Development of symbolic computation algorithms & their applications in kinematics

A Project Report

submitted by

K. VIKRANTH REDDY

in partial fulfilment of the requirements
for the award of the degree of

MASTER OF TECHNOLOGY
(Automotive Engineering)
and
BACHELOR OF TECHNOLOGY
(Engineering Design)
Under the guidance of
Dr. Sandipan Bandyopadhyay

DEPARTMENT OF ENGINEERING DESIGN
INDIAN INSTITUTE OF TECHNOLOGY MADRAS.

May 2015

THESIS CERTIFICATE

This is to certify that the report titled **Development of symbolic computation algorithms & their applications in kinematics**, submitted by **K. Vikranth Reddy**, to the Indian Institute of Technology, Madras, for the award of the degree of **Dual Degree**, is a bona fide record of the research work done by him under my supervision. The contents of this report, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Dr. Sandipan Bandyopadhyay
Research Guide
Assistant Professor
Dept. of Engineering Design
IIT Madras, 600 036

Prof. Srikanth Vedantam
Head of the Department (incharge)
Professor
Dept. of Engineering Design
IIT Madras, 600 036

Place: Chennai

Date: 22nd May 2015

ACKNOWLEDGEMENTS

I am extremely grateful to Dr. Sandipan Bandyopadhyay for his patient guidance throughout the period of one year. I am extremely indebted to him for providing this life-transforming opportunity and guiding me to work in the fields of robotics and computer algebra systems for a period of 2 years. Without his valuable insights, this work would have not only possible, but also inconceivable. I would like to thank all the professors in this department and department of Mathematics for helping me to learn several things in my life and their continuous support.

It only because of my parents and grand parents, I have learnt so much in life. I am thankful to them and my sister Deepika for the support and guidance that has been continuously moulding me.

I thank all my close friends with whom I had *gen* discussions and arguments over the years. My wing mates Rahul, Saradhi, Yeshwanth and Jagadeesh with whom I share the most rapport with, have been the most helpful and able. I whole-heartedly thank them for all the fun and support. I would like to thank all of my classmates and my schoolmates Ashok, Sainadh, Asif, Bala, Srikar and Mohit who have been really helpful.

I would like to thank my labmates Murali, Saurav, Anirban, Rohit, Edison and Aditya for all the help which they had given me during the course of the project. Thanks to Mr. Jagannath Raju of Systemantics India Pvt. Ltd. for offering me a great position in his company . I would like to thank Mrs. Nagasri for making me believe that I could enter IIT. Lastly, I would like to thank Govt. of India for having faith in the student community at IIT Madras and for the concessions provided purely for the education of its citizens.

ABSTRACT

KEYWORDS: Symbolic computations, Computer Algebra Systems(CAS), Expression simplification, Parallelization, Resultants.

This report presents a work on a developing some symbolic computation algorithms which can be implemented in any Computer Algebra System(CAS). Symbolic computation may be seen as working with expression trees representing mathematical formulae and applying various rules to transform them. Computer algebra may be seen as developing constructive algorithms to compute algebraic quantities in various arithmetic domains, possibly involving indeterminates. Symbolic computation allows a wider range of expression, while computer algebra admits greater algorithmic precision (Watt, 2006). The research challenges in symbolic computation at the close of the twentieth century are formidable (Kaltofen, 2000).

Simplification of large symbolic expressions is the topic that is described, initially. Modifying a given expression into its monomial-based canonical form and its subsequent simplification is the main motive behind this algorithm. A aspect for parallelization is detected in this algorithm resulting in a more efficient algorithm which performs simplification of an algebraic expression in parallel over available processors. In the next chapter, two algorithms for addition and multiplication of polynomials namely, `PolyAdd` and `PolyMult` respectively, are presented. The objective behind developing these algorithms is to find the output in polynomial form with grouping of like terms taking place inside the routine itself. These algorithms differ from those that are used in current CASs in the sense that, given any two polynomials, the result of their addition or multiplication is again a polynomial with all the coefficients simplified and without any extraneous factors or sub-expressions.

Finally, methodologies for solving system of polynomials equations are presented. Some preliminaries on finding resultants of polynomials and determinants of matrices with symbolic entries are described. Later, two algorithms `NDet` and `PolyRes` are described briefly which are used to compute determinants of large symbolic matrices and

resultants of polynomials in several unknowns, respectively. A comparative study has been made to show the improved efficiency of `NDet` in calculating determinants over current determinant finding routines. Following that, the applications of `PolyRes` routine in finding maximal singularity-free region of a 6-DoF spatial manipulator around a given centre point in $\mathbb{R}^3$ is explained and results are enumerated.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

CAS	Computer Algebra System
SPM	Stewart Platform Manipulator
RSPM	Regular Stewart Platform Manipulator
SRSPM	Semi-Regular Stewart Platform Manipulator
MCF	Monomial-based Canonical Form
RAM	Random Access Memory

NOTATION

(c_1, c_2, c_3)	Rodrigue's parameters
I_k	k^{th} invariant of input matrix
I_1	First invariant, i.e., trace of the matrix
(x_0, y_0, z_0)	Given centre-point of the maximal-sphere in $\mathbb{R}^3$
(θ, α, β)	Ball parameters that define the orientation of the top platform of SRSPM
r	Radius of maximal singularity free region centred at (x_0, y_0, z_0)
r^s_{max}	Radius of maximal singularity free sphere centred at (x_0, y_0, z_0)
r^c_{max}	Radius of the base circle of maximal singularity free cylinder centred at (x_0, y_0)

CHAPTER 1

INTRODUCTION

1.1 Introduction to symbolic computations

Symbolic computation is the science of computation on symbols representing mathematical objects, which can be numbers, functions, polynomials, groups, ideals and matrices (Boyle and Caviness (1990), Watt (2006), Cohen (2003)). Software that perform symbolic calculations is called a Computer Algebra Systems(CAS). Notable results have been achieved in symbolic computation over the last two decades. Algorithms have been discovered for integration in finite terms and for computing closed form solutions of differential equations. Fast algorithms have been devised for factoring polynomials and computing greatest common divisors(Boyle and Caviness (1990)). Powerful interactive systems for doing symbolic computation have been designed and built, e.g., Mathematica, Maple, Macsyma etc. The software has improved the productivity of scientists and engineers; it has made possible the solution of problems that were previously intractable (Watt (2006), Mekwi (2004)).

In robotics, the inverse kinematics problem mathematically reduces to the solution of systems of multivariate algebraic equations. The exact structure of these systems depends on the type of robot considered. Symbolic computation has been used in the simulation of the dynamics of a robot system. Other areas of application of symbolic computations include High-energy Physics, Celestial mechanics, Group theory, Geometric modelling, Signal processing, Control theory, Algebraic Geometry and Number theory (Boyle and Caviness, 1990).

The advantages of symbolic computations are that automation and computational flexibility can be attained in mathematical problem solving and typically, the computations are exact, in contrast to most numeric calculations where computations use approximate floating point arithmetic. Some issues with such algorithms are that unlike in numeric computation, it is difficult to estimate the time and memory requirements for symbolic computation and most importantly, the success of a symbolic computation

depends critically on the size of intermediate expressions generated during the computation (Boyle and Caviness, 1990).

Symbolic computation algorithms have proven useful in a wide range of applications from formal verification to computer algebra. However, the time required by these algorithms restricts the size of problems which can be solved. There are two ways of obtaining faster symbolic algorithms. The first one is to use new insight into the problem domain to improve the algorithm. The second way is to use parallelization (Marco and Martinez, 2004).

1.2 Motivation

In computer algebra, many problems of moderate size become intractable because of the time and the memory required for their solution. This is why reducing time and memory requirements as much as possible is an essential point in developing computer algebra algorithms. Hence, an attempt has been made in developing some algorithms which can be used as back-end tools in current CASs for expression simplification, performing arithmetic over polynomials and solving systems of polynomial equations.

1.3 Objective

The main objective of this report is to provide some tools for manipulating symbolic expressions and use them in solving complex equations with several algebraic and trigonometric terms. Several of the intermediate objectives that are achieved to realise the final objective are:

- To simplify large-scale symbolic expressions employing a method that is not subjective, but deterministic.
- To develop a methodology for performing operations on polynomials.
- To develop a determinant finding routine which can handle symbolic matrices of higher orders.
- To solve system of multi-variate polynomial equations employing the previous methodologies developed.

1.4 Organization of the report

This report begins by giving an introduction to the symbolic computations and their applications, followed by the motivation to work on developing algorithms to tackle some of the useful problems in kinematics and robotics. Chapter 2 discusses simplification methodologies that are present currently in CASs, followed by a novel parallel simplification scheme, its applications and some of the results achieved by its implementation. In Chapter 3, algorithms developed for polynomial addition and multiplication are discussed. The use of these, along with two determinant finding algorithms is discussed in Chapter 4. This chapter ends with a brief discussion on the applications of multi-variate polynomial solving algorithms and results that are achieved in finding the determinant of a symbolic matrix and maximal singularity free zone of a 6-DoF spatial parallel manipulator.

CHAPTER 2

SIMPLIFICATION OF SYMBOLIC EXPRESSIONS

2.1 Introduction

A common impediment for many researchers engaged in computational mathematics is that the result of a symbolic operation, is a very long, complicated symbolic expression, which although technically correct, is not very helpful; only later to discover, often indirectly, i.e., by substituting random numerals for the variables, that in fact the complicated expression simplifies further to something much more elegant and useful (Bailey *et al.*, 2014).

In CASs, simplifications are normally done through rewriting rules. The new rules are obtained by applying several algebraic and trigonometric simplification transformations (Cohen, 2003). The simplest of rewriting consists of rules that always reduce the size of the expression, like $a - a = 0$ or $\sin(0) = 0$. Other rewriting rules sometimes increase, and at other times decrease the size of the expressions to which they are applied. The problem is to recognize quickly the *similar* terms in order to combine or to cancel them. In fact, the method for finding like terms, consisting of testing every pair of terms, is too costly for being practicable with very long sums and products.

The central question to consider when designing a simplification algorithm is: what does it mean for an expression to be simpler than another? For example, consider the following expression:

$$(x + 1)^4 - x^4$$

for which, the simplified form can be the factorized form:

$$(2x^2 + 2x + 1)(2x + 1)$$

or the expanded form in which the leading term x^4 cancelled out:

$$4x^3 + 6x^2 + 4x + 1.$$

In (Carette, 2004) and (Cohen, 2003), the authors tried to give a formal definition of the concept of simplification for general expressions in the context of CASs. The way in which the simplification transformations are used in *simplification functions* of some of the currently available CASs is described in Sec 2.2. In our work, two of the most commonly used simplification routines are `Mathematica`'s (Wolfram Research, 2014) `Simplify`, and `FullSimplify`. Empirically, these routines tend to run on a very large memory and become so slow that they sometimes do not return results for over a day or ever to simplify expressions that arose in works such as (Bandyopadhyay and Ghosal, 2006) and (Kodati and Bandyopadhyay, 2013).

In these works, the intermediate expressions are polynomials in several variables with complex algebraic and trigonometric terms. These polynomials are simplified with a novel scheme called monomial-based simplification which is explained in Section 2.3. Later, in Section2.4, parallelization of this algorithm and the process of *parallel simplification* over shared memory multi-processors is described. Finally, in Section. 2.5 some of the significant results achieved by this scheme are presented.

2.2 Existing simplification routines

- Macsyma (Martin and Fateman, 1971) sorts the operands of sums and products with a comparison function that is designed to order the like terms in consecutive places, and thus are easily detected. Some transformation rules are applied on these terms to simplify the expression.
- In Maple (Heck, 2003), the hash function is designed for generating collisions when like terms are entered, and combine them as soon as they are introduced. This design of the hash function also allows the recognition of expressions or subexpressions that appear repeatedly in a computation, and it helps to store them only once. Not only this saves some memory, it also speeds up computation, by avoiding repetition of same operations on several identical expressions.
- In Mathematica, the `Simplify` command tries expanding, factoring, and performing many other transformations on expressions, keeping track of the simplest form obtained. The user can choose the `ComplexityFunction`, which is a heuristic to rank the complexity of different forms of an expression and simplification is done based on the chosen method.

Consider the following expression:

$$\begin{aligned} p(x,y) =& a_1a_2x^4 + a_1b_2x^3y + a_1c_2x^2y^2 + a_1d_2x^2 + a_2b_1x^3y + a_2c_1x^2y^2 \\ &+ a_2d_1x^2 + b_1b_2x^2y^2 + b_1c_2xy^3 + b_1d_2xy + b_2c_1xy^3 + b_2d_1xy \\ &+ c_1c_2y^4 + c_1d_2y^2 + c_2d_1y^2 + d_1d_2. \end{aligned} \tag{2.1}$$

Using `LeafCount`[1] as the heuristic for `ComplexityFunction`, `Mathematica` returns the simplified expression as:

$$p(x,y) = (a_1x^2 + y(b_1x + c_1y) + d_1)(a_2x^2 + y(b_2x + c_2y) + d_2), \tag{2.2}$$

whereas, with `ByteCount`[2] as the heuristic, the output is:

$$p(x,y) = (a_1x^2 + b_1xy + c_1y^2 + d_1)(a_2x^2 + b_2xy + c_2y^2 + d_2). \tag{2.3}$$

This shows that the definition of *simplified* form of an expression in the domain of symbolic computations is subjective.

To overcome such problems associated with heuristic simplification schemes, a deterministic scheme of simplification for a class of expressions involving both trigonometric and algebraic terms is developed and is explained below.

2.3 Monomial-based simplification

A monomial is defined as a mathematical entity which can be a number, a variable or a product of a number and a variable where all the exponents are whole numbers.
E.g.: $-7, x, 2x^2, 3a^2b^3$ etc.

A polynomial is a finite sum of monomials. For example, a polynomial in two variables (namely, x and y) can be written as:

$$p(x,y) = \sum_{i=1}^{k} a_i x^{m_i} y^{n_i},$$

where, each $a_i \in \mathbb{C}$, and, $m_i, n_i \in \mathbb{Z}^+$.

[1] `LeafCount` of an expression gives the total number of corresponding indivisible sub-expressions.

[2] `ByteCount` of an expression gives the number of Bytes used internally by `Mathematica` to store the expression.

- A polynomial in a single variable is called a 'univariate polynomial' and a polynomial in more than one variable is called a 'multivariate polynomial'
- The polynomial expression 'E' in the variables $(x_1, x_2, x_3, \ldots x_n)$ can be decomposed from monomial form as:

$$\mathrm{E} = \boldsymbol{D} \cdot \boldsymbol{q}, \tag{2.4}$$

where, $\boldsymbol{q}$ is the vector containing the power products, and, $\boldsymbol{D}$ is the vector containing the corresponding coefficients.

For the simplification process, we express the polynomial as a sum of monomials in its variables by retaining the non-zero coefficients of individual power-products in explicit form. In analogy with the above equation, the simplification scheme can be written as (Bandyopadhyay and Ghosal, 2006):

$$\text{Simplify(E)} = \text{Simplify}(\boldsymbol{D}) \cdot \boldsymbol{q},$$

The advantages of this simplification method are:

1) Trivial zeroes of the expression can be easily identified, i.e., if we have an equation of the form E = 0, by converting E to monomial-based canonical form as per Eq. (2.4), we get the power products in the vector $\boldsymbol{q}$. The GCD of the power products if any, corresponds to trivial zeroes of E. Cancelling off the GCD from $\boldsymbol{q}$, these trivial zeroes can be eliminated.

2) Any pattern in power products can be identified. It is possible for the power-products to show patterns, which can be identified from the vector $\boldsymbol{q}$. For example, certain variable may occur only in its even powers in $\boldsymbol{q}$. Therefore, the square (or some higher even power) of the variable can be replaced by another symbol, reducing the degree of the resulting equation.

2.4 Parallel simplification on shared memory multi-processors

Many users utilize CASs as tools performing 'small scale' mathematical calculations that would be tedious and error-prone when performed by hand. For such calculations, systems running on a single processor computer are quite satisfactory. There exist some problems whose solutions may involve, for instance, large scale symbolic computations

requiring a significant amount of computational resources, or a combination of symbolic and numerical computations used in the context of multivariable simulation (the CAS environment becomes an *interface* to a set of computational kernels) (Petcu *et al.*, 2006). However, users often encounter the obvious limitations of a single-processor system: processor speed and available memory. Hence, availability of parallel and distributed versions of CASs can be truly beneficial in handling such large problems.

Decomposing a large complicated expression into smaller blocks and parallelizing their subsequent simplification is the main motive behind *parallel simplification*. The implementation of symbolic simplification algorithm on shared multiprocessors using Mathematica is shown schematically in Fig. 2.1.

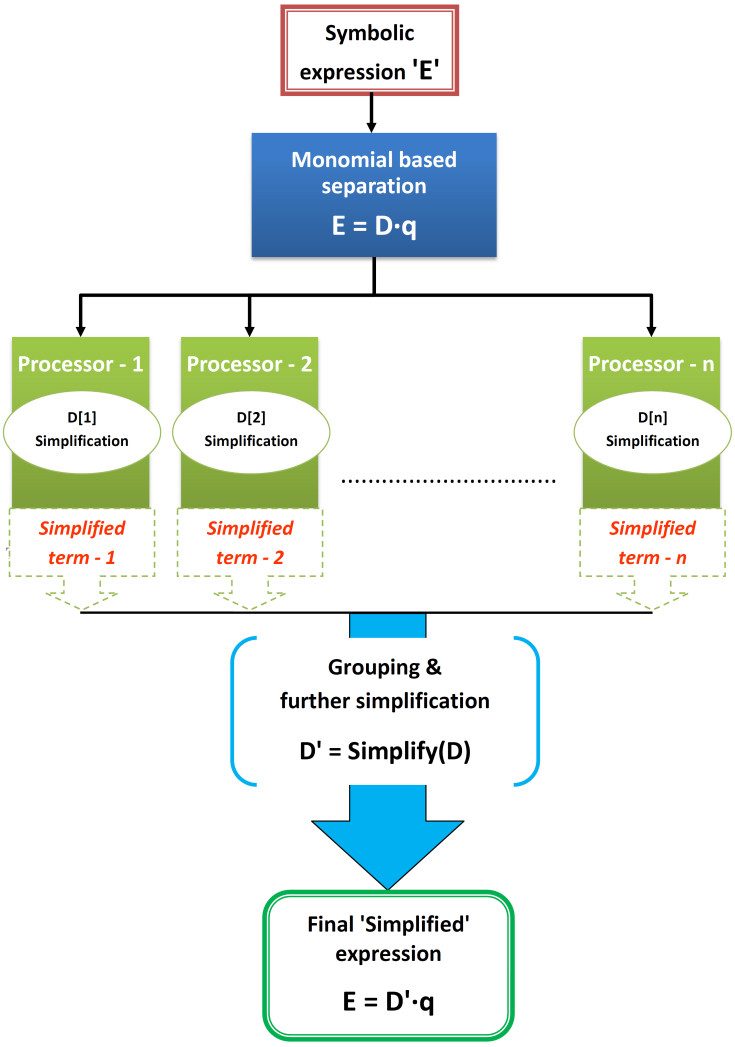

Figure 2.1: Schematic showing the monomial-based parallel simplification

Two examples are presented subsequently, showcasing the effectiveness in parallelizing the code. The first example shows an efficient way of polynomial simplification

in finding the singularity expressions of a 6-DoF Stewart-Gough platform (Bandyopadhyay and Ghosal, 2006), which happens to be a cubic surface in $\mathbb{R}^3$. In the next example, simplification of the expressions representing the forward kinematics of double wishbone suspension system for generic road and steering inputs involving both algebraic and trigonometric terms (Kodati and Bandyopadhyay, 2013) is described.

The most significant result for the above-mentioned applications is that firstly, we were able to attempt to solve such problems which happen to be computationally challenging, and as well achieve clock speed-ups between factors of 2 and 10 over the sequential versions of simplifications by employing parallelization over available processors.

2.5 Applications and results

Employing the parallel simplification methodology, significant improvements have been achieved in simplifying: a) forward kinematic expressions of suspension systems, and b) singularity manifold expression of a Semi Regular Stewart Platform Manipulator (SRSPM). The results are presented below.

2.5.1 Forward kinematics of suspension systems

Madhu et al.,(2013) presented a methodology for solving kinematics of a double-wishbone suspension system, one of the most common independent suspension systems used in passenger/race cars. The suspension is modelled as a combination of a four-bar and a five-bar linkage, and the loop-closure equations are derived, solving which, the location and orientation of wheel-hub in space are found for a given set of road and steering inputs.

The orientation of the king-pin axis is represented using Rodrigue's parameters (c_1, c_2, c_3) (see e.g., (Selig, 1996)) which are the unknown variables in the system of three polynomial equations $\{f_1(c_1, c_2, c_3), f_2(c_1, c_2, c_3), f_3(c_1, c_2, c_3)\}^\top = \mathbf{0}$ that represent the forward kinematics of the suspension system. The coefficients of these polynomials are made up of 26 architecture parameters of the suspension. These expressions

Table 2.1: Sizes of the expressions $\{f_1, f_2, f_3\}$ before and after simplification

Expression	Size in KB. before simplification	Size in KB. after simplification
$f_1(c_1, c_2, c_3)$	79.216	11.164
$f_2(c_1, c_2, c_3)$	114.392	6.461
$f_3(c_1, c_2, c_3)$	77.856	29.062

are simplified using `Mathematica`[3], to their monomial-based canonical forms. The simplification is carried out in parallel, by launching 12 computation kernels, in each of the available CPUs. The sizes[4] of these expressions before and after simplification are given in Table. 2.1.

However, the important point to note is that the time taken for serial simplification (using the in-built `Simplify` command) is 45.46 sec., whereas, it took about 0.46 sec. for the parallel simplification, i.e., a clock speed-up factor of around 100 is achieved with parallelization.

2.5.2 Singularity manifold expression of a Semi Regular Stewart Platform Manipulator

One of the best known parallel manipulators is the Stewart, or the Stewart-Gough Platform. The general form of the Stewart Platform manipulator(SPM) consists of a mobile platform and a fixed base. These are connected by six extensible links, often referred to as the "legs", through kinematic pairs which are equivalent to a SPS (Spherical Prismatic and Spherical) chain. The prismatic joints, usually actuated by hydraulic jacks, are used to vary the lengths of the legs.

In many of the SPMs the 6 joints of the base and the end-effector lie on a circle, and they are called Regular Stewart Platform Manipulators (RSPMs). In a RSPM, if the lengths of the alternate sides are equal for both the base and the end-effector, then it is called a Semi Regular Stewart Platform (SRSPM). The solid model of a SRSPM is shown in Fig. 2.2.

Finding the singularity manifold expression in symbolic form for a Stewart platform

[3]With Mathematica version 9 for 64 bit Linux on a 6-core Intel Core i7-4930K CPU @ 3.40 GHz clock speed and 64 GB of RAM.

[4]The "size", in this context, refers to the amount of *computer memory* needed to represent/store an expression in Mathematica's internal format.

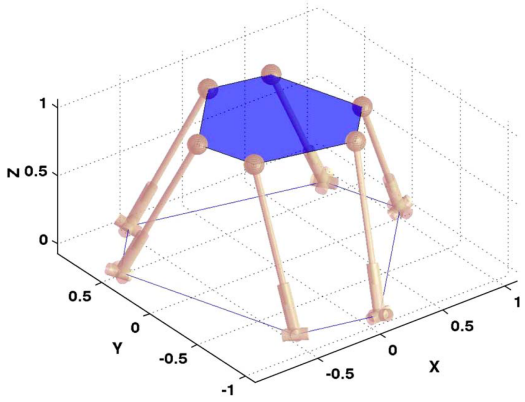

Figure 2.2: Solid model of the SRSPM (adapted from Bandyopadhyay and Ghosal (2006))

is a classical problem in computational kinematics. The expression is obtained by finding the determinant of a 6×6 wrench transformation matrix ($\boldsymbol{H}$) for the top platform (see, (Bandyopadhyay and Ghosal, 2006)), the size of which turns out to be 3.35 MB. The singularity manifold corresponds to a surface in $\mathbb{R}^3$ for a fixed orientation of the top-platform and is a polynomial in Cartesian variables (x, y, z). The simplification scheme for this expression is shown schematically in Fig. 2.3.

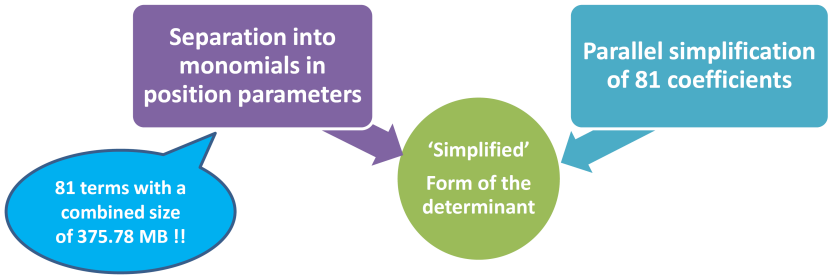

Figure 2.3: Schematic describing the methodology for the simplification of the SRSPM singularity manifold expression

The determinant of the matrix $\boldsymbol{H}$ is obtained in non-simplified form using the default `Mathematica` routine `Det`. The expression, at this stage, contains several sub-expressions with nested multiplications and additions. In order to find the degree of this polynomial, one has to expand and simplify it. The following steps are followed for simplification:

- The polynomial $f(x, y, z)$ is first expanded to its monomial-based canonical form (as per Eq. 2.4), i.e., $f(x, y, z) = \boldsymbol{D} \cdot \boldsymbol{q}$. The resulting coefficient array is of length 81, the size of which turns out to be 375.78 MB[5].

[5]It would take roughly around 120 A4 sheets, if one wished to print an expression of this size.

- The coefficients in $\boldsymbol{D}$ are then simplified in parallel over 12 computational kernels in each of 12 available processors. After simplification, the new coefficient array $\boldsymbol{D'}$ contains 24 non-zero coefficients and the rest are identified as zeros.
- The size of $\boldsymbol{D'}$ turns out to be 1.71 MB, which is a drastic reduction when compared with the input size of 375.78 MB.
- The serial simplification time using `Simplify` is 546 min (around 9.1 hrs.) whereas, the time taken to simplify these coefficients in parallel, as explained above, is 47.60 min.

2.6 Conclusions

In this chapter, simplification of symbolic expressions has been explained and some simplification routines in the current CASs are explained. A novel simplification methodology proposed in (Bandyopadhyay and Ghosal, 2006) is described. The efficiency of this algorithm is further improved by recognising the fact that it can be parallelized over several multi-processors which share a common memory. Finally, some applications where this simplification routine is used are presented along with some significant improvements achieved over the available simplification routines.

CHAPTER 3

OPERATIONS OVER POLYNOMIALS

3.1 Introduction

Arithmetic may be performed on polynomials, i.e., polynomials may be added, subtracted, multiplied, and divided, just like numbers. If two polynomials are added, subtracted, or multiplied, the result is a new polynomial. Polynomials are added or subtracted using the associative law of addition (grouping all their terms together into a single sum), possibly followed by reordering, and combining the like terms. Polynomial multiplication is performed by cross-multiplying all pairs of terms of each polynomial followed by collecting coefficients of like monomials (Kurosh, 1975).

The operations of addition and multiplication are defined for polynomials in n unknowns as follows: the sum of the polynomials $f(x_1, x_2, \ldots, x_n)$ and $g(x_1, x_2, \ldots, x_n)$ is a polynomial whose coefficients are obtained by adding the corresponding coefficients of the polynomials f and g. The product of two monomials is defined by the equation:

$$(ax_1^{k_1}x_2^{k_2}\ldots x_n^{k_n})(bx_1^{l_1}x_2^{l_2}\ldots x_n^{l_n}) = (ab)x_1^{k_1+l_1}x_2^{k_2+l_2}\ldots x_n^{k_n+l_n}, \tag{3.1}$$

after which the product of polynomials f and g is defined as the result of term wise multiplication and subsequent collection of like terms.

3.2 Monomial-based Addition and Multiplication

Two novel algorithms for addition and multiplication of two polynomials, which are either univariate or multivariate, with operations on vectors of coefficients and power products are presented in this section. The main motive behind developing these algorithms is to obtain the output in such a way that its polynomial structure is retained.

The novelty in these algorithms is that, all the operations are performed only on arrays containing coefficients and power-products.

3.2.1 Monomial-based addition - `PolyAdd`

In this algorithm, the polynomials are initially decomposed to their monomial-based canonical forms as per Eq.2.4. Later, the power product and coefficient vectors are concatenated individually into two arrays and indices of identical terms in the concatenated array of power products are found. Using these indices, the terms in the final coefficient array are added and simplified. The output, i.e., sum of the given polynomials, is the dot product of the new coefficient and power product vectors. The algorithm is schematically shown in Fig 3.1.

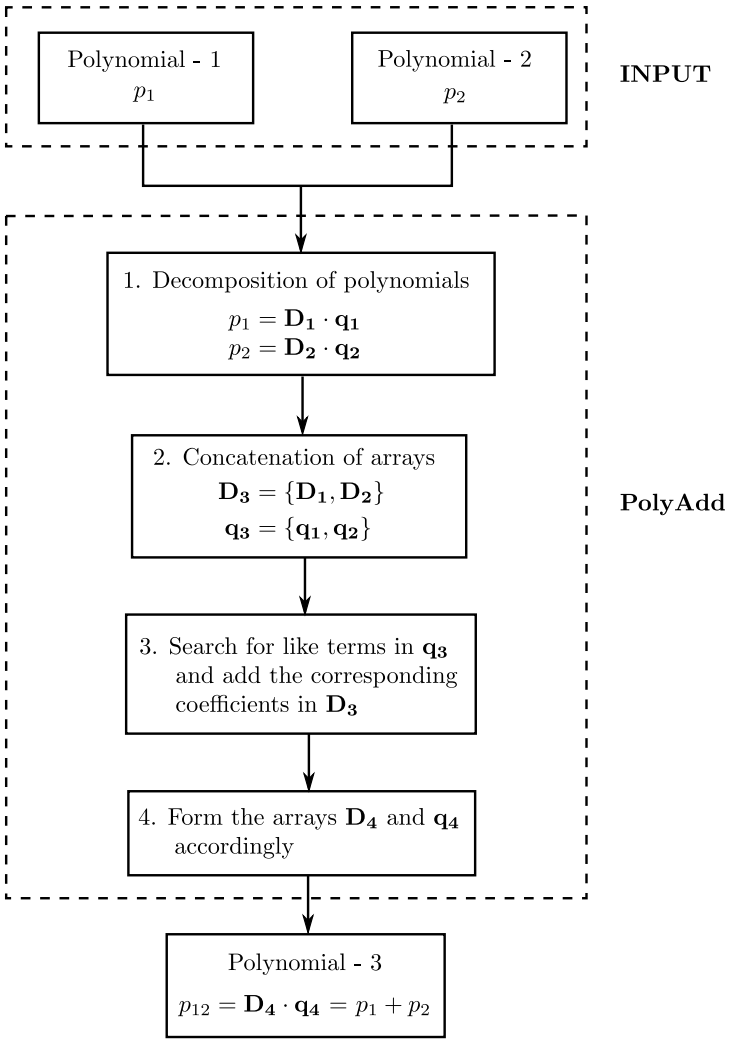

Figure 3.1: Schematic describing the algorithm for polynomial addition

An example is given below which shows the steps described above, on two univariate polynomials. Consider the two polynomials to be:

$$p_1(x) := a_1x^3 + b_1x^2 + c_1x + d_1, \tag{3.2}$$

$$p_2(x) := a_2x^2 + b_2x + c_2. \tag{3.3}$$

Step-1: Decompose the polynomials into vectors of coefficients and power products:

$$p_1(x) = \boldsymbol{D_1} \cdot \boldsymbol{q_1} = \{a_1, b_1, c_1, d_1\} \cdot \{x^3, x^2, x, 1\}, \tag{3.4}$$

$$p_2(x) = \boldsymbol{D_2} \cdot \boldsymbol{q_2} = \{a_2, b_2, c_2\} \cdot \{x^2, x, 1\}. \tag{3.5}$$

Step-2: Concatenate the coefficient and power product vectors individually as:

$$\boldsymbol{D_3} = \{\boldsymbol{D_1}, \boldsymbol{D_2}\} = \{a_1, b_1, c_1, d_1, a_2, b_2, c_2\}, \tag{3.6}$$

$$\boldsymbol{q_3} = \{\boldsymbol{q_1}, \boldsymbol{q_2}\} = \{x^3, x^2, x, 1, x^2, x, 1\}. \tag{3.7}$$

Step-3: Search for like terms in $\boldsymbol{q_3}$ and add the corresponding coefficients in $\boldsymbol{D_3}$. Form the new vectors $\boldsymbol{q_4}$ and $\boldsymbol{D_4}$, accordingly:

$$\boldsymbol{q_4} = \{x^3, x^2, x, 1\}, \tag{3.8}$$

$$\begin{aligned}\boldsymbol{D_4} &= \{\boldsymbol{D}_3[1], \boldsymbol{D}_3[2] + \boldsymbol{D}_3[5], \boldsymbol{D}_3[3] + \boldsymbol{D}_3[6], \boldsymbol{D}_3[4] + \boldsymbol{D}_3[7]\} \\ &= \{a_1, b_1 + a_2, c_1 + b_2, d_1 + c_2\}.\end{aligned} \tag{3.9}$$

Step-4: Output is the dot product of new coefficient and power product arrays:

$$\begin{aligned}p_{12} &= p_1 + p_2 \\ &= \boldsymbol{D_4} \cdot \boldsymbol{q_4} \\ &= a_1x^3 + (b_1 + a_2)x^2 + (c_1 + b_2)x + d_1 + c_2.\end{aligned}$$

3.2.2 Monomial-based multiplication - `PolyMult`

In this algorithm, a similar procedure is employed; after the polynomials are decomposed to their monomial-based canonical forms, outer products[1] of the power products and coefficient vectors are calculated and the resulting matrices are flattened (i.e., the rows of the matrix are concatenated to form a single row) down to vectors. Finally, using indices of the identical terms in the power product vector, the terms in the final coefficient array are added and simplified. The output, i.e., product of the two polynomials is the dot product of the new coefficient and power product vectors. The schematic for `PolyMult` is shown in Fig. 3.2.

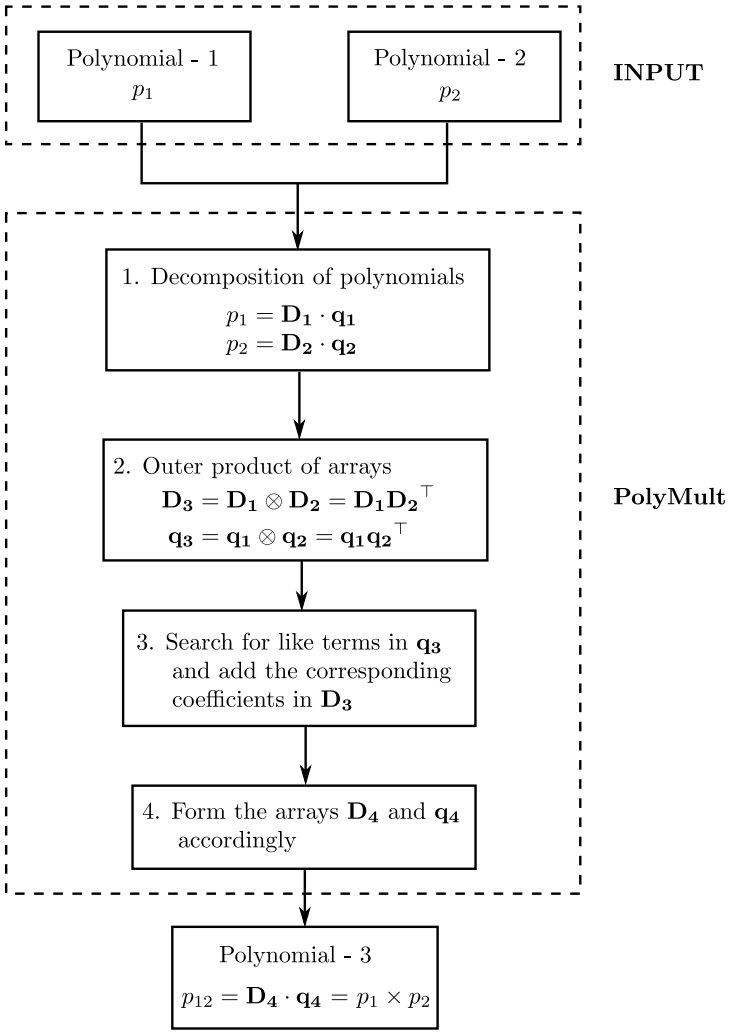

Figure 3.2: Schematic describing the algorithm for polynomial multiplication

Below given is an example which shows the steps described above when applied to two

[1]The *outer product* of two vectors $\boldsymbol{v}_1, \boldsymbol{v}_2$ is a square matrix containing all possible multiplications of pair-wise terms and is denoted by $\boldsymbol{v}_1 \otimes \boldsymbol{v}_2 = \boldsymbol{v}_1 \boldsymbol{v}_2^\top$.

univariate polynomials. Again, consider the two polynomials given in Eqs. (3.2, 3.3).

Step-1: Decompose the polynomials into vectors of coefficients and power products as shown in Eqs. (3.4, 3.5).

Step-2: Take the outer products of $\boldsymbol{D}_1$, $\boldsymbol{D}_2$ and $\boldsymbol{q}_1$, $\boldsymbol{q}_2$ and flatten them as:

$$\begin{aligned}\boldsymbol{D}_3 =& \boldsymbol{D}_1 \otimes \boldsymbol{D}_2 = \{a_1a_2, a_1b_2, a_1c_2, b_1a_2, b_1b_2, b_1c_2, c_1a_2, c_1b_2, c_1c_2, d_1a_2, d_1b_2, d_1c_2\},\\ \boldsymbol{q}_3 =& \boldsymbol{q}_1 \otimes \boldsymbol{q}_2 = \{x^5, x^4, x^3, x^4, x^3, x^2, x^3, x^2, x, x^2, x, 1\}.\end{aligned}$$

Step-3: Search for like terms in $\boldsymbol{q_3}$ and add the corresponding coefficients in $\boldsymbol{D_3}$. Form the new vectors $\boldsymbol{q_4}$ and $\boldsymbol{D_4}$, accordingly:

$$\begin{aligned}\boldsymbol{q}_4 =& \{x^5, x^4, x^3, x^2, x, 1\},\\ \boldsymbol{D}_4 =& \{\boldsymbol{D}_3[1], \boldsymbol{D}_3[2] + \boldsymbol{D}_3[4], \boldsymbol{D}_3[3] + \boldsymbol{D}_3[5] + \boldsymbol{D}_3[7], \boldsymbol{D}_3[6] + \boldsymbol{D}_3[8] + \boldsymbol{D}_3[10],\\ & \boldsymbol{D}_3[9] + \boldsymbol{D}_3[11], \boldsymbol{D}_3[12]\}.\end{aligned}$$

Step-4: Output is the dot product of new coefficient and power product arrays:

$$\begin{aligned}p_{12} =& p_1 \times p_2\\ =& \boldsymbol{D_4} \cdot \boldsymbol{q_4}\\ =& a_1a_2x^5 + (a_1b_2 + b_1a_2)x^4 + (a_1c_2 + b_1b_2 + c_1a_2)x^3\\ & + (b_1c_2 + c_1b_2 + d_1a_2)x^2 + (c_1c_2 + d_1b_2)x + d_1c_2\end{aligned}$$

3.3 Conclusions

In this chapter, two algorithms are presented for the addition and multiplication of two polynomials. In these algorithms, the first step is to decompose the polynomials into arrays containing coefficients and the corresponding power-products. Later, by performing some operations on these arrays, the final results are obtained. The main outcome of these algorithms is that the resulting polynomials retain the monomial structure with simplified coefficients and perfect grouping of like-terms.

CHAPTER 4

SOLVING MULTI-VARIATE POLYNOMIAL EQUATIONS

4.1 Introduction

Formulæfor finding the roots of univariate polynomial equations of degree 2 in terms of square roots have been known since ancient times, and for polynomials of degree 3 or 4 similar formulas were found later in the 16th century (Jacobson, 1985). It was proved that there can be no general (finite) formula, involving only arithmetic operations and radicals, that expresses the roots of a polynomial of degree 5 or greater in terms of its coefficients (see (Jacobson, 1985)). However, numerical approximation of roots of polynomial equations in one unknown can be done easily on a computer by several root-finding algorithms like Jenkins-Traub method (Jenkins and Traub, 1970), Laguerre's method (Mekwi, 2004), Durand-Kerner method (Terui and Sasaki, 2002) and all the CASs implement several of these algorithms.

Simultaneous polynomial equations are generally solved by finding numerical approximations for solutions or by eliminating one variable at each step until a final univariate in one of the remaining unknowns is obtained. Given a linear system, one can convert it to a triangular system via Gaussian elimination. For the non-linear case, given a polynomial system $F = \mathbf{0}$, one can convert (decompose or triangularize) it to a finite set of triangular sets (Moreno-Mazaa and Chena, 2011).

For example, let $F = \{f_1(x_1, x_2, \ldots, x_n), f_2(x_1, x_2, \ldots, x_n), \ldots, f_n(x_1, x_2, \ldots, x_n)\}$. Then the triangular system of equations associated with F can be represented as:

$$g_1(x_1) = 0 \tag{4.1}$$

$$g_2(x_1, x_2) = 0 \tag{4.2}$$

$$\ldots$$

$$g_n(x_1, x_2, \ldots, x_n) = 0 \tag{4.3}$$

The solutions of this system are obtained by first solving for x_1 from $g_1(x_1) = 0$ and substituting the value(s) of x_1 in $g_2, \ldots, g_n$. Then, g_2 is solved for x_2, the values of which are again substituted in $g_3, \ldots, g_n$. This process is continued until all the unknowns are solved for.

4.2 Resultant

The *resultant* of two polynomials is a polynomial expression of their coefficients, which is equal to zero if and only if the polynomials have at least one common root, or, equivalently, a common factor over their field of coefficients (see Cox *et al.* (2007), Cohen (2003), Salmon (1866)). An important application of the resultant is the elimination of one variable from a system of two polynomial equations. The resultant of the polynomials can be calculated, e.g., by finding the determinant of either Sylvester's matrix or Bezout's matrix associated with them. The construction of these two matrices for any given pair of polynomials is described in the subsections below.

4.2.1 Sylvester's method

Sylvester's method for finding resultant involves constructing *Sylvester's matrix* (Salmon, 1866) which is a square matrix associated with the pair of polynomials which can be either univariate or multivariate. The entries of the Sylvester matrix of two polynomials are coefficients of the polynomials with respect to the variable to be eliminated. The determinant of the Sylvester's matrix of two polynomials is their resultant, which is zero when the two polynomials have a common root. Forming such a matrix is shown below with an example.

Suppose x is the variable that needs to be eliminated from the two simultaneous equations:

$$p_1 : A_0x^m + A_1x^{m-1} + A_2x^{m-1} + \cdots + A_{m-1}x + A_m = 0 \tag{4.4}$$

$$p_2 : B_0x^n + B_1x^{n-1} + B_2x^{n-1} + \cdots + B_{n-1}x + B_n = 0 \tag{4.5}$$

where, each of A_i, B_j can be polynomials in variables other than x. Multiply, p_1 by

$x, x^2, x^3, \ldots, x^{n-1}$ and p_2 by $x, x^2, x^3, \ldots, x^{m-1}$ to generate a total of $(m+n)$ linearly independent equations and express the complete system of equations as:

$$\boldsymbol{C}\boldsymbol{x} = \mathbf{0}$$

where, $\boldsymbol{C}$ is the Sylvester's matrix formed with coefficients of p_1, p_2 w.r.t. x, and, $\boldsymbol{x}$ is $\{x^{m+n-1}, x^{m+n-2}, \ldots, x^2, x, 1\}^\top$. Constructing a Sylvester matrix is a simple process whereas the order of matrix is large and is equal to sum of degrees of the polynomials with respect to the variable to be eliminated.

4.2.2 Bezout's method

Bezout's method of finding resultant involves finding *Bezout's matrix* first, which is similar to Sylvester's matrix whereas the order of matrix is less than that of a corresponding Sylvester's matrix and is equal to the maximum degree of the two polynomials, with respect to the variable that needs to be eliminated. It is thus a smaller matrix than the Sylvester matrix, but has more complicated entries and its determinant still equals the resultant of the polynomials. The construction of Bezout's matrix can be found in (Cayley, 1857).

4.3 Determinant of a matrix with polynomial entries

The computation of determinants of matrices whose entries are multivariate polynomials is very common in different problems of computer algebra such as solving systems of polynomial equations, computing multi polynomial resultants, implicitizing curves and surfaces and so on (see, (Cox *et al.*, 2007), (Marco and Martinez, 2004)). In this sense, our work can serve in many applications as a component of more complex algorithms. Some preliminary algorithms which are required to expand the determinant of such a matrix are explained in this section.

4.3.1 Newton's identity

The determinant of a matrix $\boldsymbol{B}$ with polynomial entries is obtained by explicitly computing the invariants of the matrix. The formula required for computation of the invariants is obtained from the matrix-form of Newton's identities as used in (Bandyopadhyay and Ghosal, 2004) and is given below.

$$I_k = \frac{(-1)^{k+1}}{k}(\mathrm{tr}(\boldsymbol{B}^k) + \sum_{i=1}^{k-1}(-1)^i I_i \mathrm{tr}(\boldsymbol{B}^{k-i})),$$

$$I_1 = \mathrm{tr}(\boldsymbol{B}), \quad \text{trace of the matrix.}$$

where, 'n' is the order of the matrix $\boldsymbol{B}$, k = 1, . . . , n, and I_k is the k^{th} invariant of $\boldsymbol{B}$.

The routines `PolyAdd` and `PolyMult` are used to add and multiply respectively, the polynomials that are obtained at every step of this recursive formula to obtain the final determinant in polynomial form.

As one can observe, finding determinant from the above identity requires computation of trace of a matrix raised to a power. In the following section, a method to find the same is explained.

4.3.2 Trace of a matrix raised to a power

The trace of an $n \times n$ square matrix is defined to be the sum of the elements on the principal diagonal. The trace of a product of two matrices can be rewritten as the sum of entry-wise products of elements:

$$\mathrm{tr}(\boldsymbol{A}^\top \boldsymbol{B}) = \mathrm{tr}(\boldsymbol{A}\boldsymbol{B}^\top) = \sum_{i,j} \boldsymbol{A}_{i,j}\boldsymbol{B}_{i,j}$$

Taking advantage of the above property, one can find trace of $\boldsymbol{A}^k$ by computing $\boldsymbol{A}^{\frac{k}{2}}$ if k is even and $\boldsymbol{A}^{\frac{k+1}{2}}$, $\boldsymbol{A}^{\frac{k-1}{2}}$ if k is odd. More explicitly,

$$\text{if } k \text{ is even, } \mathrm{tr}(\boldsymbol{A}^k) = \mathrm{tr}(\boldsymbol{A}_1^\top \boldsymbol{A}_2) \text{ where, } \boldsymbol{A}_1 = (\boldsymbol{A}^{\frac{k}{2}})^\top \text{ and } \boldsymbol{A}_2 = \boldsymbol{A}^{\frac{k}{2}}$$

$$\text{if } k \text{ is odd, } \mathrm{tr}(\boldsymbol{A}^k) = \mathrm{tr}(\boldsymbol{A}_1^\top \boldsymbol{A}_2) \text{ where, } \boldsymbol{A}_1 = (\boldsymbol{A}^{\frac{k+1}{2}})^\top \text{ and } \boldsymbol{A}_2 = \boldsymbol{A}^{\frac{k-1}{2}}$$

Thus, if one has to find $\mathrm{tr}(\boldsymbol{A}^6)$, it is enough to compute $\boldsymbol{A}^3$. Now, suppose we want to find the determinant of a 15×15 matrix $\boldsymbol{H}$. As per the described methodology, one has to find $\boldsymbol{H}^7$ and $\boldsymbol{H}^8$. The question is, can we find exponents of a matrix with least number of multiplications? A general method called 'Exponentiating by squaring' is the answer and its methodology, implementation are described below.

4.3.3 Exponentiation by squaring

Exponentiation by squaring is a method for fast computation of large positive integer powers of a number or a polynomial or a square matrix. The basic method is to square the matrix, store the result, and perform multiplications until required exponent is reached. Below is the pseudo-code showing implementation of the recursive algorithm:

```
function expbysqr(A, n)
      If n = 1, Return(A);
      else if n = even, Return(expbysqr(A^2, n/2));
      else if n = odd, Return(A * expbysqr(A^2, (n-1)/2));
end;
```

As one can observe, the above algorithm uses already computed exponents repeatedly as the recursion happens. The efficiency of this algorithm can be increased further by a technique known as 'Memoization' (Michie, 1968). *Memoizing* a function makes it faster by trading space for time. Memoization means caching the results of earlier calculations so that we don't have to repeat the same calculations later.

4.3.4 Determinant and Resultant routines - `NDet` and `PolyRes`

Implementing all the algorithms/techniques described in the previous sections in the formula for finding determinant by invariants, we finally came up with two algorithms: one for finding determinant of any matrix with symbolic or numerical entries (called `NDet`) and the other for finding resultant of two polynomials w.r.t. to a specified variable (called `PolyRes`). The pseudo-codes for the algorithms are given below which are followed by some applications and results.

Below is the pseudo-code showing implementation of Newton's identity for `NDet` algorithm:

```
function NDet(A)
    n = Length(A); % Size of input matrix
    inv = ConstantArray[0,n]; % Initializing an array to store invariants
    t1 = ConstantArray[0,n]; % Initializing an array to store traces
    inv[[1]] = tr(A); % First invariant is the trace of input matrix
    for i = 1 to n
      t1[[i]] = tr(A^i); % Finding traces employing the product rule
      temp = 0; % Initializing a temporary variable
      for j = 1 to i-1 % Determinant calculation
        t2 = t2 + (-1)^j*inv[[j]]*t1[[i-j]];
        j = j+1;
      end for loop;
      inv[[i]] = (-1)^((i+1)/i)*(t1[[i]]+t2);
      i = i+1;
    end for loop;
    Return(inv[[n]]); % Return the determinant of input matrix
end;
```

Below shown in Fig.4.1 is the schematic for `PolyRes` followed by the pseudo-code for the same, which finds the resultant of two polynomials (p_1, p_2) w.r.t. one of the unknown variables (var).

This algorithm is entirely similar to the `NDet` algorithm except that additions and multiplications are replaced with `PolyAdd` and `PolyMult` respectively.

```
function PolyRes(p1, p2, var)
    B = bezoutmatrix(p1, p2, var); % Construct Bezout's matrix
    n = Length(B); % Size of Bezout's matrix
    inv = ConstantArray[0,n]; % Initializing an array to store invariants
    t1 = ConstantArray[0,n]; % Initializing an array to store traces
    inv[[1]] = tr(B); % First invariant is the trace of input matrix
    for i = 1 to n
```

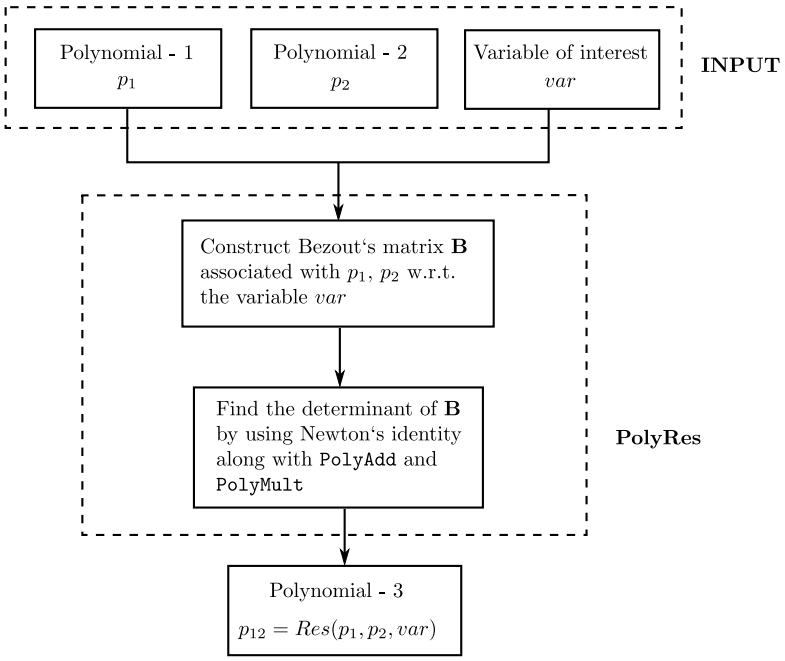

Figure 4.1: Schematic describing the algorithm for finding resultant of a pair of polynomials w.r.t. a variable of interest

```
        t1[[i]] = tr(B^i); % Finding traces employing the product rule
        temp = 0; % Initializing a temporary variable
        for j = 1 to i-1 % Determinant calculation
          t2 =  PolyAdd[t2,(-1)^j*PolyMult[inv[[j]],t1[[i-j]]]];
          j = j+1;
        end for loop;
        inv[[i]] = (-1)^((i+1)/i)*(PolyAdd[t1[[i]],t2]);
        i = i+1;
    end for loop;
    Return(inv[[n]]); % Return the resultant of input polynomials
end;
```

4.4 Applications and results

This section contains some of the significant results achieved by employing the above described algorithms. The first application shows the efficiency of `NDet` algorithm in finding determinants of symbolic matrices, over determinant finding routines in current CASs. In the second application, the usage of `PolyRes` algorithm is described in

finding maximal singularity-free region for a given SRSPM, around a given point in space.

4.4.1 Symbolic determinants

The following table shows the efficiency[1] of `NDet` routine as compared to *Mathematica*'s in-built command `Det`, in computing the determinant of a random symbolic square matrix.

Table 4.1: Comparison of times taken and output sizes obtained by using `NDet` and `Det` routines

Order of the input matrix	Time taken by **`Det`** routine in sec.	Time taken by **`NDet`** routine in sec.	Size of output in MB. using **`Det`**	Size of output in MB. using **`NDet`**
6	0.016	0.016	0.070	0.582
7	0.047	0.016	0.487	2.448
8	0.078	0.031	3.897	8.334
9	0.172	0.062	35.074	34.337
10	0.639	0.078	350.747	117.394
11	4.914	0.157	3858.213	391.105

For matrices with higher orders, *Mathematica* takes huge amount of time and RAM to compute the determinant whereas, with `NDet` routine, that is not the case. Table 4.2 shows the times taken to compute determinants of a symbolic matrix with order more than 11.

Using this routine, one can find the determinant of a 20×20 matrix of symbolic entries within a minute without any hike in RAM usage. This is a significant result as currently available CASs like `Maple`, `SINGULAR` etc. cannot find determinants of symbolic matrices which are significantly large in size without consuming huge amounts of memory and time.

[1] On a system with Mathematica version 9 for 64 bit Windows on a 4-core Intel Core i7-2670QM CPU @ 2.20 GHz clock speed and 6 GB of RAM

Table 4.2: Times taken by `NDet` routine for finding determinant of a random symbolic matrix of order greater or equal to 12

Order of the matrix	Time taken by **`NDet`** routine in sec.
12	0.281
13	1.092
14	1.638
15	3.416
16	5.070
17	13.588
18	21.072
19	41.834
20	57.585

4.4.2 Determining maximal singularity-free region of 6-DoF spatial manipulator

In this section, we address the determination of the maximal singularity-free workspace around a prescribed position for a given architecture of an SRSPM. The singularity manifold expression in $\mathbb{R}^3$ for this manipulator is reported in Bandyopadhyay and Ghosal (2006). Finding such a singularity-free region involves solving a system of polynomial equations with Cartesian variables (x, y, z) as the unknowns. The techniques employed for obtaining solutions of the polynomial equation, form the main part of this section.

Parallel mechanisms possess significant advantages over serial mechanisms in terms of dynamic properties, load-carrying capacity, high accuracy as well as stiffness or stability. They are widely used as flight and vehicle simulators, high-precision machining centres, mining machines, motion simulators and so on(Bandyopadhyay and Ghosal (2006), Li *et al.* (2007)). However, unlike in serial mechanisms, where there exist only boundary singularities, the closed-loop nature of parallel mechanisms generates complex singularities inside the workspace, which makes the workspace analysis and the trajectory planning of parallel mechanisms a very difficult problem (Jiang and Gosselin, 2009). The idea behind trajectory planning algorithms (see, Sen *et al.* (2003), Merlet (1994), Dash *et al.* (2003)) is to connect an initial configuration to a final configuration through a singularity-free path, which is a fixed path for a given task.

Therefore, it is highly desirable to develop algorithms that can locate singularity-free zones inside the workspace. One such attempt is made in this work to find the

singularity free zones inside the workspace of a 6-DoF SRSPM. The kinematic model of the SRSPM that is discussed in Sec. 2.5.2 is shown in Fig. 4.2. Below described are the two different approaches followed to find a sphere of maximum radius (r^s_{max}) and cylinder of maximum radius (r^c_{max}) around a given centre point, given the singularity manifold expression.

Maximal singularity-free sphere

The singularity manifold expression of an SRSPM can be represented as:

$$f(x,y,z) = a_1x^2z + a_2x^2 + a_3xyz + a_4xy + a_5xz^2 + a_6xz + a_7x + a_8y^2z \\ +a_9y^2 + a_{10}yz^2 + a_{11}yz + a_{12}y + a_{13}z^3 + a_{14}z^2 + a_{15}z + a_{16}, \quad (4.6)$$

where, each of $a_i \in \mathbb{R}$ are dependent only on the orientation and architecture parameters of the SRSPM considered.

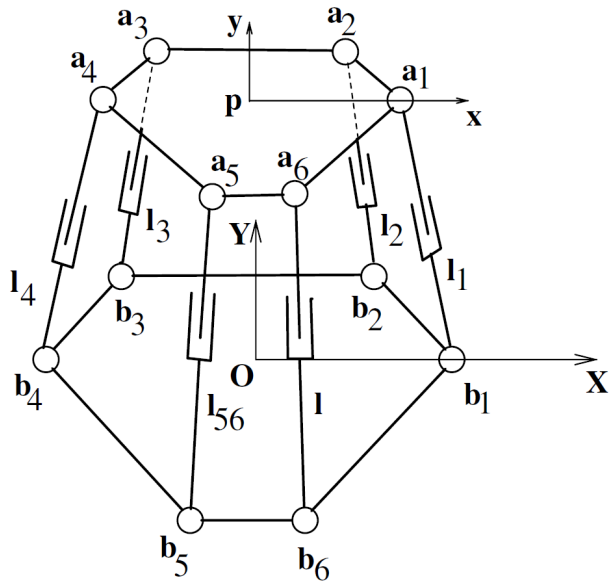

Figure 4.2: Kinematic model model of the SRSPM (adapted from Bandyopadhyay and Ghosal (2006))

To start with, the equation of sphere is given by:

$$g(x,y,z,r) := (x-x_0)^2 + (y-y_0)^2 + (z-z_0)^2 - r^2 = 0, \quad (4.7)$$

where, (x_0, y_0, z_0) is the given centre point and r is the radius of the sphere, the maxi-

mum value of which (r^s_{max}) has to be calculated. In order to do so, one has to find the point of intersection of surface given by Eq.(4.6) and the sphere such that the surfaces are tangential to each other. The methodology for finding the same is shown in Fig. 4.5 and is explained below:

- Tangency condition is applied for both the surfaces at their point of intersection which has to be solved for. Two schematics depicting the conditions for tangency and non-tangency between the singularity surface and the sphere are shown in Fig. 4.3 and Fig. 4.4, respectively.

$$\text{Normal to the surface } f(x,y,z)=0 : \boldsymbol{n}_1 = \nabla f$$
$$\text{Normal to the surface } g(x,y,z)=0 : \boldsymbol{n}_2 = \nabla g$$
$$\text{For tangency, normals should be parallel : } \nabla f \times \nabla g = \mathbf{0}$$

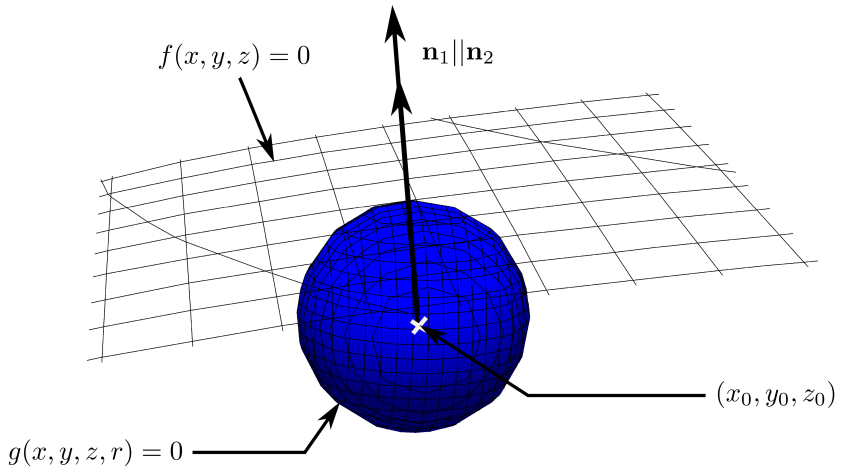

Figure 4.3: Schematic depicting tangency between the surfaces $f = 0$ and $g = 0$

- The result of $\nabla f \times \nabla g$ is a vector with 3 components: $\{h_1(x,y,z), h_2(x,y,z), h_3(x,y,z)\}^\top$, which are linearly independent. Any two of the components can be considered for the elimination stage.

- We have now, a system of 4 equations: $f(x,y,z)=0,\ g(x,y,z,r)=0,\ h_1(x,y,z)=0,$ $h_2(x,y,z)=0$ in 4 unknowns x, y, z, r.
- Using `PolyRes`, the resultants of $(f,\ h_1)$ and of $(h_1,\ h_2)$ w.r.t. the unknown variable x are found, which are denoted by $F_1(y,z)$ and $F_2(y,z)$, respectively. The degrees of the unknowns y and z in F_1 are 6, 8 respectively, and in F_2 are 5, 7 respectively.
- These polynomials are extracted symbolically and are simplified to their monomial-based canonical forms. The sizes of the expressions after simplification are 0.281 MB and 1.025 MB respectively.

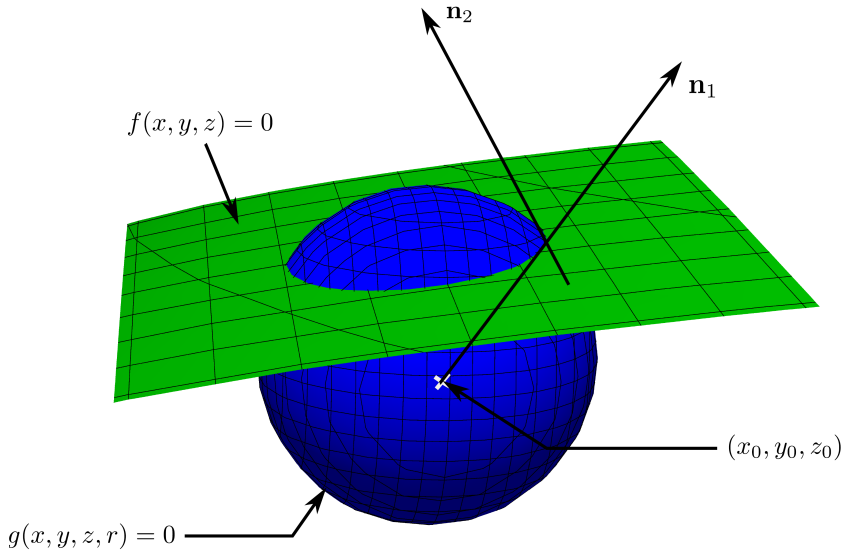

Figure 4.4: Schematic depicting non-tangency between the surfaces $f = 0$ and $g = 0$

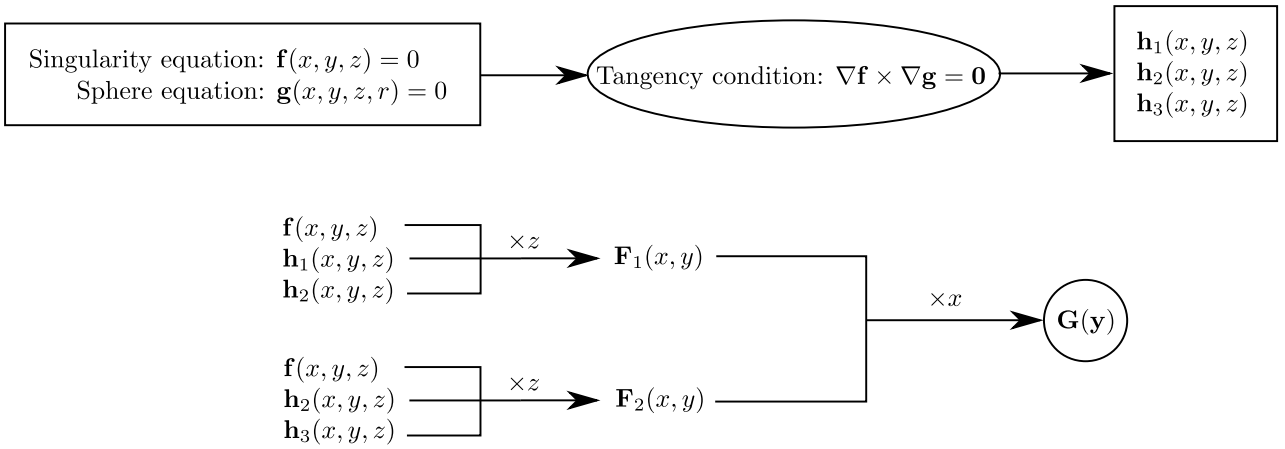

Figure 4.5: Schematic describing the process of elimination for finding maximal sphere

- In order to find the resultant of F_1 and F_2 w.r.t. y, Bezout's matrix associated with them, $\boldsymbol{H}$ has been constructed. The elements of this matrix are polynomials in z, the degrees of which are shown in the 6×6 matrix below:

$$\begin{pmatrix} 7 & 6 & 5 & 4 & 3 & 2 \\ 14 & 13 & 12 & 11 & 10 & 7 \\ 13 & 12 & 11 & 10 & 9 & 6 \\ 12 & 11 & 10 & 9 & 8 & 5 \\ 11 & 10 & 9 & 8 & 7 & 4 \\ 10 & 9 & 8 & 7 & 6 & 3 \end{pmatrix}$$

The determinant of $\boldsymbol{H}$ is computed[2] entirely in symbolic form using `PolyRes`, the size of which turns out to be 25.088 GB. The computation time is around 37 minutes.

- This determinant is the final resultant $\mathbf{G}(\mathbf{z})$ which is a univariate polynomial in z of degree 52.

At this stage, we tried substituting the numerical values for coefficients of $G(z)$. However, the substitution time turns out to around 3 days as the expression is too large to handle. Therefore, the numerical values are substituted in $\boldsymbol{H}$ and the coefficients of $G(z)$ are found numerically.

- $G(z)$ is solved numerically in `Mathematica` with a high precision of 150 and real solutions for z are extracted. These solutions are substituted back in $F_1(y, z)$, $F_2(y, z)$ and all the possible real solutions for y are computed.
- Solutions for the remaining unknown x are found by back-substituting solutions for y and z in f, h_1, h_2 and collecting common values. Finally, these solutions for (x, y, z) are validated by finding residuals of f, h_1, h_2, h_3.
- The validated solutions are substituted in $g(x, y, z, r)$ to find the radius of maximum singularity-free sphere corresponding to the centre point (x_0, y_0, z_0).

Employing the above procedure, we found out the maximal singularity-free sphere for a real SRSPM which is being used a flight-simulator. The architecture parameters and the range of heave motion for the top platform for the SRSPM are taken from Egner (1996). The ball parameters (see, Mishchenko and Fomenko (1988)) (θ, α, β) that define the orientation of the top platform, for the initial position are: $(45°, 60°, 30°)$.

In order to find the 'home' position of the manipulator corresponding to (x_0, y_0) = (0, 0), the heave motion (z value) of the top platform is varied from 1.35m. to 1.95m.,

[2]Using Mathematica version 10.0 for 64 bit Debian 7.8 on a server with 32 AMD Opteron-6376 processors @ 1.4 GHz clock speed and 256 GB of RAM.

with increments of 0.5mm. The position is found at z = 1.95m., where the radius corresponding to maximal-singularity free sphere is maximum among all the values in range of motion.

The home position for the manipulator is at (0, 0, 1.95) and it is chosen as the fixed centre of the maximal-sphere, i.e., $(x_0, y_0, z_0) = (0, 0, 1.95)$. The orientation parameters (θ, α, β) are varied by $\pm 10°$ each, i.e., from $(35°, 50°, 20°)$ to $(55°, 70°, 40°)$ with increments of $0.5°$. The maximum radius among these 68,921 points is found by parallel evaluation on a system[3] with 25 cores and 256 GB of RAM. The time taken for this entire scanning is 5 hours and 23 minutes. The value of maximum radius is obtained as 1.856m, i.e., r^s_{max} = 1.856m. The ball parameters corresponding to r^s_{max} are (θ, α, β) = $(35°, 70°, 20°)$. The sphere with $r = r^s_{max}$ along with the singularity surface is shown in Fig. 4.6.

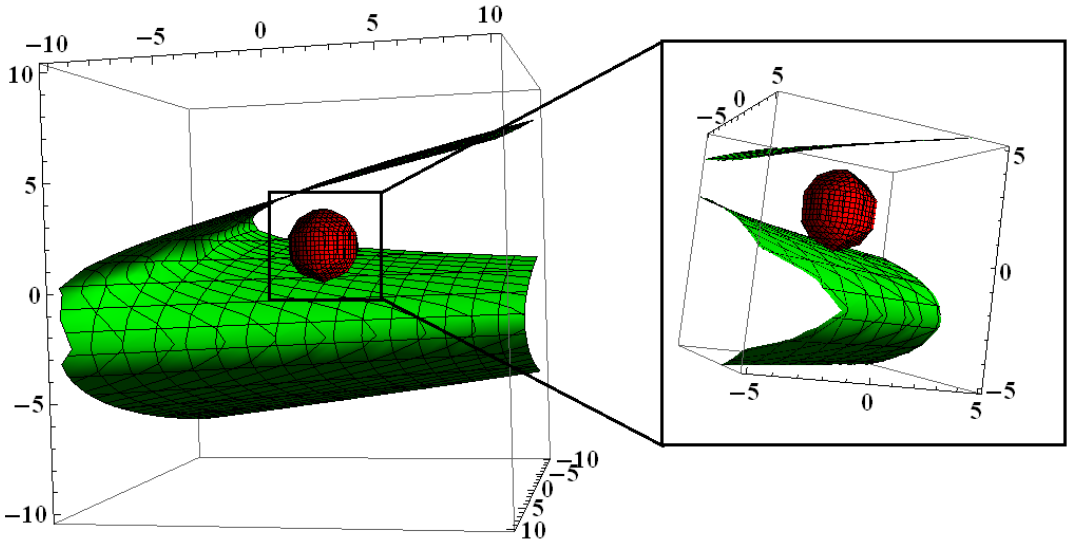

Figure 4.6: Maximal singularity free sphere for the initial grid

Another evaluation for finding the sphere is performed, this time with variation in ball parameters being $\pm 20°$. Keeping home position unchanged, the orientation parameters are varied from $(25°, 40°, 10°)$ to $(65°, 80°, 50°)$ again with increments of $0.5°$. The maximum radius among these 5,31,441 points is again found by parallel evaluation on the same system. The time taken for this entire scanning is 40 hours and 16 minutes. This time, the value of r^s_{max} is 1.928m. and is obtained at (θ, α, β) = $(65°, 80°, 50°)$.

[3]With Mathematica version 10.0 for 64 bit Debian 7.8 on a server with 32 AMD Opteron-6376 processors @ 1.4 GHz clock speed and 256 GB of RAM.

The maximal sphere along with the singularity surface is shown in Fig. 4.7.

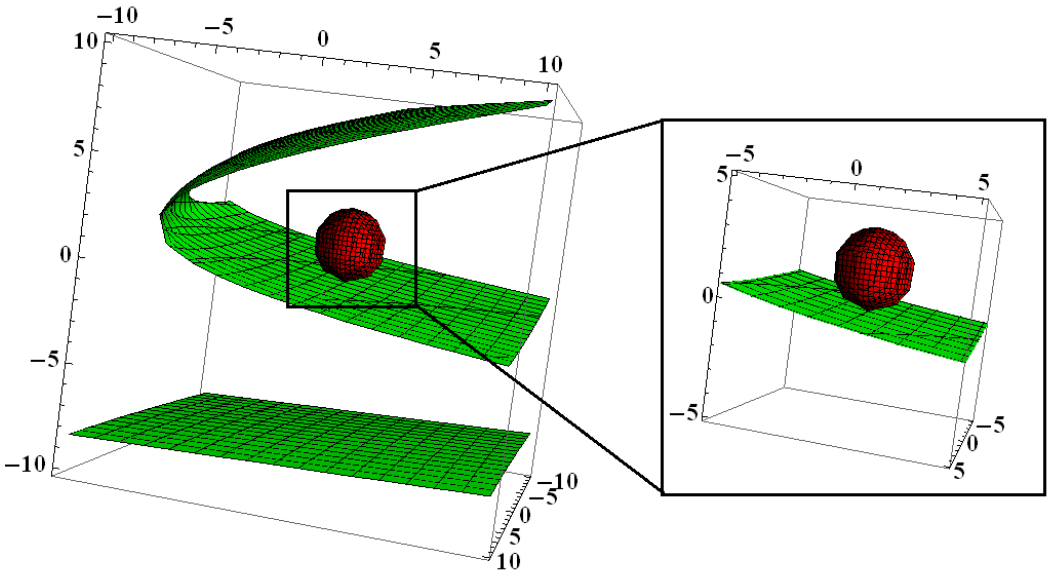

Figure 4.7: Maximal singularity free sphere for the final grid

As one can observe, the value of r^s_{max} obtained in the first evaluation (let, the value be represented by $(r^s_{max})_1$) is less than that obtained in the second evaluation (let, the value be represented by $(r^s_{max})_2$), which may seem counter-intuitive. But, $(r^s_{max})_1$ is obtained at one of the extreme values in the corresponding range of ball-parameters. Therefore, the variation in those parameters is increased for the second evaluation to find a better value for r^s_{max}.

Again, after the second evaluation, $(r^s_{max})_2$ is obtained at one of the extreme values in the range of variation of ball-parameters. This shows that the range of variation of (θ, α, β) has to be further increased if one wish to find the value of r^s_{max} for the entire range of orientations of the top-platform.

Maximal singularity-free cylinder

To find the maximal singularity free cylinder, let us assume that the equation of circle in X-Y plane is given by:

$$g(x, y, r) := (x - x_0)^2 + (y - y_0)^2 - r^2 = 0, \tag{4.8}$$

where, (x_0, y_0) is the given centre point and r is the radius of the base circle, the maximum value of which, has to be calculated. In this case, the variable z is free. To calculate the value of r^c_{max}, one has to find the point of intersection of surface given by Eq.(4.6) and the cylinder such that both the surfaces are tangential to each other. The methodology for finding the same is shown in Fig 4.8 and is explained below.

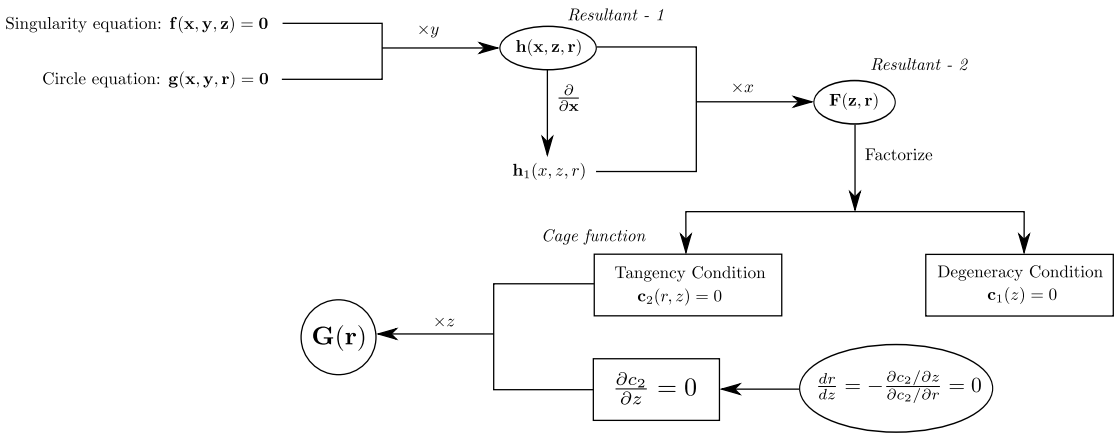

Figure 4.8: Schematic describing the process of elimination for finding maximal cylinder

The singularity manifold expression $f(x, y, z) = 0$ represents a hyperbola in X-Y plane (Bandyopadhyay and Ghosal, 2006). The first step is to find the point of intersection of the circle $g = 0$ with this conic-section such that both the curves are tangential to each other as shown in Fig. 4.9.

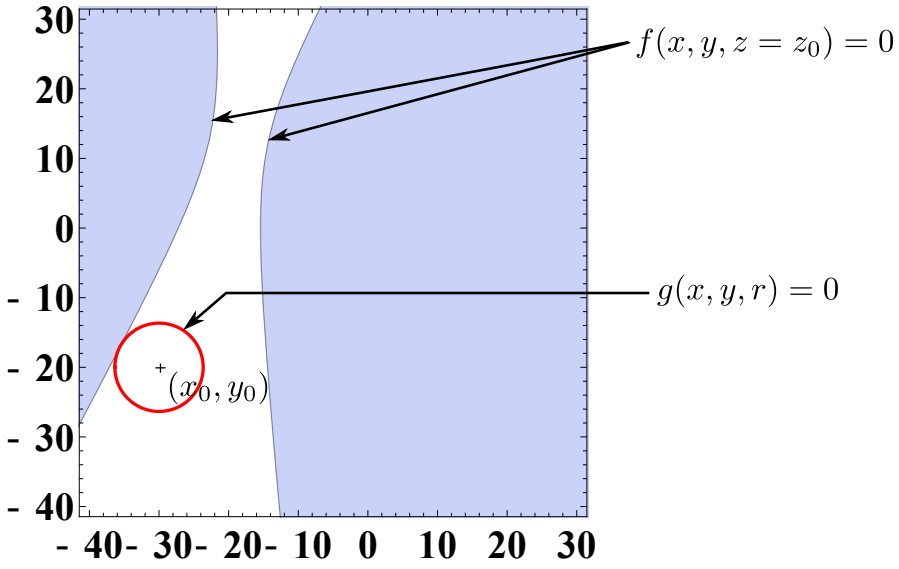

Figure 4.9: Circle and a hyperbola maintaining tangency

- One of the unknowns y is eliminated from f and g by taking resultant of the two, which is $h(x, z, r)$.
- The resultant of h and its derivative w.r.t. x results in a polynomial $F(z, r)$ which upon factorization, splits into two polynomials $c_1(z)$ and $c_2(r, z)$.
- $c_1(z) = 0$ corresponds to the values of z where the hyperbola degenerates to a pair of straight lines. This equation is independent of r and hence, it is not considered for further steps.
- The second equation $c_2(r, z) = 0$ corresponds to the values of radii for a given z, such that f and g are tangential. We call this function the 'Cage function' as the maximal cylinder will be enclosed in the surface generated by it. The surfaces generated by the cage function for different values of (x_0, y_0) are shown in Figs. (4.10, 4.11).

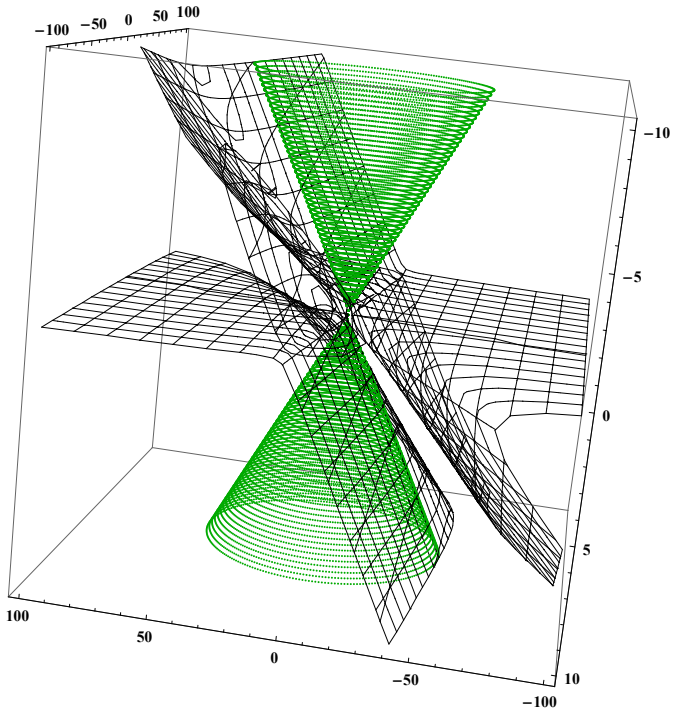

Figure 4.10: Surface generated by the cage function for (x_0, y_0) = (0, 0)

- Now, in this 'cage' function, we have to find the minimum value of r and the corresponding value of z. This is achieved by solving $c_2(r, z) = 0$ and $\frac{dr}{dz} = 0$ simultaneously.
- The resultant of these two equations w.r.t. z is the polynomial $G(r)$ which is a univariate in r and is of degree 224. This polynomial is solved with a precision of 150 and the real solutions for the radius corresponding to maximal singularity-free cylinder are validated numerically.

Employing the above procedure, we found out the maximal singularity-free cylinder for a real SRSPM known as INRIA prototype (Li *et al.*, 2007), which is shown

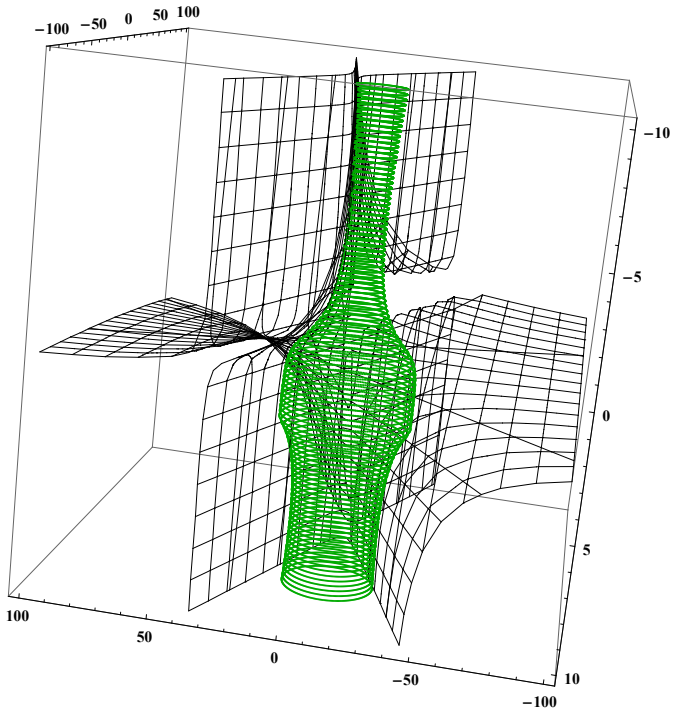

Figure 4.11: Surface generated by the cage function for (x_0, y_0) = (-10, -50)

in Fig. 4.12. The architectural parameters and the range of heave motion for the top platform for the SRSPM are taken from Bandyopadhyay and Ghosal (2006). The ball parameters (θ, α, β) that define the orientation of the top platform, for the initial position are: $(53°, 75°, 45°)$ and centre position (x_0, y_0) is (20, 20).

The value of r^c_{max} is found out to be 0.337m and the value of z at which r reaches r^c_{max} is 1.226m. The total running time of this algorithm is around 12min, on the same machine that is used in the previous case.

4.5 Conclusions

In this chapter, methodologies for solving system of multivariate polynomials and their applications in computational kinematics are presented. The concept of 'resultant' is introduced and two common methods for finding the resultant of a pair of polynomials namely, Sylvester's method and Bezout's method are described. In both os these methods, one has to find the determinant of a matrix with symbolic/polynomial entries. Addressing which, two algorithms - one for finding determinant of a symbolic matrix called as `NDet` and the other for finding determinant of a matrix with polynomial en-

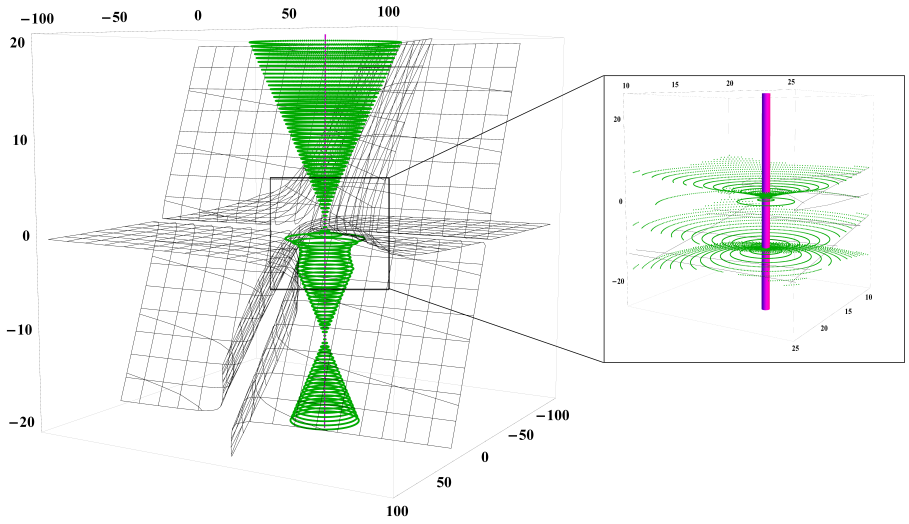

Figure 4.12: Maximal singularity-free cylinder for (x_0, y_0) = (20, 20)

tries called as `PolyRes`, are presented. Finally, some applications of these algorithms are presented along with some significant results achieved.

APPENDIX A

Rodrigue's parameters

The rotation of a reference frame $\{b\}$ w.r.t. another frame $\{a\}$ is represented by a rotation matrix ${}^a_b\boldsymbol{R}$ and Rodrigue's parameters provide one such parametrisation of $SO(3)$. The rotation matrix ${}^a_b\boldsymbol{R}$ in terms of the three Rodrigue's parameters (c_1, c_2, c_3) is given by:

$$ {}^a_b\boldsymbol{R} = \frac{1}{c_\Delta} \begin{pmatrix} 1 + c_1^2 - c_2^2 - c_3^2 & 2(c_1c_2 - c_3) & 2(c_1c_3 + c_2) \\ 2(c_1c_2 + c_2) & 1 - c_1^2 + c_2^2 - c_3^2 & 2(c_2c_3 - c_1) \\ 2(c_1c_3 - c_2) & 2(c_2c_3 + c_2) & 1 - c_1^2 - c_2^2 + c_3^2 \end{pmatrix} $$

where, $c_\Delta = 1 + c_1^2 + c_2^2 + c_3^2$.

REFERENCES

1. **Bailey, D. H.**, **J. M. Borwein**, and **A. D. Kaiser** (2014). Automated simplification of large symbolic expressions. *Journal of Symbolic Computation*, **60**, 120–136.

2. **Bandyopadhyay, S.** and **A. Ghosal** (2004). Analytical determination of principal twists in serial, parallel and hybrid manipulators using dual vectors and matrices. *Mechanism and Machine Theory*, **39**(12), 1289 – 1305. ISSN 0094-114X.

3. **Bandyopadhyay, S.** and **A. Ghosal** (2006). Geometric characterization and parametric representation of the singularity manifold of a 6-6 Stewart platform manipulator. *Mechanisms and Machine Theory*, **41**(11), 1377–1400.

4. **Boyle, A.** and **B. Caviness** (1990). Future directions for research in symbolic computation. *Society for Industrial and Applied Mathematics, Philadelphia*.

5. **Carette, J.**, Understanding expression simplification. *In Proceedings of the 2004 International Symposium on Symbolic and Algebraic Computation*, ISSAC '04. ACM, New York, NY, USA, 2004. ISBN 1-58113-827-X.

6. **Cayley, A.** (1857). Note sur la methode d'elimination de Bezout. *J. Reine Angew. Math.*, **53**, 366–367.

7. **Cohen, J.**, *Computer Algebra and Symbolic Computation: Mathematical Methods*. Number v. 1 in A.K. Peters Series. Peters, 2003. ISBN 9781568811598.

8. **Cox, D. A.**, **J. Little**, and **D. O'Shea**, *Ideals, Varieties, and Algorithms: An Introduction to Computational Algebraic Geometry and Commutative Algebra, (Undergraduate Texts in Mathematics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007, 3rd. edition. ISBN 0387356509.

9. **Dash, A. K.**, **I.-M. Chen**, **S. H. Yeo**, and **G. Yang**, Singularity-free path planning of parallel manipulators using clustering algorithm and line geometry. *In Proceedings of the 2003 IEEE International Conference on Robotics and Automation, ICRA'03.*, volume 1. IEEE, 2003.

10. **Egner, S.** (1996). Semi-numerical solution to 6/6 Stewart-platform kinematics based on symmetry. *Applicable Algebra in Engineering, Communication and Computing*, **7**(6), 449–468.

11. **Heck, A.**, *Introduction to MAPLE*. Springer-Verlag, New York, 2003, 3rd edition.

12. **Jacobson, N.**, *Basic Algebra*. W.H. Freeman, 1985, 1st. edition. ISBN 9780716714804.

13. **Jenkins, M.** and **J. Traub** (1970). A three-stage variable-shift iteration for polynomial zeros and its relation to generalized rayleigh iteration. *Numerische Mathematik*, **14**(3), 252–263. ISSN 0029-599X.

14. **Jiang, Q.** and **C. M. Gosselin** (2009). Determination of the maximal singularity-free orientation workspace for the Gough–Stewart platform. *Mechanism and Machine Theory*, **44**(6), 1281–1293.

15. **Kaltofen, E.** (2000). Challenges of symbolic computation: my favorite open problems. *Journal of Symbolic Computation*, **29**(6), 891–919.

16. **Kodati, M.** and **S. Bandyopadhyay** (2013). Kinematic analysis of double-wishbone suspension system. *Computer Methods in Applied Mechanics and Engineering*, 562–567. In Proceedings National Conference on Machines and Mechanisms (iNaCoMM), December 18–20, Roorkee, India.

17. **Kurosh, A.**, *Higher Algebra*. Mir Publishers, 1975. ISBN 9780828507240.

18. **Li, H.**, **C. M. Gosselin**, and **M. J. Richard** (2007). Determination of the maximal singularity-free zones in the six-dimensional workspace of the general Gough–Stewart platform. *Mechanism and machine theory*, **42**(4), 497–511.

19. **Marco, A.** and **J.-J. Martinez** (2004). Parallel computation of determinants of matrices with polynomial entries. *Journal of Symbolic Computation*, **37**(6), 749–760.

20. **Martin, W. A.** and **R. J. Fateman**, The macsyma system. *In Proceedings of the second ACM symposium on Symbolic and algebraic manipulation*. ACM, 1971.

21. **Mekwi, W. R.** (2004). *Iterative methods for roots of polynomials*. M.Sc. in mathematical modelling and scientific computing, University of Oxford, Oxford, United Kingdom.

22. **Merlet, J.-P.**, Trajectory verification of parallel manipulators in the workspace. *In Proceedings of the 1994 IEEE International Conference on Robotics and Automation, ICRA'94.*. IEEE, 1994.

23. **Michie, D.** (1968). Memo functions and machine learning. *Nature*, **218**(5136), 19–22.

24. **Mishchenko, A.** and **A. Fomenko**, *A course of differential geometry and topology:*. Mir Publishers, 1988.

25. **Moreno-Mazaa, M.** and **C. Chena** (2011). Algorithms for computing triangular decomposition of polynomial systems. *Journal of Symbolic computations*, 83–90. In Proceedings of International Symposium on Symbolic and Algebraic Computation(ISSAC), June 8–11, San Jose, USA.

26. **Petcu, D.**, **D. Tepeneu**, **M. Paprzycki**, and **T. Ida** (2006). Symbolic computations on grids. *Engineering the Grid*.

27. **Salmon, G.**, *Lessons Introductory to the Modern Higher Algebra*. Hodges, Smith, 1866.

28. **Selig, J. M.**, *Geometric fundamentals of robotics*. Springer, New York, 1996, 2nd. edition.

29. **Sen, S.**, **B. Dasgupta**, and **A. K. Mallik** (2003). Variational approach for singularity-free path-planning of parallel manipulators. *Mechanism and Machine Theory*, **38**(11), 1165–1183.

30. **Terui, A.** and **T. Sasaki** (2002). Durand-Kerner method for the real roots. *Japan Journal of Industrial and Applied Mathematics*, **19**(1), 19–38. ISSN 0916-7005.

31. **Watt, S. M.**, Making computer algebra more symbolic. *In Proceedings of Transgressive Computing*. 2006.

32. **Wolfram Research**, *Mathematica*. Wolfram Research, Inc., Champaign, Illinois, 2014, version 10.0 edition.