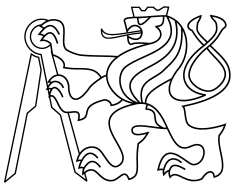
CENTER FOR
MACHINE PERCEPTION



CZECH TECHNICAL
UNIVERSITY IN PRAGUE

BACHELOR THESIS

Minimal Problem Solver Generator

Pavel Trutman

pavel.trutman@fel.cvut.cz

May 14, 2015

Available at
http://cmp.felk.cvut.cz/~trutmpav/bachelor-thesis/thesis/thesis.pdf

Thesis Advisor: Ing. Tomáš Pajdla, PhD.

This work has been supported by FP7-SPACE-2012-312377
PRoViDE and ATOM TA02011275 grants.

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

Czech Technical University in Prague
Faculty of Electrical Engineering

Department of Cybernetics

BACHELOR PROJECT ASSIGNMENT

Student: Pavel T r u t m a n

Study programme: Cybernetics and Robotics

Specialisation: Robotics

Title of Bachelor Project: Minimal Problem Solver Generator

Guidelines:

1. Review the state of the art in solving the polynomial systems using linear algebra [1, 2] and the automatic generator of the polynomial solvers [3, 4].
2. Implement the improvement [4] of [3] into the existing automatic generator of solvers.
3. Implement a variation of algorithm [1], review its behavior, and suggest how to take over some of its elements to the automatic generator and implement it.
4. Demonstrate the functionality of the new solver generator and compare it with the original solver.

Bibliography/Sources:

[1] Faugere, J.-C. (June 1999): "A new efficient algorithm for computing Gröbner bases (F4)". Journal of Pure and Applied Algebra (Elsevier Science) 139 (1): 61–88. doi:10.1016/S0022-4049(99)00005-5. ISSN 0022-4049.
[2] Faugere, J.-C. (July 2002): "A new efficient algorithm for computing Gröbner bases without reduction to zero (F5)". Proceedings of the 2002 international symposium on Symbolic and algebraic computation (ISSAC) (ACM Press): 75–83.
[3] Kukelova, Z.: Algebraic Methods in Computer Vision. PhD Thesis. CTU in Prague 2013. http://cmp.felk.cvut.cz/~kukelova/webthesis/docs/Kukelova-phd-2013.pdf
[4] Kukelova, Z.; Bujnak, M.; Heller, J.; Pajdla, T.: Singly-Bordered Block-Diagonal Form for Minimal Problem Solvers. ACCV 2014.

Bachelor Project Supervisor: Ing. Tomáš Pajdla, Ph.D.

Valid until: the end of the summer semester of academic year 2015/2016

L.S.

doc. Dr. Ing. Jan Kybic
Head of Department

prof. Ing. Pavel Ripka, CSc.
Dean

Prague, December 9, 2014

České vysoké učení technické v Praze
Fakulta elektrotechnická

Katedra kybernetiky

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Student: Pavel T r u t m a n

Studijní program: Kybernetika a robotika (bakalářský)

Obor: Robotika

Název tématu: Generátor řešení minimálních problémů

Pokyny pro vypracování:

1. Prozkoumejte současné metody řešení polynomiálních rovnic [1, 2] a automatický generátor postupu řešení [3, 4].
2. Implementujte vylepšení [4] generátoru [3] do existujícího generátoru postupů řešení.
3. Implementujte variaci algoritmu [1], experimentujte s jeho chováním a navrhněte, co převzít z [1] do implementace generátoru a implementujte to.
4. Demonstrujte funkčnost nového generátoru postupu řešení a porovnejte ji se starým generátorem.

Seznam odborné literatury:

[1] Faugere, J.-C. (June 1999): "A new efficient algorithm for computing Gröbner bases (F4)". Journal of Pure and Applied Algebra (Elsevier Science) 139 (1): 61–88. doi:10.1016/S0022-4049(99)00005-5. ISSN 0022-4049.
[2] Faugere, J.-C. (July 2002): "A new efficient algorithm for computing Gröbner bases without reduction to zero (F5)". Proceedings of the 2002 international symposium on Symbolic and algebraic computation (ISSAC) (ACM Press): 75–83.
[3] Kukelova, Z.: Algebraic Methods in Computer Vision. PhD Thesis. CTU in Prague 2013. http://cmp.felk.cvut.cz/~kukelova/webthesis/docs/Kukelova-phd-2013.pdf
[4] Kukelova, Z.; Bujnak, M.; Heller, J.; Pajdla, T.: Singly-Bordered Block-Diagonal Form for Minimal Problem Solvers. ACCV 2014.

Vedoucí bakalářské práce: Ing. Tomáš Pajdla, Ph.D.

Platnost zadání: do konce letního semestru 2015/2016

L.S.

doc. Dr. Ing. Jan Kybic
vedoucí katedry

prof. Ing. Pavel Ripka, CSc.
děkan

V Praze dne 9. 12. 2014

Acknowledgements

I would like to thank my advisor Tomáš Pajdla for introducing me into Gröbner basis methods for solving polynomial systems and for his guidance and advices which enabled me to finish this thesis. I would also like to thank Zuzana Kúkelová for presenting me the automatic generator and for her comments to improvements I have implemented. Special thanks go to my family for all their support.

Author's declaration

I declare that I have work out the presented thesis independently and that I have listed all information sources used in accordance with the Methodical Guidelines about Maintaining Ethical Principles for Writing Academic Theses.

Prohlášení autora práce

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze dne ....

Podpis autora práce

Abstract

Many problems in computer vision lead to polynomial systems solving. Therefore, we need an easy way how to generate an efficient solver for each problem. On this purpose, the automatic generator has been presented. In this thesis, we improve the automatic generator so we will be able to generate more efficient and numerically stable solvers.

To improve the automatic generator we review and implement several methods used in the state of the art Gröbner basis solvers. Especially, we focus on the F_4 Algorithm by Jean-Charles Faugère. Solvers, generated by the automatic generator, can be sped up when efficient methods are used to work with sparse matrices. We describe and implement method which is based on matrix partitioning. This method significantly speeds up the Gauss-Jordan elimination of sparse matrices.

We demonstrate the enhancements of the automatic generator on several important minimal problems. We show that the solvers generated by the new automatic generator are faster and numerically more stable than the solvers generated by the old version of the automatic generator.

Keywords: computer vision, robotics, minimal problems, polynomials equations, Gröbner basis

Abstrakt

Mnoho problémů v počítačovém vidění vede na řešení polynomiálních rovnic. Proto potřebujeme jednoduchý způsob, jak generovat efektivní postupy řešení každého z problémů. Z tohoto důvodu byl představen automatický generátor. V této práci vylepšíme automatický generátor, takže budeme schopni generovat ještě rychlejší a numericky stabilnější postupy řešení polynomiálních systémů.

Abychom mohli vylepšit automatický generátor, prozkoumáme a následně implementujeme několik metod používaných v současných nástrojích na řešení soustav polynomiálních rovnic pomocí Gröbnerových bází. Zaměříme se zejména na algoritmus F_4 představený Jean-Charlesem Faugèrem. Postupy řešení problémů, vygenerované pomocí automatického generátoru, mohou být ještě dále zrychleny, pokud použijeme efektivní metody pro práci s řídkými maticemi. Popíšeme a implementujeme metodu, která je založená na rozkladu matic. Tato metoda výrazně urychluje Gauss-Jordanovu eliminaci řídkých matic.

Vylepšení automatického generátoru předvedeme na několika významných minimálních problémech. Ukážeme, že postupy řešení problémů vygenerované novým automatickým generátorem jsou rychlejší a numericky stabilnější než postupy vygenerované původní verzí automatického generátoru.

Klíčová slova: počítačové vidění, robotika, minimální problémy, polynomiální rovnice, Gröbnerovy báze

Contents

1. Introduction **5**
- 1.1. Motivation . . . 5
- 1.2. Thesis structure . . . 5
- 1.3. Notation used . . . 6

2. Polynomial system solving **7**
- 2.1. Buchberger Algorithm . . . 7
  - 2.1.1. First implementation . . . 7
  - 2.1.2. Improved Buchberger Algorithm . . . 8
- 2.2. F_4 Algorithm . . . 9
  - 2.2.1. Improved Algorithm F_4 . . . 9
  - 2.2.2. Function Update . . . 12
  - 2.2.3. Function Reduction . . . 12
  - 2.2.4. Function Symbolic Preprocessing . . . 12
  - 2.2.5. Function Simplify . . . 13
  - 2.2.6. Selection strategy . . . 13
- 2.3. F_5 Algorithm . . . 14

3. Automatic generator **16**
- 3.1. Description of the automatic generator . . . 16
  - 3.1.1. Definition of the minimal problem . . . 16
  - 3.1.2. Equations parser, instantiating . . . 17
  - 3.1.3. Monomial basis B computation . . . 17
  - 3.1.4. Polynomial generator . . . 17
  - 3.1.5. Removing unnecessary polynomials and monomials . . . 19
  - 3.1.6. Construction of the action matrix . . . 19
  - 3.1.7. Solver generator . . . 20
  - 3.1.8. Usage . . . 21
- 3.2. Improvements . . . 21
  - 3.2.1. Reimplementation . . . 21
  - 3.2.2. Multiple elimination solver . . . 22
  - 3.2.3. Removing redundant polynomials . . . 24
  - 3.2.4. Matrix partitioning . . . 24
  - 3.2.5. F4 strategy . . . 27
    - Implementation in Maple . . . 27
    - Integration into the automatic generator . . . 27
- 3.3. Benchmark . . . 30
  - 3.3.1. Structure . . . 30
  - 3.3.2. Usage . . . 32

4. Experiments **33**
- 4.1. Multiple elimination solver . . . 33
- 4.2. Matrix partitioning . . . 34

4.3. F_4 strategy 35

5. Conclusion **39**

A. Contents of the enclosed CD **40**

Bibliography **41**

List of Algorithms

1. Simple Buchberger Algorithm 7
2. Improved Buchberger Algorithm 8
3. Update 10
4. Improved Algorithm F_4 11
5. Reduction 12
6. Symbolic Preprocessing 13
7. Simplify 14
8. Sel – The normal strategy for F_4 14

9. Polynomial generator – One elimination solver 18
10. Remove unnecessary polynomials 20
11. Polynomial generator – Multiple elimination solver 23
12. Remove redundant polynomials 25

List of Symbols and 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 the polynomial 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 the polynomials f_1 and f_2.
SBBD form	Singly-bordered block-diagonal form.
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

1.1. Motivation

Many problems in computer vision can be formulated using systems of algebraic equations. Examples are the minimal problems [17] which arise when computing geometrical models from image data. The polynomial systems arising from this problems are often not trivial and they consists of many polynomial equations of higher degree in many unknowns, and therefore general algorithms for solving polynomial systems are not efficient for them. So special solvers for each problem have been developed to solve these system efficiently and numerically robustly.

Minimal problems have a wide range of applications, for example in 3D reconstruction, recognition, robotics and augmented reality. In these applications, the solvers of minimal problems are only small parts of large computation systems which are supposed be fast or even to work in real-time applications. Moreover, these systems need to compute the solutions of the minimal problems repeatedly for large number of parameters. Therefore, very efficient solvers are required in computer vision.

Many solvers for minimal problems have been designed ad hoc for concrete problems, and therefore they can not be used or easily modified to solve different or even similar problems. So the automatic generator [13] has been proposed. This tool generates Gröbner basis solvers automatically which enables us to generate an efficient solver for each problem we want to solve.

There are several ways, how the solver using Gröbner basis methods can be generated. The implementation presented in [13] generates polynomials that are required for solving the system systematically. But other methods can be used. In this thesis, we review the state of the art methods for solving polynomials systems and suggest which methods can be taken over to improve the automatic generator and we implement them.

The automatic generator deals with sparse matrices in most cases. Therefore, we may consider to implement some methods which enable us to work with sparse matrices in an efficient way to save computation time and memory. In this thesis, we focus on how to improve the Gauss-Jordan elimination of sparse matrices. We use the recent work [12] which presents the significant speed up of Gauss-Jordan elimination of sparse matrices. The speed up is caused by transforming matrices into the singly-bordered block-diagonal forms by the paritioning tool PaToH [4]. This method is based on the fact that more eliminations of smaller matrices are faster than one elimination of a big matrix.

1.2. Thesis structure

In this thesis, we firstly review the state of the art methods for computing Gröbner basis of polynomial systems. We start with describing simple, but easily understandable, algorithms and continue with more difficult, but also more efficient, algorithms. It is crucial to us to better understand these algorithms because we will use some techniques from them to improve the automatic generator lately in this thesis.

Secondly, we briefly describe the automatic generator. Then, we suggest some improvements of the automatic generator to generate efficient and numerically stable solvers. Some techniques implementened in the automatic generator may be efficient for some minimal problem, but may be inefficient for another. Therefore, we present a benchmark tool which enables us to choose the best methods to generate an efficient solver in the end.

Thirdly, we run some experiments to show how the implemented improvements have enhanced the automatic generator. We compare the solvers generated by the new automatic generator and the solvers generated by the old implementation.

At last, we conclude by rewieving the contributions of this thesis.

1.3. Notation used

We have decided to use the notation from [5] in the whole thesis. We just remind that polynomial is a sum of terms and term is a product of a coefficient and a monomial. Be aware that in some literature [1, 7, 8] the meanings of words term and monomial are switched.

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, 5].

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 the superfluous polynomials from G according to Gebauer and Möller [9].

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 there is no 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 that such a pair is not moved from the set C to the set D but it is only removed from the set C. Moreover, 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, even if the pairs could be removed. 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 in the end because in the Gröbner basis G, polynomials with leading monomials that 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 [7] by Jean-Charles Faugère is an improved version of the Buchberger's Algorithm. The F_4 Algorithm 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 it is still an open question how to best select the pairs [7]. Some selection strategies are described in the section 2.2.6 on page 13. 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 $\mathrm{LM}(m_1 f_1) = \mathrm{lcm}(\mathrm{LM}(f_1), \mathrm{LM}(f_2))$. 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 its 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 [9] 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.

The 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 $m = m' \cdot \mathrm{LM}(f)$ 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: $m = m' \cdot \mathrm{LM}(f)$ 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 mf 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 $\mathrm{LM}(m'f') = \mathrm{LM}(mf)$. 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 the 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 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** $u \in$ list of all divisors of m **do**
2: **if** $\exists j$ $(1 \leq j \leq d)$ such that $(uf) \in F_j$ **then**
3: $\tilde{F}_j$ is the Row Echelon Form of F_j
4: there exists a (unique) $p \in \tilde{F}_j$ such that $\mathrm{LM}(p) = \mathrm{LM}(uf)$
5: **if** $u \neq m$ **then**
6: **return** *Simplify*$(\frac{m}{u}, p, \mathcal{F})$
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: $d \leftarrow \min\{\deg(\mathrm{lcm}(p)) \mid p \in P\}$
2: $P_d \leftarrow \{p \in P \mid \deg(\mathrm{lcm}(p)) = d\}$
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 [8] by Jean-Charles Faugère was proposed to eliminate these reductions to zero. The F_5 Algorithm saves computation time by removing useless critical pairs which will reduce to zero. The syzygies [5] are used to recognize useless critical pairs in advance.

There are several approaches how to use syzygies to remove useless pairs. For example the idea of [15] 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 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 still appear later. 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 to be considered, we use the example taken from [8]. 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 tf_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 have 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 [18]. Another review and implementation in Magma was done by Till Stegers [19]. Since there is no proof of termination of the F_5 Algorithm, a modification [6] has been introduced such that it always terminates.

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 [17] 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 [13]. 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 [13] 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 [16].

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 [13, 11].

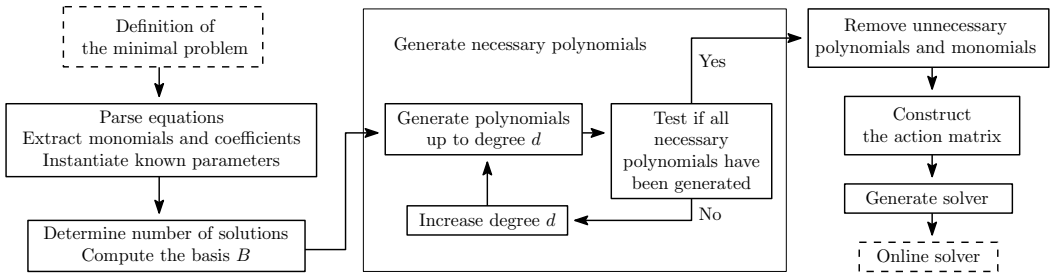


Figure 3.1. Block diagram of the automatic generator.

3.1.1. Definition of the minimal problem

Definitions of minimal problems to be solved are given in separate functions that are stored in the folder `minimalProblems`. Each of the definitions has to contain the 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, which means that we extract monomials and parameters used and obtain total degrees of the polynomials. Then, we instantiate each known parameter with a random number from $\mathbb{Z}_p$. We assign a unique identifier to each parameter used. The reason is that we need to track the parameters through the process of manipulating with the 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. Because there is no function or simple script to compute the basis in MATLAB, we have to do it by calling an external software.

The easiest 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 on older computers.

The second implemented option is to use the algebraic geometry software Macaulay2 [10]. 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 to 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 why 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 that systems of polynomials can be represented by matrices. This representation enables us to easily multiply 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 $f_i \in F$ be polynomials where F is a set of polynomials from the input. Let $maxdeg$ be the maximal total degree of all polynomials f_i. At the beginning we put all polynomials $\{mf_i \mid f_i \in F;\ \deg(mf_i) = \deg(f_i), \ldots, maxdeg\}$ into the matrix M, where m is a monomial. Then, we perform the Gauss-Jordan elimination on the matrix M and save the result as matrix $\tilde{M}$. Next 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 variable. If not, we have to add more polynomials to the matrix M. We increment $maxdeg$ by one and add all polynomials $\{mf_i \mid f_i \in F;\ \deg(mf_i) = maxdeg\}$ to the matrix M. Then, we continue with the elimination and with the checking the action matrix requirements as explained 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 *CheckActionMatrixConditions* 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: $M \leftarrow \{mf_i \mid f_i \in F;\ \deg(mf_i) = \deg(f_i), \ldots, maxdeg;\ m \text{ is a monomial}\}$
3: $\tilde{M} \leftarrow$ Reduction to a Row Echelon Form of M
4: $var \leftarrow CheckActionMatrixConditions(\tilde{M}, variables, algB)$
5: **while** $var = \emptyset$ **do**
6: $\quad maxdeg \leftarrow maxdeg + 1$
7: $\quad M \leftarrow M \cup \{mf_i \mid f_i \in F;\ \deg(mf_i) = maxdeg;\ m \text{ is a monomial}\}$
8: $\quad \tilde{M} \leftarrow$ Reduction to a Row Echelon Form of M
9: $\quad var \leftarrow CheckActionMatrixConditions(\tilde{M}, variables, algB)$
10: **end while**
11: **return** $(\tilde{M}, var)$

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

When 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 polynomial generation in the code generator module.

3.1.5. Removing unnecessary polynomials and monomials

Since the polynomials were generated systematically in the previous step, 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 polynomials which are not necessary as possible.

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 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, this heuristic is used. If we have successfully removed k polynomials, we try to remove $2k$ 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 $\tilde{M}$.

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 [11]. 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 of M_{x_k} corresponds to a monomial $b_i \in B$. Let the monomials b_i be sorted such that if $b_l \prec b_k$ then $k < l$ where $\prec$ is the monomial ordering used. We put coefficients of the polynomial $m_i = \overline{(x_k b_i)}^F$ to the i-th row 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. $x_k b_i = b_j$ for some $b_j \in B$. That means that $x_k b_i$ is irreducible by F and m_i is a monomial in B. In this case we set $M_{x_k}(i,j) = 1$ and $M_{x_k}(i,k) = 0$ where $k \neq j$.
2. $x_k b_i \neq b_j$ for all $b_j \in B$. In this case there is f such that $\mathrm{LM}(m_i) = \mathrm{LM}(f)$

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: $v \leftarrow CheckActionMatrixConditions(\tilde{M}, variable, algB)$
15: **if** $v = variable$ **then**
16: $up \leftarrow down + 1$
17: $step \leftarrow 2step$
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 $filter$

where $f \in F$ so $m_i = x_k b_i - f$. Since all monomials of f except $\mathrm{LM}(f)$ are from B, all monomials of m_i are also from B. We put coefficient of m_i at the monomial b_j on the (i, j) position in the matrix M_{x_k}.

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 preserving its structure. The current version of the automatic generator can generate solvers for MATLAB and Maple but new code generators can be easily added. It can be set in the minimal problem definition by setting `cfg.exportCode` which solvers will be generated. For instance

to create both MATLAB and Maple solvers, we set `cfg.exportCode = {'matlab' 'maple'}`.

To create a 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 used even by non-expert users and to be easily expanded or improved.

At first, the script `setpaths.m` should be executed from the root directory of the automatic generator. 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 it is explained 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 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, let us present generating of a solver for the 6-point focal length problem [3]. 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 [13] can be done. For example, if we want to generate multiple elimination solvers as suggested in [11], 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 [7], can be taken over and implemented into the automatic generator. Because we are mostly working with sparse matrices in the automatic generator, Gauss-Jordan elimination for sparse matrices can be implemented to save some computation time.

3.2.1. Reimplementation

The previous implementation [13] of the automatic generator was implemented in the 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. Our first task was to reimplement the old implementation into new version of MATLAB. All changes of the code were minor and just on the implementation level.

One important problem was that the Maple Toolbox for MATLAB in not compatible with the MATLAB Symbolic Math Toolbox anymore. One of the options in the monomials basis B module was to use the Maple toolbox to compute the basis B. Because of the new incompatibility, this option can no longer be used and the algebraic geometry software Macaulay2 [10] have to be used instead.

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 a strategy how to generate polynomials for one eliminiation solvers. However, it may be better to create multiple elimination solvers in some cases. 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.

To generate polynomials for multiple elimination solvers, we generate all polynomials up to degree d in each step and then, we 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 increasing the degree d of generated polynomials leads to a large number of new generated polynomials. The number of monomials of degree d in n unknowns is $\binom{d+n-1}{n-1}$. Therefore, we add $m\binom{d-eqdeg+n-1}{n-1}$ polynomials in one iteration where d is the total degree of new polynomials, $eqdeg$ is the total degree of given polynomials and m is number of given polynomials. This number grows rapidly when increasing d and n is large. Therefore, we want to hold d as low as possible.

Now, let us look at the process of generating polynomials in more details. The pseudocode is shown as Algorithm 11. Let the $maxdeg$ be the 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 product 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 $maxdeg$ which is the maximal total degree of all already generated polynomials. The variable $step$ tells how much we want to increase the total degree in one step. We save the $maxdeg + 1$ from the previous iteration into the variable $mindeg$ to keep the track to which degree we have generated polynomials already. We get new matrix M_2 by copying the matrix $\tilde{M}_1$ and we add all polynomials with total degrees from $mindeg$ to $maxdeg$ to M_2. These polynomials are multiples of polynomials from $\tilde{M}_1$ by 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 the iteration. That situation happens when two matrices $\tilde{M}_j$ and $\tilde{M}_{j-1}$ have the 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 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 by 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: $M_1 \leftarrow \{mf_i \mid f_i \in F;\ \deg(mf_i) = \deg(f_i), \ldots, maxdeg;\ m \text{ is a monomial}\}$
4: $\tilde{M}_1 \leftarrow$ Reduction to a Row Echelon Form of M_1
5: $var \leftarrow CheckActionMatrixConditions(\tilde{M}_1, variables, algB)$
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: $\quad\quad M_j \leftarrow \tilde{M}_{j-1} \cup \Big\{mf_i \mid f_i \in \tilde{M}_{j-1};$
$\quad\quad\quad \deg(mf_i) = mindeg, \ldots, maxdeg;\ m \text{ is a monomial}\Big\}$
12: $\quad\quad \tilde{M}_j \leftarrow$ Reduction to a Row Echelon Form of M_j
13: $\quad$ **until** number of non-zero rows of $\tilde{M}_j$ = number of non-zero rows of $\tilde{M}_{j-1}$
14: $\quad j \leftarrow j - 1$
15: $\quad var \leftarrow CheckActionMatrixConditions(\tilde{M}_j, variables, algB)$
16: **end while**
17: **return** $(\tilde{M}_j, var)$

We have to keep track about how the matrices M_j were built to be able to restore the process of generation of polynomials and generate the code of the solver in the solver generator module. 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 in $\tilde{M}_{j-1}$ 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 described 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 because there are many dependent rows in the corresponding matrices M_j. These zero rows in matrices $\tilde{M}_j$ give 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 non-zero rows as before the removal and will have 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 non-zero 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 in the automatic generator settings.

3.2.4. Matrix partitioning

In the automatic generator, we are dealing with matrices that are mostly sparse, so some efficient techniques can be used to work with them. This will often result in 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 [12]. 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 $n^3 \to k \cdot \left(\frac{n}{k}\right)^3$. 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 by computing the permutation matrices again.

Let us say we want to eliminate matrix M. First of all, we have to 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 [4] 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 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

Algorithm 12 Remove redundant polynomials

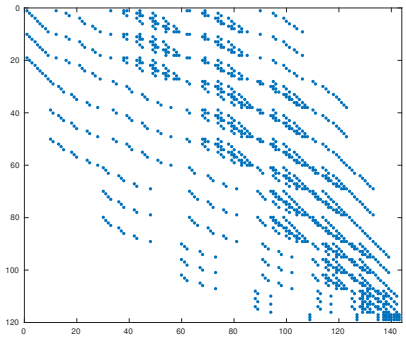
Input:
M a matrix representing a set of polynomials
$\tilde{M}$ a matrix in a Row Echelon Form representing a set of polynomials
Output:
M a matrix representing a set of polynomials
$\tilde{M}$ a matrix in a Row Echelon Form representing a set of polynomials

1: $toRemove \leftarrow$ number of zero rows of $\tilde{M}$
2: $nonZero \leftarrow$ number of non-zero rows of $\tilde{M}$
3: $rows \leftarrow$ number of rows of M
4: $step \leftarrow \max\{\lfloor toRemove/4 \rfloor, 1\}$
5: $up \leftarrow 1$
6: $filter \leftarrow \{1, 2, \ldots, rows\}$
7: **while** $toRemove \neq 0$ **do**
8: $down \leftarrow up + step - 1$
9: **if** $down > rows$ **then**
10: $down \leftarrow rows$
11: $step \leftarrow down - up + 1$
12: **end if**
13: $filter_{Old} \leftarrow filter$
14: $filter \leftarrow filter \backslash \{up, up + 1, \ldots, down\}$
15: $\tilde{M} \leftarrow$ Reduction to a Row Echelon Form of M only with rows specified by $filter$
16: **if** number of non-zero rows of $\tilde{M} < nonZero$ **then**
17: **if** $step = 1$ **then**
18: $up \leftarrow up + 1$
19: **else**
20: $step \leftarrow \max\{\lfloor step/4 \rfloor, 1\}$
21: **end if**
22: $filter \leftarrow filter_{Old}$
23: **else**
24: $toRemove \leftarrow$ number of zero rows of $\tilde{M}$
25: $up \leftarrow down + 1$
26: $step \leftarrow \min\{2step, toRemove\}$
27: **end if**
28: **end while**
29: **return** (M only with rows specified by $filter, \tilde{M}$)

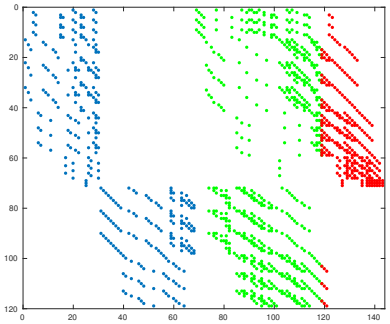
all rows of M to get the identity matrix in the left top corner. However, we get non-eliminated submatrix in the right bottom corner of M. After eliminating this submatrix, the rows above this submatrix are still not eliminated. If this is the last elimination in the solver, we have to eliminate only the rows which we need to build the action matrix from. 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 a 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

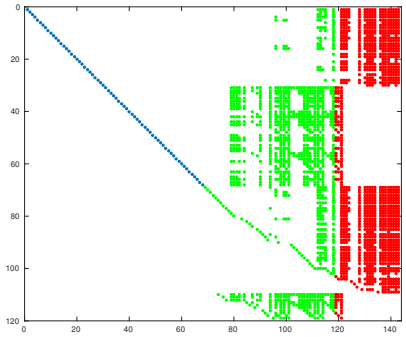
blocks M_{11} of size 71×30 and M_{22} of size 48×38 and 51 coupling columns which can not be assigned to any diagonal block. After two independent Gauss-Jordan eliminations of the blocks M_{11} and M_{22} and row permutation, we get the identity matrix in the left top corner, see Figure 3.2c. Now, we 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, uneliminated submatrix of size 68×75 lefts in the right top corner of the matrix. 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.



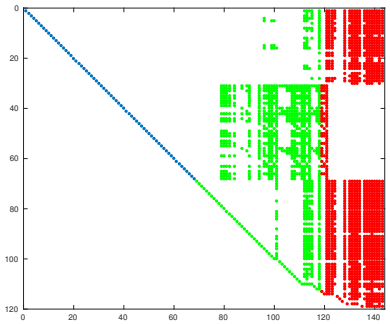
(a) The input matrix.



(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 matrix partitioning does not have to be efficient for all minimal problems, it can be easily enabled or disabled in the automatic generator. By setting `cfg.matrixPartitioning = 'all'` matrix partitioning is used to all Gauss-Jordan eliminations in the solver. If `cfg.matrixPartitioning = 'last'` is set, matrix partitioning is used only to the last Gauss-Jordan elimination in the solver. Matrix partitioning can be 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 are generated systematically in the polynomial generator module, as it is described in section 3.1.4, many of them are supefluous, and therefore need not to be generated. Since the automatic generator consists of independent modules, we can replace the polynomial generator module by some better implementation.

We have described some of the state of the art techniques how polynomial systems can be solved in the chapter 2. 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 [7] Algorithm so we have implemented this technique into the automatic generator.

Implementation in Maple

Before we could start implementing the F_4 strategy into the automatic generator, we had to deeply understand the F_4 algorithm. Therefore, we have implemented the F_4 Algorithm in Maple according to [7].

We have choosen the software Maple because the F_4 Algorithm is implemented there by J. Ch. Faugère himself in the Groebner package. Therefore, we are able to compare our implementation with the implementation by Faugère.

Our implementation of the F_4 algorithm is available at `http://cmp.felk.cvut.cz/~trutmpav/bachelor-thesis/F4` and it is divided into the same functions as described in [7]. The Gröbner basis generated by the F_4 Algorithm are not reduced Gröbner basis, and therefore we have added the reduction of Gröbner basis to get the reduced Gröbner basis at the end of the algorithm. This enables us to easily compare the results with results computed by the Faugère's implementation.

The main function is named *F4* in our implementation and it is called `F4(F, Sel, ordering)` where `F` is a set of input polynomials, `Sel` is a function which selects critical pairs as descibed in the section 2.2.6 and `ordering` is a monomial ordering. Output of this function is the reduced Gröbner basis. Our implementation prints information as the number of pairs, number of selected pairs and their total degree and sizes of matrices during the computation, see an example of the output in Figure 3.3. The implementation by Faugère can be called by `Groebner[Basis](F, ordering, method=fgb)` where the parameters `F` and `ordering` have the same meanings as the parameters in our implementation. We can force the Faugère's implementation to print some information about the process by setting `infolevel[GroebnerBasis] := 5`, see an example of the output in Figure 3.4. Therefore, we can compare, not only the results, but the sizes of the matrices, too. There are many choices in the F_4 Algorithm which can be differently implemented. For example, the list of divisors may be sorted differently in the function *Simplify*. Therefore, the sizes of the matrices may differ in our implementation and in Faugère's implementation, as you can see in the example below.

Example We show both implementations on the cyclic 4 problem taken from [7]. You can see the input and the output of our implementation in Figure 3.3 and of Faugère's implementation in Figure 3.4.

Integration into the automatic generator

Implementation of the F_4 Algorithm in the automatic generator is the same as J. Ch. Faugère has described in [7] and we did in the section 2.2. The only difference is that we need to track how the polynomials are constructed to be able to reconstruct the process in the solver generator module. In the F_4 Algorithm, to be concrete, in the function

```
> F4([a*b*c*d - 1, a*b*c + a*b*d + a*c*d + b*c*d, a*b + b*c + a*d + c*d, a + b + c + d],
     SelMinDeg, tdeg(a, b, c, d));

        0.00: |P| = 3, Selected 1 of degree 2
        0.00: Symbolic Preprocessing: |L| = 2, |Fd| = 0
        0.01: Reducing matrix of size 3x7
        0.02: Reduction finished

        0.02: |P| = 2, Selected 1 of degree 3
        0.02: Symbolic Preprocessing: |L| = 2, |Fd| = 3
        0.03: Reducing matrix of size 4x9
        0.03: Reduction finished

        0.03: |P| = 2, Selected 2 of degree 4
        0.03: Symbolic Preprocessing: |L| = 4, |Fd| = 7
        0.04: Reducing matrix of size 8x12
        0.05: Reduction finished

        0.05: |P| = 2, Selected 2 of degree 5
        0.05: Symbolic Preprocessing: |L| = 4, |Fd| = 15
        0.06: Reducing matrix of size 6x12
        0.06: Reduction finished

        0.06: |P| = 3, Selected 3 of degree 6
        0.07: Symbolic Preprocessing: |L| = 6, |Fd| = 21
        0.08: Reducing matrix of size 10x14
        0.08: Reduction finished

        0.09: |P| = 2, Selected 2 of degree 7
        0.09: Symbolic Preprocessing: |L| = 4, |Fd| = 31
        0.10: Reducing matrix of size 7x10
        0.10: Reduction finished

        0.10: Computing reduced Groebner basis
```

$$b^2 + 2bd + d^2$$
$$a + b + c + d$$
$$bc^2 + c^2d - bd^2 - d^3$$
$$bd^4 + d^5 - b - d$$
$$c^3d^2 + c^2d^3 - c - d$$
$$d^4c^2 + bc - bd + cd - 2d^2$$
$$bcd^2 + c^2d^2 - bd^3 + cd^3 - d^4 - 1$$

Figure 3.3. The input and the output of the cyclic 4 problem using our implementation of the F_4 Algorithm in Maple.

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 polynomials may be replaced by other polynomials taken from $\tilde{F}_i$. To sum up, the matrix F_i is built from

```
> infolevel[GroebnerBasis] := 5:
> Groebner[Basis]([a*b*c*d - 1, a*b*c + a*b*d + a*c*d + b*c*d, a*b + b*c + a*d + c*d,
                   a + b + c + d], tdeg(a, b, c, d), method=fgb);

-> FGb
 domain: rat_int_cof
 Set offset primes to 0/20000

 Lin Bk ignored [NEW lib]/S:0 -> 8/
 [2](3x7)100%/
 [3](5x10)100%/
 [4](8x12)100%/
 [5](6x12)100%/
 [6](11x16)100%/
 [7](8x11)0.0%/100%/

 Mingbasis2
 (7x24)100%/restore Z1 Copy 80.00 for 24/30 exposants
  {done}
  SWAP Z1/2 Memory usage (estimate):  0.000
   7 polynomials, 30 terms
    total time:         0.014 sec
    -------------------------------
```

$$
\begin{gathered}
b^2 + 2bd + d^2 \\
a + b + c + d \\
bc^2 + c^2d - bd^2 - d^3 \\
bd^4 + d^5 - b - d \\
c^3d^2 + c^2d^3 - c - d \\
d^4c^2 + bc - bd + cd - 2d^2 \\
bcd^2 + c^2d^2 - bd^3 + cd^3 - d^4 - 1
\end{gathered}
$$

Figure 3.4. The input and the output of the cyclic 4 problem using Faugère's implementation of the F_4 Algorithm in Maple.

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 the track of which polynomials from which matrices $\tilde{F}_i$ (we can look at the input polynomials as if it is a matrix $\tilde{F}_0$) are multiplied with which monomials. 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. Therefore, there will be no reduction to zero in the generated solver 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 systematic polynomial generator as described in sections 3.1.4 and 3.2.2 is used. This is done

by setting `cfg.PolynomialsGenerator = 'systematic'`. 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 the 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

In the automatic generator as presented in this thesis, there are many different methods which can be used to generate solvers. Efficiency of the generated solvers depends on the method choosen. It shows up that different methods are efficient for different minimal problems. Therefore, we need a tool which will generate more solvers for a choosen minimal problem. Each solver will be generated by a different method used. In the end, the tool will compare the generated solvers so we will be able to choose which solver we will use in applications. The tool have to compare the solver in many aspects because different applications have different requirements. For example, in one application we need solvers which are very fast, in another we may prefere solvers that are slower but more numerically stable. Therefore, we next present such a tool which we call the benchmark of the automatic generator.

3.3.1. Structure

We will briefly describe the structure of the benchmark of the automatic generator. The structure is shown in Figure 3.5 and consists of independent blocks which can be easily modified and replaced.

At the beginning, we have to load the definition of the minimal problem which we want to generate a solver for. These definitions are described in the section 3.1.1.

Next, we load the benchmark templates. A benchmark template is a set of settings of the automatic generator that will be used to generate a solver. For example, if we want to compare one elimination and multiple elimination solvers, we will have two benchmark templates. In the first one, the settings will be set to generate one elimination solver, in the second one, the generation of multiple elimination solver will be set. We store a set of benchmark templates which are related to each other in one MATLAB function. In the example above, we will have both of the templates in one function called *bench_elimination*. Another example can be, that if we would like to benchmark the polynomial generation methods, i.e. if the F_4 method is better than the systematic method, we will have two benchmark templates. The first one will set the automatic generator to generate solvers by the systematical method used. The second one will set the automatic generator to generate solvers by using the F_4 strategy. Both this templates will be in one function called *bench_polynomialGenerator*. We keep all these functions in the folder `benchmark` in the automatic generator.

When the benchmark templates are loaded, we use the automatic generator to generate the solver for each benchmark template. Then, we run each solver on each instance of parameters and we write down the time spent by the computation. These instances

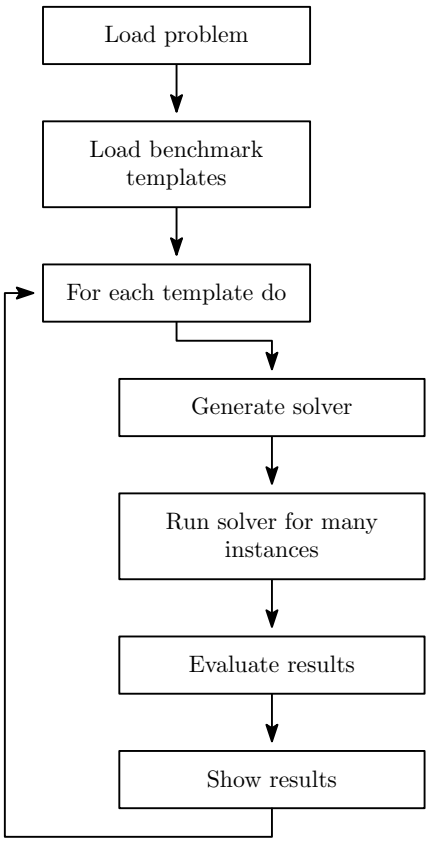


Figure 3.5. Block diagram of the benchmark of the automatic generator.

of parameters may be specified by user or if they are not provided, they are randomly generated. The random generator of parameters assign to each parameter random number from the normal distribution with the zero mean and with standard deviation equal to one.

The main part of the benchmark is the evaluation of the results. It is left to the user to implement the evaluation function. This function gets the set of instanced parameters, results from the solver and the correct results, if they are provided with the set of instanced parameters. These correct results may be precomputed in other software only to check the correctness of the results from the generated solvers. The evaluation function returns the set of errors, e.g. how much the results from the solver differ from the correct values. If the evaluation function is not provided by the user, the default evaluation function is used. This function gets the set of given polynomials, substitutes the unknowns by computed solutions and evaluates each polynomial. In the best case, we should get zero for each polynomial. But we obtain non-zero values. Absolute values of these errors depend on the numerical stability of the solver so we are comparing the numerical stability of the generated solvers.

Finally, we have to show the results in some well-arranged way. The presentation of the results is dependent on which data we want to show. Therefore, we leave the implementation of this part to the user. If the user do not specify the function which presents the results, the default presentation function is used. In this function, we get $\log_{10}$ of absolute values of the errors and show them as histogram. To be able to com-

pute the $\log_{10}$ of the errors we have to remove all zero values from the set of errors. We also print the percentage of zero errors amongs all errors.

3.3.2. Usage

The benchmark of the automatic generator is designed to be used easily, but also to be easily modified to give us the results we want to know. The benchmark of minimal problem `minimalProblem` is called by the command `gbs_Benchmark(minimalProblem, benchmark, inputData, correctOutput, validationFunction, renderFunction)` where the first two arguments are compulsory and the others are optional. The parameter `benchmark` is a function handler to the function which provides the set of benchmark templates. The `inputData` is a set of parameters on which we want to test the generated solvers. If this argument is not provided, random generated parameters are used. The `correctOutput` is a set of expected results of the solvers. These correct outputs are used to compare the numerical stability of the solvers. The function handler `validationFunction` is a handler to a function which evalutes the results and returns the set of errors. The default `validationFunction` substitutes computed solutions into the given polynomials and evalutes them. The function handler `renderFunction` is a handler to a function which shows the results. If this function is not provided, the default function is used. This default function renders histogram of $\log_{10}$ of absolute values of errors.

For clarity, we provide an example. We want to benchmark solvers for the 6-point focal length problem [3] and we are considering one elimination and multiple elimination solvers. We have no real data to test these solvers, and therefore we have no expected results. We want to use the default validation function and the default function for presenting the results of the benchmark. In this case, we call the benchmark with these parameters `gbs_Benchmark('sw6pt', @bench_elimination)`. From the shown results, we decide which solver we will use in the application.

4. Experiments

To show the usefulness of our improvements of the automatic generator, we have compared several solvers of some minimal problems using different methods in the automatic generator. We have used the benchmark tool of the automatic generator to generate the solvers and to compare the results.

We have divided the experiments into three parts. In each part, we are comparing easily comparable methods on which the speed up of the new implemented methods can be straightforwardly seen. In the first section, we are comparing one elimination solvers against multiple elimination solvers. In the second part, we are comparing solvers without the matrix partitioning to solvers with matrix partitioning used in the last elimination only and to solvers with matrix partitioning used in all eliminations in the solver. In the last section, we are comparing solvers with different strategies of polynomial generation. One solver is generated by the systematical generator while the second one is generated using the F_4 strategy.

To be able to compare the solvers, we have measured the time of computing the solutions for each set of parameters for each solver. In the tables below, we have picked up the maximal and minimal values and medians of the times for each solver. We have also written the sizes of matrices to eliminate and approximate numbers of operations for each solver to the tables. By number of operations we mean number of operations which are needed by each solver to obtain the solutions from the set of parameters. To this number we count operations required by the Gauss-Jordan eliminations only for simplicity. To eliminate a matrix of dimensions $m \times n$, we need to do $(\max\{m, n\})^3$ operations which is the upper bound of the time complexity of the Gauss-Jordan elimination. To be able to compare the numerical stability of the solvers, we have substituted each computed solution back into the given equations and wrote down the results as errors. We have removed the errors equal to zero and computed the $\log_{10}$ of absolute values of errors. We have presented these values as histograms for each solver.

We have choosen the 9-point relative pose different radial distortion problem [14] for the testing. This problem consists of four polynomial equations in four unknowns. The number of all parameters is 63. The definition of this minimal problem can be found under the name `ku9pt` in the folder `minimalProblems` in the automatic generator. To generate the solvers, we have used the default settings of the automatic generator obtained by calling of the function *gbs_InitConfig* if we do not specify differently in the description of each experiment. We have tested the generated solvers on randomly generated data which remained the same within each experiment. Each solver was tried on 1 000 instances of parameters. All test were performed on Intel Xeon E5-2630 2.30 GHz based computer. The MATLAB R2014a 64-bit was used to run the tests.

4.1. Multiple elimination solver

In this part, we are comparing one elimination solver to multiple elimination solvers. We have generated one solver according to the description in the section 3.1.4. This first solver consist only of one elimination in the end. The second and the third solvers have been generated as explained in the section 3.2.2. The second solver has been generated

with the variable *step* set to 1, and therefore an elimination is performed always when the maximal total degree of the generated polynomials is increased by 1. This second solver consist of four Gauss-Jordan eliminations. The third solver has been generated with the variable *step* set to 2. This means, that an elimination is performed when the maximal total degree of the generated polynomials is increased by 2. This solver consists of two eliminations.

We have used the benchmark templates specified in the function *bench_elimination* in the folder `benchmark` in the automatic generator. All other settings remained set to default values.

The computation times, sizes of matrices to eliminate and numbers of operations required by these solvers are shown in the Table 4.1. The numerical stability of the solvers is shown in the Figure 4.1 as histogram of $\log_{10}$ of absolute values of errors.

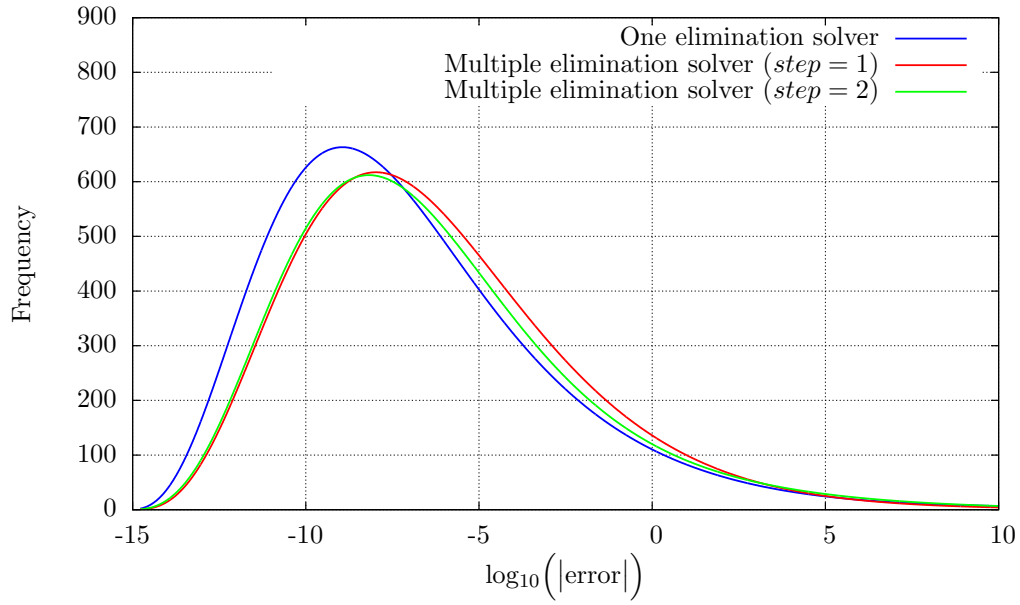


Figure 4.1. Histogram of $\log_{10}$ of absolute values of errors for one elimination and multiple elimination solvers.

You can see that the numerical stability of the multiple elimination solvers is slightly worse than the numerical stability of the one elimination solver. In the contrary, the multiple elimination solvers are approximately 1.5 times faster than the one elimination solver. Interesting is that the second and third solvers are equivalently fast, but one of them consists of four eliminations and the second one only of two eliminations. Therefore, we can not say that more eliminations lead to faster solvers. So, it is important to run benchmarks to find the optimal number of eliminations for each minimal problem and then choose the best solver for the application at hand.

4.2. Matrix partitioning

In this section, we compare the solvers using matrix partitioning, as described in the section 3.2.4, on multiple elimination solvers as described in the section 3.2.2. We have set the variable *step* to one so the generated solvers consist of four eliminations. The first solver has been generated without matrix partitioning. In the second one, matrix partitioning was used in the last elimination only and the third solver has been

generated with matrix partitioning in all four eliminations.

The benchmark templates used for this comparison are specified in the function *bench_matrixPartitioning* in the folder `benchmark` in the automatic generator. The variable *step* was set to 1 and all other settings remained set to default values.

The computation times, sizes of matrices to eliminate and numbers of operations required by these tree solvers are in the Table 4.2 and the numerical stability is shown as histogram of $\log_{10}$ of absolute values of errors in the Figure 4.2.

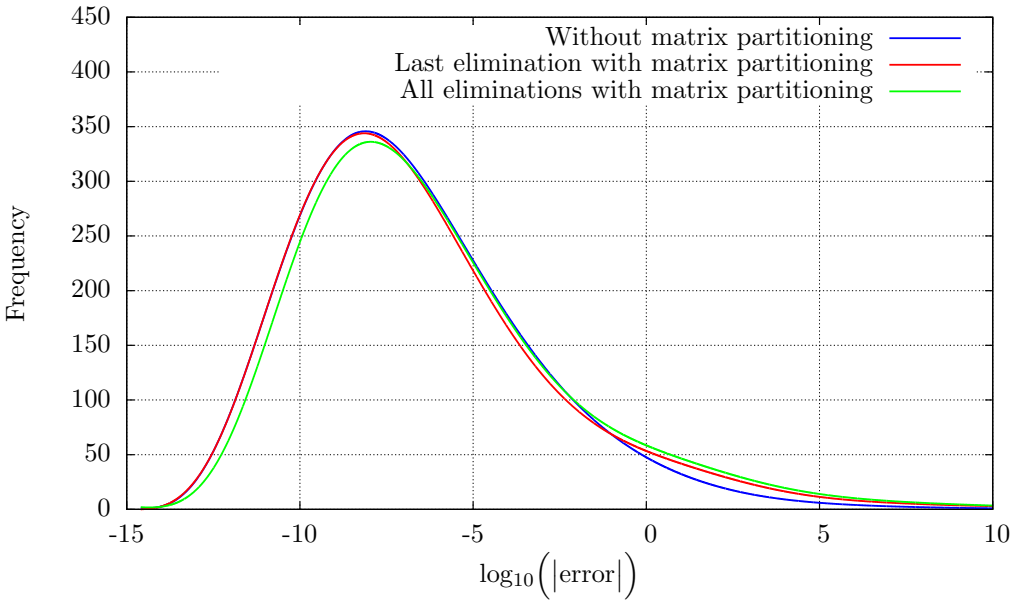


Figure 4.2. Histogram of $\log_{10}$ of absolute values of errors for solver without matrix partitioning, for solver with matrix partitioning in the last elimination only and for solver with matrix partitioning in all eliminations.

We can see that the numerical stability remained practically the same for all three solvers. The solver with matrix partitioning in the last elimination is approximately twice faster than the solver without matrix paritioning. There is no big difference between the solver with matrix partitioning in the last elimination and the solver with matrix partitioning in all eliminations in the solver. The reason is, that when eliminating any other than the last matrix in the solver, we have to eliminate all the coupling columns. We do not have to do this when it is the last elimination. Therefore, the speed up of the elimination of other than the last matrix is smaller than the speed up of the last elimination.

4.3. F_4 strategy

At last, we have compared the solver generated by the systematical polynomial generator with the solver generated with the F_4 strategy. The first solver has been generated according to the decription in the section 3.1.4 using the systematical polynomial generator. The second one uses the F_4 strategy described in the section 3.2.5.

We have used the benchmark templates that are defined in the function *bench_polynomialGenerator*, which is stored in the folder `benchmark` in the automatic generator. All other settings remained set to default values.

The numerical stabilities of both solvers are compared in the Figure 4.3 as histogram

of $\log_{10}$ of absolute values of errors. The computation times, sizes of matrices to eliminate and numbers of operations required by these solvers are in the Table 4.3.

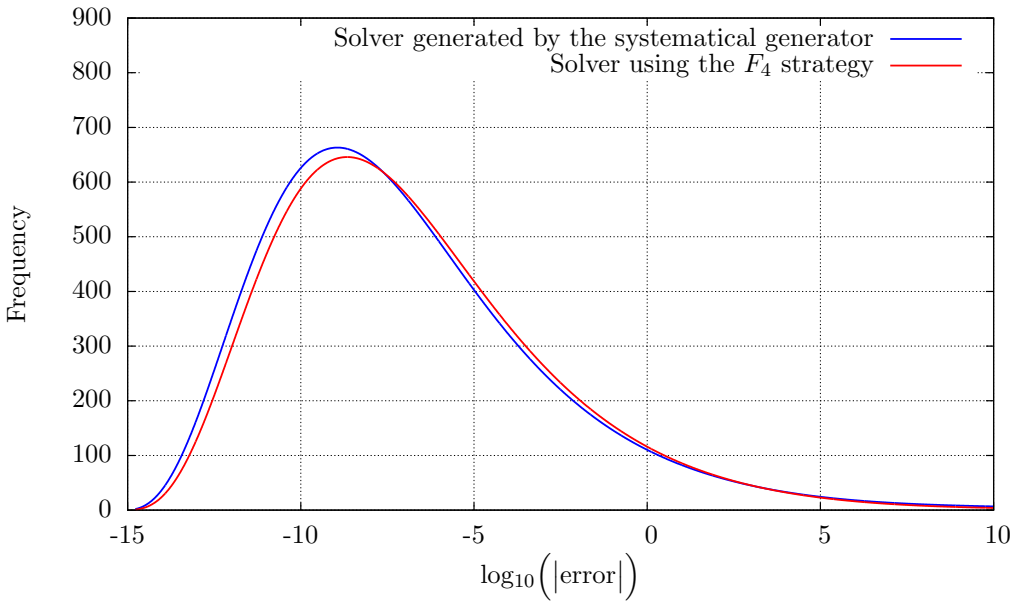


Figure 4.3. Histogram of $\log_{10}$ of absolute values of errors for solver generated by the systematical generator and for solver using the F_4 strategy.

From the results, we can see that the numerical stability has remained the same for both solvers. The solver which uses the F_4 strategy is about 4 times faster than the solver generated by the systematical polynomial generator.

	One elimination solver	**Multiple elimination solver** ($step = 1$)	**Multiple elimination solver** ($step = 2$)
minimal time	2.159 s	1.044 s	1.060 s
median of times	2.930 s	2.097 s	2.015 s
maximal time	3.129 s	2.301 s	2.444 s
sizes of matrices	185×209	$64 \times 104; 80 \times 119$ $95 \times 125; 49 \times 73$	133×181 81×105
number of operations	9 129 329	5 152 165	7 087 366

Table 4.1. Computing times of one and multiple elimination solvers.

	Without matrix partitioning	**Matrix partitioning in the last elimination**	**Matrix paritioning in all eliminations**
minimal time	1.353 s	0.956 s	0.794 s
median of times	2.185 s	1.071 s	0.932 s
maximal time	2.604 s	1.189 s	2.273 s
1^{st} matrix	64×104	64×104	$30 \times 47; 34 \times 44; 14 \times 35; 50 \times 35$
2^{nd} matrix	80×119	80×119	$41 \times 48; 39 \times 49; 5 \times 29; 75 \times 29$
3^{rd} matrix	95×125	95×125	$50 \times 24; 45 \times 46; 32 \times 56; 63 \times 56$
4^{th} matrix	49×73	$29 \times 18; 20 \times 15; 16 \times 40; 0 \times 0$	$29 \times 18; 20 \times 15; 16 \times 40; 0 \times 0$
number of operations	5 152 165	4 859 537	1 775 775

Table 4.2. Computing times of solver without matrix partitioning, of solver with matrix partitioning in the last elimination only and of solver with matrix partitioning in all eliminations.

	Systematical generator used	F_4 **strategy used**
minimal time	2.213 s	0.615 s
median of times	2.895 s	0.713 s
maximal time	3.320 s	1.161 s
sizes of matrices	185×209	$2 \times 12; 13 \times 30$ $22 \times 46; 0 \times 0$ $52 \times 85; 36 \times 65$ $37 \times 62; 68 \times 92$ $44 \times 68; 0 \times 0$ $0 \times 0; 15 \times 39$
number of operations	9 129 329	2 405 581

Table 4.3. Computing times of solver generated by the systematical generator and of the solver using the F_4 strategy.

5. Conclusion

In this work, we have focused on how to solve systems of polynomial equations fast and how to automatically generate efficient solvers for these systems.

In the first part, we have reviewed the state of the art algorithms for computing Gröbner basis of polynomial systems. We have described the Buchberger Algorithm [2], then we have explained the F_4 Algorithm [7] in details and in the end, we have pointed out the main features of the F_5 Algorithm [8].

In the second part, the automatic generator [13] of minimal problem solvers has been presented. This tool enables us to easily generate solvers for systems of polynomial equations which arise when solving minimal problems in computer vision. We have described the process of generation of solvers in detail. Then, we have suggested several improvements of the automatic generator and we have implemented them. For example, we have presented an improvement which allows us to generate multiple elimination solvers, which are usually better for systems of polynomial equations in many unknowns. We have also shown that the solvers can be sped up when Gauss-Jordan elimination for sparse matrices is used. Next, we have taken over a strategy from the F_4 Algorithm [7] and we have implemented it into the automatic generator. For better understanding, we have implemented the F_4 Algorithm [7] in Maple first. The description of this implementation is provided in this section, too. In the end, we have presented the benchmark of the automatic generator. This tool helps us to decide which generated solver is better for our application.

In the end, we have taken the 9-point relative pose different radial distortion problem [14] and compared solvers generated with different methods for this problem on set of randomly generated data. We have shown that solvers generated with the new implemented methods may be faster than solvers generated by the old implementation of the automatic generator. We have noticed the most visible speed up when the F_4 strategy was used. In this case, the solver using the F_4 strategy is four times faster than the solver generated by the systematical generator for the 9-point relative pose problem [14].

A. Contents of the enclosed CD

- `/`
  - `AutomaticGenerator` ..... folder with the automatic generator
    - `benchmark` ..... folder contaning benchmark templates
      - `bench_elimination.m` ..... definitions of benchmark templates that are used in the experiment 4.1
      - `bench_matrixPartitioning.m` ..... definitions of benchmark templates that are used in the experiment 4.2
      - `bench_polynomialGenerator.m` .... definitions of benchmark templates that are used in the experiment 4.3
    - `gbsMacaulay` ..... folder with auxiliary files required for calling Macaulay2 [10]
      - ⋮
    - `generator` ..... folder with main functions of the automatic generator
      - ⋮
    - `minimalProblems` ..... folder with minimal problem definitions
      - `ku9pt.m` ..... definition of the 9-point relative pose different radial distortion problem [14]
      - `sw5pt.m` ..... definition of the 5-point relative pose problem [20]
      - `sw6pt.m` ..... definition of the 6-point focal length problem [3]
      - `test.m` ..... definition of the test problem (two equations in two unknowns)
    - `prerequisites` ..... folder with installation files of Macaulay2 [10] and PaToH [4]
      - ⋮
    - `solvers` ..... folder where generated solvers are stored
    - `installation.txt` ..... installation how to
    - `license.txt` ..... license file
    - `setpahs.m` ..... script which add all required paths to the MATLAB environment
  - `F4`
    - `F4.mw` ..... implementation of the F_4 Algorithm [7] in Maple
  - `thesis`
    - `thesis.pdf` ..... digital copy of this thesis

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. 6, 7, 8

[2] Bruno Buchberger. *Ein Algorithmus zum Auffinden der Basiselemente des Restklassenringes nach einem nulldimensionalen Polynomideal*. PhD thesis, Mathematical Institute, University of Innsbruck, Austria, 1965. 7, 39

[3] Martin Byröd, Klas Josephson, and Kalle Åström. Improving numerical accuracy of grobner basis polynomial equation solvers. In *Eleventh IEEE International Conference on Computer Vision*, ICCV 2007, October 14–20 2007. 21, 32, 40

[4] Umit V. Catalyurek and Cevdet Aykanat. Patoh: Partitioning tool for hypergraphs, 2011. Version 3.2. 5, 24, 40

[5] 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. 6, 7, 14

[6] 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. 15

[7] 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. 6, 9, 11, 21, 27, 39, 40

[8] 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. 6, 14, 15, 39

[9] Rüdiger Gebauer and Hans-Michael Möller. On an installation of buchberger's algorithm. *Journal of Symbolic Computation*, 6(2–3):275–286, October 1988. 8, 12

[10] 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]. 17, 22, 40

[11] Zuzana Kukelova. *Algebraic Methods in Computer Vision*. PhD thesis, Department of Cybernetics, Faculty of Electrical Engineering, Czech Technical University in Prague, February 2013. 16, 19, 21

[12] 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. 5, 24

[13] 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. 5, 16, 21, 39

[14] Zuzana Kukelova, Martin Byröd, Klas Josephson, Tomas Pajdla, and Kalle Åström. Fast and robust numerical solutions to minimal problems for cameras with radial distortion. *Computer Vision and Image Understanding*, 114(2):234–244, February 2010. 33, 39, 40

[15] 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. 14

[16] Tomas Pajdla, Zuzana Kukelova, and Martin Bujnak. Automatic generator. http://cmp.felk.cvut.cz/minimal/automatic_generator.php. [Online; accessed 2015-04-28]. 16

[17] Tomas Pajdla, Zuzana Kukelova, and Martin Bujnak. Minimal problems in computer vision. http://cmp.felk.cvut.cz/minimal/. [Online; accessed 2015-04-28]. 5, 16

[18] A. J. M. Segers. Algebraic attacks from a gröbner basis perspective. Master's thesis, Department of Mathematics and Computing Science, Technische Universiteit Eindhoven, 2004. 15

[19] Till Stegers. Faugère's f5 algorithm revisited. Master's thesis, Department Of Mathematics, Technische Universit ät Darmstadt, revisited version 2007. 15

[20] Henrik Stewenius, Christopher Engels, and David Nister. Recent developments on direct relative orientation. *ISPRS Journal of Photogrammetry and Remote Sensing*, 60(4):284–294, May 2006. 40