011.
25. S. C. RTCA. DO-333 formal methods supplement to DO-178C and DO-278A. Technical report, Dec 2011.
26. T. Wang, R. Jobredeaux, and E. Feron. A graphical environment to express the semantics of control systems, 2011. arXiv:1108.4048.
27. T. Wang, R. Jobredeaux, H. Herencia-Zapana, P.-L. Garoche, A. Dieumegard, E. Feron, and M. Pantel. From design to implementation: an automated, credible autocoding chain for control systems. *CoRR*, abs/1307.2641, 2013.
28. S. Zhang. Quadratic maximization and semidefinite relaxation. *Mathematical Programming*, 87(3):453–465, 2000.

11 Appendix

11.1 Vectorization Functions

The function **vecs** is similar to the standard vectorization function but specialized for symmetric matrices. It is defined as, for $1 \leq i < j \leq n$ and $M \in \mathbb{S}^n$,

$$\text{vecs}\, M = \begin{bmatrix} M_{11}, \ldots, \sqrt{2} M_{ij}, \ldots, M_{nn} \end{bmatrix}^{\mathrm{T}}. \tag{62}$$

The factor $\sqrt{2}$ ensures the function **vecs** preserves the distance defined by the respective inner products of $\mathbb{S}^n$ and $\mathbb{R}^{\frac{n(n+1)}{2}}$. The function **mats** is the inverse of **vecs**. The function **krons**, denoted by the symbol $\otimes_{sym}$, is similar to the standard Kronecker product but specialized for symmetric matrix equations. It has the property

$$(Q_1 \otimes_{sym} Q_2)\, \text{vecs}\,(M) = \text{vecs}\left(\frac{1}{2} \left(Q_1 M Q_2^{\mathrm{T}} + Q_2 M Q_1^{\mathrm{T}} \right) \right). \tag{63}$$

Let $Q_1 = TZ$ and $Q_2 = T_{inv}$ and $M = \Delta X$, we get

$$(TZ \otimes_{sym} T_{inv})\, \text{vecs}\,(\Delta X) = \text{vecs}\left(\frac{1}{2} \left(TZ\Delta X T_{inv} + T_{inv}\Delta X ZT \right) \right). \tag{64}$$

Additionally, let $Q_1 = T$, $Q_2 = XT_{inv}$, and $M = \Delta Z$, we get

$$(T \otimes_{sym} XT_{inv})\, \text{vecs}\,(\Delta Z) = \text{vecs}\left(\frac{1}{2} \left(T\Delta Z X T_{inv} + T_{inv}X\Delta ZT \right) \right). \tag{65}$$

Combining (64) and (65), we get exactly the left hand side of the third equation in (17). Given a ΔZ, we can compute ΔX by solving $Ax = b$ for x where

$$\begin{aligned} A &= (TZ \otimes_{sym} T_{inv}) \\ \Delta X &= \text{mats}\,(x) \\ b &= \text{vecs}\,(\sigma\mu I - T_{inv}XT_{inv}) - (T \otimes_{sym} XT_{inv})\, \text{vecs}\,(\Delta Z). \end{aligned} \tag{66}$$

11.2 Annotated Code

```matlab
%% Example SDP Code: Primal-Dual Short-Step Algorithm
% ensures F0>0;
{
    F0 =[1, 0; 0, 0.1];
}
% ensures transpose(F1)==F1;
{
    F1 =[-0.750999 0.00499; 0.00499 0.0001];
}
% ensures transpose(F2)==F2;
{
    F2 =[0.03992 -0.999101; -0.999101 0.00002];
}
% ensures transpose(F3)==F3;
{
    F3 =[0.0016 0.00004; 0.00004 -0.999999];
}
% ensures smat(b)>0;
{
    b=[0.4; -0.2; 0.2];
}
F=[vecs(F1); vecs(F2); vecs(F3)];
% ensures Ft==transpose(F);
{
    Ft=F';
}
% ensures n>=1;
{
    n=length(F0);
}
% ensures m>=1;
{
    m=length(b);
}
% ensures Z>0;
{
    Z=mats(lsqr(F,-b),n);
}
% ensures X>0

% ensures trace(X*Z)<=0.1;
{
    X=[0.3409 0.2407; 0.2407 0.9021];
}
% requires Z>0 && X>0;
% ensures   trace(X*Z)>0;
{
    % empty code
```

```
49  }
50  % ensures  transpose(P)==P;
51  {
52      P=mats(lsqr(Ft,vecs(-X-F0)),n);
53      p=vecs(P);
54  }
55  % ensures  epsilon >0
56  {
57      epsilon=1e-8;
58  }
59  % ensures  sigma==0.75;
60  {
61      sigma=0.75;
62  }
63  % requires  trace(X*Z)<=0.1;
64  % ensures    phi<=0.1;
65
66  % requires  trace(X*Z)>0;
67  % ensures    phi>0;
68
69  % ensures    phi==trace(X*Z);
70  {
71      phi=trace(X*Z);
72  }
73  % requires  phi>0;
74  % ensures    phi-0.76/0.75*phi<0;
75  {
76      % empty code
77  }
78  % requires  phi-0.76/0.75*phi<0;
79  % ensures    phi-0.76*phim<0;
80  {
81      phim=1/0.75*phi;
82  }
83  % ensures    mu==trace(X*Z)/n;
84  {
85      mu=trace(X*Z)/n;
86  }
87
88  % requires  phi>0 && phi<=0.1;
89  % ensures    phi>0 && phi<=0.1;
90
91  % requires  phi-0.76*phim<0;
92  % ensures    phi-0.76*phim<0;
93
94  % requires  norm(X*Z-mu*eye(n,n))<=0.3105*mu && X>0 && Z>0;
95  % ensures    norm(X*Z-mu*eye(n,n))<=0.3105*mu && X>0 && Z>0;
96  {
97      while (phi>epsilon) do
98      % requires  phi>0 && phi<=0.1 && phi==trace(X*Z);
```

```
99      % ensures   trace(X*Z)>0 && trace(X*Z)<=0.1;

100

101     % requires norm(X*Z-mu*eye(n,n))<=0.3105*mu;
102     % requires mu==trace(X*Z)/n;
103     % ensures   norm(X*Z-trace(X*Z)/n*eye(n,n))<=0.3105*trace(X*
             Z)/n;

104

105     {
106        % empty code
107     }
108     % requires trace(X*Z)>0 && trace(X*Z)<=0.1;
109     % ensures   trace(Xm*Zm)>0 && trace(Xm*Zm)<=0.1;

110

111     %  requires X>0 && Z>0;
112     %  ensures   Xm>0 && Zm>0;

113

114     %  requires norm(X*Z-trace(X*Z)/n*eye(n,n))<=0.3105*trace(X
             *Z)/n;
115     %  ensures   norm(Xm*Zm-trace(Xm*Zm)/n*eye(n,n))<=0.3105*
             trace(Xm*Zm)/n;
116     {
117        Xm=X;
118        Zm=Z;
119     }
120        pm=p;
121     % requires n>=1;
122     % ensures   trace(Xm*Zm)==n*mu;

123

124     %  requires norm(Xm*Zm-trace(Xm*Zm)/n*eye(n,n))<=0.3105*
             trace(Xm*Zm)/n;
125     %  ensures   norm(Xm*Zm-mu*eye(n,n))<=0.3105*mu;
126     {
127        mu=trace(Xm*Zm)/n;
128     }
129     %  requires norm((Zm^(0.5))^(-1)*mats(lsqr(F,zeros(m,1)),n)
             *Zm^(0.5))<=0.7;
130     %  ensures   norm(Zhi*mats(dZm,n)*Zhi)<=0.7;

131

132     %  requires norm(Zm^(0.5))^(-1)*mats(lsqr(krons((Zm^(0.5))
             ^(-1)*Zm,(Zm^(0.5))',n,m),vecs(sigma*mu*eye(n,n)-(Zm
             ^(0.5))*Xm*(Zm^(0.5)))-krons((Zm^(0.5))^(-1),(Zm^(0.5))
             '*Xm,n,m)*lsqr(F,zeros(m,1))),n)*mats(lsqr(F,zeros(m,1)
             ),n)*Zm^(0.5))<=0.3105*sigma*mu;
133     %  ensures   norm(Zhi*mats(dXm,n)*mats(dXm,n)*Zhi)<=0.3105*
             sigma*mu;
134     {
135        % requires n*mu==trace(Xm*Zm);
136        % requires trace(Xm*mats(lsqr(F,zeros(m,1)),n))+trace(
                mats(lsqr(krons((Zm^(0.5))^(-1)*Zm,(Zm^(0.5))',n,m),
                vecs(sigma*mu*eye(n,n)-(Zm^(0.5))*Xm*(Zm^(0.5)))-
```

```
            krons ((Zm^(0.5))^(-1),(Zm^(0.5))'*Xm,n,m)*lsqr(F,
            zeros(m,1))),n)*Zm)+trace(Xm*Zm)-sigma*n*mu==0;
137     % ensures    trace(Xm*mats(lsqr(F,zeros(m,1)),n))+trace(
            mats(lsqr(krons((Zm^(0.5))^(-1)*Zm,(Zm^(0.5))',n,m),
            vecs(sigma*mu*eye(n,n)-(Zm^(0.5))*Xm*(Zm^(0.5)))-
            krons((Zm^(0.5))^(-1),(Zm^(0.5))'*Xm,n,m)*lsqr(F,
            zeros(m,1))),n)*Zm)+trace(Xm*Zm)-sigma*trace(Xm*Zm)
            ==0;
138     {
139        % empty
140     }
141     % requires  trace(Xm*mats(lsqr(F,zeros(m,1)),n))+trace(
            mats(lsqr(krons((Zm^(0.5))^(-1)*Zm,(Zm^(0.5))',n,m),
            vecs(sigma*mu*eye(n,n)-(Zm^(0.5))*Xm*(Zm^(0.5)))-
            krons((Zm^(0.5))^(-1),(Zm^(0.5))'*Xm,n,m)*lsqr(F,
            zeros(m,1))),n)*Zm)+trace(Xm*Zm)-sigma*trace(Xm*Zm)
            ==0;
142     % ensures    trace(Xm*mats(lsqr(F,zeros(m,1)),n))+trace(
            mats(lsqr(H,vecs(sigma*mu*eye(n,n)-Zh*Xm*Zh)-G*lsqr(F
            ,zeros(m,1))),n)*Zm)+trace(Xm*Zm)-sigma*trace(Xm*Zm)
            ==0;
143     {
144        Zh=Zm^(0.5);
145        Zhi=Zh^(-1);
146        G=krons(Zhi,Zh'*Xm,n,m);
147        H=krons(Zhi*Zm,Zh',n,m);
148     }
149     % requires  trace(Xm*mats(lsqr(F,zeros(m,1)),n))+trace(
            mats(lsqr(H,vecs(sigma*mu*eye(n,n)-Zh*Xm*Zh)-G*lsqr(F
            ,zeros(m,1))),n)*Zm)+trace(Xm*Zm)-sigma*trace(Xm*Zm)
            ==0;
150     % ensures    trace(Xm*mats(dZm,n))+trace(mats(dXm,n)*Zm)+
            trace(Xm*Zm)-sigma*trace(Xm*Zm)==0;
151
152     % ensures   dZm==lsqr(F,zeros(m,1));
153     {
154        r=sigma*mu*eye(n,n)-Zh*Xm*Zh;
155        dZm=lsqr(F,zeros(m,1));
156        dXm=lsqr(H, vecs(r)-G*dZm);
157     }
158  }
159  % requires  sigma==0.75;
160  % requires  trace(Xm*mats(dZm,n))+trace(mats(dXm,n)*Zm)+
         trace(Xm*Zm)-sigma*trace(Xm*Zm)==0;
161  % ensures    trace(Xm*mats(dZm,n))+trace(mats(dXm,n)*Zm)+
         trace(Xm*Zm)-0.75*trace(Xm*Zm)==0;
162  {
163     % empty code
164  }
165  % requires   dZm==lsqr(F,zeros(m,1));
```

```
166    % ensures    lsqr(Ft,-dXm)'*F*dZm==0;
167    {
168       % empty code
169    }
170    % requires  lsqr(Ft,-dXm)*F*dZm==0;
171    % requires  transpose(F)==Ft;
172    % ensures    dot(Ft*lsqr(Ft,-dXm),dZm)==0;
173    {
174       % empty code
175    }
176    % requires  dot(Ft*lsqr(Ft,-dXm),dZm)==0;
177    % ensures    trace(mats(Ft*lsqr(Ft,-dXm),n)*mats(dZm,n))==0;
178    {
179       % empty code
180    }
181    % requires  trace(mats(Ft*lsqr(Ft,-dXm),n)*mats(dZm,n))==0;
182    % requires  Ft*lsqr(Ft,-dXm)==-dXm
183    % ensures    trace(mats(dXm,n)*mats(dZm,n))==0
184    {
185       % empty code
186    }
187    % requires  trace(mats(dXm,n)*mats(dZm,n))==0;
188    % requires  trace(Xm*mats(dZm,n))+trace(mats(dXm,n)*Zm)+
               trace(Xm*Zm)-0.75*trace(Xm*Zm)==0;
189    % ensures    trace((Xm+mats(dXm,n))*(Zm+mats(dZm,n)))-0.75*
               trace(Xm*Zm)==0;
190    {
191       % empty code
192    }
193    %   requires  mats(H*dXm)+mats(G*dZm)==sigma*mu*eye(n,n)-Zh*
               Xm*Zh;
194    %   requires  norm(Zhi*mats(dXm,n)*mats(dXm,n)*Zhi)<=0.3105*
                sigma*mu;
195    %   ensures    0.5*norm(Zhi*((Zm+mats(dZm,n)*(Xm+mats(dXm,n)-
               sigma*mu*eye(n,n))*Zh+Zh'*((Xm+mats(dXm,n)*(Zm+mats(dZm
               ,n)-sigma*mu*eye(n,n))*Zhi)<=0.3105*sigma*mu;
196    {
197       % empty
198    }
199    {
200       dpm=lsqr(Ft,-dXm);
201       p=pm+dpm
202    }
203    %   requires  norm(Zhi*mats(dZm,n)*Zhi)<=0.7;
204    %   ensures    Zm+mats(dZm,n)>0;
205    {
206       % empty
207    }
208    % require  trace((Xm+mats(dXm,n))*(Zm+mats(dZm,n)))-0.75*
               trace(Xm*Zm)==0;
```

```
209    % ensures trace(X*Z)−0.75*trace(Xm*Zm)==0;
210
211    %  requires 0.5*norm(Zhi*((Zm+mats(dZm,n)*(Xm+mats(dXm,n)−
          sigma*mu*eye(n,n))*Zh+Zh'*((Xm+mats(dXm,n)*(Zm+mats(dZm
          ,n)−sigma*mu*eye(n,n))*Zhi')<=0.3105*sigma*mu;
212    %  ensures   0.5*norm(Zhi*(X*Z−sigma*mu*eye(n,n))*Zh+Zh'*(Z*
          X−sigma*mu*eye(n,n))*Zhi')<=0.3105*sigma*mu;
213
214    %  requires Zm+mats(dZm,n)>0;
215    %  ensures   Z>0;
216    {
217      X=Xm+mats(dXm,n);
218      Z=Zm+mats(dZm,n);
219    }
220    % requires trace(Xm*Zm)>0;
221    % ensures   0.01*trace(Xm*Zm)>0;
222    {
223      % empty code
224    }
225    % requires trace(X*Z)−0.75*trace(Xm*Zm)==0
226    % requires 0.01*trace(Xm*Zm)>0;
227    % ensures   trace(X*Z)−0.76*trace(Xm*Zm)<0;
228    {
229      % empty code
230    }
231    %  requires Z>0 && Zm>0;
232    %  ensures   norm(Z^(0.5)*X*Z^(0.5)−sigma*mu*eye(n,n))<=0.5*
          norm(Zhi*(X*Z−sigma*mu*I)*Zh+Zh'*(Z*X−sigma*mu*I)*Zhi')
          ;
233    {
234      % empty
235    }
236    % requires  trace(X*Z)−0.75*trace(Xm*Zm)==0
237    % requires  trace(Xm*Zm)>0 && trace(Xm*Zm)<=0.1;
238    % ensures   trace(X*Z)>0 && trace(X*Z)<=0.1;
239
240    % requires  trace(X*Z)−0.76*trace(Xm*Zm)<0
241    % ensures   trace(X*Z)−0.76*phim<0;
242    {
243        phim=trace(Xm*Zm);
244    }
245    % requires  trace(X*Z)>0 && trace(X*Z)<=0.1;
246    % ensures   phi>0 && phi<=0.1;
247
248    % requires  trace(X*Z)−0.76*phim<0;
249    % ensures   phi−0.76*phim<0;
250
251    % ensures   phi==trace(X*Z);
252    {
253        phi=trace(X*Z);
```

```
254   }
255   %  requires  trace(X*Z)-sigma*trace(Xm*Zm)==0;
256   %  requires  trace(Xm*Zm)==n*mu;
257   %  ensures   trace(X*Z)-sigma*n*mu==0;
258   {
259     % empty
260   }
261   %  requires  trace(X*Z)-sigma*n*mu==0;
262   %  requires  norm(Z^(0.5)*X*Z^(0.5)-sigma*mu*eye(n,n))<=0.5*
          norm(Zhi*(X*Z-sigma*mu*I)*Zh+Zh'*(Z*X-sigma*mu*eye(n,n)
          )*Zhi');
263   %  requires  0.5*norm(Zhi*(X*Z-sigma*mu*eye(n,n))*Zh+Zh'*(Z*
          X-sigma*mu*eye(n,n))*Zhi')<=0.3105*sigma*mu;
264   %  ensures   norm(Z^(0.5)*X*Z^(0.5)-trace(X*Z)/n*eye(n,n))
          <=0.3105*trace(X*Z)/n;
265   {
266     % empty
267   }
268   %  requires  norm(Z^(0.5)*X*Z^(0.5)-trace(X*Z)/n*eye(n,n))
          <=0.3105*trace(X*Z)/n;
269   %  ensures   norm(Z^(0.5)*X*Z^(0.5)-mu*eye(n,n))<=0.3105*mu;
270   {
271     mu=trace(X*Z)/n;
272   }
273   %  requires  norm(Z^(0.5)*X*Z^(0.5)-mu*eye(n,n))<=0.3105*mu;
274   %  ensures   norm(XZ-mu*eye(n,n))<=0.3105*mu;
275
276   %  requires  Z>0;
277   %  requires  norm(Z^(0.5)*X*Z^(0.5)-mu*eye(n,n))<=0.3105*mu;
278   %  ensures   X>0;
279   {
280     % empty
281   }
282     end
283 }
```