Practical Gröbner Basis Computation

Bjarke Hammersholt Roune[*]
Cornell University
Department of Mathematics
Ithaca 14853-4201, USA
www.broune.com

Michael Stillman[†]
Cornell University
Department of Mathematics
Ithaca 14853-4201, USA
mike@math.cornell.edu

ABSTRACT

We report on our experiences exploring state of the art Gröbner basis computation. We investigate signature based algorithms in detail. We also introduce new practical data structures and computational techniques for use in both signature based Gröbner basis algorithms and more traditional variations of the classic Buchberger algorithm. Our conclusions are based on experiments using our new freely available open source standalone C++ library.

1. INTRODUCTION

Since F5 [9], there have been several signatured-based algorithms. Recently Gao, Volny and Wang (GVW) [10] have introduced a signature algorithm that generalizes several previous algorithms. Arri and Perry (AP) [1] have published a very similar algorithm. We will refer to both algorithms as *SB* for Signature Basis algorithm.

We give another way of describing SB (Section 2) and we employ a rewriting criterion that improves on that of GVW. We show how this rewriting criterion can be used to eliminate S-pairs (Section 3.2). This enables us to characterize the number of S-pair reductions that SB performs in terms of the final basis and the initial submodule of the module of syzygies (Theorem 6).

We introduce new practical data structures and computational techniques for use in both signature based Gröbner basis algorithms and more traditional variations of the classic Buchberger algorithm (sections 3 and 4).

Our experiments are based on our new freely available open source standalone C++ library (Section 5) [19]. The library was written with the intention to use it in Macaulay 2, but it does not depend on any component of Macaulay 2 and could be used by any other system as well. It is also possible to take just the data structures from the library and use them in another codebase. We welcome all to read, use and modify the code. We are happy to receive suggestions for improvements.

Due to the page limit the paper is of necessity compressed. **The proofs and other supporting material are available in appendices that do not appear in the printed paper**, but that are available online [19].

2. THE SB ALGORITHM

In this section we introduce a simplest possible version of SB from first principles. The description differs from both GVW and AP though the results are not new. See Appendix A for proofs.

2.1 Notation and Terminology

Let R be a polynomial ring over a field. Let $\mathcal{G}$ be a finite set of non-zero polynomials of R indexed as $g_1, \ldots, g_m$. Consider the free module R^m and define the homomorphism $u \mapsto \overline{u}\colon R^m \to R$ by $\overline{u} := \sum_{i=1}^m u_i g_i$. We say that u is a *representation* of $\overline{u}$. Then by definition $\langle \mathcal{G} \rangle := \overline{R^m} = \{\overline{u} \,|u \in R^m\}$.

Let $\boldsymbol{e}_1, \ldots, \boldsymbol{e}_m$ be the standard basis of unit vectors in R^m. Let $\leq$ denote two different term orders – one on R^m and one on R. We require these two orders to be related such that $a \leq b \Leftrightarrow a\boldsymbol{e}_i \leq b\boldsymbol{e}_i$ for all monomials a, b and $i = 1, \ldots, n$. For both orders we use the convention that $0 < t$ for all terms t.

Let the *signature* $\mathfrak{s}(u)$ be the $\leq$-maximal term of $u \in R^m$ and define $\mathfrak{s}(0) = 0$. Let the *lead term* $\mathrm{in}(f)$ be the $\leq$-maximal term of $f \in R$ and define $\mathrm{in}(0) = 0$. In this way every $u \in R^m$ has two main associated characteristics – the signature $\mathfrak{s}(u)$ and the lead term $\mathrm{in}(\overline{u})$ of its image in R.

We will consider extensions $\mathcal{G}_n \supseteq \mathcal{G}$ of additional non-zero polynomials from $\langle \mathcal{G} \rangle$ indexed as $g_1, \ldots, g_m, g_{m+1}, \ldots, g_n$ where $m \leq n$. Define $v \mapsto \overline{v}$, the basis $\boldsymbol{e}_1, \ldots, \boldsymbol{e}_n$ and representation relative to $\mathcal{G}_n$ as above. Each extension $\mathcal{G}_n$ will come with a homomorphism $\phi\colon R^n \to R^m$ such that $\overline{u} = \overline{\phi(u)}$. Then in particular $\phi(\boldsymbol{e}_i)$ for $i > m$ is a representation of g_i in terms of $g_1, \ldots, g_m$. We extend the definition of signature from R^m to R^n by $\mathfrak{s}(u) := \mathfrak{s}(\phi(u))$. A *syzygy* is a $u \in R^n$ or $u \in R^m$ such that $\overline{u} = 0$. None of the $\boldsymbol{e}_i$ are syzygies as $\overline{\boldsymbol{e}_i} = g_i \neq 0$.

Note that the signature of $u \in R^n$ depends on ϕ even if that is not apparent from the notation $\mathfrak{s}(u)$. An element $f \in R$ can have many different representations in R^n with distinct signatures. We write $a \simeq b$ for two terms a and b if $a = sb$ where s is a non-zero element of the ground field. So $a \leq b \leq a$ if and only if $a \simeq b$.

A *monomial* is a polynomial with exactly one term. A

[*] Supported by The Danish Council for Independent Research | Natural Sciences.

[†] Partially supported by NSF grant DMS-1002210

ISSAC 2012 Grenoble, France

monomial or term with a coefficient of 1 is *monic*. Neither terms nor monomials are necessarily monic. In particular $\mathfrak{s}(u)$ and $\mathrm{in}(\overline{u})$ for $u \in R^n$ are not necessarily monic.

2.2 Division With Signatures

SB uses a notion of division called $\mathfrak{s}$*-division*. It is similar to classic polynomial division. The main difference is that we start with an element $u \in R^n$ instead of an element of R and we take signature into account. The result of division of $u \in R^n$ by $\mathcal{G}_n$ is a quotient $q \in R^n$ and a remainder $r \in R^n$ such that

1. $u = q + r$,
2. $\mathrm{in}(\overline{u}) \geq \mathrm{in}(q_i g_i)$ for $i = 1, \ldots, n$,
3. if $\mathfrak{s}(u) \geq \mathfrak{s}(a\boldsymbol{e}_i)$ for a monomial a then $\mathrm{in}(\overline{a\boldsymbol{e}_i})$ does not equal any term of $\overline{r}$,
4. $\mathfrak{s}(u) \geq \mathfrak{s}(q)$.

We $\mathfrak{s}$*-divide* to get the quotient q and we $\mathfrak{s}$*-reduce* to get the remainder r. u is *reduced* if $q = 0$ and otherwise u is *reducible*. We say that u *reduces to zero* if $\overline{r} = 0$.

The first three conditions are similar to conditions on the quotient and remainder in classic polynomial division. The fourth condition disallows some reduction steps. Without signatures the condition for g_i to reduce a term t of f is just that $\mathrm{in}(g_i)|t$. We would then determine the monomial a such that $\mathrm{in}(ag_i) = t$ and reduce to $f - ag_i$.

With signatures it is not sufficient that $\mathrm{in}(g_i)|t$ in order for $\boldsymbol{e}_i$ to reduce a term t of $\overline{u}$. Here $\boldsymbol{e}_i$ reduces t when there is a term a such that $\mathrm{in}(\overline{a\boldsymbol{e}_i}) = t$ and $\mathfrak{s}(a\boldsymbol{e}_i) \leq \mathfrak{s}(u)$. The outcome of such a reduction step is then $u - a\boldsymbol{e}_i$. So a reduction step happens when it can be carried out without strictly increasing the signature. A reduction step is *singular* if $\mathfrak{s}(a\boldsymbol{e}_i) \simeq \mathfrak{s}(u)$. When $\boldsymbol{e}_i$ reduces t it is convenient to also say that $a\boldsymbol{e}_i$ reduces t so that a is introduced right then.

In *top* $\mathfrak{s}$*-reduction* the reduction stops when the lead term of $\overline{u}$ cannot be reduced. We say that u is *top reducible* if the lead term of $\overline{u}$ can be reduced and otherwise u is *top reduced.*

In $\mathfrak{s}$-division the signature is not allowed to increase. *Regular division* is like $\mathfrak{s}$-division except that the coefficient of the signature is not allowed to change either. So the difference is that singular reduction steps are not allowed.

A basis $\mathcal{G}_n$ is a *signature Gröbner basis* if all $u \in R^n$ $\mathfrak{s}$-reduce to zero. Then $\mathcal{G}_n$ maps to a Gröbner basis $\overline{\mathcal{G}_n}$ as then all elements of $\langle\overline{\mathcal{G}_n}\rangle = \overline{R^n}$ reduce to zero.

2.3 S-pairs

The *S-pair* between $\boldsymbol{e}_i$ and $\boldsymbol{e}_j$ is $\mathcal{S}(i,j) := \mathcal{S}(\boldsymbol{e}_i, \boldsymbol{e}_j) := \frac{\mathrm{in}(g_j)}{d}\boldsymbol{e}_i - \frac{\mathrm{in}(g_i)}{d}\boldsymbol{e}_j$ where $d := \gcd(\mathrm{in}(g_i), \mathrm{in}(g_j))$. If $\mathfrak{s}(a\boldsymbol{e}_i) \simeq \mathfrak{s}(b\boldsymbol{e}_j)$ then the S-pair is *singular* and otherwise it is *regular.* By "S-pair" we always mean "regular S-pair".

SB reduces S-pairs using regular reduction steps and adds the regular reduced result to the basis if it is not a syzygy and not singular top reducible. Theorem 1 implies that when all S-pairs have been processed in this fashion, then the basis is a signature Gröbner basis.

THEOREM 1. *Let T be a term of R^m. Assume for all S-pairs p with $\mathfrak{s}(p) \leq T$ that if p' is the result of regular reducing p, then p' is singular top reducible or a syzygy. Then all elements $u \in R^n$ with $\mathfrak{s}(u) \leq T$ $\mathfrak{s}$-reduce to zero.*

The outcome of polynomial reduction depends on the choice of reducer, so the choice of reducer can change what the intermediate basis is in the classic Buchberger algorithm. In contrast to this, Lemma 2 implies that the regular reduction of S-pairs has a uniquely determined remainder.

LEMMA 2. *Let $L \in R^m$ be a term such that all $v \in R^n$ with $\mathfrak{s}(v) < L$ $\mathfrak{s}$-reduce to zero. Let $a, b \in R^n$ such that $\mathfrak{s}(a) = \mathfrak{s}(b) \leq L$. Then* $\mathrm{in}(\overline{a}) = \mathrm{in}(\overline{b})$ *if a and b are regular top reduced. Also, $\overline{a} = \overline{b}$ if a and b are regular reduced.*

Theorem 3 implies that the S-pairs of a signature Gröbner basis give rise to a Gröbner basis of the syzygy module of $\mathcal{G}$. Note that we are talking about the syzygy module of the *original* basis $\mathcal{G}$ rather than the syzygy module of a Gröbner basis of the same ideal. The former is generally much harder to compute than the latter. GVW present this result, while AP essentially prove it but do not state it.

THEOREM 3. *Let $\mathcal{G}_n$ be a signature Gröbner basis and let $u \in R^m$ be a syzygy. Then there is an S-pair p that regular reduces to a syzygy p' such that $\mathfrak{s}(p')$ divides $\mathfrak{s}(u)$.*

SB is known to terminate — see Appendix A.4.

2.4 Pseudo code

Here is pseudo code for a *simplest possible version* of SB. This pseudo code should not be taken as a guide to efficient implementation. The code computes a signature Gröbner basis $\mathcal{G}_n$ and the initial submodule H of the syzygy module of $g_1, \ldots, g_m$. An actual implementation would keep track of monic pairs (h_i, s_i) where $h_i := \overline{\boldsymbol{e}_i}$ and $s_i := \mathfrak{s}(\boldsymbol{e}_i)$ instead of maintaining a full representation $\phi(\boldsymbol{e}_i)$ of each g_i.

```
**SignatureBuchberger**($\{g_1, \ldots, g_m\} \subseteq R$)
  $n \leftarrow m$
  $S \leftarrow \{\mathcal{S}(i,j) \mid 1 \leq i < j \leq m$ and $\mathcal{S}(i,j)$ is regular$\}$
  $H \leftarrow \langle 0 \rangle \subseteq R^m$
  while $S \neq \emptyset$ do
    $p \leftarrow$ an element of $S$ with $\leq$-minimal signature
    $S \leftarrow S \setminus \{p\}$
    $p' \leftarrow$ result of regular reducing $p$
    if $\overline{p'} = 0$ then
      $H \leftarrow H + \langle \mathfrak{s}(p') \rangle$
    else if $p'$ is not singular top reducible then
      $n \leftarrow n + 1$
      $\phi(\boldsymbol{e}_n) \leftarrow \phi(p')$   {implies $g_n = \overline{p'}$ and $\mathfrak{s}(\boldsymbol{e}_n) = \mathfrak{s}(p')$}
      $S \leftarrow S \cup \{\mathcal{S}(i,n) \mid i < n$ and $\mathcal{S}(i,j)$ is regular$\}$
    end if
  end while
```

3. SIGNATURE ALG. IMPROVEMENTS

In this section we show techniques that improve signature Gröbner basis computation. Several of these techniques apply to signature algorithms in general rather than just SB.

3.1 S-pair Elimination

There are many S-pairs that it is not necessary for SB to reduce in order to arrive at a signature Gröbner basis. We say that we *eliminate* an S-pair when we determine that it is not necessary to reduce that S-pair. The S-pair elimination criteria that we present here in Section 3.1 are already present in both GVW and AP.

Three things can happen when SB regular reduces an S-pair in signature T and gets a remainder r. First, r might be a syzygy in which case its signature is added to the set of known syzygy signatures. Second, r might be singular top reducible in which case r is thrown away. Third, if r is not a syzygy and not singular top reducible, then r is added to the basis. For these three cases we say respectively that T is a *syzygy*, *singular* or *basis* signature. These notions are well defined since Lemma 2 implies that regular reduction of S-pairs yields a uniquely determined remainder.

Recall from the definition of S-pair that any mention in this paper of S-pairs refers to regular S-pairs. So SB can right away eliminate any S-pair that is not regular.

If there is more than one S-pair in signature T, then we only have to reduce one of them since Lemma 2 implies that they will all have the same remainder upon regular reduction. Section 3.2 develops this topic further.

Suppose that T is an S-pair signature and we know of a syzygy signature L that divides T. Then T is a syzygy signature by Corollary 4, which allows us to eliminate all S-pairs in signature T. Call this the *signature criterion.* Since SB considers S-pairs in ascending order of signature, we observe that the only syzygy signatures that the signature criterion might not eliminate are those that come from an element of a minimal Gröbner basis of the module of syzygies.

Corollary 4. *Let $u \in R^n$ such that all $v \in R^n$ with $\mathfrak{s}(v) < \mathfrak{s}(u)$ reduce to zero. Suppose there exists a syzygy $h \in R^n$ whose signature divides the signature of u. Then u regular reduces to zero.*

Table 2 gives information about how many S-pairs are eliminated due to each criterion. The criteria are checked in the given order. Appendix B.1 has more details.

3.2 Rewriting and the Singular Criterion

We present a technique that makes reductions easier to perform and that provides a criterion for eliminating all S-pairs of singular signature.

If there are two or more S-pairs in the same signature T, then we only have to regular reduce one of them. Since reduction proceeds by decreasing the lead term, we can heuristically speed up reduction by choosing an S-pair p whose lead term $\mathrm{in}(\overline{p})$ is minimal. Both GVW and AP make this suggestion. In F5, the situation where one S-pair is preferred over another is called a *rewriting criterion.*

Selecting the minimum lead term S-pair would require us to calculate the lead term of each S-pair. If $\mathfrak{s}(\mathcal{S}(\alpha,\beta)) = \mathfrak{s}(t\alpha)$, then we get the same result from regular reducing $\mathcal{S}(\alpha,\beta)$ as for regular reducing $t\alpha$. So we should select the term from M with minimal lead term, where $M := \{t\boldsymbol{e}_i \,|t \text{ is a monomial and } \mathfrak{s}(t\boldsymbol{e}_i) = T\}$. Let $t\boldsymbol{e}_i$ be an element of M with minimal lead term. Note that $t\boldsymbol{e}_i$ might not come from any S-pair in signature T. If $t\boldsymbol{e}_i$ is not regular top reducible, then we know that T is a singular signature, so no regular reduction need take place. We call this criterion for eliminating S-pairs the *singular criterion.* Corollary 5 implies that the singular criterion eliminates an S-pair if and only if it is of singular signature.

AP independently came up with the idea of the singular criterion, and they additionally remark that if $t\boldsymbol{e}_i$ does not come from an S-pair in signature T, and if $t\boldsymbol{e}_i$ has a strictly lower lead term than any element of M that does come from an S-pair in signature T, then the singular criterion will apply. An implication of that is that if $\mathfrak{s}(\mathcal{S}(\alpha,\beta)) = \mathfrak{s}(q\alpha)$ and there exists a $w\boldsymbol{e}_k$ such that $\mathfrak{s}(w\boldsymbol{e}_k) = \mathfrak{s}(q\alpha)$ and $\mathrm{in}(\overline{w\boldsymbol{e}_k}) < \mathrm{in}(\overline{q\alpha})$ then we can eliminate $\mathcal{S}(\alpha,\beta)$ right away.

Corollary 5. *Let p be an S-pair and let p' be the result of regular reducing p. Let M be the finite set*

$$M := \{a\boldsymbol{e}_i \,|a \text{ is a monomial and } \mathfrak{s}(a\boldsymbol{e}_i) = \mathfrak{s}(p)\}.$$

Then all elements of M regular reduce to p'. Also, p' is singular top reducible if and only if some element of M is regular top reduced.

3.3 Koszul Syzygies for S-Pair Elimination

The *Koszul syzygy* between $\boldsymbol{e}_i$ and $\boldsymbol{e}_j$ is $\mathcal{K}(i,j) := g_j\boldsymbol{e}_i - g_i\boldsymbol{e}_j$. If $\mathfrak{s}(g_j\boldsymbol{e}_i) \not\simeq \mathfrak{s}(g_i\boldsymbol{e}_j)$ then the Koszul syzygy is *regular.* By "Koszul syzygy" we always mean "regular Koszul syzygy".

The signature of $\mathcal{K}(i,j)$ is $\max(\mathrm{in}(g_i)\,\mathfrak{s}(\boldsymbol{e}_j), \mathrm{in}(g_j)\,\mathfrak{s}(\boldsymbol{e}_i))$. We can use the Koszul syzygy to eliminate all S-pairs in that signature. We call this S-pair elimination criterion the *Koszul criterion.* The idea goes back to at least F5.

In the classic Buchberger algorithm, we can eliminate an S-pair between two polynomials if their lead terms are relatively prime. In SB this is a special case of the Koszul criterion, since then $\mathfrak{s}(\mathcal{K}(i,j)) = \mathfrak{s}(\mathcal{S}(i,j))$. Even so, we call this the *relatively prime criterion* and do not consider such S-pairs to be eliminated by the Koszul criterion.

Many S-pairs that could be eliminated by the Koszul criterion are already eliminated by the signature criterion. We consider such S-pairs to be eliminated by the signature criterion rather than the Koszul criterion. The signature criterion eliminates all syzygy signatures that are divisible by some other syzygy signature. So the Koszul criterion eliminates those signatures that are the signature of a Koszul syzygy, that are minimal among all syzygy signatures and that are not eliminated by the relatively prime criterion.

A straight forward way to make use of Koszul syzygies would be to insert all Koszul syzygy signatures into the set of known syzygies and let the signature criterion also work with Koszul syzygies. Since Koszul syzygy signatures are rarely minimal syzygy signatures, this can greatly increase the size of the set of known syzygies. We can remove the Koszul signatures that are non-minimal among the syzygy signatures that we know at any given point, but many of the signatures that are left after that will still not be minimal among the set of all syzygy signatures. So adding the Koszul signatures to the set of known syzygy signatures can cause significant overhead in space for storing all these signatures, and significant overhead in time for checking all these signatures when using the signature criterion.

An implication of the discussion so far is that the Koszul criterion only eliminates an S-pair that would not already have been eliminated by the signature criterion when the Koszul signature is equal to the S-pair signature. This implies that we can make full use of the Koszul criterion by maintaining a priority queue of Koszul signatures. The priority queue allows us to determine the minimum Koszul signature in the queue. S-pairs are processed in increasing order of signature, so if we have gotten to an S-pair with signature T and the minimum Koszul signature L is less than T, then we can throw L away since it will never equal any future S-pair signatures. If $T = L$ then we can eliminate the S-pair using the Koszul criterion. If $T < L$ then the Koszul criterion cannot eliminate the S-pair.

The priority queue approach does not alleviate the memory overhead of Koszul syzygies. We still have to construct all the Koszul signatures, which by itself can take a lot of time. One observation that helps is that if the S-pair $\mathcal{S}(i,j)$ is eliminated by the signature criterion using a known syzygy signature T, then $T | \mathfrak{s}(\mathcal{S}(i,j)) | \mathfrak{s}(\mathcal{K}(i,j))$. So if $\mathcal{S}(i,j)$ is eliminated by the signature criterion, then there is no reason to construct the Koszul signature $\mathfrak{s}(\mathcal{K}(i,j))$.

Another observation is that $\mathfrak{s}(\mathcal{S}(i,j)) \leq \mathfrak{s}(\mathcal{K}(i,j))$. S-pairs are processed in order of increasing signature, so if $\mathcal{K}(i,j)$ ends up eliminating an S-pair, then that S-pair must have a higher signature than $\mathcal{S}(i,j)$ does. So we do not need to construct $\mathcal{K}(i,j)$ until we process the S-pair $\mathcal{S}(i,j)$. If the S-pair is eliminated or reduces to zero, then we do not have to construct the Koszul syzygy at all. These two observations reduce the overhead in time and space of the Koszul criterion to almost nothing.

We can now characterize how many S-pairs SB reduces when using these S-pair elimination techniques.

THEOREM 6. *Let $\mathcal{G}_n$ be a minimal signature Gröbner basis. Let H be the initial submodule of the module of syzygies of $\mathcal{G}$. Then SB reduces one S-pair for each element of $\mathcal{G}_n \setminus \mathcal{G}$. SB also reduces one S-pair for each minimal generator of H that is not the signature of any Koszul syzygy among the elements of $\mathcal{G}_n$. SB reduces no more S-pairs than that.*

Table 2 shows that the relatively prime and Koszul criteria eliminate only a small proportion of all S-pairs. However, for several examples, that is still a significant proportion of the amount of S-pairs that are reduced, so these criteria can have a significant impact on the final number of reductions.

3.4 Compare Ratios Instead of Signatures

SB can spend a lot of time comparing signatures. We present a technique for replacing many of these signature comparisons with comparison of just a single integer.

We start with the example of computing the signature of an S-pair $p := \mathcal{S}(i,j)$. Let $A := \mathfrak{s}(\boldsymbol{e}_i)$, $a := \mathrm{in}(\overline{\boldsymbol{e}_i})$, $B := \mathfrak{s}(\boldsymbol{e}_j)$ and $b := \mathrm{in}(\overline{\boldsymbol{e}_j})$. Then $\mathfrak{s}(p)$ is the larger one of $E := \frac{b}{\gcd(a,b)}A$ and $F := \frac{a}{\gcd(a,b)}B$, so a straight forward way of getting $\mathfrak{s}(p)$ is to compute E and F and then compare the two to see which is larger. Computing $\mathfrak{s}(p)$ in this way can take a lot of time when there are many basis elements. For a faster solution, observe that (we allow negative exponents)

$$E < F \quad \Leftrightarrow \quad bA < aB \quad \Leftrightarrow \quad \tfrac{A}{a} < \tfrac{B}{b}.$$

So if we store the ratio of signature to lead term (the *sig-lead ratio*) with each basis element, then we can determine which of E and F is larger by comparing the stored ratios instead of having to compute both E and F. The next step is then to compute only the one out of E and F that is the signature. The signature of the Koszul syzygy $\mathcal{K}(i,j)$ is the larger one of bA and aB, so the same technique works there.

We speed up the comparison of sig-lead ratios by associating an integer τ_i to each basis element $\boldsymbol{e}_i$ such that $\tau_i < \tau_j \Leftrightarrow \frac{A}{a} < \frac{B}{b}$. In this way sig-lead ratio comparisons can be done as an integer comparison, which is much faster. To support fast insertions of new basis elements we use a simple approach based on spacing the integers far apart to begin with and just rebuilding the whole datastructure if a new basis element would need to have an integer between k and $k+1$ for some k. There exists a faster approach [3] for updating the integers τ_i, but we have not needed it.

Ratio comparisons also occur in $\mathfrak{s}$-reduction. Suppose we want to $\mathfrak{s}$-reduce the lead monomial of u by a basis element e_i. Let $A := \mathfrak{s}(u)$, $a = \mathrm{in}(\overline{u})$, $B = \mathfrak{s}(\boldsymbol{e}_i)$ and $b = \mathrm{in}(\overline{\boldsymbol{e}_i})$. Suppose that $b|a$. Then the $\mathfrak{s}$-reduction can be performed if $\frac{a}{b}B \leq A$, which is equivalent to $\frac{B}{b} \leq \frac{A}{a}$. Unfortunately, the sig-lead ratio $\frac{A}{a}$ has to be computed for each term being reduced. In our experiments it has been slower to compute an appropriate integer to associate to the ratio than to just use $\frac{A}{a}$ directly for comparisons. This is still faster than deciding $\frac{a}{b}B < A$ directly since that expression involves a division and a multiplication for every comparison.

Ratio comparisons are also relevant to finding the best module term to reduce as described in Section 3.2. We need to find the element of M from Corollary 5 with minimum lead term. Each basis element α whose signature $A := \mathfrak{s}(\alpha)$ divides T contributes the element $A' := \frac{T}{A}\alpha$ to M. The lead term is $\mathrm{in}(\overline{A'}) = \frac{T}{A}a$ where $a := \mathrm{in}(\overline{\alpha})$. If β is another basis element that also contributes an element B' to M, then we need to perform the lead term comparison

$$\tfrac{T}{A}a < \tfrac{T}{B}b \quad \Leftrightarrow \quad \tfrac{a}{A} < \tfrac{b}{B} \quad \Leftrightarrow \quad \tfrac{A}{a} > \tfrac{B}{b},$$

which can be done as a sig-lead ratio comparison.

The sig-lead ratio comparison technique also applies to what we call *base divisors* — see Section 3.5.

3.5 Base Divisors for S-Pair Elimination

When a new element is added to the basis, we construct the S-pairs between the new element and all the previous elements. In many cases almost all of these S-pairs can be eliminated right away by the signature criterion, so if there are many S-pairs then just constructing the signature for each S-pair and then checking the signature criterion on that signature can take up a lot of time.

We present a new S-pair elimination criterion that we call the *base divisor criterion*. The new criterion is strictly weaker than the signature criterion, but it is faster to check.

We check the base divisor criterion before the signature criterion, so all the S-pairs that the base divisor criterion eliminates then do not need to be checked by the slower signature criterion. Table 2 shows that the base divisor criterion eliminates a substantial amount of S-pairs. Table 3 shows the drop in performance if we do not check the base divisor criterion before using the signature criterion. For yang1 the base divisors are a 35% performance improvement, and they eliminate 71% of the S-pairs.

Let β be a new basis element that we have just added to the basis. We consider the S-pairs $\mathcal{S}(\beta,\gamma)$ between β and each other basis element γ. We aim to eliminate $\mathcal{S}(\beta,\gamma)$ without computing the signature $\mathfrak{s}(\mathcal{S}(\beta,\gamma))$.

The idea here is to consider a fixed previous basis element α that has certain properties. We call α a *base divisor*. We would like it to be true that $\mathfrak{s}(\mathcal{S}(\alpha,\gamma)) | \mathfrak{s}(\mathcal{S}(\beta,\gamma))$, since then we can eliminate $\mathcal{S}(\beta,\gamma)$ when $\mathcal{S}(\alpha,\gamma)$ has a syzygy signature. We use a triangle of bits, one bit for each S-pair, so we can see in constant time if we know $\mathcal{S}(\alpha,\gamma)$ to have a syzygy signature. The criterion has two parts depending on whether γ has a high or low sig-lead ratio (see Section 3.4).

High Ratio Base Divisors

A basis element α can be used as a *high ratio base divisor* when $\mathrm{in}(\overline{\alpha}) | \mathrm{in}(\overline{\beta})$. A high ratio base divisor α can eliminate $\mathcal{S}(\beta,\gamma)$ when γ has higher sig-lead ratio than both of α and β. Theorem 7 spells out the precise criterion.

Theorem 7. *Let $\alpha, \beta, \gamma \in R^n$ such that* $\mathrm{in}(\overline{\alpha})\,|\,\mathrm{in}(\overline{\beta})$ *and* $\frac{\mathfrak{s}(\gamma)}{\mathrm{in}(\overline{\gamma})} > \frac{\mathfrak{s}(\alpha)}{\mathrm{in}(\overline{\alpha})}, \frac{\mathfrak{s}(\beta)}{\mathrm{in}(\overline{\beta})}$. *Then* $\mathfrak{s}\left(\mathcal{S}\left(\alpha, \gamma\right)\right) | \, \mathfrak{s}\left(\mathcal{S}\left(\beta, \gamma\right)\right)$.

We can use a kd-tree (see Section 4.2) on the initial terms of the basis elements to quickly determine all the possible base divisors. The base divisor that will eliminate the most S-pairs is the one with the highest sig-lead ratio, so we use that one. Sometimes there is no high ratio base divisor.

Low Ratio Base Divisors

A basis element α can be used as a *low ratio base divisor* when $\mathfrak{s}(\alpha) \,|\, \mathfrak{s}(\beta)$. A low ratio base divisor α can in some cases eliminate $\mathcal{S}(\beta, \gamma)$ when γ has lower sig-lead ratio than both of α and β. Theorem 8 spells out the precise criterion.

Theorem 8. *Let $\alpha, \beta, \gamma \in R^n$ such that $\mathfrak{s}(\alpha) \,|\, \mathfrak{s}(\beta)$ and* $\frac{\mathfrak{s}(\gamma)}{\mathrm{in}(\overline{\gamma})} < \frac{\mathfrak{s}(\alpha)}{\mathrm{in}(\overline{\alpha})}, \frac{\mathfrak{s}(\beta)}{\mathrm{in}(\overline{\beta})}$. *Let* $x^p := \frac{\mathrm{in}(\overline{\alpha})\,\mathfrak{s}(\beta)}{\mathfrak{s}(\alpha)}$, $x^a := \mathrm{in}(\overline{\alpha})$ *and* $x^b := \mathrm{in}(\overline{\beta})$. *Define v by $v_i := \infty$ for $b_i \leq p_i$ and $v_i :=$* $\max(p_i, a_i)$ *otherwise. Then* $\mathfrak{s}\left(\mathcal{S}\left(\alpha, \gamma\right)\right) | \, \mathfrak{s}\left(\mathcal{S}\left(\beta, \gamma\right)\right)$ *if and only if* $\mathrm{in}(\overline{\gamma})|x^v$.

To use Theorem 8 to eliminate $\mathcal{S}(\beta, \gamma)$, we have to check if $\mathrm{in}(\overline{\gamma})|x^v$. Since v does not depend on γ, we need only compute it once. We need not check that $\mathrm{in}(\overline{\gamma})|x^v$ if $\frac{\mathfrak{s}(\alpha)}{\mathrm{in}(\overline{\alpha})} | \frac{\mathfrak{s}(\beta)}{\mathrm{in}(\overline{\beta})}$ since then $v_i = \infty$ for every entry, but such α are rare.

In order for the sig-lead ratio requirement to be satisfied as often as possible, we choose an α with maximum sig-lead ratio. A kd-tree (see Section 4.2) on the basis signatures can quickly find all the possible low ratio base divisors.

The numbers in Table 3 and Table 4 are based on two base divisors per β, as that minimized the total runtime.

4. DATA STRUCTURES

We present data structures that are useful both for classic Buchberger algorithms and for signature algorithms.

4.1 Ordering Terms During Reduction

Both polynomial reduction and $\mathfrak{s}$-reduction operate by having a *current polynomial* f and adding monomial multiples mg_i to f. The basic operations for keeping track of the terms of f are to extract the maximal remaining term of f and to add polynomials of the form mg_i to f. So we need a *priority queue* on the terms of f. Adding elements to a priority queue is called a *push* while removing the maximal element is called a *pop*. We investigate priority queues for keeping track of the terms of f.

One solution is to store f directly as a polynomial whose terms are sorted. Yan pointed out that this can be very slow, as we can end up looking at every term of f for every insertion even when mg_i has only two terms. Yan introduced the *geobucket priority queue* which alleviates this issue [20].

Heaps are a popular priority queue. Monagan and Pearce present experiments that indicate that heaps are better than geobuckets for polynomial multiplication and division [18].

The priority queue on terms of f can contain terms a and b such that $a \simeq b$. We would like to replace such a and b with $a+b$ so that we have fewer terms to order which is faster and takes less memory. Fateman investigates the idea of using a hash table on the terms in the priority queue to collect like terms [7]. Hash tables do not order their entries, so it is still necessary to keep a separate priority queue. We say that a priority queue is *hashed* if it uses a hash table in front of the priority queue. Fateman reports that a hashed priority queue is not the best option for monomial multiplication due to the overhead imposed by the hash table.

Johnson had the idea that instead of keeping track of the terms of mg_i, we could instead have a priority queue containing only the maximal term of mg_i [14]. Once that term is extracted, we would then insert the second-most-maximal term of mg_i and so on. This requires annotating values in the priority queue with information about m, about g_i and about which term is the next one. In this way the priority queue will contain fewer elements which implies fewer comparisons and a smaller memory footprint. We say that a priority queue using this idea is *compressed*, since it compresses information from all of mg_i into a single entry.

When a compressed item in a priority queue is replaced by its successor term, then we are replacing the maximal value in the priority queue with a smaller value. Call this pop-push operation *replace-top*. Many priority queues can do a replace-top operation faster than a pop followed by a push. Heaps are one example. The *tournament tree* is especially good at replace-top operations. For that reason we investigate using tournament trees in polynomial division.

We have implemented a heap, a geobucket and a tournament tree for use in polynomial division as well as hashed and compressed versions of those data structures. We have made considerable effort to implement these data structures in an efficient manner — see Appendix C.1. We have focused on the general case, so we have not used packed representations of the monomials.

Table 4 compares combinations of these techniques. When two terms are compared, they might be determined to be equal. In that case the two terms can be replaced by their sum, though handling this imposes an overhead. In Table 4 the row "dedup" indicates whether duplicates are removed in this way. In our experiment the hashed heap, geobucket and tournament tree have similar performance and they are faster than the other options. Whether the dedup and compression options are an advantage depends on the particular configuration that they are applied to — see Table 4. We do not list times for dedup in combination with hashing, since hashing removes all duplicates.

4.2 Monomial Ideal Data Structures

Monomial ideal computations occur in several places in both signature and classic Gröbner basis algorithms. The most apparent example is in reduction, where it is necessary to determine a basis element whose lead term divides the term being reduced. This involves deciding the membership problem on the monomial ideal that is generated by the lead terms of the basis elements. We call this operation a *divisor query*. We investigate data structures for divisor queries and related operations. See Appendix C.3 for more details.

A straight forward divisor query algorithm is to check every monomial in the data structure for whether it divides the query monomial. We call this scheme a *monomial list*.

Milowski proposed the *monomial tree* data structure [17]. The monomial tree is a *trie* on the exponent vectors of the monomial ideal. Milowski shows that this data structure can be significantly faster than a monomial list in many cases. Unfortunately the monomial tree degenerates into a higher-overhead monomial list if all the monomials have distinct exponents of the first variable.

The toric Gröbner basis implementation 4ti2 uses an unpublished binary tree data structure due to Peter Malkin that we will call a *support tree*. Monomials are stored in the leaves. The leaf that a monomial a goes into depends on the support of the exponent vector of a. Starting at the root of the tree, go to the right child if $x_1|a$ and otherwise go left. Do the same thing at the next node for x_2 and so on. A leaf is split into two smaller leaves if it contains too many monomials. This data structure works well for toric ideals as about half the exponents are zero. Unfortunately the support tree degenerates into a higher-overhead monomial list if most of the monomials have similar support.

We propose the use of *kd-trees*. Kd-trees are used extensively in computer graphics to keep track of sets of points. The exponent vector of a monomial is also a point, so kd-trees can also be used as a data structure for monomial ideals. Both the monomial tree and the support tree can be described as special cases of kd-trees.

Kd-trees are binary trees. In our kd-tree implementation, the monomials are in the leaves and each interior node contains a pure power x_i^k. A monomial a goes into the right subtree if $x_i^k|a$ and otherwise it goes into the left subtree. When looking for a divisor of a monomial a, we then do not need to consult the right subtree if x_i^k does not divide a.

Divmasks are a widely known technique to speed up divisor queries. In the most general terms, a divmask involves a function d from monomials to the set of vectors $\{0,1\}^k$ such that if $a|b$ then $d(a) \leq d(b)$. We call such a function a *divmap*. The idea is then that if $d(a) \not\leq d(b)$, then we already know that a does not divide b so we do not have to check it. Furthermore, checking if one 0-1 vector is dominated by another can be done very quickly on a computer by letting each entry in the vector $d(a)$ correspond to a bit in a word $w(a)$ of memory. Then $d(a) \leq d(b)$ if and only if the bitwise-and of $w(a)$ with the bitwise negation of $w(b)$ is zero. In C notation this is `(a & ~b) == 0`.

In our implementation the divmaps $d_{x_i^t}$ are parametrized by a pure power x_i^t. Then $d_{x_i^t}(a) = 1$ if $x_i^t|a$. We choose the divmaps based on the monomials in the data structure and periodically recalculate the divmaps so that they are always appropriate for the monomials in the datastructure.

The divmask version of our monomial list keeps a divmask for each monomial. The divmasks eliminate around 98% of all divisibility checks for most examples when using the monomial list — see Appendix C.3. We also have a divmask version of our kd-tree where the internal nodes have a divmask of the gcd of the monomials in that subtree, and the leaves also have a divmask for each monomial. The subtree rooted at a node does not have to be searched for a divisor of a if the divmask at that node implies that the gcd of the monomials in the leaves does not divide a.

Table 3 shows the performance of divmasks, kd-trees and monomial lists. The baseline is a divmask kd-tree. The kd-tree and the divmask monomial list are both an improvement on a monomial list, and the combination of both techniques (baseline) is the fastest in every case.

4.3 The S-pair Triangle

There are $\binom{n}{2}$ S-pairs among n basis elements, and for large n the time spent on ordering those S-pairs that are not eliminated can be significant. Ordering S-pairs by signature are a requirement in SB. It it not required but is still a good idea for the classic Buchberger algorithm since it can be a large speed up to reduce certain S-pairs first. Other than the time spent on ordering S-pairs, just storing the S-pairs can also consume a large amount of memory; especially so for signature algorithms since signature Gröbner bases are larger than minimal Gröbner bases. We present an S-pair data structure that is fast and uses little memory.

We want to order S-pairs according to some total order $\prec$. We need a data structure that can give us the minimum S-pair according to $\prec$, and the data structure needs to support insertion of new S-pairs every time a new element is added to the basis. So what we need is a *priority queue* on S-pairs.

A straight forward solution is to use a heap, geobucket or tournament tree (see Section 4.1) to order the S-pairs. A problem here is that S-pairs are frequently ordered according to a monomial, such as in the case of signature algorithms, so it is necessary to store a monomial for every S-pair in the queue in order to allow fast comparison by $\prec$. For a basis with 10,000 elements in 50 variables that requires storing up to 5 billion exponents, which at 4 bytes per exponent translates into 20GB of memory. Many of those S-pairs are likely going to be eliminated, but the memory overhead is still 4GB even if 80% of the S-pairs are eliminated before putting them into the queue.

We propose the *S-pair triangle*, which is a priority queue for S-pairs that only needs to store a single integer per S-pair in the queue. It is based on the observation that new S-pairs are constructed in large batches every time a new element is added to the basis. We sort the new batch of S-pairs according to $\prec$ and maintain a (small) priority queue on the $\prec$-minimal element from each batch. If we place the basis elements in a row, and we place each sorted batch as a column above the corresponding basis element, then we get the triangle shape that the data structure is named after.

The minimal S-pair in the small priority queue (that has an element from each column) is also the minimal S-pair over all. To extract the minimal S-pair p, we remove it from the small priority queue and insert the next-smallest element from the column that p comes from in the triangle. This is analogous to the compressed priority queues from Section 4.1, where the columns of the triangle play the same role as reducers do in the compressed priority queues.

The main attractive property of the S-pair triangle is that we can throw out the monomials associated to the S-pairs once they are sorted into columns. We can do so because we only ever need to compare the $\prec$-minimal S-pair from each column with any other S-pair. So instead of having to store up to $\binom{n}{2}$ monomials, we only need to store up to n monomials — one monomial for each of the n columns. Another memory consumption benefit is that all the S-pairs $\mathcal{S}(i,j)$ in column j share the same j, so we only need to store i. So we only need memory for one integer per S-pair.

For very large bases even just one integer per S-pair in the queue can add up to a substantial amount of memory. In our implementation we use a 16 bit integer for the columns of the first $2^{16} = 65,536$ basis elements, and we then use a 32 bit integer for basis elements beyond that. This technique halves the memory used on S-pairs in most cases compared to using a 32 bit integer for all columns.

We have implemented the S-pair triangle with a tournament tree in front and with a heap in front. We have also implemented an S-pair priority queue with all the S-pairs in a heap and in a tournament tree. The baseline is the tournament tree in front of an S-pair triangle. In Table 3 we see that only yang1 and mayr42 stress the S-pair queue. For those two examples we see that the baseline S-pair triangle with a tournament tree in front is the fastest while the S-pair triangle with a heap in front is a little slower. The pure heap and tournament tree are much slower and they make the program consume more than 4 GB of ram on yang1. We conclude that the S-pair triangle is fast and uses little memory and it works well with a tournament tree in front.

5. EXPERIMENTS

We have written an implementation of SB that we use for these comparisons [19]. It has all the improvements that we present in this paper. Its main current weakness is that it does not use F4 reduction. We have also started writing a classic Buchberger implementation for comparison. It benefits from our data structures and the S-pair elimination component is state of the art but it is otherwise naïve.

We have chosen the examples to show a wide range of behaviors. Some of the ideals such as yang1 stress the handling of monomials and divisor queries, while others such as hcyclic8 stress the reduction procedure. We make the input bases for these examples available online [19]. The example joswig101 uses an elimination order that eliminates the first 4 variables, breaking ties with grevlex. The input bases are interreduced.

All benchmarks were run on an Apple MacBookPro intel core i7, at 2.66GHz, with 8 GB of RAM. We use Magma V2.18-1, Maple 14's FGb, Singular 3-1-3, Macaulay2 version 1.4 (Algorithm LinearAlgebra) and our classic Buchberger algorithm implementation "A". The Macaulay 2 times for inhomogeneous ideals are not given as the F4 implementation in Macaulay 2 only works for homogenous ideals currently. The FGb entries with a "*" are cases where Maple initially used FGb to do the calculation, but then terminated FGb and used its own less fast internal code instead.

We would like to give a definitive discussion about the relative merits of the various Gröbner basis algorithms, but we unfortunately find that the task is currently impossible. The best Gröbner basis implementations for the common case of inputs that cause a lot of time to be spent on polynomial reduction are currently those in Magma and FGb, and it is not possible to inspect the source code of either of those systems. As such, there is no way to be certain about what it is that causes these implementations to work well. Collecting experimental data without knowing what the experiment being done actually is is not the highest scientific ideal.

We give an analysis of the differences in Table 1 with the caveat that we must necessarily make assumptions that we cannot verify for the reasons just given.

Our naïve Buchberger implementation does very well on yang1 and mayr42 which we suspect is because the other systems do not use kd-trees for divisor queries and those two ideals stress the divisor query infrastructure due to having many variables and many elements in the final Gröbner basis. It does poorly on the remaining ideals because it is not a mature implementation.

Macaulay 2 does well on hcyclic8. That is because of the use of F4, as the very similar non-F4 Buchberger implementation in M2 (not shown) is much slower on hcyclic8. FGb and M2 use the same amount of time on hcyclic8. Our best guess is that Magma is significantly faster here because its F4 implementation is very good.

Table 3 shows the times for computing signature bases on our set of examples. The baseline algorithm uses a hashed geobucket for $\mathfrak{s}$-reduction, and a divmask kd-tree for divisor queries and all optimizations that we have presented.

The term order on R^m for all these experiments is the Schreyer (induced) order, which is the same as GVW's g2 order. GVW reports and our experience confirms that this is often the best order for computing signature bases. There are notable exceptions. For example, consider the free module order: higher component is greater. On ties, use the Schreyer order. Then the signature basis for the joswig101 example is computed in 6.2 seconds, with 1242 basis elements, much faster than all the Gröbner times reported for that example in Table 1. However, the result is still much larger than the reduced Gröbner basis.

The SB implementation is the fastest on jason210 and performs reasonably on the other ideals except for yang1. The reason for the slow performance on yang1 is that the signature Gröbner basis is much larger than the minimal Gröbner basis for yang1. For the other ideals we suspect that FGb and Magma are faster because they use F4 reduction, so we suspect that the comparison is not useful for determining if SB is faster than F5.

Recall that SB computes the initial module of the syzygy module of the original basis. GVW explain how to compute the Gröbner basis of the syzygy module from this initial syzygy module. Since SB often computes the initial syzygy module in about the same time as it takes to compute a Gröbner basis for most of these examples, SB should be the best algorithm for computing Gröbner bases of syzygy modules.

APPENDIX

A. THE SB ALGORITHM

These appendices contain material that we had to cut to fit the page within the ISSAC page limit of 8 pages. In particular the appendices contain proofs of all the theorems from the paper. The theorem numbers are not consecutive because we repeat the theorems that appear in the main paper and they retain their original number.

A.1 Setup

Our notation and terminology differs from both that of Gao, Volny and Wang and of Arri and Perry. Part of the difference is that we attempt to make the language and description of the SB algorithm as close as possible to that for the classic Buchberger algorithm.

All notation for signature algorithms have used a pair (f, s) where s is the signature of f. Gao, Volny and Wang let $f \in R^n$ rather than $f \in R$. We consider it an advantage not to have three symbols $p = (f, s)$ tied up for every pair, and we also avoid any ambiguity about whether the word "pair" refers to an S-pair or a basis element. We have not used the natural concept that $\text{in}(u) = \mathfrak{s}(u)$, since then we can talk about both the signature $\mathfrak{s}(u)$ and lead term $\text{in}(\overline{u})$ of u without ambiguity. The use of a bar to signify the mapping $u \mapsto \overline{u}$ is also not standard, but it is convenient since our other notation requires heavy use of the mapping.

An implementation of the algorithm does not need to maintain a full representation $u \in R^n$ of each polynomial $\overline{u}$ just because the mathematical arguments concern an element of R^n. An implementation only needs to store $\mathfrak{s}(u)$ and $\overline{u}$.

Example	p	nvars	neqns	homog?	order	nGB	magma	FGb	Sing	M2	A	SB
joswig101	101	5	5	no	elim(4)	5	56.3	n/a	122.6	n/a	*	93.0
jason210	32003	8	3	yes	grevlex	900	6.0	57.5	2.3	12.1	4.6	1.4
katsura10	101	10	10	no	grevlex	272	.5	1.9	7.6	n/a	22.9	2.8
katsura11	101	11	11	no	grevlex	537	3.5	13.2	63.1	n/a	253.0	18.4
hcyclic8	101	9	8	yes	grevlex	1182	3.5	12.5	43.0	12.5	162.0	111.5
yang1	101	48	66	yes	grevlex	4761	29.0	*	85.3	64.1	5.6	1333.0
mayr42	101	51	44	yes	grevlex	8534	54.2	347.0	218.3	89.6	26.9	273.0

Table 1: Input data and time in seconds for several implementations.

	joswig101	jason210	katsura10	katsura11	hcyclic8	yang1	mayr42
#spairs:	1,209,790	987,715	37,950	148,240	14,680,071	1,998,099,720	523,633,341
elim via non-regular criterion	655	1,191	206	698	2,821	111,120	362,703
elim via base divisor criterion	686,300	346,714	14,864	62,634	6,711,383	1,415,552,384	364,970,054
elim via signature criterion	281,682	573,998	10,998	46,170	7,154,919	409,866,276	149,612,312
#spairs queued	241,153	65,812	11,882	38,738	810,948	172,569,940	8,688,272
elim via duplicate signature	219,554	55,475	9,283	32,834	713,321	116,423,105	4,337,133
elim via signature criterion(late)	16,720	5,235	2,110	4,967	75,987	45,461,188	4,036,194
elim via Koszul criterion	11	1	71	88	30	14,165	376
elim via rel. prime criterion	0	2	71	148	7	31,762	5,507
elim via singular criterion(late)	3,101	3,338	0	0	15,430	10,490,908	111,466
#spairs which need reduction	1,767	1,761	347	701	6,173	148,812	197,596
reduce to SB elements	1,551	1,403	266	534	5,411	63,150	32,318
reduce to new syzygy signatures	216	358	81	167	762	85,662	165,278

Table 2: Number of S-pairs eliminated by the various criteria in the SB algorithm.

	joswig101	jason210	katsura10	katsura11	hcyclic8	yang1	mayr42
#SB	1,556	1,406	276	545	5,419	63,216	32,362
#monomials	760,690	519,315	100,626	387,769	3,281,515	1,224,044	64,724
baseline	93	1	3	18	112	1333	273
no fast ratio	134	1	2	20	120	2753	565
no base divisors	90	1	2	19	112	2022	478
early koszul	93	2	2	18	115	1341	337
divmask monomial list	84	2	2	19	139	3917	835
monomial list	179	9	4	37	1270	> 8 hours	> 30 min
kd-tree	100	2	2	19	119	2113	419
spair-tourTree	93	1	2	19	114	> 4 GB	320
spair-heap	118	2	3	24	147	> 4 GB	379
spair-heap-triangle	92	1	2	18	112	1488	277

Table 3: Time in seconds for variants of the SB algorithm.

Reducer	Tour tree						Heap						Geobucket					
Hashed	x	x	.	.	.	.	x	x	.	.	.	.	x	x	.	.	.	.
Dedup	.	.	x	x	.	.	.	.	x	x	.	.	.	.	x	x	.	.
Compressed	x	.	x	.	x	.	x	.	x	.	x	.	x	.	x	.	x	.
joswig101	101	95	247	640	226	614	104	93	173	655	241	609	96	93	124	139	206	439
jason210	1	2	2	4	2	4	1	2	2	4	2	3	1	2	2	2	2	3
katsura10	3	2	12	37	10	33	3	2	6	35	11	31	3	2	4	6	11	21
katsura11	21	19	106	373	91	348	21	19	57	404	105	357	21	19	39	45	97	245
hcyclic8	121	113	538	2062	470	1848	123	117	296	2209	517	1968	125	116	230	258	505	1275
yang1	1330	1330	1329	1335	1330	1336	1330	1332	1329	1334	1330	1334	1337	1339	1338	1335	1338	1335
mayr42	274	273	273	274	273	274	272	279	273	274	273	274	282	273	272	277	273	277

Table 4: Time in seconds using different reducer data structures in the SB algorithm.

If $a \simeq b$ then we say that a and b are *like*, as in the phrase "collecting like terms". In section 2.1 we define the signature of 0. This is never relevant for the SB algorithm as we never add two elements with like signatures, so there is no occasion for the coefficient of the signature to become zero. It might seem that we should simply say that the zero syzygy does not have a signature and leave it at that. However, even non-zero syzygies can have a zero signature. This happens when a non-zero syzygy $v \in R^n$ maps to $\phi(v) = 0 \in R^m$. Even though that is never relevant for the SB algorithm, defining $\phi(v)$ in all cases allows us to avoid addressing the issue in the paper.

A.2 Division With Signatures

The definition of $\mathfrak{s}$-division is intended to be as close as possible to the definition of classic polynomial division. We had originally described classic polynomial division in the paper, and then introduced $\mathfrak{s}$-division after that with as few changes as possible to underscore the analogy. Our original classic polynomial division description is here, as well as pseudo code for classic polynomial division, $\mathfrak{s}$-division and regular division.

We had originally referred to $\mathfrak{s}$-reduction as *signature reduction*, but we adopted Arri and Perry's name for it because it saved space and we had tremendous trouble fitting the paper into the limit of 8 pages. We debated referring to $\mathfrak{s}$-reduction as just *reduction* with no prefix, but that risked confusing $\mathfrak{s}$-reduction with one of regular reduction, classic polynomial reduction and a singular reduction step.

Polynomial division in R

The usual notion of polynomial division is to divide a polynomial $f \in R$ by a finite set of polynomials $\mathcal{G}_n \subseteq R$. This yields a quotient $q \in R^n$ and a remainder $r \in R$ such that

1. $f = \overline{q} + r$,
2. $\mathrm{in}(f) \geq \mathrm{in}(q_i g_i)$ for $i = 1, \ldots, n$,
3. no term of r is divisible by any $\mathrm{in}(g_i)$ for $g_i \in \mathcal{G}_n$.

If f is zero then so are q and r. We *divide* to get the quotient q and we *reduce* to get the remainder r. The process is the same so the distinction between division and reduction is just what part of the result (q, r) that we most care about. We say that f is *reduced* if $q = 0$ and otherwise f is *reducible*.

If $r = 0$ then call q a *representation* of f and then f has a representation. Since polynomial division by something that is not a Gröbner basis does not have a unique outcome, f might have a representation even if the polynomial division algorithm does not arrive at a zero remainder.

To compute (q, r) we perform reduction steps. If t is a term of f then we can reduce that term by $g_i \in \mathcal{G}_n$ when $\mathrm{in}(g_i)$ divides t. Let a be a monomial such that $\mathrm{in}(ag_i) = t$. Then we perform a reduction step by subtracting ag_i from f and adding $a\boldsymbol{e}_i$ to q. We continue this process to complete the reduction.

There is a distinction between reduction and *top reduction*. In top reduction the reduction is halted as soon as the initial term of f cannot be reduced. We say that f is *top reducible* if its initial term can be reduced and otherwise it is *top reduced*.

The following pseudo code implements polynomial division in R. It returns a pair (q, r) where $q \in R^n$ is the quotient and $r \in R$ is the remainder.

```
Reduce($f \in R$)
  $q \leftarrow 0 \in R^n$   {$q$ is the quotient}
  $r \leftarrow 0 \in R$   {$r$ is the sum of those terms of $f$ that we have not been able to reduce}
  while $f \neq r$ do
    $t \leftarrow \mathrm{in}(f - r)$   {$t$ is the maximal term of $f$ that we have not yet processed}
    $d \leftarrow 0$   {$d$ will store any divisor that we may find}
    for $i = 1, \ldots, n$ do {look for a divisor}
      if $\mathrm{in}(g_i) | t$ then
        $d \leftarrow \frac{t}{\mathrm{in}(g_i)} \boldsymbol{e}_i$   {for efficiency you would stop the for loop here}
      end if
    end for
    if $d \neq 0$ then
      $f \leftarrow f - \overline{d}$   {reduce by $d$}
      $q \leftarrow q + d$   {record that we reduced by $d$}
    else
      $r \leftarrow r + t$   {record that the term $t$ could not be reduced}
    end if
  end while
  return $(q, r)$   {$q \in R^n$ is the quotient and $r \in R$ is the remainder}
```

A.2.1 $\mathfrak{s}$-reduction in R^n

The following pseudo code implements $\mathfrak{s}$-reduction in R^n. It returns a pair (q, r) where $q \in R^n$ is the quotient and $r \in R^n$ is the remainder.

```
SReduce($u \in R^n$)
  $T \leftarrow \mathfrak{s}(u)$   {The signature of $u$ can change, so we have to record it}
  $q \leftarrow 0 \in R^n$   {$q$ is the quotient}
  $r \leftarrow 0 \in R$   {$r$ is the sum of those terms of $\overline{u}$ that we have not been able to reduce}
  while $\overline{u} \neq r$ do
    $t \leftarrow \mathrm{in}(\overline{u} - r)$   {$t$ is the maximal term of $\overline{u}$ that we have not yet processed}
    $d \leftarrow 0$   {$d$ will store any divisor that we may find}
    for $i = 1, \ldots, n$ do {look for a divisor}
      if $\mathrm{in}(g_i) | t$ and $\mathfrak{s}\left(\frac{t}{\mathrm{in}(g_i)} \boldsymbol{e}_i\right) \leq \mathfrak{s}(u)$ then
        $d \leftarrow \frac{t}{\mathrm{in}(g_i)} \boldsymbol{e}_i$   {for efficiency you would stop the for loop here}
      end if
    end for
    if $d \neq 0$ then
      $u \leftarrow u - d$   {reduce by $d$}
      $q \leftarrow q + d$   {record that we reduced by $d$}
    else
      $r \leftarrow r + t$   {record that the term $t$ could not be reduced}
    end if
  end while
  return $(q, u)$   {$q \in R^n$ is the quotient and $u \in R^n$ is now the remainder as $\overline{u} = r$}
```

A.2.2 Regular Division in R^n

The result of regular division of $u \in R^n$ by $\mathcal{G}_n$ is a quotient $q \in R^n$ and a remainder $r \in R^n$ such that

1. $u = q + r$,

2. $\mathrm{in}(\overline{u}) \geq \mathrm{in}(q_i g_i)$ for $i = 1, \ldots, n$,

3. if $\mathfrak{s}(u) > \mathfrak{s}(a\boldsymbol{e}_i)$ for a monomial a then $\mathrm{in}(\overline{a\boldsymbol{e}_i})$ does not equal any term of r,

4. $\mathfrak{s}(u) > \mathfrak{s}(q)$.

These conditions are identical to the ones for $\mathfrak{s}$-reduction except that "$\geq$" has been replaced with "$>$" for the last condition. In the same way, the condition for $a\boldsymbol{e}_i$ to regular reduce a term t of $\overline{u}$ becomes

$$\mathrm{in}(\overline{a\boldsymbol{e}_i}) = t \quad \text{and} \quad \mathfrak{s}(a\boldsymbol{e}_i) < \mathfrak{s}(u)\,.$$

Note that $\mathfrak{s}(a\boldsymbol{e}_i) < \mathfrak{s}(u)$ is equivalent to $\mathfrak{s}(u) = \mathfrak{s}(u - a\boldsymbol{e}_i)$. To see this, recall that the signature includes a coefficient. So a regular reduction step happens when it can be carried out without changing the signature.

The pseudo code for $\mathfrak{s}$-reduction can be modified to carry out regular reduction by replacing the line

```
    **if** $\mathrm{in}(g_i)|t$ and $\mathfrak{s}\left(\frac{t}{\mathrm{in}(g_i)}\boldsymbol{e}_i\right) \leq \mathfrak{s}(u)$ **then**
```

with

```
    **if** $\mathrm{in}(g_i)|t$ and $\mathfrak{s}\left(\frac{t}{\mathrm{in}(g_i)}\boldsymbol{e}_i\right) < \mathfrak{s}(u)$ **then**
```

We define *regular division*, *regular reduction*, *regular reduced*, *regular reducible*, *regular top reduced* and *regular top reducible* analogously to how those terms are defined for $\mathfrak{s}$-reduction.

A.3 S-pairs

We give the proofs that are missing from Section 2.3. The arguments are essentially the same as the ones given by Gao, Volny and Wang, though stated in a different way.

THEOREM 1. *Let T be a term of R^m. Assume for all S-pairs p with $\mathfrak{s}(p) \leq T$ that if p' is the result of regular reducing p, then p' is singular top reducible or a syzygy. Then all elements $u \in R^n$ with $\mathfrak{s}(u) \leq T$ $\mathfrak{s}$-reduce to zero.*

PROOF. Suppose to get a contradiction that there is a $u \in R^n$ with $\mathfrak{s}(u) \leq T$ such that u does not $\mathfrak{s}$-reduce to zero. Assume without loss of generality that $\mathfrak{s}(u)$ is $\leq$-minimal such that u does not reduce to zero. We may also assume that u is top reduced.

By Lemma 9 there is an S-pair p whose signature divides $\mathfrak{s}(u)$. Also, ap' is regular top reduced where p' is the result of regular reducing p and a is the monomial such that $\mathfrak{s}(ap) = \mathfrak{s}(u)$.

Now $\mathfrak{s}(ap') = \mathfrak{s}(u)$ and both ap' and u are regular top reduced, so by Lemma 2 we get that $\mathrm{in}(\overline{ap'}) = \mathrm{in}(\overline{u})$. Then anything that top reduces ap' will also top reduce u. We know that ap' is top reducible since p' is top reducible by assumption. Thus u is top reducible which is a contradiction. □

LEMMA 2. *Let $L \in R^m$ be a term such that all $v \in R^n$ with $\mathfrak{s}(v) < L$ $\mathfrak{s}$-reduce to zero. Let $a, b \in R^n$ such that $\mathfrak{s}(a) = \mathfrak{s}(b) \leq L$. Then $\mathrm{in}(\overline{a}) = \mathrm{in}(\overline{b})$ if a and b are regular top reduced. Also, $\overline{a} = \overline{b}$ if a and b are regular reduced.*

PROOF. It suffices to prove that $\overline{a} = \overline{b}$ if a and b are regular reduced as the other statement is a corollary of that.

Suppose to get a contradiction that $\overline{a - b} \neq 0$. As $\mathfrak{s}(a) = \mathfrak{s}(b) = L$ we get that $\mathfrak{s}(a - b) < L$ so $a - b$ reduces to zero. Then in particular $a - b$ is top reducible. Swap a and b if necessary to ensure that $\mathrm{in}(\overline{a - b})$ has the same monic part as a term in a. Then that term of a is *regular* reducible since $\mathfrak{s}(a - b) < \mathfrak{s}(a)$. This contradicts the assumption that a is regular reduced. □

LEMMA 9. *Let $u \in R^n$ be top reduced and have non-zero signature. Assume that all $v \in R^n$ with $\mathfrak{s}(v) < \mathfrak{s}(u)$ reduce to zero. Then there exists an S-pair p whose signature divides the signature of u. Also, cp' is regular top reduced where p' is the result of regular reducing p and c is the monomial such that $\mathfrak{s}(cp) = \mathfrak{s}(u)$.*

PROOF. We construct an S-pair p whose signature divides $\mathfrak{s}(u)$. Then the rest follows from Lemma 10

Consider initial terms: If $\alpha, \beta \in R^n$ are both top reducible and $\mathfrak{s}(\alpha) \leq \mathfrak{s}(\alpha + \beta)$ and $\mathfrak{s}(\beta) \leq \mathfrak{s}(\alpha + \beta)$ then $\alpha + \beta$ is top reducible or $\mathrm{in}(\overline{\alpha}) + \mathrm{in}(\overline{\beta}) = 0$. We will use this argument with $\alpha := u - \mathfrak{s}(u)$ and $\beta := \mathfrak{s}(u)$.

Let $L := \mathfrak{s}(u)$. Then $u - L$ has smaller signature than u does so it reduces to zero and in particular it is top reducible. Also L is top reducible since it top reduces itself. Yet the sum $(u - L) + L = u$ is not top reducible so

$$\mathrm{in}(\overline{u - L}) + \mathrm{in}(\overline{L}) = 0.$$

Construct $\boldsymbol{p}$: Let $a\boldsymbol{e}_i := L = \mathfrak{s}(u)$. As $u - L$ reduces to zero there is a reducer $b\boldsymbol{e}_j$ such that

$$\mathrm{in}(\overline{b\boldsymbol{e}_j}) = \mathrm{in}(\overline{u - L}), \qquad \mathfrak{s}(b\boldsymbol{e}_j) \leq \mathfrak{s}(u - L)\,.$$

Let $p := \mathcal{S}(i, j)$. Then $p = \frac{a}{c}\boldsymbol{e}_i + \frac{b}{c}\boldsymbol{e}_j$ where $c := \gcd(a, b)$ and $\mathfrak{s}(cp) = L$ since

$$\mathrm{in}(\overline{a\boldsymbol{e}_i}) = \mathrm{in}(\overline{L}) = -\,\mathrm{in}(\overline{u - L}) = -\,\mathrm{in}(\overline{b\boldsymbol{e}_j})$$

and

$$\mathfrak{s}(a\boldsymbol{e}_i) > \mathfrak{s}(u - L) \geq \mathfrak{s}(b\boldsymbol{e}_j)\,.$$

So p is an S-pair whose signature divides $L = \mathfrak{s}(u)$. □

LEMMA 10. *Let $L \in R^m$ be a term such that all $v \in R^n$ with $\mathfrak{s}(v) < L$ reduce to zero. Let p be an S-pair whose signature divides L. Then there exists an S-pair q whose signature also divides L and with the following additional property. Let b be the monomial such that $\mathfrak{s}(bq) = L$ and let q' be the result of regular reducing q. Then bq' is not regular top reducible.*

PROOF. Pick a monomial a such that $\mathfrak{s}(ap) = L$ and let p' be the result of regular reducing p. We can assume that ap' is regular top reducible as otherwise we are done. This implies that $a > 1$ so $\mathfrak{s}(p) < L$ so p reduces to zero.

We are now going to construct an S-pair q such that $\mathfrak{s}(bq) = L$ and $\mathrm{in}(\overline{ap}) > \mathrm{in}(\overline{bq})$. If bq' is not regular top reducible then we are done. Otherwise we can do the same thing again to get yet a third S-pair with the same properties and so on. This process must terminate as the initial terms of the S-pair multiples ap, bq, $\ldots$ decrease strictly at each step and there are only finitely many S-pairs (alternatively, $\leq$ is a well order).

Construct $\boldsymbol{c}\boldsymbol{e_i}$: As $\mathfrak{s}(p') = \mathfrak{s}(p) < L$ we know that p' reduces to zero. Yet $\overline{p'}$ is non-zero, so there must be a singular reducer $c\boldsymbol{e}_i$ such that

$$\mathrm{in}(\overline{c\boldsymbol{e}_i}) = \mathrm{in}(\overline{p'}), \qquad \mathfrak{s}(c\boldsymbol{e}_i) \simeq \mathfrak{s}(p')\,.$$

Recall that $\simeq$ is equality up to a non-zero element of the ground field.

Construct $d\boldsymbol{e}_j$: As ap' is regular top reducible there is a regular top reducer $d\boldsymbol{e}_j$ such that

$$\mathrm{in}(\overline{d\boldsymbol{e}_j}) = \mathrm{in}(\overline{ap'}), \qquad \mathfrak{s}\left(d\boldsymbol{e}_j\right) < \mathfrak{s}\left(ap'\right).$$

Construct q: Let $q := \mathcal{S}\left(i, j\right)$. Then $q = \frac{ac}{b}\boldsymbol{e}_i - \frac{d}{b}\boldsymbol{e}_j$ where $b := \gcd(ac, d)$ and $\mathfrak{s}\left(bq\right) \simeq \mathfrak{s}\left(ap'\right) = L$ since

$$\mathrm{in}(\overline{ac\boldsymbol{e}_i}) = \mathrm{in}(\overline{ap'}) = \mathrm{in}(\overline{d\boldsymbol{e}_j}), \qquad \mathfrak{s}\left(ac\boldsymbol{e}_i\right) \simeq \mathfrak{s}\left(ap'\right) > \mathfrak{s}\left(d\boldsymbol{e}_j\right).$$

Then $\mathrm{in}(\overline{d\boldsymbol{e}_j}) > \mathrm{in}(\overline{bq})$ as the initial term cancels in the subtraction $\overline{ac\boldsymbol{e}_i - d\boldsymbol{e}_j} = \overline{q}b$, so as claimed we see that

$$\mathrm{in}(\overline{ap}) \geq \mathrm{in}(\overline{ap'}) = \mathrm{in}(\overline{d\boldsymbol{e}_j}) > \mathrm{in}(\overline{bq}).$$

□

THEOREM 3. *Let $\mathcal{G}_n$ be a signature Gröbner basis and let $u \in R^m$ be a syzygy. Then there is an S-pair p that regular reduces to a syzygy p' such that $\mathfrak{s}\left(p'\right)$ divides $\mathfrak{s}\left(u\right)$.*

PROOF. We see that u is reduced since it is a syzygy. Then by Lemma 9 there exists an S-pair p whose signature divides the signature of u. Also, ap' is regular reduced where p' is the result of regular reducing p and a is the monomial such that $\mathfrak{s}\left(ap\right) = \mathfrak{s}\left(u\right)$. Then Lemma 2 implies that $\overline{ap'} = \overline{u} = 0$. So $\overline{p'} = 0$ and we are done. □

A.4 Termination

Huang proves that a GVW-like algorithm terminates [13], and Gao, Volny and Wang refer to Huang for termination. Huang gives a counterexample to show that SB does not always terminate when the term order on the module and the term order on the ring are not compatible in the sense that $a < b \Leftrightarrow a\boldsymbol{e}_i < b\boldsymbol{e}_i$.

Eder and Perry [6] prove that an F5-like algorithm terminates using an incremental term order on the module ("position over term"). We give Perry and Eder's proof here. Their proof requires no changes to apply to the SB algorithm other than to be adapted to our notation.

We do not need to mention the issue of compatibility of the term orders in Theorem 11 since that assumption already appears in Section 2.1.

THEOREM 11. *Suppose that $\phi\left(\boldsymbol{e}_i\right)$ is not top reducible by $\mathcal{G}_{i-1}$ and that all $v \in R^m$ with $\mathfrak{s}\left(v\right) < \mathfrak{s}\left(\boldsymbol{e}_i\right)$ do $\mathfrak{s}$-reduce to zero with respect to $\mathcal{G}_{i-1}$ for each $i \geq m$. Then the sequence of g_i's is finite.*

PROOF. Let R' be a polynomial ring containing all the variables $x_1, \ldots, x_k$ of R. Also, let R' contain variables y_{ij} for $i = 1, \ldots, m$ and $j = 1, \ldots, k$. Define the function $f : R \times \{\text{terms of } R^m\} \to R'$ by

$$f(g, sx^v\boldsymbol{e}_i) := \mathrm{in}(g)y_i^v y_{i1}$$

where s is a non-zero element of the ground field. Then $f(g, T)|f(g', T')$ if and only if both $\mathrm{in}(g)|\,\mathrm{in}(g')$ and $T|T'$.

Consider the sequence of monomial ideals I_m, I_{m+1}, ... defined by

$$I_n := \langle f(\overline{\boldsymbol{e}_1}, \mathfrak{s}\left(\boldsymbol{e}_1\right)), \ldots, f(\overline{\boldsymbol{e}_n}, \mathfrak{s}\left(\boldsymbol{e}_n\right))\rangle \subseteq R'.$$

If $f(\overline{\boldsymbol{e}_i}, \mathfrak{s}\left(\boldsymbol{e}_i\right))|f(\overline{\boldsymbol{e}_j}, \mathfrak{s}\left(\boldsymbol{e}_i\right))$ for $i < j$ then $\phi\left(\boldsymbol{e}_j\right)$ is top reducible by Lemma 12. We have assumed that none of the $\phi\left(\boldsymbol{e}_j\right)$ are top reducible, so the sequence of monomial ideals I_n is strictly increasing and therefore finite. □

LEMMA 12. *Let $u \in R^n$ such that all $v \in R^n$ with $\mathfrak{s}\left(v\right) < \mathfrak{s}\left(u\right)$ $\mathfrak{s}$-reduce to zero. If $\mathrm{in}(\overline{\boldsymbol{e}_i})|\,\mathrm{in}(\overline{u})$ and $\mathfrak{s}\left(\boldsymbol{e}_i\right)|\,\mathfrak{s}\left(u\right)$ then u is top reducible.*

PROOF. Let a and b be monomials such that

$$\mathrm{in}(\overline{a\boldsymbol{e}_i}) = \mathrm{in}(\overline{u}) \quad \text{and} \quad \mathfrak{s}\left(b\boldsymbol{e}_i\right) = \mathfrak{s}\left(u\right).$$

If $a \leq b$ then $a\boldsymbol{e}_i$ top reduces u and we are done. Otherwise $a > b$ so that

$$\mathrm{in}(\overline{u - b\boldsymbol{e}_i}) = \mathrm{in}(\overline{u}) \quad \text{and} \quad \mathfrak{s}\left(u - b\boldsymbol{e}_i\right) < \mathfrak{s}\left(u\right).$$

Then $u - b\boldsymbol{e}_i$ is top reducible and whatever top reduces it will top reduce u. □

A.5 Pseudo Code

We have to reiterate that the pseudo code given in Section 2.4 is a simplest possible version of the algorithm. Any reasonable implementation would include at least the S-pair elimination criteria introduced in Section 3 as well as many other improvements that have been developed for Gröbner basis computation. The pseudo code is intended as a way to succintly state the essence of the SB algorithm without getting bogged down in the complexities of an efficient implementation.

B. SIGNATURE IMPROVEMENTS

We add notes to the material in Section 3 and give the proofs that we did not have space to include in the main paper.

B.1 S-pair Elimination

Something that we did not have space to dwell on in the main paper is the widely understood distinction between eliminating an S-pair *early* and eliminating it *late*. There are two points in an S-pair's life time where it is natural to try to eliminate it. The first opportunity is right as it gets created, and the second opportunity is right before it would otherwise cause a reduction to be carried out.

Everything else being equal, it is better to eliminate an S-pair early rather than late. The reason for that is that if an S-pair is eliminated late, then it had to be stored for a time, which increases memory consumption, and it had to be compared to other S-pairs to determine which S-pair has the minimal signature, which takes time. Sometimes an S-pair can only be eliminated late, for example when the syzygy signature that eliminates the S-pair is only discovered after the S-pair is constructed.

It would also be possible to eliminate an S-pair sometime between early and late. However, as there are many S-pairs, trying to eliminate them all every time new information comes in would take a lot of time. The S-pair triangle (see Section 4.3) minimizes the overhead of storing many S-pairs, so we are not too concerned about eliminating S-pairs early, even though we do want to do so when possible.

As we state at the end of Section 3.1, Arri and Perry remark [1, Remark 20] that it is possible to apply the singular criterion early. We have implemented this technique and we report times for using it in Table 3. From that table we see that applying the singular criterion early causes our program to run slower, for example for mayr42 the time increases from 273s to 337s. This is due to the extra time it takes to check the singular criterion on all of the S-pairs that are not eliminated by the other early criteria. We have used a kd-tree to check the singular criterion — otherwise it would have been much slower. Applying the singular criterion early does decrease the number of queued S-pairs by 53% for yang1, which is an advantage since it decreases the amount of memory used to store pending S-pairs.

We postpone the duplicate criterion and the relatively prime criterion to maximize the number of S-pairs that are eliminated. Let p and q be two S-pairs with the same signature and suppose that the relatively prime criterion can eliminate p. We can eliminate q due to the duplicate criterion and then eliminate p due to the relatively prime criterion. The order there is important. If we first eliminate p due to the relatively prime criterion, then we will not have any way to eliminate q. We postpone the two criteria so that we can check if any S-pair in a given signature is relatively prime. In contrast, we do not postpone the signature criterion because if it applies to one S-pair in a signature, then it applies to all of them so there is no reason to delay.

Note that criteria that get applied early look more impressive in Table 2 because they get checked for many more S-pairs. For example if an S-pair can be eliminated both by the signature criterion and the Koszul criterion, then that will count only as a hit for the signature criterion since that S-pair is then eliminated so that the Koszul criterion never gets the opportunity to eliminate it.

Corollary 4. *Let $u \in R^n$ such that all $v \in R^n$ with $\mathfrak{s}(v) < \mathfrak{s}(u)$ reduce to zero. Suppose there exists a syzygy $h \in R^n$ whose signature divides the signature of u. Then u regular reduces to zero.*

Proof. Let u' be the result of regular reducing u. Let a be the monomial such that $\mathfrak{s}(ah) = \mathfrak{s}(u)$. Now ah and u' have the same signature and they are both regular reduced so Lemma 2 implies that $\overline{u'} = \overline{ah} = 0$. □

Corollary 5. *Let p be an S-pair and let p' be the result of regular reducing p. Let M be the finite set*

$$M := \{a\boldsymbol{e}_i \,|a \text{ is a monomial and } \mathfrak{s}(a\boldsymbol{e}_i) = \mathfrak{s}(p)\}.$$

Then all elements of M regular reduce to p'. Also, p' is singular top reducible if and only if some element of M is regular top reduced.

Proof. All elements in M regular reduce to p' by Lemma 2. Suppose that $a\boldsymbol{e}_i \in M$ is regular top reduced. Then $\mathrm{in}(\overline{a\boldsymbol{e}_i}) = \mathrm{in}(\overline{p'})$ by Lemma 2 so $a\boldsymbol{e}_i$ top reduces p'. Suppose instead that p' is top reducible. Then there is a singular top reducer $a\boldsymbol{e}_i$ such that $\mathrm{in}(\overline{a\boldsymbol{e}_i}) = \mathrm{in}(\overline{p'})$ and $\mathfrak{s}(a\boldsymbol{e}_i) \simeq \mathfrak{s}(p') = \mathfrak{s}(p)$. Then $a\boldsymbol{e}_i$ is regular top reduced since p' is. Also, there is an element s of the ground field such that $sa\boldsymbol{e}_i \in M$. □

B.2 Base Divisors

The base divisor criterion from Section 3.5 is mainly useful when there are a very large amount of basis elements, as that is when handling S-pairs can take a lot of time. The technique also requires storing a triangle of $\binom{k}{2}$ bits where k is the size of the basis. That can take a significant amount of memory when k is very large. So an overhead in memory is only imposed when there is at the same time a significant advantage in time.

Running out of memory is worse than taking a little longer, since the computation cannot proceed if there is not enough memory. However, if the computer runs out of memory when using the base divisor technique, then it is possible to simply drop the triangle of bits and stop using the base divisor technique from that point onward. So in this way we can view the base divisor technique as a way to use spare memory to speed up the computation, but that memory can be freed if it is needed.

We give the proofs from Section 3.5 that we did not have space to include in the main paper.

Theorem 7. *Let $\alpha, \beta, \gamma \in R^n$ such that $\mathrm{in}(\overline{\alpha})|\,\mathrm{in}(\overline{\beta})$ and $\frac{\mathfrak{s}(\gamma)}{\mathrm{in}(\overline{\gamma})} > \frac{\mathfrak{s}(\alpha)}{\mathrm{in}(\overline{\alpha})}, \frac{\mathfrak{s}(\beta)}{\mathrm{in}(\overline{\beta})}$. Then $\mathfrak{s}(\mathcal{S}(\alpha,\gamma))\,|\,\mathfrak{s}(\mathcal{S}(\beta,\gamma))$.*

Proof. To ease notation, let $x^a := \mathrm{in}(\overline{\alpha})$, $x^b := \mathrm{in}(\overline{\beta})$ and $x^c := \mathrm{in}(\overline{\gamma})$. The assumptions about sig-lead ratios imply that

$$\mathfrak{s}(\mathcal{S}(\alpha,\gamma)) = \frac{\mathrm{in}(\overline{\alpha})}{\gcd(\mathrm{in}(\overline{\alpha}),\mathrm{in}(\overline{\gamma}))}\,\mathfrak{s}(\gamma)$$

and that

$$\mathfrak{s}(\mathcal{S}(\alpha,\gamma)) = \frac{\mathrm{in}(\overline{\beta})}{\gcd(\mathrm{in}(\overline{\beta}),\mathrm{in}(\overline{\gamma}))}\,\mathfrak{s}(\gamma).$$

So in vector notation we need to prove for each i that

$$\min(b_i, c_i) - \min(a_i, c_i) \le b - a.$$

The case $\boldsymbol{a_i, b_i \ge c_i}$: Equivalent to $a_i \le b_i$.
The case $\boldsymbol{a_i > c_i > b_i}$: Does not happen as $a_i \le b_i$.
The case $\boldsymbol{a_i \le c_i \le b_i}$: Equivalent to $c_i \le b_i$.
The case $\boldsymbol{a_i, b_i \le c_i}$: Equivalent to $b_i - a_i \le b_i - a_i$. □

Theorem 8. *Let $\alpha, \beta, \gamma \in R^n$ such that $\mathfrak{s}(\alpha)\,|\,\mathfrak{s}(\beta)$ and $\frac{\mathfrak{s}(\gamma)}{\mathrm{in}(\overline{\gamma})} < \frac{\mathfrak{s}(\alpha)}{\mathrm{in}(\overline{\alpha})}, \frac{\mathfrak{s}(\beta)}{\mathrm{in}(\overline{\beta})}$. Let $x^p := \frac{\mathrm{in}(\overline{\alpha})\,\mathfrak{s}(\beta)}{\mathfrak{s}(\alpha)}$, $x^a := \mathrm{in}(\overline{\alpha})$ and $x^b := \mathrm{in}(\overline{\beta})$. Define v by $v_i := \infty$ for $b_i \le p_i$ and $v_i := \max(p_i, a_i)$ otherwise. Then $\mathfrak{s}(\mathcal{S}(\alpha,\gamma))\,|\,\mathfrak{s}(\mathcal{S}(\beta,\gamma))$ if and only if $\mathrm{in}(\overline{\gamma})|x^v$.*

Proof. The assumptions about sig-lead ratios imply that

$$\mathfrak{s}(\mathcal{S}(\alpha,\gamma)) = \frac{\mathfrak{s}(\gamma)}{\gcd(\mathrm{in}(\overline{\alpha}),\mathrm{in}(\overline{\gamma}))}\,\mathfrak{s}(\alpha)$$

and that

$$\mathfrak{s}(\mathcal{S}(\beta,\gamma)) = \frac{\mathfrak{s}(\gamma)}{\gcd(\mathrm{in}(\overline{\beta}),\mathrm{in}(\overline{\gamma}))}\,\mathfrak{s}(\beta).$$

Let $x^q := \frac{\mathfrak{s}(\beta)}{\mathfrak{s}(\alpha)}$, $x^c := \mathrm{in}(\overline{\gamma})$ and define r by $r_i = \infty$ for $b_i \le q_i + a_i$ and $r_i = q_i + a_i$ otherwise. Then what we need to prove for each i is that

$$\min(b_i, c_i) - \min(a_i, c_i) \le q_i$$

if and only if $c \le \max(a, r)$.

The case $\boldsymbol{c_i \le a_i}$: Both inequalities are always satisfied in this case since $A|B$ implies that $q_i \ge 0$.

The case $\boldsymbol{c_i > a_i}$: We need to prove that $\min(b_i, c_i) \le q_i + a_i$ if and only if $c_i \le r_i$, which follows quickly from considering the two cases $b_i > q_i$ and $b_i \le q_i$. □

B.3 Sig-Lead Ratios

Section 3.4 shows that sig-lead ratios occur in many places in the SB algorithm. We have wondered if there might be some mathematical significance to that, but we have not yet found one.

B.4 Stop on Detecting Gröbner Basis

SB sometimes computes a Gröbner basis much sooner than it computes a signature Gröbner basis. So if all we want is a Gröbner basis, then we want to stop early.

Call a basis element α *essential* if its lead term $\mathrm{in}(\overline{\alpha})$ is not divisible by the lead term of any other basis element. Eder, Gash and Perry had the idea to stop the F5 algorithm early when no S-pair remains that is between two essential basis elements [5]. This idea also works for SB if we additionally require that at least one S-pair $\mathcal{S}(\beta, \gamma)$ has been reduced for each non-essential basis element β where $\mathrm{in}(\overline{\beta}) \mid \mathrm{in}(\overline{\gamma})$. This is not hard to prove using the classic Buchberger S-pair elimination criteria. Unfortunately, this criterion for detecting a Gröbner basis is not useful for SB — usually very little computation can be skipped.

There are sophisticated approaches designed to ensure termination of F5 using classic Gröbner basis criteria [2, 5, 11]. We are only concerned with increasing speed by stopping early, and we have used a very simple approach. To get an "if and only if" criterion for detecting a Gröbner basis, we run a classic Buchberger algorithm on the basis. The classic computation never adds a polynomial to the basis; when it discovers a lead term that cannot be reduced by the current basis, it pauses until that lead term can be reduced. This can cause significant overhead, but it is guaranteed to stop the computation as soon as the basis is a Gröbner basis.

Using this technique, we see a 30x speed up on yang1 and a 17x slowdown on katsura11. This technique is only a benefit when the signature Gröbner basis is much larger than the minimal Gröbner basis, but if we already know that ahead of time, then probably we should not use SB in the first place. For hcyclic8, the very last basis element to be added to the signature basis is in fact an essential basis element. So for hcyclic and examples like it, there is not much to win from this idea even if there were a zero-overhead way of doing it.

C. DATA STRUCTURES

We give more background on the material in Section 4.

C.1 Ordering Terms During Reduction

We are quite surprised at our result that once you apply a hash table, then it matters little whether you use a geobucket, a heap or a tournament tree to order the terms. This suggests that there are many like terms when performing polynomial reduction in SB. If that hypothesis is correct it explains why hash tables are performing well in our experiments, since hash tables immediately sum all the like terms into a single term. If that leaves only a few terms that go into the priority queue, then that also explains why we are seeing that the choice of priority queue is less important to the running time. This is a topic that we want to investigate more closely in future work. An important topic that we have not yet addressed is what happens when using packed monomials. The packed monomial technique only applies to ideals with few enough variables and small enough degree of monomials in the basis, but many interesting ideals fall into that category.

Some readers may wonder why we bother investigating the classic setup for polynomial reduction when everyone knows that matrix-based reduction as in F4 [8] is much better. We have several reasons. Priority queues are used throughout the implementation for several different purposes, so it is important to investigate priority queues for Gröbner basis computation regardless. F4 is not a win for polynomial reduction of a single polynomial, which is an operation that algorithms outside of Gröbner basis computation sometimes have to do.

More importantly, from our conversations with researchers in the field, we know that implementing F4 to be more efficient than the classic approach is tricky and that several people have failed in their attempts. We have even been given the advice to write an F4 implementation such that the reduction steps taken are exactly the same as those that would be done by classic polynomial reduction, so that the benefit would accrue only from the replacement of monomials with column-index-integers. The F4 implementation in Macaulay 2 follows this advice, and as Table 1 shows, Macaulay 2 gets a very respectable time on hcyclic-8 even though F4 is the only special thing it does. This indicates to us that it is possible that better data structures for classic polynomial reduction might make it just as good as F4 is. We cannot determine as a field if that is true or not without looking for better data structures for classic polynomial reduction, and that includes considering better choices of priority queues. The requirement to store very large matrices in memory is also a disadvantage of F4.

We give more details on how we have implemented the priority queues. Most any text book on heaps will explain how to pack a heap into an array and the basic heap algorithms for inserting and removing elements. However, we have been unable to find a reference that collects the various techniques that go beyond the basics. The literature that mentions the word "heap" consists of more than 40,000 articles on Google Scholar, so it seems likely that one of them explains how to write a good implementation. Yet we have found no such article, so we collect the improvements to heaps that we know of here, since these techniques are necessary to be able to replicate our findings about heaps.

Start indices at 1

If the heap's root is placed at index 0 of the array, then the formulas for the left child $l(n)$, right child $r(n)$ and parent $p(n)$ of the node at index n are (division by 2 rounds down)

$$p(n) = \frac{n-1}{2}, \quad l(n) = 2n+1, \quad r(n) = 2n+2.$$

If the heap's root is placed at index 1, the formulas become instead

$$p(n) = \frac{n}{2}, \quad l(n) = 2n, \quad r(n) = 2n+1.$$

The latter formulas are more efficient, so place the root at index 1 instead of 0. This can be achieved by leaving the space at index 0 unused, or if using pointers it can be done by subtraxting 1 from the pointer to the array. Though be aware that subtracting 1 from a pointer to an array will calculate an invalid pointer. Even merely calculating an invalid pointer without dereferencing it invokes undefined behavior according to the C++ standard. However, actual systems seem to have no problem with it.

Make pop move element to bottom of heap before replacement with leaf

Pop on a heap is frequently described by replacing the top element by the right-most bottom leaf and then moving it down until the heap property is restored. This requires 2 comparisons per level that we go past, and we are likely to go far down the heap since we moved a leaf to the top of the heap. So we should expect a little less than $2\log(n)$ comparisons.

Instead, as is a common technique in implementations, we can leave a hole in the heap where the top element was. Then we move that hole down the heap by iteratively moving the larger child up. This requires only 1 comparison per level that we go past. In this way the hole will become a leaf. At this point we can move the right-most bottom leaf into the position of the hole and move that value up until the heap property is restored. Since the value we moved was a leaf we do not expect it to move very far up the tree. So we should expect a little more than $\log n$ comparisons, which is better than before.

This technique is widely used in heap implementations, where the heuristic argument above is borne out in practice through showing better heap performance.

Support replace-top

Suppose you want to remove the max element and also insert a new element. Then you can follow the pop algorithm described above, except that instead of moving the right-most bottom leaf into the vacant position, you use the new value you wish to push. This is more efficient than a pop followed by a push.

Make the elements have a size in bytes that is a power of 2

The parent, left-child and right-child formulas work on indices and they cannot be made to work directly on pointer values. So we are going to be working with indices and that implies looking up values in an array p from an index i. If p points to an array of Ts which each take up s bytes, then the element at offset i in the array has address $p + s * i$. Using C++ notation, we have that

```
&(p[i]) == static_cast<char*>(p) + sizeof(T) * i.
```

The computer has to perform this computation to get at the element with index i. s is a compile-time constant, and the multiplication can be done more efficiently if sizeof(T) is a power of two. We found an increase in performance by adding 8 padding bytes to increase s from 24 to 32. Reduced efficiency of the cache probably means that this is not a win for sufficiently large data sets.

Pre-multiply indices

The only thing a heap ever does with an index other than finding parent, left-child and right-child is to look the index up in an array. So if we keep track of $j := s * i$ instead of an index i, then we could do a lookup of the element at index i without the multiplication that would otherwise be necessary since the address of the element with index i is $p + s * i = p + j$. In C++ notation, we have that

```
&(p[i]) == static_cast<char*>(p) + sizeof(T) * i
        == static_cast<char*>(p) + j
```

Then the left-child and right-child formulas for j-values become respectively $2j$ and $2j + s$. The parent formula becomes $(j/(2s)) * s$ where the division rounds down. In C++ notation, we have that the parent of j is

```
(j / (2 * sizeof(T))) * sizeof(T).
```

If s is a power of 2 then this will compile to 2 bit-shifts. That is one operation more than the usual parent using indices i. However we then save one operation on lookup. So the net effect is that finding the parent takes the same amount of time this way, while lookup of left-child and right-child becomes faster.

Memory optimization

There are heap-like priority queues that are designed to make better use of the CPU cache. For example a 4-heap should have fewer cache misses and the amount of extra overhead is not that much. LaMarca and Ladner report in 1996 that they get a 75% performance improvement from going to aligned 4-heaps [15]. However, Hendriks reports in 2010 that [16]:

> The improvements to the implicit heap suggested by LaMarca and Ladner to improve data locality and reduce cache misses were also tested. We implemented a four-way heap, that indeed shows a slightly better consistency than the two-way heap for very skewed input data, but only for very large queue sizes. Very large queue sizes are better handled by the hierarchical heap.

Based on this we have chosen not to investigate alternative cache-friendly heap layouts further.

C.2 Tournament Trees

A tournament tree is a classic priority queue data structure that consists of a binary tree where each node is labeled by a value. Each interior node's value is the maximum of the values of its two children. Thus the top of the tree will have the maximum element of the tree. The data structure is called a tournament tree since such trees describe for example a tennis tournament where the roots describe the players and the internal nodes describe a match between two players. The player at the root of the tree won all his matches and thus wins the tournament.

Our tournament trees are complete binary trees so that we can pack them into an array just as is typically done for heaps. We use the same code to navigate the tree, so the comments for navigating heaps also apply to our tournament tree implementation. The pre-multiply indices improvement is not useful, though, since we only ever go up the tree, and that optimization does not speed up calculating the parent of a node.

Replacing a value a with another value b is especially fast in a tournament tree. Every value is annotated with the index of the leaf node where it comes from, so given any position in the tree, we can quickly jump to the leaf with the same value. Once the leaf for a is located, we change its value to b and follow the path from that leaf to the node while updating the values in the nodes along the way. This requires only one comparison per level of the tree even in the worst case.

C.3 Monomial Ideal Data Structures

We add detail to Section 4.2.

C.3.1 Kd-trees

A leaf in our kd-tree is split into two smaller leaves if it contains too many monomials. The index of the new internal node is $i+1$ if its parent has index i, starting over at the first variable if there is no variable $i+1$. We tried choosing the exponent k in x_{i+1}^k to be the median exponent of x_{i+1} among the monomials in the leaf being split, but we found that it works just as well to let k be the average of the maximum and minimum exponents of x_{i+1} among the monomials.

In our implementation we use a special encoding of the binary tree such that each node on a left-going path down the tree is packed into a single "super node". This technique was very complicated to implement and gave only a tiny improvement in speed (<5%), so we recommend the usual representation of a binary tree where an internal node has a pointer to each of its children.

The kd-tree can become unbalanced after many insertions and deletions, especially since we never remove leaves. To combat this, we completely rebuild the tree after a certain percentage of the tree has changed since the last rebuild.

C.3.2 Divmasks

Table 5 shows the hit rates for divmasks in our implementation of SB when using a monomial list. We say that a pair of monomials (a,b) is a *divmask hit* if the divmask proves that a cannot divide b. We say that (a,b) is a *divmask miss* if a does not divide b, but the divmask fails to prove that. We say that (a,b) is *divisible* if a divides b. A divmask can do nothing useful in case of divisibility. The *hit rate* for divmasks is the ratio of hits to the sum of hits and misses. The *effective hit rate* is the ratio of hits to all checks for divisibility involving a divmask.

For most of the examples we get a 99% hit rate and a 98% effective hit rate. This is the case even for mayr42 where there are 51 variables, while our divmasks are 32 bits long so the divmask is constructed based on the first 32 variables only. joswig101 gets a lower hit rate of 84%, but Table 3 still shows a large speed up from using divmasks. The lower hit rate is likely due to the example having only 5 variables.

For the divmap $d_{x_i^t}$ we chooose the exponent t to be the average of the minimum and maximum exponent of x_i among the monomials in the data structure. We also tried to use the median, but there was no advantage.

We use a 32 bit word for the divmasks. If there are less than 32 variables then each variable x_i gets more than one divmap, and we space the exponents t in the range between the minimum and maximum exponent of x_i. If there are more than 32 variables, then the variables past the first 32 are ignored when constructing a divmask.

Even if there are more than 32 variables and the program is being compiled for and run on a 64 bit CPU, it has been slower in our experiments to use a 64 bit divmask. They do give a higher hit rate on for example yang1, but the hit rate is already high for 32 bits, so the increased hit rate did not make up for the extra overhead of dealing with more bits. We tried 16 bit divmasks too and they were also worse than 32 bits.

C.4 The S-pair triangle

In Section 4.3 we mention a scheme to use 16 bits for entries in columns in an S-pair triangle when possible and then switching to 32 bits for columns past 2^{16}. This idea can be extended to use b bits from column 2^{b-1} to column 2^b-1. This scheme is complicated to implement and we calculated that the memory savings are tiny. We conclude that the split into 16 and 32 bits already extracts most of the possible benefit, so there is not much reason to go further.

On yang1 we spend a substantial amount of time on sorting the S-pairs in each column of the S-pair triangle. The sorting algorithm we use to sort each column is the `std::sort` function from the standard C++ library of GCC. It uses a variant of quicksort. We suspect that the S-pairs are being constructed in roughly increasing order of signature, since the basis is ordered by signature, so the array being sorted should be already in roughly sorted order — not exactly in sorted order, but closer to it than a random permutation. Sorting algorithms that run more quickly on roughly sorted data are called *adaptive*. Quicksort is still $\Theta(n\log n)$ even when running on already sorted input. There might be an advantage to be had from using an adaptive sorting algorithm to sort the columns of the S-pair triangle.

D. OUR CLASSIC BUCHBERGER ALGORITHM IMPLEMENTATION

Our classic Buchberger implementation uses the data structures that we propose in this paper. Other than that, it has an interesting S-pair elimination criterion based on an old unpublished theorem of Dave Bayer. We had originally described classic S-pair elimination criteria in the paper, but we had to cut it to make the paper fit within the page limit. What we wrote on the matter is now in this appendix.

D.1 The lcm and Relatively Prime Criteria

Many S-polynomials reduce to zero and it is better to predict that and eliminate the S-pair instead of performing the reduction. Recall that if all S-pairs have a representation (see Appendix A.2), then the current polynomial basis is a Gröbner basis. Buchberger already introduced two criteria for making that prediction. Let a, b and c be basis elements. If $\operatorname{in}(a)$ and $\operatorname{in}(b)$ are relatively prime then (a,b) can be eliminated. This is Buchberger's first criterion. Call it the *relatively prime criterion.* If $\operatorname{in}(c)|\operatorname{lcm}(\operatorname{in}(a),\operatorname{in}(b))$ and both $\mathcal{P}(a,c)$ and $\mathcal{P}(b,c)$ have a representation, then so does $\mathcal{P}(a,b)$. This is Buchberger's second criterion. Call it the *lcm criterion.*

Keeping track of S-pairs takes both time and space in addition to the time spent on reductions. Therefore it is good to eliminate S-pairs as early as possible in the computation. So if $\operatorname{in}(c)|\operatorname{lcm}(\operatorname{in}(a),\operatorname{in}(b))$ then it is tempting to eliminate (a,b) right away even if $\mathcal{P}(a,c)$ or $\mathcal{P}(b,c)$ have not been eliminated yet. This could be done using the argument that we will eventually process (a,c) and (b,c) so we will eventually eliminate (a,b) so we might as well do it right away. *Do not believe this argument.* The argument is incorrect because of the possibility that $\operatorname{in}(a)|\operatorname{lcm}(\operatorname{in}(c),\operatorname{in}(b))$. In that case we would eliminate (a,b) based on an assumption that we will process (a,c) later, and we would also eliminate (a,c) based on an assumption that we will process (a,b) later.

There is a revised approach which does work. Assume that $\operatorname{in}(c)|\operatorname{lcm}(\operatorname{in}(a),\operatorname{in}(b))$. Then we can eliminate (a,b) right away if

$$\operatorname{lcm}(a,c)\neq\operatorname{lcm}(a,b) \text{ or } (a,c) \text{ is eliminated,}$$

and

$$\operatorname{lcm}(b,c)\neq\operatorname{lcm}(a,b) \text{ or } (b,c) \text{ is eliminated.}$$

Here an S-pair is also considered to be eliminated if it has been reduced. In this case there is no possibility of a circular argument. There are many alternatives to this particular

	joswig101	jason210	katsura10	katsura11	hcyclic8	mayr42
# divmask hits	6,347,442,512	674,026,292	81,661,319	703,178,965	19,629,163,403	327,387,283,068
# divmask misses	1,256,265,766	10,583,467	773,448	4,198,228	204,086,138	2,988,880,796
# divisibilities	1,109,093,540	1,667,422	533,541	2,605,217	56,053,435	160,872,762
hit rate	83.5%	98.5%	99.1%	99.4%	99.0%	99.1%
effective hit rate	72.9%	98.2%	98.4%	99.0%	98.7%	99.0%

Table 5: Divmask hit rates

way of applying the lcm criterion. Suppose $\operatorname{lcm}(\operatorname{in}(a), \operatorname{in}(b)) = \operatorname{lcm}(\operatorname{in}(a), \operatorname{in}(c)) = \operatorname{lcm}(\operatorname{in}(b), \operatorname{in}(c))$. Then one and only one of the S-pairs among a, b, c can be eliminated. The approach outlined here in effect lets the ordering on the S-pairs decide - the S-pair ordered to be reduced last is the one to get eliminated. If the S-pair ordering is good then the last pair should also be the pair that we would most like to avoid reducing, so this is a good choice. Furthermore, this scheme is simple to think about and simple to implement correctly.

Using the lcm criterion in this way requires a way to quickly determine if a given S-pair (a, b) has been eliminated. We solved this problem by keeping a triangular array of $\binom{k}{2}$ bits (not bytes) which supports constant time access to a bit for each S-pair.

The monomial data structures from Section 4.2 can be used to find the divisors of $\operatorname{lcm}(\operatorname{in}(a), \operatorname{in}(b))$. However, there can be so many S-pairs that the lcm criterion becomes a bottleneck even with those data structures. To get around this, we cache the elements of B that often end up as the c such that $\operatorname{in}(c) \mid \operatorname{lcm}(\operatorname{in}(a), \operatorname{in}(b))$. If c is the element that eliminates (a, b) using the lcm criterion, then we associate c to both of a and b, and we forget any earlier such association to a and b. The next time we consider an S-pair involving a, we see that c was useful before and check c first before performing a full search of all divisors. Given a pair (a, b), there will be two cached generators to check — one for a and one for b. We were surprised by the effectiveness of this approach — see Appendix D.3.

D.2 The Graph Criterion

The lcm criterion has been improved on by Gebauer and Möller [12]. They propose a technique that quickly constructs a near-minimal set of S-pairs. It is well-known that in order to obtain a Gröbner basis, it suffices to reduce only those S-pairs which correspond to a minimal generating set of the syzygy module of the lead terms of the basis. Caboara, Kreuzer and Robbiano show that there is an advantage to be had by getting this minimal set instead of relying on heuristics [4]. They propose an algorithm to obtain the minimal set which in some cases leads to a speed up. There is also an overhead to the computation so that it does not always pay off.

In the 1980's, Dave Bayer came up with an unpublished alternative graph-based characterization of the minimal set of S-pairs that form a generating set. We use it to obtain the minimal set without much overhead. Given a polynomial basis B and a monomial m, define an undirected graph G_m with vertices

$$\{g \in B \,|\operatorname{in}(g) \text{ divides } m\}$$

such that (a, b) is an edge if $\operatorname{lcm}(\operatorname{in}(a), \operatorname{in}(b)) \neq m$ or if (a, b) has already been eliminated. We eliminate an S-pair (a, b) if a and b are connected in $G_{\operatorname{lcm}(a,b)}$. The set of S-pairs that are not eliminated then correspond to a minimal Gröbner basis of the set of syzygies on the initial terms of the basis. Call this criterion the *graph criterion*. Observe that the lcm criterion as described above is a special case of this more powerful criterion, but that the lcm criterion has less overhead. We use both criteria in our implementation.

If there is a cycle in G_m, then any edge (a, b) on the cycle with $\operatorname{lcm}(\operatorname{in}(a), \operatorname{in}(b)) = m$ can be eliminated. As before, we in effect let the ordering on the S-pairs choose which S-pair it would least like to do.

In practical terms we implement this criterion by computing the graph $G_{\operatorname{lcm}(a,b)}$ for each S-pair (a, b) just before $\mathcal{P}(a, b)$ would otherwise have been reduced. The nodes of the graph can be determined quickly using the monomial data structures from Section 4.2. The graphs are usually small and the edges are quick to construct, so the overhead is not much. Especially not since the early and late lcm criterion and the relatively prime criterion already eliminate most of the S-pairs. It should be possible to make significant optimizations by caching parts of the graph, but we have seen no need to improve this part of our implementation as it does not take much time.

D.3 Evaluating S-pair Elimination Criteria

Table 6 shows how many S-pairs are eliminated by each criterion. Every row in this table shows something interesting. The relatively prime criterion is very effective on yang1, eliminating 52% of the S-pairs. For yang1, the cache idea works so well that 98% of the S-pairs that the lcm criterion can eliminate are eliminated already from just looking at the cache. The graph criterion gets only an extra 4% eliminated S-pairs compared to not using it on yang1, and on 4by4 the number is down to 0.2%.

This business of counting the number of S-pairs that are eliminated is a good first approximation, but ultimately what counts is the time saved due to the eliminated S-pairs. In some cases a small number of zero reductions can account for a large amount of the running time of the algorithm, so it can be the case that the graph criterion can eliminate just a few S-pairs and still contribute a significant speed up.

Another point is that it is not very informative to compare the number of zero reductions between Table 6 and Table 2. The overhead from the SB algorithm is not just in reducing to zero, it is also in reducing to a basis element that is part of the signature Gröbner basis but not part of the minimal Gröbner basis. A better comparison would be in terms of total number of divisor queries, monomial multiplications, monomial comparisons and ground field operations. Even that is not perfect, but it is better than just looking at the number of reductions performed, let alone just looking at the number of reductions to zero. This is a kind of measure that we are looking into having our implementation be able to report.

E. REFERENCES

Example	4by4	jason210	mayr42	yang1
#S-pairs	108,811	404,550	36,410,311	11,331,180
rel prime	27,683	2	664,223	5,426,819
lcm cache hits	50,051	353,112	30,995,451	5,643,927
lcm simple hits	26,182	45,604	4,411,353	162,472
lcm graph hits	12	81	5,363	3,964
#reductions	4883	5751	333,921	93,998
0-reductions	4416	4851	325,387	89,237

Table 6: Classic Buchberger S-pair elimination

[1] A. Arri and J. Perry. The f5 criterion revised. *Journal of Symbolic Computation*, 46(9):1017 – 1029, 2011.

[2] G. Ars. *Applications des bases de Gröbner á la cryptographie.* PhD thesis, Universitè de Rennes I, 2005.

[3] M. A. Bender, R. Cole, E. D. Demaine, M. Farach-Colton, and J. Zito. Two simplified algorithms for maintaining order in a list. In *Proceedings of ESA*, pages 152–164, 2002.

[4] M. Caboara, M. Kreuzer, and L. Robbiano. Efficiently computing minimal sets of critical pairs. *Journal of Symbolic Computation*, 38(4):1169 – 1190, 2004.

[5] C. Eder, J. Gash, and J. Perry. Modifying faugére's f5 algorithm to ensure termination. *ACM Commun. Comput. Algebra*, 45(1/2):70–89, July 2011.

[6] C. Eder and J. E. Perry. Signature-based algorithms to compute Gröbner bases. In *Proceedings of the 36th international symposium on Symbolic and algebraic computation*, ISSAC '11, pages 99–106, New York, NY, USA, 2011. ACM.

[7] R. Fateman. Comparing the speed of programs for sparse polynomial multiplication. *SIGSAM Bull.*, 37:4–15, March 2003.

[8] J.-C. Faugère. A new efficient algorithm for computing gröbner bases (f_4). *Journal of Pure and Applied Algebra*, 139(1):61–88, June 1999.

[9] J.-C. Faugère. A new efficient algorithm for computing gröbner bases without reduction to zero (f_5). In *Proceedings of ISSAC*, July 2002.

[10] S. Gao, F. V. IV, and M. Wang. A new algorithm for computing groebner bases. Cryptology ePrint Archive, Report 2010/641, 2010. `http://eprint.iacr.org/`.

[11] J. M. Gash. *On efficient computation of Grobner bases.* PhD thesis, University of Indiana, 2008.

[12] R. Gebauer and H. M. Möller. On an installation of buchberger's algorithm. *J. Symb. Comput.*, 6:275–286, December 1988.

[13] L. Huang. A new conception for computing gröbner basis and its applications. arXiv:1012.5425.

[14] S. C. Johnson. Sparse polynomial arithmetic. *SIGSAM Bull.*, 8:63–71, August 1974.

[15] A. LaMarca and R. Ladner. The influence of caches on the performance of heaps. *J. Exp. Algorithmics*, 1, January 1996.

[16] C. L. Luengo Hendriks. Revisiting priority queues for image analysis. *Pattern Recogn.*, 43:3003–3012, September 2010.

[17] R. A. Milowski. Computing irredundant irreducible decompositions of large scale monomial ideals. In *Proceedings of the 2004 international symposium on Symbolic and algebraic computation*, ISSAC '04, pages 235–242, New York, NY, USA, 2004. ACM.

[18] M. Monagan and R. Pearce. Polynomial division using dynamic arrays, heaps, and packed exponent vectors. In *LNCS 4770*, pages 295–315. Springer Verlag, 2007.

[19] M. Stillman and B. H. Roune. Online version of this paper with appendices and data files. `www.broune.com/papers/issac2012.html`.

[20] T. Yan. The geobucket data structure for polynomials. *J. Symb. Comput.*, 25:285–293, March 1998.