15424-project

Beyond $*$: Visualizing Quantifier Elimination for Real Arithmetic

$$\frac{?}{\exists x\,\forall y\,\varphi}\,\mathbb{R}$$

Abstract: The existence of a quantifier elimination algorithm for real arithmetic is one of the foundational results that enables formal reasoning and verification of CPS. Most of the well-known algorithms for quantifier elimination are extremely complicated, too inefficient to be used on even the simplest of formulae, or both. This makes studying quantifier elimination algorithms difficult. This project aims to rectify this problem by providing a writeup and implementation of the Cohen-Hörmander algorithm along with visualizations to aid understanding.

Introduction

The modeling of cyber-physical systems (CPS), and the subsequent formal verification of the model, are made possible by a multitude of results. Foremost among these is the development of a logic (such as differential dynamic logic) that can express desirable properties of a CPS as logical formulae, along with a set of inference rules that can be used to construct proofs of these formulae. Manually constructing these computationally-verifiable proofs from the axioms of all the formal logics involved would be far too painful even for simple CPS. It thus becomes important to seek methods of automating the proof construction.

While it is impossible for many useful logics to fully automate the proof construction process [3], it is certainly possible to automate portions of it. Rather surprisingly, a 1931 result by Tarski [7], along with related more recent developments [5], show that the entire proof construction process can be automated once the desired goal has been reduced to proving a formula of real arithmetic. This result is absolutely foundational for CPS verification, both from a theoretical and a practical perspective. The existence of an automatic verification procedure for formulae of real arithmetic allows one to abstract away the formal reasoning process behind the real arithmetic proof goals, and simply give inference rules such as the following, which holds whenever $\bigwedge \Gamma \implies \bigvee \Delta$ is a valid formula of real arithmetic [6].

$$\frac{*}{\Gamma \vdash \Delta}\,\mathbb{R}$$

Practically speaking, the real arithmetic proof goals that result from attempts to prove properties of CPS are often prohibitively complex for manual methods.

Given the significance of this result, and the rather mysterious nature of the real arithmetic proof rule, the question “what is really going on here?” likely crosses many students’ minds. A little research would reveal a number of algorithms for automatically deciding the truth of a sentence of real arithmetic (called quantifier elimination (QE) algorithms), but many of the choices have significant disadvantages for someone studing real QE for the first time:

- **Tarski's original algorithm** was a very important theoretical breakthrough, but complicated and so inefficient that it isn't useful for anything besides theoretical purposes [4]. Given the complexity of this algorithm, understanding it would be difficult, and given the inefficiency, interaction with an implementation (which would be very useful for understanding) would not be feasible.
- **Cylindrical-Algebraic Decomposition (CAD)** is the state of the art when it comes to practical QE, so it doesn't suffer from the inefficiency problem of Tarski's algorithm [2]. However, it is incredibly complicated: so much so that it took experts in the field 30 years to produce a working implementation (citation needed). As such, it is likely not suitable for a student in an introductory CPS class.
- **Virtual substitution** is efficient [9], and simple enough to be part of CMU's introductory 15-424 *Logical Foundations of Cyber-Physical Systems* course. The only shortcoming of this algorithm is that it isn't complete, in the sense that there are theoretical limitations (given by the well-known Abel-Ruffini theorem) that prevent it from deciding the truth of arbitrary sentences of real arithmetic. Understanding this algorithm is thus not equivalent to understanding what's going on behind the scenes of the $\mathbb{R}$ proof rule.

However, there is a (not too well-known) alternative that offers a reasonable balance: the **Cohen-Hörmander** algorithm [1]. It is simple enough to be described in this paper, complete in the sense that it can (in principle) decide the truth of any sentence of real arithmetic, and efficient enough to admit implementations that one can actually interact with. This work thus aims to introduce an audience of students taking introductory logic courses to real quantifier-elimination by providing a writeup, a number of visuals, and an implementation of the Cohen-Hörmander algorithm.

Related Work

[10] develops a tool that can be used to visualize the data structures computed by the cylindrical algebraic decomposition algorithm. This present work is analogous to that one in the sense that we provide an implementation that allows users to view and understand the main data structure (the sign matrix) computed by the Cohen-Hörmander algorithm.

[4] presents the Cohen-Hörmander algorithm and provides an implementation in OCaml. Our presentation of the algorithm is based on this one. While the implementation provided by this book is likely more readable than ours, we improve accessibility and usability of the implementation by embedding it in a website and enabling verbose output that allows the reader to trace the operation of the algorithm.

Our presentation of the algorithm also differs from both of the above in that it includes *animated* visualizations intended to complement text-based explanations.

Background

In this section, we very briefly review some of the background material that is necessary to properly define the problem solved by the Cohen-Hörmander algorithm.

Real Arithmetic

The first-order theory of real closed fields is a formal language for stating properties of the real numbers. The language is built up recursively as follows:

Terms: Terms are the construct that the language uses to refer to real numbers, or to combine existing numbers into other ones. They are built up via the following inference rules

$$\frac{c \in \mathbb{Q}}{c \text{ term}} \qquad \frac{x \text{ var}}{x \text{ term}} \qquad \frac{e_1 \text{ term} \quad e_2 \text{ term}}{e_1 + e_2 \text{ term}} \qquad \frac{e_1 \text{ term} \quad e_2 \text{ term}}{e_1 \cdot e_2 \text{ term}}$$

Formulae: Formulae are the construct that the language uses to express assertions about real numbers. The basic, or atomic, formulae are constructed as follows:

$$\frac{e_1 \text{ term} \quad e_2 \text{ term}}{e_1 = e_2 \text{ form}} \qquad \frac{e_1 \text{ term} \quad e_2 \text{ term}}{e_1 < e_2 \text{ form}} \qquad \frac{e_1 \text{ term} \quad e_2 \text{ term}}{e_1 > e_2 \text{ form}} \qquad \frac{e_1 \text{ term} \quad e_2 \text{ term}}{e_1 \leq e_2 \text{ form}} \qquad \frac{e_1 \text{ term} \quad e_2 \text{ term}}{e_1 \geq e_2 \text{ form}}$$

Formulae can be joined together using boolean connectives:

$$\frac{\varphi \text{ form}}{\neg\varphi \text{ form}} \qquad \frac{\varphi \text{ form} \quad \psi \text{ form}}{\varphi \wedge \psi \text{ form}} \qquad \frac{\varphi \text{ form} \quad \psi \text{ form}}{\varphi \vee \psi \text{ form}} \qquad \frac{\varphi \text{ form} \quad \psi \text{ form}}{\varphi \implies \psi \text{ form}} \qquad \frac{\varphi \text{ form} \quad \psi \text{ form}}{\varphi \iff \psi \text{ form}}$$

Finally, variables occuring in terms can be given meaning by way of quantifiers.

$$\frac{\varphi \text{ form} \quad x \text{ var}}{\forall x\, \varphi \text{ form}} \qquad \frac{\varphi \text{ form} \quad x \text{ var}}{\exists x\, \varphi \text{ form}}$$

Real arithmetic is what we get when we give these syntactic constructs their natural meaning over the real numbers. That is, the symbols $+, \cdot, =, <$, etc denote addition, multiplication, equality, and comparison of real numbers respectively. Quantifiers are interpreted to range over the real numbers.

With these constructs, we can formally express many properties of the real numbers. Some examples include:

- Every number is positive, negative, or zero: $\forall x\,(x > 0 \vee x < 0 \vee x = 0)$.
- Every number has an additive inverse: $\forall x\,\exists y\,(x + y = 0)$.
- Every nonzero number has a multiplicative inverse: $\forall x\,(x = 0 \vee \exists y\,(x \cdot y = 1))$.

However, not everything that we intuitively think of as a property of the real numbers can actually be accurately expressed in this language. A typical example is the supremum property: the assertion that every nonempty set of real numbers which is bounded above has a least upper bound has no equivalent in this language [8]. As we shall shortly see, the expressiveness (or lack thereof) of this language is key to the operation of the Cohen-Hörmander algorithm.

Now we can properly define what the Cohen-Hörmander algorithm actually does. It is a quantifier-elimination algorithm: it takes as input a formula φ in this language, and produces a formula ψ which contains no quantifiers and whose free variables are a subset of the free variables of φ. Moreover, ψ and φ have the same truth value regardless of how we choose to substitute for the real variables.

- $\exists y\,(x < y \wedge y \leq 0)$ might be reduced to $x < 0$, because regardless of which value in $\mathbb{R}$ we choose to assign to x, the two formulaes are either both true or both false.
- $\forall x\,(x > 0 \vee x < 0 \vee x = 0)$ might be reduced to $\top$, the formula which is always true. Indeed, since the original formula has no free variables, the quantifier-eliminated formula also cannot have free variables, and thus must be equivalent to either $\top$ (true) or $\bot$ (false).

Real Analysis/Algebra

In this section, we list a few definitions and theorems of basic analysis/algebra that are useful in understanding the Cohen-Hörmander algorithm.

Definition (Sign):The sign of a real number x is

- $+$, or positive, when $x > 0$
- 0 when $x = 0$
- $-$, or negative, when $x < 0$

Theorem (Intermediate value): If f is a continuous function of x (in particular, if f is a polynomial in x), $a < b$, and the signs of $f(a)$ and $f(b)$ do not match, then f has a root in $[a, b]$.

Theorem (Mean value): If f is a differentiable function of x (in particular, if f is a polynomial in x), and $a < b$, then there exists $c \in (a, b)$ such that

$$f'(c) = \frac{f(b) - f(a)}{b - a}$$

Definition (Polynomials with rational coefficients): $\mathbb{Q}[x]$ denotes the set of all polynomials in x with coefficients in $\mathbb{Q}$.

Theorem (Polynomial division): If $a, b \in \mathbb{Q}[x]$ and $b \neq 0$, there exists unique $q, r \in \mathbb{Q}[x]$ satisfying

- $a = bq + r$
- $\deg(r) < \deg(b)$.

The Algorithm

Our approach will be to first treat the simpler univariate case: formulas of the form $\forall x\, \varphi$ or $\exists x\, \varphi$ such that the only variable in φ is x. Unfortunately, due to time constraints and the absence of the required libraries for JavaScript, we do not describe or implement the multivariate generalization here.

Univariate Case

A priori, deciding the truth of a formula of the form $\forall x\, \varphi$ or $\exists x\, \varphi$ requires looping through every single $x \in \mathbb{R}$, substituting that value into φ, checking the result, and then combining the results for all $x \in \mathbb{R}$ in the manner that suits the quantifier. This approach works when the model we're concerned with is finite, but since $\mathbb{R}$ is very much not finite, this cannot possibly yield a useful algorithm.

Understanding the Sign Matrix

The first step in unlocking a decision procedure for real arithmetic is to look closely at what formulae in this language can express. All formulae are ultimately built up from atomic formulae, and atomic formulae are built out of terms, so we'll start there.

Recall the inductive construction of terms: we started with rational constants and variables (in our present case, only x), and we were allowed to combine terms into larger terms by adding and multiplying. Using multiplication only, starting with rational constants and x, we'll get terms of the form qx^n, where $q \in \mathbb{Q}$ and $n \in \mathbb{N}$. These are **monomials** (with rational coefficients) in x. Add addition into the mix, and since multiplication distributes over addition, we arrive at our first important observation: **a term is a polynomial in x with rational coefficients.**

Atomic formula were constructed as e_1 CMP e_2, where e_1, e_2 are terms and CMP was one of $=$, $<$, $>$, $\leq$, or $\geq$. Since terms are polynomials in x, atomic formulae are comparisons between polynomials. But e_1 CMP e_2 is equivalent to $e_1 + (-1) \cdot e_2$ CMP 0 for any choice of CMP, and $e_1 + (-1) \cdot e_2$ is also a term, and therefore a polynomial. Thus,

every atomic formula asserts something about the sign of a polynomial. For example, the atomic formula $3x^2 + 2 \geq 2x + 1$ equivalently asserts that the polynomial $3x^2 - 2x + 1$ is positive or zero.

This realization is key to transforming our infinite loop over $\mathbb{R}$ that we had in our initially proposed algorithm into a finite loop. Polynomials (with the zero polynomial being an easy special case) have only finitely many roots. Between two consecutive roots (also before the first root, and after the last root), a polynomial must maintain the same sign, since if it changed sign, by the intermediate value theorem there would have to be another root in the middle. The upshot is that if $x_1, \ldots, x_n$ are the roots of a polynomial p in increasing order, then by knowing the sign of p at one point in each of the intervals $(-\infty, x_1), (x_1, x_2), \ldots, (x_{n-1}, x_n), (x_n, \infty)$, we know the sign of p for every $x \in \mathbb{R}$. Since the truth of an atomic formula p CMP 0 at a point x is a function of the sign of p at x, **given a finite data structure which specifies the signs of p over the intervals $(-\infty, x_1), (x_1, x_2), \ldots, (x_{n-1}, x_n), (x_n, \infty)$, we can evaluate an atomic formula at any $x \in \mathbb{R}$.** Note that the signs at the points $x_1, \ldots, x_n$ are all 0, since $x_1, \ldots, x_n$ are the roots of p, so in the case where we're dealing with just a single polynomial, we don't need to include the sign information at the roots in the data structure.

A quantifier-free formula φ is just a propositional combination of a bunch of atomic formulae, and as such, knowing the truth value of each of the composite atomic formulae is sufficient to determine the truth value of φ. Above we discussed how to obtain a finite data structure that gives us the truth value of an atomic formulae at any $x \in \mathbb{R}$ - so all we need to do is combine these data structures for all the (finitely many) atomic formulae that are in φ, and we obtain a finite data structure which can be used to evaluate φ at any point x. This is exaftly the **sign matrix** data structure which is the core of the Cohen-Hörmander algorithm.

More formally, let S_φ be the set of all polynomials that occur in atomic formulae within φ. Then the rows of the sign matrix are indexed by the polynomials in S_φ. If $x_1, \ldots, x_n$ are an exhaustive list of all the roots of the polynomials in S_φ with $x_1 < x_2 < \cdots < x_n$, then the columns of the sign matrix are indexed by the list

$$(-\infty, x_1), x_1, (x_1, x_2), x_2, \ldots, (x_{n-1}, x_n), x_n, (x_n, \infty)$$

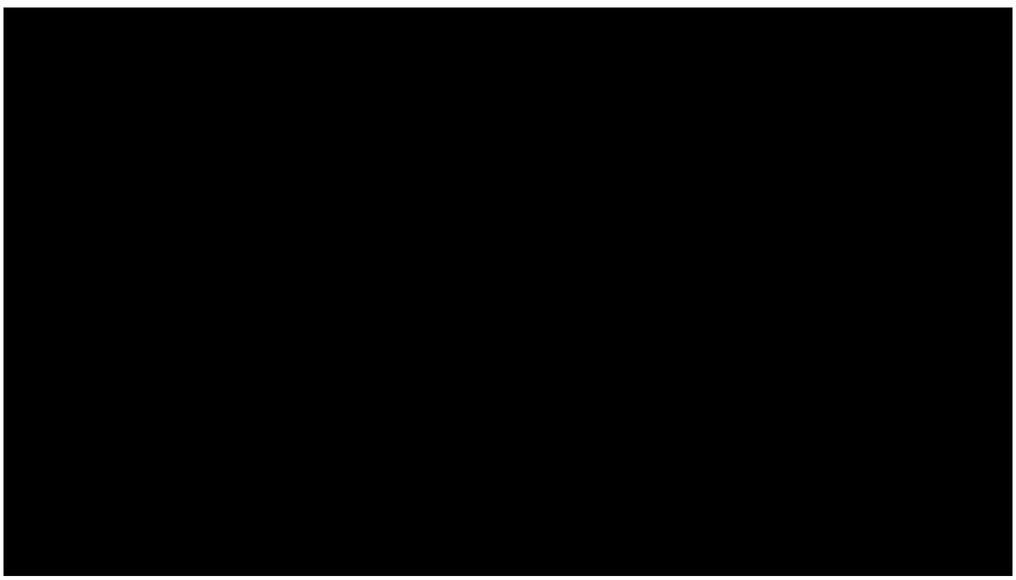
i.e, the singleton sets at the roots and the intervals between them. The entry of the sign matrix at row $p \in S_\varphi$ and column I is just the sign of p on I. Just as before, note that the signs of all the polynomials are invariant on each interval, because if a polynomial (or any continuous function, for that matter) changes sign on an interval, it must have a root in that interval. But since $x_1, \ldots, x_n$ is a list of ALL of the roots of the polynomials in S_φ, and no interval listed above contains any of these points, this is not possible. Note also that in this case, where we potentially have multiple polynomials, we do need to specify the sign of each polynomial at each root: the presence of x_i as a column only means that x_i is a root of one of the polynomials involved - the other polynomials may have nonzero sign at x_i.

Here's an example of a sign matrix for the set of polynomials p_1, p_2, p_3, where $p_1(x) = 4x^2 - 4$, $p_2(x) = (x+1)^3$, and $p_3(x) = -5x + 5$.

	$(-\infty, x_1)$	x_1	(x_1, x_2)	x_2	(x_2, ∞)
p_1	$+$	0	$-$	0	$+$
p_2	$-$	0	$+$	$+$	$+$
p_3	$+$	$+$	$+$	0	$-$

And here's an animation that illustrates the meaning of the sign matrix.

Finally, here's an animation that shows how this sign matrix can be used to compute the truth value of the formula $\forall x\,[(p_1(x) \geq 0 \wedge p_2(x) \geq 0) \vee (p_3(x) > 0)]$. Note how the fact that the signs of the polynomials (and thus the truth values of the atomic formulae) are invariant over each column of the sign matrix allows us to effectively iterate over all of $\mathbb{R}$ by only checking each column of the sign matrix.



One important thing to note is that while the sign matrix relies crucially on the ordering of the roots of the polynomials involved, it **doesn't actually contain any numerical information about the roots** themselves. In our toy example, it's easy to see that $x_1 = -1$ and $x_2 = 1$, but this isn't recorded in the sign matrix, nor is it necessary for the final decision procedure.

Computing the Sign Matrix

Unfortunately, building the sign matrix for an arbitrary set of polynomials isn't as simple as telling the computer to "draw a graph," as we did in the animation above. However, to compute the sign matrix for the set of polynomials $p_1, \ldots, p_n$, we first remove the 0 polynomial if its in the set - it can be added back at the end of the algorithm by simply setting its sign to 0 everywhere. Then we construct a set containing the following polynomials:

- $p_2, \ldots, p_n$
- p_1'

- The remainder r_1 that results upon dividing p_1 by p_1'
- The remainders $r_2, \ldots, r_n$ that result upon dividing p_1 by p_i, for $2 \leq i \leq n$

We recursively compute the sign matrix for this set, and use it to construct the sign matrix for $p_1, \ldots, p_n$. It's not at all clear how the seemingly arbitrarily constructed polynomials above should help us build a sign matrix for $p_1, \ldots, p_n$, so we first discuss that.

Including $p_2, \ldots, p_n$ makes sense: these polynomials are in the "target set" $p_1, \ldots, p_n$ as well. Having information on how their signs change at their roots and the intervals between them certainly helps us build a sign matrix for $p_1, \ldots, p_n$ - it reduces our worries to figuring out the behavior of p_1.

The reason for including p_1' is revealed by the following property: if p_1' has no roots on an interval (a, b), then p_1 can have at most one root in (a, b). The reason for this is that in between any two distinct roots of p_1, there must exist a turning point of p_1, which corresponds to a root of p_1'.

So, if p_1 has two distinct roots x_1, x_2 in (a, b), p_1' must also have a root in (a, b); the contrapositive of this is the desired statement. The same reasoning holds for intervals that are infinite in either or both directions. Given a sign matrix for a set of polynomials including p_1', we know that p_1' will not have any roots in any of the intervals. This is the reason why we include p_1' in the input for the recursive call - it limits the number of "extra roots" that p_1 can have to at most one per interval in the recursively computed sign matrix.

The remainders are included to help us deduce the sign of p_1 at the roots of the recursively computed sign matrix. Since the r_is are defined as remainders when doing polynomial division, there exist polynomials q_i, $1 \leq i \leq n$, such that

$$p_1(x) = q_1(x)p_1'(x) + r_1(x)$$

$$p_i(x) = q_i(x)p_i(x) + r_i(x) \qquad (2 \leq i \leq n)$$

for every $x \in \mathbb{R}$. In particular, we can substitute in any x_k in the recursively computed sign matrix that is the root of p_i, and we get $p_1(x_k) = q_i(x_k)p_i(x_k) + r_i(x_k)$, but since $p_i(x_k) = 0$, this reduces to $p_1(x_k) = r_i(x_k)$. This means that the sign of p_1 at any point x_k in the recursively computed sign matrix that is the root of some p_i, $2 \leq i \leq n$ (resp. p_1') is the same as the sign of r_i (resp. r_1). Since r_i is part of the input to the recursive call, we can read the sign of r_i at x_k from the sign matrix to obtain the sign of p_1.

But why bother with division for this? Indeed, if we added the polynomials $p_1 + p_1', p_1 + p_2, \ldots, p_1 + p_n$ to the input lists instead of these remainders, we could obtain the signs of p_1 at the roots of the polynomials $p_1', p_2, \ldots, p_n$ just as easily (the sign of p_1 at a root x_k of p_i is the sign of $p_1 + p_i$ at x_k). The reason is that we want our recursion to terminate. Given an initial input set of polynomials $p_1, \ldots, p_n$, we construct the set $p_1', p_2, \ldots, p_n, r_1, \ldots, r_n$ as the input to the recursive call. This set could potentially have more than twice the size of the initial set, so a termination argument for the recursion can't possibly be based on decreasing size of the input set. However, the following observations will help us:

- p_1' has smaller degree than p_1
- Because r_1 is the remainder upon dividing p_1 by p_1', r_1 has smaller degree than p_1' (which already has smaller degree than p_1).
- Because r_i is the remainder upon dividing p_1 by p_i, r_i has smaller degree than p_i.

Now, if we choose p_1 such that it has maximal degree in the input set (so that every p_i has degree at most the degree of p_1), the last observation can be replaced by

- Because r_i is the remainder upon dividing p_1 by p_i, r_i has smaller degree than $\mathbf{p_1}$.

Then, when we combine all the observations, we see that the input to the recursive call is constructed by removing a polynomial p_1 of maximal degree, and replacing it with a bunch of polynomials of smaller degree. Thus, every time we recurse

- Either we degree the maximum degree of the input set,
- Or we decrease the number of polynomials in the input having the maximum degree by 1

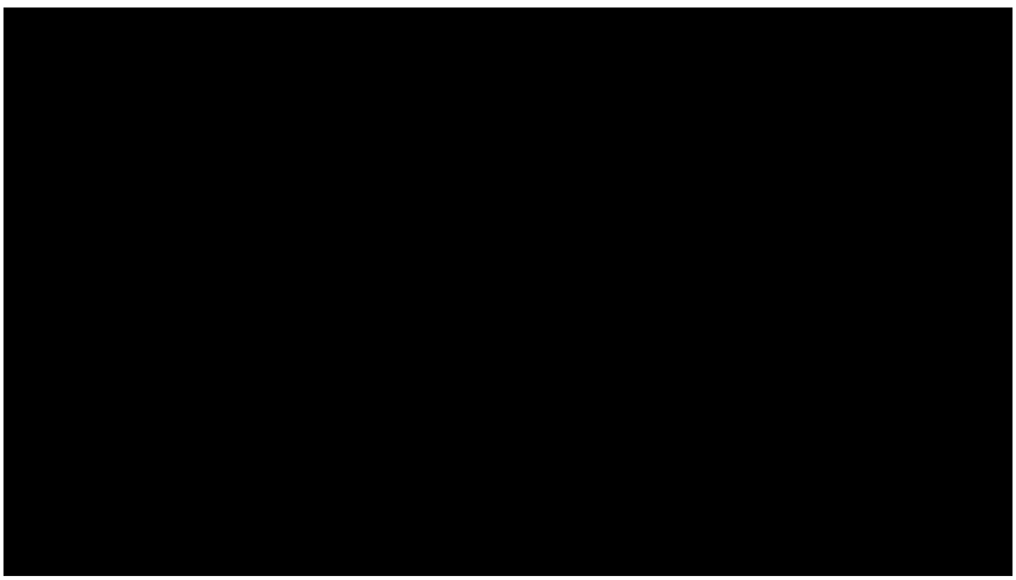
Any recursion having this property will terminate; indeed, the property implies that the set of pairs `(# of polys with max degree, max degree)` form a strictly decreasing sequence in ω^2, which must be finite as ω^2 is well-ordered. Note that the alternative method that we proposed (replacing the remainders with $p_1 + p_1', p_1 + p_2, \ldots, p_1 + p_n$), does not have the above properties, and there is no reason why the alternative method should produce a terminating recursion.

The base case of the recursion can be taken to be any set of constant polynomials. Constant polynomials never change their sign, so the sign matrix in this case has just one column: $(-\infty, \infty)$. The sign of a constant polynomial $p(x) = c$ is simply the sign of c.

Almost all the pieces are in place. We've described how to construct the input to the recursive call, we've provided a base case, and we've argued that the recursion terminates. We now explain how to construct the solution given the output of the recursive call. Most of the work was already done in explaining the intuition behind choosing the input to the recursive call. The signs of $p_2, \ldots, p_n$ at all of their roots and the intervals between them is part of the output of the recursive call already, so we worry only about p_1.

We start by going through the roots x_k in the recursively obtained sign matrix. Any time we see a 0 in the column x_k in any of the rows $p_1', p_2, \ldots, p_n$ (i.e. x_k is a root of at least one of $p_1', p_2, \ldots, p_n$), we assign p_1 the sign of the corresponding remainder (i.e. r_1 for roots of p_1', and r_i for roots of p_i, $2 \leq i \leq n$). For now, we don't do anything for any of the other columns. Here's an example where $p_1(x) = x^2 - 1, p_2(x) = x^2 + 2x$, which gives us $p_1'(x) = 2x, p_2(x) = x^2 + 2x, r_1(x) = -1, r_2(x) = -2x - 1$ as the input for the recursive call, and the following recursively computed sign matrix:

	$(-\infty, x_1)$	x_1	(x_1, x_2)	x_2	(x_2, x_3)	x_3	(x_3, ∞)
p_1'	$-$	$-$	$-$	$-$	$-$	0	$+$
p_2	$+$	0	$-$	$-$	$-$	0	$+$
r_1	$-$	$-$	$-$	$-$	$-$	$-$	$-$
r_2	$+$	$+$	$+$	0	$-$	$-$	$-$



The column x_1 is a root of p_2, so the sign of the corresponding remainder r_2 is lifted to p_1. The column x_2 is not a root of either p_1' or p_2, so we don't assign a sign to p_1 for that column - this is indicated by the $\varnothing$ sign. The column x_3 is a root of both p_1' and p_2: in the animation we choose the lift the sign from r_1 to p_1, but lifting from r_2 would have yielded the same result. This step is called `DETERMINING SIGN of p_1 AT ROOTS of p_1', p_2, ..., p_n` in the implementation.

From this point onwards, we don't need any of the information from the remainder polynomials, so we implement a step which removes them from the sign matrix. In doing so, we may end up with "root" columns which are no longer the root of any polynomial in the matrix (e.g. x_2 in the following example). We merge these columns with their adjacent intervals to maintain the invariant that the columns of the sign matrix alternate between roots and intervals. This step is called `REMOVING REMAINDER INFORMATION AND FIXING MATRIX` in the provided implementation.

Since all roots where we didn't deduce a sign for p_1 were removed in this last step, we now have a sign matrix with signs for p_1 at all roots. We just have to treat the intervals. There are two questions we have to answer for each interval:

- Is there a root of p_1 in the interval? Since p_1' is part of the recursively computed sign matrix, it has no roots in any of the intervals. Our reasoning from before then shows that p_1 can have at most one root in each interval, so we only have to see if the number of roots is 0 or 1.
- What is the sign of p_1 on the interval? Or, if there is a root of p_1 in the interval that splits the interval into pieces, what is the sign of each polynomial on the new pieces?

There are four different types of intervals: the bi-infinite interval $(-\infty, \infty)$ that typically only is produced in the base case, the half-infinite intervals $(-\infty, x_1)$ and (x_n, ∞) that usually occur as the first and last columns of the sign matrix, and the typical bounded interval between two consecutive roots (x_i, x_{i+1}). Fortunately, they can all be treated using the same logic, but there is an important pre-processing step for the unbounded intervals. We give them pseudo-endpoints; these are the signs assumed by p_1 as $x \to \infty$ and $x \to -\infty$ respectively.

These signs can be computed from the sign of the derivative p_1' in the unbounded intervals. Note that the sign matrix is already fully populated in the row p_1', so we can simply read off the sign of p_1' in the necessary intervals.

- If p_1' is positive as $x \to \infty$ (i.e. if in the interval of the sign matrix which touches ∞, the sign of p_1' is positive), then p_1 must eventually increase forever. Since we're dealing with polynomials which cannot have asymptotic growth, increasing forever implies increasing without bound. This means that p_1 must be positive as $x \to \infty$, so we say that the sign of p_1 at the pseudo-endpoint ∞ is positive. Likewise, if p_1' is negative as $x \to \infty$, the sign of p_1 at ∞ is negative.
- If p_1' is positive as $x \to -\infty$ (i.e. if in the interval of the sign matrix which touches $-\infty$, the sign of p_1' is positive), then if we go far enough to the left (past all the roots of p_1), p_1 must eventually be decreasing as we move further left towards $-\infty$ (the sign of the derivative gets flipped here because x is moving to the left, in the opposite of the canonical direction). So we say that the sign of p_1 at the pseudo-endpoint $-\infty$ is negative. Likewise, if p_1' is negative as $x \to \infty$, we say that the sign of p_1 at $-\infty$ is positive.
- The last case, where p_1' is zero on an interval, implies that p_1' is the zero polynomial. This means that p_1, a polynomial of maximal degree, is constant. But then we wouldn't have entered this recursive case anyway - sets of constant polynomials are handled by the base case. So this case is impossible here.

Now that we have assigned pseudo-endpoints to the unbounded intervals, along with the sign of p_1 at these pseudo-endpoints, we proceed with the assumption that every interval has two endpoints, and the sign of p_1 is known at both endpoints. This is true for the pseudo-endpoints as we just stated, and it is true for the "real" endpoints (the roots) because the previous step already determined the signs of p_1 at all roots in the sign matrix.

Given an interval with endpoints a, b (both are either the pseudo-endpoints $-\infty, \infty$ or roots x_i), we split into cases based on the sign of p_1 at a and b. There are 9 possibilities, but fortunately many of them are similar.

- If $p_1(a)$ is positive while $p_1(b)$ is negative, the intermediate value theorem guarantees the existence of a root of p_1 in (a, b). Note that (a, b) is an interval in a sign matrix containing p_1', so p_1' has no roots in (a, b). By our earlier observation, this means that p_1 has at most one root in (a, b). Thus, p_1 has exactly one root in (a, b), which we call c. This root splits the interval (a, b) into three pieces: (a, c), c, and (c, b). The signs of all polynomials on these new pieces are determined as follows:
  - All the "old" polynomials $p_1', p_2, \ldots, p_n$ have invariant sign on the entire interval (a, b), so they certainly have invariant sign on the subsets of this interval (a, c), c, and (c, b). Thus, the sign of any of these polynomials on the pieces (a, c), c, and (c, b) is the same as their sign on (a, b), and we can just copy the sign over.
  - p_1 itself is certainly 0 at c, since c is a root of p_1. Moreover, we know that c is the only root of p_1 in (a, b). This implies that there are no roots of p_1 in (a, c) or in (c, b). Since p_1 is positive at a, we have that p_1 must be positive over all of (a, c); else there would be a root of p_1 in (a, c). Likewise, since p_1 is negative at b, p_1 must be negative over all of (c, b).

- The case where $p_1(a)$ is negative while $p_1(b)$ is positive is exactly dual to this. Again, there must exist a root c of p_1 in (a, b), splitting the interval into three pieces (a, c), c, and (c, b). The signs of $p_1', p_2, \ldots, p_n$ on the new pieces are copied from their signs on (a, b), and the signs of p_1 on (a, c), c, and (c, b) are negative, zero, and positive respectively.
- If $p_1(a)$ is positive while $p_1(b)$ is either positive or zero, we look at the sign of p_1' on the interval. As discussed before, p_1 is not constant, so p_1' is either positive or negative on the interval, implying that p_1 is either strictly increasing or strictly decreasing on the interval. In the increasing case, since $x \in (a, b) \implies x > a$, we have $p_1(x) > p_1(a) > 0$ for all $x \in (a, b)$, and so p_1 has no roots in (a, b) and is positive on (a, b). In the decreasing case, since $x \in (a, b) \implies x < b$, we have $p_1(x) > p_1(b) \geq 0$ for all $x \in (a, b)$, and so again p_1 has no roots in (a, b) and is positive on (a, b).
- The cases where $p_1(a)$ is negative and $p_1(b)$ is either negative or zero is identical to the above.
- The cases where we instead fix $p_1(b)$ to be positive (resp. negative) and allow $p_1(a)$ to be either positive or zero (resp. negative or zero) use the same reasoning as well.
- The last remaining case is of $p_1(a) = p_1(b) = 0$. This is impossible, as the mean value theorem would guarantee the existence of $c \in (a, b)$ with $p_1'(c) = (p_1(b) - p_1(a))/(b - a) = 0$, i.e. a root of p_1' in (a, b), which we know cannot exist.

Combining the above reasoning with our pre-processing, we've described now how to compute the signs of p_1 on each interval, as well as inject roots where necessary. That is, we can deduce signs for p_1 on every interval in the sign matrix, as well as add new columns as needed when there are roots of p_1 that weren't already in the matrix. With this, we've completed adding in all the sign information for p_1, and the sign information for $p_2, \ldots, p_n$ was already in the recursively computed sign matrix. Thus, we run one final filtration/merging pass to remove rows corresponding to polynomials not in the original input list $p_1, p_2, \ldots, p_n$ (this last step is called `FILTERING AND MERGING RESULT`, and is very similar to the previous `REMOVING REMAINDER INFORMATION AND FIXING MATRIX` step), and we output the result.

Sign Matrix Calculation Implementation

We provide an implementation of sign matrix computation via this algorithm here. The implementation is also embedded below. Enter a comma separated list of polynomials with rational coefficients. Enough output will be produced to trace the steps described above. Here are a couple of example inputs:

- $p_1(x) = x^2 - 1, p_2(x) = x^2 + 2x$ can be entered as `x^2 - 1, x^2 + 2x`.

- $p_1(x) = 4x^2 - 4$, $p_2(x) = (x+1)^3$, and $p_3(x) = -5x + 5$ can be entered as `4x^2 - 4, x^3 + 3x^2 + 3x + 1, -5x + 5`. Expansion must be done manually as the parser is not intelligent.
- If you try your own inputs, please keep them relatively small in length and degree. The implementation is not optimized in the least, and will likely not run in time for large inputs.

Some notes on how to read the output:

- In the presentation above, the columns of the sign matrix were the roots and intervals, while the rows were the polynomials. In the implementation it was more convenient to print the roots and intervals as the rows and have an association list mapping each polynomial in each row to its sign. For example, here are the "mathematical" and "program" notations for the sign matrix of p_1, p_2, p_3, where $p_1(x) = 4x^2 - 4$, $p_2(x) = (x+1)^3$, and $p_3(x) = -5x + 5$.

	$(-\infty, x_1)$	x_1	(x_1, x_2)	x_2	(x_2, ∞)
p_1	$+$	0	$-$	0	$+$
p_2	$-$	0	$+$	$+$	$+$
p_3	$+$	$+$	$+$	0	$-$

```
neginf: (4x^2-4, +), (x^3+3x^2+3x+1, -), (-5x+5, +),
root: (4x^2-4, 0), (x^3+3x^2+3x+1, 0), (-5x+5, +),
interval: (4x^2-4, -), (x^3+3x^2+3x+1, +), (-5x+5, +),
root: (4x^2-4, 0), (x^3+3x^2+3x+1, +), (-5x+5, 0),
posinf: (4x^2-4, +), (x^3+3x^2+3x+1, +), (-5x+5, -),
```

- The row type `inf` denotes the bi-infinite interval $(-\infty, \infty)$.
- In the `DETERMINING SIGN ON INTERVALS` step, each time we encounter an interval, we print the "context" - this is the information about the signs on the surrounding roots that is necessary to process the interval.
- For reasons unknown, the creator of the polynomial/rational number library we use decided to output rational numbers in decimal format. For example, the rational number $1/6$ is outputted as `0.1(6)`. Writing rational numbers in fraction form for the input seems to work though.

Comma-separated list of polynomials

Print recursively (may generate a lot of output)

Run

Univariate Decision Procedure Implementation

This script combines the sign matrix computation with the decision procedure we described earlier to evaluate the truth of univariate formulae. The input format is not very polished; here are a couple of examples and notes on the input format

- The formula $\exists x\, (x^2 + 1 \leq 0)$ is inputted as follows:

```
{
quantifier: "exists",
formula: {
  poly: new Polynomial("x^2+1"),
  signs: ["-", "0"]
```

```
}
}
```

- The formula $\forall x\,(\neg(x^2+1 \leq 0))$ is inputted as follows:

```
{
quantifier: "forall",
formula: {
  conn: "not",
  sf: {
    poly: new Polynomial("x^2+1"),
    signs: ["-", "0"]
  }
}
}
```

- The formula $\forall x\,[(p_1(x) \geq 0 \wedge p_2(x) \geq 0) \vee (p_3(x) > 0)]$, where $p_1(x) = 4x^2 - 4$, $p_2(x) = (x+1)^3$, and $p_3(x) = -5x + 5$, is inputted as follows:

```
{
quantifier: "forall",
formula: {
  conn: "or",
  sf: [
    {
      conn: "and",
      sf: [
        {
          poly: new Polynomial("4x^2-4"),
          signs: ["+", "0"]
        },
        {
          poly: new Polynomial("x^3+3x^2+3x+1"),
          signs: ["+", "0"]
        }
      ]
    },
    {
      poly: new Polynomial("-5x+5"),
      signs: ["+"]
    }
  ]
}
}
```

- When inputting polynomials, it is important to use the variable `x` and to leave no white space in the string.
- Again, please stick to small examples (in terms of number of unique polynomials and their degree) in order for the computation to complete quickly.

Run

Deliverables

This webpage embeds all the deliverables, and serves as both the final project and the term paper. We summarize all the deliverables briefly here:

- The written explanation given in the above sections.
- The animations that complement the explanations, embedded in the appropriate locations. The code used to generate the animations and high-quality renders of the animations are also available; see here
- The implementations of the sign matrix calculation and univariate decision procedure are available here.

Acknowledgements

Thanks to Prof. Platzer for making me aware of the Cohen-Hörmander algorithm and providing resources to learn more about it.

Thanks to Grant Sanderson (3blue1brown) and the other manim developers for the framework, without which creating the animations seen above would have been impossible.

Thanks to Robert Eisele (GitHub user infusion) for the Fraction.js rational number library and the Polynomial.js univariate polynomial library.

References

[1] Cohen, Paul J. 1969. “Decision Procedures for Real and p-Adic Fields.” *Communications on Pure and Applied Mathematics* 22 (2): 131–51. https://doi.org/10.1002/cpa.3160220202.
[2] Collins, George E. 1975. “Quantifier Elimination for Real Closed Fields by Cylindrical Algebraic Decompostion.” In *Automata Theory and Formal Languages*, edited by H. Brakhage, 134–83. Berlin, Heidelberg: Springer Berlin Heidelberg.
[3] Gödel, Kurt. 1931. “Über Formal Unentscheidbare sätze Der Principia Mathematica Und Verwandter Systeme i.” *Monatshefte für Mathematik Und Physik* 38 (1): 173–98. https://doi.org/10.1007/BF01700692.
[4] Harrison, John. 2009. *Handbook of Practical Logic and Automated Reasoning*. 1st ed. USA: Cambridge University Press.
[5] McLaughlin, Sean, and John Harrison. 2005. “A Proof-Producing Decision Procedure for Real Arithmetic.” In *Automated Deduction – CADE-20*, edited by Robert Nieuwenhuis, 295–314. Berlin, Heidelberg: Springer Berlin Heidelberg.
[6] Platzer, Andr. 2018. *Logical Foundations of Cyber-Physical Systems*. 1st ed. Springer Publishing Company, Incorporated.
[7] Tarski, Alfred. 1998. “A Decision Method for Elementary Algebra and Geometry.” In *Quantifier Elimination and Cylindrical Algebraic Decomposition*, edited by Bob F. Caviness and Jeremy R. Johnson, 24–84. Vienna: Springer Vienna.
[8] Väänänen, Jouko. 2020. “Second-order and Higher-order Logic.” In *The Stanford Encyclopedia of Philosophy*, edited by Edward N. Zalta, Fall 2020. https://plato.stanford.edu/archives/fall2020/entries/logic-higher-order/; Metaphysics Research Lab, Stanford University.
[9] Weispfenning, V. 1997. “Quantifier Elimination for Real Algebra — the Quadratic Case and Beyond.” *Applicable Algebra in Engineering, Communication and Computing* 8 (2): 85–101. https://doi.org/10.1007/s002000050055.
[10] Wilson, David John. 2014. “Advances in Cylindrical Algebraic Decomposition.” Department of Computer Science, University of Bath.