Synthesis for Rational Linear Arithmetic

Final report

Sebastien Vasey

EPFL

{firstname.lastname}@epfl.ch

Contents

1. Introduction

In this report, I describe `rchoosec`, a tool to synthesize code snippets from specifications written in the language of linear rational arithmetic. `rchoosec` produces a Scala[1] subroutine that can readily be integrated in a more complex project. The main idea is to let the programmer *describe* what is wanted rather than *how* to obtain it. Another goal is to produce code that is *specialized* to the task at hand, so that it runs faster than calling a generic solver.

I outline some possible practical applications in control theory. In particular, I present a "proof of concept" simulator of a rocket in a gravitational field. The rocket's engine is partially controlled by synthesized code (see Section 5.3).

2. Examples

Given two rational numbers a and b, consider the problem of finding a rational number x such that $x < a$ and $x \leq b$. This can be directly written as a constraint for `rchoosec` as follows:

```
RChoose (x)
{
  x < a && x <= b
}
```

This declares the *variable* x, the *parameters* a and b, and the relation that x must satisfy with respect to the parameters.

From this description, `rchoosec` will produce a Scala method taking as arguments the parameter (in alphabetical order) and returning a list of elements corresponding to the variables [2]. For the example above, the following code is produced:

```
def foo (a : Rational, b : Rational) : List[Rational] =
{
  val one__ : Rational = Rational(1,1)
  val x__ : Rational = ((List[Rational](a,b)).min) − (one__)
  List(x__)
}
```

Here, `Rational` is a type implementing arbitrary precision rational numbers. Synthesis for limited precision types such as floating points is described in Section 5.2 .

Notice that given any a and b, there are infinitely many x satisfying the constraints. One can try to restrict the solution space by maximizing over all possible x. This is done by adding the function to maximize in square brackets just before the constraints, like this:

```
RChoose (x)[x]
{
  x < a && x <= b
}
```

Notice that if $a \leq b$, a is only a least upper bound (i.e *not* a maximum) to the original problem, therefore no solution exists. In such a case, `rchoosec` generates a *precondition*: a formula of the parameters that is true if and only if the resulting problem has a solution. In that case, the precondition would be $b < a$. The code generated by the powerful, but slow, Fourier-Motzkin method (see Section 3.4) is:

[1] `http://www.scala-lang.org/`

[2] I use a list instead of a tuple, as the tuple size in Scala is limited, and there could be many variables.

```
val mOne__ : Rational = Rational(−1,1)
val zero__ : Rational = Rational(0,1)
if ((((mOne__) ∗ (a)) + (b)) < (zero__))
  List(b)
else
  throw(new NoSolutionException("Pre−condition not satisfied"))
```

Other problems that can make a constraint become unsatisfiable are unfeasibility and unboundedness (see Section 4.1.2).

For more complicated formulas, the Fourier-Motzkin method becomes too slow, so generating an exact pre-condition is difficult. In that case, `rchoosec` will use a special symbol, "`unknown`", inside the precondition to indicate uncertainty. Typically, the precondition then becomes only necessary, but not sufficient. Consider the following constraints:

```
RChoose (x,y)[2∗x + y] {
  2∗x + y <= 42 &&
  (1/2)∗x + (3/4)∗y < a &&
  (5/9)∗a − 4∗y <= x &&
  exists (v, 0 <= v && v + 1 == a) &&
  forall (w, w < 1 || a < w ∗ 1000)
}
```

This demonstrates the syntax to use quantifiers and input rational numbers. Note that the variables in this example are x and y, and the only parameter is a.

The generated precondition for this example is: $\texttt{unknown} \wedge (-1 + \frac{a}{1000} < 0) \wedge (1 - a \leq 0)$. The last two relations are due to the quantified parts of the constraints, whereas the first one indicates that something more may need to hold for the problem to be satisfiable. Indeed, it turns out the assignment $a = 2$ satisfies the second part of the precondition but the synthesized code will still return an error:

```
choosec.synthesis.NoSolutionException: Problem is bounded by least upper bound 464/45
and feasible but has no optimal solution
```

3. Synthesis and the Fourier-Motzkin method

In this Section, I give the basic definitions and describe the Fourier-Motzkin synthesis method.

3.1 Ordered Fields

Definition 1. An *ordered field* $(F, +, \star, \leq)$ is a field together with a total order $\leq$ satisfying the following properties:

1. For any $a, b, c \in F$, if $a \leq b$, then $a + c \leq b + c$
2. For any $a, b \in F$, if $a \geq 0$ and $b \geq 0$, then $ab \geq 0$.

The rationals, computable, or real numbers are well known examples of ordered fields, whereas finite fields or the complex numbers cannot be ordered to satisfy the above properties. Moreover, it is easy to see that any ordered field must contain the rational numbers as a subfield. [Lang 1993].

To avoid any numerical issue, my implementation uses "pure" rational numbers (i.e implemented using unbounded precision integers) by default, but the techniques described in this report work for any ordered field [3], so I will formulate them in this setting whenever possible. Note that the finite-precision floating point numbers, as implemented on a modern computer, are not an ordered field as they form a finite set. However they are very fast to compute with, and one can argue they are a good "practical approximation" of an ordered field, so my implementation can also cope with them, with the limitations described in Section 5.2.

I will use $\mathbb{Q}$ to refer to the set of rational numbers, and the symbol $\mathbf{Q}$ to refer to an arbitrary ordered field. For an arbitrary set S, and $m, n \in \mathbb{Z}_{>0}$, S^n will denote the set of n components vectors of elements in S, and $S^{m\times n}$ will denote the set of matrices with m lines, n columns, and elements in S.

3.2 Linear ordered-field arithmetic

Definition 2. Given an ordered field $\mathbf{Q}$, a *formula of linear* $\mathbf{Q}$*-arithmetic* is a first order boolean formula whose relation symbols are $\leq, <$, or $=$, and whose terms are of the form

$$q_1x_1 + q_2x_2 + ... + q_nx_n$$

where $q_i \in \mathbf{Q}$ for all i, and the x_is are variable symbols.

For example, a formula of linear rational arithmetic is $2a + 3b < \frac{1}{2}x \vee (x = 42 \wedge \neg\forall y : y \leq x)$, whereas $x_1x_2 = 0$ or $ax_1 + bx_2 < 1$ are not formulas of linear arithmetic.

3.3 Synthesis

Given a formula of linear $\mathbf{Q}$ arithmetic $\phi[x_1, ..., x_n, a_1, ..., a_m]$[4], one is interested in the following questions:

1. Given values $b_1, ..., b_m \in \mathbf{Q}$ for the *parameters* $a_1, ..., a_m$ does there exist $y_1, ..., y_n \in \mathbf{Q}$ such that

$$\phi[y_1/x_1, ..., y_n/x_n, b_1/a_1, ..., b_m/a_m] \quad (1)$$

   is true ?
2. If they exist, how can they be (efficiently) computed ?

Formally, to answer the first question one wants to find a *precondition* $\tilde{\phi}[a_1, ..., a_m]$ such that $\tilde{\phi} \Leftrightarrow \exists x_1...\exists x_n\phi$. Because it will be evaluated on a computer, $\tilde{\phi}$ needs to be quantifier-free. Thus one wants to perform *quantifier elimination* on ϕ.

To answer the second question, given $b_1, ..., b_m$ such that $\tilde{\phi}[b_1/a_1, ..., b_m/a_m]$ one wants to compute *witnesses* $y_1(b_1, ..., b_m), ..., y_n(b_1, ..., b_m)$ such that (1) is true (i.e one wants to find an explicit model for $\phi[b_1/a_1, ..., b_m/a_m]$).

A more elaborate formal definition of a synthesis procedure is given in [Kuncak et al. 2010]. Here it suffices to say that `rchoosec` takes as input a formula of linear $\mathbf{Q}$ arithmetic and a (non-empty) subset $\{x_1, ..., x_n\}$ of variables appearing in the formula, and outputs a precondition (also in the language of linear $\mathbf{Q}$-arithmetic) and code to compute the witnesses as described above. This code is called the *synthesized* code in this report.

`rchoosec` also takes as input a linear function $f : \mathbf{Q}^n \to \mathbf{Q}$ to be maximized over all possible satisfiable values. Here, linear means f must be of the form $f(x_1, ..., x_n) = q_1x_1 + ... + q_nx_n$, with $q_i \in \mathbf{Q}$ for all i. In order to

[3] The implementation could for example be modified to handle the computable numbers.

[4] The notation $\phi[y_1, ..., y_n]$ is used to denote a formula whose free variables are in $\{y_1, ..., y_n\}$, whereas $\phi[t_1/x_1, ..., t_n/x_n]$ denotes the formula where x_i has been replaced by t_i.

be able to synthesize more efficient code, it is sometimes convenient to consider this input separately. However, observe that maximizing f over all $x_1, ..., x_n$ such that ϕ holds is equivalent to finding $x_1, ..., x_n$ such that

$$\phi_f := \phi \land \forall y_1 ... \forall y_n : (\neg\phi[y_1/x_1, ..., y_n/x_n] \lor f(y_1, ..., y_n) \leq f(x_1, ..., x_n)). \qquad (2)$$

holds. Therefore adding an objective function does not change the general problem. The following definitions summarize the above discussion.

Definition 3 (Synthesis Problem)**.** A *synthesis problem* is a triple (ϕ, f, x), where ϕ is a formula of linear $\mathbf{Q}$ arithmetic, x is a vector of $n \geq 1$ free variables of ϕ, and $f : \mathbf{Q}^n \to \mathbf{Q}$ is a linear objective function.

Definition 4 (Satisfiability)**.** A synthesis problem $(\phi[x_1, ..., x_n, a_1, ..., a_m], f, (x_1, ..., x_n))$ is said to be *satisfiable* for parameter values $b_1, ..., b_m$ if $\exists x_1 ... \exists x_n \phi_f[b_1/a_1, ..., b_m/a_m]$ is true. The problem is said to be *unsatisfiable* if it is not satisfiable.

Computability One of the main reason for restricting oneself to the language of ordered-field linear arithmetic is that there exists a decision procedure for this theory, i.e there is an algorithm to decide if any closed formula of the language is true or false[5]. In contrast, it is a corollary of Gödel's incompleteness theorem that if one allows e.g multiplication between variables, then the problem is no longer decidable for rational arithmetic [Robinson 1949] (although it stays decidable for e.g arithmetic over the computable numbers [Tarski 1951]).

Efficiency Moreover, if one disallows disjunctions, any formula can be decided quickly using methods from linear programming. This assertion is made precise in appendix D.

Thus one can hope there also exists asymptotically efficient synthesis algorithms for some particular cases. In contrast, integer programming is well-known to be NP-hard, so synthesis for integer linear arithmetic is (believed to be) computationally more difficult, although it can also be done (see [Kuncak et al. 2010]).

3.4 The Fourier-Motzkin synthesis method

The Fourier Motzkin quantifier elimination method does the following: given a formula ϕ of linear arithmetic and one of its free variable x, it returns a new formula ψ that does *not* have x as a free variable such that $\exists x\phi \Leftrightarrow \psi$. Because $\forall x\phi$ is equivalent to $\neg\exists x\neg\phi$, the method can be used as a decision procedure for linear arithmetic. Moreover, the method can also be adapted to synthesis, as is explained in details in [Kuncak et al. 2010]. In this Section, I will summarize the Fourier-Motzkin method as it is used in `rchoosec` . I assume one wants to synthesize the formula ϕ without any objective function to maximize (one can always use (2) if needed), for the single variable x only (synthesis for several variables is done recursively).

DNF assumption Given $\phi[x]$, first convert it to prenex disjunctive negation-normal form, i.e find an equivalent formula $\psi[x]$ of the form

$$\psi = Q_1 x_1 Q_2 x_2 ... Q_m x_m \,(\psi_1 \lor \psi_2 \lor ... \lor \psi_n)$$

Where $Q_i \in \{\forall, \exists\}$, and the ψ_is are conjunctions, with the negations "pushed as far inside as possible". In the worst case, such a conversion can be done in $\mathcal{O}(2^{|\phi|})$ time, where $|\phi|$ is e.g the size of a binary encoding for ϕ. It is then clear that all the quantifiers can be eliminated by using the Fourier-Motzkin quantifier elimination method recursively.

Note however that even though eliminating an existential quantifier is simple, eliminating a universal quantifier is not, as conversions to DNF must be done several times. For example, consider the formula $\forall y(\phi_1 \lor \phi_2 \lor ... \lor \phi_n)$. It can be rewritten $\neg\exists y\neg(\phi_1 \lor ... \lor \phi_n) \equiv \neg\exists y\Psi$, where $\Psi[y]$ is in disjunctive normal form (i.e potentially

[5] Observe that deciding a formula is the special case of synthesis when the set of variables includes *all* free variables in the formula, i.e when there are no parameters.

exponentially larger than $\phi_1 \vee ... \vee \phi_n$). Once quantifier elimination has been done on Ψ, the negation of the result must again be converted to DNF... As a matter of fact, synthesis of universal quantifiers has been observed to be very slow using the method. It follows that (2) is not a practical formula to use. The methods described in Section 4 consider the objective function as a separate input.

Synthesizing disjunctions Because of what has been said above, one can suppose $\phi[x]$ is in disjunctive negation-normal form without quantifiers. One can actually fully remove negations by observing that $\neg(a < b) \equiv b \leq a$, $\neg(a = b) \equiv (0 < a - b) \vee (0 < b - a)$ and so on. Furthermore, observe that $\exists x(\phi_1 \vee \phi_2) \equiv \exists x \phi_1 \vee \exists x \phi_2$, thus it suffices to recursively synthesize ϕ_1 and ϕ_2. Then if $\psi_1, \psi_2, code_1, code_2$ are the preconditions and the synthesized code for the first and second conjunctions respectively, the precondition for ϕ will be $\psi_1 \vee \psi_2$, and the code will be of the form:

```
if ($\psi_1$)
    $code_1$
else if ($\psi_2$)
    $code_2$
else
    error ("Pre−condition not satisfied")
```

Synthesizing conjunctions To synthesize a conjunction ϕ, observe that if some relation in the conjunction can be put in the form $x = f(a_1, ..., a_k)$, where $x \notin \{a_1, ..., a_k\}$), then this gives an expression for x as a function of the parameters, so the code for x has been found. Otherwise, ϕ can be put in the following form:

$$\phi = \left(\bigwedge_{i=1}^{n_l}(l_i < x)\right) \wedge \left(\bigwedge_{i=1}^{n_u}(x < u_i)\right) \wedge \left(\bigwedge_{i=1}^{n_L}(L_i \leq x)\right) \wedge \left(\bigwedge_{i=1}^{n_U}(x \leq U_i)\right) \tag{3}$$

Where x does not occur in any l_i, u_i, L_i, U_i. It is then easy to see there is a solution if and only if those bounds are consistent, i.e the precondition should be:

$$\left(\bigwedge_{i,j}(l_i < u_j)\right) \wedge \left(\bigwedge_{i,j}(l_i < U_j)\right) \wedge \left(\bigwedge_{i,j}(L_i < u_j)\right) \wedge \left(\bigwedge_{i,j}(L_i \leq U_j)\right) \tag{4}$$

If ϕ originally had n literals, then (4) will have $\mathcal{O}(n^2)$ literals in the worst case. Therefore if v variables needs to be eliminated, the Fourier Motzkin synthesis method will take $\mathcal{O}(n^{2^v})$ time in the worst case. Thus the algorithm has doubly-exponential complexity, even if one does not take into account the conversion to disjunctive normal form.

To find an explicit value for x, let L be the maximum of all lower bounds, and U the minimum of all upper bounds[6]. Then if (4) is satisfied, then either $L = U$, and $x = U$ is a solution to (3), or $L < U$, and any x such that $L < x < U$ is a solution.

Finding a good midpoint As any ordered field contains the rationals, a solution that always works is to take $x = \frac{L+U}{2}$, i.e the average between L and U. However, this may not always be optimal, so in `rchoosec` computing x is left to the implementer of the ordered field's data type. More on this can be found in Section 5.1.4

[6] If there are no lower bounds, one can simply take $x = U - 1$. Similarly, if there are no upper bounds $x = L + 1$ is a solution.

4. Synthesis methods based on linear programming

Even though it is very powerful, the Fourier-Motzkin method has a very high computational cost.

The methods presented in this Section are a better alternative for many practical cases. Their main drawback is their lack of generality: they often cannot determine a sufficient precondition to the satisfiability of the problem. As a direct consequence, they cannot (in general) be used to synthesize formulas with quantifiers.

I first recall some definitions from linear programming, and then move on to explain the methods in detail.

4.1 Preliminaries

4.1.1 The "unknown" symbol

To deal with the fact that the synthesis methods may not return a meaningful precondition anymore, I use a new symbol, `unknown`, in linear arithmetic formulas to represent uncertainty. Informally, the symbol can be seen as a black box that can contain either true or false. A formal definition of the resulting three-valued logic can be found in [Kleene 1952].

4.1.2 Unsatisfiability

The following definitions show all the ways in which a synthesis problem can fail to be satisfiable.

Definition 5 (Feasibility)**.** A synthesis problem $P := (\phi[x_1, ..., x_n, a_1, .., a_m], f, (x_1, ..., x_n))$ is said to be *feasible* for the parameter values $b_1, ..., b_m$ if $(\phi, 0, (x_1, ..., x_n))$ is satisfiable for those parameter values, i.e $\phi[b_1/a_1, ..., b_m/a_m]$ has a model.

Definition 6 (Boundedness)**.** A feasible synthesis problem $P := (\phi[x_1, ..., x_n, a_1, ..., a_m], f, (x_1, ..., x_n))$ is said to be *bounded* for the parameter values $b_1, ..., b_m$ if there exists $C \in \mathbf{Q}$ such that for all $y_1, ..., y_n \in \mathbf{Q}$, $\phi[y_1/x_1, ..., y_n/x_n, b_1/a_1, ..., b_m/a_m]$ implies that $f(y_1, ..., y_n) \leq C$. The smallest such C is called the *least upper bound* of P [7]

Definition 7 (Maximizability)**.** A feasible and bounded synthesis problem $P := (\phi[x_1, ..., x_n, a_1, ..., a_m], f, (x_1, ..., x_n))$ is said to be *maximizable* for the parameter values $b_1, ..., b_m$ if the least upper bound L of the problem is also a maximum, i.e there exists $y_1, ..., y_n$ such that $\phi[y_1/x_1, ..., y_n/x_n, b_1/a_1, ..., b_m/a_m]$ and $f(y_1, ..., y_n) = L$.

Examples Assuming x is the variable one wants to maximize, and there are no parameters, the problems associated with the formulas $(x < 1) \wedge (x > 1)$, $x > 1$, $x < 1$, and $x \leq 1$ are unfeasible, unbounded, unmaximizable, and maximizable respectively. The least upper bound of the last two is 1, and it is only a maximum for the last one.

Lemma 1. A synthesis problem is satisfiable (for some given parameter values) if and only if it is maximizable for those values.

Proof. Follows directly from the definitions. □

4.1.3 Linear Programming

I recall a few elementary definitions and theorems from linear programming. I closely follow [Eisenbrand 2011], but the proofs can also be found in other introductory books like [Bertsimas and Tsitsiklis 1997] or [Matousek and Gärtner 2006].

[7] Such a smallest element always exists: the Fourier-Motzkin synthesis method can be used to compute it.

Definition 8 (Linear program)**.** Let $m, n \in \mathbb{Z}_{>0}, A \in \mathbf{Q}^{m\times n}, b \in \mathbf{Q}^m, c \in \mathbf{Q}^n$. A *linear program (LP)* is an optimization problem that can be put in the form

$$\max\{c^T x : Ax \leq b, x \in \mathbf{Q}^n\} \tag{5}$$

where the inequality is taken componentwise.

Definition 9 (Linear program with strict inequalities)**.** Let $m, n, r \in \mathbb{Z}_{>0}, A_1 \in \mathbf{Q}^{m\times n}, b_1 \in \mathbf{Q}^m, c \in \mathbf{Q}^n, A_2 \in \mathbf{Q}^{r\times n}, b_2 \in \mathbf{Q}^r$. A *linear program with strict inequalities (LPWS)* is an optimization problem of the form

$$\max\{c^T x : A_1 x \leq b_1 \wedge A_2 x < b_2, x \in \mathbf{Q}^n\} \tag{6}$$

where the inequalities are taken componentwise. The linear program corresponding to (6) is the one where the strict inequality is replaced by a non-strict one.

Definition 10 (Dual of a linear program)**.** The *dual* of (5) is defined to be the linear program

$$\min\{b^T y : A^T y = c, y \geq 0, y \in \mathbf{Q}^m\} \tag{7}$$

(5) is also called *primal* with respect to (7).

Note that a linear program is maximizable if and only if it is feasible and bounded. Such a relation does not hold for a linear program with strict inequalities.

If the LP is bounded, one can always do a change of coordinates to let the matrix A have full column rank. Hence in what follows I assume the LP is bounded and that the matrix A has full column rank, unless stated otherwise.

Definition 11 (Basis)**.** A *basis* for the LP (5) is a set $B \subseteq \{1, ..., m\}$ such that $|B| = n$ and A_B is invertible [8]. The *vertex* of a basis B is $x_B^* := A_B^{-1} b_B$. A basis is said to be *feasible* if its vertex is such that $Ax_B^* \leq b$.

Definition 12 (Roof)**.** A *roof* for the LP (5) is a basis B such that the LP

$$\max\{c^T x : A_B x \leq b_B\} \tag{8}$$

is bounded.

Lemma 2. Let B be a basis. Then B is a roof if and only if $c \in \text{cone}(\{a_i \mid a_i \in B\})$, i.e c can be written $c = \sum_{i\in B} \lambda_i a_i^T$ with $\lambda_i \geq 0$ for all $i \in B$.

Intuitively, a roof can be thought of as a set of half-spaces bounding the LP polyhedron from above. Note that Lemma 2 implies that the roofs of a linear program do *not* depend on the vector b. A feasible basis can be thought of as simply a vertex of the LP polyhedron. The relationship between the two is given by:

Lemma 3. A feasible basis B is a roof if and only if its vertex is an optimal solution of the LP (5) . Such a basis is said to be *optimal*.

Given an initial roof B, one can use the *dual simplex algorithm* (Algorithm 1) to find an optimal basis.

The primal simplex algorithm is similar, but starts with a feasible basis and moves up along the constraint polyhedron to find a roof. My implementation uses the dual simplex algorithm for reasons that will be explained in Section 4.4.

The termination and efficiency of Algorithm 1 depend on how steps two and three are implemented. The way i and j are chosen is called the *pivot rule*. While hundreds of such rules have been devised, it is still an important

[8] The notation A_B denotes the submatrix of A formed with the lines indexed by B. I sometimes write a_i for $A_{\{i\}}$.

Algorithm 1 Dual simplex algorithm

```
1: while $B$ is not feasible do
2:     Find $i \in \{1, ..., m\} - B$ such that $a_i x^*_B > b_i$.
3:     Find $j \in B$ such that $B' := (B \cup \{i\}) - \{j\}$ is a roof, and the vertex of $B'$ is feasible for $B$, i.e
       $A_B x^*_{B'} \leq b_B$.
4:     If such a $j$ does not exist, the LP is unfeasible, otherwise let $B := B'$.
5: end while
```

open problem whether there exists a rule that lets the simplex method take polynomial time in the worst case. The question is related to the polynomial Hirsch conjecture on the diameter of polyhedrons, see e.g [Eisenbrand et al. 2010]. There are however polynomial time algorithms for linear programming not based on the simplex method [Schrijver 1986], while the time complexity of the simplex algorithm has been shown to be polynomial time if one allows a "small" random perturbation of the input [Spielman and Teng 2001].

The relation between feasible bases and roofs is strongly linked to the following Theorem.

Theorem 1 (Strong duality)**.** Let $A \in \mathbf{Q}^{m \times n}$ be an arbitrary matrix. Then the primal LP (5) is feasible and bounded if and only if its dual (7) is feasible and bounded. In that case the optimal values coincide. Moreover, if (5) is feasible and unbounded, (7) is unfeasible, and similarly if (7) is feasible and unbounded, (5) is unfeasible.

Corollary 1 (Feasibility is almost as difficult as optimality)**.** If (5) is feasible and bounded, an optimal value can be found by finding a point inside the polyhedron $\{(x, y)^T \in \mathbf{Q}^{n+m} \mid Ax \leq b \wedge A^T y = c \wedge y \geq 0 \wedge b^T y \leq c^T x\}$.

4.2 Handling disjunctions and unboundedness

To use linear programming methods starting from a general synthesis problem $P := (\phi, f, (x_1, ..., x_n))$, the first steps is to eliminate negations, quantifiers and disjunctions. Then it is easy to see the resulting problem is in fact a linear program with strict inequalities.

To eliminate quantifiers from ϕ, the Fourier-Motzkin method must unfortunately be used. Eliminating negations from ϕ is done as explained in Section 3.4. To deal with disjunctions, one first needs to convert ϕ to disjunctive normal form. Hence I assume ϕ is in the form

$$\phi = \phi_1 \vee \phi_2 .. \vee \phi_k$$

where the ϕ_is are conjunctions. Synthesis methods for the problems $P_i := (\phi_i, f, (x_1, ..., x_n))$ are described in the next sections, so I assume they are available, and that $k \geq 2$.

4.2.1 General idea

Suppose one only cares about synthesizing code, not about the precondition. Given parameter values $b_1, ..., b_m \in \mathbf{Q}$, suppose that all P_is are maximizable for those values, and let $X_i := (x_{1,i}, ..., x_{n,i})$ be the corresponding ith optimal assignment. Then P is itself maximizable, and the synthesized code should output the assignment among the X_is that maximizes f.

Consider however what happens when e.g $k = 2$ and P_1 is not maximizable. Then the satisfiability of P depends on *why* P_1 is not maximizable: for example if P_1 is "only" unfeasible, but P_2 is maximizable, then P is maximizable and the synthesized code should output X_2. On the other hand if P_1 is feasible but unbounded, then P is also feasible but unbounded, so it is not satisfiable. Therefore it is necessary to make the synthesized code for the P_i return their exact *results* rather than just errors if they are unsatisfiable. To define this precisely, I first extend $\mathbf{Q}$ in order to express unsatisfiability results:

Definition 13. Given an ordered field $\mathbf{Q}$, define

$$\hat{\mathbf{Q}} := (\mathbf{Q} \times \{-1, 0\}) \cup \{-\infty, \infty\} \tag{9}$$

and order it using the lexicographical order, considering $-\infty$ as a smallest element, and ∞ as a largest element.

Intuitively, $\hat{\mathbf{Q}}$ can be thought of as $\mathbf{Q}$ with a smallest and a largest element added, and such that for every $x \in \mathbf{Q}$, there exists x_ϵ such that $x_\epsilon < x$, but for all $y < x$ with $y \in \mathbf{Q}$, $y < x_\epsilon$. In other words, there exists an element "infinitesimally smaller" than x.

Definition 14 (Optimal value of a LPWS)**.** The *optimal value* of a linear program with strict inequalities L is an element r of $\hat{\mathbf{Q}}$ defined as:

- $r = -\infty$ if L is unfeasible.
- $r = \infty$ if L is feasible but unbounded.
- $r = (q, -1)$ if L is feasible and bounded but not maximizable, with least upper bound q.
- $r = (q, 0)$ if L is maximizable with maximum q.

Definition 15 (Result of a LPWS)**.** The *result* of a linear program with strict inequalities with variables $x_1, ..., x_n$ is a tuple (r, x) where

- $r \in \hat{\mathbf{Q}}$ is the optimal value of the LPWS.
- $x \in \mathbf{Q}^n$ is an optimal vertex if the LPWS is maximizable, and 0 otherwise.

Example The result of the LPWS $\max\{2x : x \leq 1\}$ is $((2, 0), 1)$, whereas the result of the LPWS $\max\{2x : x < 1\}$ is $((2, -1), 0)$. From the ordering defined in Definition 13, one sees that the optimal value of the first LPWS is larger than the second one.

From now on, I assume that the synthesized code for the P_is returns the result of the corresponding LPWS. Let $code_1$, ..., $code_k$ be the synthesized codes for the P_is, then one can synthesize the following code for P (I assume the input parameters are $b_1, ..., b_m$)

```
val ($r_1$, $y_1$) = $code_1(b_1, ..., b_m)$
...
val ($r_k$, $y_k$) = $code_k(b_1, ..., b_m)$
val $j$ = arg max$\{r_i \mid i \in \{1, ..., k\}\}$
if ($r_j \in \mathbf{Q} \times \{0\}$)
    return $y_j$
else
    error (''Problem is unsatisfiable'')
```

This simply expresses the fact that if the LPWS that achieves the largest value is maximizable, then the problem is maximizable, otherwise it is not.

This code works fine, but it is not clear what precondition should go with it. Moreover, one needs to compute the results of *all* P_is before obtaining a result, whereas if e.g P_1 were unbounded for the particular parameter values, then there would be no need to consider the other problems. For this reason it is useful to look more closely at what causes unboundedness in order to synthesize more efficient code.

4.2.2 Handling unboundedness

The following results characterize the unboundedness of a synthesis problem whose constraint is a conjunction.

Lemma 4. Suppose the LPWS L in (6) is feasible. Then it is unbounded if and only if its corresponding LP L' is feasible and unbounded.

Proof. The "only if" part is straightforward. To prove the "if" part, introduce a new variable λ and the constraint $\lambda \leq c^T x$ to both L and L'. Apply Fourier-Motzkin elimination to L' to eliminate all variables but λ. By hypothesis, L' is unbounded, so λ must not have any upper bound in the resulting formula ψ. Because L' is a LP, all the inequalities in ψ are non-strict. Applying Fourier-Motzkin elimination to L will only result in some of those non-strict constraints becoming strict, but λ still will not admit any upper bound. Therefore L is unbounded. □

Theorem 2. If a synthesis problem P without disjunctions, negations or quantifiers is feasible but unbounded for some parameter values $d_1, ..., d_r$, then for *any* parameter values, P is either unfeasible or unbounded.

Proof. Consider the LPWS associated with P and the parameter values $d_1, ..., d_r$. By Lemma 4, it suffices to consider the corresponding linear program L that can be written $\max\{c^T x : Ax \leq b\}$. Note that only b depends on the choice of parameter values. By hypothesis, L is feasible but unbounded. By Theorem 1, the dual linear program $\min\{b^T y : A^T y = c, y \geq 0\}$ is unfeasible. But unfeasibility of the dual does not depend on b, and therefore does not depend on the parameter values. Thus the dual LP is always unfeasible, hence the primal LP is always either unfeasible or feasible but unbounded. □

The proof of Theorem 2 suggests an algorithm for finding the P_is that can potentially be unbounded: if the dual LP corresponding to P_i is feasible, P_i is always bounded (if it is feasible). Otherwise, it is always either unfeasible or unbounded. Testing feasibility of the dual LP can be done at synthesis time because this does not depend on the problem's parameters.

Suppose $P_1, ..., P_r$ are potentially unbounded, whereas $P_{r+1}, ..., P_k$ are always bounded. If $r = k$, no problems are always bounded, so one immediately see that P is always unsatisfiable. Otherwise, $P_1, ..., P_r$ all need to be *unfeasible* in order for P to be satisfiable. Once this is known, the code in Section 4.2.1 can be used for $P_{r+1} \vee ... \vee P_k$.

Boundedness assumption One can synthesize the potentially unbounded problems assuming the objective function is zero, since only feasibility matters. Thus when synthesizing a conjunction, *one can assume it cannot be unbounded*. I will always make this assumption in what follows.

Precondition Let $\psi_1, ..., \psi_k$ be the synthesized preconditions for the problems $P_1, ..., P_k$, assuming $P_1, ..., P_r$ were synthesized with zero objective function. Then a necessary (but *not* always sufficient) condition for P is

$$\psi = (\neg\psi_1 \wedge \neg\psi_2 ... \wedge \neg\psi_r) \wedge (\psi_{r+1} \vee ... \vee \psi_k) \tag{10}$$

The first part expresses that all potentially unbounded problems must be unfeasible, and the second one that at least one of the other problems must be maximizable. This is not a sufficient condition, since even if one problem is maximizable, there could exist an unmaximizable problem with a larger least upper bound. When there are no strict inequalities in $\phi_{r+1}, ...\phi_k$, then this cannot happen, hence in that case ψ is indeed an *equivalent* condition[9]. Otherwise, the synthesized precondition will be $\psi \wedge$ `unknown` to indicate the fact that it is not sufficient.

[9] Note however that the symbol `unknown` could appear in the ψ_is

Code The synthesized codes for potentially unbounded problems need not return a full result, only a boolean that is true if and only if the problem is feasible. The final synthesized code for P will then check that all those booleans are false before using the code from Section 4.2.1 for the bounded problems.

Now that disjunctions have been handled, the next Section focuses on bounded synthesis problems whose constraint formulas are conjunctions. Three different methods are given.

4.3 The Fourier-Motzkin method for optimization problems

If one is willing to let the precondition become only necessary but not sufficient, it turns out the Fourier-Motzkin synthesis method can be used also with an objective function in a much more efficient manner than what was presented in Section 3.4. I found the idea in [Schrijver 1986].

Given $\phi[x_1, ..., x_n, a_1, ..., a_m]$ and the linear objective function f, use the Fourier-Motzkin synthesis method on $\phi \wedge (\lambda \leq f(x_1, ..., x_n))$, considering λ as a parameter. One obtains an equivalent formula $\psi[\lambda, a_1, ..., a_m]$ and some code $code$ to produce $x_1, ..., x_n$ from $\lambda, a_1, ..., a_m$.

Because ϕ is a conjunction, ψ is also a conjunction, which can be written $\psi = \psi_1[a_1, ..., a_m] \wedge \psi_2[\lambda, a_1, ..., a_m]$, where each literal of ψ_2 contains λ. It is clear ψ_1 is a necessary and sufficient condition to the feasibility of the synthesis problem, so it is also a necessary condition to its maximizability. By hypothesis, the problem is bounded whenever feasible, so λ must have at least one upper bound in ψ_2.

Fixing the parameters, one can view the upper bounds as elements of $\hat{\mathbf{Q}}$ as follows: if q is a strict upper bound, view it as $(q, -1)$. If q is a non-strict upper bound, view it as $(q, 0)$. Let L be the resulting set of elements of $\hat{\mathbf{Q}}$. From correctness of the Fourier-Motzkin method, using the fact that ψ is an equivalent formula to ϕ, it follows that the optimal value of the corresponding LPWS is given by $\min L$.

To sum up, the following code (with parameter input $b_1, ..., b_m$) will be synthesized [10]:

```
val $\lambda = \min L$
if ($\lambda \in \mathbf{Q} \times \{0\}$)
    return $(\lambda, code(\lambda, b_1, ..., b_m))$
else
    return $(\lambda, 0)$
```

Note that even though this is not written explicitly, L depends on $b_1, ..., b_m$.

$\psi_1 \wedge$ `unknown` can be used as a precondition. The `unknown` symbol is not needed if L contains only elements of $\mathbf{Q} \times \{0\}$, i.e all the upper bounds on λ are non-strict. In that case the code above can also be optimized in the obvious way.

This method is simple to implement, and no specific problems appear when dealing with strict inequalities, as opposed to the next two methods. However its computational cost is as high as that of Fourier Motzkin elimination...

4.4 Calling a solver at runtime

As already explained, the synthesis problem $(\phi, f, (x_1, ..., x_n))$, where ϕ is a conjunction without quantifiers or negations can be seen as a LPWS

$$\max\{c^T x : A_1 x \leq b_1 \wedge A_2 x < b_2\} \tag{11}$$

[10] The code assumes the precondition ψ_1 holds

where only the vectors b_1 and b_2 depend on the parameters. Since the parameters will be known only at runtime, a simple idea is to generate code that calls a solver for (11). Because some precomputations depend only on A_1, A_2 and c, they can be done at compile-time. I first assume (11) is equivalent to a linear program of the form

$$\max\{c^T x : Ax \leq b\} \tag{12}$$

I then explain how the method can be adapted if the problem is really a LPWS.

4.4.1 Precomputations

Making A full rank The first step is to perform a coordinate transformation so that A has full column rank. One then obtains an equivalent LP problem $\max\{c'^T x' : A'x' \leq b\}$. At the end of the synthesized code, the coordinate transformation is played backward in order to obtain the solution to the original problem. The details are explained in [Eisenbrand 2011]. Because the problem is bounded by hypothesis, such a transformation is always possible. Algorithmically, one simply does Gauss-Jordan elimination on A, obtaining a matrix U such that AU is in column-reduced form. This matrix essentially describes the change of coordinates to be done. Note that because this transformation does not depend on b, it can be done at compile time. From now on, A is assumed to have full column rank.

Finding an initial roof As shown by Lemma 2, the roofs of (12) do not depend on b and hence do not depend on the parameters. For that reason, my runtime solver uses the dual simplex algorithm, so that I can precompute its initial roof at compile-time. It would also have been possible to use the primal simplex algorithm and compute an initial feasible basis by considering the dual LP $\min\{b^T y : A^T y = c, y \geq 0\}$. However, $b^T y$ is not a linear function, so converting to the dual slightly complicates matters...

To compute the initial roof, the LP

$$\max\{c^T x, Ax \leq 0, c^T x \leq 1\} \tag{13}$$

is solved, with its initial roof containing $c^T x \leq 1$, and $n - 1$ linearly independent lines of A such that together with c they form a linearly independent set (such lines exist because A has full column rank).

By construction, (13) is feasible (0 is a solution) and bounded. If the final optimal roof B contains the line $c^T x \leq 1$, this means the LP $\max\{c^T x : Ax \leq 0\}$ does not have a roof (any of its roof has vertex zero, but B has a non-zero vertex). Thus (12) does not have a roof so is either unfeasible or unbounded. Because it is bounded by hypothesis, it must be unfeasible, so in that case the synthesis result can immediately be returned.

If B does not contain the line $c^T x \leq 1$, B is also a roof for (12) and hence the synthesized code can simply call the LP solver with B as a starting roof (the matrix A_B^{-1} can also be computed in advance). In that case, the returned precondition will simply be `unknown`, as nothing can be inferred before the solver has run. This is one of the main drawback of this method.

4.4.2 Solving a LPWS

Up to now, this report has explained how linear programs can be solved, but nothing has been said about LPWS. It turns out the result of a LPWS can be found by solving some related linear programs. To solve the LPWS (11), the first thing to do is to solve the corresponding LP

$$\max\{c^T x : A_1 x \leq b_1 \wedge A_2 x \leq b_2\} \tag{14}$$

The precomputations described in Section 4.4.1 can be done for (14). By Lemma 4, (14) is bounded. If it is unfeasible, then clearly (11) is also unfeasible. Otherwise, if x^* is an optimal solution of (14), and $A_2 x^* < b_2$,

x^* is also an optimal solution of (11) with optimal value $M := c^T x^*$. If not, it might still be possible to find an optimal solution of the LPWS among all possible optimal solutions of (14). To find out, add a new variable δ and solve a second LP:

$$\max\{\delta : A_1 x \leq b_1 \wedge A_2 x + \delta(1, 1, ..., 1)^T \leq b_2 \wedge M \leq c^T x \wedge 0 \leq \delta \leq \epsilon\} \tag{15}$$

Where $\epsilon > 0$ can be chosen arbitrarily (a small ϵ may reduce the number of iterations). Observe that (15) is feasible (as (14) is feasible) and bounded by construction. Moreover if B is an optimal roof for (14), $B \cup \{m\}$ is an initial roof for (15), where m is the index of the line describing the constraint $\delta \leq \epsilon$. Let δ^* be the optimal value of (15).

If $\delta^* > 0$, then taking the first components of the corresponding optimal vertex gives a solution to the original LPWS. Otherwise, $\delta^* = 0$, so (11) is not maximizable. However, it must still be determined whether it is feasible or not. To do so, (15) is solved without the constraint $M \leq c^T x$. If the resulting optimal value is greater than zero, (11) is feasible but not maximizable, with least upper bound M. Otherwise it is unfeasible.

Using this method, three LPs of similar size must be solved in the worst case. I have been unable to find a more efficient algorithm...

4.5 Multiparametric Linear Programming

One can wonder whether more can be done than what is described in Section 4.4: since the matrix A and the vector c are fixed, wouldn't it be possible to *specialize* the simplex algorithm to make it run faster for that particular case ?

The general problem of specializing an algorithm if part of its input is known at compile time is called *partial evaluation*[Jones et al. 1993], and several powerful techniques exist to do this automatically. However, I found it hard to partially evaluate Algorithm 1 directly, for several reasons:

1. As already mentioned, the best known worst-case upper bound on the number of iterations is exponential.
2. The result of each iteration depends heavily on the vector b .

Still, some form of "indirect" partial evaluation can be done. The idea is to split the parameter space into *polyhedral regions*. If the parameters are in a given region, their corresponding maximizer can be determined by evaluating a simple linear function. This Section describes why this is even possible, and how it can be done.

The dual version of the algorithm I present here has first been described to me by Friedrich Eisenbrand (see Acknowledgments).

4.5.1 Initial setup

I first assume the synthesis problem can be described as a LP

$$\max\{c^T x : Ax \leq b\} \tag{16}$$

For $A \in \mathbf{Q}^{m \times n}$, where only b depends on the parameters. How to adapt the method if the problem is a LPWS is explained in Section 4.5.4.

Note that in (16), b depends on the parameters in a linear way. In fact, b can be written

$$b = w + F\Theta \tag{17}$$

where $\Theta \in \mathbf{Q}^r$ is the vector containing the r parameters, and $w \in \mathbf{Q}^m, F \in \mathbf{Q}^{m\times r}$ are known at compile-time. Therefore the LP can be rewritten

$$\max\{c^T x : Ax \leq w + F\Theta\} \tag{18}$$

Finding a solution to (18) as a function of the parameters is called *multiparametric linear programming*. The algorithm I will present looks a a little like the one given in [Gal and Nedoma 1972], but is slightly simpler, at the cost of ignoring the issue of overlapping regions. Other improvements have been published, see e.g [Borelli et al. 2003].

4.5.2 Main idea

Lemma 5. The set K of all parameters for which (18) is feasible is convex.

Proof. Let $\Theta_1,\ \Theta_2\ \in\ K$. By definition of K, there exists $x_1, x_2 \in \mathbf{Q}^n$ such that $Ax_1 \leq w + F\Theta_1$, $Ax_2 \leq w + F\Theta_2$. Thus for any $0 \leq \lambda \leq 1$,

$$\begin{aligned} A(\lambda x_1 + (1-\lambda)x_2) &= \lambda A x_1 + (1-\lambda)Ax_2 \\ &\leq \lambda(w + F\Theta_1) + (1-\lambda)(w + F\Theta_2) \\ &= w + F(\lambda\Theta_1 + (1-\lambda)\Theta_2) \end{aligned}$$

Therefore $\lambda\Theta_1 + (1-\lambda)\Theta_2 \in K$. ☐

Lemma 6. Let $J := \{1, ..., m\}$. A set of lines $B \subseteq J$ is an optimal basis of (18) for *some* parameter values, if and only if the following conditions are true:

1. A_B is invertible
2. $(A_B^{-1})^T c \geq 0$
3. The polyhedron defined by the equation

$$\left(A_{\bar{B}} A_B^{-1} F_B - F_{\bar{B}}\right)\Theta \leq w_{\bar{B}} - A_{\bar{B}} A_B^{-1} w_B \tag{19}$$

   is non-empty. Here I have defined $\bar{B} := J - B$.

Proof. By Lemma 6, it is enough to show B is a feasible roof. The first condition expresses that B is a basis, the second that it is a roof (using Lemma 2). B is feasible if its vertex x^* satisfies $Ax^* \leq w + F\Theta$. By definition of x^*, $A_B x^* = w_B + F_B\Theta$, thus B is feasible if and only if $A_{\bar{B}} x^* \leq w_{\bar{B}} + F_{\bar{B}}\Theta$. Using the fact that $x^* = A_B^{-1}(w_B + F_B\Theta)$, and rearranging the terms, one obtains (19). ☐

Note that if $c = 0$, any basis will satisfy the second condition of the Lemma. Because one wants to reduce the number of optimal bases, it is a good idea to replace c with a non-zero vector in that case. Taking any non-zero line of A will do, for example.

Assuming a list $B_1, B_2, ..., B_k$ of all optimal bases satisfying Lemma 6 is available, finding the solution to (18) for a given Θ is easy: search through the list until a basis B is found such that (19) is satisfied. Then the optimal solution is given by $x^* = A_B^{-1}(w_B + F_B\Theta)$, and the maximal value by $c^T x^*$. The synthesized code would look like this:

```
if ($\Theta$ satisfies (19) with $B = B_1$){
    val xopt = $A_{B_1}^{-1}(w_{B_1} + F_{B_1}\Theta)$
    val r = $(c^T x^*, 0)$
    return (r, xopt)
}
else if ($\Theta$ satisfies (19) with $B = B_2$){
  ...
}
...
else if ($\Theta$ satisfies (19) with $B = B_k$){
  ...
}
else {
   // Problem is unfeasible
   return $(-\infty, 0)$
}
```

In contrast to calling a solver at runtime, this code only evaluates simple arithmetic expressions. In particular, testing if (19) is true will evaluate a conjunction of linear arithmetic with $m - n$ literals. In the worst case, $k(m - n)$ literals need to be evaluated, which can be a problem if k is large. Although I have not implemented this, it is possible to make the search take only about $\log(k)$ operations if one is willing to do a lot more preprocessing [Tøndel et al. 2002].

One can output a necessary and sufficient precondition by taking the disjunction of (19) for all optimal bases.

4.5.3 Finding all the regions

One question remains: how can the list of all optimal bases be found ? One simple strategy is to test all possible n-subsets of $\{1, ..., m\}$ and add only those that satisfy the conditions of Lemma 6. Note however that to test whether (19) has a solution, the corresponding linear program needs to be solved, so to test this for all possible subsets, one would need to solve $\binom{m}{n}$ linear programs in the worst case, which is clearly prohibitive. The approach presented below is better in some cases.

Definition 16 (Neighboring basis)**.** Two n elements sets B_1 and B_2 are said to be *neighbors* if $|B_1 \cap B_2| = n-1$.

Definition 17 (Graph of optimal bases)**.** Let S be the set of all bases satisfying the conditions of Lemma 6. The *graph of optimal bases* is defined as the undirected graph with vertex set S in which two bases share an edge if and only if they are neighbors.

Theorem 3. The graph G of optimal bases is connected.

Proof. If $|S| \leq 1$, there is nothing to prove. Otherwise, let B_1, B_2 be two different optimal bases in G. Let Θ_1, Θ_2 be two different parameter vectors for which B_1 and B_2 respectively are optimal bases. By Lemma 5, $\Theta(\lambda) := \lambda\Theta_1 + (1-\lambda)\Theta_2$ admits an optimal basis for any $0 \leq \lambda \leq 1$. Intuitively, one has to choose among those bases to form a path from B_1 to B_2 in G. To do this, consider (18) as a parametric linear program in λ only, i.e of the form $\max\{c^T x : Ax \leq w + F(\Theta(\lambda))\}$ which is equivalent to $\max\{c^T x : Ax \leq \tilde{w} + \tilde{F}\lambda\}$, for some $\tilde{w} \in \mathbf{Q}^m, \tilde{F} \in \mathbf{Q}^{m\times 1}$. One can consider the associated LP

$$\max\left\{ \begin{pmatrix} c^T & 0 \end{pmatrix} \begin{pmatrix} x \\ \lambda \end{pmatrix} : \begin{pmatrix} A & -\tilde{F} \\ 0 & 1 \\ 0 & -1 \end{pmatrix} \begin{pmatrix} x \\ \lambda \end{pmatrix} \leq \begin{pmatrix} \tilde{w} \\ 1 \\ 0 \end{pmatrix} \right\} \tag{20}$$

The last two lines indicate that $0 \leq \lambda \leq 1$. Note that the matrix of the LP has full column rank. By the above discussion, (20) is feasible (in fact it is feasible for any λ), and bounded, as one of $B_1 \cup \{m+1\}$, or $B_2 \cup \{m+2\}$ is a roof for it. To each roof of (20) correspond at least one optimal basis in G, and conversely to each optimal basis in G correspond at least one roof in the LP. Let B be an optimal basis for (20), and B^* be a corresponding basis in G. Starting from the roof corresponding to B_1, run the simplex algorithm to reach B. To the sequence of visited bases corresponds a path from B_1 to B^* in G. Similarly, there is a path from B_2 to B^*. Hence there is a path from B_1 to B_2. □

Theorem 3 suggests the following algorithm: find an initial optimal basis B. List all the neighboring bases of B, and find out which ones are optimal. Then repeat the same steps for those bases, until the full graph has been built. In other words, do a graph traversal on G.

Finding the initial basis An initial optimal basis can be found in two steps. First, find a parameter vector Θ^* such that (18) is feasible. This can be done by solving the LP with no objective function, considering both x and Θ as variables. Second, find an optimal basis for the linear program $\max\{c^T x : Ax \leq w + F\Theta^*\}$.

Computational complexity The number of neighbors of an n element subset of $\{1, ..., m\}$ is $n(m-n)$. Thus if there are k optimal bases, the algorithm will check $\mathcal{O}(n(m-n)k)$ sets for optimality. If, e.g F has a low column rank, there could be much fewer optimal bases than $\binom{m}{n}$, so for such cases the algorithm is much faster than the brute-force approach.

However, it is very easy to come up with examples where the number of regions grow exponentially. Consider for example the dual of (18)

$$\max\{-(w + F\Theta)^T y : y \in P\}$$

Take $r = m, w = 0, F = I_m$, and P the cube, i.e $P := \{y \in \mathbf{Q}^m \mid -1 \leq y_i \leq 1, i \in \{1, ..., m\}\}$. Then for any of the 2^m vertices of P, there is a Θ such that it is an optimal vertex.

It follows that the method also has computational complexity too high to be used on “large” problems. However the complexity is still asymptotically better than that of Fourier-Motzkin elimination.

4.5.4 Handling strict inequalities

If the problem to be synthesized is the LPWS $\max\{c^T x : A_1 x \leq b_1 \wedge A_2 x < b_2\}$, techniques similar to the one explained in Section 4.4.2 can be used: find all the optimal bases for the LPs $\max\{c^T x : A_1 x \leq b_1 \wedge A_2 x \leq b_2\}$, $\max\{\delta : A_1 x \leq b_1 \wedge A_2 x + \delta(1, 1, ..., 1)^T \leq b_2 \wedge c^T x \geq \lambda\}$, and $\max\{\delta : A_1 x \leq b_1 \wedge A_2 x + \delta(1, 1, ..., 1)^T \leq b_2\}$, where λ is considered a parameter which will be replaced by the maximal value of the first LP. Then generate the code equivalent to the algorithm described in Section 4.4.2.

When the problem has strict inequalities, the synthesized precondition is no longer sufficient so one needs to take its conjunction with the `unknown` symbol before returning it.

5. Experimental results and practical applications

In this Section, I describe `rchoosec` , my implementation of the algorithms presented in the previous sections. I explain how I measured its performance on some sample synthesis problems. I also present the integration of synthesized code in a rocket simulation.

5.1 Implementation details

All the synthesis methods for linear rational arithmetic described in this report have been implemented in a Scala program I call `rchoosec` . The program takes as input a constraint in linear rational arithmetic and produces the synthesized code in Scala. This Section gives some details on its implementation.

5.1.1 Basic Usage

This documents exactly what `rchoosec` does, and the input it takes.

The RChoose language For convenience, `rchoosec` can read its input from a text file written in the `RChoose` language. The simple syntax of this language has been illustrated in Section 2, and should be self-explanatory. A context-free grammar for the language is given in appendix A.

It is of course also possible to directly construct the constraints in Scala using the internal representation used by `rchoosec` .

Choosing the synthesis method The synthesis method can be chosen by the user via command line options. They are documented in the `README` file coming with the program. If no specific method is given, the Fourier-Motzkin method for optimization problems will be used for "small" problems, and a solver will be called at runtime for large problems. Here "small" means that the matrix A of the LP has less than 9 lines, and less than 5 variables. Those numbers are somewhat arbitrary, see Section 6 for ideas on how to better choose a default method.

Precondition `rchoosec` outputs the precondition on standard output, and also writes it as a comment in the generated code.

Generated code `rchoosec` generates Scala code for an `Object` containing two methods:

- The synthesized method: takes as arguments the parameters and return values for the variables so that they satisfy the constraints, or throw an exception if there are no solutions. This first checks that the precondition is satisfied.
- A `main` method that can be used for testing: reads the parameters from standard input, launch the synthesized method, and prints the results.

The object also contains several variables defining rational constants used in the generated code (see Section 5.1.2), as well as the LP solver used in the synthesized code. The solver is defined outside of the synthesized method, so that it can keep its state from one invocation of the method to the next. This is useful for optimizations like remembering the last optimal basis, and trying it again the next time the solver is called, see Section 5.1.5.

5.1.2 Code Generation

Code generation in `rchoosec` is not as simple as printing the code samples shown in sections 3.4 and 4. Indeed, they are too general: it may happen that some condition inside an `if` statement is always false, or that the value of a variable is a constant that can be propagated further. Thus it is important to do some *optimization* before printing the synthesized code.

To do this, I first synthesize code in a simple intermediate language, internally called `Simple`. I run the following optimizations on the resulting code:

1. Constant propagation and constant folding
2. Dead code elimination

3. Some symbolic folding, e.g if the expression $x + 2x$ appears, it can be simplified to $3x$
4. Hash Consing

The first two optimizations are described in [Schwartzbach 2008]. Hash consing simply declares one variable for each different constant rational number appearing in the code: the idea is that since e.g zero appears several times, it will only be initialized once. This also makes the code much more readable, since one can give very short names to those variables.

Those optimizations are done on the code repeatedly, until a fixed point is reached.

5.1.3 The simplex solver

`rchoosec` has its own simplex solver. The solver works for any ordered field, and is exact, i.e it assumes there are no numerical errors in all of the ordered field's operations operations.

Performance The solver was not written with performance in mind. For example, all matrix operations are performed functionally, without any destructive update. The results of Section 5.4 confirm the program is slow, even compared to other exact solvers.

Pivot rule The solver implements the dual simplex algorithm (Algorithm 1). The lexicographical pivoting rule is used, as described in [Eisenbrand 2011]. This rule guarantees that the solver will always terminate (i.e no cycling can occur), even if the LP is degenerate.

5.1.4 Arithmetic operations

In this Section, I describe how I implemented some arithmetic operations that are useful for synthesis.

Finding a good midpoint between two rationals In the Fourier-Motzkin synthesis method, one often needs to find a number x such that $L < x < U$. An obvious such x is $\frac{L+U}{2}$. It turns out however that this choice can be far from optimal for rationals, as it increases the size of x unnecessarily. As an extreme example, suppose $L = \frac{-1}{2^{1024}}$, $U = \frac{1}{2^{1024}-1}$. Then the average would be $\frac{-1}{2^{1025}(2^{1024}-1)}$ which must use about 2000 bits to store the denominator, but $x = 0$ is also a midpoint that takes a much smaller amount of memory...

Because of this problem, the synthesized code uses the abstract method `upTo` instead of directly computing an average. It is up to the implementer of the ordered field's data type to provide the method. For rationals, I use a binary-search-like algorithm that finds an x with *smallest* denominator. I now describe this algorithm. If $L < 0$, and $U > 0$, take $x = 0$. Otherwise, by symmetry one can assume $0 \leq L < U$.

Let $L = \frac{p_1}{q_1}$, $U = \frac{p_2}{q_2}$, where $q_1, q_2 \in \mathbb{Z}_{>0}, p_1, p_2 \in \mathbb{Z}_{\geq 0}, \gcd(p_1, q_1) = \gcd(p_2, q_2) = 1$. One wants to find $x := \frac{p}{q}$, with $\gcd(p, q) = 1, p, q \in \mathbb{Z}_{>0}$ such that

- $x < U$, i.e $\frac{p}{q} < \frac{p_2}{q_2}$, i.e $pq_2 < p_2 q$, i.e
$$pq_2 \leq p_2 q - 1 \tag{21}$$
- $L < x$, i.e
$$p_1 q \leq p q_1 - 1 \tag{22}$$

If q is fixed, (21) implies

$$p \leq \left\lfloor \frac{p_2 q - 1}{q_2} \right\rfloor \tag{23}$$

and (22) implies

$$p \geq \left\lceil \frac{p_1 q + 1}{q_1} \right\rceil \tag{24}$$

Assuming consistency of the bounds, the minimal p for a given q^* is

$$p^* = \left\lceil \frac{p_1 q^* + 1}{q_1} \right\rceil \tag{25}$$

In conclusion, to know if a given q^* can be used as denominator, it suffices to check that (24) and (23) are consistent, then a corresponding p can be found by (25). The algorithm simply does a binary search to find an optimal denominator q^*. One can for example start with lower bound 1, and upper bound $2q_1q_2$. However, by doing straightforward manipulations of (22) and (21), one can show that[11]

$$\left\lceil \frac{q_1 + q_2}{p_2 q_1 - p_1 q_2} \right\rceil \leq q^* \leq \left\lceil \frac{q_1 q_2 + q_1 + q_2}{p_2 q_1 - p_1 q_2} \right\rceil \tag{26}$$

The algorithm uses these bounds instead. As a simple example, consider what happens on input $L = \frac{1}{2}, U = \frac{2}{3}$. Then $\left\lceil \frac{q_1+q_2}{p_2q_1-p_1q_2} \right\rceil = 5$, and it turns out indeed $\frac{1}{2} < \frac{3}{5} < \frac{2}{3}$. This is a much better solution than the average of L and U, $\frac{7}{12}$. Note that the denominator of the average, 12, is even worse than the upper bound given by (26) which is 11. This example shows the lower-bound is tight. The example $L = \frac{1}{7}, U = \frac{6}{7}$ gives 2 as minimal denominator, which is tight with respect to the upper bound in (26).

Notice that since p^* increases whenever q^* increases, the algorithm actually gives a rational of *smallest encoding size* between L and U.

Approximating a real number by a rational If one chooses not to synthesize floating point code (see Section 5.2), the synthesized code must be called with pure rational arguments. Initializing a rational number from a floating point number x is of course a conceptually simple operation, but one may want to merely *approximate* x with a rational of low denominator. More generally, given a real number x and a natural number N, one wants to (quickly) find the best rational approximation r to x with denominator at most N. Clearly, such a rational will not always have denominator N; for example, if $x = \frac{1}{3}$, $N = 100$, the obvious best approximation is x itself, and x cannot be written exactly as a rational with denominator 100. There are less trivial examples, e.g $\frac{22}{7}$ is a better approximation to π than $3.14 = \frac{314}{100}$.

The problem is well known, and there is an efficient algorithm to solve it using continued fractions, see e.g [Schrijver 1986]. This is what I ended up implementing.

Converting a rational to a floating point value This must be done when synthesizing code for floating point types (see Section 5.2). In `rchoosec` , this is simply implemented by converting the numerator and the denominator to floating point, and dividing them. It is clear this does not work for corner cases, like e.g $\frac{2^{4096}}{2^{4096}-1}$, so `rchoosec` fails in those cases.

5.1.5 Optimizations and tricks

I describe some optimizations to the methods described in sections 3.4 and 4 that are implemented in `rchoosec` .

[11] A proof is given in appendix B.

Checking feasibility Before starting synthesis of a formula without conjunctions, quantifiers or negations, `rchoosec` first checks whether there exists parameters for which it has a solution. This is done by considering the parameters as variables, and calling a simplex solver.

Column-reducing the constraint matrix If any of the synthesis method of Section 4 is used, `rchoosec` actually always applies the change of coordinate described in Section 4.4.1 to make the constraint matrix full rank. There are several reasons for doing this:

1. The number of variables is reduced.
2. This is an alternative to solving the dual LP to find out whether the problem can be unbounded: as explained in [Eisenbrand 2011], when the problem is unbounded either the coordinate transformation fails, or an initial roof cannot be found. Since those are steps that are done anyway when calling a solver at runtime, they may as well be used for the other methods.

Eliminating equalities Given a formula ϕ without negations, conjunctions or quantifiers, ϕ needs to be converted to the LPWS in "standard" form $A_1x \leq b_1 \wedge A_2x < b_2$. If ϕ contains the equality $a = b$, a straightforward trick is to write that equality as $a \leq b \wedge b \leq a$. This is not the best thing to do however: solving linear programs is harder than doing Gaussian elimination ! Thus what is done in `rchoosec` instead is to first perform Gaussian eliminations to eliminate the equalities in ϕ, synthesizing code as explained in Section 3.4, and only then use the more advanced methods of Section 4.

A generalization of this idea is to try to use the Fourier-Motzkin method "as long as possible", i.e if a variable in ϕ can be eliminated *without* making the formula's size grow, this must be attempted. Unfortunately, I haven't had time to implement this extension...

Remembering the last optimal roof When synthesizing code that calls a solver at runtime, it is expected that the code will be called several times in one program run. Moreover, in some cases the parameter values may not differ that much from one call to another, hence the optimal basis is likely to stay the same. For that reason, `rchoosec` 's solver keeps track of the last optimal roof found for each pair of constraint matrix A and objective vector c. This last optimal roof is used as an initial roof the next time the solver is called.

If the parameter values are very different from one call to the next, there is no reason to believe the last optimal roof is better or worse than the one computed at synthesis time, thus the effect of this optimization should be somewhat neutral.

If, however, the parameters stay close, this optimization can lead to dramatic improvements, as shown in Section 5.4.3

5.2 Synthesis for floating point types

Because they have finite-precision, floating point types cannot be considered as ordered fields. Because of numerical imprecisions, the results given by LP solvers using them are almost never exact. While this may not be a problem for some problems in engineering, several theoretical applications need exact solutions. For example, when using an LP solver to find bounds to integer programming problems, it matters very much whether a maximum is larger than 200.000001, or just larger than 199.999999 . In general, many complicated issues arise when one wants to write correct programs that use floating point types, see e.g [Monniaux 2007].

For that reason, I chose to always do the computations exactly. However, the synthesized code can still be *computed* using exact numbers, but *executed* with floating points. This can work especially well when the generated code is very simple, as with the Fourier-Motzkin and multiparametric linear programming methods. In that case however, the performance win may not be worth it, and approximating the floating point values with rationals (see Section 5.1.4) could be a better option.

LP solver for floating point arithmetic The LP solver implemented in `rchoosec` does not work well with floating point types: the code was not written with numerical stability in mind, so very small roundoff error can change what happens when doing Gaussian elimination, and make a theoretically non-singular matrix become singular. In fact, in most of the tests I ran some assertions failed in the middle of the algorithm.

To implement a numerically robust LP solver, techniques like matrix factorization must be used to make sure the algorithm can always follow its course.

Testing simple synthesized code with floating point types On the other hand, synthesized code that does not call a solver can be run without problems with floating point types. This does not mean it always returns correct results. In particular, deciding whether a problem is maximizable or only feasible and bounded can be very tricky. I tested the synthesized code with the Scala `Double` type, but also with the `SmartFloat` type [Darulova and Kuncak 2011]. In the latter, an interval is maintained at all time in which one can be sure the exact value lies.

Consider what happens when synthesizing code for the last example of Section 2 (using multiparametric linear programming). Recall that the exact result for input parameter $a = 2$ is that the problem is bounded but not maximizable, with least upper bound $\frac{464}{45} \approx 10.31$. However, when synthesizing for the Scala `Double` type, and running the code with input 2, the result is $x = 30.5, y = -19.0$. The maximal value is $2x + y = 51$ which is much larger than 10.31, and also blatantly violates the constraint $2x + y \leq 42$. Thus even in this small example, numerical imprecisions can lead to catastrophic consequences !

It turns out such a disaster does not happen when using the Fourier-Motzkin method on the same problem. There is a simple explanation: recall that to solve a LPWS using an LP solver, one needs to add a new variable δ and maximize it over a related LP, with constraint $0 \leq \delta$. The result of the LPWS depends on whether $\delta = 0$, or $\delta > 0$. Thus even a very small imprecision when computing δ can make it become non-zero. In conclusion, the code synthesized using multiparametric linear programming is numerically very unstable if the constraint has strict inequalities, which is not the case for the Fourier-Motzkin method.

Notice that if one synthesizes (still with the multiparametric LP method) the code for `SmartFloats`, and runs it with the same input, the output becomes

```
comparison failed!   uncertainty interval: [-2.2786005886354417E-15,
                                            1.834511378785379E-15]
Total time: 159 ms
List([30.49999999999998,30.50000000000002], [-19.000000000000014,-18.999999999999986])
```

The “comparison failed!” error indicates that it could not be decided what the result of a comparison between two values was, as their intervals overlapped. Thus it can still be caught that something wrong happened !

5.3 Practical application: a rocket controller

A lot of problems in optimal control theory can be reduced to the solving of a linear program [Zadeh and Walen 1962]. Because control laws are typically implemented in embedded hardware (that sometimes has to work under real time constraints), it makes sense to try to specialize the LP solver to the problem at hand in order to generate faster code. In fact, it turns out some of those problems can be written as synthesis problems for linear rational arithmetic, in the sense of this report [Bemporad et al. 2002]. Typically, the parameters are the real world conditions, and the variables describe the control law, i.e what the controller should do in response to those conditions. Code synthesized by a multiparametric LP solver has for example been integrated into a system to maximize the adhesion of a car tire to the road [Borrelli et al. 2001].

In this Section, I describe how I used the methods of this report to synthesize code to make a falling rocket land using the minimal amount of fuel. The problem is also briefly described in both [Bertsimas and Tsitsiklis 1997] and [Matousek and Gärtner 2006].

5.3.1 Setup

I describe the one-dimensional version of the problem. Consider a rocket initially at position $x_0 > 0$ that needs to land at position $x = 0$. The rocket has initial velocity v_0 and acceleration a_0[12]. The rocket is subject to a gravitational acceleration $g < 0$, supposed constant. The rocket's booster can produce an upward acceleration of up to $G > 0$ [13], but can also produce lower accelerations. If the booster produces an acceleration C during a time ΔT, it is assumed $C\Delta T$ units of fuel will be used.

The problem is to land while spending the least amount of fuel [14].

5.3.2 Approximation as a synthesis problem

Continuous time equations Formally, one wants to find a *fuel use distribution* $\{D(t) \mid t \in \mathbb{R}\}$ and a time T minimizing $\int_0^T D(t)\mathrm{d}t$ under the constraints:

1. $0 \leq T$
2. $0 \leq D(t) \leq G$
3. $v = \frac{\mathrm{d}x}{\mathrm{d}t}$
4. $a = \frac{\mathrm{d}v}{\mathrm{d}t}$
5. $a(t) = D(t) + g$
6. $x(t) \geq 0$
7. $x(0) = x_0$
8. $v(0) = v_0$
9. $a(0) = a_0$
10. $x(T) = 0$
11. $v(T) = 0$

I do not know of an analytic solution to those equations. They can however be approximated in discrete time:

Finite discrete time equations Suppose one knows that $T \leq T_{max}$. Then one can sample the time axis into N points $t_0 = 0, t_1, ..., t_{N-1} = T_{max}$, where $t_i = \frac{iT_{max}}{N-1}$. Using the appropriate units, one can rewrite the problem above as follows: find $D \in \mathbb{R}^n$, and $m \in \mathbb{Z}$ minimizing $\sum_{i=0}^{m-1} D_i$ under the constraints

1. $1 \leq m \leq N$
2. $0 \leq D \leq (G, G, ..., G)^T$
3. $x_i = v_i + x_{i-1}$ for $i \in \{1, ..., N-1\}$.
4. $v_i = a_i + v_{i-1}$ for $i \in \{1, ..., N-1\}$.
5. $a_i = D_i + g$ for $i \in \{1, ..., N-1\}$.

[12] The initial acceleration does not influence the rocket in theory, however it may influence the results given by the discrete-time model.

[13] G is assumed to stay constant throughout the fall.

[14] This is essentially what a human player must do in "Lunar Lander"-like video games [Edwards 2009].

6. $x \geq 0$
7. $x_{m-1} = 0$
8. $v_{m-1} = 0$

Notice that if m is fixed, the constraints describe a bounded linear program. If m is unknown, one can solve for the disjunction of the constraints for $m = 1, m = 2, ..., m = N$. In practice, m is not known, but since T_{max} is an approximate bound on the landing time, one knows m should be close to N, so only the constraints for the last M values of m will be synthesized.

In conclusion, the problem is a synthesis problem of linear real arithmetic. In an implementation, the variables can of course be approximated by rationals. The constraints can easily be simplified so that the only variable is the vector D, and the parameters are the initial conditions (i.e x_0, v_0, a_0, G and g).

5.3.3 The controller algorithm

Suppose code for the above problem has been synthesized for a fixed number of time samples N. N denotes the number of variables in the problem, and should typically be small to make the synthesized code fast (the rocket is still falling down while the onboard controller is working !). Thus the approximation to the continuous-time problem may be very poor. To compensate for this fact, the idea is to solve the problem repeatedly as the rocket is falling down, updating the fuel use distribution on the way.

For example, suppose at time zero $G = 20, g = -10, v_0 = a_0 = 0, x_0 = 100$. Call the synthesized code for those values and obtain a first fuel use distribution D_0. The synthesized code took some time to run, so the time is now $t_0 > 0$. One could use this distribution for the rest of the flight, but what one will do instead is to call the synthesized code again with parameters $G, g, v(t_0), a(t_0), x(t_0)$. Only *while* the code is executing will D_0 be used as fuel use distribution. Then the new fuel distribution will be used, while the synthesized code will be called again, and so on.

Typically, the discrete time equations become more accurate as the time before impact gets smaller, so one expects to have more and more accurate solution as the flight goes on. Moreover, even if there was no approximation error in the discrete to continuous approximation, there could still be measurement errors or other unforeseen factors outside the continuous-time model that the method can take into account.

The default fuel use distribution What initial fuel use distribution should be used when the synthesized code is run for the *first* time ? A simple idea is to set $D(t) = 0$ for $t \leq t_c$, and $D(t) = G$ for $t \geq t_c$, where t_c is the critical time at which the rocket must brake to land with zero velocity. It turns out t_c can be computed directly by solving a quadratic equation (the derivation is given in appendix C).

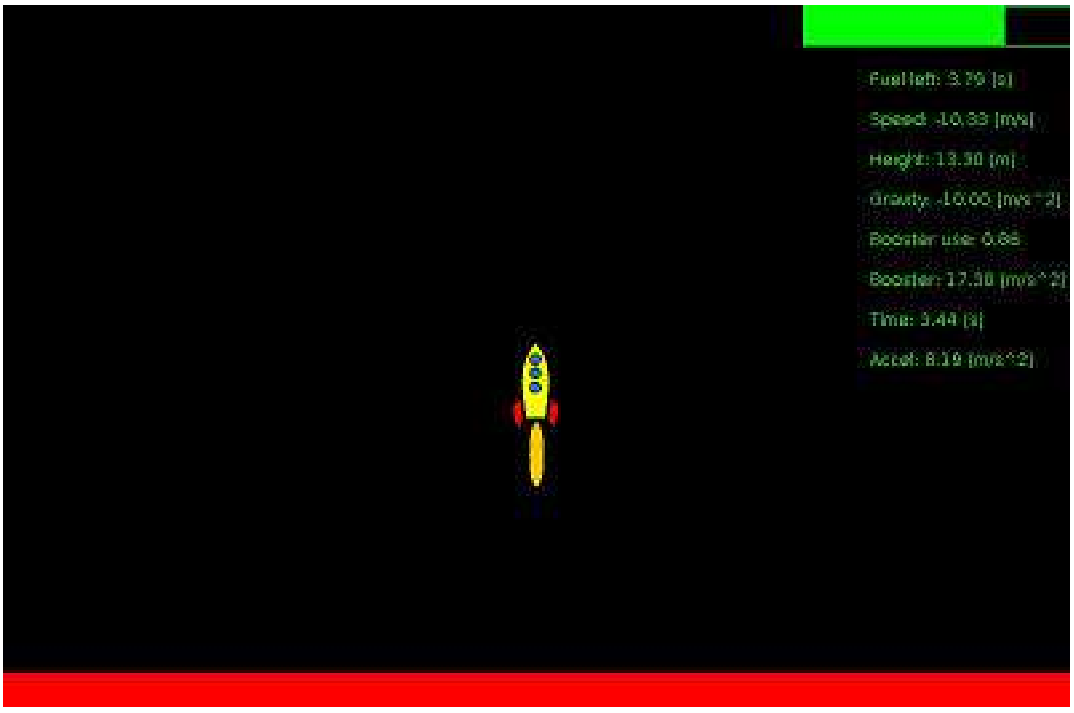
Guessing an impact time The only remaining problem is to find an upper estimate for the landing time T_{max}: even though the synthesis problem itself does not depend on T_{max}, its value is needed to convert the parameters to the right units. Finding a good estimate is important: if T_{max} is too small, the problem will become unfeasible. If it is too large fuel will be wasted to keep the rocket in the air for longer than necessary.

A lower estimate is easy to obtain: for example, one can take the time at which the rocket would crash if the boosters stayed unused, or the time at which it will land under the default fuel use distribution. Both of those estimates can be computed by solving a quadratic equation. Unfortunately, I have not found a clever way of finding a good upper estimate. I ended up implementing an iterative algorithm that starts from a lower estimate and increase it progressively until the problem becomes feasible.

In the full controller algorithm (Algorithm 2), K and λ are constant that have to be fine-tuned. I use $K = 10$ and $\lambda = 1.1$. In practice, it takes time to find a first estimate, but it can then be re-used by the next calls, so very few iterations of the while loop are run.

Algorithm 2 Controller algorithm

```
1: Find a lower estimate $T_0$, or use the estimate made by a previous call to the controller.
2: $T_{max} := T_0$
3: **while** The problem is not feasible for $T_{max}$ and less than $K$ tries have been made **do**
4:     $T_{max} := \lambda T_{max}$
5: **end while**
6: Return the fuel use distribution given by the synthesized code, or the default distribution if the problem is
   still unfeasible after the $K$ tries.
```



```
Fuel left: 3.79 [s]
Speed: -10.33 [m/s]
Height: 13.30 [m]
Gravity: -10.00 [m/s^2]
Booster use: 0.86
Booster: 17.30 [m/s^2]
Time: 3.44 [s]
Accel: 8.19 [m/s^2]
```

Figure 1. Screenshot of the graphical interface for the rocket simulation

As a sanity check, it is also checked at each iteration whether the rocket would crash if one took $D(t) = G$ for all t. If this is the case, no more computations are done and this distribution is used.

5.3.4 Implementation and practical results

To test the above algorithm, I implemented a simple simulator displaying the current state of the rocket in a graphical interface (Figure 1). The initial conditions can be given on the command line, as well as the synthesis parameters: the synthesis method, the number N of sample to take, and the number M of conjunctions to consider. I did most of my tests with $N = 10$, $M = 1$, synthesizing code that runs a solver at runtime.

The simulation is made such that the rocket still falls down while the the fuel use distribution is computed. The time T one simulated second takes in the "real world" can be varied in order to simulate a slower or faster onboard computer. For example, if $T = 0.5$ and it takes one second (on the computer running the simulation) to compute the fuel use distribution, the rocket would have fallen for 2 (simulated) seconds in the meantime.

The variables in the simulation all have the Scala type `Double`, but they are converted to rationals with small denominator (see Section 5.1.4) when calling the synthesized code.

Table 1. Performance of the `rchoosec` LP solver on selected Netlib problems (times are in seconds)

Problem	Time (phase 1)	# of iterations (phase 1)	Time (phase 2)	# of iterations (phase 2)	Total time
`itest2`	0.1	2	-	-	0.1
`galenet`	0.2	2	-	-	0.2
`itest6`	0.2	2	-	-	0.2
`afiro`	1.4	23	1.0	18	2.4
`kb2`	5.4	100	4.9	100	10.3
`boeing2`	298.0	440	347.0	894	645.0

My tests have shown that a rocket controlled by Algorithm 2 manages to land successfully. However, I have also compared what is spent in that case with the fuel spent if one only uses the default fuel distribution. In most of the simulations using the default distribution lead to slightly less or the same amount of fuel being used in the end. Thus it seems too many approximations are done to make the method useful in practice. However the simulation is still a good benchmark to test code synthesized by different methods in a "real use" scenario.

5.4 Performance measurement

In this Section, I describe the tests I have run to measure the performance of the synthesis methods described in this report and of the synthesized code.

5.4.1 Performance of the simplex solver

To measure the performance of my simplex solver, I used example linear programs publicly-available from the Netlib repository[15]. The Netlib collection contains linear programs of practical or theoretical interest donated by industrial companies or academics [16]. The size of the constraint matrices for the problems available range from 10×4 to 6331×22275. Most problems have more than 100 variables.

My exact solver takes a reasonable time only on the smallest examples, so I decided to use the problems `itest2` (10×4), `galenet` (9×8), `itest6` (12×8), `afiro` (28×32), `kb2` (44×41), and `boeing2` (167×143). The first three are unfeasible problems (I did not find feasible problems of similar size), and the last three are feasible and bounded. To test my solver, I converted the problem input files to the `RChoose` language, and ran `rchoosec` to synthesize code without parameters.

The time spent by the solver and the number of pivot steps used for each problem is summarized in table 1. I separate the time spent for phase 1 (making the constraint matrix full rank and finding an initial roof) and phase 2 (calling Algorithm 1). The roof LP is the auxiliary linear program one needs to solve to find an initial roof. It turns out the first three problems can be seen to be unfeasible already at that point, so the solver never enters phase 2. All the times in this report were measured on a GNU/Linux system with an Athlon 5000+ processor and 2 Gb of RAM. Note that since I use some datastructures that do not behave exactly the same from one run of the program to another, the number of simplex iterations one obtains can vary from run to run (a different ordering of the matrix line can lead to a different number of iterations).

One sees that phase 1 takes about half the total solving time. In fact, row-reducing a matrix is slow in my implementation, whereas one simplex iteration is done relatively quickly. The performance of synthesizing code calling a solver versus calling a solver directly are discussed in Section 5.4.3. Based on the data above,

[15] `http://www.netlib.org/`

[16] For example, the `boeing2` problem used for my benchmark "has to do with flap settings on aircraft for economical operations" according to the Netlib `README`.

Table 2. Description of synthesis problems

Problem	m	n	r	R	s
`itest2`	5	1	3	3	0.0
`galenet`	8	6	2	2	0.7
`itest6`	9	7	1	1	0.8
`afiro`-feas	28	7	25	9	0.6
`afiro`	28	7	25	9	0.6
`kb2`-feas	44	11	30	16	0.0
`kb2`	44	11	30	16	0.0
Rocket, $N = 5$	15	4	5	4	0.3
Rocket, $N = 10$	30	9	5	4	0.7
Rocket, $N = 40$	120	39	5	4	0.9

one can conjecture that the synthesized code will take about twice less time, since phase 1 (and computing the inverse of A_B in phase 2) is done at synthesis time.

Another observation is that solving LPs exactly is not feasible in practice, except for "small" sizes: the `boeing2` problem with about 100 variables shows the limit of my solver, but nowadays LPs with "only" a few thousands of variables and constraints are considered small [Fourer 2000] . Moreover, my implementation is slow even compared to other *exact* solvers. For example the solver `exlp` [17] solves `boeing2` almost instantly on the test machine.

Another difficulty that arises is that rational numbers may grow in size as the problem gets solved. The final solution of `kb2` has rationals with about 30 decimal digits in their numerator and numerator. Surprisingly, the solution of `boeing2` has rationals of reasonable size (less than 10 digits for both the numerator and denominator).

5.4.2 Performance of the synthesis methods

To measure how fast the various methods presented in this report can synthesize code, I parametrized the first five Netlib problems presented in the previous Section by considering some of their variables as parameters, and removing some constraints from the unfeasible one to allow feasibility. When an objective function was specified, both the problem without objective function and the original problem were synthesized. I also synthesized the rocket controller synthesis problem described in Section 5.3.2 with different numbers N of samples (with $M = 1$ as the number of disjunction).

The synthesized problems are described in table 2, and the results are given in table 3. A problem name with suffix "-feas" means its objective function has been removed. I have called m, n, r the number of constraint, variables, and parameters respectively. R denotes the column rank of the matrix F when one writes the problem as a multiparametric LP. s is defined as the number of zero entries of the constraint matrix (after the change of coordinate to make it full rank) divided by the total number of elements. This gives an indication of how sparse the resulting matrix is. Whenever four numbers separated by commas appear, they refer to the result for each of the synthesis methods: the first number is the result for the "pure" Fourier-Motzkin method as described in Section 3.4, the second to the Fourier-Motzkin method for optimization (Section 4.3), the third to the method of calling a solver at runtime (Section 4.4), and the last to the multiparametric LP method (Section 4.5). The indicated compiled code size is for code compiled with the Scala compiler *without* optimization. The column "# of regions" gives the number of critical region the parameter space was split into when applying the multiparametric LP algorithm.

[17] `http://members.jcom.home.ne.jp/masashi777/exlp.html`

Table 3. Performance of synthesis methods (times are in seconds, sizes in kilobytes)

Problem	Synthesis time	Source code size	Compiled code size	# of regions
`itest2`	$2, 3, 2, 3$	$2, 2, 2, 2$	$8, 8, 10, 7$	1
`galenet`	$2, 3, 3, 3$	$2, 2, 3, 3,$	$6, 7, 11, 7$	4
`itest6`	$2, 3, 3, 3$	$2, 2, 4, 2$	$6, 7, 13, 7$	1
`afiro`-feas	$6, 5, 4, 20$	$13, 13, 13, 146$	$22, 21, 25$,?	70
`afiro`	?,$5, 4, 14$	-,$15, 13, 76$	-,$27, 25, 53$	30
`kb2`-feas	$14, 16, 7, 24$	$563, 386, 52, 533$	?,414*, 100, ?	10
`kb2`	?, $16, 6, 26$	-,$387, 52, 533$	-,414*, 101, ?	10
Rocket, $N = 5$	?, $20, 3, 20$	-,$8, 4, 26$	-,$18, 12, 21$	14
Rocket, $N = 10$	?, ?, 4, ?	-, -, 9,-	-, -, 20, -	> 3750
Rocket, $N = 40$	?, ?, 10, ?	-, -, 68, -	-, -, 95, -	> 3750

Whenever a time is indicated as "?", this means the program was aborted after it ran for more than 15 minutes. A compiled code size of "?" indicates that the compiler crashed when compiling the code. This is due to a limit in the size of a method in Java [18]. A workaround it to split the synthesized method into several smaller ones, but I have not implemented it: it is not a priori obvious when a method becomes too large or how to split it. Likewise, a compiled code size followed by a star ($*$) means the code was correctly compiled, but cannot be run because of a `java.lang.ClassFormatError: Invalid method Code length` exception.

Note that the multiparametric LP code uses a default maximizer even for problems without objective function. If this is not done, the number of regions becomes 115 for `apiro`-feas, and 21 for `kb2`-feas .

Observe that the "pure" Fourier-Motzkin method cannot in practice be used for non-trivial optimization problems, and the improved version must be preferred instead.

As should be expected, calling a solver at runtime produces smaller code for non-trivial problems than the alternatives. The tradeoff in synthesized code performance will be discussed in the next Section. Code size for the two version of Fourier-Motzkin elimination are similar (the only difference between the two methods when there are no objective vector is that for the improved one, a change of coordinate is first done to make the matrix A full rank). Notice that the change of coordinate reduced code size in `kb2`-feas. Despite theoretical advantages (i.e better computational complexity), in practice it seems that multiparametric LP is more or less on par with the Fourier-Motzkin method as far as compiled code size is concerned.

The parametrized version of `apiro` and `kb2` can be observed to have very few decision regions. This could partially be explained by the fact that their parameter matrix F has a relatively low rank (compared to their number of parameters).

On the other hand it seems the rocket synthesis problems are much harder than their Netlib counterpart for a comparable size, as they can be synthesized only using the simplex method as soon as their size becomes reasonable. I have not found a satisfactory explanation for this fact (sparsity does not seem to influence the results much).

5.4.3 Performance of the generated code

Since the Netlib problems were not specifically tailored for synthesis, it is difficult to know how to meaningfully test the resulting generated codes. Hence I chose to only test the generated code for the Rocket problems. The results are given in table 4.

[18] The exact exception is `ch.epfl.lamp.fjbg.JCode$OffsetTooBigException: offset too big to fit in 16 bits`. See `https://issues.scala-lang.org/browse/SI-1133` for a related bug.

Table 4. Performance of synthesized code (times are in seconds)

Problem	Time dilatation	# of calls	Avg solving time	Max solving time
Rocket, $N = 5$, FM	1	810	0.002	0.10
Rocket, $N = 5$, MPLP	1	880	0.001	0.07
Rocket, $N = 5$, simplex	1	$154, 347, 661$	$0.03, 0.003, 0.002$	$0.41, 0.06, 0.12$
Rocket, $N = 10$, simplex	2	$130, 378, 4227$	$0.09, 0.02, 0.002$	$0.23, 0.66, 0.32$
Rocket, $N = 40$, simplex	8	$6, 23, 4254$	$7.03, 2.29, 0.02$	$8.37, 3.29, 2.01$

The code was tested in a rocket simulation with initial conditions $x_0 = 500, g = -10, G = 20, v_0 = a_0 = 0$. The "time dilatation" column indicates how many real seconds elapsed during one simulated second. The "# of calls" column indicates the number of times the code was called during the simulation, and the next columns give the average and maximum time it took to terminate. As indicated in the previous Section, I could only use the Fourier-Motzkin and multiparametric LP synthesis methods when taking $N = 5$. For $N = 10$ and $N = 40$, only calling a solver at runtime ended-up being feasible. The three numbers in each column of "simplex" lines indicate the result when, in order

1. A solver was called directly (i.e no synthesis was done at all).
2. Code calling a solver at runtime was synthesized, but the solver does not remember the previous optimal roof.
3. Code calling a solver at runtime was synthesized, and the solver remembers the previous optimal roof.

The maximal solving times are not to be taken too literally: they can vary quite a lot from one run to another.

According to the first two lines it seems the Multiparametric LP method generated better code than the Fourier-Motzkin method, but more results are needed to judge that fact more carefully. One can also see that calling a solver at runtime (with all the proper optimizations) is *not* slower on average than code synthesized using other methods. The maximum solving time seems larger, however.

The results of the last three lines show that synthesizing the code calling the solver pays off: because phase 1 is done at compile-time, one obtains a large improvement for the small problems, and an improvement of about a factor 3 for the large one. Remembering the last optimal roof leads to dramatic improvements for that particular synthesis problem. Indeed, the rocket's position, velocity and acceleration do not change much from one call to the next, hence the optimal basis will not vary much. One sees improvements of one or two orders of magnitudes in the average solving time. Still, this optimization cannot do anything to improve the maximal solving time.

6. Conclusion and possible extensions

As the content of this report shows, I have successfully implemented several synthesis methods for linear rational arithmetic. It seems that calling an LP solver at runtime becomes the only viable solution as the constraint formula gets larger. Some precomputations can be done at synthesis-time, and it may still be possible to partially evaluate the simplex algorithm more aggressively.

More testing is needed to really evaluate the relative performance of the various methods, in particular it is still not clear how calling a solver at runtime compares to code synthesized via multiparametric linear programming or Fourier-Motzkin elimination for large problems (whenever those methods terminate in a reasonable amount of time).

Even though it must be considered only as a proof of concept, the rocket controller example presented in Section 5.3 shows synthesis has applications to real world problems.

In the rest of this Section, I present some possible extensions to the work presented in this report.

Better elimination of quantifiers and negations Because they do not occur often in practice, I did not give much thought to the handling of quantifiers and negations; they are currently eliminated using the Fourier-Motzkin method. However, the results of appendix D suggests there may exist more efficient ways, at least for some special cases. For example, one should keep a negation of the form $\neg(a = b)$ as it is instead of introducing disjunctions by rewriting it as $a < b \vee b < a$.

Better choice of default synthesis method One could modify `rchoosec` to try various synthesis method in order, and return the result of the best one that succeeded. For example, it could first try the pure Fourier-Motzkin method for one minute, then if it still hasn't succeeded abort and try the improved Fourier-Motzkin method for one minute, and so on. This could of course also be parallelized.

Improving the simplex solver Writing a good simplex solver is hard: thousands of free or commercial ones have been written, and no "canonical" implementation has emerged [19]. Still, it would be interesting to know how far one can go with an exact solver written in Scala. Of course, one could try improving the solver so that it is more or less numerically stable, i.e so that it can also handle floating point inputs. Another option would be to make `rchoosec` simply call an external solver.

Integration into a compiler Currently, integrating synthesized code into a larger program is cumbersome: one has to write the `RChoose` file, synthesize it, then add an import statement in the larger program. Thus it would be a good idea to write a Scala compiler plugin to make the integration seamless, as was done in Comfusy [Kuncak et al. 2010]. Given the trouble I had when compiling non-trivial generated code using the Scala compiler, one may also consider generating code in other languages that have more robust compilers available, like C.

Improving the rocket controller As explained in Section 5.3, I have not found an elegant way to estimate a good upper bound to the impact time. One idea is to try to generate some smooth, increasing, fuel use distribution and use the resulting landing time as an estimate. More generally, it may be possible to solve the continuous time equations given in 5.3.2, or at least to find some "good" distribution that works well in practical cases. For example, it would be interesting to know how close to the optimum the "all or nothing" distribution used as default in the current controller algorithm is.

On a different register, currently nothing special is done if there is not enough fuel to use a given distribution. However, one could write a linear program to minimize the speed on impact and add the additional fuel constraint. This could then be integrated into the controller algorithm.

Another fun improvement would be to implement the possibility of taking manual control of the rocket at any time during the simulation. This would make the experience more interactive.

Mixed rational/integer synthesis A natural extension is to restrict some of the variables to be integers. The problem then becomes much harder (integer programming is NP-hard), but solvers still exists, and a quantifier elimination procedure for the theory can probably also be found.

Extending the RChoose syntax As explained in appendix D, introducing disjunctions in the constraint formula can make the synthesis problem much harder. One can try to avoid that the user uses disjunctions by introducing some commonly-used functions like `max`, `min` or `abs` in the `RChoose` language. Clearly, those constructs do not add more power to the language, because e.g `abs(x)` can always be eliminated by replacing it with a new variable y, adding the constraints $(y = x \vee y = -x) \wedge y \geq 0$. Similarly, one can write $\max(x, y) = -\min(-x, -y) = \frac{|x-y|+x+y}{2}$. Moreover a variable x can still be restricted to $\{0, 1\}$ with the

[19] [Fourer 2000] has more information on available LP solvers.

constraint $|2x - 1| = 1$, so the corresponding decision problem stays NP-hard. However, in a lot of practical cases, one can rewrite the constraint as a linear program, i.e without introducing any disjunction.

For example, the problems of finding the largest ball contained in a polyhedron, or the best fitting line according to the 1-norm fall in this category [Matousek and Gärtner 2006]. In the rocket controller example, one may want to minimize the maximal acceleration rather than the amount of fuel spent (although it is clear in that case that the solution is simply a uniform fuel distribution), and it turns out this can also be rewritten as a linear program.

Thus it would be interesting to integrate some re-writing rules for `max`, `min` and `abs` to `rchoosec` , so that if an equivalent linear program exists it is found (a simple example of such a rewriting rule is that the constraint $\max(x, y) \leq U$ is equivalent to $x \leq U \wedge y \leq U$).

Acknowledgments

I would like to thank Viktor Kuncak for his helpful comments and ideas. I also thank Eva Darulova, Ali-Sinan Köksal, and Philippe Suter for interesting discussion and feedback.

Last but not least, I am most grateful to Friedrich Eisenbrand for his very clear explanations on multiparametric linear programming.

References

A. Bemporad, F. Borrelli, and M. Morari. Model predictive control based on linear programming – the explicit solution. *IEEE Transactions on Automatic Control*, 47(12):1974–1985, 2002.

D. Bertsimas and J. N. Tsitsiklis. *Introduction to linear optimization*. Athena Scientific, 1997.

F. Borelli, A. Bemporad, and M. Morari. Geometric algorithm for multiparametric linear programming. *Journal of optimization theory and applications*, 118(3):512–540, 2003.

F. Borrelli, A. Bemporad, M. Fodor, and D. Hrovat. A hybrid approach to traction control. In *Hybrid Systems*, pages 162–174, 2001.

E. Darulova and V. Kuncak. On the Design and Implementation of SmartFloat and AffineFloat. Technical report, 2011.

B. Edwards. Forty years of lunar lander, 2009. URL `http://technologizer.com/2009/07/19/lunar-lander/`. [Online; accessed 5-June-2011].

F. Eisenbrand. Course notes for the discrete optimization course. 2011. URL `http://disopt.epfl.ch/files/content/sites/disopt/files/shared/Opt2011/lecture_115.pdf`.

F. Eisenbrand, N. Hähnle, A. Razborov, and T. Rothvoss. Diameter of Polyhedra: Limits of Abstraction. *Mathematics of Operations Research*, 35(4):786–794, 2010. ISSN 1526-5471. doi: 10.1287/moor.1100.0470.

R. Fourer. Linear programming frequently asked questions, 2000. URL `http://www.neos-guide.org/NEOS/index.php/Linear_Programming_FAQ`. [Online; accessed 6-June-2011].

T. Gal and J. Nedoma. Multiparametric linear programming. *Management Science*, 18(7):406–422, 1972.

N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prenctice Hall International, 1993.

S. C. Kleene. *Introduction to Metamathematics*. New York: Van Nostrand, 1952.

V. Kuncak, M. Mayer, R. Piskac, and P. Suter. Complete functional synthesis. In *PLDI*, 2010.

S. Lang. *Algebra*. Addison-Wesley Pub. Co,, 3rd edition, 1993.

J. Matousek and B. Gärtner. *Understanding and using linear programming*. Springer, 2006.

D. Monniaux. The pitfalls of verifying floating-point computations. *ACM Transactions on Programming Languages and Systems*, 30:1–41, 2007. doi: 10.1145/1353445.1353446.

J. Robinson. Definability and decision problems in arithmetic. *The Journal of Symbolic Logic*, 14(2):98–114, 1949.

A. Schrijver. *Theory of linear and integer programming*. Wiley, 1986.

M. I. Schwartzbach. Lecture Notes on Static Analysis. 2008. URL `http://www.brics.dk/~mis/static/static.pdf`.

D. A. Spielman and S.-H. Teng. Smoothed analysis of algorithms: why the simplex algorithm usually takes polynomial time. In *ACM Symposium on Theory of Computing*, pages 296–305, 2001. doi: 10.1145/380752.380813.

W. A. Stein et al. Sage mathematics software, 2011. URL `http://www.sagemath.org`.

A. Tarski. A decision method for elementary algebra and geometry. RAND Corporation, 1951.

P. Tøndel, T. Johansen, and A. Bemporad. Computation and approximation of piecewise affine control laws via binary search trees. In *Decision and Control*, volume 3, pages 3144–3149, 2002.

L. A. Zadeh and L. Walen. On optimal control and linear programming. *IEEE Transactions on Automatic Control*, 7: 45–46, 1962.

Appendices

A. Grammar for the RChoose language

$$
\begin{array}{rcl}
\textit{Goal} & \to & \textbf{RChoose}\ (\ \texttt{<IDENTIFIER>}\ (,\ \texttt{<IDENTIFIER>}\ )^{*})([\ \textit{AExpr}\ ])?\{\ \textit{BExpr}\ \}\ \texttt{<EOF>} \\
\textit{BExpr} & \to & \textit{BExpr}\ ||\ \textit{BExpr} \\
\textit{BExpr} & \to & \textit{BExpr}\ \&\&\ \textit{BExpr} \\
\textit{BExpr} & \to & \textbf{!}\ \textit{BExpr} \\
\textit{BExpr} & \to & \textit{AExpr}\ (<\ \mid\ <=\ \mid\ ==)\ \textit{AExpr} \\
\textit{BExpr} & \to & (\ \textbf{forall}\ \mid\ \textbf{exists}\ )(\ \texttt{<IDENTIFIER>}\ ,\textit{BExpr}\ ) \\
\textit{BExpr} & \to & (\ \textit{BExpr}\ ) \\
\textit{BExpr} & \to & \textbf{true}\ \mid\ \textbf{false} \\
\textit{AExpr} & \to & \textit{AExpr}\ (+\mid\ -\ \mid / \mid *)\ \textit{AExpr} \\
\textit{AExpr} & \to & (\ \textit{AExpr}\ ) \\
\textit{AExpr} & \to & \texttt{<IDENTIFIER>} \\
\textit{AExpr} & \to & \texttt{<INTEGER>}
\end{array}
$$

- `<IDENTIFIER>` represents a sequence of letters, digits and underscores, starting with a letter and which is not a keyword. Identifiers are case-sensitive.
- `<INTEGER>` represents a sequence of decimal digits, with no leading zeros (the sequence can have arbitrary length).
- `<EOF>` represents the special end-of-file character.

Note that a file respecting this grammar may still not be a valid `RChoose` file, e.g because it has an expression of the form `x*y`: the *AExpr* nodes should be linear arithmetic expressions.

B. Bounds on the denominator of a rational midpoint

Lemma 7. Let $x, y \in \mathbb{R}$, and suppose $x \leq y - 1$. Then $\lceil x \rceil \leq \lfloor y \rfloor$.

Proof. Clearly, $\lceil x \rceil \leq y$. Because $\lceil x \rceil$ is an integer, the result follows. □

Theorem 4. Let $p_1, p_2 \in \mathbb{Z}_{\geq 0}, q_1, q_2 \in \mathbb{Z}_{>0}$, such that $\gcd(p_1, q_1) = \gcd(p_2, q_2) = 1$ and $\frac{p_1}{q_1} < \frac{p_2}{q_2}$. Let q be the smallest integer for which there is a $p \in \mathbb{Z}_{>0}$ such that

$$\frac{p_1}{q_1} < \frac{p}{q} < \frac{p_2}{q_2} \tag{27}$$

Then q satisfies

$$\left\lceil \frac{q_1 + q_2}{p_2 q_1 - p_1 q_2} \right\rceil \leq q \leq \left\lceil \frac{q_1 q_2 + q_1 + q_2}{p_2 q_1 - p_1 q_2} \right\rceil \tag{28}$$

Proof. Notice first that the two bounds are always well defined, as $\frac{p_1}{q_1} < \frac{p_2}{q_2}$ implies that $p_2 q_1 - p_1 q_2 > 0$. Moreover, it has already been explained in Section 5.1.4 that (27) holds if and only if q satisfies

$$\left\lceil \frac{p_1 q + 1}{q_1} \right\rceil \leq \left\lfloor \frac{p_2 q - 1}{q_2} \right\rfloor \tag{29}$$

Thus

$$\begin{aligned} 0 \geq \left\lceil \frac{p_1 q + 1}{q_1} \right\rceil - \left\lfloor \frac{p_2 q - 1}{q_2} \right\rfloor &\geq \frac{p_1 q + 1}{q_1} - \frac{p_2 q - 1}{q_2} \\ = \frac{p_1 q_2 q + q_2 - p_2 q_1 q + q_1}{q_1 q_2} &= \frac{q(p_1 q_2 - p_2 q_1) + q_1 + q_2}{q_1 q_2} \end{aligned}$$

Rearranging terms, one obtains the lower bound in (28). To check the upper bound, observe that, as a consequence of Lemma 7, a *sufficient* condition for (29) to hold is

$$\frac{p_1 q + 1}{q_1} \leq \frac{p_2 q - 1}{q_2} - 1$$

Rearranging terms, one obtains

$$q \geq \frac{q_1 q_2 + q_1 + q_2}{p_2 q_1 - p_1 q_2}$$

Any q satisfying this bound satisfies (27). Thus the smallest such q is *at most* $\left\lceil \frac{q_1 q_2 + q_1 + q_2}{p_2 q_1 - p_1 q_2} \right\rceil$. This completes the proof.

□

C. Finding the critical braking time

C.1 Description of the problem

A rocket is falling from position $x_0 > 0$ with initial velocity v_0, at constant acceleration $-g$. One wants to brake with acceleration $G > g$, starting at some critical time $t_c \geq 0$, until landing, such that one arrives at position 0 with velocity 0. One wants to find t_c and the time of arrival t_a at which $x(t_a) = 0$ and $v(t_a) = 0$.

C.2 Solution

When $t \leq t_c$, the rocket's position is described by the equation

$$x(t) = -\frac{1}{2}gt^2 + v_0 t + x_0 \tag{30}$$

Similarly, when $t > t_c$, the position is given by

$$x(t) = \frac{a}{2}(t - t_c)^2 + (t - t_c)v_{t_c} + x_{t_c} \tag{31}$$

Where $a := G - g$, and $x_{t_c} := x(t_c)$, $v_{t_c} := v(t_c) = -gt_c + v_0$ can be obtained directly from (30). Deriving (31), one obtains that for $t \geq t_c$

$$v(t) = a(t - t_c) + v_{t_c} \tag{32}$$

Hence $v(t_a) = 0$ implies $a(t_a - t_c) + (-gt_c + v_0) = 0$, i.e $t_c = \frac{v_0 + at_a}{a+g} = \frac{v_0 + (G-g)t_a}{G}$. Plugging this expression for t_c into (31) and setting $t = t_a$, one obtains a quadratic polynomial $x(t_a) := \alpha t_a^2 + \beta t_a + \gamma$ in t_a. To avoid dying of boredom, I computed the coefficients α, β, γ using SAGE [Stein et al. 2011], and obtained

$$\begin{aligned}\alpha &= \frac{g^2}{2G} - \frac{g}{2}\\ \beta &= -\frac{gv_0}{G} + v_0\\ \gamma &= \frac{v_0^2}{2G} + x_0\end{aligned}$$

Solving the quadratic equation $x(t_a) = 0$, and taking the positive solution, one can obtain a value for t_a, and thus a value for t_c.

As a sanity check, suppose $v_0 = 0, G = 2g$. Then one obtains $\alpha = \frac{g^2}{4g} - \frac{g}{2} = -\frac{g}{4}, \beta = 0, \gamma = x_0$. Solving the equations, it turns out $x(t_c) = \frac{x_0}{2}$, which confirms the intuition that the braking should occur exactly in the geometric middle of the fall.

D. Efficiency of decision for linear rational arithmetic

In this appendix, I discuss the computational efficiency of deciding various versions of linear rational arithmetic.

Most of the results can be generalized to other ordered fields, but the fact that there is a polynomial time algorithm to solve linear programs is specific to the rationals, so Theorem 7 may not hold for other fields.

Theorem 5. Deciding a formula of linear rational arithmetic is NP-Hard.

Proof. 0-1 integer programming (where the variables are constrained to be either zero or one) is well known to be NP-Hard [Schrijver 1986]. Any problem of 0-1 integer programming with variables $x_1, ..., x_n$ and constraint $Ax \leq b$ can be expressed in rational linear arithmetic with the formula

$$\exists x_1 \exists x_2 ... \exists x_n Ax \leq b \wedge ((x_1 = 0) \vee (x_1 = 1) \wedge ... \wedge (x_n = 0 \vee x_n = 1))$$

Therefore deciding a formula of linear rational arithmetic is at least as hard as 0-1 integer programming, which is NP-Hard. □

Disallowing disjunctions does not fully solve the problem:

Theorem 6. Deciding a formula of linear rational arithmetic without disjunctions is NP-Hard.

Proof. The constraints of the form $(x_i = 0 \lor x_i = 1)$ in the proof of Theorem 5 can be equivalently expressed without disjunction as

$$\neg(\neg(x_i = 0) \land \neg(x_i = 1))$$

□

The problem was that negating a conjunction still results in a disjunction. What is needed is to force the formula to have a *negation-normal form* without disjunctions. One can then suppose the formula ϕ to be of the form

$$\phi = Q_1 x_1, ..., Q_n x_n \psi$$

with

$$\psi = (\psi_1 \land \psi_2 ... \land \psi_m)$$

Where $Q_i \in \{\forall, \exists\}$, and each $\psi_i[x_1, ..., x_n]$ is in one of the three following forms:

$$\begin{aligned} \psi_i &= f(x_1, ..., x_n) \leq C_i \\ \psi_i &= f(x_1, ..., x_n) < C_i \\ \psi_i &= f(x_1, ..., x_n) \neq C_i \end{aligned}$$

with f a linear function, $C_i \in \mathbb{Q}$.

The next Theorem shows that imposing this restriction is enough.

Theorem 7. Deciding a formula ϕ of linear rational arithmetic that has a negation-normal form without disjunctions can be done in polynomial time.

To prove the Theorem, I first need several lemmas.

Lemma 8. If $Q_n = \forall$, then either ϕ is equivalent to $Q_1 x_1, ..., Q_{n-1} x_{n-1} \psi$ or ϕ is false. Which of these alternatives is true can be decided in polynomial time.

Proof. If x_n does not appear in any ψ_i, the first alternative is true. Otherwise, once all the variables $x_1, ..., x_{n-1}$ are fixed, it is clearly always possible to find an x_n that violates one of the ψ_is in which it appears, so ϕ is false. □

Lemma 9. Suppose $Q_k = \forall$, $Q_{k+1} = ... = Q_n = \exists$. Then either $\forall x_1 ... \forall x_k \exists x_{k+1} ... \exists x_n \psi$ is true, ϕ is equivalent to $Q_1 x_1 ... Q_{k-1} x_{k-1} Q_{k+1} x_{k+1} ... Q_n x_n \psi[0/x_k]$, or ϕ is false. Which of these alternatives is true can be decided in polynomial time.

Proof. Let $\Psi[x_1, ..., x_k] := \exists x_{k+1} ... \exists x_n \psi$. If the first alternative is not true, this means there exists $y_1, ..., y_k \in \mathbb{Q}$ such that $\Psi[y_1/x_1, ..., y_k/x_k]$ is false. Apply Fourier-Motzkin elimination to ψ to eliminate the variables $x_{k+1}, ..., x_n$. The result is a formula $\hat{\psi}[x_1, ..., x_k]$ such that $\Psi \equiv \hat{\psi}$. If x_k does not appear in $\hat{\psi}$, the second alternative is true. Otherwise, ϕ is equivalent to $Q_1x_1...Q_{k-1}x_{k-1}\forall x_k\hat{\psi}$, and x_k appears in $\hat{\psi}$. By the proof of Lemma 8, ϕ must be false.

An efficient way to test whether x_k appear in $\hat{\psi}$ is to first find values $z_1, ..., z_{k-1}, z_{k+1}, ..., z_n$ such that

$$\psi[z_1/x_1, ..., y_k/x_k, z_{k+1}/x_{k+1}, ..., z_n/x_n]$$

is true (if no such values exists, this means $\exists x_1, ..., \exists x_{k-1}\forall x_k\exists x_{k+1}...\exists x_n\psi$ is false, so definitely ϕ is false as well), then check whether $\forall x_k\Psi[z_1/x_1, ..., z_{k-1}/x_{k-1}]$ is true. If it is false, then $\hat{\psi}$ must contain x_k. If it is true, it cannot contain it. The proof of Theorem 7 explains how to do these operations in polynomial time. □

Proof sketch for Theorem 7. lemmas 8 and 9 show one can assume without loss of generality that $Q_1 = ... = Q_r = \forall$, and $Q_{r+1} = ... = Q_n = \exists$ for some r.

I will describe only what happens in two special cases:

1. All the quantifiers are existential, and the only relations used are $\leq$ and $\neq$
2. Both types of quantifiers are used, but the only relation used is $\leq$.

All throughout the proofs I will use the fact that feasibility of a linear program can be decided in polynomial time [Schrijver 1986]. The ideas presented for those cases should carry over to the general case. Notice for example that if the only quantifiers are existential, and no $\neq$ appear, then the problem is equivalent to deciding whether a LPWS has a solution. As shown in Section 4.4.2, this is essentially equivalent to solving a linear program, which can be done in polynomial time.

Finally, Corollary 1 suggests it should not be too difficult to adapt the methods presented below if one additionally wants to maximize some objective function.

D.1 Solving a LP with negations

The first case is equivalent to deciding the feasibility of a linear program $Ax \leq b$ with additional constraints $c_1x \neq d_1 \wedge ...c_mx \neq d_m$ for some matrix C and vector d. Let $H_1, ..., H_m$ be the hyperplanes defined as $H_i := \{x \mid c_ix = d_i\}$, and let $P := \{x \mid Ax \leq b\}$. Then one wants to find a point x^* in $P - \bigcup_{i=1}^m H_i$. Such a point exists if and only if $P \nsubseteq \bigcup_{i=1}^m H_i$ i.e if and only if $P \cap (\bigcup_{i=1}^m H_i) \neq P$.

To find x^* (if it exists), first define x_i^* to be a point in $P - H_i$. Such a point exists if and only if one of the sets $\{x \mid Ax \leq b \wedge c_ix < d_i\}$, $\{x \mid Ax \leq b \wedge c_ix > d_i\}$ is non-empty, which can be checked by solving the two corresponding LPWS. I claim x^* exists if and only if $x_1^*, ..., x_m^*$ exists. To see this, define inductively $y_1 := x_1^*$, and

$$y_k := \begin{cases} y_{k-1} & \text{if } c_ky_{k-1} \neq d_k \\ \frac{\lambda y_{k-1}+(1-\lambda)x_k^*}{2} & \text{otherwise} \end{cases}$$

Where $0 < \lambda \leq 1$ must be chosen sufficiently small, so that $c_iy_k \neq d_i$ for all $i \in \{1, ..., k-1\}$. This is always possible, as there are only finitely many hyperplanes, but infinitely many choices for λ. I show by induction that $y_k \in P - \bigcup_{i=1}^k H_i$ from which it will follow that $x^* = y_m$ is a solution. The base case follows from the definition of x_1^*. For the induction step, observe first that because P is convex, $y_{k-1} \in P$ (by induction), and $x_k^* \in P$ (by construction), $y_k \in P$. The fact that $y_k \notin \cup_{i=1}^k H_i$ also follows from its construction.

To find out whether each x_i^* exists, one needs to solve at most $2m$ LPWS, each of which can be solved in polynomial time, so the initial constraint can also be decided in polynomial time.

D.2 Solving a quantified LP

The second case is equivalent to deciding when a multiparametric linear program $Ax \le w + F\Theta$ ($x \in \mathbb{Q}^n, A \in \mathbb{Q}^{m\times n}, F \in \mathbb{Q}^{m\times r}, \Theta \in \mathbb{Q}^r, w \in \mathbb{Q}^m$) is feasible for *all* values of Θ. This is the case if and only if the dual

$$\min\{(w + F\Theta)^T y : A^T y = 0, y \ge 0\} \tag{33}$$

is feasible and bounded for all parameter values of Θ. Feasibility of the dual does not depend on Θ, and can be checked in polynomial time by solving a linear program. If the dual is feasible, the only thing that can happen is that it is unbounded. This is the case if and only if there exists $d \in \mathbb{Q}^m$ such that

$$A^T d = 0, d \ge 0 \tag{34}$$

and for some Θ^*, $(w + F\Theta^*)^T d < 0$ (because if such a d exists, λd is also feasible for arbitrarily large values of λ). Thus for (33) to always be bounded, one needs that for all Θ and all d satisfying (34), $C_{\Theta,d} := (w + F\Theta)^T d \ge 0$. From the special case where $\Theta = 0$, it follows that $w^T d \ge 0$.

Moreover, suppose that for some Θ, $(F\Theta)^T d > 0$. Then for some $\lambda > 0$ sufficiently large, one has that $(w + F(-\lambda)\Theta)^T d < 0$. This implies that $(F\Theta)^T d = 0$, for all Θ, i.e d is in the orthogonal complement $F^\perp$ of F. By the fundamental theorem of linear algebra, this is equivalent to saying that $d \in \ker F^T$.

From the above discussion, it follows that (33) is unbounded for some Θ^* if and only if there exists a $d \in \mathbb{Q}^m$ that does not satisfy the conditions above, i.e such that

$$A^T d = 0 \wedge d \ge 0 \wedge (d^T w < 0 \vee F_1^T d \ne 0 \vee F_2^T d \ne 0 \vee ... \vee F_m^T d \ne 0) \tag{35}$$

The existence of such a d can be checked by solving one LPWS and m LPs with negations, which can be done in polynomial time by the result of the previous Section. □