May 2025

Project Report – CCA

SUBDIVISION ALGORITHMS AND REAL ROOT ISOLATION

Project carried out by

Simon SEPIOL-DUCHEMIN

Joshua SETIA

Project supervised by

Mohab SAFEY EL DIN

Contents

1 Introduction

Finding the roots of a polynomial is a fundamental problem that has many modern applications in numerous fields such as computer vision, robotics, and computational geometry (Petitjean, 1999). Industrial applications often involve input polynomials of a large degree (sometimes larger than 10,000) and coefficients of large bit size (sometimes several millions).

However, the Abel-Ruffini theorem (Stewart, 2004) states that there is no general algebraic solution in radicals for polynomials of degree 5 or higher. This implies that computing exactly the roots by means of radicals is impossible for such polynomials in general.

To overcome these limitations, real root isolation algorithms are used. These algorithms provide a hybrid symbolic numerical approach to locate and approximate the real roots of univariate polynomials without requiring a general algebraic solution. They take as input a univariate polynomial with integer coefficients. They return a finite list of intervals (with rational boundaries) containing all the real roots such that each interval contains one and only one real root of the input polynomial.

1.1 Bisection Algorithms

Bisection algorithms are a kind of divide and conquer algorithm for finding the roots of a continuous function f based on the intermediate value theorem (Rudin, 1976). It takes as input an interval $[a, b]$ such that $f(a)f(b) < 0$, and computes the value of $f(c)$ such that c is the middle point of the interval $[a, b]$. If $f(c) = 0$ or if the difference between a and b is less than a chosen threshold, it returns c. If not it starts again on either $[a, c]$ if $f(a)f(c) < 0$ or $[c, b]$ if $f(c)f(b) < 0$.

1.2 State of the art

There exist multiple methods to isolate the real roots of a univariate polynomial. Classical symbolic methods, such as Descartes-based methods (Vincent–Collins–Akritas) and Sturm sequences, recursively subdivide intervals using sign variation tests. They achieve a worst-case bit-complexity of $\widetilde{O}(n^4\tau^2)$ for degree n and coefficient size τ. Continued-fraction methods, which use linear fractional transformations to converge on each root, have been improved to match the previous complexity. More recent hybrid symbolic–numeric methods aim to combine the speed of numerical computation with the guarantees of symbolic techniques. Such guarantees include completeness (no missed roots), precision control and termination. For instance, Rouillier–Zimmermann's hybrid isolator (Rouillier and Zimmermann, 2004) begins with fast multiprecision filters and switches to exact computation only when needed. The ANewDsc algorithm (Sagraloff and Mehlhorn, 2016) goes further by incorporating Newton iterations and local Taylor approximations, accelerating convergence near clustered roots while maintaining rigorous guarantees. These methods preserve the $\widetilde{O}(n^4\tau^2)$ complexity but are often significantly faster in practice (Rouillier and Zimmermann, 2004; Sagraloff and Mehlhorn, 2016). Maple's `RealRootIsolate` command is based on hybrid methods influenced by the Approximate Bitstream Newton–Descartes algorithm developed by A. Kobel and M. Sagraloff. The article (Kobel et al., 2016) describes an improvement over the ANewDsc algorithm (Sagraloff and Mehlhorn, 2015).

1.3 Our Work

We investigated Descartes' rule of signs and its proof, allowing us to bound the number of real positive roots of a polynomial in a given interval. Then we studied the general idea of the bisection algorithm. We implemented the functions required by the algorithm one by one testing their efficiency compared to the one proposed by the Flint library. Having all those functions, we implemented the bisection algorithm and tested it against the Maple isolation function.

1.4 Outline of the Report

This report first presents the theoretical background required, including Descartes' Rule of Signs and the bounds on the roots of a univariate polynomial. Then, we explain how to use these results to isolate the roots, including implementation details and optimizations. In the end, a section is dedicated to the practical results obtained.

2 Theoretical Foundations

2.1 Bounding Real Roots

2.1.1 Cauchy's and Lagrange's Bounds

Cauchy's and Lagrange's bounds provide an upper bound on the moduli of all complex roots (Vigklas, 2010).

Lagrange's bound states that every root z satisfies:

$$|z| \leq \max\left\{1, \sum_{i=0}^{n-1}\left|\frac{a_i}{a_n}\right|\right\}.$$

On the other hand, Cauchy's bound states:

$$|z| \leq 1 + \max\left\{\left|\frac{a_{n-1}}{a_n}\right|, \left|\frac{a_{n-2}}{a_n}\right|, \ldots, \left|\frac{a_0}{a_n}\right|\right\}.$$

2.1.2 Other Bounds (e.g., Local-Max)

The "local-max" linear complexity bound (ub_{LM}) (Vigklas, 2010) pairs each negative coefficient $a_k < 0$ with the largest preceding positive coefficient a_m ($m > k$) encountered so far. The value of a_m is effectively scaled by $\frac{1}{2^t}$, where t is its usage count. The bound is the maximum of the computed roots :

$$ub_{LM} = \max_{\{k|a_k<0\}}\left(\frac{-a_k \cdot 2^{t_m}}{a_m}\right)^{1/(m-k)}$$

This approach runs in linear time.

For example, on $f(x) = 4x^2 - 2x - 1$, we have the pairs $(4, -2)$ and $(4, -1)$, so we compute the roots $\frac{2*2^1}{4}$ and $(\frac{1*2^2}{4})^{(1/2)}$ and return $\frac{2*2^1}{4} = 1$.

2.2 Descartes' Rule of Signs

Definition 2.1. *We write the polynomial $f(x)$ as*

$$f(x) = \sum_{i=0}^{n} f_i x^{b_i},$$

where we have integer powers $0 \leq b_0 < b_1 < \cdots < b_n$, and nonzero integer coefficients $f_i \neq 0$. If $b_0 > 0$, then we can divide the polynomial by x^{b_0}, which would not change its number of positive roots. Thus, without loss of generality, we assume that $b_0 = 0$.

Let $V(f)$ be the number of sign changes of the coefficients of f, meaning the number of indices k such that $f_k f_{k+1} < 0$.

Let $Z(f)$ be the number of positive roots (counting multiplicity).

Let f' be the derivative of f.

Theorem 2.2. *The number of positive roots (counting multiplicity) of f is equal to the number of sign changes in the coefficients of f, minus a nonnegative even number (Basu et al., 2006, pg.54).*

Lemma 2.3. *If $f_n f_0 > 0$, then $Z(f)$ is even.*
If $f_n f_0 < 0$, then $Z(f)$ is odd.

Proof. If $f_n f_0 > 0$, then both are negative or both are positive, so for example if both are positive f start at $f(0) = f_0$ and tends to $+\infty$ as $x \to +\infty$, so as f is continuous, by the intermediate value theorem it must cross the x-axis an even number of time. So $Z(f)$ is even. The case of $f_n f_0 < 0$ is similar. □

Proof of Theorem 2.2. Firstly, if $f_n f_0 > 0$, then there is an even number of sign change, and if $f_n f_0 < 0$, there is an odd number of sign change. So, the parity of $Z(f)$ is the same as $V(f)$.

We need to show that $Z(f) < V(f)$, we proceed by induction on n, the number of coefficients of f.

For $n = 1$, we have $f(x) = f_1 x^{b_1} + f_0$, $x = \sqrt[b_1]{\frac{-f_0}{f_1}}$, if there is no change of sign $f_0 f_1 > 0$ and $\frac{-f_0}{f_1} < 0$ so there is no real positive root. If there is one sign change, $\frac{-f_0}{f_1} > 0$ so there is exactly one real positive root.

Now assume $n \geq 2$.

By using the induction hypothesis, $Z(f') = V(f') - 2s$ for some integer $s \geq 2$.

By Rolle's theorem, there exists at least one positive root of f' between any two different positive roots of f.

Any k-multiple positive root of f is a $k-1$ multiple root of f'. Thus $Z(f') \geq Z(f) - 1$.

If $f_0 f_1 > 0$, then $V(f') = V(f)$, else $V(f') = V(f) - 1$. In both case, $V(f') \leq V(f)$.

Together, we have

$$Z(f) \leq Z(f') + 1 = V(f') - 2s + 1 \leq V(f) - 2s + 1 \leq V(f) + 1$$

Further, since $Z(f)$ and $V(f)$ have the same parity, we have $Z(f) \leq V(f)$. □

3 The Descartes Bisection Algorithm

Here f is a univariate polynomial with integer coefficients, and B is a bound on the real positive roots of f.

3.1 Overall Strategy

It can be observed that if the number of sign changes is 0 or 1, then the number of positive real roots is respectively 0 or 1. To isolate the real positive roots of a univariate polynomial f on a given interval, it is first required to use a change of variable on f that maps this interval to $[0, +\infty]$, in order to use Descartes' rule of signs. We can then check the number of sign changes. If it is zero there is no root in this interval. If it is one, there is exactly one root in this interval. If it is 2 or more, we can recall the algorithm on the first half of the interval with $f(\frac{x}{2})$, and the second half of the interval with $f(\frac{x+1}{2})$. We can always transform the problem back to the $[0, 1]$ interval by applying the substitution $x \mapsto x \times \mathrm{B}$ in f.

3.2 Example

Let f be the univariate polynomial $f(x) = x^2 - 4x + 3$. Its real roots are 1 and 3 so we can use 4 as a bound on the roots. The search interval becomes $[0, 1]$, and we examine over $Q(x) = f(4x) = 16x^2 - 16x + 3$.

1. Map the interval to $[0, \infty]$, We compute $Q_\infty(x) = Q(\frac{1}{x+1}) = 3x^2 - 10x + 3$. There are 2 sign changes so we call the function on $Q_{left}(x) = Q(x/2) = 4x^2 - 8x + 3$ and $Q_{right} = Q(\frac{x+1}{2}) = 4x^2 - 1$
2. Compute $Q_{left}(\frac{1}{x+1}) = 3x^2 - 2x - 1$, there is one sign change so the interval $[0, 1/2[$ is isolating one root.
3. Compute $Q_{right}(\frac{1}{x+1}) = -x^2 - 2x + 3$, there is one sign change so the interval $]1/2, 1]$ is isolating one root.
4. Finally, we can map the intervals we found back to the initial coordinate system by multiplying each interval by B. We then find the intervals $[0, 2[$ and $]2, 4]$, each isolating one root.

3.3 Interval Transformations

3.3.1 Mapping $[0, +\infty]$ to $[0, 1]$

We first obtain a bound on the greatest root of f as in Section 2.1. Then we consider the rescaling $x \mapsto x \times \mathrm{B}$, so that all positive roots lie in $[0, 1]$.

3.3.2 Mapping $[0, 1]$ to $[0, +\infty)$

We apply the change of variable

$$t = \frac{1}{x+1}.$$

As x varies over $[0, +\infty)$, t varies over $(0, 1]$:

$$x = 0 \quad \Rightarrow \quad t = 1, \qquad x \to +\infty \quad \Rightarrow \quad t \to 0.$$

3.3.3 Mapping $[0, 1/2]$ and $[1/2, 1]$ to $[0, 1]$

Let f be a function on $[0, 1]$. Consider the transformations:

$$t = \frac{x}{2}, \qquad u = \frac{x+1}{2}.$$

Then:

$$x = 0 \quad \Rightarrow \quad (t, u) = (0, 1/2), \qquad x = 1 \quad \Rightarrow \quad (t, u) = (1/2, 1).$$

Thus, as x runs over $[0, 1]$, t covers $[0, 1/2]$ and u covers $[1/2, 1]$.

3.4 Handling Special Cases

Because Descartes' rule of signs gives a bound on the number of positive roots, we need to check if there is a root at 0. Checking a root on 0 is just checking if the constant coefficient of the polynomial is 0.

3.5 The Recursive Algorithm Pseudocode

Algorithm 1: Recursive isolation

Input: Polynomial $f \in \mathbb{Z}[X]$ with all positive roots in $[0, 1]$
Input: c an integer
Input: k an integer, an array of solutions sol
Input: nb_sol an integer encoding the number of roots found
Output: Array of solutions

```
1  if f(0) = 0 then
2      Add 0 to the solution array;
3      Update f(x) ← f(x)/x;
       // Divide out the root at 0
4      Recursively call the algorithm on f(x);
5  Substitute x ← 1/(x+1) in f(x) to obtain Q(x);
6  Compute the number of sign changes in Q(x);
7  if number of sign changes = 0 then
8      return // No root in (0, 1)
9  else if number of sign changes = 1 then
10     Add the current interval to the solution array;
11 else
12     Recursively call the algorithm on f(x/2);
13     Recursively call the algorithm on f((x+1)/2);
```

4 Key Algorithmic Component: Taylor Shift

The Descartes bisection algorithm requires the polynomial transformations $f(x/2)$, $f(\frac{1}{x+1})$ and $f((x+1)/2)$. $f(x/2)$ involves simple coefficient scaling, however computing $f((x+1)/2)$ and $f(\frac{1}{x+1})$ efficiently is more complex. The core operation is the **Taylor shift**, computing $g(x) = f(x+a)$ for a polynomial $f(x) = \sum_{i=0}^{n} f_i x^i$. In our case, we only need $a = 1$. There are numerous algorithms that compute Taylor shifts (Von Zur Gathen and Gerhard, 1997), and each have different complexities. We will focus on two specific ones : Horner and Divide and Conquer algorithms.

4.1 Algorithms for Taylor Shift

The algorithms presented in this section come from Von Zur Gathen and Gerhard (1997). We use the same notation as in 2.1.

4.1.1 Horner's method

$f(x+1) = f_0 + (x+1)(f_1 + ... + (x+1)f_n...)$, it takes n steps.

Lemma 4.1 (Horner complexity)**.** *Horner's method for Taylor shifts has an arithmetic operation complexity of* $\mathcal{O}(n^2)$.

Proof. For each step i, we have at most $i-1$ additions and i multiplications in $\mathbf{Z}$ (multiplying by x is just shifting the coefficients by 1). This results in a total of $\mathcal{O}(n^2)$ arithmetic operations in $\mathbf{Z}$. □

4.1.2 Divide and conquer strategy

Let's say we want to compute the Taylor shift $f(x+1)$ where $f \in \mathbb{Z}[x]$ is of degree n. In a precomputational stage, we compute $(x+1)^i$ for $0 \leq i < \lfloor \frac{n+1}{2} \rfloor$. In the main stage we write $f = f^{(0)} + x^{\lfloor \frac{n+1}{2} \rfloor} f^{(1)}$, with $f^{(0)}, f^{(1)} \in \mathbb{Z}[x]$ of degree less than $\lfloor \frac{n+1}{2} \rfloor$. Then we have :

$$g(x) = f^{(0)}(x+1) + (x+1)^{\lfloor \frac{n+1}{2} \rfloor} f^{(1)}(x+1),$$

where we compute $f^{(0)}(x+1)$ and $f^{(1)}(x+1)$ recursively.

Lemma 4.2 (Divide and Conquer complexity)**.** *Let* $n \to \mathcal{M}(n)$ *be such that there exists an algorithm for multiplying two polynomials of degree* n *in operations. The divide and conquer approach for Taylor shifts has an arithmetic operation complexity of* $\mathcal{O}(\log_2(n)\mathcal{M}(n))$.

Proof. In this method, we split the polynomial in $\log_2(n)$ blocks. For each block, we do some polynomial multiplications of size up to n, with each multiplication costing $\mathcal{M}(n)$ operations. This results in a total of $\mathcal{O}(\log_2(n)\mathcal{M}(n))$ arithmetic operations in $\mathbf{Z}$.

□

5 Iterative Taylor shift

We implemented an iterative Taylor shift, slightly outperforming the one provided by flint in the context of the isolation function. The key observation is that during the isolation function, we only compute Taylor shifts on polynomials of the same degree except when a root is found on 0 which is rare. Because every polynomial of the same degree is decomposed in the same way by the iterative Taylor shift, we can compute the decomposition and the powers of $(x+1)$ at the beginning of the isolation function and use those data for every Taylor shift.

5.1 Benchmark results

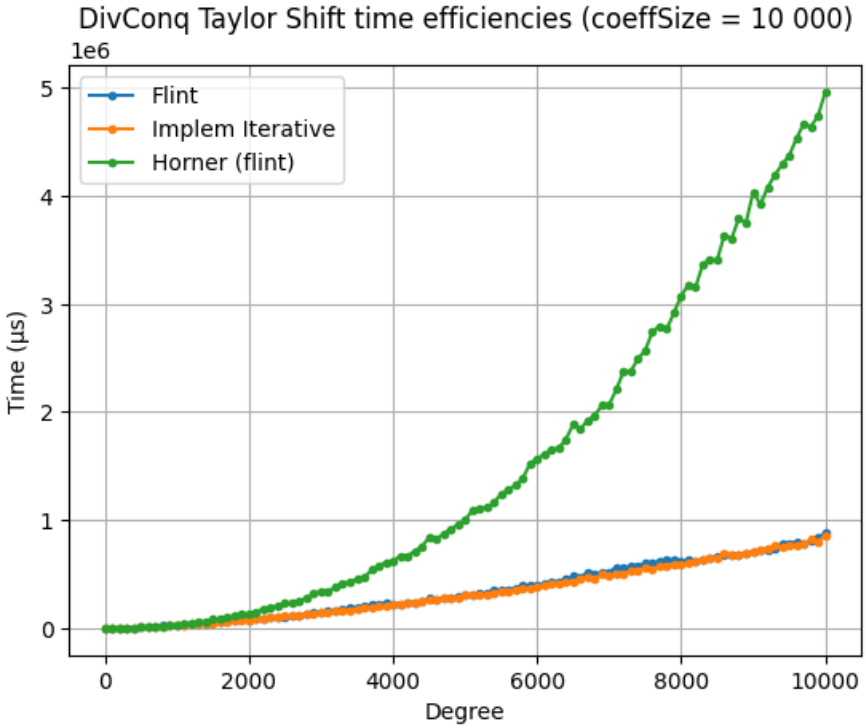

Figure 1: Taylor Shift Benchmark — Growing degree

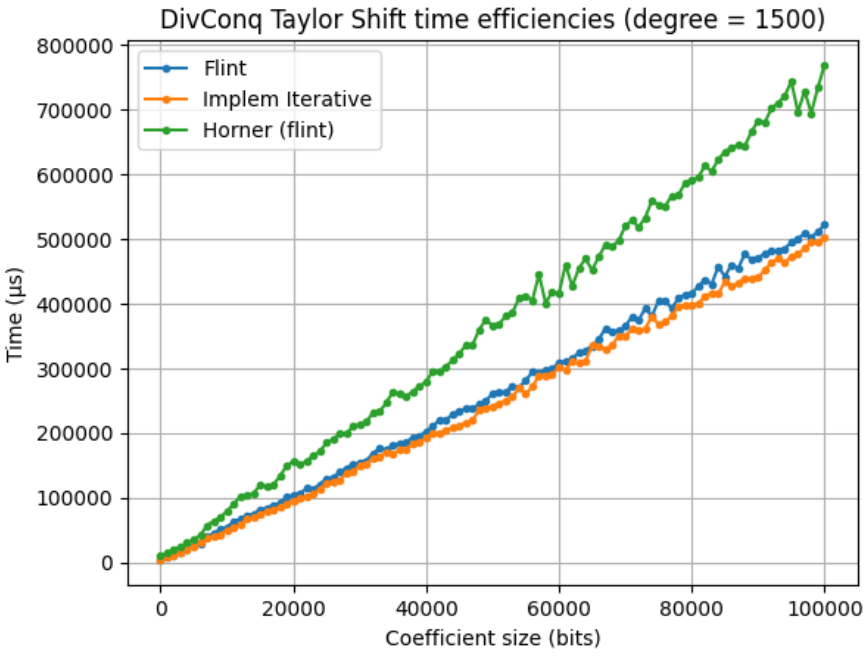

Figure 2: Taylor Shift Benchmark — Growing coefficient size

We benchmarked our iterative implementation of the Divide and Conquer approach for the Taylor Shift, along with Flint's native Horner Taylor Shift and Flint's native Taylor Shift function which chooses the best algorithm depending on the polynomial's size. Our benchmarks were done on polynomials growing in degree from 0 to 10 000 (growing by 100 each step) with a fixed coefficient bit size of 10 000, and growing in coefficient bit size from 1 to 100 001 (growing by 1000 each step) with a fixed degree of 1500.

When the degree of the polynomial is growing, our results match the expected theoretical expectations :

- For Horner's method, the execution time grows quadratically, matching with the algorithm's complexity of $\mathcal{O}(n^2)$.
- Flint's adaptive Taylor Shift function and our implementation behave similarly, as both use the Divide and Conquer approach when the degree surpasses the threshold. We can see that the execution time grows approximately by $\mathcal{O}(M(n)\log n)$. This result also matches the complexity of the algorithm, considering that polynomial multiplication is done at best in $\mathcal{O}(nlog(n)log(log(n))$ using the Cantor and Kaltofen algorithm (1991), based on the Fast Fourier Transform (see Neiger 2025).

Although our implementation behaves similarly to Flint's adaptive function as the degree grows, we can see that it is asymptotically faster when the coefficient size is growing. This matches the results we were aiming for, by saving running time for the pre-computation step of the algorithm.

We also compared our iterative implementation with Flint's version of Taylor shifts by measuring directly on our isolation function. When we benchmarked implementations directly on the isolation function, we used polynomials of growing degree from 1 to 1001 (growing by 10 each step) with a fixed coefficient bit size of 10 000, and polynomials of growing coefficient bit size from 1 to 100 001 (by 1000 each step) with a fixed degree of 100. As you can see on the results in the following page, our implementation slightly outperforms Flint's for most input sizes.

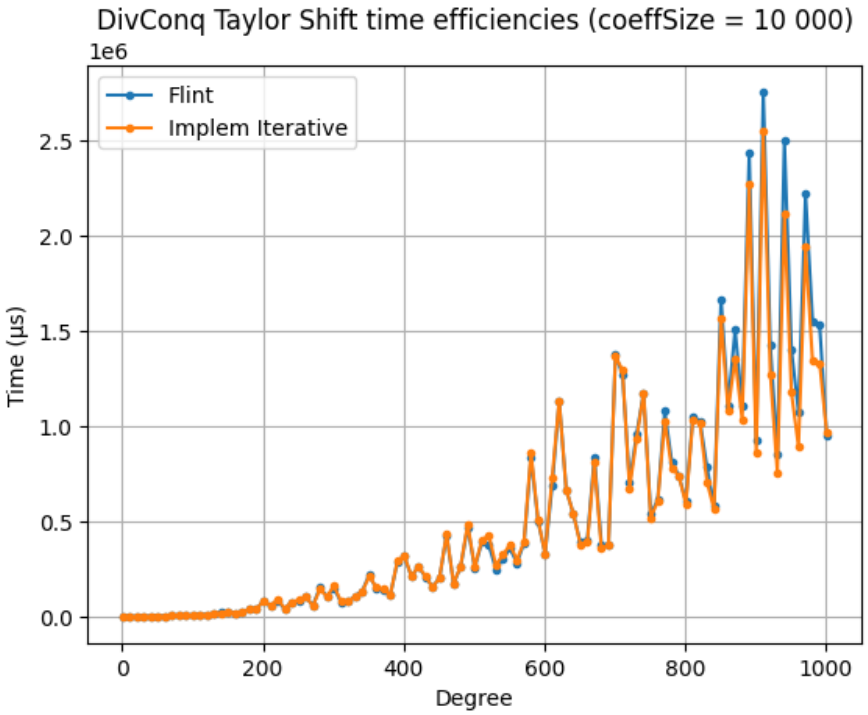

Figure 3: Taylor Shift in isolation Benchmark — Growing degree

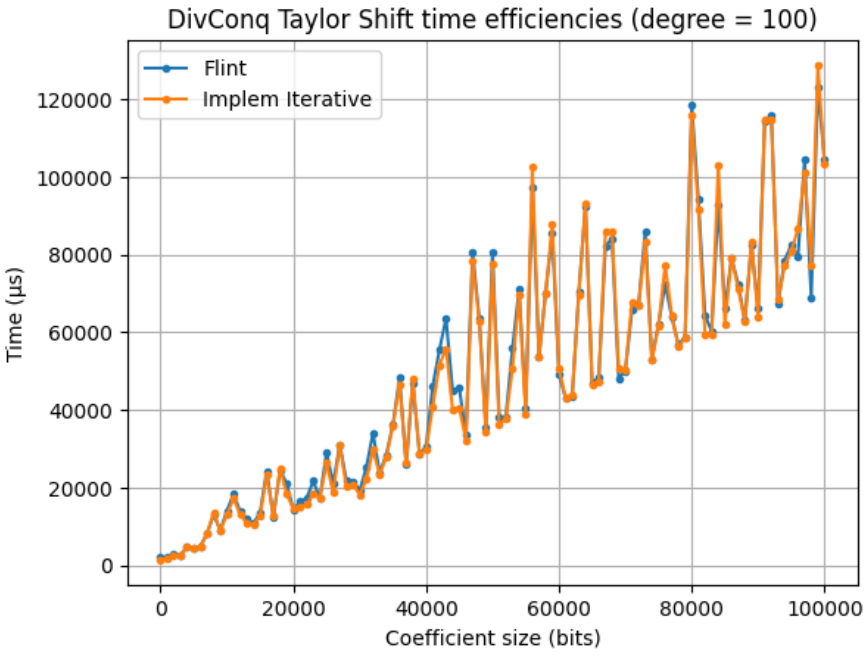

Figure 4: Taylor Shift in isolation Benchmark — Growing coefficient size

5.2 Evolution of Practical Performance

Historically, when von zur Gathen and Gerhard published their results in 1997, Horner's method was actually faster than the Divide and Conquer approach in running time, although the latter has better asymptotic complexity. As explained earlier, the Divide and Conquer complexity $\mathcal{O}(\log_2(n)\mathcal{M}(n))$ depends on polynomial multiplication efficiency. At the time this research paper was published, polynomial multiplication was done in $\mathcal{O}(n^2)$ using the naive algorithm, or $\mathcal{O}(n^{1.59})$ using Karatsuba's algorithm (published in 1962). The resulted complexity would only surpass Horner's when using faster polynomial multiplication algorithms, based on Fast Fourier Transform. However, such algorithms were not implemented for practical use until the early 2000's. Such improvements were also made possible thanks to hardware improvement and compiler optimization. As a result, the Divide and Conquer algorithm is now asymptotically faster than Horner's method in running time as well, as confirmed by our own benchmarks.

5.3 FLINT Library

FLINT: Fast Library for Number Theory (Hart et al., 2024) is a C library for fast number theory computations. It provides data structures and functions for working with integers, polynomials, and other algebraic objects. This project specifically uses the `fmpz` and `fmpz_poly` types provided by FLINT.

5.4 Coefficient Truncation

5.4.1 Bound on the bit growth of the Taylor shift

When performing the Taylor Shift for the polynomial transformation $f(\frac{1}{x+1})$, our goal is to then apply Descartes' rule of signs on the transformed polynomial. Hence, we are only really interested in the transformed polynomial's coefficients' signs, rather then their actual value. Taking this into consideration, we can save computation time by truncating each coefficients of the polynomial before performing the Taylor Shift. We then inspect the coefficients after the Taylor Shift in a way that Descartes' rule of signs gives the same results as if we had not truncated the coefficients.
By truncating we mean getting rid of the coefficients' least significant bits. The problem is finding if a coefficient sign is reliable after the truncation and the Taylor shift. To find this out, we must first understand how the bit size of the coefficients evolves after performing a Taylor shift $f(x+1)$.

Lemma 5.1 (Bit size growth under Taylor shift)**.** *Let $f(x) = \sum_{i=0}^{d} f_i x^i$ and $T(f)$ be the largest bit size of a coefficient of the initial polynomial f. We have :*

$$T(f(x+1)) \leq T(f) + d$$

Proof. By performing the Taylor Shift, we have :

$$f(x+1) = \sum_{i=0}^{d} f_i (x+1)^i$$

With the binomial theorem, we have :

$$\sum_{i=0}^{d} f_i (x+1)^i = \sum_{i=0}^{d} f_i \left(\sum_{k=0}^{i} \binom{i}{k} x^k \right)$$

Here, the notation of $f(x+1)$ is grouped by initial coefficients f_i instead of powers of x. For a single power p of x, we have :

$$f(x+1)_p = \left(\sum_{i=p}^{d} f_i \binom{i}{p} \right) x^p$$

Let f_p be the p^{th} coefficient of the polynomial $f(x+1)$. We have :

$$\begin{aligned} f(x+1)_p &\leq T(f) \sum_{i=p}^{d} \binom{i}{p} \\ &\leq f_p \sum_{i=0}^{d-p} \binom{p+i}{i} x^p \\ &\leq f_p \sum_{i=0}^{d-p} \binom{d}{i} x^p \end{aligned}$$

Because $\sum_{i=0}^{d-p} \binom{d}{i} \leq \sum_{i=0}^{d} \binom{d}{i} = 2^d$, we then have :

$$T(f(x+1)) \leq T(f) + d.$$

□

5.4.2 Coefficient truncation and sign changes count

Lemma 5.2 (Reliable sign detection after coefficient truncation)**.** *Let $f(x) = \sum_{i=0}^{d} f_i x^i$, and suppose each coefficient is truncated by removing l least significant bits, such that*

$$t_i = \frac{f_i}{2^l} \iff f_i = t_i 2^l + e_i$$

with $0 \leq e_i < 2^l$ for each i. We define the following polynomials :

$$f_t(x) = \sum_{i=0}^{d} t_i x^i, \quad E(x) = \sum_{i=0}^{d} e_i x^i.$$

Then, for each coefficient of index i, if

$$|f_t(x+1)_i| > 2^d,$$

the sign of the i^{th} coefficient of $f(x+1)$ is guaranteed to be the same as that of $f_t(x+1)_i$.

Proof. We have

$$\begin{aligned} f(x+1) &= \sum_{i=0}^{d} f_i (x+1)^i \\ &= \sum_{i=0}^{d} (t_i * 2^l + e_i)(x+1)^i \\ &= \sum_{i=0}^{d} t_i (x+1)^i 2^l + e_i (x+1)^i \\ &= f_t(x+1) 2^l + E(x+1) \end{aligned}$$

For $f_t(x+1)_i$ to have to correct sign, we want its sign to remain unchanged when we multiply it by 2^l and add the error term.

$$|f_t(x+1)_i * 2^l| > |E(x+1)_i| \iff |f_t(x+1)_i| > \frac{|E(x+1)_i|}{2^l}$$

We know that $T(E(x+1)) \leq T(E(x)) + d = l + d$, so

$$\frac{|E(x+1)_i|}{2^l} < 2^d.$$

Meaning that if $|f_t(x+1)_i| > 2^d$ the i^{th} coefficient has a reliable sign because the error is too small to change it. In the function counting the number of sign changes, we can check if a coefficient has a reliable sign by comparing it to 2^d. We can ignore all unreliable coefficient and stop the count if we find two sign changes as we will need to bisect. If we find less than two sign changes, we compute the Taylor shift on the original polynomial and count the sign with full precision. □

5.4.3 Benchmark results

We benchmarked the performance of our isolation function with and without coefficients truncation. As you can see in the results on the following page, this optimization allowed us to save significant computation time.

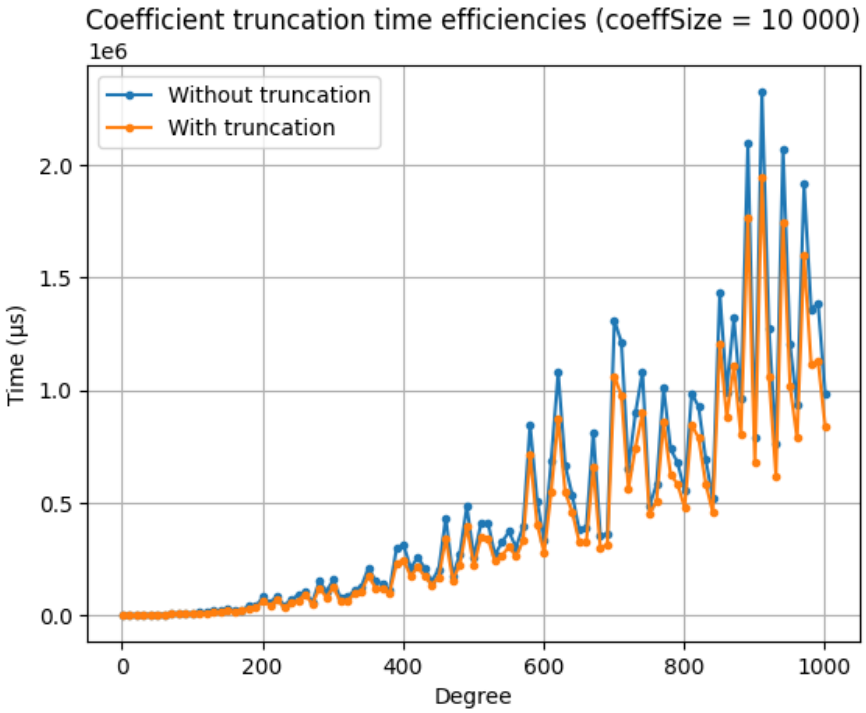

Figure 5: Coefficients truncation Benchmark — Growing degree

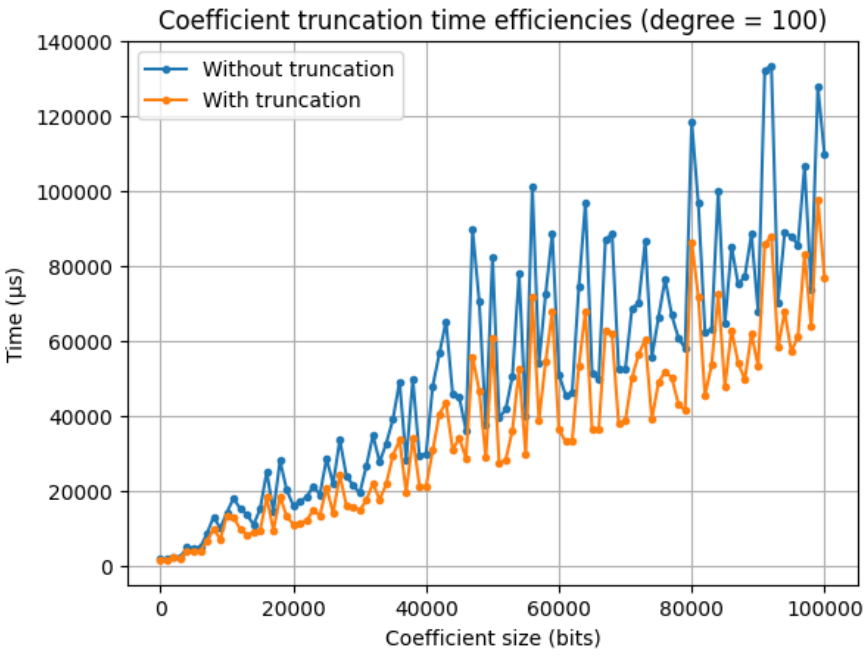

Figure 6: Coefficients truncation Benchmark — Growing coefficient size

6 Experimental Results

Table 1: Computation Time for Hermite and Chebyshev Polynomials

Degree	**Chebyshev Time (s)**	**Hermite Time (s)**
50	0.005608	0.003175
100	0.027256	0.029004
150	0.108052	0.141251
200	0.352504	0.414945
250	0.857163	0.997298
300	1.748651	2.190265
350	2.901181	3.161642
400	3.438392	4.842559
450	5.619110	8.092590
500	9.360566	12.220592
550	15.201526	21.877603
600	20.843555	28.144817
650	28.628735	37.495660
700	36.417224	48.467094
750	47.927813	62.917642

7 Conclusion

Our goal throughout this project was to implement quasi optimal algorithms for arithmetic operations on univariate polynomials. These algorithms could then be used in the real root isolation implementation. We successfully managed to build an asymptotically faster Taylor shift function than Flint's for this particular context. We were also able to implement fast and quasi optimal algorithms for affine transformations when performing a change of variable on a polynomial. We showed that those implementations match their expected theoretical complexities. We also optimized the use of Descartes' rule of signs by truncating the coefficients of the polynomial before performing the change of variable.

Overall, we were able to implement a bisection algorithm that uses fast arithmetic operations on univariate polynomials, by using quasi optimal algorithms. To further explore the optimization of our project, the next step would be to use high-performance implementations through parallelization.

References

Basu, S., Pollack, R., and Roy, M.-F. (2006). *Algorithms in Real Algebraic Geometry (Algorithms and Computation in Mathematics)*. Springer-Verlag, Berlin, Heidelberg.

Hart, W., Johansson, F., et al. (2024). *FLINT: Fast Library for Number Theory*. FLINT Project. Version 3.0.1, Documentation.

Kobel, A., Rouillier, F., and Sagraloff, M. (2016). Computing real roots of real polynomials ... and now for real! In *Proceedings of the ACM on International Symposium on Symbolic and Algebraic Computation*, ISSAC '16, page 303–310. ACM.

Neiger, V. (2025). Introduction to algebraic algorithms (flag, mu4in902, epu-n8-ial). `https://vincent.neiger.science/` (consulté en 2025). Polycopié de cours, Sorbonne Université.

Petitjean, S. (1999). Algebraic geometry computer vision: Polynomial systems, real complex roots. *Journal of Mathematical Imaging and Vision*, 10:191–220.

Rouillier, G. and Zimmermann, P. (2004). Efficient isolation of polynomial real roots. In *Proceedings of the International Conference on Computer Algebra and its Applications (CAA)*. Available at `https://sites.math.rutgers.edu/~zeilberg/akherim/rouillier2004.pdf`.

Rudin, W. (1976). *Principles of Mathematical Analysis*. McGraw-Hill, 3rd edition. See Theorem 4.23, Chapter 4, pages 92–93.

Sagraloff, M. and Mehlhorn, K. (2015). Computing real roots of real polynomials.

Sagraloff, M. and Mehlhorn, K. (2016). Computing real roots of real polynomials... and now for real! *arXiv preprint*, arXiv:1605.00410.

Stewart, I. (2004). *Galois Theory*. Chapman and Hall/CRC, 3rd edition. See Chapter 9, Section "Solvability by Radicals", pages 198–205.

Vigklas, P. S. (2010). *Upper Bounds on the Values of the Positive Roots of Polynomials*. PhD thesis, University of Thessaly, Volos, Greece. Supervisor: Dr. Alkiviadis Akritas.

Von Zur Gathen, J. and Gerhard, J. (1997). Fast algorithms for taylor shifts and certain difference equations. In *Proceedings of the 1997 international symposium on Symbolic and algebraic computation*, pages 40–47.