Adaptive and Precise Root Solving for Polynomials

Jamie Morgenstern
University of Chicago
jamiemmt@cs.uchicago.edu

Umut A. Acar
Toyota Technical Institute, Chicago
umut@tti-c.org

Abstract

Many algorithmic and scientific applications require support for finding the roots of a polynomial. This is a non-trivial task; there is no closed-form solution for computing the roots of an arbitrary polynomial (excluding polynomials of degree four or less), and because floating-point arithmetic has a limited degree and range of precision. Computing the error bounds for such arithmetic is difficult. These factors together make it difficult and expensive to compute the roots of a polynomial exactly. In particular, there is no library or other support in SML for computing the roots of a polynomial.

In this paper, we give a brief overview of our ongoing project to develop an efficient root solver. We describe techniques for 1) finding the roots of a polynomial exactly, 2) finding the roots of a polynomial up to some user-specified precision, and 3) comparing or totally ordering the roots of a polynomial. Our implementation supports these functionalities. We describe some preliminary experimental results, ongoing work, and future directions.

1. Introduction

Standard ML is an excellent language for developing complex algorithmic and scientific software, because it provides clean interfaces to high-level data structures (e.g., lists, vectors) and enables developing algorithms and data structures in a composable, scalable fashion by offering a strong type and module system. Support for various libraries that would be crucial for developing applications (algorithmic and scientific software) is relatively weak, compared to libraries for system development (compilers and interpreters). This is somewhat unfortunate, given the potential impact that the language could achieve in the broader computer science community.

One important part of algorithmic and scientific software systems is a root solver that computes the roots of polynomials. Such systems often require the roots to be either computed exactly or computed within some specified precision. In some cases, it is also important to be able to compare the roots precisely. Since polynomials often have irrational roots, and as computers only support finite precision, floating-point arithmetic operations, it can be expensive to support these operations. In fact, libraries for solving the roots of polynomials have been developed for more widely used languages such as C, but not for ML.

In this paper, we describe an ongoing project for developing an SML library for computing the roots of polynomials. Since solving for the roots of a polynomial exactly can be very expensive (e.g., because a root itself may require a high number of bits to represent). Our primary goal is to offer an operation that enables the programmer to control the cost of root solving because applications often don't need such a degree of precision. To this end, our library supports solving for the root of a polynomial up to a programmer-specified precision, which can be refined adaptively as necessary. To this end, we propose two approaches. The first approach uses Sturm sequences, a classic technique, for computing the roots of polynomials by performing a top-down search on the real line. This approach ensures precise computation of roots but can be expensive. The second approach also uses Sturm sequences, but instead of employing a top-down search, starts with the provided approximate roots to perform a bottom-up search to locate the roots. If the approximations are accurate, then the bottom-up search can perform significantly better than the first. The motivation behind this technique approach is that, using numerical techniques such as the bisection method, or Newton's method, it is often possible to compute efficiently approximate roots with reasonable accuracy.

We present an implementation of the proposed techniques and an experimental evaluation. Our implementation is an SML library (the source code is available via the first authors home page). It is a prototype implementation; it currently has some limitations, which we expect to overcome for the final version of this paper. The implementation, though, enables us to perform a preliminary experimental evaluation. Our experiments show that our root solvers can find the roots precisely using both the top-down search technique, and the bottom-up technique starting with approximations. The results show that the bottom-up technique can perform nearly an order of magnitude faster in locating the precise roots when given reasonably good approximations.

2. Finding Roots with Sturm Sequences

This section describes a classic algorithm using Sturm sequences to compute the roots of a polynomial. This algorithm requires the coefficients to be drawn from a field, i.e., a ring with a division operation defined on the coefficients. We then describe a variant of the algorithm that does not requires division, i.e., the coefficients are drawn from a ring instead of a field.

2.1 Sturm Sequences

By the Fundamental Theorem of Algebra, we know that a polynomial $p(x)$ of degree n has n roots, counting the real, complex, and repeated roots. In many practical applications, we simply need to count and to find the number of distinct, real roots of $p(x)$. The *Sturm sequences* are a vital part of a technique that generalizes Euclid's algorithm to create a finite sequence of polynomials $[p_0, p_1, ..., p_k]$ to find the number of real, distinct roots. For a given polynomial $p(x)$ of degree n, we define its Sturm se-

quence as a sequence of polynomials $[p_0(x), p_1(x), ..., p_k(x)]$, where $p_0(x) = p(x)$, $p_1(x) = p'(x)$ (p_1 is the derivative of $p(x)$) and for all $i > 1$, $p_i(x)$ is computed recursively as the remainder of the previous two polynomials in the sequence $-\texttt{remainder}(p_{i-2}(x), p_{i-1}(x))$. This process is iterated k times, where k is the smallest integer such that $p_k(x) = 0$.

We evaluate a Sturm sequence at t by evaluating each polynomial in the sequence, creating a sequence of real numbers. From this sequence, we define the *sign changes* $S(p_0(x), .., p_k(x))(t)$ of a Sturm sequence $p_0(x), ...p_k(x)$ at t as

$$S(p_0(x), .., p_k(x))(t) = |\{\forall i.0 \leq i < k \mid p_i(t) * p_{i+1}(t) < 0\}|.$$

The sign changes is a count of the number of times the sign changes (omitting 0's) in the sequences $(p_0(x), ..., p_k(x))$. For example, if the sequence evaluates to $[1, -3.2, 0, 6, 0, -7]$, then the number of sign changes is 3.

We say that a polynomial $p(x)$ is *square free* if all irreducible factors of $p(x)$ are distinct. Necessarily, all the roots of a square free polynomial are distinct. For a square-free polynomial, Sturm's Theorem allows us to find the number of real, distinct roots of a polynomial in an interval $(a, b]$. More specifically,

Theorem 1 (Sturm's Theorem)
Given a square-free polynomial $p(x)$ and its Sturm sequence $[p_0, ..., p_k]$, the number of real roots of $p(x)$ in $(a, b]$ is

$$|S(p_0(x), ..., p_k(x))(a) - S(p_0(x), ..., p_k(x))(b)|$$

2.2 Finding Roots

Given a square-free polynomial $p(x) = a_n x^n + \ldots + a_0$, we can use Sturm's Theorem (Theorem 1) to compute the real roots of $p(x)$. The idea is to first generate the Sturm sequence of $p(x)$ and then perform a binary search to locate the roots. We start the binary search with the interval $[-b(p(x)), +b(p(x))]$, where $b(p(x))$ is a bound on the magnitude of the real roots of $p(x)$ established by the Cauchy bound:

$$b(p(x)) = 1 + \frac{\max_{0 \leq i < n} |a_i|}{|a_{n-1}|}$$

We then recursively divide the interval into smaller intervals (by cutting in the middle) until each interval contains only one or fewer roots. We apply Theorem 1 to determine the number of roots contained in an interval. This requires computing the number of sign changes of the Sturm sequences at the endpoints of the interval.

We also define two equivalence relations, $\sim_1$ on polynomials and $\sim_2$ on Sturm sequences. We say $p(x) \sim_1 q(x)$ iff $q(x) = ap(x)$ or $p(x) = aq(x)$ for some constant $a > 0$. Then, given two Sturm sequences $[p_0(x), ..., p_k(x)] \sim_2 [q_0(x), ..., q_k(x)]$ iff $p_i(x) \sim_1 q_i(x)$ for all $i \in [0, \ldots, k]$. Note that $p(x) \sim_1 q(x)$ implies $p(x)$ and $q(x)$ have the same roots and same signs at every value in $\mathbb{R}$. Similarly, $[p_0(x), \ldots p_k(x)] \sim_2 [q_0(x), \ldots q_k(x)]$ implies that the two sequences have the same sign changes at every point.

3. Algorithms

Sturm's Theorem provides a way to find the real roots of a polynomial by applying a binary search on the real line. For this process to be efficient and effective, we need to generate efficiently the Sturm sequence of a polynomial, perform efficient binary search, and find a way to deal with non-square-free polynomials. This section outlines a technique for generating Sturm sequences and computing roots that does not require a division operation.

As we describe in this section, one drawback of the root-finding approach based on Sturm sequences is that it can require time that is cubic in the degree of the polynomial. This can be expensive in practice. We therefore describe an algorithm that can locate the roots of a polynomial starting with some approximations that can be computed by using a faster technique. For example, we can first use floating-point arithmetic to compute the approximate roots, and then use our algorithm to locate the precise roots.

All our algorithms are guided by a precision parameter δ supplied by the user. The precision parameter determines the precision up to which the roots are computed.

3.1 Generation of Sturm Sequences

We show a technique for generating Sturm sequences without using the division operation (over the coefficients). Figure 1 shows our algorithm. The function `mkSeq` takes a polynomial and returns the Sturm sequence generated recursively by `mkSeqAux`, which builds the sequence by extending it with the negative remainder of the two most recent polynomials in the sequence. The `prem` function computes a positively scaled remainder of two polynomials $r(x) \sim_1 r'(x)$, where $r(x)$ is the remainder calculated using division and $r'(x)$ is our algorithm's result. It follows the same procedure as the (polynomial) long division but instead of dividing the polynomials by the leading coefficient of the quotient, it scales the quotient by the absolute value of the leading coefficient of the divisor. As a result, the computed remainder is a positive constant multiple of the actual remainder.

Since $r_i(x) \sim_1 r'_i(x)$ for all i, where $r_i(x)$ is the remainder calculated with division, and $r'_i(x)$ is the remainder calculated without division,

$$[p_0(x), p'(x), r_1(x), \ldots r_{k-2}(x)] \sim_2 [p_0(x), p'(x), r'_1(x), \ldots r'_{k-2}(x)]$$

Lemma 2
The roots of a polynomial with Sturm sequence $[p_0(x), \ldots, p_k(x)]$ may be computed with the sequence

$$[q_0(x), \ldots, q_k(x)] \sim_2 [p_0(x), \ldots, p_k(x)]$$

Proof: The algorithm which uses the Sturm sequence to find roots evaluates the Sturm sequence at a given value t and the calculates the sign changes in the evaluated sequence. The number of sign changes, and not the actual values of the evaluated polynomials, affect the iterations of the algorithm. As $a_i p_i(x)$ has the same roots as $p_i(x)$,and a_i is positive, the sign of $a_i p_i(x)$ at value t will be the same as the sign of $p_i(x)$ at any value t ■

```
remainder(p(x), q(x)) =
let p(x) = a_n x^n + ... + a_o
    q(x) = b_m x^m + ... + b_0
in
  if n − m = 0 then |b_m| * p(x) − a_n * q(x)
  else if n − m < 0 then p(x)
  else sremainder(|b_m| * p(x) − a_n * x^(n−m) * q(x), q(x))
end

mkSeqAux (l) =
  let l = p_i(x) :: p_{i−1}(x) :: _ in
    if p_i(x) ≠ 0 then
      p_{i+1}(x) = −remainder(p_i(x), p_{i−1}(x))
      mkSeqAux(p_{i+1}(x) :: l)
    else l

mkSeq (p(x)) = mkSeqAux ([p'(x), p(x)])
```

Figure 1. Algorithm for generating Sturm sequence of a polynomial.

Our techniques for generating the Sturm sequence and finding the roots apply only to square-free polynomials. We can check that a polynomial $p(x)$ is square free by verifying that the second to last polynomial, $p_{k-1}(x)$ generated by our algorithm (Figure 1) is constant. If not, then it is known that $p(x)$ is not square free, and that $p_{k-1}(x)$ is a square-free polynomial with the same roots as $p(x)$. Thus, we can use $p_{k-1}(x)$ instead of $p(x)$ to generate a Sturm sequence.

We bound the time for generating the Sturm sequence of a polynomial as a function of degree of the polynomial.

Theorem 3
We can generate the modified Sturm sequence for a square-free polynomial of degree n in $O(n^3)$ time without using division.

Proof: Each element of the modified sequence is calculated by finding the remainder of the previous two elements, leading to a strictly decreasing chain of lower-degree polynomials. The length of the sequence is therefore bounded by n.

If $p(x)$ is square-free, the iterations required to calculate the remainder of polynomials p and q is again bounded by the degree of p. Each iteration requires p and q to be scaled by a constant (taking $O(n)$ multiplications), and subtraction of the resulting two polynomials $p' = qc*p$ and $q' = pc*q$ ($O(n)$ subtractions). Thus, the remainder of p and q can be calculated in $O(n^2)$ time, where k is the degree of p.

If $p(x)$ is not square-free, the algorithm above yeilds a square-free polynomial $p_{k-1}(x)$ with the same roots as $p(x)$. Then, this process may be repeated with $p_{k-1}(x)$ and will terminated in at most twice as many steps as the algorithm would take on a square-free polynomial with degree n.

■

Similarly, we can bound the time to evaluate the Sturm sequence at a given value for x.

Theorem 4
Evaluating a Sturm sequence of a n-degree polynomial

$$p_0, ..., p_k$$

requires $O(n^2)$ time.

Proof: Horner's rule shows how to evaluate a polynomial of degree n in $O(n)$ time. Since the Sturm sequence of an n-degree polynomial has at most $n + 2$ polynomials with degree n or less, the bound follows. ■

¡¡¡¡¡¡¡ .mine

3.2 The Top-Down Algorithm

After generating the Sturm sequence of a polynomial, we can find its roots by performing a binary search starting with an interval determined by $b(p(x))$ (Section 2.2). Here, we bound the time for finding the roots using a binary search. For ease of notation, we define ======= ¿¿¿¿¿¿¿ .r8316

3.3 The Top-Down Algorithm

After generating the Sturm sequence of a polynomial, we can find its roots by performing a binary search starting with an interval determined by $b(p(x))$ (Section 2.2). Here, we bound the time for finding the roots using a binary search. For ease of notation, we define

$$SS(p(x), x_1) = S(p_0(x), \ldots, p_k(x))(x_1)$$

ie, $SS(p(x), x_1)$ is the number of sign changes of the Sturm sequence of $p(x)$ evaluated at x_1.

Theorem 5
Let $p(x) = a_n x^n + \ldots + a_0$ be a polynomial with roots $\{x_1, \ldots, x_r\}$ ($r \leq n$), δ be the precision parameter, and d be the minimum distance between any two roots, i.e., $d = \min\{|x_i - x_j|\}$. The time of the Top-Down algorithm is bounded by (`findRoots`) terminates in $O(n^3 \log(\frac{b(p(x))}{n \min(\delta,d)})$.

Proof:
The Top-Down algorithm (`findRoots`) requires evaluating the Sturm sequence at the left and right endpoints of each interval. This process recurs until all roots are isolated in their own intervals and the specified bound on the precision δ is reached (i.e., the interval has size less than δ). Therefore, the search tree has depth $\log(\frac{b(p(x))}{\min(\delta,d)})$. Since the polynomial has n roots, we visit n paths down this tree. The total number of nodes visited is therefore bounded by $O(n+n(D-\log n))$, where D is the depth of the tree, i.e., $O(n + n\log(\frac{b(p(x))}{n \min(\delta,d)})$. Since visiting an interval requires evaluating the Sturm sequence in $O(n^2)$ time (Theorem 4) the bound follows.

■

3.4 An example of the Correction algorithm

Consider the polynomial

$$p = (x - 2.05)(x - 2.33)(x - 2.35)(x - 2.77)$$

and the list of approximate roots

```
fun correctRoots (S, A, δ) =
let
  b ← findCauchyBound (S)
  I ← {(δ · ⌊(r+b)/δ⌋, δ · ⌊(r+b)/δ⌋ + δ] | r ∈ A}
  M ← ref ∅

  fun intervalMatched (i) =
  let r_n ← numRoots (S, i)
      r_m ← |{j ∈ M | j ∩ i ≠ ∅}|
  in (r_n = r_m) end


  fun visit (i as (a, b)) =
    if intervalMatched (i) then
      visit ((2a − b, 2b − a))
    else
      refine (i)

  refine(I = (a, b)) =
  let
    s ← b − a
    if (s ≤ δ) then
      I
    else
      let
        i_1 ← (a,a + s/3)
        i_2 ← (a + s/3,a + 2s/3)
        i_3 ← (a + 2s/3,a + s)
    in
      if intervalMatched (i_1, M) then
        if intervalMatched (i_2, M) then
          refine (i_3)
        else refine (i_2, M)
      else refine (i_1, M)
    end

in
  for each r ∈ I do
    i ← visit (r, M)
    M ← M ∪ {i}
end
```

Figure 2. Pseudo-code for the bottom-up algorithm.

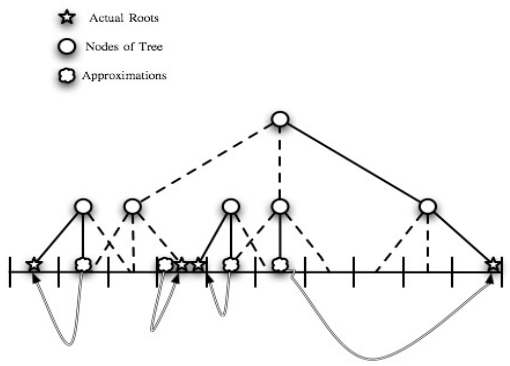

Figure 3. Example Tree

$$[2.15, 2.35, 2.45, 2.55]$$

The Sturm sequence for p, the list of approximate roots, and the precision parameter of $\delta = 0.1$ are supplied to the correction algorithm. Assume, for simplicity, that $b(p(x))$ and δ fix a lattice as in Figure 3.3.

First, the algorithm places each approximate root into the appropriate intervals:

$$[(2.1, 2.2), (2.3, 2.4), (2.4, 2.5), (2.5, 2.6)]$$

Next, the algorithm takes $(2.1, 2.2)$ and checks the Sturm sequence to determine whether or not it contains at least one root. Since it does not, it triples the size of the interval to $(2.0, 2.3)$ and checks again. This time, the Sturm sequence says there is one root in the interval, and since the `foundList` is empty, it refines down to $(2.0, 2.1)$ and adds the interval to the matched set.

The algorithm then checkes the interval $(2.3, 2.4)$, determines there is a root in the interval which has yet to be paired, and adds $(2.3, 2.4)$ to the matched set = $\{(2.3, 2.4), (2.0, 2.1)\}$.

Next, the algorithm takes the interval $(2.4, 2.5)$ and determines there is no root in the interval, and enlarges to $(2.3, 2.6)$. There, it determines there are two roots (one of which has not yet been paired) and thus refines the interval to $(2.3, 2.4)$. At this time, matched set =$\{(2.3, 2.4), (2.3, 2.4), (2.0, 2.1)\}$.

Finally, the algorithm takes $(2.5, 2.6)$ and checks if there is a root in the interval. When it determines there is no root there, it enlarges the interval to $(2.4, 2.7)$ and checks again. Here, the Sturm sequence evaluates to 2, but 2 roots have already been paired with this interval, so the interval is enlarged again to $(2.1, 3.0)$. Here, the Sturm sequence finds another root, and the algorithm refines the interval to $(2.7, 2.8)$. The algorithm returns the set $\{(2.7, 2.8), (2.3, 2.4), (2.3, 2.4), (2.0, 2.1)\}$

4. Implementation

We present an overview of a full interface to an adaptive, precise root solver and a prototype implementation. Figure 4 shows the interface consisting of the signatures for the coefficients (`COEFFICIENT`), intervals (`INTERVALS`), polynomials (`POLYNOMIALS`), Sturm sequences (`STURM_SEQUENCE`), and the root solver (`ROOT_SOLVER`). Our prototype implementation provides structures matching these signatures. The code for the implementation can be found on the first authors web page[1].

[1] The url for first authors web page is `http://people.cs.uchicago.edu/~jamiemmt`

Coefficients. Our polynomials consist of coefficients with the signature `COEFFICIENT`, drawn from a ring (no division operation is needed). In addition to usual ring operations, the signature requires a conversion function between arbitrary precision integers (`IntInf`) and coefficients. Our current implementation provides a structure `Coef` that uses two double-precision floating point numbers (`real` in SML). Ideally, we would provide an implementation that can deal with arbitrary precision, but this has not been a focus of the project thus far.

Intervals. An interval is simply a pair of coefficients.

¡¡¡¡¡¡¡ .mine $mkSeq$ uses the Sturm Sequence Generation algorithm (Figure 1) to create a Sturm sequence for its input polynomial $p(x)$. The function returns a Sturm Sequence; if the argument polynomial is not square-free, the function prints to the screen "Warning, not square free. Providing corrected sequence". In this case, the function creates a sequence for the second-to-last element of the first sequence (it determines if the original polynomial $p(x)$ was square free by checking the degree of the last nonzero element

```
signature COEFFICIENT =
sig
  type t
  val fromIntInf: IntInf.int -> t
  val toIntInf: t -> IntInf.int
  val + : t * t -> t
  val - : t * t -> t
  val * : t * t -> t
  val ~ : t -> t
end
structure Coef : COEFFICIENT = struct ... end

signature INTERVAL =
sig
  type t = Coef.t * Coef.t
  left : t -> Coef.t
  right: t -> Coef.t
end
signature Iv: INTERVAL = struct ... end

signature POLYNOMIAL =
sig
  type t
  val fromList : Coef.t list -> t
  val + : t * t -> t
  val - : t * t -> t
  val * : t * t -> t
  val deg : t -> int
  val shift : t * int -> t
  val scale : t * Coef.t -> t
  val eval : t * Coef.t -> Coef.t
end
structure Poly : POLYNOMIAL = struct ... end

signature STURM_SEQUENCE =
sig
  type t
  val fromPolynomial : Poly.t -> t
  val eval : t * Coef.t -> int
  val cauchy_bound: Poly.t -> Coef.t
  val gcd : Poly.t * Poly.t -> Poly.t
  val remainder: Poly.t * Poly.t -> Poly.t
end
structure Sturm : STURM_SEQUENCE = struct ... end

signature ROOT_SOLVER =
sig
  val findRoots: Sturm.t * Coef.t -> Iv.t list
  val correctRoots: Sturm.t * Coef.t list * Coef.t
                -> Iv.t list
  val refine: Sturm.t * Iv.t * Coef.t -> Coef.t
  val compare: (Iv.t * Sturm.t) * (Iv.t * Sturm.t)
                -> order
end
structure Solver: ROOT_SOLVER = srtuct ... end
```

Figure 4. Coefficient, Polynomial, Sturm Sequence, and Solver signature

($p_{k-1}(x)$)of the sequence; if $deg(p_{k-1}(x) = 0$, $p(x)$ is square free, and if not, returns $mkSeq(p_{k-1}(x))$). This sequence (as described in **??** can be used to isolate roots of $p(x)$. =======

Polynomials. The polynomial interface supports a function for constructing polynomials from a list of coefficients and the usual set of operations on polynomials. Our implementation, the structure `Poly`, simply represents polynomials as a list of coefficients. To evaluate efficiently a polynomial, we use Horner's rule, which requires linear time in the degree of the polynomial. ¿¿¿¿¿¿¿ .r8316

¡¡¡¡¡¡¡ .mine $eval$ evaluates the input sequence at the given input time, counts the sign changes, and returns a tuple of the input time and the sign changes when the Sturm sequence is evaluated at that time. This is implemented using Horner's rule, ensuring $O(n)$-bounded time to evaluate a polynomial of degree n. It is particularly important that this be implemented efficiently. Each evaulation of a Sturm sequence is actually a list of evaluations of polynomials.

$remainder$ calculates (up to a positive constant) the remainder of dividing the first polynomial by the second. The implementation utilizes several key observations fromSection 3.1: namely, the calculated remainder $r'(x)$ is a positive constant multiple of the classical $r(x)$ calculated with division ($r'(x) \sim_1 r(x)$).

The gcd function implements an extended Euclidean algorithm for calculating the greatest common divisor of two polynomials $p(x)$ and $q(x)$ [Knuth(1997)]. In particular, recall an Euclidean domain requires a total order on the elements of the domain (for use with polynomials, degree is this ordering) to calculate the quotient and remainder($r_1(x)$) of $p(x)$ and $q(x)$. This process is carried out iteratively (using $q(x)$ as $p_2(x)$ and $r_1(x)$ as $q_2(x)$)until =======

Sturm sequences. The interface for Sturm sequences provides the key functions for creating and operating on Sturm sequences, particularly for use with the Solver signature. The function `fromPolynomial` uses the Sturm sequence Generation algorithm (Figure 1) to create a Sturm sequence for the input polynomial. The function returns a Sturm Sequence. If the input polynomial, $p(x)$, is not square-free, then it prints a warning and provides the corrected sequence for a square-free polynomial with the same roots as the original. Our implementation of this function follows the description in Section **??**. The function `eval` evaluates the given Sturm sequence at an input value, and returns the number of sign changes in the sequence. As with polynomials, we evaluate the members of the Sturm sequence using Horner's rule, which requires linear time in the degree of the polynomial. The function `remainder` calculates (up to a positive constant) the remainder of dividing the first polynomial by the second. The implementation utilizes several key observations from Section **??**: namely, the calculated remainder $r'(x)$ is a positive constant multiple of the classical $r(x)$ calculated with division ($r'(x) \sim_1 r(x)$). The function `gcd` implements an extended Euclidean algorithm for calculating the greatest common divisor of two polynomials $p(x)$ and $q(x)$ [Knuth(1997)]. In particular, recall that an Euclidean domain requires a total order on the elements of the domain (for use with polynomials, degree is this ordering) to calculate the quotient and remainder ($r_1(x)$) of $p(x)$ and $q(x)$. This process is carried out iteratively (using $q(x)$ as $p_2(x)$ and $r_1(x)$ as $q_2(x)$) until ¿¿¿¿¿¿¿ .r8316 $r_i(x) = 0$. Then, $q_{i+1}(x)$ (the last divisor) is the greatest common divisor of $p(x)$ and $q(x)$, up to a positive constant factor. We use `gcd` for the purpose of the `compare` function (described below).

4.1 Root Solver

The interface for the root solver provides functionality for computing the roots of a polynomial (`findRoots`) directly from its Sturm sequence, and from a set of approximations (`correctRoots`). The interface requires a function `compare` for comparing precisely the roots of two polynomials lying in specified intervals. More precisely, given two intervals, `compare` orders the roots based on the leftmost roots in the specified intervals. If the leftmost roots are equal then `compare` returns `EQUAL`. The function `refine` takes an interval with at least one root and a precision parameter, and bisects the interval repeatedly, returning an interval with at least one root (the leftmost interval of size less than the precision parameter).

¡¡¡¡¡¡¡ .mine

4.2 Solver signature

The SOLVER signature implements both the top down (`findRoots`, Section 3.2)and bottom up (`correctRoots`, **??**) algorithms.

The function $findRoots$ takes Sturm sequence together with a precision parameter, and returns a list of roots (l, r) with the number of roots in each (l, r). $refine$ takes an interval with at least one root and a precision parameter, and bisects the interval repeatedly, returning an interval with at least one root (the leftmost interval of size less than the precision parameter).

The types $Root.t$, and t', respectively, represent intervals in different forms. $Root.left$ and $Root.right$ allow users to access the left and right endpoints of an element of type $Root.t$. The t' type is a simple representation of a (Root.t * number of roots in Root.t).

$correctRoots$ is an implementation of our correction algorithm described in **??**;

it takes a list of approximate roots, the Sturm sequence of a polynomial, and a precision paramete, returning the list of correct intervals and tvals.

$compare$ determines which of two polynomials has the first root in the union of two intervals, given the two intervals, the number of roots in each, and the Sturm sequences of the two polynomials. The Sturm sequences are necessary to check overlapping intervals; Gubas and Karavelas describe an algorithm to order $(a_1, b_1], (a_2, b_2[$. The difficult cases are those when the intervals overlap. If this is the case, refinement of the intervals and further evaluations of the Sturm sequences are used to perform these refinements.

In the process of ordering, we also must compare two roots of two different polynomials and check for equality of those roots. If $p(x)$ and $q(x)$ each have exactly one root in $(a, b]$, then $gcd(p(x), q(x))$ will have a root on that interval if and only if $\exists c \in (a, b]$ such that $0 = p(c) = q(c) = gcd(p(x), q(x))(c)$ (ie, it suffices to check if the gcd of $p(x)$ and $q(x)$ has a root in $(a, b]$. We perform this check when each polynomial has exactly one root in a given interval by calculating the greatest common divisor of the polynomials and checking the gcd's Sturm sequence on the same interval. ======= Our implementation, structure `Solver`, uses the top-down and the bottom algorithms (Sections 3.3 and **??**) to implement `findRoots` and `correctRoots` respectively. The function `refine` is a specialized version of the top-down algorithm that starts with the given interval. It bisects the interval repeatedly, re-

```
signature SOLVER =
  sig
    (* interval with # of roots*)
    type t' = Root.t * int
    val findRoots: SturmSeq.t * C.t ->
      t' list
    val refine: SturmSeq.t * Root.t * C.t ->
      Rootval compare: (t' * SturmSeq.t) *
      (t' * SturmSeq.t) -> order
    val correctRoots: C.t list * SturmSeq.t *
      * C.t ->
      t' list
  end
```

Figure 5. Solver signature

turning an interval with at least one root (the leftmost interval of size less than the precision parameter). The function `compare` determines which of two polynomials has the first root in the union of two intervals, given the two intervals, the number of roots in each, and the Sturm sequences of the two polynomials. The Sturm sequences are necessary to check overlapping intervals The difficult cases are those when the intervals overlap. If this is the case, refinement of the intervals and further evaluations of the Sturm sequences are used to perform these refinements. In the process of ordering, we also must compare two roots of two different polynomials and check for equality of those roots. If $p(x)$ and $q(x)$ each have exactly one root in $(a, b]$, then `gcd(p(x),q(x))` will have a root on that interval if and only if $\exists c \in (a, b]$ such that $0 = p(c) = q(c) = \texttt{gcd}(p(x), q(x))(c)$ (i.e., it suffices to check if the gcd of $p(x)$ and $q(x)$ has a root in $(a, b]$. We perform this check when each polynomial has exactly one root in a given interval by calculating the greatest common divisor of the polynomials and checking the gcd's Sturm sequence on the same interval. ¿¿¿¿¿¿¿ .r8316

5. Experiments

We have performed an experimental evaluation of our implementation, which we report here.

5.1 Measurements

We run our experiments on an i686 with 2GHz AMD Athlon processor running Linux (kernel version 2.6.18) with 1G memory. We use the MLton compiler with run-time flags "-runtime gc-summary", which directs the compiler to report separately the garbage-collection times. Our measurements show that garbage-collection time is negligible for our experiments. We therefore report wall-clock times. We measure the following quantities:

- the time for evaluating a polynomial;
- the time for generating a Sturm sequence of a polynomial;
- the time for evaluating the Sturm sequence of a polynomial;
- the time for isolating the roots of a random polynomial with the top-down algorithm (`findRoots`) for various precision measures (δ),
- the time for finding the roots of a polynomial starting from approximations using the bottom-up algorithm (`correctRoots`).

When measuring the time for evaluating a polynomial $p(x)$ or a Sturm sequence of $p(x)$, we use $x = 1.5$ (any other value for x would have been fine). We have repeated all our experiments with 10,000 polynomials (for each considered degree) and report averages.

When evaluating our root solvers, and Sturm sequences, we have only been able to experiment with polynomials up to degree 7. This is a limitation of our implementation, which uses fixed-precision coefficient for representing Sturm sequences Section 4.

5.2 Data Generation

For our experiments, we generate polynomials randomly. To generate a degree n polynomial, we randomly generate n floating-points numbers, $r_1, \ldots, r_n$ and set the polynomial $p(x)$ to be $p(x) = (x - r_1)(x - r_2)\ldots(x - r_n)$. This approach enables us to determine precisely the roots, i.e., $r_1, \ldots, r_n$, without using a root solver.

5.3 Results

Figure 6 shows the average time for evaluating a randomly generated polynomial as a function of its degree. Evaluation time increases linearly with the degree. This is consistent with known upper bounds. Figure 7 shows the average time for generating the Sturm sequence of a randomly generated polynomial for varying degrees. Our analysis (Theorem 3) shows that generation time is bounded by $O(n^3)$, which is consistent with these results. Figure 8 shows the average time for evaluating the Sturm sequence of a randomly generated polynomial. Our analysis (Theorem 4) shows that evaluating a Sturm sequence takes $O(n^2)$, which is consistent with these results.

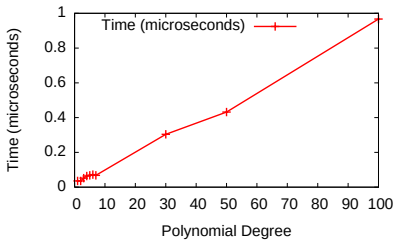

Figure 6. Time for evaluating a polynomial

n	R =1	R=10	R=100
1	0	0	0
2	.285	.286	.285
3	.143	.286	.286
4	.231	.308	.385
5	.130	.261	.348
6	.148	.222	.321

Figure 11. Ratio of the time for root finding with the bottom-up algorithm to that with the top-down algorithm.

Figure 9 shows the average time for finding the roots of a randomly generated polynomial using our top-down algorithm with different precision settings. These measurements show that the top-down algorithm requires super-linear time, which is consistent with Theorem 5. These measurements also show that the precision does not effect performance significantly. This result is also consistent with Theorem 5, which describes the time for calculating roots growing logarithmically with precision.

Figure 10 shows the average time for computing the roots of a randomly generated polynomial. For these experiments, we generate different approximate roots parameterized by the parameter R. More specifically, we approximate each root r_i of the polynomial with $a_i = r_i - R * \delta$. For example, if $R = 100$, then each root is 100 δ-intervals away from the actual root. The measurements show that the time for computing the roots grows super-linearly with the degree and that it is not highly sensitive to the accurateness of the approximation. These measurements are consistent with Theorem **??**.

All our results thus far give evidence about the effectiveness of our approach in the asymptotic. To evaluate the effectiveness more accurately, we performed several other comparisons.

Table 1 shows the ratio of the time for finding the roots of a polynomial using our top-down algorithm to polynomial evaluation. We consider different precision parameters. The results show that top-down root finding can be orders of magnitude more expensive that evaluating the polynomial. This is expected, because root finding is asymptotically $\Omega(n^2)$ slower than evaluating a polynomial. Indeed, the gap increases as the degree of the polynomial increases. We do not observe a significant increase in runtime with an increase in precision. This is consistent with our expectation, as the top-down algorithm depends on the precision logarithmically.

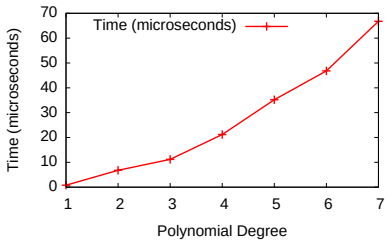

Figure 7. Time for generating a Sturm sequence

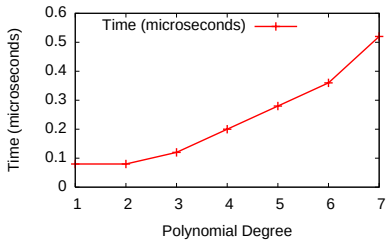

Figure 8. Time for evaluating a Sturm sequence

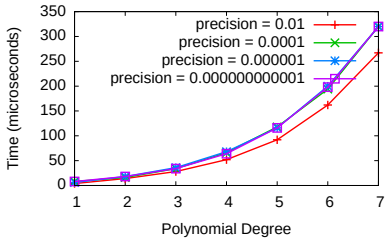

Figure 9. Time for top-down root finding.

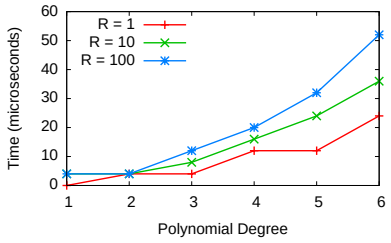

Figure 10. Bottom-up (correction) root finding.

δ	$n=1$	$n=2$	$n=3$	$n=4$	$n=5$	$n=6$	$n=7$
10^{-2}	111.1	388.9	538.5	812.5	1352.5	2250	3926.5
10^{-4}	222.2	444.4	653.8	1031.2	2647.0	2694.4	5705.9
10^{-6}	166.7	500.0	692.3	1062.5	1705.9	2777.8	4705.9
10^{-16}	222.2	500.0	653.8	1000.0	1705.9	2722.2	4705.9

Table 1. Ratio of the time for top-down root finding to that of polynomial evaluation.

Table 5.3 compares the performance of our top-down algorithm, which finds the roots by performing a top-down search guided by Sturm sequences, and our bottom-up algorithm, which finds the roots starting with given approximations. For the experiments, we generate roots parameterized by an approximation constant R. More specifically, we approximate each root r_i of the polynomial with $a_i = r_i - R * \delta$. For example, if $R = 100$, then each root is 100 δ-intervals away from the actual root. The results show that the bottom-up algorithm, i.e., starting with approximations, can be significantly faster (up to more than 8 times) than the top-down algorithm, especially for higher degree polynomials.

6. Related Work

Much of the theoretical framework and practical justification of this paper comes from [Guibas and Karavelas(1999)], a paper describing the usefulness of root-finding using sturm sequences and interval arithmetic for kinetic motion simulation.

The idea of using approximation algorithms stems from the large amount of work done in this area. Elsewhere [Fortune(2002)], an algorithm is described which uses the companion matrix of a polynomial to approximate roots within the relative error of floating-point arithmetic. This algorithm is more efficient if the roots of a polynomial are not very close together. MPSolve [Bini and Fiorentino(2000)] has been one of the fastest root approximators, though it lacks any runtime analysis. Other work [Tsigaridas and Emiris(2006)] utilizes continued fractions to exploit certain traits of polynomials with integer coefficients. Further studies [Rouillier and Zimmermann(2004)] describe a speed-up of Uspinsky's method, using bisection together with Descartes' rule of signs.

7. Discussion

Since our approach requires a ring of coefficients, instead of a field, we expect that it will be applicable in a broader range of applications than an approach that requires the division operation. Our implementation, however, needs an exponential (in the degree of the polynomial) number of bits to represent the coefficients of the Sturm sequence. More specifically, when generating the Sturm sequence for polynomials $p(x)$ and $q(x)$, our approach can take up as much space as $t(q_k)^n$, where q_k is the leading coefficient of q, n is the degree of p, and t is the space the classic approach using the division operation. This is why our current implementation can support relatively small degree polynomials (more precisely up to degree 7).

We expect to be able to fix this problem by taking advantage of the structure of the multiplication operations. More specifically, when generating Sturm sequences, instead of performing the multiplication operation, we can keep the coefficient as a tuple consisting of the actual coefficient and a multiplier, which is represented succinctly.

In this paper, we have not compare directly the effectiveness of our algorithm to numerical approaches, e.g., Bisection method, Newton's method. These algorithms, however, are relatively easy to implement, thus we expect to be able to complete this comparison in the near future.

8. Conclusion

We present techniques for solving the roots of polynomial by using fixed precision arithmetic. We use Sturm sequences to represent polynomials and use them to locate precisely roots up to a user-specified precision constant. To this end, we provide two algorithms. Our top-down algorithm uses Sturm sequences directly but requires cubic time in the degree of the polynomial. The other, bottom-up algorithm, starts with approximations computed numerically and refines them to find the exact roots. Depending on how good the approximations are, this approach can be significantly faster than the direct approach. We present a prototype, an implementation, and an experimental evaluation of the approach. Our experiments show that computing roots generally is expensive but our bottom-up algorithm can be significantly faster than the top-down algorithm. We hope that our paper will be a starting point for future work providing better support for root solving algorithms, and more generally, scientific application domains in SML.

References

[Bini and Fiorentino(2000)] D. Bini and G. Fiorentino. Numerical computation of polynomial roots using mpsolve. 2000. doi: http://www.dm.unipi.it/cluster-pages/mpsolve/mpsolve.pdf.

[Fortune(2002)] Steven Fortune. An iterated eigenvalue algorithm for approximating roots of univariate polynomials. *J. Symb. Comput.*, 33 (5):627–646, 2002. ISSN 0747-7171. doi: http://dx.doi.org/10.1006/jsco.2002.0526.

[Guibas and Karavelas(1999)] Leonidas J. Guibas and Menelaos I. Karavelas. Interval methods for kinetic simulations. In *SCG '99: Proceedings of the fifteenth annual symposium on Computational geometry*, pages 255–264, New York, NY, USA, 1999. ACM. ISBN 1-58113-068-6. doi: http://doi.acm.org/10.1145/304893.304978.

[Knuth(1997)] Donald E. Knuth. The art of computer programming, volume 1 (3rd ed.): fundamental algorithms. 1997.

[Rouillier and Zimmermann(2004)] Fabrice Rouillier and Paul Zimmermann. Efficient isolation of polynomial's real roots. *Journal of Computational and Applied Mathematics*, 162(1):33 – 50, 2004. ISSN 0377-0427. doi: DOI:10.1016/j.cam.2003.08.015. URL `http://www.sciencedirect.com/science/article/B6TYH-4B1Y720-2/2/bfd139e64f2858473bcb6b1787cc2384`. Proceedings of the International Conference on Linear Algebra and Arithmetic 2001.

[Tsigaridas and Emiris(2006)] Elias P. Tsigaridas and Ioannis Z. Emiris. Univariate polynomial real root isolation: continued fractions revisited. In *ESA'06: Proceedings of the 14th conference on Annual European Symposium*, pages 817–828, London, UK, 2006. Springer-Verlag. ISBN 3-540-38875-3. doi: http://dx.doi.org/10.1007/11841036_72.