THE NEW IS THE WELL-FORGOTTEN OLD — F4 ALGORITHM OPTIMIZATION

Anonymous authors
Paper under double-blind review

ABSTRACT

The Gröbner basis is a fundamental concept in computational algebra. F4 is one of the fastest algorithms for computing Gröbner basis. In this paper, we will discuss the process of writing effective F4. Despite the fact that this work focuses on algorithms from computational algebra, some of the results and ideas presented here may have broader applications beyond this specific subject area. In general, the theory described below can be regarded as an abstraction, as it progresses through the text. This is because the text is not actually about the F4 algorithm itself, but rather about the power of profiling, unconventional techniques, and selecting the appropriate memory model. We will provide examples of inefficient usage of the standard library, recall the fundamental principles of optimization in order to apply them as efficiently as possible to obtain the fastest F4 algorithm, using non-traditional approaches.

1 A BRIEF OVERVIEW OF THE THEORY

First, let us define the Gröbner basis.

Definition 1 *Let $f_1, \dots, f_k$ be a set of polynomials. They are called a **Gröbner basis** of an ideal $I = (f_1, \dots, f_k)$ if, for every polynomial f in I, there exists a polynomial f_i in $f_1, \dots, f_k$ such that the leading term of f is divisible by the leading term of f_i.*

Then, we define the S-pair and the key theorem for computing Groebner bases.

Definition 2 *Let a_i and a_j be leading terms of f_i and f_j. Let $L = LCM(a_i, a_j)$. The **S-polynomial** or **S-pair** is $S_{ij} = \frac{L}{a_i}f_i - \frac{L}{a_j}f_j$.*

Theorem 1 *(Buchberger criterion) Let $F = \{f_1, \dots, f_k\} \subset I$ for some ideal I. If S_{ij} is reduced to zero by F for all pairs (i, j), then F is a Gröbner basis of I.*

All algorithms for computing the Groebner basis are based on this theorem and work as follows: they check whether the S-pair has been reduced to zero, and if not, they add the reduced result to the basis. If the first algorithm for calculating Gröbner bases, the Buchberger algorithm (Buchberger, 1965), is used as directly as possible, the F4 algorithm (Faugere, 2002) may already be difficult to comprehend. The main innovation of the F4 algorithm is the use of a matrix approach rather than reducing the S-pairs individually. This allows us to bring the matrix into a triangular form, which simplifies the process.

2 A BRIEF OVERVIEW OF THE IMPLEMENTATION

In order to implement the most efficient algorithm possible, we have written the code in C++. It has been necessary to create standard classes for computational algebra, including terms, monomials, polynomials, and matrices.

For example, term is responsible for storing the degrees of variables x_i. We must be able to multiply and divide terms.

Already at this stage, some libraries have made strange decisions choosing the primary data storage structure to be the std::set or the std::list. If the motivation for using std::set, which stores sorted pairs $(x_i, degree)$, is related to a reluctance to process the order on your own, then the use of std::list is not justified in any way. Recall that std::list should only be used when there is no alternative. In practice, it is difficult to imagine a scenario where std::list would be more efficient than other container types. Computational algebra is certainly not one of those scenarios.

Understandably, the most effective way is to store the terms in memory sequentially. The simplest approach is to use std::vector, where x[i] represents the degree of x_i. It is clear that, in order to be memory-efficient, the polynomials representing a set of monomials are also implemented using the std::vector.

This is better not only because we are more efficient at hitting the cache, but also asymptotically. For example, by using the two-pointer technique, we can add two polynomials in $O(n)$ time, instead of $O(nlogn)$ using std::set.

3 F4 MATRICES

During the computation of a Gröbner basis for a given system of polynomials, the F4 algorithm produces large matrices. In this section, we will discuss these matrices and the effective methods of working with them in details.

The rows of this matrix represent polynomials, the columns represent terms in given order, and the entries in the matrix represent the coefficients of the terms in the corresponding polynomials. For example, if we have 3 polynomials $f_1 = ad + bd + cd + d^2, f_2 = ab + bc + ad + cd, f_3 = ab + b^2 + bc + bd$ the matrix would look like:

$$\begin{array}{c|ccccccc|} & ab & b^2 & bc & ad & bd & cd & d^2 \\ f_1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ f_2 & 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ f_3 & 1 & 1 & 1 & 0 & 1 & 0 & 0 \end{array}$$

The polynomials for the matrix are generated using the **SymbolicPreprocessing** method. We will not go into the details of this process. After initialization of the matrix, a Gaussian elimination takes place.

$$\begin{array}{c|ccccccc|} & ab & b^2 & bc & ad & bd & cd & d^2 \\ f_4 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ f_5 & 1 & 0 & 1 & 0 & -1 & 0 & -1 \\ f_6 & 0 & 1 & 0 & 0 & 2 & 0 & 1 \end{array}$$

After performing Gaussian elimination, we obtained the polynomials $f_4 = f_1 = ad + bd + cd + d^2, f_5 = ab+bc-bd-d^2, f_6 = b^2+2bd+d^2$. In order to reconstruct the original basis, the resulting polynomials must contain a leading term that was not present in the input polynomials used in the matrix. Let us denote the leading term of the polynomial x by $lt(x)$. In this particular example only the polynomial f_6 can be added to our basis, since $lt(f_5) = ab = lt(f_2)$ and $lt(f_4) = lt(f_1)$, since $f_4 = f_1$.

Let us denote the number of unique terms(columns) as n and the number of input polynomials(rows) as m. Then, it is well established that Gaussian elimination works for O(n^2m), since $n \geq m$ in our generated matrix.

3.1 GBLA

To better understand the situation, let us delve a little deeper into the specifics of the F4 matrix. As we previously did not provide details on the SymbolicPreprocessing method, it is now appropriate to describe the structure of the matrix.

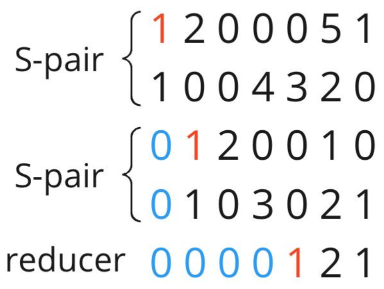

Figure 1: F4 matrix structure

SymbolicPreprocessing generates a matrix by first adding S-pairs to two rows. Let $L = LCM(a_i, a_j)$. Then $\frac{L}{a_i} f_i$ is placed in the upper row and $\frac{L}{a_j} f_j$ is placed in the lower row. SymbolicPreprocessing then finds the reducers of the added polynomials in the current basis, and also appends these reducers to the matrix. See Figure 1.

Faugère and Lachartre have proposed (Boyer et al., 2016) an approach that accelerates the process of matrix reduction multiple times. To begin, we must identify all the pivots in the matrix and then reorder the columns and rows such that all the pivots are located in the upper left corner of the resulting matrix. Once this has been accomplished, we can denote the final matrix as $\begin{pmatrix} A & B \\ C & D \end{pmatrix}$. See Figure 2.

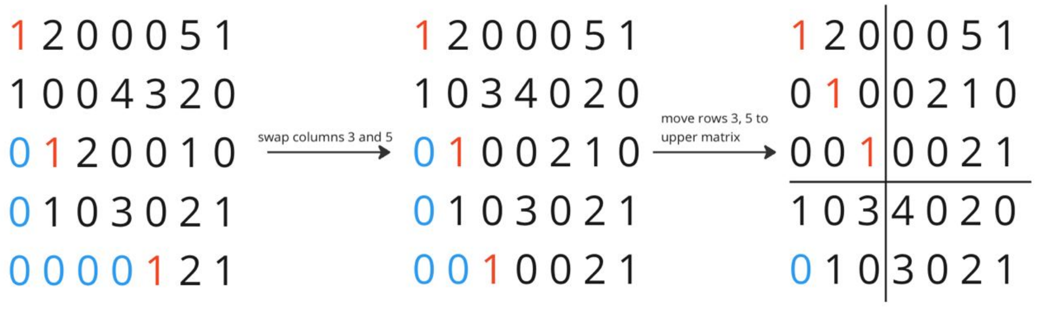

Figure 2: GBLA reordering

After reordering, Faugère-Lachartre idea proposes to replace the Gaussian elimination with the following process:

$$\begin{pmatrix} A & B \\ C & D \end{pmatrix} \xrightarrow{TRSM} \begin{pmatrix} I & A^{-1}B \\ C & D \end{pmatrix} \xrightarrow{AXPY} \begin{pmatrix} I & A^{-1}B \\ 0 & D - CA^{-1}B \end{pmatrix}.$$

Due to the construction of A, it is an upper triangular matrix and therefore always invertible. The last step of the process is Gaussian elimination of D' matrix. The new polynomials that will be added to the basis will be represented in the matrix D'.

Let us denote the number of pivots as n_{piv}. Turns out, that $n_{piv} >> n - n_{piv}$ and $n_{piv} >> m - n_{piv}$. For example, one of the matrices in the **Katsura-12** test has a size of 21182×22207 and has 17915 pivots. Now, let us calculate the complexity of such an algorithm.

TRSM takes $O(n_{piv}^2(n - n_{piv}))$, AXPY takes $O(n_{piv}(m - n_{piv})(n - n_{piv}))$ and final Gaussian elimination takes $O((n - n_{piv})^3)$, so total complexity is $O(n_{piv}^2(n - n_{piv})) \approx O(n_{piv}^2)$. Therefore, we have not only decreased the degree, but also the variable from which it is derived. In our case, GBLA allowed to speed up the algorithm by 4 times.

3.2 NOGBLA

In the same paper another approach was proposed. Despite the fact that the proposal was designed to be as optimized as possible, adhering to the principle of doing only what is necessary, it has been disregarded and I have not found any libraries that follow this approach. As the approach has been ignored, it does not have a name, and we will refer to it as **NOGBLA**. The point is that, since the solution lies only in matrix D', there is no need to modify matrices A or B. Rather, it is suggested that the lower rows be edited by the upper rows immediately, and then matrix D' be reduced by itself. This way, resulting matrix looks like $\begin{pmatrix} A & B \\ 0 & red(D - CA^{-1}B) \end{pmatrix}$.

If we try to calculate the complexity of this approach, we may become somewhat confused.
$O(n_{piv} * (m - n_{piv}) * n) + O((n - n_{piv})^3) \approx O(n_{piv} * n)$. It appears to be larger than it originally was. That is correct, the straightforward implementation has slowed the library down significantly.

And here again, we observe a new feature in the matrix structure. Given that the columns are terms and the rows are polynomials, approximately 95% of matrix values in large matrices are zero. So, we can precount non-zero values, and work only with them. Note, that we can't precount non-zero values for GBLA approach, since the whole matrix is changing. This optimization makes NOGBLA approach 25% faster than GBLA.

The discussion on matrix optimizations in F4 once again demonstrates the importance of thoroughly examining input data for optimization purposes.

4 Other algoritmic optimizations

Most of the time was dedicated to working with matrices and software optimization. However, there were also significant algorithmic optimizations.

4.1 0-reduction

Sometimes S-pairs are reduced to 0. In this case, our basis is not updated. Such reductions could be considered counterproductive, as they waste time and do not provide any additional information. Therefore, a significant portion of the discussion was devoted to predicting 0-reductions, specifically, identifying some criteria by which a particular S-pair could be immediately eliminated. Two of the most well-known are the gcd and the lcm Buchberger's criteria. These criteria are referred to by different names in various sources, so it is necessary to define them. We will write $F \xrightarrow{G} q$ if G reduces F to q.

Definition 3 *Buchberger's gcd criteria.*

If p_1 and p_2 are polynomials, with $gcd(lt(p_1), lt(p_2)) = 1$, than $S(p_1, p_2) \xrightarrow{\{p_1,p_2\}} 0$.

Definition 4 *Buchberger's lcm criteria.*

If p_1, p_2 and p_3 are polynomials in F, and $S(p_1, p_2) \xrightarrow{F} 0$ and $S(p_2, p_3) \xrightarrow{F} 0$ and $lt(p_2) \mid lcm(lt(p_1), lt(p_3))$ then $S(p_1, p_3) \xrightarrow{F} 0$.

These two criteria are essential components of any algorithm for finding Gröbner bases, as without them, the algorithm would take significantly longer to complete. However, for a considerable period of time, there has been a comprehensive criterion that integrates and supplements both Buchberger's criteria (Perry, 2010).

Definition 5 *Extended criteria If p_1, p_2 and p_3 are polynomials in F, and $t_1 = lt(p_1), t_2 = lt(p_2), t_3 = lt(p_3)$. We say that they meet the **extended criteria** if both conditions are fulfilled.*

- *$t2 \mid gcd(t_1, t_3)$ or $lcm(t_1, t_3) \mid t_2$.*
- *For every variable x_i one of the conditions is met:*

- $min(t_1[i], t_3[i]) = 0$
- $t_2[i] \leq max(t_1[i], t_3[i])$

Despite the fact that this criteria was stronger, for some reason, it was not given the attention it deserved even years later. After we implemented this additional criterion, the algorithm started to run at approximately twice the speed. This demonstrates once again that a hidden gem can be found everywhere and may already have been described in previous works.

4.2 SIMPLIFY

In the literature about the F4 algorithm and its implementations, **Simplify** method is found at every step. This method retains all the original polynomials entered into the matrix, as well as the result of reduction. Subsequently, it attempts to reuse these outcomes in order to reduce polynomials even more quickly. Some publications even claim that the speed of F4 is due precisely to this technique. However, my results strongly contradict this hypothesis.

There are several reasons for this. First, in some way, when working on F4 and optimizing matrix calculations, it is forgotten that this is not the sole purpose of the algorithm. As a consequence, by implementing Simplify on top of GBLA, the following situation arises — preprocessing of matrix takes longer, than matrix calculations. See Figure 3.

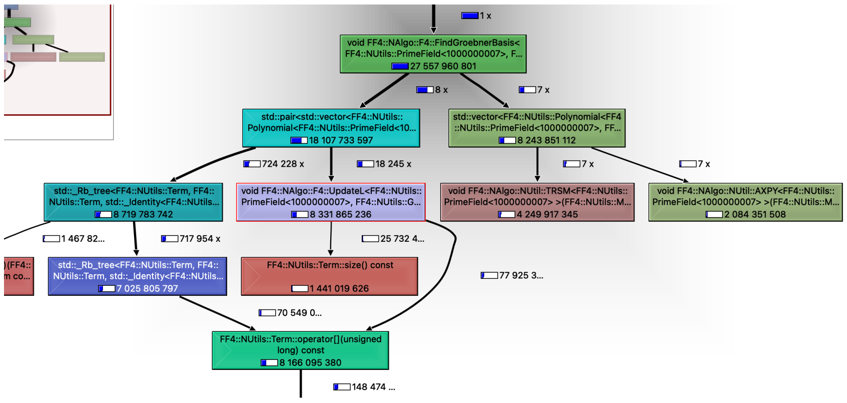
1 x
void FF4::NAlgo::F4::FindGroebnerBasis< FF4::NUtils::PrimeField<1000000007>, F...
27 557 960 801
8 x
7 x
std::pair<std::vector<FF4::NUtils:: Polynomial<FF4::NUtils::PrimeField<10...
18 107 733 597
std::vector<FF4::NUtils::Polynomial<FF4 ::NUtils::PrimeField<1000000007>, FF...
8 243 851 112
724 228 x
18 245 x
7 x
7 x
std::_Rb_tree<FF4::NUtils::Term, FF4:: NUtils::Term, std::_Identity<FF4::NUtils...
8 719 783 742
void FF4::NAlgo::F4::UpdateL<FF4::NUtils:: PrimeField<1000000007>, FF4::NUtils::G...
8 331 865 236
void FF4::NAlgo::NUtil::TRSM<FF4::NUtils:: PrimeField<1000000007> >(FF4::NUtils::M...
4 249 917 345
void FF4::NAlgo::NUtil::AXPY<FF4::NUtils:: PrimeField<1000000007> >(FF4::NUtils::M...
2 084 351 508
1 467 82...
717 954 x
25 732 4...
)(FF4:: m co...
std::_Rb_tree<FF4::NUtils::Term, FF4:: NUtils::Term, std::_Identity<FF4::NUtils...
7 025 805 797
FF4::NUtils::Term::size() const
1 441 019 626
77 925 3...
70 549 0...
FF4::NUtils::Term::operator[](unsigned long) const
8 166 095 380
148 474 ...

Figure 3: Algorithm profile with Simplify and GBLA

Second, even when using Simplify, the GBLA is still slower than a simple NOGBLA approach. It is not possible to effectively combine Simplify with NOGBLA, because most of the matrix remains unchanged in this approach.

The third and final point is that, aside from the general statements that Simplify is beneficial and efficient, you will not find anything. There are no comparisons to other approaches, nor are there any details such as profiling provided.

To be completely honest, Simplify does indeed speed up matrix calculations. However, unfortunately, the process of preparing matrices becomes prohibitively expensive, and as a result, this approach is not viable.

Therefore, we not only abandoned the traditional method, but we also explained why it was ineffective. This reminds us that one should not take information at face value, even if it is presented in a reputable publication, but rather should verify the information for themselves. This is especially easy if you need only to write code in order to verify the approach.

5 C++ OPTIMIZATIONS

As the code is all written in C++ and the goal is to have the fastest algorithm possible, software optimizations are inevitable.

When reviewing other libraries, you can notice thousands of lines of code and a significant number of (occasionally unnecessary) abstractions as well as detailed tests. This indicates that the developers took their work quite seriously. However, the issue with mathematical computation software is that mathematical optimizations alone are not sufficient.

An experienced developer would immediately recognize the significant inefficiency. This is why I believed it is possible to implement the fastest F4 algorithm from scratch. Specifically, we are referring to the std::list data structure. If you encounter std::list in your code, you may always be able to improve the performance of the code, often significantly. This is because modern processors are actually limited not by calculation, but by memory access. Since std::list allocates memory at random locations in the heap, accessing this memory can be very slow. std::list is very not-cache-friendly.

Instead, we stored the data using a std::vector — the most cache-friendly data structure of all.

It is also worth noting that the most effective code is code that performs no unnecessary operations, including avoiding data copying where possible. Sometimes, it is very easy to avoid copying and instead use links. However, sometimes it can be challenging. This is due to complex scenarios, which we will discuss.

5.1 HashMaps and Maps

Sometimes you need to map your data. C++ provides several standard data structures for this purpose, but it can be challenging to utilize them effectively. Let's look at a specific example.

You need to be able to store meta-information for strings of any length and quickly retrieve this information for a specific string. The most straighforward approach, is Map:

```
std::map<std::string, info> mp;
```

Let's denote the size of the map by T, and the lookup key length as S.

The language standard says that the lookups will be logarithmic. And, indeed, since map is a balanced binary tree, we will descend to a depth of no more than $O(log(T))$. But to understand which node to go down to, we need to compare the keys at each level. Therefore, the true asymptotics will be $O(Slog(T))$. It looks bad. Then, the first thought would be to switch to HashMap:

```
std::unordered_map<std::string, info> mp;
```

We know that the average lookup takes O(1). At this point we are happy, why not?

The key point is how a HashMap operates. Initially, it simply calculates the hash value from the key and searches for a location to store the key-value pair. However, due to the possibility of collisions, the HashMap must maintain a copy of the key in order to perform further comparisons.

And this is another area where we can achieve success. In our situation, since everything is stored within the std::vector available to us, there is no need to copy anything. The key to the solution is pointers!

```
std::unordered_map<std::string_view, info> mp;
```

Now, we have lookups for O(1) and do not copy strings! A similar approach, applied to vectors, resulted in a two-fold difference between the first and final implementation.

5.2 Know your STL

Even if you have experience programming in a given language for a significant period of time, it is possible that you may not be aware of not only certain complex aspects of the language's syntax and semantics, but also of the full range of standard library functions and data structures available. The same applies to the libraries used.

In symbolic preprocessing, it is necessary to be able to move nodes from one set to another. Straightforward approach looks like:

```
T a = first.begin();
first.erase(first.begin());
second.insert(a);
Process(a);
```

First of all, you are copying the value to a. The second issue is not significant, but set::node will need to be recreated. In the std::set class, there is a member function called *extract* that unlinks a node from the set and returns a reference to it. So, with this method you can rewrite the code like this:

```
const T& a = *first.begin();
auto extracted = first.extract(first.begin());
second.insert(std::move(extracted));
Process(a);
```

With this approach, we don't copy the value and link already created set::node to the second set. This contributed a negligible 2-3% to the overall performance, but more importantly, it resulted in well-structured and well-written code.

6 Profiling

As the goal is to create the fastest possible library, it is evident that a significant amount of time has been spent on profiling. You have already seen one such profile when the code was functioning and performing well enough. The profiles were generated using the standard valgrind tool on large test cases, and then analyzed using the qcachegrind utility. In this section, we will present two more profiles.

The first profile was generated on F4 algorithm, after all proposed optimizations, with the GBLA method. See Figure 4.

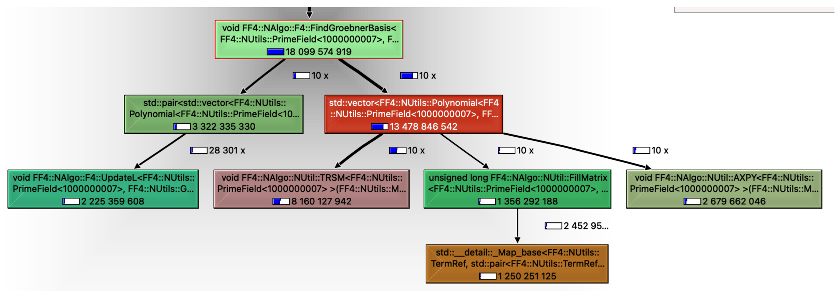
void FF4::NAlgo::F4::FindGroebnerBasis<
FF4::NUtils::PrimeField<1000000007>, F...
18 099 574 919
10 x
10 x
std::pair<std::vector<FF4::NUtils::
Polynomial<FF4::NUtils::PrimeField<10...
3 322 335 330
std::vector<FF4::NUtils::Polynomial<FF4
::NUtils::PrimeField<1000000007>, FF...
13 478 846 542
28 301 x
10 x
10 x
10 x
void FF4::NAlgo::F4::UpdateL<FF4::NUtils::
PrimeField<1000000007>, FF4::NUtils::G...
2 225 359 608
void FF4::NAlgo::NUtil::TRSM<FF4::NUtils::
PrimeField<1000000007> >(FF4::NUtils::M...
8 160 127 942
unsigned long FF4::NAlgo::NUtil::FillMatrix
<FF4::NUtils::PrimeField<1000000007>, ...
1 356 292 188
void FF4::NAlgo::NUtil::AXPY<FF4::NUtils::
PrimeField<1000000007> >(FF4::NUtils::M...
2 679 662 046
2 452 95...
std::__detail::_Map_base<FF4::NUtils::
TermRef, std::pair<FF4::NUtils::TermRef...
1 250 251 125

Figure 4: Optimized algorithm profile with GBLA

On this profile, it can be seen that the algorithm is functioning as expected. Matrix calculations consume most of the processing time, with only 17% dedicated to matrix preparation. In matrix computing, the TRSM algorithm takes up 60% of the processing time.

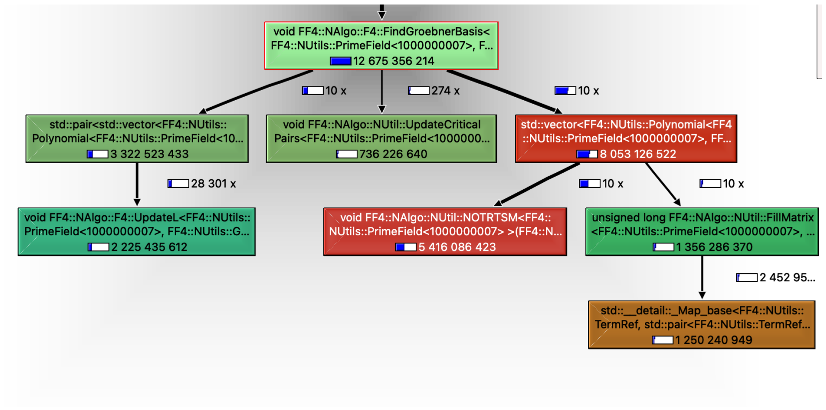
void FF4::NAlgo::F4::FindGroebnerBasis< FF4::NUtils::PrimeField<1000000007>, F...
12 675 356 214
10 x
274 x
10 x
std::pair<std::vector<FF4::NUtils:: Polynomial<FF4::NUtils::PrimeField<10...
3 322 523 433
void FF4::NAlgo::NUtil::UpdateCritical Pairs<FF4::NUtils::PrimeField<1000000...
736 226 640
std::vector<FF4::NUtils::Polynomial<FF4 ::NUtils::PrimeField<1000000007>, FF...
8 053 126 522
28 301 x
10 x
10 x
void FF4::NAlgo::F4::UpdateL<FF4::NUtils:: PrimeField<1000000007>, FF4::NUtils::G...
2 225 435 612
void FF4::NAlgo::NUtil::NOTRTSM<FF4:: NUtils::PrimeField<1000000007> >(FF4::N...
5 416 086 423
unsigned long FF4::NAlgo::NUtil::FillMatrix <FF4::NUtils::PrimeField<1000000007>, ...
1 356 286 370
2 452 95...
std::__detail::_Map_base<FF4::NUtils:: TermRef, std::pair<FF4::NUtils::TermRef...
1 250 240 949

Figure 5: Final profile of the F4 algorithm

The profile of the final version has a 1.5-fold reduction in the number of hits, which is consistent with the benchmark results. Matrix calculations are also 60% faster than in previous version with GBLA. See Figure 5.

7 Results

Finally, we can discuss benchmarks. The algorithm was implemented in a library format and benchmarked on a MacBook M2 Pro 2023. The benchmark was conducted with the highest priority, in order to minimize any interruptions from the operating system. It was compared to openf4 - the fastest open source f4 algorithm. All results are in milliseconds. See Table 1.

Table 1: Benchmarks of F4 for prime field $p = 10^9 + 7$, grevlex ordering

Test	**Our::F4**	**openf4**
sym3-3	0.058	0.312
cyclic4	0.006	0.353
cyclic5	0.207	1.02
cyclic6	2.07	5.25
cyclic7	58.3	189
katsura4	0.058	0.317
katsura5	0.524	0.791
katsura9	101	91.5
katsura10	540	522
katsura11	3187	3403
katsura12	19654	21175

Please note that, in addition to being more efficient than openf4, our solution allows for customizations of the type and the order. And all this with less code and more readability.

8 Conclusions

The results of the study indicate that even complex algorithms, for which numerous studies have been conducted and numerous implementations have been developed, can be created from scratch, and in some cases, be faster than other libraries. To accomplish this, it is essential to thoroughly investigate the literature, question every statement, and scrutinize every idea, sometimes even those that may be underestimated. Furthermore, this outcome demonstrates that efficient computing systems cannot exist without both algebraic and software optimizations.

References

Brice Boyer, Christian Eder, Jean-Charles Faugere, Sylvian Lachartre, and Fayssal Martani. Gbla: Gröbner basis linear algebra package. In *Proceedings of the ACM on International Symposium on Symbolic and Algebraic Computation*, pp. 135–142, 2016.

Bruno Buchberger. Ein algorithmus zum auffinden der basiselemente des restklassenringes nach einem nulldimensionalen polynomideal. *Ph. D. Thesis, Math. Inst., University of Innsbruck*, 1965.

Christian Eder. Computing gröbner bases – a short overview. URL `https://caramba.loria.fr/sem-slides/201409111030.pdf`.

Thomas W Dubé. The structure of polynomial ideals and gröbner bases. *SIAM Journal on Computing*, 19(4):750–773, 1990.

Jean-Charles Faugere. A new efficient algorithm for computing grobner basis. *(F4)*, 2002.

Rüdiger Gebauer and H Michael Möller. On an installation of buchberger's algorithm. *Journal of Symbolic computation*, 6(2-3):275–286, 1988.

Hoon Hong and John Perry. Are buchberger's criteria necessary for the chain condition? *Journal of Symbolic Computation*, 42(7):717–732, 2007.

Hoon Hong and John Perry. Corrigendum to? are buchberger? s criteria necessary for the chain condition??[j. symbolic comput. 42 (2007) 717? 732]. *Journal of Symbolic Computation*, 43(3): 233, 2008.

Ernst W Mayr and Albert R Meyer. The complexity of the word problems for commutative semigroups and polynomial ideals. *Advances in mathematics*, 46(3):305–329, 1982.

Dylan Peifer. The f4 algorithm. 2017.

John Perry. An extension of buchberger's criteria for gröbner basis decision. *LMS Journal of Computation and Mathematics*, 13:111–129, 2010.

A Appendix

To begin with, I would like to highlight additional literature that I referred to when doing the work, but which has not yet been mentioned in the text.

If you wish to become familiar with F4, it would be advisable to start not with the original paper, but rather with a review paper that carefully delves into the topic - Peifer (2017).

A comprehensive presentation detailing numerous optimizations to the F4 algorithm - Christian Eder, with an explanation of the GMI(Gebauer & Möller, 1988).

I would also like to draw attention to two works by the author of the extended criteria on related topics - Hong & Perry (2007; 2008).

You may have noticed that, in the paper itself, we do not discuss the asymptotic behavior of the algorithm in its entirety. This is because the analysis is too complex. You can find some of the relevant estimates in the following publications — Dubé (1990); Mayr & Meyer (1982).

If some theoretical aspects of the main body of work were not clear, it would be sufficient to familiarize oneself with the concept of a Grebner basis in any textbook, and then with the aforementioned paper by Peifer on F4.

In my work, due to space limitations, I have omitted the order of monomials in polynomials. For example, I have not included the grevlex ordering, which is the fastest, and therefore all performance tests have been conducted using this order.

I would also like to draw attention to the test families. The two main tests for Groebner bases are the cyclic and katsura tests. However, their definitions can be incredibly difficult to locate. We will provide them here in order to make the task more manageable for future generations.

$$Cyclic(n) = \begin{cases} x_0 + x_1 + \ldots + x_{n-1} = 0 \\ i = 2, \ldots n-1 : \sum_{j=0}^{j=n-1} \prod_{k=j}^{j+i-1} x_{k \ mod \ n} = 0 \\ x_0 x_1 x_2 \ldots x_{n-1} - 1 = 0 \end{cases}$$

So, for example, $Cyclic(4) = \begin{cases} x_0 + x_1 + x_2 + x_3 = 0 \\ x_0x_1 + x_1x_2 + x_2x_3 + x_3x_0 = 0 \\ x_0x_1x_2 + x_1x_2x_3 + x_2x_3x_0 + x_3x_0x_1 = 0 \\ x_0x_1x_2x_3 - 1 = 0 \end{cases}$

Definition of katsura is a little harder:

for $m \in \{-n+1, \ldots, n-1\}$

$\sum_{l=-n}^{l=n} u(l)u(m-l) = u(m)$

$\sum_{l=-n}^{l=n} u(l) = 1$

$u(l) = u(-l)$

$u(l) = 0$ for $|l| > n$

So, for example, $Katsura(4) = \begin{cases} x_0^2 + 2x_1^2 + 2x_2^2 + 2x_3^2 = x_0 \\ 2x_0x_1 + 2x_1x_2 + 2x_2x_3 = x_1 \\ x_1^2 + 2x_0x_2 + 2x_1x_3 = x_2 \\ x_0 + 2x_1 + 2x_2 + 2x_3 = 1 \end{cases}$

Additionally, I would like to include a small amount of code in the document. Here is the implementation of several methods.

NOGBLA has **NOTRSM** method, instead of TRSM method of GBLA. It looks like:

```
template <typename TCoef>
void NOTRSM(Matrix<TCoef>& matrix,
            size_t pivots,
            const std::vector<std::vector<size_t> >& nnext) {
    for (size_t i = 0; i < pivots; i++) {
        const auto& next = nnext[i];
        for (size_t j = pivots; j < matrix.N_; j++) {
            if (matrix(j, i) != 0) {
                TCoef factor = matrix(j, i);
                for (size_t k = 0; k < next.size(); k++) {
                    matrix(j, next[k]) -= factor * matrix(i, next[k]);
                }
            }
        }
    }
}
```

Here, *nnext* is the precounted vector of non-zero values.

I did not mention this previously, but in order to complete the matrix more efficiently, my algorithm fills in the correct order immediately, rather than rearranging columns and rows afterwards. This is done as follows:

```
template <typename TCoef, typename TComp>
size_t FillMatrix(TPolynomials<TCoef, TComp>& F,
                Matrix<TCoef>& matrix,
                std::vector<Term>& vTerms,
                const std::vector<Term>& diffSet,
                std::vector<std::vector<size_t> >& nnext) {
    size_t cnt = 0;
    size_t swp = 0;
    std::vector<bool> not_pivot(F.size());
    TTermHashSet leadingTerms;
```

```
        // storing leading terms, to find pivots
    std::unordered_map<Term, size_t> Mp;
        // mapping term -> column
    for (size_t i = 0; i < F.size(); i++) {
        auto [_, inserted] =
            leadingTerms.insert(F[i].GetLeadingTerm());
        if (!inserted) {
            // not pivot
            not_pivot[i] = true;
            swp++;
            continue;
        }
        // pivot
        Mp[F[i].GetLeadingTerm()] = cnt;
        vTerms[cnt] = F[i].GetLeadingTerm();
        cnt++;
    }

    cnt = diffSet.size() - 1;
    for (auto& term : diffSet) {
        if (Mp.find(term) == Mp.end()) {
            Mp[term] = cnt;
            // Rearrange columns
            vTerms[cnt] = term;
            cnt--;
        }
    }

    nnext.reserve(F.size() - swp);
    for (size_t i = 0, j = 0; i < F.size(); i++) {
        if (not_pivot[i]) {
            j++;
            continue;
        }
        // Fill pivot row
        std::vector<size_t> next;
        next.reserve(F[i].GetMonomials().size());
        for (const auto& m : F[i].GetMonomials()) {
            const auto& term = m.GetTerm();
            size_t column = Mp[term];
            matrix(i - j, column) = m.GetCoef();
            next.push_back(column);
        }
        // non-zero precalc
        nnext.push_back(std::move(next));
    }

    for (size_t i = 0, j = 0; i < F.size(); i++) {
        if (!not_pivot[i]) {
            continue;
        }
        // Fill non-pivot row
        for (const auto& m : F[i].GetMonomials()) {
            const auto& term = m.GetTerm();
            matrix(F.size() - 1 - j, Mp[term]) = m.GetCoef();
        }
        j++;
    }

    return F.size() - swp;
}
```

At the end of this work, I would like to state that using the optimizations mentioned above, even Buchberger's algorithm performed many times faster than its counterparts and even outperformed some F4 implementations. Therefore, it can be concluded that the above approach was successful.