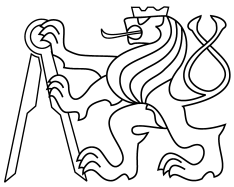
CENTER FOR
MACHINE PERCEPTION



CZECH TECHNICAL
UNIVERSITY IN PRAGUE

BACHELOR THESIS

Minimal Problem Solver Generator

Pavel Trutman

pavel.trutman@fel.cvut.cz

May 3, 2015

Available at
http://cmp.felk.cvut.cz/~trutmpav/theses/bsc-pavel-trutman.pdf

Thesis Advisor: Ing. Tomáš Pajdla, PhD.

Acknowledge grants here. Use centering if the text is too short.

Center for Machine Perception, Department of Cybernetics
Faculty of Electrical Engineering, Czech Technical University
Technická 2, 166 27 Prague 6, Czech Republic
fax +420 2 2435 7385, phone +420 2 2435 7637, www: http://cmp.felk.cvut.cz

Minimal Problem Solver Generator

Pavel Trutman

May 3, 2015

Text of acknowledgements...

Abstract

Text of abstract...

Resumé

Text of resumé...

Contents

List of Algorithms

List of Abbreviations

C(f), C(F)	Set of all coefficients of the polynomial f or of all polynomials from the set F.
deg(f)	The total degree of f.
$\overline{f}^F$	Remainder of the polynomial f on division by F.
gcd	Greatest common multiple.
lcm	Least common divisor.
LC(f), LC(F)	Leading coefficient(s) of the polynomial f or of all polynomials from the set F.
LM(f), LM(F)	Leading monomial(s) of the polynomial f or of all polynomials from the set F.
LT(f), LT(F)	Leading term(s) of the polynomial f or of all polynomials from the set F.
M(f), M(F)	Set of all monomials of the polynomial f or of all polynomials from the set F.
$S(f_1, f_2)$	S-polynomial of polynomials f_1 and f_2.
T(f), T(F)	Set of all terms of the polynomial f or of all polynomials from the set F.
$x \mid y$	x divides y.
$\lfloor x \rfloor$	$= \max\{m \in \mathbb{Z} \mid m \leq x\}$; floor function.
$\mathbb{Z}_p$	Prime field of characteristic p.

1 Introduction

Here comes introduction.

2 Polynomial system solving

Firstly we review the state of the art algorithms for computing Gröbner basis. Better understanding of these algorithms helps us to integrate them into polynomial solving algorithms based on Gröbner basis computation more efficiently.

2.1 Buchberger Algorithm

Buchberger Algorithm [2], which was invented by Bruno Buchberger, was the first algorithm for computing Gröbner basis. The algorithm is described in details in [1, 4].

2.1.1 First implementation

The first and easy, but very inefficient implementation of the Buchberger Algorithm, Algorithm 1, is based on the observation that we can extend a set F of polynomials to a Gröbner basis only by adding all non-zero remainders $\overline{S(f_i, f_j)}^F$ of all pairs from F into F until there is no non-zero remainder generated.

The main disadvantage of this simple algorithm is that so constructed Gröbner basis are often bigger than necessary. This implementation of the algorithm is also very inefficient because many of the S-polynomials that are constructed from the critical pairs are reduced to zero so after spending effort on computing them, there is nothing to add to the Gröbner basis G. How to decide which pairs need not be generated is described next.

Algorithm 1 Simple Buchberger Algorithm

Input:
F a finite set of polynomials
Output:
G a finite set of polynomials

1: $G \leftarrow F$
2: $B \leftarrow \{\{g_1, g_2\} \mid g_1, g_2 \in G,\ g_1 \neq g_2\}$
3: **while** $B \neq \emptyset$ **do**
4: select $\{g_1, g_2\}$ from B
5: $B \leftarrow B \setminus \{\{g_1, g_2\}\}$
6: $h \leftarrow S(g_1, g_2)$
7: $h_0 \leftarrow \overline{h}^G$
8: **if** $h_0 \neq 0$ **then**
9: $B \leftarrow B \cup \{\{g, h_0\} \mid g \in G\}$
10: $G \leftarrow G \cup \{h_0\}$
11: **end if**
12: **end while**
13: **return** G

2.1.2 Improved Buchberger Algorithm

The combinatorial complexity of the simple implementation of the Buchberger Algorithm can be reduced by testing out certain S-polynomials which need not be considered. To know which pairs can be deleted without treatment, we use the first and the second Buchberger's criterion [1]. Sometimes, we can even delete certain polynomials from the set G completely, knowing that every critical pair they will generate will reduce to zero and hence these polynomials themselves will be superfluous in the output set. In the next few paragraphs we will describe the implementation of the Improved Buchberger Algorithm and of the function *Update*, which deletes superfluous polynomials from G, according to Gebauer and Möller [8].

The Improved Buchberger Algorithm, Algorithm 2, has the same structure as the Simple Algorithm. The function *Update* is used at the beginning of the Improved Buchberger Algorithm to initialize the set B of critical pairs and the Gröbner basis G from the input set F of polynomials and at every moment when a new non-zero polynomial $h_0 = \overline{h}^G$ of an S-polynomial h has been found and the sets B and G are about to be updated.

Algorithm 2 Improved Buchberger Algorithm

Input:
F a finite set of polynomials
Output:
G a finite set of polynomials

1: $G \leftarrow \emptyset$
2: $B \leftarrow \emptyset$
3: **while** $F \neq \emptyset$ **do**
4: select f from F
5: $F \leftarrow F \backslash \{f\}$
6: $(G, B) \leftarrow Update(G, B, f)$
7: **end while**
8: **while** $B \neq \emptyset$ **do**
9: select $\{g_1, g_2\}$ from B
10: $B \leftarrow B \backslash \{\{g_1, g_2\}\}$
11: $h \leftarrow S(g_1, g_2)$
12: $h_0 \leftarrow \overline{h}^G$
13: **if** $h_0 \neq 0$ **then**
14: $(G, B) \leftarrow Update(G, B, h_0)$
15: **end if**
16: **end while**
17: **return** G

Now, let us look at the function *Update*, Algorithm 3. First, it makes pairs from the new polynomial h and all polynomials from the set G_{old} and puts them into the set C. The first while loop (lines 3 – 9) iterates over all pairs in the set C. In each iteration it select a pair $\{h, g_1\}$ from the set C and removes it from the set. Then it looks for another pair $\{h, g_2\}$ from the set C or the set D. If does not exists a pair $\{h, g_2\}$ such that (h, g_2, g_1) is a Buchberger triple, then the pair $\{h, g_1\}$ is put into the set D. The triple (h, g_2, g_1) of polynomials h, g_1 and g_2 is a Buchberger triple if the equivalent

conditions

$$\mathrm{LM}(g_2) \mid \mathrm{lcm}(\mathrm{LM}(h), \mathrm{LM}(g_1)) \tag{2.1}$$
$$\mathrm{lcm}(\mathrm{LM}(h), \mathrm{LM}(g_2)) \mid \mathrm{lcm}(\mathrm{LM}(h), \mathrm{LM}(g_1)) \tag{2.2}$$
$$\mathrm{lcm}(\mathrm{LM}(g_2), \mathrm{LM}(g_1)) \mid \mathrm{lcm}(\mathrm{LM}(h), \mathrm{LM}(g_1)) \tag{2.3}$$

are satisfied. From the second Buchberger's criterion, we know that if a Buchberger triple (h, g_2, g_1) shows up in the Buchberger Algorithm and the pairs $\{g_1, g_2\}$ and $\{h, g_2\}$ are amongs the critical pairs, then the pair $\{h, g_1\}$ need not be generated. That means in the code that such a pair is not moved from the set C to the set D but it is only removed from the set C. This while loop keeps all pairs $\{h, g_1\}$ where $\mathrm{LM}(h)$ and $\mathrm{LM}(g_1)$ are disjoint, i.e. $\mathrm{LM}(h)$ and $\mathrm{LM}(g_1)$ have no variable in common. The reason of this is that if two or more pairs in C have the same lcm of their leading monomials, then there is a choice which one should be deleted. So we keep the pair where the leading monomials are disjoint. Pairs with disjoint leading monomials are removed in the second while loop, so we eventually remove them all.

The second while loop (lines 11 – 17) eliminates all pairs with disjoint leading monomials. We can remove such pairs thanks to the first Buchberger's criterion. All remaining pairs are stored in the set E.

The third while loop (lines 19 – 25) eliminates pairs $\{g_1, g_2\}$ where (g_1, h, g_2) is a Buchberger triple from the set B_{old}. Then the updated set of the old pairs and the new pairs are united into the set B_{new}.

Finally, the last while loop (lines 28 – 34) removes all polynomials g whose leading monomial is a multiple of the leading monomial of h from the set G_{old}. We can eliminate such polynomials for two reasons. Firstly, $\mathrm{LM}(h) \mid \mathrm{LM}(g)$ implies $\mathrm{LM}(h) \mid \mathrm{lcm}(\mathrm{LM}(g), \mathrm{LM}(f))$ for arbitrary polynomial f. We can see that (g, h, f) is a Buchberger triple for any f which in future appears in the set G. Moreover, polynomial g will not be missed at the end, because in the Gröbner basis G, polynomials with leading monomials which are multiples of leading monomials of another polynomial from G are superfluous, i.e. they will be eliminated in the reduced Gröbner basis.

In the end of the function, the polynomial h is added into the Gröbner basis G_{new}. The output of the function *Update* is the Gröbner basis G_{new} and the set B_{new} of critical pairs.

2.2 F_4 Algorithm

The F_4 Algorithm [6] by Jean-Charles Faugère is an improved version of the Buchberger's Algorithm. The F_4 replaces the classical polynomial reduction found in the Buchberger's Algorithm by a simultaneous reduction of several polynomials. This reduction mechanism is achieved by a symbolic precomputation followed by Gaussian elimination implemented using sparse linear algebra methods. F_4 speeds up the reduction step by exchanging multiple polynomial divisions for row-reduction of a single matrix.

2.2.1 Improved Algorithm F_4

The main function of the F_4 Algorithm, Algorithm 4, consists of two parts. The goal of the first part is to initialize the whole algorithm.

Algorithm 3 Update

Input:
G_{old} a finite set of polynomials
B_{old} a finite set of pairs of polynomials
h a polynomial such that $h \neq 0$

Output:
G_{new} a finite set of polynomials
B_{new} a finite set of pairs of polynomials

1: $C \leftarrow \{\{h, g\} \mid g \in G_{old}\}$
2: $D \leftarrow \emptyset$
3: **while** $C \neq \emptyset$ **do**
4: select $\{h, g_1\}$ from C
5: $C \leftarrow C \backslash \{\{h, g_1\}\}$
6: **if** LM(h) and LM(g_1) are disjoint **or**
(lcm(LM(h), LM(g_2)) $\nmid$ lcm(LM(h), LM(g_1)) for all $\{h, g_2\} \in C$ **and**
lcm(LM(h), LM(g_2)) $\nmid$ lcm(LM(h), LM(g_1)) for all $\{h, g_2\} \in D$) **then**
7: $D \leftarrow D \cup \{\{h, g_1\}\}$
8: **end if**
9: **end while**
10: $E \leftarrow \emptyset$
11: **while** $D \neq \emptyset$ **do**
12: select $\{h, g\}$ from D
13: $D \leftarrow D \backslash \{\{h, g\}\}$
14: **if** LM(h) and LM(g) are not disjoint **then**
15: $E \leftarrow E \cup \{\{h, g\}\}$
16: **end if**
17: **end while**
18: $B_{new} \leftarrow \emptyset$
19: **while** $B_{old} \neq \emptyset$ **do**
20: select $\{g_1, g_2\}$ from B_{old}
21: $B_{old} \leftarrow B_{old} \backslash \{\{g_1, g_2\}\}$
22: **if** LM(h) $\nmid$ lcm(LM(g_1), LM(g_2)) **or**
lcm(LM(g_1), LM(h)) = lcm(LM(g_1), LM(g_2)) **or**
lcm(LM(h), LM(g_2)) = lcm(LM(g_1), LM(g_2)) **then**
23: $B_{new} \leftarrow B_{new} \cup \{\{g_1, g_2\}\}$
24: **end if**
25: **end while**
26: $B_{new} \leftarrow B_{new} \cup E$
27: $G_{new} \leftarrow \emptyset$
28: **while** $G_{old} \neq \emptyset$ **do**
29: select g from G_{old}
30: $G_{old} \leftarrow G_{old} \backslash \{g\}$
31: **if** LM(h) $\nmid$ LM(g) **then**
32: $G_{new} \leftarrow G_{new} \cup \{g\}$
33: **end if**
34: **end while**
35: $G_{new} \leftarrow G_{new} \cup \{h\}$
36: **return** (G_{new}, B_{new})

First, it generates the set P of critical pairs and initializes the Gröbner basis G. This is done by taking each polynomial from the input set F and calling the function *Update* on it, which updates the set P of pairs and the set G of basic polynomials.

The second part of the algorithm generates new polynomials and adds them into the set G. In each iteration, it selects some pairs from P using the function *Sel*. Many selection strategies are possible and is still an open question how to best select the pairs. Some selection strategies are described in the section 2.2.6 on page 11. Then, it splits each selected pair $\{f_1, f_2\}$ into two tuples. The first tuple contains the first polynomial f_1 of the pair and the monomial m_1 such that $\mathrm{LM}(m_1 \times f_1) = \mathrm{lcm}(\mathrm{LM}(f_1), \mathrm{LM}(f_2))$. The second tuple is constructed in the same way from the second polynomial f_2 of the pair. All tuples from all selected pairs are put into the set L, i.e. duplicates are removed.

Next, function *Reduction* is called on the set L. It stores result in the set $\tilde{F}^+$. In the end of the algorithm it iterates through all new polynomials in the set $\tilde{F}^+$ and calls the function *Update* on each of them. This generates new pairs into the set P of critical pairs and extends the Gröbner basis G.

This algorithm terminates when the set P of pairs is empty. Then the set G is a Gröbner basis and it is the output of the algorithm.

Algorithm 4 Improved Algorithm F_4

Input:
F a finite set of polynomials
Sel a function $List(Pairs) \to List(Pairs)$ such that $Sel(l) \neq \emptyset$ if $l \neq \emptyset$

Output:
G a finite set of polynomials

1: $G \leftarrow \emptyset$
2: $P \leftarrow \emptyset$
3: $d \leftarrow 0$
4: **while** $F \neq \emptyset$ **do**
5: select f form F
6: $F \leftarrow F \backslash \{f\}$
7: $(G, P) \leftarrow Update(G, P, f)$
8: **end while**
9: **while** $P \neq \emptyset$ **do**
10: $d \leftarrow d + 1$
11: $P_d \leftarrow Sel(P)$
12: $P \leftarrow P \backslash P_d$
13: $L_d \leftarrow Left(P_d) \cup Right(P_d)$
14: $(\tilde{F}_d^+, F_d) \leftarrow Reduction(L_d, G, (F_i)_{i=1,\ldots,(d-1)})$
15: **for** $h \in \tilde{F}_d^+$ **do**
16: $(G, P) \leftarrow Update(G, P, h)$
17: **end for**
18: **end while**
19: **return** G

2.2.2 Function Update

In the F_4 Algorithm the standard implementation of the Buchberger's Criteria such as the Gebauer and Möller installation [8] is used. Details about the function *Update* can be found in the section 2.1.2. The pseudocode of the function is shown in Algorithm 3.

2.2.3 Function Reduction

Function *Reduction*, Algorithm 5, performs polynomial division using methods of linear algebra.

Input of the function *Reduction* is a set L containing tuples of monomial and polynomial. These tuples were constructed in the main function of the F_4 Algorithm from all selected pairs.

First, the function *Reduction* calls the function *Symbolic Preprocessing* on the set L. This returns a set F of polynomials to be reduced. To use linear algebra methods to perform polynomial division, the polynomials have to be represented by a matrix. Each column of the matrix corresponds to a monomial. Columns have to be ordered with respect to the monomial ordering used so that the most right column corresponds to "1". Each row of the matrix corresponds to a polynomial from the set F. The matrix is constructed as follows. On the (i, j) position in the matrix, we put the coefficient of the term corresponding to j-th monomial from the i-th polynomial from the set F.

We next reduce the matrix to a row echelon form using, for example, Gauss-Jordan elimination. Note that this matrix is typically sparse so we can use sparse linear algebra methods to save computation time and memory. After elimination, we construct resulting polynomials by multiplying the reduced matrix by a vector of monomials from the right.

In the end, the function returns the set $\tilde{F}^+$ of reduced polynomials such that their leading monomials are not leading monomials of any polynomial from the set F of polynomials before reduction.

Algorithm 5 Reduction

Input:
L a finite set of tuples of monomial and polynomial
G a finite set of polynomials
$\mathcal{F} = (F_i)_{i=1,\ldots,(d-1)}$, where F_i is finite set of polynomials

Output:
$\tilde{F}^+$ a finite set of polynomials
F a finite set of polynomials

1: $F \leftarrow$ *Symbolic Preprocessing*$(L, G, \mathcal{F})$
2: $\tilde{F} \leftarrow$ Reduction to a Row Echelon Form of F
3: $\tilde{F}^+ \leftarrow \left\{ f \in \tilde{F} \mid \mathrm{LM}(f) \notin \mathrm{LM}(F) \right\}$
4: **return** $(\tilde{F}^+, F)$

2.2.4 Function Symbolic Preprocessing

Function *Symbolic Preprocessing*, Algorithm 6, starts with a set L of tuples each containing a monomial and a polynomial. These tuples were constructed in the main

function of the F_4 Algorithm from the selected pairs. Then, the tuples are simplified by the function *Simplify* and after multiplying polynomials with corresponding monomials, the results are put into the set F.

Next, the function goes through all monomials in the set F and for each monomial m looks for some polynomial f from the Gröbner basis G such $m = m' \times \mathrm{LM}(f)$ where m' is some monomial. All such polynomials f and monomials m' are after simplification multiplied and put into the set F. The goal of this search is to have for every monomial in F a polynomial in F with the same leading monomial. This will ensure that all polynomials from F will be reduced for G after polynomial division by linear algebra.

Algorithm 6 Symbolic Preprocessing

Input:

L a finite set of tuples of monomial and polynomial
G a finite set of polynomials
$\mathcal{F} = (F_i)_{i=1,\ldots,(d-1)}$, where F_i is finite set of polynomials

Output:

F a finite set of polynomials

1: $F \leftarrow \{multiply(Simplify(m, f, \mathcal{F})) \mid (m, f) \in L\}$
2: $Done \leftarrow \mathrm{LM}(F)$
3: **while** $\mathrm{M}(F) \neq Done$ **do**
4: m an element of $\mathrm{M}(F) \backslash Done$
5: $Done \leftarrow Done \cup \{m\}$
6: **if** m is top reducible modulo G **then**
7: $m = m' \times \mathrm{LM}(f)$ for some $f \in G$ and some monomial m'
8: $F \leftarrow F \cup \{multiply(Simplify(m', f, \mathcal{F}))\}$
9: **end if**
10: **end while**
11: **return** F

2.2.5 Function Simplify

The function *Simplify*, Algorithm 7, simplifies a polynomial $m \times f$ which is a product of a given monomial m and a polynomial f.

The function recursively looks for a monomial m' and a polynomial f' such that $\mathrm{LM}(m' \times f') = \mathrm{LM}(m \times f)$. The polynomial f' is selected from all polynomials that have been reduced in previous iterations (sets $\tilde{F}$). We select polynomial f' such that the total degree of m' is minimal.

This is done in the function *Symbolic Preprocessing* to insert polynomials that are mostly reduced and have a small number of monomials into the set F of polynomials to be reduced. This of course speeds up following reduction.

2.2.6 Selection strategy

For the speed of the F_4 Algorithm, it is very important how the critical pairs from the list of all critical pairs P are selected in each iteration. This of course depends on the implementation of the function *Sel*. There are more possible selection strategies:

- The easiest implementation is to select all pairs from P. In this case we reduce all critical pairs at the same time.

Algorithm 7 Simplify

Input:
m a monomial
f a polynomial
$\mathcal{F} = (F_i)_{i=1,\ldots,(d-1)}$, where F_i is finite set of polynomials
Output:
(m', f') a non evaluated product of a monomial and a polynomial

```
1: for u ∈ list of all divisors of m do
2:     if ∃j (1 ≤ j ≤ d) such that (u × f) ∈ F_j then
3:         F̃_j is the Row Echelon Form of F_j
4:         there exists a (unique) p ∈ F̃_j such that LM(p) = LM(u × f)
5:         if u ≠ m then
6:             return Simplify(m/u, p, F)
7:         else
8:             return (1, p)
9:         end if
10:    end if
11: end for
12: return (m, f)
```

- If the function *Sel* selects only one critical pair then the F_4 Algorithm is the Buchberger Algorithm. In this case the *Sel* function corresponds to the selection strategy in the Buchberger Algorithm.
- The best function that Faugère has tested is to select all critical pairs with a minimal total degree. Faugère calls this strategy the *normal strategy for* F_4. Pseudocode of this function can be found as Algorithm 8.

Algorithm 8 Sel – The normal strategy for F_4

Input:
P a list of critical pairs
Output:
P_d a list of critical pairs

```
1: d ← min {deg(lcm(p)) | p ∈ P}
2: P_d ← {p ∈ P | deg(lcm(p)) = d}
3: return P_d
```

2.3 F_5 Algorithm

Since in the Buchberger Algorithm or in the F_4 Algorithm we spend much computation time to compute S-polynomials which will reduce to zero, the F_5 Algorithm [7] by Jean-Charles Faugère was proposed. The F_5 Algorithm saves computation time by removing useless critical pairs which will reduce to zero. The syzygies are used to recognize useless critical pairs in advance. For more details about syzygies look into [4].

There are several approaches how to use syzygies to remove useless pairs. For example

the idea of [13] is to compute a basis of the module of syzygies together with the computing of the Gröbner basis of the given polynomial system. Then a critical pair can be removed if the corresponding syzygy is a linear combination of the the elements of the basis of syzygies.

The strategy of the F_5 Algorithm is to consider only principal syzygies without computing the basis of the syzygies. The principal syzygy is a syzygy such that $f_i f_j - f_j f_i = 0$ where f_i and f_j are polynomials. This restriction implies that not all useless critical pairs have to be removed so a reduction to zero can appear. However it was proved that if the input system is a regular sequence then there is no reduction to zero.

To show how to distinguish which pairs need not be considered we use the example taken from [7]. Consider polynomials f_1, f_2 and f_3. Then the principal syzygies $f_i f_j - f_j f_i = 0$ can be written as follows:

$$u(f_2 f_1 - f_1 f_2) + v(f_3 f_1 - f_1 f_3) + w(f_2 f_3 - f_3 f_2) \quad = \quad 0 \tag{2.4}$$

where u, v and w are arbitrary polynomials. This can be also rewritten as

$$(uf_2 + vf_3)f_1 - uf_1 f_2 - vf_1 f_3 + wf_2 f_3 - wf_3 f_2 \quad = \quad 0. \tag{2.5}$$

We can see that all relations hf_1 are such that h is in the ideal generated by polynomials f_2 and f_3. So if we have computed Gröbner basis of the polynomials f_2 and f_3 it is easy to decide which new generated polynomials can be removed. We can remove all polynomials in the form $t \times f_1$ such that t is a term divisible by leading monomial of an element of the ideal generated by f_2 and f_3. Therefore the F_5 Algorithm is an incremental algorithm so if we have polynomials $f_1, \ldots, f_m$ on the input we have to compute all Gröbner basis of the following ideals: $(f_m), (f_{m-1}, f_m), \ldots, (f_1, \ldots, f_m)$ in this order.

Many reviews, implementations and modifications of the F_5 Algorithm has been made. Let us emphasize some of them. The first implementation of the F_5 was made by Jean-Charles Faugère himself in the language C. Then there is an implementation in Magma by A. J. M. Segers [16]. Another review and implementation in Magma was done by Till Stegers [17]. Since there is no proof of termination of the F_5 Algorithm see the modification [5] which terminates in any case.

3 Automatic generator

The automatic generator of Gröbner basis solvers is used to easily solve problems leading to systems of polynomial equations. These systems usually arise when solving minimal problems [14] in computer vision. Typically, these systems are not trivial so special solvers have to be designed for concrete problems to achieve efficient and numerically stable solvers. But solvers generated for concrete problems can not be easily applied for similar or new problems and therefore the automatic generator was proposed in [12]. Solvers generated by the automatic generator can be easily used to solve complex problems even by non-experts users.

The input of the automatic generator is a system of polynomial equations with a finite number of solutions and the output is a MATLAB or a Maple code that computes solutions of the given system for arbitary coefficients. One of the goals of this thesis is to improve previous implementation [12] of the automatic generator to construct more efficient and numerically stable solvers.

The newest version of the automatic genenerator implemented in MATLAB can be downloaded from [15].

3.1 Description of the automatic generator

In this section we would like to briefly describe the procedure for generating solvers. The procedure is based on computation of the action matrix from which solutions can be obtained. The automatic generator consists of several independent modules, see Figure 3.1. Since all these modules are independent, they can be easily improved or replaced by more efficient implementations. Next we describe each of these modules, full description can be found in [12, 10].

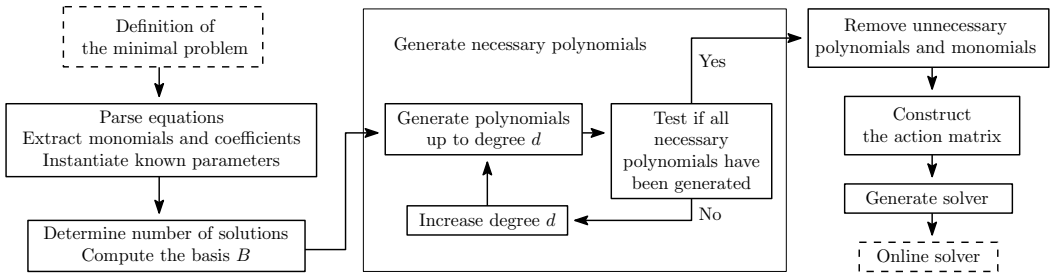


Figure 3.1 Block diagram of the automatic generator

3.1.1 Definition of the minimal problem

Definitions of minimal problems are written in separate functions that are stored in the folder `minimalProblems`. Each of the definitions has to contain few necessary information about the minimal problem. First of all, the system of polynomial equations with symbolic variables and parameters has to be provided. Next we have to specify the list of unknown variables and known parameters. Optionally if we know the monomial basis B of the polynomial system in advance we can specify it to save some computation

time. The monomial basis B is a set $\left\{m \mid \overline{m}^G = m\right\}$ where m is a monomial and G is the Gröbner basis of the given polynomial system. At last we have to set some settings for the automatic generator. We recommend to obtain the default settings by calling the function *gbs_InitConfig()* and only overwrite the settings we want to change. In the folder `minimalProblem` there are some examples which are self explanatory and can be used as templates to create new minimal problem definitions.

3.1.2 Equations parser, Instantiating

In the next step we have to parse the given equations, that means we extract used monomials and parameters and obtain total degrees of the polynomials. Then we instantiate each known parameter with a random number from $\mathbb{Z}_p$. We assign unique identifier to each used parameter. The reason is that we need to track the parameters through the process of adding polynomials in order to be able to restore the process in the solver generation module.

3.1.3 Monomial basis B computation

We need to know the monomials basis B to recognize when we have generated all polynomials that are necessary to build the action matrix. If the basis B was not provided within the definition of the minimal problem we have to compute it by ourselves. Because in MATLAB there is no function or simple script to compute the basis we have to do it by calling an external software.

The most easy solution to implement was to use the Maple toolbox for MATLAB. This enables us to call Maple functions from the MATLAB environment directly. To use this option we have to set `cfg.GBSolver = @gbs_findAlgB_maple` in the settings of the automatic generator. Unfortunately it shows up that the Maple toolbox for MATLAB in not compatible with the MATLAB Symbolic Math Toolbox in versions newer than R2008 so we do not recommend to use this option nowadays, but the option is still available to use on older computers.

The second implemented option is to use the algebraic geometry software Macaulay2 [9]. In the folder `gbsMacaulay` there is a template `code_template.m2` into which we simply write the given polynomial system. This updated file is saved as `code.m2` which is executed by Macaulay2 and the results are parsed back in MATLAB. To set up this option we need to install the software Macaulay2 and set `cfg.GBSolver = @gbs_findAlgB_macaulay` in the automatic generator settings. A problem could be that the Macaulay2 is not easy to set up under the Windows OS. Therefore the installation file of Macaulay2 is provided within the automatic generator. The only thing that has to be done is to edit the file `calc.bat` in the folder `gbsMacaulay` and follow the instructions in the file.

Because of the modularity of the generator this part can be replaced by another function computing the monomial basis B.

The last option is to compute the basis B in advance and set it into the definition of the minimal problem.

In the end we have check the number of solutions of the given polynomial system. If there is a finite number of solutions we can continue with the computation.

3.1.4 Polynomial generator

To be able to build the action matrix we have to generate enough polynomials such that after their reduction we get polynomials q_i which have leading monomials from

the set $(x_k \cdot B) \setminus B$ where x_k is a variable and all remaining monomials of q_i are from the set B. That is the reason we had to compute the basis B in the previous step.

In this part of the automatic generator we represent polynomials as row vectors so systems of polynomials can be represented by matrices. This representation enables us to easily multiplicate polynomials with monomials only by shifting the coefficients in the vectors or to reduce the whole polynomial systems by performing the Gauss-Jordan eliminations on the corresponding matrices.

Let $f_i \in F$ where F is a set of polynomials from the input. Let $maxdeg$ is a maximal total degree of all polynomials f_i. At the beginning we put into the matrix M all polynomials $\{m \times f_i \mid f_i \in F;\ \deg(m \times f_i) = \deg(f_i), \ldots, maxdeg\}$, where m is a monomial. Now we perform the Gauss-Jordan elimination on the matrix M and the result save as matrix $\tilde{M}$. Then we check if there exists a variable x_k for which all required polynomials q_i are present in $\tilde{M}$. If we find such a variable we can continue with the construction of the action matrix for the found variable. If not we have to add more polynomials to the matrix M. We increment $maxdeg$ by one and add all polynomials $\{m \times f_i \mid f_i \in F;\ \deg(m \times f_i) = maxdeg\}$ to the matrix M. Then we continue with the elimination and with the checking the action matrix requirements as written above. We repeat these steps until all required polynomials q_i are generated so the action matrix can be built. The pseudocode of this process is shown in Algorithm 9. The function *Check Action Matrix Conditions* from this code checks if all polynomials q_i are generated in $\tilde{M}$ for at least one variable from the given list of variables. If such a variable is found the function returns it otherwise it returns an empty set.

Algorithm 9 Polynomial generator – One elimination solver

Input:
F a set of polynomials
$variables$ a list of variables
$algB$ a monomial basis B

Output:
$\tilde{M}$ a matrix representing a set of polynomials
var a variable

1: $maxdeg \leftarrow \max\{\deg(f_i) \mid f_i \in F\}$
2: $M \leftarrow \{m \times f_i \mid f_i \in F;\ \deg(m \times f_i) = \deg(f_i), \ldots, maxdeg;\ m \text{ is a monomial}\}$
3: $\tilde{M} \leftarrow$ Reduction to a Row Echelon Form of M
4: $var \leftarrow Check\ Action\ Matrix\ Conditions(\tilde{M}, variables, algB)$
5: **while** $var = \emptyset$ **do**
6: $\quad maxdeg \leftarrow maxdeg + 1$
7: $\quad M \leftarrow M \cup \{m \times f_i \mid f_i \in F;\ \deg(m \times f_i) = maxdeg;\ m \text{ is a monomial}\}$
8: $\quad \tilde{M} \leftarrow$ Reduction to a Row Echelon Form of M
9: $\quad var \leftarrow Check\ Action\ Matrix\ Conditions(\tilde{M}, variables, algB)$
10: **end while**
11: **return** $(\tilde{M}, var)$

In this whole process we need to keep track how the matrix M was built. Recall that each coefficient of the polynomials f_i has unique identifier assigned in the equations parser. Because the whole matrix M contains only the polynomials f_i or their multiples with monomials therefore in the matrix M appear only the coefficients from the polynomials f_i. We just have to keep the positions of the coefficients. This is done by matrix $\hat{M}$. The matrix $\hat{M}$ is built in the same time as the matrix M by this way:

if we put a coefficient into the matrix M we also put the corresponding indentifier to the matrix $\hat{M}$ at the same possition. The matrix $\hat{M}$ enables us to recover the process of polynomials generation in the code generator module.

3.1.5 Removing unnecessary polynomials and monomials

Since in the previous step the polynomials were generated systematically there may appear some polynomials which are not necessary for the constructing of the action matrix. The goal of this part of the automatic generator is to remove as many as possible not necessary polynomials.

We can remove a polynomial r from the matrix M if the corresponding eliminated matrix $\tilde{M}$ still contains all required polynomials q_i. In this way we try to remove all polynomials from M.

Because the success of removing a polynomial depends on the previous removals, the number of removed polynomials depends on the ordering in which the polynomials are removed. In the automatic generator we start removing polynomials from the one with the largest leading monomial to the polynomial with the smallest leading monomial in order by monomial ordering used. Because it is very inefficient to remove polynomials one by one and perform each time an expensive Gauss-Jordan elimination, we can enhance the procedure by trying to remove more polynomials at the time. In the automatic generator there is used this heuristic: if we have successfully removed k polynomials, we try to remove $2 \cdot k$ polynomials in the next step. If the removal of k polynomials have failed we try to remove $\frac{1}{4}k$ polynomials in the next step. The pseudocode of this removing process is shown as Algorithm 10.

Moreover we can reduce the size of the matrix M by removing unnecessary monomials. A monomial is unnecessary when its removal does not affect the building of the action matrix. We have to keep all monomials such that they are leading monomials of polynominals in the corresponding matrix $\tilde{M}$ and all monomials that are present in the basis B. All other monomials can be removed. If we remove all such unnecessary monomials then the matrix M will have dimensions $n \times (n+N)$ where n is the number of the polynomials in the matrix M and N is the number of solutions of the given system.

3.1.6 Construction of the action matrix

This part of the automatic generator starts with the eliminated matrix $\tilde{M}$ of polynomials and variable x_k for which all required polynomials q_i are present in the $\tilde{M}$.

Let us describe the construction of the action matrix in an informal and practical way rather than by using the theory. If the theory is needed it can be found in [10]. The action matrix M_{x_k} corresponding to the variable x_k is a square matrix of dimensions $N \times N$ where N is the number of elements of the monomial basis B. Each row and column corresponds to a monomial $b_i \in B$. Let the monomials b_i are sorted such that if $b_l \prec b_k$ then $k < l$ where $\prec$ is a monomial ordering used. To the i-th row we put coefficients of the polynomial $m_i = \overline{(x_k \times b_i)}^F$ where F are polynomials corresponding to $\tilde{M}$. Because $\tilde{M}$ is in a row echelon form there are two possibilities how the i-th row can be constructed:

1. $x_k \times b_i = b_j$ for some $b_j \in B$
   That means that $x_k \times b_i$ is irreducible by F and m_i is a monomial in B. In this case we set $(i, j) = 1$ and $(i, k) = 0$ where $k \neq j$.

Algorithm 10 Remove unnecessary polynomials

Input:
M a matrix representing a set of polynomials
$variable$ a variable
$algB$ a monomial basis B
Output:
M a matrix representing a set of polynomials

1: $rows \leftarrow$ number of rows of M
2: $step \leftarrow \max\{\lfloor rows/32 \rfloor, 1\}$
3: $up \leftarrow 1$
4: $filter \leftarrow \{1, 2, \ldots, rows\}$
5: **while** $up \leq rows$ **do**
6: $down \leftarrow up + step - 1$
7: **if** $down > rows$ **then**
8: $down \leftarrow rows$
9: $step \leftarrow down - up + 1$
10: **end if**
11: $filter_{Old} \leftarrow filter$
12: $filter \leftarrow filter \backslash \{up, up + 1, \ldots, down\}$
13: $\tilde{M} \leftarrow$ Reduction to a Row Echelon Form of M only with rows specified by $filter$
14: $v \leftarrow Check\ Action\ Matrix\ Conditions(\tilde{M}, variable, algB)$
15: **if** $v = variable$ **then**
16: $up \leftarrow down + 1$
17: $step \leftarrow 2 \times step$
18: **else**
19: **if** $step = 1$ **then**
20: $up \leftarrow up + 1$
21: **else**
22: $step \leftarrow \max\{\lfloor step/4 \rfloor, 1\}$
23: **end if**
24: $filter \leftarrow filter_{Old}$
25: **end if**
26: **end while**
27: **return** M only with rows specified by $filter$

2. $x_k \times b_i \neq b_j$ for all $b_j \in B$
In this case there is f such that $\mathrm{LM}(m_i) = \mathrm{LM}(f)$ where $f \in F$ so $m_i = x_k \times b_i - f$. Since all monomials of f except $\mathrm{LM}(f)$ are from B, all monomials of m_i are also from B. On the (i, j) position of the matrix M_{x_k} we put coefficient of m_i at the monomial b_j.

Now the solutions of the given system can be easily found by computing right eigenvectors of the action matrix M_{x_k}.

3.1.7 Solver generator

The last task of the automatic generator is to create a solver which will solve the given polynomial system for an arbitrary set of parameters. The current version of the automatic generator can generate solvers for MATLAB and Maple, but new code

generators can be easily added. Which solvers will be generated can be set in the minimal problem definition by setting `cfg.exportCode`, e.g. to create both MATLAB and Maple solvers we set `cfg.exportCode = {'matlab' 'maple'}`.

To create the solver we have to restore the process of creation of the matrix M. This process is saved as the matrix $\hat{M}$ which contains unique identifiers on the positions where the given parameters have to be put. So the matrix M can be built for each given set of parameters. Then the Gauss-Jordan elimination is called on M so we get the matrix $\tilde{M}$. Now the action matrix is built in the same way as above and the solutions are extracted from it. To sum up the final solver just creates the matrix M by putting parameters to the correct places. After Gauss-Jordan elimination the action matrix is built by copying some parts of rows of $\tilde{M}$ and then the solutions are extracted by using the eigenvectors of the action matrix.

3.1.8 Usage

The automatic generator is designed to be able to be used even by non-expert users and to be easily expanded or improved.

At first the script `setpaths.m` from the root directory of the automatic generator should be executed. This will add all required paths to the MATLAB environment.

Next we have to set up the definition of the minimal problem we want to solve. How the definition have to be specified is written in the section 3.1.1. All these definitions are stored in the folder `minimalProblems`. To generate the solver we call the function `gbs_GenerateSolver(MinimalProblem)` where `MinimalProblem` is the name of the definition of the minimal problem, i.e. the name of the function in the folder `minimalProblems`. This will generate us solvers `solver_MinimalProblem.m` for the MATLAB solver and `solver_MinimalProblem.txt` for the Maple solver. These solvers are stored in the folder `solvers`.

For example we want to generate solver for the 6-point focal length problem. We have defined this problem as a function `sw6pt.m` in the folder `minimalProblems`. By calling the function `gbs_GenerateSolver('sw6pt')` we get solvers `solver_sw6pt.m` and `solver_sw6pt.txt` in the folder `solvers`.

3.2 Improvements

The bottleneck of the automatic generator is the polynomial generator module. Since the polynomials are generated systematically the matrices in the resultant solvers are often bigger than necessary which means that the solvers are not efficient. So many improvements of the automatic generator [12] can be done. For example if we want to generate multiple elimination solvers as suggested in [10] the polynomial generator module have to be improved or totaly replaced. In the same way some strategies from other algorithms, for example from the F_4 Algorithm [6], can be taken over and implemented into the automatic generator. Because we are mostly working with sparse matrices in the automatic generator the variant of Gauss-Jordan elimination for sparse matrices can be implemented to save some computation time.

3.2.1 Reimplementation

The previous implementation [12] of the automatic generator was implemented in MATLAB R2008. It shows up that new versions of MATLAB are not backward compatible so the automatic generator fails when lauched in some newer version than R2008.

So the first task was to reimplement the old implementation into new version of MATLAB. All the changes of the code were minor and just in the implementation level.

One important change is that the Maple Toolbox for MATLAB in not compatible with the MATLAB Symbolic Math Toolbox. One of the options in the monomials basis B module is to use the Maple toolbox to compute the basis B. Because of the incompatibility this option can be used no longer and the algebraic geometry software Macaulay2 [9] have to be used.

The new version of the automatic generator, as it is desribed in this thesis, is compatible with the version R2015 of MATLAB 64-bit and 32-bit under Windows and Unix operation systems.

3.2.2 Multiple elimination solver

In the section 3.1.4 there is desribed the starategy how to generate polynomials for one eliminiation solvers. But in some cases it may be better to create multiple elimination solvers. A multiple elimination solver is a solver where polynomials are generated systematically by multiplying already generated polynomials by monomials and reduced each time by Gauss-Jordan elimination. So the task of this section is to describe how the polynomial generator of the automatic generator can be improved to be able to generate polynomials for multiple elimination solvers.

In this strategy we generate in each step all polynomials up to degree d and perform a Gauss-Jordan elimination on them. We increase the degree d when no new polynomials can be generated by multiplying already generated polynomials by some monomials. We stop this process when all polynomials q_i are generated.

This strategy is very usefull especially when we are generating solvers for systems with many variables. The reason is that the increasing of the degree d of generated polynomials leads to a large number of new generated polynomials.

Now let us look on the process of generating polynomials in more details. The pseudocode is shown as Algortihm 11. Let the *maxdeg* is a maximal total degree of all polynomials from the given system F. At the beginning we put into the matrix M_1 all polynomials up to degree *maxdeg* such that they are multiples of polynomials from F and some monomials. We get the matrix $\tilde{M}_1$ by eliminating the matrix M_1. We check if there exists a variable for which all polynomials q_i are generated in $\tilde{M}_1$. If no such variable is found we generate new polynomials with higher total degree. We increase the degree *maxdeg* up to which we are generating polynomials. The variable *step* tells how much we want to increase the total degree in one step. Into the variable *mindeg* we save the *maxdeg* from the previous iteration incremented by one to keep track to which degree we have generated polynomials yet. We build new matrix M_2 from the matrix $\tilde{M}_1$ and we add all polynomials with total degrees from *mindeg* to *maxdeg*. These polynomials are multiples of polynomials from $\tilde{M}_1$ and some monomials. We save the result of the Gauss-Jordan elimination as the matrix $\tilde{M}_2$. We repeat this process until no new polynomials are added in one iteration. That situation happens when two matrices $\tilde{M}_j$ and $\tilde{M}_{j-1}$ have same number of non zero rows. In this case we check if all polynomials q_i are generated for some variable. If not we have to generate new polynomials with higher total degree. If the variable has been found we return the last generated matrix $\tilde{M}_j$ and the found variable. You may notice that when we are leaving the repeat-until loop on line 13 there are two equivalent matrices $\tilde{M}_j$ and $\tilde{M}_{j-1}$. These matrices have the same non zero rows and differ only with the number of zero rows. This is the reason why we can remove the matrix $\tilde{M}_j$ by decrementing the variable j on line 14.

Algorithm 11 Polynomial generator – Multiple elimination solver

Input:
F a set of polynomials
variables a list of variables
$algB$ a monomial basis B
step an integer

Output:
$\tilde{M}$ a matrix representing a set of polynomials
var a variable

1: $maxdeg \leftarrow \max\{\deg(f_i) \mid f_i \in F\}$
2: $j \leftarrow 1$
3: $M_1 \leftarrow \{m \times f_i \mid f_i \in F;\ \deg(m \times f_i) = \deg(f_i), \ldots, maxdeg;\ m \text{ is a monomial}\}$
4: $\tilde{M}_1 \leftarrow$ Reduction to a Row Echelon Form of M_1
5: $var \leftarrow Check\ Action\ Matrix\ Conditions(\tilde{M}_1, variables, algB)$
6: **while** $var = \emptyset$ **do**
7: $\quad mindeg \leftarrow maxdeg + 1$
8: $\quad maxdeg \leftarrow maxdeg + step$
9: $\quad$ **repeat**
10: $\quad\quad j \leftarrow j + 1$
11: $\quad\quad M_j \leftarrow \tilde{M}_{j-1} \cup \Big\{m \times f_i \mid f_i \in \tilde{M}_{j-1};$
$\quad\quad\quad \deg(m \times f_i) = mindeg, \ldots, maxdeg;\ m \text{ is a monomial}\Big\}$
12: $\quad\quad \tilde{M}_j \leftarrow$ Reduction to a Row Echelon Form of M_j
13: $\quad$ **until** number of nonzero rows of $\tilde{M}_j$ = number of nonzero rows of $\tilde{M}_{j-1}$
14: $\quad j \leftarrow j - 1$
15: $\quad var \leftarrow Check\ Action\ Matrix\ Conditions(\tilde{M}_j, variables, algB)$
16: **end while**
17: **return** $(\tilde{M}_j, var)$

To be able to restore the process of generation of polynomials and generate the code of the solver in the solver generator module we have to keep track how the matrices M_j were built. Therefore we build a matrix $\hat{M}_j$ for each matrix M_j. The first matrix $\hat{M}_1$ is built in the same way as the matrix $\hat{M}$ when generating one elimination solver. This matrix contains only the unique identifiers of parameters on positions were real values will be put. Because each matrix M_j is built from the matrix $\tilde{M}_{j-1}$, the matrix M_j contains only coefficients from the matrix $\tilde{M}_{j-1}$. So when a coefficient from (m, n) position from $\tilde{M}_{j-1}$ is put into M_j at (k, l) position, the tuple (m, n) is put at the position (k, l) of $\hat{M}_j$. So if we have the matrix $\tilde{M}_{j-1}$ and the matrix $\hat{M}_j$, the matrix M_j can be built easily.

To enable generation of multiple elimination solvers we assign an integer to `cfg.PolynomialsGeneratorCfg.GJstep` in the settings of the automatic generator. The value of the `GJstep` has the same meaning as the variable *step* from the Algorithm 11. E.g. by setting `GJstep = 1` the generated polynomials will be eliminated each time the total degree of generated polynomials is incremented by 1. If we set `GJstep = 0` the one elimination solver will be generated as written in the section 3.1.4.

3.2.3 Removing redundant polynomials

We may notice that there appear matrices $\tilde{M}_j$ with many zero rows in the process of generation polynomials for multiple elimination solvers. This is caused because there are many dependent rows in the corresponding matrices M_j. These zero rows in matrices $\tilde{M}_j$ give us no new information so we want to remove as many as possible rows from the matrices M_j such that the resulting matrices $\tilde{M}_j$ will have the same nonzero rows as before the removal and no zero rows.

We know that we can remove the same number of rows from M_j as it is the number of zero rows in $\tilde{M}_j$. But we do not know which rows we should remove. So we try to remove each row r from M_j and if the number of nonzero rows of $\tilde{M}_j$ stays the same the removal is successful, if not we have to return the row r back into M_j. We end this process of removal when there is no zero row in the matrix $\tilde{M}_j$. Because performing the Gauss-Jordan elimination in each step of removing single row is inefficient we use the same heuristic as desribed in the section 3.1.5. For better understanding we are providing the pseudocode of this removing as Algorithm 12.

Because this removing process removes only zero rows from the matrices $\tilde{M}_j$ and no others rows are touched it does not influences the process of adding polynomials. Therefore this removing process is enabled by default and can not be disabled by any option from the automatic generator settings.

3.2.4 Matrix partitioning

In the automatic generator the matrices we are dealing with are mostly sparse, so some more efficient techniques can be used to work with them. This will of course result into generation of more efficient and numerically stable solvers.

In this section we focus on how to speed up the Gauss-Jordan elimination of sparse matrices. We use the technique proposed in [11]. We observe that by permuting the rows and the columns of sparse matrices they can be transformed into matrices with block-diagonal structure known as singly-bordered block-diagonal (SBBD) form. Each diagonal block of the SBBD matrices forms an independent problem and therefore it can be independently eliminated. This speeds up the process of Gauss-Jordan elimination, because eliminating more smaller matrices is faster than eliminating one big matrix. If we divide the matrix into k independent blocks that contain comparable number of entries the speed up is approximately $n^3 \rightarrow k \cdot \left(\frac{n}{k}\right)^3$. Moreover the permutation matrices that transform matrices to the SBBD forms are precomputed during the solver generation process and therefore the resultant solver is working already with matrices in the SBBD forms and does not have to spend time to computing the permutation matrices again.

Let us say we want to eliminate matrix M. First of all we have remove columns that correspond to monomials from the basis B from the matrix, because these columns should not be permuted and eliminated. Then we need to compute the permutation matrix. To compute the permutation matrices we use the state of the art hypergraph partitioning tool PaToH [3] with settings to divide the matrix into two independent blocks. We do the permutations of rows a columns and get two diagonal blocks M_{11} and M_{22} and some coupling columns that can not be assigned to any of the diagonal blocks. Next we perform two independent eliminations of the blocks M_{11} and M_{22} and permute all rows of M to get the identity matrix in the left top corner. However we get non-eliminated submatrix in the right bottom corner of M. After eliminating this submatrix the rows above this submatrix are still not eliminated. If this is the last

Algorithm 12 Remove redundant polynomials

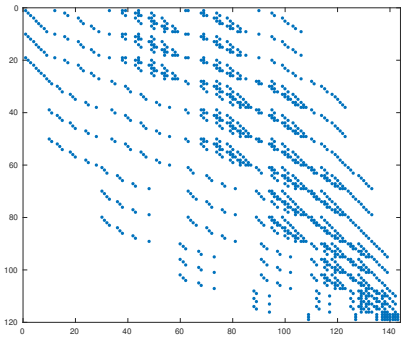
Input:
M a matrix representing a set of polynomials
$\tilde{M}$ a matrix in a Row Echelon Form representing a set of polynomials
Output:
M a matrix representing a set of polynomials
$\tilde{M}$ a matrix in a Row Echelon Form representing a set of polynomials

```
1: toRemove ← number of zero rows of M̃
2: nonZero ← number of non zero rows of M̃
3: rows ← number of rows of M
4: step ← max{⌊toRemove/4⌋, 1}
5: up ← 1
6: filter ← {1, 2, ..., rows}
7: while toRemove ≠ 0 do
8:     down ← up + step − 1
9:     if down > rows then
10:        down ← rows
11:        step ← down − up + 1
12:    end if
13:    filter_Old ← filter
14:    filter ← filter\{up, up + 1, ..., down}
15:    M̃ ← Reduction to a Row Echelon Form of M only with rows specified by filter
16:    if number of non zero rows of M̃ < nonZero then
17:        if step = 1 then
18:            up ← up + 1
19:        else
20:            step ← max{⌊step/4⌋, 1}
21:        end if
22:        filter ← filter_Old
23:    else
24:        toRemove ← number of zero rows of M̃
25:        up ← down + 1
26:        step ← min{2 × step, toRemove}
27:    end if
28: end while
29: return (M only with rows specified by filter, M̃)
```

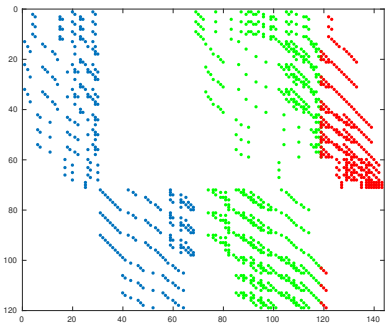
elimination in the solver we have to eliminate only the rows which we need to build the action matrix. If this is not the last elimination in the solver we have to eliminate them all.

Example Consider sparse matrix M of size 119×143 as it is shown in Figure 3.2a. Columns 119 – 143 correspond to monomials from the monomial basis B. When we remove these columns we get rectangular matrix of size 119×119 on which we execute the partitioning tool PaToH. After the permutation of the rows and columns we get the matrix in the SBBD form which can be seen in Figure 3.2b. We get two diagonal blocks M_{11} of size 71×30 and M_{22} of size 48×38 and 51 coupling columns which can not be assigned to any diagonal block. After two independent Gauss-Jordan eliminations

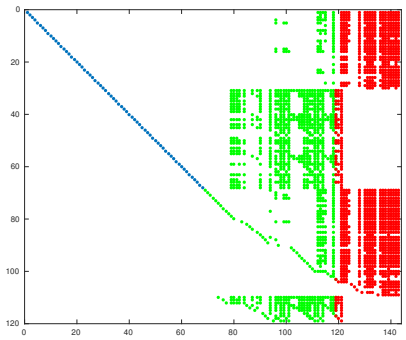
of the blocks M_{11} and M_{22} and row permutation we get the identity matrix in the left top corner, see Figure 3.2c. Now we have to perform the Gauss-Jordan elimination on submatrix of size 51×75 in the right bottom corner. After this we get matrix as it is shown in Figure 3.2d. Now in the matrix lefts uneliminated submatrix of size 68×75 in the right top corner. If this is the last elimination in the solver we do not have to eliminate it whole, but it is sufficient to eliminate only the rows we need to the constructing of the action matrix. If this is not the last elimination in the solver we have to eliminate the whole remaining uneliminated submatrix.



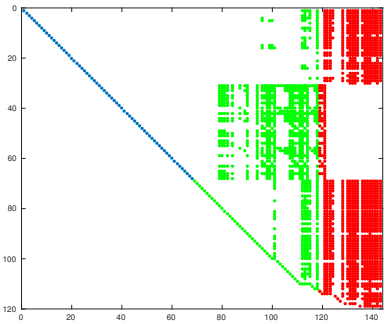
(a) The input matrix



(b) The SSBD form of this matrix



(c) A matrix obtained after two independent Gauss-Jordan eliminations



(d) A matrix obtained after elimination of the right bottom submatrix

Figure 3.2 Example of the process of eliminating matrix using matrix partitioning. Colors: blue – two independent diagonal blocks, green – coupling columns, red – columns corresponding to the basis B

Since the matrix partitioning have not to be efficient for all minimal problems, it can be easily enabled or disabled in the automatic generator. By setting `cfg.matrixPartitioning = 'all'` the matrix partitioning is used to all Gauss-Jordan eliminations in the solver. If `cfg.matrixPartitioning = 'last'` is set the matrix partitioning is used only to the last Gauss-Jordan elimination in the solver. Matrix partitioning can totally disabled by setting `cfg.matrixPartitioning = 'none'`. At last we want to warn that the tool PaToH is not available under Windows OS so the matrix partitioning can not be used under this system.

3.2.5 F4 strategy

Because polynomials in the polynomial generator module as it is described in section 3.1.4 are generated systematically, many of them are supefluous and therefore need not be generated. Since the automatic generator consists of independent modules we can replace the polynomial generator module by some better implementation.

In the chapter 2 we have described some of the state of the art techniques how polynomial systems can be solved. Therefore we can take over some of them and implement them into the automatic generator. In the section 2.2 we have understood the F_4 [6] Algorithm so we have implemented this technique into the automatic generator.

Implementation of the F_4 Algorithm in the automatic generator is the same as J. Ch. Faugère has described in [6] and we did in section 2.2. The only difference is that we need to track how the polynomials are constructing to be able to reconstruct the process in the solver generator module. In the F_4 Algorithm, to be concrete in the function *Symbolic Preprocessing*, we are constructing matrices F_i from the selected pairs and polynomials from Gröbner basis G that are multiplied by a monomial. Polynomials that are constructed from the selected critical pairs are from G and multiplied by a monomial too. But polynomials in G are just the input polynomials or polynomials from $\tilde{F}_i$ from all previous iterations. All these polynomials that are added into F_i are simplified by the function *Simplify*. That means that such a polynomial may be replaced by another polynomial taken from $\tilde{F}_i$. To sum up the matrix F_i is built from multiples of the input polynomials or polynomials from $\tilde{F}_{1,\ldots,i-1}$ and monomials. So to track the process of building the matrices F_i we just have to keep which polynomials from which matrices $\tilde{F}_i$ (we can look at the input polynomials as if it is a matrix $\tilde{F}_0$) multiplied with which monomials are used which can be easily done. In the end we have to recover how the matrix G was build, but this is the same case as reconstructing a matrix F_i, because the Gröbner basis G consists only of the input polynomials or polynomials from matrices $\tilde{F}_i$.

Unnecessary and redundant polynomials can be of course removed from the matrices F_i as we have described in sections 3.1.5 and 3.2.3. Moreover if nothing is added from the matrix $\tilde{F}_i$ to the Gröbner basis G, i.e. the matrix $\tilde{F}_i^+$ is empty, the matrix F_i can be removed totally. This leads to that in the resultant generated solver there will be no reduction to zero so much computation time can be saved. We are still working with sparse matrices so the matrix partitioning which is described in section 3.2.4 can be used to speed up the Gauss-Jordan eliminations.

By default this strategy is disabled in the automatic generator and the systematical polynomial generator as described in sections 3.1.4 and 3.2.2 is used. This is done by setting `cfg.PolynomialsGenerator = 'systematical'`. To enable the polynomial generator using the F_4 strategy set `cfg.PolynomialsGenerator = 'F4'` in the automatic generator settings.

The key function in the F_4 strategy is the *Sel* function which is descibed in section 2.2.6. While different *Sel* functions can be efficient for different minimal problems, it can be easily changed which function will be used. To use the normal strategy set `cfg.PolynomialsGeneratorCfg.Sel = @F4_SelNormal` and to select only one critical pair each time set `cfg.PolynomialsGeneratorCfg.Sel = @F4_SelFirst` in the automatic generator settings. This second function emulates the Buchberger Algorithm. Both this functions are stored in the folder `generator/F4` and everybody can implement and use his own *Sel* function.

3.3 Benchmark

4 Experiments

5 Conclusion

Bibliography

[1] Thomas Becker and Volker Weispfenning. *Gröbner Bases, A Computational Approach to Commutative Algebra*. Number 141 in Graduate Texts in Mathematics. Springer-Verlag, New York, NY, 1993. 5, 6

[2] Bruno Buchberger. *Ein Algorithmus zum Auffinden der Basiselemente des Restklassenringes nach einem nulldimensionalen Polynomideal*. PhD thesis, Mathematical Institute, University of Innsbruck, Austria, 1965. 5

[3] Umit V. Catalyurek and Cevdet Aykanat. Patoh: Partitioning tool for hypergraphs, 2011. Version 3.2. 22

[4] David Cox, John Little, and Donald O'Shea. *Ideals, Varieties, and Algorithms : An Introduction to Computational Algebraic Geometry and Commutative Algebra*. Undergraduate Texts in Mathematics. Springer, New York, USA, 2nd edition, 1997. 5, 12

[5] Christian Eder, Justin Gash, and John Perry. Modifying faugère's f5 algorithm to ensure termination. *ACM Communications in Computer Algebra*, 45(2):70–89, June 2011. 13

[6] Jean-Charles Faugère. A new efficient algorithm for computing gröbner bases (f_4). *Journal of pure and applied algebra*, 139(1–3):61–88, July 1999. 7, 19, 25

[7] Jean-Charles Faugère. A new efficient algorithm for computing gröbner bases without reduction to zero (f_5). In *Papers from the International Symposium on Symbolic and Algebraic Computation*, ISSAC '02, pages 75–83, New York, NY, USA, 2002. ACM. 12, 13

[8] Rüdiger Gebauer and Hans-Michael Möller. On an installation of buchberger's algorithm. *Journal of Symbolic Computation*, 6(2–3):275–286, October 1988. 6, 10

[9] Daniel R. Grayson and Michael E. Stillman. Macaulay2, a software system for research in algebraic geometry. http://www.math.uiuc.edu/Macaulay2/. [Online; accessed 2015-04-28]. 15, 20

[10] Zuzana Kukelova. *Algebraic Methods in Computer Vision*. PhD thesis, Department of Cybernetics, Faculty of Electrical Engineering, Czech Technical University in Prague, February 2013. 14, 17, 19

[11] Zuzana Kukelova, Martin Bujnak, Jan Heller, and Tomas Pajdla. Singly-bordered block-diagonal form for minimal problem solvers. In *Computer Vision - ACCV 2014 - 12th Asian Conference on Computer Vision, Singapore, Revised Selected Papers, Part II*, pages 488–502. Springer International Publishing, November 12–18 2014. 22

[12] Zuzana Kukelova, Martin Bujnak, and Tomas Pajdla. Automatic generator of minimal problem solvers. In *Proceedings of The 10th European Conference on Computer Vision*, ECCV 2008, October 12–18 2008. 14, 19

[13] Hans-Michael Möller, Teo Mora, and Carlo Traverso. Gröbner bases computation using syzygies. In *Papers from the International Symposium on Symbolic and Algebraic Computation*, ISSAC '92, pages 320–328, New York, NY, USA, 1992. ACM. 13

[14] Tomas Pajdla, Zuzana Kukelova, and Martin Bujnak. Minimal problems in computer vision. http://cmp.felk.cvut.cz/minimal/. [Online; accessed 2015-04-28]. 14

[15] Tomas Pajdla, Zuzana Kukelova, and Martin Bujnak. Minimal problems in computer vision. http://cmp.felk.cvut.cz/minimal/automatic_generator.php. [Online; accessed 2015-04-28]. 14

[16] A. J. M. Segers. Algebraic attacks from a gröbner basis perspective. Master's thesis, Department of Mathematics and Computing Science, Technische Universiteit Eindhoven, 2004. 13

[17] Till Stegers. Faugère's f5 algorithm revisited. Master's thesis, Department Of Mathematics, Technische Universit ät Darmstadt, revisited version 2007. 13