Bachelor Project Mathematics: A Hidden Symmetry of Kontsevich's Tetrahedral Flow on the Space of Rescaled 3D and 4D-Determinant Nambu-Poisson Brackets

D. Lipper

Supervised by A.V. Kiselev
In Collaboration with R. Buring
Second Assessment by N. Martynchuk

University of Groningen

July 2021

Abstract

We study symmetries of the space of Poisson brackets. Kontsevich's tetrahedral flow is known to preserve the class of 3d and 4d-determinant Nambu-Poisson brackets. This gives rise to dynamical systems containing differential polynomials in the right hand side. These expressions are highly symmetric, and we want to unravel their structure. To solve the problem we design an algorithm and implement it in Maple. We confirm the triple total skew-symmetry of that flow and we discover in which sense the structure of that flow is minimal. Our approach naturally generalizes to higher dimensions and other Kontsevich flows, yet it is unknown whether the minimal structure persists or not.

Contents

1 Introduction

Kontsevich's tetrahedral flow can be used to deform Poisson structures, such as 3d-determinant Nambu-Poisson brackets. These brackets are determined by the parameters a and ρ. The tetrahedral flow preserves the class of 3d-determinant Nambu-Poisson brackets, which gives rise to the evolution of the parameters a and ρ. The evolution equations for $\dot{a}$ and $\dot{\rho}$ were previously found to be skew-symmetric and totally skew-symmetric.

The differential geometry of these equations guarantees that we can can find marker-polynomials whose total skew-symmetrizations are equal to $\dot{a}$ and $\dot{\rho}$, and it is known that $\dot{a}$ can be represented by the total skew-symmetrization of three markers. We don't know how many markers we need for $\dot{\rho}$. We succeed in finding marker-polynomials that produce $\dot{a}$ and $\dot{\rho}$ by total skew-symmetrization, and all our finding match the theoretical predictions. On top of that we discover an extra kind of hyper-symmetry in these equations: for each structure type in $\dot{a}$ or $\dot{\rho}$, a suitable multiple of any nonzero marker of that structure can be used to construct $\dot{a}$ or $\dot{\rho}$ by total skew-symmetrization. This result was obtained by designing an algorithm and implementing it in Maple.

Furthermore, we study some evolution equations obtained by deforming 4d-determinant Nambu-Poisson brackets using Kontsevich's tetrahedral flow. To do this we modify our algorithm in Maple to also work in 4 dimensions. We hypothesize that the hyper-symmetry found in the 3-dimensional case persists in 4 dimensions.

This text is structured in the following way. Chapter 2 contains the relevant context and motivation for the problem, the necessary definitions and the research question. In chapter 3 some theorems are presented, which are used to suggest an algorithmic approach for solving the problem. Chapter 4 contains the implementation of this algorithmic approach in Maple for 3 dimensions. Chapter 5 contains the same algorithm, but implemented for use in 4 dimensions. In chapter 6 the results of this algorithm are presented and discussed. Finally, the conclusion and discussion can be found in chapters 7 and 8.

2 Problem Statement

This chapter is dedicated to formulating the research question. Section 2.1 provides some context and background information to the problem. Sections 2.2 and 2.3 contain the necessary definitions needed to formulate the research question, which can be found in section 2.4.

2.1 Context and Motivation

We study deformations of Poisson structures. Poisson structures arise in physics and can be described in many ways. One way to explain Poisson structures is using Hamiltonian mechanics. An example of this approach can be found in [1]. Alternatively, we could describe Poisson structures in the following way, although this approach makes the physical interpretation less immediately clear. A Poisson structure on $\mathbb{R}^n$ is an $n \times n$ matrix of $\mathbb{R}$-valued functions on $\mathbb{R}^n$ with entries P^{ij}, where $1 \leq i, j \leq n$. This matrix defines a bracket on smooth functions on $\mathbb{R}^n$ by

$$\{f, g\} = \sum_{i,j=1}^{i,j=n} P^{ij} \frac{\partial}{\partial x_i}(f) \frac{\partial}{\partial x_j}(g) \tag{1}$$

This bracket should satisfy the following three identities:

- Skew-symmetricity: $\{f, g\} = -\{g, f\}$, so $P^{ij} = \{x_i, x_j\} = -\{x_j, x_i\} = P^{ji}$
- Bi-linearity: $\{f, gh\} = \{f, g\}h + \{f, h\}g$
- Jacobi identity: $\{\{f, g\}h\} + \{\{f, h\}g\}\} + \{\{g, h\}f\} = 0$

When these three identities are satisfied the bracket $\{f, g\}$ is called a Poisson bracket and the matrix P^{ij} is called a Poisson matrix.

On $\mathbb{R}^3$ with coordinates (x, y, z) we are interested in the family of Poisson structures $P[a, \rho]$ given by $\{x, y\} = \rho \cdot \partial a / \partial z$ and the three cyclic permutations of $\{x, y, z\}$ of this identity, which also include $\{y, z\} = \rho \cdot \partial a / \partial x$ and $\{z, x\} = \rho \cdot \partial a / \partial y$. The general formula for these brackets is given by

$$\{f, g\} = \rho \cdot \left| \frac{\partial(a, f, g)}{\partial(x, y, z)} \right| = \rho \cdot \det \left\| \begin{smallmatrix} a_x & f_x & g_x \\ a_y & f_y & g_y \\ a_z & f_z & g_z \end{smallmatrix} \right\| \tag{2}$$

When ρ is equal to 1 these are called the 3D-determinant Nambu-Poisson brackets [2]. We are interested in the more general rescaled brackets where

ρ is an arbitrary function. On $\mathbb{R}^3$, multiplying a Nambu-Poisson bracket by an arbitrary function gives another Nambu-Poisson bracket [3]. If we denote (x_1, x_2, x_3) by (x, y, z) we have the Poisson matrix with entries $P[a, \rho]^{ij} = \{x_i, x_j\} = \epsilon^{ijk}\rho\partial_k(a)$, where ϵ is the Levi-Civita symbol: it is equal to 1 if (i, j, k) is a cyclic permutation of $(1, 2, 3)$, equal to -1 if (i, j, k) is any other permutation of $(1, 2, 3)$ and equal to 0 if (i, j, k) is not a permutation of $(1, 2, 3)$.

We would like to deform the Poisson structures. Given a Poisson matrix P we would like to find a family P_t of Poisson matrices such that $P_0 = P$. In order to be called a deformation the family P_t should depend on t in a real-analytic way around 0. Then there exists a power series expansion $P_t = P + tQ + ...$, where the t coefficient of the Jacobi identity for P_t is $[\![P, Q]\!] = \frac{1}{2}[\![P_t, P_t]\!] = 0$. Here the double brackets denote a differential on multi-vector fields, also called Schouten brackets, and Q is called a cocycle in the Poisson cohomology of P. That means that we can split the problem of finding a deformation in two: first find a Q such that $[\![P, Q]\!] = 0$, and then find out if $P + tQ$ can be extended to a deformation. We will focus exclusively on this first step.

We can deform Poisson matrices using Kontsevich's tetrahedral flow [4], which preserves the set of Poisson brackets. Specifically, the tetrahedral flow preserves the class of 3D-determinant Nambu-Poisson brackets. Then there exist formulas for particular Q, depending on P, satisfying $[\![P, Q]\!] = 0$. The explicit corrected formulas for this $Q(P)$ were found by Bouisaghouane, Buring and Kiselev [5]. Applying this to our case we might suspect that this $Q(P[a, \rho])$ is the result of deforming the 'ingredients' (a, ρ). Namely, deforming (a, ρ) simultaneously would mean having two families (a_t, ρ_t) with $(a_0, \rho_0) = (a, \rho)$. Assuming again that the families are real-analytic near 0 we get the power series $a_t = a+t\dot{a}+...$ and $\rho_t = \rho+t\dot{\rho}+...$. This yields the family of Poisson structures $P_t = P[a_t, \rho_t] = P[a, \rho] + t(P[a, r\dot{h}o] + P[\dot{a}, \rho]) + ...$, because P is linear in both arguments. Hence, taking the first-order term, we get the following matrix equation

$$Q(P[a, \rho]) = P[a, \dot{\rho}] + P[\dot{a}, \rho] \tag{3}$$

We want to solve this equation for $(\dot{a}, \dot{\rho})$ in terms of (a, ρ) and their derivatives. The solutions to this equation were found by Buring and Kiselev in 2019 [6], and are the object of study of this paper. The solutions for $\dot{a}$ and $\dot{\rho}$, which can be found in appendix A, contain respectively 228 and 426 monomials. Note that if we set ρ equal to 1 both $\dot{a}$ and $\dot{\rho}$ are equal to zero. The expressions for $\dot{a}$ and $\dot{\rho}$ are differential polynomials which contain three independent variables, $\{x, y, z\}$, and two dependent variables, $\{a, \rho\}$. Each monomial in these equations contains exactly nine derivatives: three derivatives of x, three derivatives of y and three derivatives of z. The expressions for $\dot{a}$ and $\dot{\rho}$ depend on the following jet variables:

$$\dot{a} = \dot{a}(\rho, a_x, a_y, a_z, a_{xx}, a_{xxx}, a_{yy}, a_{yyy}, a_{zz}, a_{zzz}, a_{xy}, a_{xyy}, a_{xz}, a_{xzz}, a_{xxy}, a_{xxz}, a_{yz}, a_{yzz}, a_{yyz}, a_{xyz}, \rho_x, \rho_y, \rho_z)$$

$$\dot{\rho} = \dot{\rho}(\rho, a_x, a_y, a_z, a_{xx}, a_{yy}, a_{zz}, a_{xy}, a_{xz}, a_{yz}, \rho_x, \rho_y, \rho_z, \rho_{xx}, \rho_{xxx}, \rho_{yy}, \rho_{yyy}, \rho_{zz}, \rho_{zzz}, \rho_{xy}, \rho_{xyy}, \rho_{xz}, \rho_{xzz}, \rho_{xxy}, \rho_{xxz}, \rho_{yz}, \rho_{yzz}, \rho_{yyz}, \rho_{xyz})$$

2.2 Skew-Symmetry

It was discovered by Buring and Kiselev that both $\dot{a}$ and $\dot{\rho}$ are skew-symmetric under permutations of $\{x, y, z\}$ [7]. This is also verified in section 4.2.1 of this paper.

Def 1. *A skew-symmetric polynomial is a polynomial that satisfies*

$$f(x_1, ..., x_n) = (-1)^{\pi} \cdot f(\pi(x_1), ..., \pi(x_n))$$

for all $\pi \in S_n$.

In our case this means that the polynomials $\dot{a}$ and $\dot{\rho}$ satisfy

$$\begin{aligned} \dot{a}(..., a_x, ..., \rho_{xy}, ...) &= (-1)^{\pi} \cdot \dot{a}(..., a_{\pi(x)}, ..., \rho_{\pi(x)\pi(y)}, ...) \\ \dot{\rho}(..., a_x, ..., \rho_{xy}, ...) &= (-1)^{\pi} \cdot \dot{\rho}(..., a_{\pi(x)}, ..., \rho_{\pi(x)\pi(y)}, ...) \end{aligned} \tag{4}$$

for all $\pi \in S_3$, since in our case we have three variables: $\{x, y, z\}$. From now on we will denote the operation of permuting the variables $\{x, y, z\}$ in $\dot{a}$ and $\dot{\rho}$ simply by $\pi(\dot{a})$ and $\pi(\dot{\rho})$.

Buring and Kiselev also discovered another kind of symmetry in the polynomials $\dot{a}$ and $\dot{\rho}$: the polynomials $\dot{a}$ and $\dot{\rho}$ are totally skew-symmetric.

Def 2. *The total skew-symmetrization of a polynomial f of n variables is the sum of the polynomial skewed by all possible permutations of its n variables. It is denoted by π_{S_n}.*

$$\pi_{S_n}(f(x_1, ...x_n)) = \sum_{\pi \in S_n} \Big((-1)^{\pi} f(\pi(x_1), ..., \pi(x_n)) \Big)$$

Buring and Kiselev found two new polynomials, called $\dot{a}'$ and $\dot{\rho}'$, which can produce $\dot{a}$ and $\dot{\rho}$ respectively by total skew-symmetrization [7]. The polynomials $\dot{a}'$ and $\dot{\rho}'$ consist of 38 and 71 terms respectively, and can be found in appendix B. These polynomials thus satisfy

$$\begin{aligned} \dot{a} &= \pi_{S_3}(\dot{a}') \\ \dot{\rho} &= \pi_{S_3}(\dot{\rho}') \end{aligned} \tag{5}$$

This is also verified in section 4.2.2 of this paper.

2.3 Marker-Monomials

In this section we introduce marker-monomials to further study the symmetry of the polynomials $\dot{a}$ and $\dot{\rho}$.

Def 3. *A marker-monomial (or marker) is a monomial where the independent variables are partitioned into three triples.*

The nine derivatives $\{x, x, x, y, y, y, z, z, z\}$ in each monomial in $\dot{a}$ and $\dot{\rho}$ can be partitioned into three triples of $\{x, y, z\}$ by replacing the three x variables by $\{u1, u2, u3\}$, the three y variables by $\{v1, v2, v3\}$ and the three z variables by $\{w1, w2, w3\}$. After these substitutions the variables are partitioned into three triples $\{u_n, v_n, w_n\}$, for $n \in \{1, 2, 3\}$. This can be done in at most $6^3 = 216$ different ways: there are 6 ways to replace $\{x, x, x\}$ by $\{u1, u2, u3\}$, 6 ways to replace $\{y, y, y\}$ by $\{v1, v2, v3\}$ and 6 ways to replace $\{z, z, z\}$ by $\{w1, w2, w3\}$. However, in practice there are often less possibilities.

Ex 1. Let's take the first term in the polynomial $\dot{a}$

$$\dot{a}_1 = -12\rho^2 a_x \rho_y a_{xy} a_{zz} a_{xyz} \tag{6}$$

We can transform this monomial into a marker-monomial by replacing the nine independent variables with three triples. If we look at monomial (6), we see that it contains a_{zz}. This 'double' z limits the number of ways to partition $\{z, z, z\}$ to 3 possibilities, because $a_{w1w2} = a_{w2w1}$, since the order of derivatives does not matter. That means we can create at most $6 \cdot 6 \cdot 3 = 108$ different markers from monomial (6).

Let's look at *one* of the 108 possible markers we can construct from monomial (6): we will partition the variables in monomial (6) by simply replacing them in order of occurrence: the first occurrence of x by $u1$, the second occurrence of x by $u2$, the first occurrence of y by $v1$, etc. This results in the marker

$$-12\rho^2 a_{u1} \rho_{v1} a_{u2v2} a_{w1w2} a_{u3v3w3} \tag{7}$$

Now the marker (7) contains three triples $\{u_n, v_n, w_n\}$, for $n \in \{1,2,3\}$, and we can permute each triple individually. For example, we could permute the triple $\{u_2, v_2, w_2\}$ by (13). Then monomial (7) would become

$$12\rho^2 a_{u1}\rho_{v1}a_{v2w2}a_{u2w1}a_{u3v3w3} \tag{8}$$

If we then place back the original variables $\{x, y, z\}$ in monomial (8) we end up with

$$12\rho^2 a_x\rho_y a_{yz}a_{xz}a_{xyz} \tag{9}$$

The example above shows how constructing markers and permuting their variables works in practice. △

In general, the number of markers that can be constructed from a monomial can be computed as follows: we start by only looking at the x variable. If the monomial contains a_x^3, a_{xxx}, ρ_x^3 or ρ_{xxx}, then there is only one way to replace $\{x, x, x\}$ by $\{u1, u2, u3\}$. If the monomial contains a_x^2, ρ_x^2, ρ_{xxn} or a_{xxn}, where $n \in \{y, z\}$, then there are three ways to replace $\{x, x, x\}$ by $\{u1, u2, u3\}$. If the monomial contains none of these combinations then there are six ways to replace $\{x, x, x\}$ by $\{u1, u2, u3\}$. We then do the same for the variables y and z. Finally, we multiply the number of combinations for each of the variables, which gives us the number of distinct markers we can construct from the monomial, as seen in example 1.

Ex 2. The polynomials $\dot{a}$ and $\dot{\rho}$ contain some monomials that can only produce one single distinct marker, for instance the following monomial.

$$a_x^3\rho_y^3 a_{xxx} \tag{10}$$

As there is only one possible way to partition the independent variables into three triples, the only marker that can be created from this monomial is

$$a_{u1}a_{u2}a_{u3}\rho_{v1}\rho_{v2}\rho_{v3}a_{w1w2w3} \tag{11}$$

△

A property to note is that distinct monomials can never produce identical markers. Since the x variables can only be replaced by $\{u_1, u_2, u_3\}$, the y variables can only be replaced by $\{v_1, v_2, v_3\}$ and the z variables can only be replaced by $\{w_1, w_2, w_3\}$, two different monomials will always produce markers that differ in the same positions as the monomials themselves. Thus different monomials give rise to different markers.

Def 4. *The total skew-symmetrization of a marker m is the total skew-symmetrization of m with respect to all three triples, denoted by $\pi^t_{S_3}$.*

$$\pi^t_{S_3}(m) = \sum_{\pi^1 \in S_3} \sum_{\pi^2 \in S_3} \sum_{\pi^3 \in S_3} (-1)^{\pi} \pi^n(m)(x, y, z)$$

where π^n denotes permuting with respect to triple n.

There are 6 possible ways of skewing the first triple, one for each permutation in S_3. Similarly, there are 6 ways of skewing the second triple and 6 ways of skewing the third triple. Together, this gives us $6^3 = 216$ possible skew combinations. Adding all 216 terms gives us the total skew-symmetrization.

Note that the total skew-symmetrization does not consist of markers, but of monomials of $\{x, y, z\}$. After all skews are completed we replace the triple variables in the resulting 216 terms by $\{x, y, z\}$. This can only be done in one way, contrary to the construction of markers, for which there are many possibilities.

The skew-symmetry and total skew-symmetry of $\dot{a}$ and $\dot{\rho}$, as seen in section 2.2, thus hold under simultaneous permutation of three triples $\{x, y, z\}$ in each monomial. That is to say, these symmetries hold when we permute each triple, $\{u_1, v_1, w_1\}$, $\{u_2, v_2, w_2\}$ and $\{u_3, v_3, w_3\}$ simultaneously. However, it is not immediately clear which variable should go into which triple. As noted before, there are (at most) 216 different ways of sorting the 9 derivatives in every monomial in $\dot{a}$ and $\dot{\rho}$ into three triples.

2.4 Research Question

Our problem consists in finding a collection of markers whose total skew-symmetrization produces $\dot{a}$ and $\dot{\rho}$. That is to say, we are looking for new polynomials $\dot{a}''$ and $\dot{\rho}''$ consisting entirely of markers, such that the total skew-symmetrizations of $\dot{a}''$ and $\dot{\rho}''$ are equal to $\dot{a}$ and $\dot{\rho}$ respectively. Symbolically this can be written as

$$\begin{aligned} \dot{a} &= \pi^t_{S_3}(\dot{a}'')(x, y, z) \\ \dot{\rho} &= \pi^t_{S_3}(\dot{\rho}'')(x, y, z) \end{aligned} \tag{12}$$

The existence of markers-polynomials with this property is guaranteed by the differential geometry of the problem. It is also known from Buring and Kiselev that there exists a solution for $\dot{a}''$ consisting of three markers. We would like to confirm this and find a solution for $\dot{\rho}''$. Moreover, we would like to know whether these solutions are unique.

3 Looking for an Approach

In this chapter we study some properties of marker-monomials that allow us to better understand the problem stated in section 2.4. Using these insights we suggest a brute force algorithmic approach to solve the problem.

3.1 Zero and Nonzero Markers

Our first object of study are so-called zero markers.

Def 5. *A zero marker is a marker whose total skew-symmetrization is zero.*

Zero markers are of particular interest to us, because, since their total skew-symmetrizations are zero, they cannot be used to represent anything. This means that we could add any number of zero markers to a possible solution and end up with another solution. However, this does not lead to very interesting solutions. Therefore we would like to find solutions that do not contain any zero markers. To achieve this we would like to find out why some markers are zero markers, and how to easily recognise these.

Th 1. *If a marker is invariant under any transposition in S_3 with respect to any of its triples it is a zero marker.*

Proof. There are two types of markers whose total skew-symmetrization is zero:

<u>Type 1:</u> The first type is a marker that is invariant under the permutation (123) or (132) with respect to one of the three triples. In fact, if the marker is invariant under one of the permutations of order 3 in S_3, it is necessarily invariant under all permutations in S_3 with respect to this triple. Since there are three odd and three even permutations in S_3, three terms get a minus coefficient when skewed an three terms do not. That means when we totally skew this marker with respect to the triple where the invariances occur, the 6 terms will cancel out, since they are all equal except for the three minus signs.

<u>Type 2:</u> The second type of marker whose total skew-symmetrization is zero is a marker that is invariant under *just one* of the three transpositions in S_3 with respect to one of the three triples. Suppose, without loss of generality, that the marker is invariant under the permutation (23) with respect to one of the three triples. So permuting this triple by (23) is the same as permuting it by (1). It follows that permuting this triple by (13) is the same as permuting by (123) and permuting by (12) is the same as permuting by (132). So when

totally skewing the marker with respect to this triple we will get three pairs of equal expressions, except for the minus signs that appear when skewing by a transposition. Since each pair contains exactly one transposition each pair will cancel out and we will be left with zero.

It is not hard to see that there can be no other types of zero markers. There are three odd and three even permutations in S_3. In a zero marker the odd and even permutations cancel each other out. That leaves us with only two possibilities: either the marker is invariant under all permutations with respect to some triple (type 1), or the permuted expressions with respect to some triple cancel each other in pairs (type 2).

In both cases, the fact that totally skewing a marker with respect to just one of the three triples results in zero is enough for the entire total skew-symmetrization to become zero. Totally skewing a marker essentially means totally skewing it with respect to the first triple, then skewing the resulting 6 monomials with respect to the second triple and finally skewing the resulting 36 monomials with respect to the first triple, resulting in 216 monomials. If one of these three total skews results in zero, the result will obviously be zero.

We distinguished two types of markers whose total skew-symmetrization is zero. The first type is invariant under all permutations in S_3 for some triple, while the second type is invariant under just one transposition in S_3 for some triple. Thus if a marker is invariant under just one transposition in S_3, for any of the three triples, the marker must be of one of these types, and thus a zero marker. $\square$

Ex 3. Let's look at an example of a zero marker:

$$6a_{v1v2w1}\rho_{v3}\rho_{w2}\rho_{w3}a_{u1}a_{u2}a_{u3} \tag{13}$$

This marker is invariant under the transposition (23) with respect to the first triple. Applying the permutation (23) to the first triple means exchanging the variables $v1$ and $w1$, but since these variables are part of the same subscript, exchanging them does not change the marker-monomial. This means that when we totally skew this marker with respect to the first triple, the resulting monomials will cancel out and we will get zero. Consequently, this marker is a zero marker. $\triangle$

Ex 4. Another slightly different example of a zero marker is given by:

$$12\rho a_{u1}a_{u2v1}a_{u3w1}\rho_{v2}\rho_{v3}\rho_{w2w3} \tag{14}$$

This marker is again invariant under the transposition (23) with respect to the first triple, but in this case the invariance only becomes apparent once we place back the original independent variables. To see this, note that both $a_{u2v1}a_{u3w1}$ and $a_{u2w1}a_{u3v1}$ will become $a_{xy}a_{xz}$ when we place back the original variables, as the order of multiplication doesn't matter. △

Def 6. *A nonzero marker is a marker whose total skew-symmetrization is not equal to zero.*

Most monomials in $\dot{a}$ and $\dot{\rho}$ can produce multiple different nonzero markers. That means that the (at most) 216 possible different markers we can construct from any monomial in $\dot{a}$ or $\dot{\rho}$ usually include multiple different nonzero markers. Theoretically, there could be monomials that cannot produce a nonzero marker by any transformation. Take for instance the monomial

$$-2a_x^3 a_y^3 \rho_{zz} a_z \tag{15}$$

No matter which way we partition the variables into three triples, the resulting marker will always be invariant under the permutation (12) with respect to any of the three triples.

Remark. The polynomials $\dot{a}$ and $\dot{\rho}$ do not contain any monomials that cannot produce a nonzero marker. This can easily be shown using the procedure *ConstructNonzeroMarker* (see section 4.3).

3.2 The Structure of Monomials

In order to study the properties of the different monomials contained in $\dot{a}$ and $\dot{\rho}$ we introduce a new concept called the structure of a monomial.

Def 7. *The structure of a (marker-)monomial is determined by the number of derivatives of a and ρ that occur in the monomial, and the way they are grouped. The structure is denoted by $a[r_1...r_n]\rho[s_1...s_m]$, where $r, s, n, m \in \mathbb{N}_{>0}$ with $0 < r_1 \leq ... \leq r_n$ and $0 < s_1 \leq ... \leq s_m$. Here r and s denote the number of derivatives of a and ρ respectively. This means the monomial has $\sum r_n$ derivatives of a and $\sum s_m$ derivatives of ρ. Moreover, these derivatives occur in "blocks", as follows: the monomial contains the dependent variable a n-times, once with r_1 number of derivatives, once with r_2 number of derivatives, etc. Similarly for ρ and s.*

Ex 5. Let's have a look at some monomials and determine their structure:

$$
\begin{aligned}
&- 12\rho^2 a_x \rho_y a_{xy} a_{zz} a_{xyz} \\
&- 6\rho\rho_y^2 a_z^2 a_{xz} \rho_{xxy}
\end{aligned} \tag{16}
$$

The first monomial contains a four times and ρ one time. Note that we ignore the ρ^2 coefficient. It not important for the structure of the monomial since it does not contain any derivative. The first a has one derivative, the second and third a have two derivatives and the last a has three derivatives. The only ρ that has a derivative has one derivative. Consequently the structure of the first monomial is $a[1223]\rho[1]$.

Similarly, the structure of the second monomial is $a[112]\rho[113]$. Note that we again ignore the ρ coefficient without any derivatives. Also note that $a_z^2 = a_z \cdot a_z$, which results in the number 1 appearing twice in the structure. △

Note that our expressions $\dot{a}$ and $\dot{\rho}$ exclusively contain terms with exactly nine derivatives, as mentioned in chapter 2. This means that the numbers in the structure of these monomials will always add up to nine.

Ex 6. The structure of a marker can be determined in exactly the same way as the structure of the monomials in the previous example. Take for instance the marker

$$
6a_{v1v2w1}\rho_{v2}\rho_{v3}\rho_{w3}a_{u1}a_{u2}a_{u3} \tag{17}
$$

Its structure is given by $a[1113]\rho[111]$. △

Remark. There can be at most 1860 different monomials of any given structure containing three triples of derivatives. This result can be computed using combinatorics. The first variable appears three times in nine positions; this can happen in $9!/3!6! = 84$ ways. The second variable appears three times in six positions; this can happen in $6!/3!3! = 20$ ways. For the last variable there is only one option left. The maximum number of possible combinations is thus $84 \cdot 20 = 1860$. This is not a sharp estimate, as it does not take the differential order constraints into account. However, this will be useful for computing all possible monomials of a given structure in section 4.5.

Note that the total skew-symmetrization of a marker only contains monomials of the same structure as the structure of the marker itself, since permuting the variables in a marker does not change the number of derivatives in the marker or the way these derivatives are grouped. Consequently any monomial in $\dot{a}$ or $\dot{\rho}$ can only be represented by a marker of its own structure. At the very least this tells us that if we want to represent $\dot{a}$ or $\dot{\rho}$ by a sum of total skew-symmetrizations, we need at least as many markers as there are different structures in $\dot{a}$ or $\dot{\rho}$ to achieve this.

Another consequence is that the problem of finding $\dot{a}''$ and $\dot{\rho}''$, as stated in section 2.4, can be split into a number of sub-problems in the following way: we can divide all monomials in $\dot{a}$ and $\dot{\rho}$ into a number of new polynomials, each containing only monomials of equal structure. This way, we can look at each of these new polynomials separately, and try to find a way to represent them as the sum of some total skew-symmetrizations.

3.3 A Brute Force Algorithm

Using the insights gained in this chapter we can suggest an approach to solve the problem stated in section 2.4. First of all, as suggested in section 3.2, we would like to divide all monomials in $\dot{a}$ and $\dot{\rho}$ into new polynomials consisting only of monomials of equal structure. For each of these new polynomials, we could try a brute force algorithm to look for suitable markers whose sum of total skew-symmetrizations are equal to these polynomials. The box below outlines a naive and greedy algorithm that could solve this problem.

1. Take a monomial from the (remainder of the) given polynomial
2. Construct a nonzero marker from this monomial
3. Totally skew the marker and place the original variables back
4. Find the monomial from step 1 in the total skew-symmetrization and compare their coefficients
5. Subtract a suitable multiple of the total skew-symmetrization from the original polynomial to make it smaller
6. If we find a combination of markers whose total skew-symmetrization produces the original polynomial we are done. Else, repeat steps 1-5

In step 4 we compare the coefficient of the monomial from step 1 with the coefficient of the same monomial in the total skew-symmetrization. It could happen, by cancellation of terms, that the original monomial is not part of the total skew-symmetrization. In that case we simply skip this marker by going back to step 2 and taking a different nonzero marker. If all nonzero markers that can be created from the monomial in step 2 are exhausted we go back to step 1 and take another monomial.

In step 5 we subtract a suitable multiple of the total skew-symmetrization from the polynomial, such that the original monomial cancels out. We are then left with a remainder of the polynomial we started with. When we go back to step 1 we pick our new monomial from this remainder, not from the original polynomial.

Finally, note that this algorithm is not a true algorithm, as it does not have any stopping criteria. After step 5, when we subtract a suitable multiple of our total skew-symmetrization from the (remainder of the) polynomial, there is no guarantee that our polynomial indeed becomes smaller. It could happen that by subtracting this total skew-symmetrization we are adding more new terms than we are cancelling out. This way our algorithm could easily end up in a loop. However, it turns out that this algorithm is capable of entirely solving the problem of finding $\dot{a}''$ and $\dot{\rho}''$. It remains to be seen why this is the case.

4 Implementation in Maple for 3D

In this chapter we implement the approach suggested in chapter 3 in Maple [8]. We provide a procedure for splitting a polynomial into polynomials containing only monomials of equal structure, as discussed in section 3.2. Next, we implement the algorithm from section 3.3. To do this we need a number of auxiliary procedures. Finally, we provide some procedures for determining the uniqueness of solutions. For the applications of these procedures see chapter 6.

4.1 Splitting Polynomials by Structure

4.1.1 MonomialStructure

We would like a procedure that splits polynomials into polynomials containing only monomials of equal structure. The first step is to have a simple procedure that determines the structure of a single monomial. The following procedure, *MonomialStructure*, does exactly that. It works by counting the number of appearances of x, y and z in the monomial, and noting to which dependent variable they are attached.

```
MonomialStructure := proc(mon)
#determines the structure of a monomial

    # loads the Algebraic and StringTools packages
    with(Algebraic):
    with(StringTools):

    # defines all used variables
    local derivativesA, derivativesRho, VarsList, i, Var, Power,
         j, derivatives, structure;

    # initializes the vectors
    derivativesA := "":
    derivativesRho := "":

    # makes a list of all the variables and its powers
    VarsList := Squarefree(mon):

    # makes two lists of the number of derivatives with respect
       to a and rho
    for i from 1 to numelems(VarsList[2]) do
        Var := convert(VarsList[2,i,1],string):
        Power := VarsList[2,i,2]:

```

```
        for j from 1 to Power do
            if Has(Var,"a") then
                derivatives := CountCharacterOccurrences(Var, "x") + CountCharacterOccurrences(Var, "y") + CountCharacterOccurrences(Var, "z"):
                derivativesA := cat(derivativesA, derivatives):
            else
                derivatives := CountCharacterOccurrences(Var, "x") + CountCharacterOccurrences(Var, "y") + CountCharacterOccurrences(Var, "z"):
                derivativesRho := cat(derivativesRho, derivatives):
            end if:
        od:
    od:

    # sorts the derivatives in ascending order and removes zeroes
    derivativesA := Sort(derivativesA):
    derivativesRho := Sort(derivativesRho):
    derivativesRho := Subs("0" = "", derivativesRho):

    # combines the lists into one string
    structure := cat("a[", derivativesA, "]rho[", derivativesRho, "]"):

end proc:
```

For example, we could try to compute the structure of a monomial in the following way:

```
MonomialStructure(a_x^3*rho_y^3*a_zzz)
                                                  a[1113]rho[111]
```

4.1.2 SplitStructures

The procedure *SplitStructures* splits a given polynomial into a number of vectors, where each vector contains only monomials of equal structure. It works by computing the structure of each monomial using *MonomialStructure*, and then sorting the monomials into the appropriate vectors.

```
SplitStructures := proc(poly)
# splits a given polynomial poly into monomials of equal structure

    # loads the ArrayTools, ListTools and LinearAlgebra packages
```

```
    with(ArrayTools):
    with(ListTools):
    with(LinearAlgebra):

    # defines all used variables
    local polyVector, Split, structureTypes, i, structure, column, SplitAdd, j;

    # converts the polynomial to a vector
    polyVector := convert(poly, list):

    # defines the outputs
    Split := Matrix(numelems(polyVector)):
    structureTypes := Vector([]):

    # computes the structure of each polynomial term and sorts it into the correct column of Split
    for i from 1 to numelems(polyVector) do
        structure := MonomialStructure(polyVector[i]):

        if has(structureTypes, structure) = false then
            structureTypes := Concatenate(1, structureTypes, structure):
            structureTypes := Vector([structureTypes]):
        end if:

        column := ListTools[Search](structure, structureTypes):
        Split[i,column] := polyVector[i]:
    od:

    # deletes all zero columns from Split
    Split := DeleteColumn(Split, numelems(structureTypes)+1..numelems(polyVector)):

    # sums all monomials of the same structure
    SplitAdd := Vector(numelems(structureTypes)):
    for j from 1 to numelems(structureTypes) do
        SplitAdd[j] := add(Column(Split,j)):
    od:

    # outputs a vector containing monomials of equal structure in the same element
    return SplitAdd;

end proc:
```

4.2 Skewing and Totally Skewing

4.2.1 Skew

The procedure *Skew* is a simple procedure that skews a polynomial. This procedure takes as input a polynomial, a permutation in S_3 and a vector containing the three variables that need to be skewed, and outputs the skewed polynomial. It first converts the polynomial expression and given variables to strings and then replaces the given variables by their permuted variant. Finally, the procedure reorders the variables so that they are again in alphabetical order, as skewing can change the order of the variables, and converts the polynomial back to an expression.

```
Skew := proc(x, perm, vars)
# skew-symmetrizes the variables in the vector vars in a
    polynomial x by permutation perm

    # loads the GroupTheory and StringTools packages
    with(GroupTheory):
    with(StringTools):

    # defines all used variables
    local permSign, permInverse, xString, varString, j, i, xExpr
       ;

    # computes the sign and inverse of the permutation
    permSign := PermParity(perm):
    permInverse := PermInverse(perm):

    # converts the polynomial and variables to strings
    xString := convert(x, string):
    varString := Vector(3):
    for j from 1 to 3 do
        varString[j] := convert(vars[j],string):
    od:

    # substitutes the variables by the permuted variables
    xString := Subs({varString[1]=varString[permInverse[1]],
        varString[2]=varString[permInverse[2]],varString[3]=
        varString[permInverse[3]]}, xString):

    # reorders the variables in the permuted polynomial to their
         original order
    for i from 1 to 2 do
        xString := RegSubs(cat(varString[2],varString[1]) = cat(
            varString[1],varString[2]), xString):
```

```
        xString := RegSubs(cat(varString[3],varString[1]) = cat(
            varString[1],varString[3]), xString):
        xString := RegSubs(cat(varString[3],varString[2]) = cat(
            varString[2],varString[3]), xString):
    od:

    # applies the permutation sign to the expression
    xExpr := parse(xString):
    xExpr := permSign * xExpr;

end proc:
```

We can test the *Skew* procedure by skewing the polynomials $\dot{a}$ and $\dot{\rho}$. The result should satisfy the equations (4). For example, we can try permuting $\{x, y, z\}$ in $\dot{a}$ by permutation (12) and then subtracting $\dot{a}$.

```
Skew(adot, Perm([[1, 2]]), [x,y,z]) - adot
                                                                  0
```

We confirm the result is zero, as previously discovered by Buring and Kiselev [6]. This holds for both $\dot{a}$ and $\dot{\rho}$, and for all permutations in S_3.

4.2.2 SkewTotal

The next procedure, *SkewTotal*, is a procedure that computes the total skew-symmetrization of a polynomial. This procedure takes as input a polynomial and a vector containing the three variables that need to be skewed, and outputs the totally skew-symmetrization of the given polynomial with respect to the given variables. It uses our previous procedure, *Skew*, to skew the polynomial by each permutation in S_3.

```
SkewTotal := proc(poly, vars)
# skews a polynomial totally in S3 with respect to the variables
    in vars

    # loads the GroupTheory package
    with(GroupTheory):

    # defines all used variables
    local S3, xSkewTotal, i;
    xSkewTotal := poly:

    # defines the nontrivial permutations in the symmetric group
        S3
```

```
    S3 := Vector([Perm([[1, 2]]), Perm([[1, 3]]), Perm([[2, 3]])
       , Perm([[1, 2, 3]]), Perm([[1, 3, 2]])]):

    # permutes the polynomial by permutation i and adds result
       to xSkewTotal
    for i from 1 to 5 do
        xSkewTotal := xSkewTotal + Skew(poly,S3[i],vars):
    od:

end proc:
```

We can test the *SkewTotal* procedure by totally skewing the polynomials $\dot{a}'$ and $\dot{\rho}'$. The result should satisfy equations (5). For example, we can compute the total skew-symmetrization of $\dot{a}'$ with respect to the variables $\{x, y, z\}$ and then subtract $\dot{a}$.

```
SkewTotal(adotUnskew, [x,y,z]) - adot
                                                    0
```

We again confirm the result is zero, as previously discovered by Buring and Kiselev [7]. This also holds for $\dot{\rho}'$ and $\dot{\rho}$.

4.3 Working with Markers

4.3.1 ExpandPowers

The following procedure, *ExpandPowers*, takes as input a monomial and outputs a string containing the same monomial, but with all powers written as simple multiplications. We need this procedure in order to construct marker-monomials.

```
ExpandPowers := proc(mon)
# given a monomial outputs a string with all powers written as
    multiplications

    # load the Algebraic package
    with(Algebraic):

    # defines all used variables
    local Expanded, Coeff, VarsList, NumVars, monString, i,
       VarString, Power, j;

    # expands the monomial, returning all coefficients,
       variables and powers
    Expanded := Squarefree(mon):
```

```
    Coeff := convert(Expanded[1],string):
    VarsList := Expanded[2]:
    NumVars := numelems(VarsList):
    monString := "":

    # writes the monomial as a string with powers written as
       multiplications
    for i from 1 to NumVars do
        VarString := convert(VarsList[i,1],string):
        Power := VarsList[i,2]:
        for j from 1 to Power do
            monString := cat(monString,"*",VarString):
        od:
    od:
    monString := cat(Coeff,monString);

end proc:
```

For instance, let's take a simple monomial containing some powers and write it as a string using only multiplications.

```
ExpandPowers(a_x*a_y^2*rho_z^3)
                                "a_x*a_y*a_y*rho_z*rho_z*rho_z"
```

4.3.2 Triples2Variables

The procedure *Triples2Variables* converts markers back to monomials of $\{x, y, z\}$ by replacing $\{u_n\}$ by x, $\{v_n\}$ by y and $\{w_n\}$ by z. After replacing the variables, it also reorders $\{x, y, z\}$ alphabetically. This is necessary because skewing markers can change the variable ordering.

```
Triples2Variables := proc(poly)
# replaces the triple variables in a polynomial poly by the
   original variables

    # loads the StringTools package
    with(StringTools):

    # defines all used variables
    local vars, triples, xString, i, j, k, xExpr;

    # defines the variables and new triple names
    vars := [x,y,z]:
    triples := Matrix([[u1,v1,w1],[u2,v2,w2],[u3,v3,w3]]):

    # replaces the triple variables by the original variables
```

```
    xString := convert(poly,string):
    for i from 1 to 3 do
        for j from 1 to 3 do
            xString := RegSubs(convert(triples[i,j],string) =
                convert(vars[j],string), xString):
        od:
    od:

    # reorders the variables in the permuted polynomial to their
        original order
    for i from 1 to 2 do

        xString := RegSubs(convert(cat(vars[2],vars[1]),string)
            = convert(cat(vars[1],vars[2]),string), xString):
        xString := RegSubs(convert(cat(vars[3],vars[1]),string)
            = convert(cat(vars[1],vars[3]),string), xString):
        xString := RegSubs(convert(cat(vars[3],vars[2]),string)
            = convert(cat(vars[2],vars[3]),string), xString):
    od:
    xExpr := parse(xString);

end proc:
```

For example, let's try to replace the triple variables in a marker by the original variables:

```
Triples2Variables(-12*rho^2*a_u1*rho_v1*a_v2w2*a_u2w1*a_u3v3w3)
                                 -12*rho^2*a_x*rho_y*a_yz*a_xz*a_xyz
```

4.3.3 VerifyMarker

For the construction of nonzero markers we would like to be able to check efficiently whether a marker is a nonzero marker. The procedure *VerifyMarker* uses Theorem 1 to check if a given marker contains an invariance. It skews the marker by all transpositions in S_3 with respect to all three triples, and then subtracts the original marker to see if there is an invariance. If the marker contains an invariance it returns 0. Else, it returns 1.

```
VerifyMarker := proc(marker)
# checks a marker for invariances

    # defines all used variables
    local triples, Mon, i;

    # defines the triples
```

```
    triples := Matrix([[u1,v1,w1],[u2,v2,w2],[u3,v3,w3]]):

    # converts the marker to a monomial
    Mon := Triples2Variables(marker):

    # checks invaraince for all transpisitions in all triples
    for i from 1 to 3 do
        if Mon + Triples2Variables(Skew(marker, Perm([[1, 2]]),
            triples[i])) = 0 then
            return 0;
        elif Mon + Triples2Variables(Skew(marker, Perm([[1, 3]])
            , triples[i])) = 0 then
            return 0;
        elif Mon + Triples2Variables(Skew(marker, Perm([[2, 3]])
            , triples[i])) = 0 then
            return 0;
        end if:
    od:

    # outputs 0 if the marker has an invariance, or else 1
    return 1;

end proc:
```

4.3.4 SkewTotalMarker

The procedure *SkewTotalMarker* computes the total skew-symmetrization of a given marker, with the original variables $\{x, y, z\}$ placed back. It uses *SkewTotal* three times, once for each triple, to produce all 216 skew combinations (see chapter 2.3). Then it uses *Triples2Variables* to replace the triple variables by the original variables.

```
SkewTotalMarker := proc(marker)
# skews markers totally with respect to all three triples

    # defines all used variables
    local triples, skew1, skew2, skew3;

    # defines the triple variables
    triples := Matrix([[u1,v1,w1],[u2,v2,w2],[u3,v3,w3]]):

    # totally skews the markers with respect to the first triple
    skew1 := SkewTotal(marker,triples[1]):

    # totally skews the markers with respect to the second
       triple
```

```
    skew2 := SkewTotal(skew1,triples[2]):

    # totally skews the markers with respect to the third triple
    skew3 := SkewTotal(skew2,triples[3]):

    # replaces the triple variables by the original variables
    skew3 := Triples2Variables(skew3):

end proc:
```

4.3.5 ConstructNonzeroMarker

Finally, we have a procedure that constructs a nonzero marker from a given monomial. *ConstructNonzeroMarker* partitions the nine subscripts in a given monomial into three triples, in all 216 possible ways. To ensure that every variable will be replaced, it uses the *ExpandPowers* procedure before replacing the variables. After each marker construction it uses *VerifyMarker* to check whether we found a nonzero marker. The first encountered nonzero marker is given as output.

```
ConstructNonzeroMarker := proc(mon)
# given a monomial mon constructs a nonzero marker

    # loads the StringTools and combinat package
    with(StringTools):
    with(combinat):

    # defines all used variables
    local xTriples, yTriples, zTriples, monExpanded, marker, i,
        j, k, markerExpr, skewedMarker, skewed;

    # defines the triples and all their possible permutations
    xTriples := permute([u1,u2,u3]):
    yTriples := permute([v1,v2,v3]):
    zTriples := permute([w1,w2,w3]):

    # converts the monomial to a string with the powers written
        as multiplications
    monExpanded := ExpandPowers(mon):

    # goes through all 216 possible markers
    for i from 1 to 6 do
    for j from 1 to 6 do
    for k from 1 to 6 do
        marker := monExpanded:
```

```
        marker := Substitute(marker,"x",convert(xTriples[i,1],
            string)):
        marker := Substitute(marker,"x",convert(xTriples[i,2],
            string)):
        marker := Substitute(marker,"x",convert(xTriples[i,3],
            string)):
        marker := Substitute(marker,"y",convert(yTriples[j,1],
            string)):
        marker := Substitute(marker,"y",convert(yTriples[j,2],
            string)):
        marker := Substitute(marker,"y",convert(yTriples[j,3],
            string)):
        marker := Substitute(marker,"z",convert(zTriples[k,1],
            string)):
        marker := Substitute(marker,"z",convert(zTriples[k,2],
            string)):
        marker := Substitute(marker,"z",convert(zTriples[k,3],
            string)):

        # checks whether we found a nonzero marker
        if VerifyMarker(marker) = 1 then
            markerExpr := parse(marker):
            return markerExpr;
        end if:
    od:
    od:
    od:
    return 0;

end proc:
```

4.4 The Algorithm

Using the procedures from section 4.2 and 4.3 we can construct the algorithm suggested in section 3.3. This algorithm is implemented in the procedure *ConstructSolution*. It takes a polynomial as input and tries to compute a marker-polynomial whose total skew-symmetrization is equal to the given polynomial.

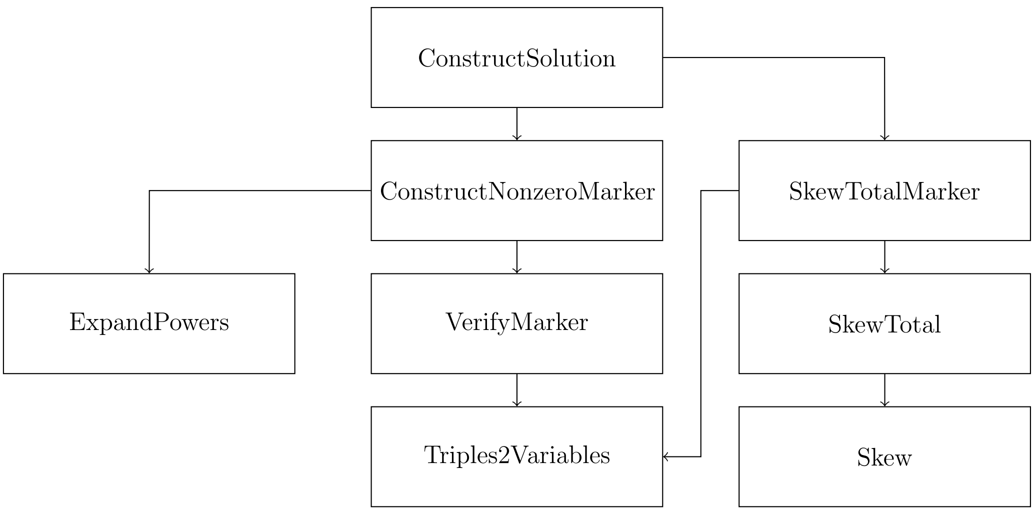
It takes the first monomial of the given polynomial and constructs a nonzero marker from this monomial using *ConstructNonzeroMarker*, as well as its total skew-symmetrization using *SkewTotalMarker*. Next it makes a version of the monomial and the total skew-symmetrization without coefficients, to find out if and where the monomial appears in its total skew-symmetrization. If the monomial doesn't appear in the total skew-symmetrization, it is skipped and the algorithm moves on to the next monomial. If it does appear in the total skew-symmetrization, it computes the factor between their coefficients.

Then, the algorithm subtracts a multiple of the total skew-symmetrization from the polynomial, using the coefficient factor it just computed. The loop is repeated, using a monomial from the remaining polynomial, until the polynomial becomes zero. The output is a sum of all the used markers. The sum of their total skew-symmetrizations will be equal to the polynomial the algorithm started with. Again, note that this algorithm has no stopping criteria. It simply tries a number of markers naively, until it (hopefully) finds a collection of markers that solves the problem.

```
ConstructSolution := proc(poly)
# given a polynomial poly looks for a representation of the
    polynomial by the total skew-symmetrization of markers

    # defines all used variables
    local polySum, usedMarkers, mon, MarkAndSkew, monCoeff,
        monNoCoeff, marker, skewed, skewedVector, skewedCoeffs,
        skewedNoCoeffs, coeffsFactor, i;

    # copies the polynomial and defines the output
    polySum := poly:
    usedMarkers := 0:

    # initializes the counter
    i:=1:

    # loops until we find a representation for the polynomial
    while polySum <> 0 do

        # takes the first monomial from the polynomial
        mon := convert(polySum, list)[i]:

        # constructs a nonzero marker and its total skew-
            symmetrization
        marker := ConstructNonzeroMarker(mon):
        if marker = 0 then
            i:=i+1:
            next
        end if:
        skewed := SkewTotalMarker(marker):

        # makes a version of the monomial without the
            coefficient
        monCoeff := coeffs(mon):
```

```
        monNoCoeff := mon *~ monCoeff^~(-1):

        # makes a vector of the skewed expression without coefficients
        skewedVector := Vector([convert(skewed, list)]):
        skewedCoeffs := Vector([coeffs(skewed)]):
        skewedNoCoeffs := skewedVector *~ skewedCoeffs^~(-1):

        # compares the coefficients between the equal monomials
        if member(monNoCoeff, skewedNoCoeffs, 'position') = false then
            i:=i+1:
            next
        end if:
        coeffsFactor := monCoeff / skewedCoeffs[position]:

        # subtracts a multilple of the skewed expression from the polynomial
        polySum := polySum - coeffsFactor * skewed:
        usedMarkers := usedMarkers + coeffsFactor * marker:

        # resets the counter
        i:=1:
    od:

    return usedMarkers;

end proc:
```

The following diagram illustrates the way in which all auxiliary procedures used in the algorithm are connected.



4.5 Verifying Uniqueness

Now that we have all the needed procedures to implement the approach outlined in chapter 3, we provide some procedures that can be used to determine the uniqueness of solutions found by *ConstructSolution*.

4.5.1 ConstructAllMonOfStruc

The procedure *ConstructAllMonOfStruc* constructs a vector containing all possible monomials of the same structure as the given monomial. As noted in section 3.2, there can be at most 1860 different markers of any given structure. This procedure runs over all 1860 possibilities. For each possible monomial, it checks whether this monomial was already found at an earlier iteration. If not, it is added to the output.

```
ConstructAllMonOfStruc := proc(mon)
# Constructs a vector containing all monomials of the same
    structure as the given monomial

    # loads the StringTools, ArrayTools and combinat packages
    with(StringTools):
    with(ArrayTools):
    with(combinat):

    # defines all used variables
    local structuresList, monExpanded, monEmpty, varPermutations
        , i, j, k, monTest, monExpr;

    # defines the output vector
    structuresList := Vector([]):

    # changes all independent vars in the given monomial to "c"
    monExpanded := ExpandPowers(mon):
    monEmpty := Subs({"x"="c","y"="c","z"="c"}, monExpanded):

    # lists all possible permutations of "xxxyyyzzz"
    varPermutations := permute([x,x,x,y,y,y,z,z,z]):

    # constructs a monomial for each variable permutation
    for i from 1 to 1680 do
        monTest := monEmpty:
        for j from 1 to 9 do
            monTest := Substitute(monTest, "c", convert(
                varPermutations[i,j],string)):
        od:

```

```
        # reorders the variables in the permuted polynomial to
            their original order
        for k from 1 to 2 do
            monTest := RegSubs("yx" = "xy", monTest):
            monTest := RegSubs("zx" = "xz", monTest):
            monTest := RegSubs("zy" = "yz", monTest):
        od:
        monExpr := parse(monTest):

        # checks if this structure is already in the list, if
            not is is added
        if has(structuresList,monExpr) = false then
            structuresList := Concatenate(1,structuresList,
                monExpr):
            structuresList := Vector([structuresList]):
        end if:
    od:

    return structuresList;

end proc:
```

4.5.2 SortMarkerVariables

The procedure *SortMarkerVariables* sorts the triple variables in a given marker alphabetically. We will need this in order to compare whether two markers are equal. After all, the order of differentiation doesn't matter. This procedure works by simply replacing all incorrect orderings by their correct variant.

```
SortMarkerVariables := proc(marker)
# sorts the indepdent variables in a marker alphabetically

    # loads the StringTools package
    with(StringTools):

    # defines all used variables
    local xString, i, markerExpr;

    # converts the marker to a string
    xString := convert(marker,string):

    # reorders the variables pairwise
    for i from 1 to 2 do
        xString := RegSubs("s2s1" = "s1s2", xString):
        xString := RegSubs("s3s1" = "s1s3", xString):
```

```
        xString := RegSubs("s3s2" = "s2s3", xString):
        xString := RegSubs("t2t1" = "t1t2", xString):
        xString := RegSubs("t3t1" = "t1t3", xString):
        xString := RegSubs("t3t2" = "t2t3", xString):
        xString := RegSubs("u2u1" = "u1u2", xString):
        xString := RegSubs("u3u1" = "u1u3", xString):
        xString := RegSubs("u3u2" = "u2u3", xString):

        xString := RegSubs("t1s1" = "s1t1", xString):
        xString := RegSubs("t2s1" = "s1t2", xString):
        xString := RegSubs("t3s1" = "s1t3", xString):
        xString := RegSubs("t1s2" = "s2t1", xString):
        xString := RegSubs("t2s2" = "s2t2", xString):
        xString := RegSubs("t3s2" = "s2t3", xString):
        xString := RegSubs("t1s3" = "s3t1", xString):
        xString := RegSubs("t2s3" = "s3t2", xString):
        xString := RegSubs("t3s3" = "s3t3", xString):

        xString := RegSubs("u1s1" = "s1u1", xString):
        xString := RegSubs("u2s1" = "s1u2", xString):
        xString := RegSubs("u3s1" = "s1u3", xString):
        xString := RegSubs("u1s2" = "s2u1", xString):
        xString := RegSubs("u2s2" = "s2u2", xString):
        xString := RegSubs("u3s2" = "s2u3", xString):
        xString := RegSubs("u1s3" = "s3u1", xString):
        xString := RegSubs("u2s3" = "s3u2", xString):
        xString := RegSubs("u3s3" = "s3u3", xString):

        xString := RegSubs("u1t1" = "t1u1", xString):
        xString := RegSubs("u2t1" = "t1u2", xString):
        xString := RegSubs("u3t1" = "t1u3", xString):
        xString := RegSubs("u1t2" = "t2u1", xString):
        xString := RegSubs("u2t2" = "t2u2", xString):
        xString := RegSubs("u3t2" = "t2u3", xString):
        xString := RegSubs("u1t3" = "t3u1", xString):
        xString := RegSubs("u2t3" = "t3u2", xString):
        xString := RegSubs("u3t3" = "t3u3", xString):
    od:

    # outputs the sorted marker
    markerExpr := parse(xString):
    return markerExpr;

end proc:
```

4.5.3 ConstructAllNonzeroMarkers

This procedure is very similar to the procedure *ConstructNonzeroMarker* from section 4.3.5, and works in much the same way. However, it takes as input a list of monomials instead of just one monomial, and instead of stopping at the first nonzero marker it finds it constructs a vector containing all nonzero markers that can be constructed from the monomials in the given list. It uses the procedure *SortMarkerVariables* to determine whether two markers are equal.

```
ConstructAllNonzeroMarkers := proc(monList)
# given a list of monomials monList constructs all nonzero markers from these monomials

    # loads the StringTools, combinat and ArrayTools packages
    with(StringTools):
    with(combinat):
    with(ArrayTools):

    # defines all used variables
    local xTriples, yTriples, zTriples, nonzeroMarkers, monExpanded, marker, n, i, j, k, markerExpr;

    # defines the triples and all their possible permutations, and the output vector
    xTriples := permute([s1,s2,s3]):
    yTriples := permute([t1,t2,t3]):
    zTriples := permute([u1,u2,u3]):
    nonzeroMarkers := Vector([]):

    # goes through all given monomials
    for n from 1 to numelems(monList) do
    monExpanded := ExpandPowers(monList[n]):
    print(n);

    # goes through all 216 possible markers
    for i from 1 to 6 do
    for j from 1 to 6 do
    for k from 1 to 6 do
        marker := monExpanded:
        marker := Substitute(marker, "x", convert(xTriples[i,1], string)):
        marker := Substitute(marker, "x", convert(xTriples[i,2], string)):
        marker := Substitute(marker, "x", convert(xTriples[i,3], string)):
        marker := Substitute(marker, "y", convert(yTriples[j,1],
```

```
            string)):
        marker := Substitute(marker, "y", convert(yTriples[j,2],
            string)):
        marker := Substitute(marker, "y", convert(yTriples[j,3],
            string)):
        marker := Substitute(marker, "z", convert(zTriples[k,1],
            string)):
        marker := Substitute(marker, "z", convert(zTriples[k,2],
            string)):
        marker := Substitute(marker, "z", convert(zTriples[k,3],
            string)):

        # checks whether we found a nonzero marker; if so it is
            added to the output
        if VerifyMarker(marker) = 1 then
            markerExpr := SortMarkerVariables(marker):
            if member(markerExpr,nonzeroMarkers) = false then
                nonzeroMarkers := Concatenate(1,nonzeroMarkers,
                    markerExpr):
                nonzeroMarkers := Vector([nonzeroMarkers]):
            end if:
        end if:
    od:
    od:
    od:
    od:

    return nonzeroMarkers;

end proc:
```

4.5.4 SkewTotalAndCompare

Finally, we have the procedure *SkewTotalAndCompare*. This procedure takes as input a list of markers and a polynomial. For each marker on the list, it checks whether its total skew-symmetrization is a multiple of the given polynomial. It uses the procedure *SkewTotalMarker* from section 4.3.4 to compute the total skew-symmetrizations. This procedure is used in chapter 6 to verify the uniqueness of solutions.

```
SkewTotalAndCompare := proc(markerList,poly)
# given a list of markers and a polynomial checks whether the
    total skew-symmetrization of every marker in the list is a
    multiple of the given polynomial

    # defines all used variables
```

```
    local testMon, testMonCoeff,testMonNoCoeff, i, skewed,
       skewedVector, skewedCoeffs, skewedNoCoeffs, coeffsFactor;

    # takes a monomial from the polynomial as a test and
       constructs a version without coefficient
    testMon := convert(poly,list)[1]:
    testMonCoeff :=  coeffs(testMon):
    testMonNoCoeff := testMon * testMonCoeff^(-1):

    # goes trough all markers in the list
    for i from 1 to numelems(markerList) do

        # computes the total skew-symmetrization of a marker and
            makes a version without coefficients
        skewed := SkewTotalMarker(markerList[i]):
        skewedVector := convert(skewed,list):
        skewedCoeffs := Vector([coeffs(skewed)]):
        skewedNoCoeffs := skewedVector *~ skewedCoeffs^~(-1):

        # checks if the test monomial is part of the total skew-
           symmetrization
        if member(testMonNoCoeff, skewedNoCoeffs, 'position') =
           false then
            printf("Total skew-symmetrization is not a multiple
               for marker %a \n", markerList[i]);
            next
        end if:

        # checks if the total skew-symmetrization is a multiple
           of the given polynomial
        coeffsFactor := testMonCoeff / skewedCoeffs[position]:
        if poly - coeffsFactor * skewed <> 0 then
            printf("Total skew-symmetrization is not a multiple
               for marker %a \n", markerList[i]);
            next
        end if:
    od:

end proc:
```

5 Implementation in Maple for 4D

This chapter contains the same procedures that can be found in chapter 4, but modified to work in 4 dimensions instead of 3. However, the procedures for verifying the uniqueness of solutions found in section 4.5 are not included for the 4-dimensional case. For details about the workings of these procedures and for some examples please refer to their 3-dimensional counterparts in chapter 4. These procedures will be applied to some evolution equations obtained by deforming 4d-determinant Nambu-Poisson brackets, to find out whether the properties found in 3 dimensions generalize to 4 dimensions.

5.1 Splitting Polynomials by Structure

```
MonomialStructure4D := proc(mon)
#determines the structure of a monomial

    # loads the Algebraic and StringTools packages
    with(Algebraic):
    with(StringTools):

    # defines all used variables
    local derivativesA0, derivativesA1, derivativesRho, VarsList
       , i, Var, Power, j, derivatives, structure;

    # initializes the vectors
    derivativesA0 := "":
    derivativesA1 := "":
    derivativesRho := "":

    # makes a list of all the variables and its powers
    VarsList := Squarefree(mon):

    # makes two lists of the number of derivatives with respect
       to a and rho
    for i from 1 to numelems(VarsList[2]) do
        Var := convert(VarsList[2,i,1],string):
        Power := VarsList[2,i,2]:

        for j from 1 to Power do
            if Has(Var,"a0") then
                derivatives := CountCharacterOccurrences(Var,"x
                   ")+CountCharacterOccurrences(Var,"y")+
                   CountCharacterOccurrences(Var,"z")+
                   CountCharacterOccurrences(Var,"w"):
```

```
                derivativesA0 := cat(derivativesA0, derivatives):
            elif Has(Var,"a1") then
                derivatives := CountCharacterOccurrences(Var,"x")+CountCharacterOccurrences(Var,"y")+CountCharacterOccurrences(Var,"z")+CountCharacterOccurrences(Var,"w"):
                derivativesA1 := cat(derivativesA1, derivatives):
            else
                derivatives := CountCharacterOccurrences(Var,"x")+CountCharacterOccurrences(Var,"y")+CountCharacterOccurrences(Var,"z")+CountCharacterOccurrences(Var,"w"):
                derivativesRho := cat(derivativesRho, derivatives):
            end if:
        od:
    od:

    # sorts the derivatives in ascending order and removes zeroes
    derivativesA0 := Sort(derivativesA0):
    derivativesA1 := Sort(derivativesG):
    derivativesRho := Sort(derivativesH):
    derivativesA0 := Subs("0" = "", derivativesA0):
    derivativesA1 := Subs("0" = "", derivativesA1):
    derivativesRho := Subs("0" = "", derivativesRho):

    # combines the lists into one string
    structure := cat("a0[", derivativesA0, "]a1[", derivativesA1, "]rho[", derivativesRho, "]"):

end proc:
```

```
SplitStructures4D := proc(poly)
# splits a given polynomial poly into monomials of equal structure

    # loads the ArrayTools, ListTools and LinearAlgebra packages
    with(ArrayTools):
    with(ListTools):
    with(LinearAlgebra):

    # defines all used variables
    local polyVector, Split, structureTypes, i, structure, column, SplitAdd, j;
```

```

    # converts the polynomial to a vector
    polyVector := convert(poly, list):

    # defines the outputs
    Split := Matrix(numelems(polyVector)):
    structureTypes := Vector([]):

    # computes the structure of each polynomial term and sorts
       it into the correct column of Split
    for i from 1 to numelems(polyVector) do
        structure := MonomialStructure4D(polyVector[i]):

        if has(structureTypes, structure) = false then
            structureTypes := Concatenate(1, structureTypes,
               structure):
            structureTypes := Vector([structureTypes]):
        end if:

        column := ListTools[Search](structure, structureTypes):
        Split[i,column] := polyVector[i]:
    od:

    # deletes all zero columns from Split
    Split := DeleteColumn(Split, numelems(structureTypes)+1
       ..numelems(polyVector)):

    # sums all monomials of the same structure
    SplitAdd := Vector(numelems(structureTypes)):
    for j from 1 to numelems(structureTypes) do
        SplitAdd[j] := add(Column(Split,j)):
    od:

    # outputs a vector containing monomials of equal structure
       in the same element
    return SplitAdd;

end proc:
```

5.2 Skewing and Totally Skewing

```
Skew4D := proc(x, perm, vars)

# skew-symmetrizes the variables in the vector vars in a
   polynomial x by permutation perm

```

```
    # loads the GroupTheory and StringTools packages
    with(GroupTheory):
    with(StringTools):

    # defines all used variables
    local permSign, permInverse, xString, varString, j, i, xExpr
        ;

    # computes the sign and inverse of the permutation
    permSign := PermParity(perm):
    permInverse := PermInverse(perm):

    # converts the polynomial and variables to strings
    xString := convert(x, string):
    varString := Vector(4):
    for j from 1 to 4 do
        varString[j] := convert(vars[j],string):
    od:

    # substitutes the variables by the permuted variables
    xString := Subs({varString[1]=varString[permInverse[1]],
        varString[2]=varString[permInverse[2]],varString[3]=
        varString[permInverse[3]],varString[4]=varString[
        permInverse[4]]}, xString):

    # reorders the variables in the permuted polynomial to their
        original order
    for i from 1 to 2 do

        xString := RegSubs(cat(varString[2],varString[1]) = cat(
            varString[1],varString[2]), xString):
        xString := RegSubs(cat(varString[3],varString[1]) = cat(
            varString[1],varString[3]), xString):
        xString := RegSubs(cat(varString[4],varString[1]) = cat(
            varString[1],varString[4]), xString):
        xString := RegSubs(cat(varString[3],varString[2]) = cat(
            varString[2],varString[3]), xString):
        xString := RegSubs(cat(varString[4],varString[2]) = cat(
            varString[2],varString[4]), xString):
        xString := RegSubs(cat(varString[4],varString[3]) = cat(
            varString[3],varString[4]), xString):
    od:

    # applies the permutation sign to the expression
    xExpr := parse(xString):
    xExpr := permSign * xExpr;

end proc:
```

```
SkewTotal4D := proc(poly, vars)
# skews a polynomial totally in S4 with respect to the variables in vars

    # loads the GroupTheory package
    with(GroupTheory):

    # defines all used variables
    local S4, xSkewTotal, i;
    xSkewTotal := poly:

    # defines the nontrivial permutations in the symmetric group S4
    S4 := Vector([Perm([[1, 2]]), Perm([[1, 3]]), Perm([[1, 4]]), Perm([[2, 3]]), Perm([[2, 4]]), Perm([[3, 4]]), Perm([[1, 2, 3]]), Perm([[1, 3, 2]]), Perm([[1, 3, 4]]), Perm([[1, 4, 3]]), Perm([[1, 2, 4]]), Perm([[1, 4, 2]]), Perm([[2, 3, 4]]), Perm([[2, 4, 3]]), Perm([[1, 2], [3, 4]]), Perm([[1, 3], [2, 4]]), Perm([[1, 4], [2, 3]]), Perm([[1, 2, 3, 4]]), Perm([[1, 2, 4, 3]]), Perm([[1, 3, 2, 4]]), Perm([[1, 3, 4, 2]]), Perm([[1, 4, 2, 3]]), Perm([[1, 4, 3, 2]])]):

    # permutes the polynomial by permutation i and adds result to xTotalSkew
    for i from 1 to 23 do
        xSkewTotal := xSkewTotal + Skew4D(poly,S4[i],vars):
    od:

end proc:
```

5.3 Working with Markers

```
Triples2Variables4D := proc(poly)
# replaces the triple variables in a polynomial poly by the original variables

    # loads the StringTools package
    with(StringTools):

    # defines all used variables
    local vars, triples, xString, i, j, k, xExpr;

    # defines the variables and new triple names
    vars := [x,y,z,w]:
```

```
    triples := Matrix([[s1,t1,u1,v1],[s2,t2,u2,v2],[s3,t3,u3,v3
       ]]):

    # replaces the triple variables by the original variables
    xString := convert(poly,string):
    for i from 1 to 3 do
        for j from 1 to 4 do
            xString := RegSubs(convert(triples[i,j],string) =
                convert(vars[j],string), xString):
        od:
    od:

    # reorders the variables in the permuted polynomial to their
        original order
    for k from 1 to 2 do

        xString := RegSubs(convert(cat(vars[2],vars[1]),string)
            = convert(cat(vars[1],vars[2]),string), xString):
        xString := RegSubs(convert(cat(vars[3],vars[1]),string)
            = convert(cat(vars[1],vars[3]),string), xString):
        xString := RegSubs(convert(cat(vars[4],vars[1]),string)
            = convert(cat(vars[1],vars[4]),string), xString):
        xString := RegSubs(convert(cat(vars[3],vars[2]),string)
            = convert(cat(vars[2],vars[3]),string), xString):
        xString := RegSubs(convert(cat(vars[4],vars[2]),string)
            = convert(cat(vars[2],vars[4]),string), xString):
        xString := RegSubs(convert(cat(vars[4],vars[3]),string)
            = convert(cat(vars[3],vars[4]),string), xString):
    od:
    xExpr := parse(xString);

end proc:
```

```
VerifyMarker4D := proc(marker)
# checks a marker for invariances

    # defines all used variables
    local triples, Mon, i;

    # defines the triples
    triples := Matrix([[s1,t1,u1,v1],[s2,t2,u2,v2],[s3,t3,u3,v3
       ]]):

    # converts the marker to a monomial
    Mon := Triples2Variables4D(marker):

    # checks invaraince for all transpisitions in all triples
```

```
    for i from 1 to 3 do
        if Mon + Triples2Variables4D(Skew4D(marker, Perm([[1, 2]]), triples[i])) = 0 then
            return 0;
        end if:
        if Mon + Triples2Variables4D(Skew4D(marker, Perm([[1, 3]]), triples[i])) = 0 then
            return 0;
        end if:
        if Mon + Triples2Variables4D(Skew4D(marker, Perm([[1, 4]]), triples[i])) = 0 then
            return 0;
        end if:
        if Mon + Triples2Variables4D(Skew4D(marker, Perm([[2, 3]]), triples[i])) = 0 then
            return 0;
        end if:
        if Mon + Triples2Variables4D(Skew4D(marker, Perm([[2, 4]]), triples[i])) = 0 then
            return 0;
        end if:
        if Mon + Triples2Variables4D(Skew4D(marker, Perm([[3, 4]]), triples[i])) = 0 then
            return 0;
        end if:
    od:

    # outputs 0 if the marker has an invariance or 1 if it does not
    return 1;

end proc:
```

```
SkewTotalMarker4D := proc(marker)
# skews markers totally with respect to all three triples

    # defines all used variables
    local triples, skew1, skew2, skew3;

    # defines the triple variables
    triples := Matrix([[s1,t1,u1,v1],[s2,t2,u2,v2],[s3,t3,u3,v3]]):

    # totally skews the markers with respect to the first triple
    skew1 := SkewTotal4D(marker,triples[1]):

    # totally skews the markers with respect to the second
```

```
       triple
    skew2 := SkewTotal4D(skew1,triples[2]):

    # totally skews the markers with respect to the third triple
    skew3 := SkewTotal4D(skew2,triples[3]):

    # replaces the triple variables by the original variables
    skew3 := Triples2Variables4D(skew3):

end proc:
```

```
ConstructNonzeroMarker4D := proc(mon)
# given a monomial mon constructs a nonzero marker

    # loads the StringTools and combinat package
    with(StringTools):
    with(combinat):

    # defines all used variables
    local xTriples, yTriples, zTriples, wTriples, monExpanded,
       marker, i, j, k, l, markerExpr;

    # defines the triples and all their possible permutations
    xTriples := permute([s1,s2,s3]):
    yTriples := permute([t1,t2,t3]):
    zTriples := permute([u1,u2,u3]):
    wTriples := permute([v1,v2,v3]):

    # converts the monomial to a string with the powers written
       as multiplications
    monExpanded := ExpandPowers(mon):

    # goes through all 216 possible markers
    for i from 1 to 6 do
    for j from 1 to 6 do
    for k from 1 to 6 do
    for l from 1 to 6 do
        marker := monExpanded:
        marker := Substitute(marker, "x", convert(xTriples[i,1],
           string)):
        marker := Substitute(marker, "x", convert(xTriples[i,2],
           string)):
        marker := Substitute(marker, "x", convert(xTriples[i,3],
           string)):
        marker := Substitute(marker, "y", convert(yTriples[j,1],
           string)):
        marker := Substitute(marker, "y", convert(yTriples[j,2],
```

```
            string)):
        marker := Substitute(marker, "y", convert(yTriples[j,3],
            string)):
        marker := Substitute(marker, "z", convert(zTriples[k,1],
            string)):
        marker := Substitute(marker, "z", convert(zTriples[k,2],
            string)):
        marker := Substitute(marker, "z", convert(zTriples[k,3],
            string)):
        marker := Substitute(marker, "w", convert(wTriples[l,1],
            string)):
        marker := Substitute(marker, "w", convert(wTriples[l,2],
            string)):
        marker := Substitute(marker, "w", convert(wTriples[l,3],
            string)):

        # checks whether we found a nonzero marker
        if VerifyMarker4D(marker) = 1 then
            markerExpr := parse(marker):
            return markerExpr;
        end if:
    od:
    od:
    od:
    od:
    return 0;

end proc:
```

5.4 The Algorithm

```
ConstructSolution4D := proc(poly)
# given a polynomial poly looks for a representation of the
    polynomial by the total skew-symmetrization of markers

    # defines all used variables
    local polySum, usedMarkers, mon, marker, skewed, monCoeff,
        monNoCoeff, skewedVector, skewedCoeffs, skewedNoCoeffs,
        coeffsFactor, i;

    # copies the polynomial and defines the output
    polySum := poly:
    usedMarkers := 0:

    # initializes the counter
    i := 1:
```

```

    # loops until we find a basis for the polynomial
    while polySum <> 0 do

        # takes the first monomial from the polynomial
        mon := convert(polySum,list)[i]:

        # constructs a nonzero marker and its total skew-symmetrization
        marker := ConstructNonzeroMarker4D(mon):
        if marker = 0 then
            i := i + 1:
            next
        end if:
        skewed := SkewTotalMarker4D(marker):

        # makes a version of the monomial without the coefficient
        monCoeff := coeffs(mon):
        monNoCoeff := mon *~ monCoeff^~(-1):

        # makes a vector of the skewed expression without coefficients
        skewedVector := Vector([convert(skewed,list)]):
        skewedCoeffs := Vector([coeffs~(skewedVector)]):
        skewedNoCoeffs := skewedVector *~ skewedCoeffs^~(-1):

        # compares the coefficients between the equal monomials
        if member(monNoCoeff, skewedNoCoeffs, 'position') = false then
            i := i + 1:
            next
        end if:
        coeffsFactor := monCoeff / skewedCoeffs[position]:

        # subtracts a multilple of the skewed expression from the polynomial
        polySum := polySum - coeffsFactor * skewed:
        usedMarkers := usedMarkers + coeffsFactor * marker:

        # resets the counter
        i := 1:
    od:

    return usedMarkers;

end proc:
```

6 Analyzing the Results

In this chapter we apply the procedures provided in chapter 4 to the equations $\dot{a}$ and $\dot{\rho}$, and we analyze and discuss the results we obtained from this. In section 6.1 we analyze the different structures found in $\dot{a}$ and $\dot{\rho}$, and in section 6.2 we look at the solutions for $\dot{a}''$ and $\dot{\rho}''$ obtained by our algorithm. In section 6.3 we apply the 4-dimensional procedures from chapter 5 to two evolution equations obtained by deforming 4d-determinant Nambu-Poisson brackets using Kontsevich's tetrahedral flow to find out whether the results obtained in 3 dimensions generalize to 4 dimensions.

6.1 The Structures in $\dot{a}$ and $\dot{\rho}$

First of all, we use the procedure *SplitStructures* from section 4.1 to sort all monomials in $\dot{a}$ and $\dot{\rho}$ into polynomials containing only monomials of equal structure. This allows us to see exactly what structures are present in $\dot{a}$ and $\dot{\rho}$, and how many monomials of each structure there are. Running the procedure *SplitStructures* for $\dot{a}$ and $\dot{\rho}$ gives the following result:

$\dot{a}$ contains	$\dot{\rho}$ contains
54: $a[1113]\rho[111]$	54: $a[111]\rho[1113]$
	102: $a[112]\rho[1112]$
102: $a[1123]\rho[11]$	102: $a[112]\rho[113]$
	96: $a[122]\rho[112]$
72: $a[1223]\rho[1]$	72: $a[122]\rho[13]$

Table 1: The structures in $\dot{a}$ and $\dot{\rho}$

As we can see there are eight different structures in $\dot{a}$ and $\dot{\rho}$, three of which in $\dot{a}$ and five of which in $\dot{\rho}$. The problem stated in section 2.4 can thus be split into eight parts: we are looking for a marker-polynomial to represent each of these eight single-structure polynomials by total skew-symmetrization.

If we take a closer look at the number of monomials in each single-structure polynomial, we notice that some polynomials contain an equal number of monomials. In fact, these single-structure polynomials are related to each other in the following ways:

The 54 monomials of structure $a[1113]\rho[111]$ and $a[111]\rho[1113]$ are exactly the same except for two differences: for every monomial the triple derivative of a in $a[1113]\rho[111]$ becomes a triple derivative of ρ in $a[111]\rho[1113]$, and the monomials in $a[111]\rho[1113]$ have coefficients that are twice the coefficients of the monomials in $a[1113]\rho[111]$.

The 72 monomials of structure $a[1223]\rho[1]$ and $a[122]\rho[13]$ are exactly the same except for one difference: for every monomial the triple derivative of a in $a[1223]\rho[1]$ becomes a triple derivative of ρ in $a[122]\rho[13]$. Specifically, the coefficients of the monomials in $a[1223]\rho[1]$ and $a[122]\rho[13]$ are the same.

The 102 monomials of structure $a[1123]\rho[11]$ and $a[112]\rho[113]$ are exactly the same except for one difference: for every monomial the triple derivative of a in $a[1123]\rho[11]$ becomes a triple derivative of ρ in $a[112]\rho[113]$. The 102 monomials of structure $a[112]\rho[1112]$ are related to the monomials in $a[112]\rho[113]$ in the following way: for every monomial the triple derivative of ρ in $a[112]\rho[113]$ is split into a double double and single derivative of ρ in $a[112]\rho[1112]$. Also, the monomials in $a[112]\rho[113]$ have coefficients that are minus the coefficients of the monomials in $a[112]\rho[1112]$.

6.2 Solutions to the Problem

We can attempt to run the procedure *ConstructSolution* from section 4.4 for each of the eight single-structure polynomials in $\dot{a}$ and $\dot{\rho}$. Alternatively, we could also try to run the procedure *ConstructSolution* for $\dot{a}$ and $\dot{\rho}$ directly. As it turns out, both attempts give us the same solution:

$$\begin{aligned}\dot{a}'' = {} & 2a_{u1}a_{u2}a_{u3}\rho_{w1}\rho_{w2}\rho_{w3}a_{v1v2v3} - 6\rho a_{u1v2}a_{u2}a_{u3}\rho_{w1}\rho_{w3}a_{v1v3w2} \\ & - 6\rho^2 a_{u1}a_{u2u3}a_{v1v2}\rho_{w3}a_{v3w1w2}\end{aligned} \tag{18}$$

$$\begin{aligned}\dot{\rho}'' = {} & -2a_{u1}a_{u2}a_{u3}\rho_{v1}\rho_{v2}\rho_{v3}\rho_{w1w2w3} + 6a_{u1v2}a_{u2}a_{u3}\rho_{v1}\rho_{v3}\rho_{w2}\rho_{w1w3} \\ & - 12\rho a_{u1}a_{u2u3}a_{v1v2}\rho_{v3}\rho_{w1}\rho_{w2w3} - 6\rho a_{u1v2}a_{u2}a_{u3}\rho_{v1}\rho_{v3}\rho_{w1w2w3} \\ & + 6\rho^2 a_{u1}a_{u2u3}a_{v1v2}\rho_{v3}\rho_{w1w2w3}\end{aligned} \tag{19}$$

We can easily check that these solutions are indeed correct by computing their total skew-symmetrizations and subtracting $\dot{a}$ or $\dot{\rho}$ respectively. Note that the structures of these eight markers correspond exactly to the eight structures found in $\dot{a}$ and $\dot{\rho}$, as expected. We confirm the earlier discovery by Buring and Kiselev that $\dot{a}$ can be represented by three markers. Moreover, we find that we can represent $\dot{\rho}$ using five markers.

But what about the uniqueness of these solutions? For one, we can conclude that this is the smallest possible solution. It consists of eight markers: three markers for $\dot{a}$ and five markers for $\dot{\rho}$. This means that the total skew-symmetrization of each of these markers is equal to all monomials of that structure in $\dot{a}$ or $\dot{\rho}$. We managed to represent each single-structure polynomial in $\dot{a}$ and $\dot{\rho}$ by a single marker. It is not possible to find a smaller solution, since there are eight different structures in $\dot{a}$ and $\dot{\rho}$, and we need at least one marker of each structure to completely represent $\dot{a}$ and $\dot{\rho}$.

While playing with the algorithm in Maple we notice something peculiar: it seems that our naive algorithm always finds a solution, regardless of which monomial we start with in the first step. Specifically, the total skew-symmetrization of every nonzero marker the algorithm encounters is always exactly a multiple of all monomials of that structure in $\dot{a}$ or $\dot{\rho}$. We hypothesize that this is the case because for any of the eight structure types in $\dot{a}$ and $\dot{\rho}$ the total skew-symmetrization of *every* nonzero marker of these structures is always exactly a multiple of all monomials of that structure in $\dot{a}$ or $\dot{\rho}$.

To verify this we use a brute force approach using the procedures from section 4.5. First of all, the procedure *ConstructAllMonOfStruc* gives us a list of all possible monomials of a given structure. For instance, we can construct a list of all possible monomials of structure $a[1113]\rho[111]$ (with coefficient 1). Then, using this list of all monomials and the procedure *ConstructAllNonzeroMarkers*, we can construct all possible nonzero markers of structure $a[1113]\rho[111]$. Since we know that it is only possible to represent monomials by markers of their own structure, this list of all possible nonzero markers of structure $a[1113]\rho[111]$ contains all markers that can possibly be used to represent the monomials of structure $a[1113]\rho[111]$ in $\dot{a}$. Finally, using the procedure *SkewTotalAndCompare* we can compute the total skew-symmetrization of every nonzero marker of structure $a[1113]\rho[111]$ and check if it is a multiple of all monomials of structure $a[1113]\rho[111]$ in $\dot{a}$.

We can run these procedures for all eight structure types in $\dot{a}$ and $\dot{\rho}$. The result is surprising, and our hypothesis is confirmed: it turns out that for any of the eight structure types in $\dot{a}$ or $\dot{\rho}$, the total skew-symmetrization of any nonzero marker of that structure is always a rational multiple of all monomials of that structure in $\dot{a}$ or $\dot{\rho}$. The eight single-structure polynomials in $\dot{a}$ and $\dot{\rho}$ thus each have a special kind of hyper-symmetry: they can be exactly represented by some multiple of the total skew-symmetrization of any nonzero marker of their own structure. In other words, for each of the eight structure types in $\dot{a}$ and $\dot{\rho}$ the total skew-symmetrizations of any nonzero markers of that structure are multiples of one another.

Ex 7. To illustrate that this hyper-symmetry is a special property of the monomials in $\dot{a}$ and $\dot{\rho}$ and does not occur in most monomials of arbitrary structure, consider the following example. Let's take a monomial in two dimensions containing six derivatives, three derivatives of x and three of y, and of structure $a[12]\rho[12]$. An example of such a monomial is given by

$$a_x \rho_x a_{yy} \rho_{xy} \tag{20}$$

Consider the following nonzero markers, constructed from this monomial:

$$\begin{gathered} a_{s1}\rho_{s2}a_{t1t3}\rho_{s3t2} \\ a_{s1}\rho_{s2}a_{t2t3}\rho_{s3t1} \end{gathered} \tag{21}$$

The total skew-symmetrizations of these two markers are not multiples of one another, even though they are of the same structure! This shows that the hyper-symmetry in $\dot{a}$ and $\dot{\rho}$ is indeed a special property. △

From the hyper-symmetry of $\dot{a}$ and $\dot{\rho}$ we can conclude that the solutions found in equations (18) and (19) are not only minimal, but also maximal. That is to say, we cannot find any solutions that do not consist of exactly three and five markers respectively. Moreover, it is now clear that these solutions are definitely not unique. There are in fact a great number of solutions. For each structure in $\dot{a}$ and $\dot{\rho}$, any nonzero marker of that structure can be part of a solution. For each single-structure polynomial from $\dot{a}$ and $\dot{\rho}$ there are thus as many possible solutions as there are nonzero markers of that structure. The total number of solutions can be found by multiplying the number of distinct nonzero markers of each structure type in $\dot{a}$ or $\dot{\rho}$.

This hyper-symmetry explains why the algorithm from section 3.3 always manages to find a solution to this problem, even though it is a naive algorithm without any stopping criteria: it is impossible for the algorithm to encounter a nonzero marker that cannot be part of a solution. Thus, it is simply impossible for the algorithm to end up in a loop.

6.3 The Tetrahedral Flow in 4D

We would like to know whether this observed hyper-symmetry in 3 dimensions generalizes to 4 dimensions. To this end we look at the dynamical system obtained by deforming 4d-determinant Nambu-Poisson brackets, again using Kontsevich's tetrahedral flow. This system was found by Buring in 2021 [9] and contains three differential equations, $\dot{a_0}$, $\dot{a_1}$ and $\dot{\rho}$. The equations for $\dot{a_0}$ and $\dot{a_1}$ contain 33084 terms each, while the equation for $\dot{\rho}$ contains 90024 terms.

The 4-dimensional case is quite a natural extension of the 3-dimensional case: in the 4-dimensional case every monomial contains exactly twelve derivatives; three of x, three of y, three of z and three of w. Constructing markers thus works by partitioning the independent variables into three quadruples, which we denote by $\{s_n, t_n, u_n, v_n\}$ for $n \in \{1, 2, 3\}$. The total skew-symmetrization of such a marker in 4 dimensions is calculated with respect to S_4, which contains 24 permutations. The total skew-symmetrization of a marker in 4 dimensions thus consist of $24^3 = 13824$ terms. As in the 3-dimensional case, the existence of marker-polynomials whose total skew-symmetrizations are equal to $\dot{a_0}$, $\dot{a_1}$ and $\dot{\rho}$ is guaranteed by the differential geometry of these equations.

We only analyze the equations for $\dot{a_0}$ and $\dot{a_1}$, leaving the analysis of the much larger equation for $\dot{\rho}$. This analysis proceeds in the same way as the analysis of the 3-dimensional case. Using the procedure *SplitStructures4D* from section 5.1 we can sort all the monomials in $\dot{a_0}$ and $\dot{a_1}$ into polynomials containing only monomials of equal structure. Running this procedure for $\dot{a_0}$ and $\dot{a_1}$ gives us the following result:

$\dot{a_0}$ contains	$\dot{a_1}$ contains
4512: $a_0[1123]a_1[122]$	4512: $a_0[122]a_1[1123]$
4512: $a_0[1223]a_1[112]$	4512: $a_0[112]a_1[1223]$
3168: $a_0[1113]a_1[122]\rho[1]$	3168: $a_0[122]a_1[1113]\rho[1]$
7872: $a_0[1123]a_1[112]\rho[1]$	7872: $a_0[112]a_1[1123]\rho[1]$
3168: $a_0[1223]a_1[111]\rho[1]$	3168: $a_0[111]a_1[1223]\rho[1]$
3984: $a_0[1113]a_1[112]\rho[11]$	3984: $a_0[112]a_1[1113]\rho[11]$
3984: $a_0[1123]a_1[111]\rho[11]$	3984: $a_0[111]a_1[1123]\rho[11]$
1848: $a_0[1113]a_1[111]\rho[111]$	1848: $a_0[111]a_1[1113]\rho[111]$

Table 2: The structures in $\dot{a_0}$ and $\dot{a_1}$

As we can see there are sixteen different structures in total, eight of which in $\dot{a_0}$ and eight of which in $\dot{a_1}$. The problem of representing these polynomials by the total skew-symmetrization of markers can thus be split into sixteen parts.

If we take a closer look at the structures in $\dot{a_0}$ and $\dot{a_1}$ we might suspect that the equations for $\dot{a_0}$ and $\dot{a_1}$ are somehow related. In fact, the polynomial $\dot{a_0}$ is minus the polynomial $\dot{a_1}$, but with the dependent variables a_0 and a_1 interchanged. Note that if we plug in $\rho = 1$ we do not get $\dot{a_0} = \dot{a_1} = 0$, as in the 3-dimensional case. Instead, the structures that have no ρ derivatives remain, so 9024 monomials remain for $\dot{a_0}$ and 9024 monomials remain for $\dot{a_1}$.

Using the procedure *ConstructSolution4D* from section 5.4 for each of the sixteen single-structure polynomials in $\dot{a}_0$ and $\dot{a}_1$ we find the following marker-polynomials, denoted by $\dot{a}_0''$ and $\dot{a}_1''$, whose total skew-symmetrizations are equal to $\dot{a}_0$ and $\dot{a}_1$ respectively:

$$
\begin{aligned}
\dot{a}_0'' = &+ 3a0_{s1u2u3}a0_{t1t2}a1_{s2}a1_{s3v1}a1_{t3u1}a0_{v2}a0_{v3}\rho^3 \\
&+ 6a0_{s1u2}a0_{t1}a0_{t2v3}a0_{u3v1v2}a1_{t3u1}a1_{s2}a1_{s3}\rho^3 \\
&+ 3a0_{v2}a0_{t1u2v3}a1_{v1}\rho_{s1}a0_{u1}a0_{u3}a1_{s2t3}a1_{s3t2}\rho^2 \\
&- 6a0_{s1v3}a1_{s2t1}\rho_{v1}a0_{s3u1v2}a0_{u2}a0_{u3}a1_{t2}a1_{t3}\rho^2 \\
&- 6a0_{s1v2v3}a0_{t}1a0_{u2}a0_{u3v1}a1_{s2}a1_{s3u1}a1_{t3}\rho_{t2}\rho^2 \\
&+ 6a0_{s1}a0_{s2v3}a0_{s3u1}a0_{t1t2t3}a1_{v1}\rho_{v2}a1_{u2}a1_{u3}\rho^2 \\
&- 6a0_{s1}a0_{t1u2u3}a1_{u1v2}a0_{t2}a0_{t3}a1_{s2}a1_{s3}\rho_{v1}\rho_{v3}\rho \\
&+ 6a0_{v2}a0_{s1}a0_{s2s3t1}a0_{t2t3}\rho_{v1}\rho_{v3}a1_{u1}a1_{u2}a1_{u3}\rho \\
&- 2a0_{s1}a0_{t2}a0_{t3u1u2}a0_{u3}a1_{t1}a1_{s2}a1_{s3}\rho_{v1}\rho_{v2}\rho_{v3}
\end{aligned}
\tag{22}
$$

$$
\begin{aligned}
\dot{a}_1'' = &+ 3a0_{s1}a1_{t1u2}a1_{u1u3v2}a0_{s2t3}a0_{s3t2}a1_{v1}a1_{v3}\rho^3 \\
&- 3a0_{s1t2}a1_{u1}a1_{u2u3v1}a0_{t1}a0_{t3}a1_{s2v3}a1_{s3v2}\rho^3 \\
&- 6a0_{u1}a1_{t1t2v3}\rho_{t3}a0_{u2v1}a0_{u3v2}a1_{s1}a1_{s2}a1_{s3}\rho^2 \\
&+ 6a0_{s1}a0_{t1t3}a0_{u2}a1_{v2}a1_{s2v1v3}a1_{s3t2}a1_{u1}\rho_{u3}\rho^2 \\
&+ 6a0_{t1u2}a1_{u1v2}\rho_{u3}a1_{s1s2s3}a0_{v1}a0_{v3}a1_{t2}a1_{t3}\rho^2 \\
&+ 3a1_{v1}a1_{s1s2s3}\rho_{t1}a1_{t2v3}a1_{t3v2}a0_{u1}a0_{u2}a0_{u3}\rho^2 \\
&- 6a0_{t1u2}a1_{u1u3v2}a0_{v1}a0_{v3}\rho_{t2}\rho_{t3}a1_{s1}a1_{s2}a1_{s3}\rho \\
&- 6a1_{t1u2v3}a1_{u1u3}a1_{v1}a1_{v2}\rho_{t2}\rho_{t3}a0_{s1}a0_{s2}a0_{s3}\rho \\
&+ 2a1_{s1}a1_{s2s3t1}\rho_{v1}a1_{v2}a1_{v3}\rho_{t2}\rho_{t3}a0_{u1}a0_{u2}a0_{u3}
\end{aligned}
\tag{23}
$$

We can easily check that these solutions are indeed correct by computing their total skew-symmetrizations and subtracting $\dot{a}_0$ or $\dot{a}_1$ respectively. We confirm that $\dot{a}_0$ and $\dot{a}_1$ can be exactly represented by the total skew-symmetrization of markers. Note that the structures of these markers correspond exactly to the structures found in $\dot{a}_0$ and $\dot{a}_1$, as expected.

However, note that in this case we represented $\dot{a}_0$ and $\dot{a}_1$ using nine markers for each polynomial, one more than there are structure types in $\dot{a}_0$ and $\dot{a}_1$. If we look at our obtained markers we see that the 7872 monomials of structure $a_0[1123]a_1[112]\rho[1]$ and the 7872 monomials of structure $a_0[112]a_1[1123]\rho[1]$ are represented by two markers. This is an important difference compared to the 3-dimensional case, where all monomials of equal structure could always be represented by a single marker.

In fact, when running the procedure *ConstructSolution4D* for all monomials of structure $a_0[1123]a_1[112]\rho[1]$ or $a_0[112]a_1[1123]\rho[1]$ in $\dot{a_0}$ and $\dot{a_1}$ we find that our algorithm, which worked perfectly for the 3-dimensional case, sometimes does end up in a loop and fails to find a solution. This behaviour depends on the starting monomial of the algorithm (which can be varied by changing the value of the counter *'i'* in line 13 of *ConstructSolution4D*). We suspect that it is still possible for every nonzero marker of one of these structures to be part of a solution, but that the two markers needed to represent all monomials of these structures in $\dot{a_0}$ and $\dot{a_1}$ should somehow complement each other. In other words, while we suspect that every nonzero marker of these structures can be part of a solution, not every combination of nonzero markers of these structures works as a solution.

If this is indeed the case we can postulate that the hyper-symmetry discovered in the 3-dimensional tetrahedral flow persists in some way in its 4-dimensional counterpart. For fourteen of the sixteen structures in $\dot{a_0}$ and $\dot{a_1}$ we suspect the same hyper-symmetry as in the 3-dimensional case: for each of these structures a multiple of every nonzero marker is a solution and all nonzero markers of equal structure are multiples of one another. For the structures $a_0[1123]a_1[112]\rho[1]$ and $a_0[112]a_1[1123]\rho[1]$ we suspect that a multiple of every nonzero marker can still be part of a solution, but all nonzero markers of equal structure are no longer always multiples of one another. These hypotheses still need to be verified, possibly using a similar approach to the one used in section 6.2, by modifying the procedures from section 4.5 to work in 4 dimensions.

7 Conclusion

The deformation of 3d-determinant Nambu-Poisson brackets by Kontsevich's tetrahedral flow gives rise to the differential polynomial equations $\dot{a}$ and $\dot{\rho}$, which were previously found to be skew-symmetric and totally skew-symmetric. The existence of markers whose total skew-symmetrizations are equal to $\dot{a}$ and $\dot{\rho}$ was guaranteed by the differential geometry of this problem, and it was known that $\dot{a}$ can be represented using three markers.

We confirmed that we can represent $\dot{a}$ using three markers, and we found that we can represent $\dot{\rho}$ using five markers. We discovered that we cannot represent $\dot{a}$ or $\dot{\rho}$ using anything other than three or five nonzero markers respectively (ignoring zero markers). Moreover, we discovered that $\dot{a}$ and $\dot{\rho}$ are hyper-symmetric: the total skew-symmetrization of any nonzero marker of the structures in $\dot{a}$ and $\dot{\rho}$ is always a multiple of all monomials of that structure in $\dot{a}$ and $\dot{\rho}$.

The deformation of 4d-determinant Nambu-Poisson brackets by Kontsevich's tetrahedral flow gives rise to the differential polynomial equations $\dot{a_0}$ and $\dot{a_1}$. The existence of markers whose total skew-symmetrizations are equal to $\dot{a_0}$ and $\dot{a_1}$ was guaranteed by the differential geometry of this problem. We confirmed that we can represent $\dot{a_0}$ and $\dot{a_1}$ using total skew-symmetrizations of markers, and discovered that for both $\dot{a_0}$ and $\dot{a_1}$ this can be done using nine markers. We hypothesize that the hyper-symmetry discovered in the 3-dimensional case persists in the 4-dimensional case.

8 Discussion

To further analyze the symmetries that arise in the differential equations obtained by deforming Nambu-Poisson brackets we suggest the following. We strongly suspect that the hyper-symmetry discovered in the 3-dimensional case persists in 4 dimensions. This should be verified, for instance by modifying the procedures from section 4.5 to work in 4 dimensions and checking if every nonzero marker of the structures in the 4-dimensional equations $\dot{a_0}$ and $\dot{a_1}$ can be part of a solution. Specifically, for the monomials of structure $a_0[1123]a_1[112]\rho[1]$ and $a_0[112]a_1[1123]\rho[1]$ in $\dot{a_0}$ and $\dot{a_1}$ we would like to know how the two nonzero markers needed to represent each of these structures should complement each other. Moreover, the 4-dimensional equation for $\dot{\rho}$ was not analyzed in this report and could be analyzed in the same way as the equations for $\dot{a_0}$ and $\dot{a_1}$, using the procedures from chapter 5.

Finally, it would be interesting to know whether the hyper-symmetry not only persists in 4 dimensions, but whether it persists in arbitrary dimensions. To study this we could modify the procedures from chapter 4 to work in arbitrary dimensions. We could also study the symmetry of differential equations obtained by deforming Nambu-Poisson brackets using other Kontsevich flows, for example the 5-wheel cocycle flow.

References

[1] L.D. Landau and E.M. Lifshitz. *Mechanics*, pages 131–138. Course of Theoretical Physics. Butterworth-Heinemann, 3 edition, 1976.

[2] Y. Nambu. *Generalized Hamiltonian Dynamics*, pages 2405–2412. Physical Review D, 7(8). 1973.

[3] C.L. Gengoux, A. Pichereau, and P. Vanhaecke. *Poisson Structures*, page 251. Springer-Verlag, Berlin, 2013.

[4] M. Kontsevich. *Formality conjecture. Deformation theory and symplectic geometry*, pages 139–156. Kluwer Academic Publishers, Dordrecht, 1997. (Ascona 1996, D. Sternheimer, J. Rawnsley, S. Gutt, eds).

[5] A. Bouisaghouane, R. Buring, and A. Kiselev. *The Kontsevich Tetrahedral Flow Revisited*, page 7. 2017. arXiv:1608.01710v4 [math.QA].

[6] R. Buring and A. Kiselev. 2019.11.27. Sent as personal communication from R. Buring to A. Kiselev.

[7] R. Buring and A. Kiselev. 2020.12.04. Sent as personal communication from R. Buring to A. Kiselev.

[8] *Maple 2020.2*. Maplesoft; a division of Waterloo Maple Inc.

[9] R. Buring. 2021.06.13. Sent as personal communication from R. Buring to A. Kiselev.

Acknowledgements

I would like to thank my supervisor, Arthemy Kiselev, for providing me with this extremely interesting research topic, for his excellent and thought-provoking feedback, and for inspiring me with his enthusiasm.

I would like to thank Ricardo Buring for suggesting an approach to solve the problem, for always being available to answer my questions, for helping me with the references, for helping we write the introduction and for checking my results.

I would like to thank Nikolay Martynchuk for providing the second assessment of this report.

I would like to thank Jasper Janssen for proofreading the final version of this report.

Appendices

A Polynomials $\dot{a}$ and $\dot{\rho}$

```
adot := -12*rho^2*a_x*rho_y*a_xy*a_zz*a_xyz+12*rho^2*a_x*rho_y*
    a_xy*a_xz*a_yzz+12*rho^2*a_x*rho_y*a_xy*a_xzz*a_yz-12*rho^2*
    a_x*rho_y*a_xz*a_yy*a_xzz+12*rho^2*a_x*rho_y*a_xz*a_yz*a_xyz
    -12*rho^2*a_x*rho_y*a_yzz*a_yz*a_xx+6*rho^2*a_x*rho_y*a_xx*
    a_zzz*a_yy+6*rho^2*a_x*rho_y*a_xx*a_zz*a_yyz+6*rho^2*a_x*
    rho_y*a_zz*a_yy*a_xxz-12*rho^2*a_x*rho_z*a_xy*a_yz*a_xyz-12*
    rho^2*a_x*rho_z*a_xy*a_xz*a_yyz+12*rho^2*a_x*rho_z*a_xy*a_zz*
    a_xyy-12*rho^2*a_x*rho_z*a_xz*a_xyy*a_yz+12*rho^2*a_x*rho_z*
    a_xz*a_yy*a_xyz+12*rho^2*a_x*rho_z*a_yz*a_xx*a_yyz-6*rho^2*
    a_x*rho_z*a_xx*a_zz*a_yyy-6*rho^2*a_x*rho_z*a_xx*a_yzz*a_yy
    -6*rho^2*a_x*rho_z*a_zz*a_yy*a_xxy-12*rho^2*a_y*rho_x*a_xy*
    a_xz*a_yzz-12*rho^2*a_y*rho_x*a_xy*a_xzz*a_yz+12*rho^2*a_y*
    rho_x*a_xy*a_zz*a_xyz-12*rho^2*a_y*rho_x*a_xz*a_yz*a_xyz+12*
    rho^2*a_y*rho_x*a_xz*a_yy*a_xzz+12*rho^2*a_y*rho_x*a_yzz*a_yz
    *a_xx-6*rho^2*a_y*rho_x*a_xx*a_zz*a_yyz-6*rho^2*a_y*rho_x*
    a_xx*a_zzz*a_yy-6*rho^2*a_y*rho_x*a_zz*a_yy*a_xxz+12*rho^2*
    a_y*rho_z*a_xy*a_xz*a_xyz+12*rho^2*a_y*rho_z*a_xy*a_yz*a_xxz
    -12*rho^2*a_y*rho_z*a_xy*a_zz*a_xxy+12*rho*a_x*a_y*rho_z*
    rho_x*a_yy*a_xzz-24*rho*a_x*a_y*rho_z*rho_x*a_yz*a_xyz+12*rho
    *a_x*a_y*rho_z*rho_x*a_zz*a_xyy+24*rho*a_x*a_y*rho_z*rho_y*
    a_xz*a_xyz-12*rho*a_x*a_y*rho_z*rho_y*a_xx*a_yzz-12*rho*a_x*
    a_y*rho_z*rho_y*a_zz*a_xxy+24*rho*a_x*a_z*rho_y*rho_x*a_yz*
    a_xyz-12*rho*a_x*a_z*rho_y*rho_x*a_zz*a_xyy-12*rho*a_x*a_z*
    rho_y*rho_x*a_yy*a_xzz-24*rho*a_x*a_z*rho_z*rho_y*a_xyz*a_xy
    +12*rho*a_x*a_z*rho_z*rho_y*a_xx*a_yyz+12*rho*a_x*a_z*rho_z*
    rho_y*a_xxz*a_yy+12*rho*a_z*a_y*rho_x*rho_y*a_xx*a_yzz-24*rho
    *a_z*a_y*rho_x*rho_y*a_xz*a_xyz+12*rho*a_z*a_y*rho_x*rho_y*
    a_zz*a_xxy+24*rho*a_z*a_y*rho_x*rho_z*a_xyz*a_xy-12*rho*a_z*
    a_y*rho_x*rho_z*a_xx*a_yyz-12*rho*a_z*a_y*rho_x*rho_z*a_xxz*
    a_yy-6*rho^2*a_x*rho_y*a_zzz*a_xy^2-6*rho^2*a_x*rho_y*a_xz^2*
    a_yyz-6*rho^2*a_x*rho_y*a_yz^2*a_xxz+6*rho^2*a_x*rho_z*a_yzz*
    a_xy^2+6*rho^2*a_x*rho_z*a_xz^2*a_yyy+6*rho^2*a_x*rho_z*a_yz
    ^2*a_xxy+6*rho^2*a_y*rho_x*a_zzz*a_xy^2+6*rho^2*a_y*rho_x*
    a_xz^2*a_yyz+6*rho^2*a_y*rho_x*a_yz^2*a_xxz-6*rho^2*a_y*rho_z
    *a_xzz*a_xy^2-6*rho^2*a_y*rho_z*a_xz^2*a_xyy-6*rho^2*a_y*
    rho_z*a_yz^2*a_xxx-6*rho^2*a_z*rho_x*a_yzz*a_xy^2-6*rho^2*a_z
    *rho_x*a_xz^2*a_yyy-6*rho^2*a_z*rho_x*a_yz^2*a_xxy+6*rho^2*
    a_z*rho_y*a_xzz*a_xy^2+6*rho^2*a_z*rho_y*a_xz^2*a_xyy+6*rho
    ^2*a_z*rho_y*a_yz^2*a_xxx-6*rho*a_x^2*rho_y^2*a_zz*a_xyz-6*
    rho*a_x^2*rho_y^2*a_xy*a_zzz+6*rho*a_x^2*rho_y^2*a_xz*a_yzz
    +6*rho*a_x^2*rho_y^2*a_xzz*a_yz-6*rho*a_x^2*rho_z^2*a_xyy*
```

```
a_yz-6*rho*a_x^2*rho_z^2*a_xy*a_yyz+6*rho*a_x^2*rho_z^2*a_xz*
a_yyy+6*rho*a_x^2*rho_z^2*a_yy*a_xyz-6*rho*a_y^2*rho_x^2*
a_xzz*a_yz+6*rho*a_y^2*rho_x^2*a_zz*a_xyz+6*rho*a_y^2*rho_x
^2*a_xy*a_zzz-6*rho*a_y^2*rho_x^2*a_xz*a_yzz-6*rho*a_y^2*
rho_z^2*a_xxx*a_yz+6*rho*a_y^2*rho_z^2*a_xxz*a_xy-6*rho*a_y
^2*rho_z^2*a_xx*a_xyz+6*rho*a_y^2*rho_z^2*a_xxy*a_xz+6*rho*
a_z^2*rho_x^2*a_xy*a_yyz-6*rho*a_z^2*rho_x^2*a_xz*a_yyy-6*rho
*a_z^2*rho_x^2*a_yy*a_xyz+6*rho*a_z^2*rho_x^2*a_xyy*a_yz+6*
rho*a_z^2*rho_y^2*a_xxx*a_yz+6*rho*a_z^2*rho_y^2*a_xx*a_xyz
-6*rho*a_z^2*rho_y^2*a_xxy*a_xz-6*rho*a_z^2*rho_y^2*a_xxz*
a_xy+6*a_x^2*a_y*rho_x*a_zzz*rho_y^2+6*a_x^2*a_y*rho_x*a_yyz*
rho_z^2+12*a_x^2*a_y*a_xyz*rho_y*rho_z^2-6*a_x^2*a_y*a_xzz*
rho_z*rho_y^2-6*a_x^2*a_z*rho_x*rho_y^2*a_yzz-6*a_x^2*a_z*
rho_x*rho_z^2*a_yyy+6*a_x^2*a_z*a_xyy*rho_z^2*rho_y-12*a_x^2*
a_z*a_xyz*rho_z*rho_y^2+6*a_x*a_y^2*rho_x^2*rho_z*a_yzz-6*a_x
*a_y^2*rho_x^2*a_zzz*rho_y-12*a_x*a_y^2*rho_x*a_xyz*rho_z
^2-6*a_x*a_y^2*a_xxz*rho_z^2*rho_y+6*a_x*a_z^2*rho_x^2*rho_z*
a_yyy-6*a_x*a_z^2*rho_x^2*a_yyz*rho_y+12*a_x*a_z^2*rho_x*
a_xyz*rho_y^2+6*a_x*a_z^2*a_xxy*rho_z*rho_y^2+12*a_z*a_y^2*
rho_x^2*rho_z*a_xyz+6*a_z*a_y^2*rho_x^2*rho_y*a_xzz-6*a_z*a_y
^2*rho_x*a_xxy*rho_z^2+6*a_z*a_y^2*a_xxx*rho_y*rho_z^2-6*a_z
^2*a_y*rho_x^2*a_xyy*rho_z-12*a_z^2*a_y*rho_x^2*a_xyz*rho_y
+6*a_z^2*a_y*rho_x*rho_y^2*a_xxz-6*a_z^2*a_y*a_xxx*rho_z*
rho_y^2+6*a_x^3*a_yzz*rho_y^2*rho_z-6*a_x^3*a_yyz*rho_z^2*
rho_y-6*a_x^2*a_y*a_xyy*rho_z^3+6*a_x^2*a_z*a_xzz*rho_y^3+6*
a_x*a_y^2*a_xxy*rho_z^3-6*a_x*a_z^2*a_xxz*rho_y^3+6*a_y^3*
a_xxz*rho_x*rho_z^2-6*a_y^3*a_xzz*rho_x^2*rho_z-6*a_z*a_y^2*
a_yzz*rho_x^3+6*a_z^2*a_y*a_yyz*rho_x^3+6*a_z^3*a_xyy*rho_x
^2*rho_y-6*a_z^3*a_xxy*rho_x*rho_y^2-12*rho^2*a_y*rho_z*a_xz*
a_xxz*a_yy+12*rho^2*a_y*rho_z*a_xz*a_yz*a_xxy-12*rho^2*a_y*
rho_z*a_yz*a_xx*a_xyz+6*rho^2*a_y*rho_z*a_xx*a_zz*a_xyy+6*rho
^2*a_y*rho_z*a_xx*a_yy*a_xzz+6*rho^2*a_y*rho_z*a_zz*a_yy*
a_xxx+12*rho^2*a_z*rho_x*a_xy*a_xz*a_yyz+12*rho^2*a_z*rho_x*
a_xy*a_yz*a_xyz-12*rho^2*a_z*rho_x*a_xy*a_zz*a_xyy-12*rho^2*
a_z*rho_x*a_xz*a_yy*a_xyz+12*rho^2*a_z*rho_x*a_xz*a_xyy*a_yz
-12*rho^2*a_z*rho_x*a_yz*a_xx*a_yyz+6*rho^2*a_z*rho_x*a_xx*
a_zz*a_yyy+6*rho^2*a_z*rho_x*a_xx*a_yzz*a_yy+6*rho^2*a_z*
rho_x*a_zz*a_yy*a_xxy-12*rho^2*a_z*rho_y*a_xy*a_yz*a_xxz+12*
rho^2*a_z*rho_y*a_xy*a_zz*a_xxy-12*rho^2*a_z*rho_y*a_xy*a_xz*
a_xyz-12*rho^2*a_z*rho_y*a_xz*a_yz*a_xxy+12*rho^2*a_z*rho_y*
a_xz*a_xxz*a_yy+12*rho^2*a_z*rho_y*a_yz*a_xx*a_xyz-6*rho^2*
a_z*rho_y*a_xx*a_zz*a_xyy-6*rho^2*a_z*rho_y*a_xx*a_yy*a_xzz
-6*rho^2*a_z*rho_y*a_zz*a_yy*a_xxx+6*rho*a_x^2*rho_x*rho_y*
a_zz*a_yyz+6*rho*a_x^2*rho_x*rho_y*a_zzz*a_yy-12*rho*a_x^2*
rho_x*rho_y*a_yzz*a_yz-6*rho*a_x^2*rho_x*rho_z*a_yzz*a_yy-6*
rho*a_x^2*rho_x*rho_z*a_zz*a_yyy+12*rho*a_x^2*rho_x*rho_z*
a_yyz*a_yz+6*rho*a_x^2*rho_z*rho_y*a_zz*a_xyy+12*rho*a_x^2*
rho_z*rho_y*a_yzz*a_xy-6*rho*a_x^2*rho_z*rho_y*a_yy*a_xzz-12*
rho*a_x^2*rho_z*rho_y*a_xz*a_yyz-6*rho*a_x*a_y*rho_x^2*a_zz*
```

```
a_yyz-6*rho*a_x*a_y*rho_x^2*a_zzz*a_yy+12*rho*a_x*a_y*rho_x
^2*a_yzz*a_yz+6*rho*a_x*a_y*rho_y^2*a_zz*a_xxz-12*rho*a_x*a_y
*rho_y^2*a_xzz*a_xz+6*rho*a_x*a_y*rho_y^2*a_xx*a_zzz-6*rho*
a_x*a_y*rho_z^2*a_xxz*a_yy+6*rho*a_x*a_y*rho_z^2*a_xx*a_yyz
+12*rho*a_x*a_y*rho_z^2*a_yz*a_xxy-12*rho*a_x*a_y*rho_z^2*
a_xz*a_xyy-12*rho*a_x*a_z*rho_x^2*a_yyz*a_yz+6*rho*a_x*a_z*
rho_x^2*a_zz*a_yyy+6*rho*a_x*a_z*rho_x^2*a_yzz*a_yy-6*rho*a_x
*a_z*rho_y^2*a_xx*a_yzz+6*rho*a_x*a_z*rho_y^2*a_zz*a_xxy+12*
rho*a_x*a_z*rho_y^2*a_xzz*a_xy-12*rho*a_x*a_z*rho_y^2*a_yz*
a_xxz+12*rho*a_x*a_z*rho_z^2*a_xyy*a_xy-6*rho*a_x*a_z*rho_z
^2*a_xxy*a_yy-6*rho*a_x*a_z*rho_z^2*a_xx*a_yyy-6*rho*a_y^2*
rho_x*rho_y*a_zz*a_xxz+12*rho*a_y^2*rho_x*rho_y*a_xzz*a_xz-6*
rho*a_y^2*rho_x*rho_y*a_xx*a_zzz+12*rho*a_y^2*rho_x*rho_z*
a_yz*a_xxz+6*rho*a_y^2*rho_x*rho_z*a_xx*a_yzz-6*rho*a_y^2*
rho_x*rho_z*a_zz*a_xxy-12*rho*a_y^2*rho_x*rho_z*a_xzz*a_xy
-12*rho*a_y^2*rho_z*rho_y*a_xz*a_xxz+6*rho*a_y^2*rho_z*rho_y*
a_xxx*a_zz+6*rho*a_y^2*rho_z*rho_y*a_xzz*a_xx+12*rho*a_z*a_y*
rho_x^2*a_xz*a_yyz-6*rho*a_z*a_y*rho_x^2*a_zz*a_xyy-12*rho*
a_z*a_y*rho_x^2*a_yzz*a_xy+6*rho*a_z*a_y*rho_x^2*a_yy*a_xzz
+12*rho*a_z*a_y*rho_y^2*a_xz*a_xxz-6*rho*a_z*a_y*rho_y^2*
a_xxx*a_zz-6*rho*a_z*a_y*rho_y^2*a_xzz*a_xx+6*rho*a_z*a_y*
rho_z^2*a_yy*a_xxx+6*rho*a_z*a_y*rho_z^2*a_xyy*a_xx-12*rho*
a_z*a_y*rho_z^2*a_xy*a_xxy-12*rho*a_z^2*rho_x*rho_y*a_yz*
a_xxy+6*rho*a_z^2*rho_x*rho_y*a_xxz*a_yy-6*rho*a_z^2*rho_x*
rho_y*a_xx*a_yyz+12*rho*a_z^2*rho_x*rho_y*a_xz*a_xyy+6*rho*
a_z^2*rho_x*rho_z*a_xxy*a_yy+6*rho*a_z^2*rho_x*rho_z*a_xx*
a_yyy-12*rho*a_z^2*rho_x*rho_z*a_xyy*a_xy+12*rho*a_z^2*rho_z*
rho_y*a_xy*a_xxy-6*rho*a_z^2*rho_z*rho_y*a_xyy*a_xx-6*rho*a_z
^2*rho_z*rho_y*a_yy*a_xxx-12*a_x^2*a_y*rho_x*rho_y*rho_z*
a_yzz+12*a_x^2*a_z*rho_x*rho_y*rho_z*a_yyz+12*a_x*a_y^2*rho_x
*rho_y*rho_z*a_xzz+12*a_x*a_z*a_y*rho_x^2*a_yzz*rho_y-12*a_x*
a_z*a_y*rho_x^2*rho_z*a_yyz-12*a_x*a_z*a_y*rho_x*rho_y^2*
a_xzz+12*a_x*a_z*a_y*rho_x*rho_z^2*a_xyy-12*a_x*a_z*a_y*a_xxy
*rho_z^2*rho_y+12*a_x*a_z*a_y*a_xxz*rho_z*rho_y^2-12*a_x*a_z
^2*rho_x*rho_y*rho_z*a_xyy-12*a_z*a_y^2*rho_x*rho_y*rho_z*
a_xxz+12*a_z^2*a_y*rho_x*rho_y*rho_z*a_xxy-2*a_x^3*a_zzz*
rho_y^3+2*a_x^3*a_yyy*rho_z^3+2*a_y^3*a_zzz*rho_x^3-2*a_y^3*
a_xxx*rho_z^3+2*a_z^3*a_xxx*rho_y^3-2*a_z^3*a_yyy*rho_x^3:
rhodot := -12*rho*rho_x*rho_y*a_x*a_z*rho_xyy*a_zz-12*rho*rho_x*
rho_y*a_x*a_z*rho_xzz*a_yy+24*rho*rho_x*rho_y*a_x*a_z*rho_xyz
*a_yz+12*rho*rho_x*rho_y*a_x*a_xy*rho_yz*a_zz-12*rho*rho_x*
rho_y*a_x*a_xy*a_yz*rho_zz-12*rho*rho_x*rho_y*a_x*a_xz*a_yz*
rho_yz+6*rho^2*rho_x*a_y*rho_zzz*a_xy^2+6*rho^2*rho_x*a_y*
rho_yyz*a_xz^2+6*rho^2*rho_x*a_y*rho_xxz*a_yz^2-6*rho^2*rho_x
*a_z*rho_yzz*a_xy^2-6*rho^2*rho_x*a_z*rho_yyy*a_xz^2-6*rho^2*
rho_x*a_z*rho_xxy*a_yz^2-6*rho^2*rho_y*a_x*rho_zzz*a_xy^2-6*
rho^2*rho_y*a_x*rho_yyz*a_xz^2-6*rho^2*rho_y*a_x*rho_xxz*a_yz
^2+6*rho^2*rho_y*a_z*rho_xzz*a_xy^2+6*rho^2*rho_y*a_z*rho_xyy
*a_xz^2+6*rho^2*rho_y*a_z*rho_xxx*a_yz^2+6*rho^2*rho_z*a_x*
```

```
rho_yzz*a_xy^2+6*rho^2*rho_z*a_x*rho_yyy*a_xz^2+6*rho^2*rho_z
*a_x*rho_xxy*a_yz^2-6*rho^2*rho_z*a_y*rho_xzz*a_xy^2-6*rho^2*
rho_z*a_y*rho_xyy*a_xz^2-6*rho^2*rho_z*a_y*rho_xxx*a_yz^2+6*
rho*rho_x^2*a_y^2*rho_zzz*a_xy-6*rho*rho_x^2*a_y^2*rho_xzz*
a_yz+6*rho*rho_x^2*a_y^2*rho_xyz*a_zz-6*rho*rho_x^2*a_y^2*
rho_yzz*a_xz-12*rho*rho_x^2*a_y*rho_xz*a_yz^2-6*rho*rho_x^2*
a_z^2*a_xz*rho_yyy-6*rho*rho_x^2*a_z^2*rho_xyz*a_yy+6*rho*
rho_x^2*a_z^2*rho_xyy*a_yz+6*rho*rho_x^2*a_z^2*a_xy*rho_yyz
+12*rho*rho_x^2*a_z*rho_xy*a_yz^2+6*rho*rho_y^2*a_x^2*rho_xzz
*a_yz+6*rho*rho_y^2*a_x^2*rho_yzz*a_xz-6*rho*rho_y^2*a_x^2*
rho_zzz*a_xy-6*rho*rho_y^2*a_x^2*rho_xyz*a_zz+12*rho*rho_y^2*
a_x*rho_yz*a_xz^2-6*rho*rho_y^2*a_z^2*rho_xxy*a_xz-6*rho*
rho_y^2*a_z^2*a_xy*rho_xxz+6*rho*rho_y^2*a_z^2*rho_xyz*a_xx
+6*rho*rho_y^2*a_z^2*a_yz*rho_xxx-12*rho*rho_y^2*a_z*rho_xy*
a_xz^2+6*rho*rho_z^2*a_x^2*a_xz*rho_yyy+6*rho*rho_z^2*a_x^2*
rho_xyz*a_yy-6*rho*rho_z^2*a_x^2*rho_xyy*a_yz-6*rho*rho_z^2*
a_x^2*a_xy*rho_yyz-12*rho*rho_z^2*a_x*a_xy^2*rho_yz-6*rho*
rho_z^2*a_y^2*a_yz*rho_xxx+6*rho*rho_z^2*a_y^2*a_xy*rho_xxz
-6*rho*rho_z^2*a_y^2*rho_xyz*a_xx+6*rho*rho_z^2*a_y^2*rho_xxy
*a_xz+12*rho*rho_z^2*a_y*rho_xz*a_xy^2-6*rho_x^3*a_z*a_y*
rho_zz*a_yy+6*rho_x^3*a_z*a_y*rho_yy*a_zz-6*rho_x^2*rho_y*a_x
*rho_zzz*a_y^2-6*rho_x^2*rho_y*a_x*rho_yyz*a_z^2-6*rho_x^2*
rho_y*a_y^2*a_xz*rho_zz+6*rho_x^2*rho_y*a_y^2*a_z*rho_xzz+6*
rho_x^2*rho_y*a_y^2*rho_xz*a_zz-12*rho_x^2*rho_y*a_z^2*a_y*
rho_xyz-12*rho_x^2*rho_y*a_z^2*rho_yz*a_xy+12*rho_x^2*rho_y*
a_z^2*a_yz*rho_xy-6*rho_x^2*rho_y*a_z^2*rho_xz*a_yy+6*rho_x
^2*rho_y*a_z^2*a_xz*rho_yy+6*rho_x^2*rho_z*a_x*rho_yzz*a_y
^2+6*rho_x^2*rho_z*a_x*a_z^2*rho_yyy+6*rho_x^2*rho_z*a_y^2*
rho_xy*a_zz+12*rho_x^2*rho_z*a_y^2*rho_yz*a_xz+12*rho_x^2*
rho_z*a_y^2*a_z*rho_xyz-6*rho_x^2*rho_z*a_y^2*a_xy*rho_zz-12*
rho_x^2*rho_z*a_y^2*rho_xz*a_yz-6*rho_x^2*rho_z*a_z^2*a_y*
rho_xyy-6*rho_x^2*rho_z*a_z^2*rho_xy*a_yy+6*rho_x^2*rho_z*a_z
^2*rho_yy*a_xy-6*rho_x*rho_y^2*a_x^2*rho_yzz*a_z+6*rho_x*
rho_y^2*a_x^2*rho_zzz*a_y+6*rho_x*rho_y^2*a_x^2*a_yz*rho_zz
-6*rho_x*rho_y^2*a_x^2*rho_yz*a_zz+12*rho_x*rho_y^2*a_x*a_z
^2*rho_xyz+6*rho_x*rho_y^2*a_y*a_z^2*rho_xxz-12*rho_x*rho_y
^2*a_z^2*rho_xy*a_xz+6*rho_x*rho_y^2*a_z^2*rho_yz*a_xx-6*
rho_x*rho_y^2*a_z^2*a_yz*rho_xx-6*rho_x^3*a_y^2*rho_yz*a_zz
-6*rho_x^3*a_y^2*rho_yzz*a_z+6*rho_x^3*a_y^2*a_yz*rho_zz+6*
rho_x^3*a_z^2*a_y*rho_yyz-6*rho_x^3*a_z^2*a_yz*rho_yy+6*rho_x
^3*a_z^2*rho_yz*a_yy+6*rho_x^2*rho_y*a_z^3*rho_xyy-6*rho_x^2*
rho_z*rho_xzz*a_y^3-6*rho_x*rho_y^2*a_z^3*rho_xxy+6*rho_x*
rho_z^2*rho_xxz*a_y^3-6*rho_y^3*a_x^2*a_xz*rho_zz+6*rho_y^3*
a_x^2*a_z*rho_xzz+6*rho_y^3*a_x^2*rho_xz*a_zz-6*rho_y^3*a_z
^2*a_x*rho_xxz-6*rho_y^3*a_z^2*a_xx*rho_xz+6*rho_y^3*a_z^2*
a_xz*rho_xx+6*rho_z*rho_y^2*rho_yzz*a_x^3-6*rho_z^2*rho_y*
rho_yyz*a_x^3+6*rho_z^3*a_x^2*rho_yy*a_xy-6*rho_z^3*a_x^2*a_y
*rho_xyy-6*rho_z^3*a_x^2*rho_xy*a_yy+6*rho_z^3*a_y^2*a_x*
rho_xxy-6*rho_z^3*a_y^2*a_xy*rho_xx-12*rho^2*rho_x*a_y*a_xy*
```

```
rho_yzz*a_xz-12*rho^2*rho_x*a_y*a_xy*rho_xzz*a_yz+12*rho^2*
rho_x*a_y*a_xy*rho_xyz*a_zz+12*rho^2*rho_x*a_y*a_xz*rho_xzz*
a_yy-12*rho^2*rho_x*a_y*a_xz*rho_xyz*a_yz+12*rho^2*rho_x*a_y*
rho_yzz*a_yz*a_xx-6*rho^2*rho_x*a_y*a_xx*rho_yyz*a_zz-6*rho
^2*rho_x*a_y*a_xx*rho_zzz*a_yy-6*rho^2*rho_x*a_y*rho_xxz*a_zz
*a_yy+12*rho^2*rho_x*a_z*a_xy*rho_yyz*a_xz-12*rho^2*rho_x*a_z
*a_xy*rho_xyy*a_zz+12*rho^2*rho_x*a_z*a_xy*rho_xyz*a_yz-12*
rho^2*rho_x*a_z*a_xz*rho_xyz*a_yy+12*rho^2*rho_x*a_z*a_xz*
rho_xyy*a_yz-12*rho^2*rho_x*a_z*rho_yyz*a_yz*a_xx+6*rho^2*
rho_x*a_z*a_xx*rho_yyy*a_zz+6*rho^2*rho_x*a_z*a_xx*rho_yzz*
a_yy+6*rho^2*rho_x*a_z*rho_xxy*a_zz*a_yy+12*rho^2*rho_y*a_x*
a_xy*rho_yzz*a_xz+12*rho^2*rho_y*a_x*a_xy*rho_xzz*a_yz-12*rho
^2*rho_y*a_x*a_xy*rho_xyz*a_zz-12*rho^2*rho_y*a_x*a_xz*
rho_xzz*a_yy+12*rho^2*rho_y*a_x*a_xz*rho_xyz*a_yz-12*rho^2*
rho_y*a_x*rho_yzz*a_yz*a_xx+6*rho^2*rho_y*a_x*a_xx*rho_yyz*
a_zz+6*rho^2*rho_y*a_x*a_xx*rho_zzz*a_yy+6*rho^2*rho_y*a_x*
rho_xxz*a_zz*a_yy+12*rho^2*rho_y*a_z*a_xy*rho_xxy*a_zz-12*rho
^2*rho_y*a_z*a_xy*rho_xxz*a_yz-12*rho^2*rho_y*a_z*a_xy*a_xz*
rho_xyz+12*rho^2*rho_y*a_z*a_xz*rho_xxz*a_yy-12*rho^2*rho_y*
a_z*a_xz*rho_xxy*a_yz+12*rho^2*rho_y*a_z*rho_xyz*a_yz*a_xx-6*
rho^2*rho_y*a_z*a_xx*rho_xzz*a_yy-6*rho^2*rho_y*a_z*a_xx*
rho_xyy*a_zz-6*rho^2*rho_y*a_z*rho_xxx*a_zz*a_yy-12*rho^2*
rho_z*a_x*a_xy*rho_yyz*a_xz-12*rho^2*rho_z*a_x*a_xy*rho_xyz*
a_yz+12*rho^2*rho_z*a_x*a_xy*rho_xyy*a_zz+12*rho^2*rho_z*a_x*
a_xz*rho_xyz*a_yy-12*rho^2*rho_z*a_x*a_xz*rho_xyy*a_yz+12*rho
^2*rho_z*a_x*rho_yyz*a_yz*a_xx-6*rho^2*rho_z*a_x*a_xx*rho_yzz
*a_yy-6*rho^2*rho_z*a_x*a_xx*rho_yyy*a_zz-6*rho^2*rho_z*a_x*
rho_xxy*a_zz*a_yy-12*rho^2*rho_z*a_y*a_xy*rho_xxy*a_zz+12*rho
^2*rho_z*a_y*a_xy*rho_xxz*a_yz+12*rho^2*rho_z*a_y*a_xy*a_xz*
rho_xyz-12*rho^2*rho_z*a_y*a_xz*rho_xxz*a_yy+12*rho^2*rho_z*
a_y*a_xz*rho_xxy*a_yz-12*rho^2*rho_z*a_y*rho_xyz*a_yz*a_xx+6*
rho^2*rho_z*a_y*a_xx*rho_xzz*a_yy+6*rho^2*rho_z*a_y*a_xx*
rho_xyy*a_zz+6*rho^2*rho_z*a_y*rho_xxx*a_zz*a_yy-6*rho*rho_x
^2*a_x*a_y*rho_yyz*a_zz-6*rho*rho_x^2*a_x*a_y*rho_zzz*a_yy
+12*rho*rho_x^2*a_x*a_y*rho_yzz*a_yz-12*rho*rho_x^2*a_x*a_z*
rho_yyz*a_yz+6*rho*rho_x^2*a_x*a_z*rho_yyy*a_zz+6*rho*rho_x
^2*a_x*a_z*rho_yzz*a_yy+12*rho*rho_x^2*a_y*a_z*rho_yyz*a_xz
+6*rho*rho_x^2*a_y*a_z*rho_xzz*a_yy-6*rho*rho_x^2*a_y*a_z*
rho_xyy*a_zz-12*rho*rho_x^2*a_y*a_z*rho_yzz*a_xy-12*rho*rho_x
^2*a_y*a_xy*rho_yz*a_zz+12*rho*rho_x^2*a_y*a_xy*a_yz*rho_zz
+12*rho*rho_x^2*a_y*a_xz*a_yz*rho_yz-12*rho*rho_x^2*a_y*a_xz*
rho_zz*a_yy+12*rho*rho_x^2*a_y*rho_xz*a_yy*a_zz-12*rho*rho_x
^2*a_z*a_xy*a_yz*rho_yz+12*rho*rho_x^2*a_z*a_xy*rho_yy*a_zz
+12*rho*rho_x^2*a_z*a_xz*rho_yz*a_yy-12*rho*rho_x^2*a_z*a_xz*
a_yz*rho_yy-12*rho*rho_x^2*a_z*rho_xy*a_yy*a_zz+6*rho*rho_x*
rho_y*a_x^2*rho_yyz*a_zz+6*rho*rho_x*rho_y*a_x^2*rho_zzz*a_yy
-12*rho*rho_x*rho_y*a_x^2*rho_yzz*a_yz+12*rho*rho_x*rho_y*a_x
*rho_xz*a_yz^2-6*rho*rho_x*rho_y*a_y^2*rho_xxz*a_zz+12*rho*
rho_x*rho_y*a_y^2*a_xz*rho_xzz-6*rho*rho_x*rho_y*a_y^2*a_xx*
```

```
rho_zzz-12*rho*rho_x*rho_y*a_y*rho_yz*a_xz^2-6*rho*rho_x*
rho_y*a_z^2*rho_yyz*a_xx+6*rho*rho_x*rho_y*a_z^2*rho_xxz*a_yy
-12*rho*rho_x*rho_y*a_z^2*rho_xxy*a_yz+12*rho*rho_x*rho_y*a_z
^2*rho_xyy*a_xz+12*rho*rho_x*rho_y*a_z*rho_yy*a_xz^2-12*rho*
rho_x*rho_y*a_z*rho_xx*a_yz^2-6*rho*rho_x*rho_z*a_x^2*rho_yyy
*a_zz-6*rho*rho_x*rho_z*a_x^2*rho_yzz*a_yy+12*rho*rho_x*rho_z
*a_x^2*rho_yyz*a_yz-12*rho*rho_x*rho_z*a_x*rho_xy*a_yz^2-12*
rho*rho_x*rho_z*a_y^2*rho_xzz*a_xy+6*rho*rho_x*rho_z*a_y^2*
rho_yzz*a_xx-6*rho*rho_x*rho_z*a_y^2*rho_xxy*a_zz+12*rho*
rho_x*rho_z*a_y^2*rho_xxz*a_yz-12*rho*rho_x*rho_z*a_y*rho_zz*
a_xy^2+12*rho*rho_x*rho_z*a_y*rho_xx*a_yz^2-12*rho*rho_x*
rho_z*a_z^2*rho_xyy*a_xy+6*rho*rho_x*rho_z*a_z^2*a_xx*rho_yyy
+6*rho*rho_x*rho_z*a_z^2*rho_xxy*a_yy+12*rho*rho_x*rho_z*a_z*
a_xy^2*rho_yz-12*rho*rho_y^2*a_x*a_y*a_xz*rho_xzz+6*rho*rho_y
^2*a_x*a_y*a_xx*rho_zzz+6*rho*rho_y^2*a_x*a_y*rho_xxz*a_zz
+12*rho*rho_y^2*a_x*a_z*rho_xzz*a_xy+6*rho*rho_y^2*a_x*a_z*
rho_xxy*a_zz-6*rho*rho_y^2*a_x*a_z*rho_yzz*a_xx-12*rho*rho_y
^2*a_x*a_z*rho_xxz*a_yz-12*rho*rho_y^2*a_x*a_xy*a_xz*rho_zz
+12*rho*rho_y^2*a_x*a_xy*rho_xz*a_zz-12*rho*rho_y^2*a_x*a_xz*
rho_xz*a_yz+12*rho*rho_y^2*a_x*a_xx*a_yz*rho_zz-12*rho*rho_y
^2*a_x*a_xx*rho_yz*a_zz-6*rho*rho_y^2*a_z*a_y*rho_xzz*a_xx-6*
rho*rho_y^2*a_z*a_y*rho_xxx*a_zz+12*rho*rho_y^2*a_z*a_y*
rho_xxz*a_xz+12*rho*rho_y^2*a_z*a_xy*rho_xz*a_xz-12*rho*rho_y
^2*a_z*a_xy*a_zz*rho_xx+12*rho*rho_y^2*a_z*rho_xx*a_yz*a_xz
+12*rho*rho_y^2*a_z*a_xx*rho_xy*a_zz-12*rho*rho_y^2*a_z*a_xx*
rho_xz*a_yz-12*rho*rho_z*rho_y*a_x^2*rho_yyz*a_xz+12*rho*
rho_z*rho_y*a_x^2*rho_yzz*a_xy+6*rho*rho_z*rho_y*a_x^2*
rho_xyy*a_zz-6*rho*rho_z*rho_y*a_x^2*rho_xzz*a_yy+12*rho*
rho_z*rho_y*a_x*rho_zz*a_xy^2-12*rho*rho_z*rho_y*a_x*rho_yy*
a_xz^2+6*rho*rho_z*rho_y*a_y^2*rho_xzz*a_xx+6*rho*rho_z*rho_y
*a_y^2*rho_xxx*a_zz-12*rho*rho_z*rho_y*a_y^2*rho_xxz*a_xz+12*
rho*rho_z*rho_y*a_y*rho_xy*a_xz^2+12*rho*rho_z*rho_y*a_z^2*
rho_xxy*a_xy-6*rho*rho_z*rho_y*a_z^2*a_yy*rho_xxx-6*rho*rho_z
*rho_y*a_z^2*rho_xyy*a_xx-12*rho*rho_z*rho_y*a_z*rho_xz*a_xy
^2+12*rho*rho_z^2*a_x*a_y*rho_xxy*a_yz-12*rho*rho_z^2*a_x*a_y
*rho_xyy*a_xz-6*rho*rho_z^2*a_x*a_y*rho_xxz*a_yy+6*rho*rho_z
^2*a_x*a_y*rho_yyz*a_xx+12*rho*rho_z^2*a_x*a_z*rho_xyy*a_xy
-6*rho*rho_z^2*a_x*a_z*a_xx*rho_yyy-6*rho*rho_z^2*a_x*a_z*
rho_xxy*a_yy+12*rho*rho_z^2*a_x*a_xy*a_xz*rho_yy+12*rho*rho_z
^2*a_x*a_xy*a_yz*rho_xy-12*rho*rho_z^2*a_x*rho_xy*a_xz*a_yy
+12*rho*rho_z^2*a_x*a_xx*rho_yz*a_yy-12*rho*rho_z^2*a_x*a_xx*
a_yz*rho_yy-12*rho*rho_z^2*a_y*a_z*rho_xxy*a_xy+6*rho*rho_z
^2*a_y*a_z*a_yy*rho_xxx+6*rho*rho_z^2*a_y*a_z*rho_xyy*a_xx
-12*rho*rho_z^2*a_y*a_xy*rho_xy*a_xz-12*rho*rho_z^2*a_y*a_xy*
a_yz*rho_xx+12*rho*rho_z^2*a_y*rho_xx*a_yy*a_xz-12*rho*rho_z
^2*a_y*a_xx*rho_xz*a_yy+12*rho*rho_z^2*a_y*a_xx*a_yz*rho_xy
+12*rho_x^2*rho_y*a_x*a_y*rho_yz*a_zz-12*rho_x^2*rho_y*a_x*
a_y*a_yz*rho_zz+12*rho_x^2*rho_y*a_x*a_y*rho_yzz*a_z+6*rho_x
^2*rho_y*a_x*a_z*rho_zz*a_yy-6*rho_x^2*rho_y*a_x*a_z*rho_yy*
```

```
a_zz-12*rho_x^2*rho_y*a_z*a_y*rho_xy*a_zz+12*rho_x^2*rho_y*
a_z*a_y*a_xy*rho_zz+6*rho_x^2*rho_z*a_x*a_y*rho_zz*a_yy-6*
rho_x^2*rho_z*a_x*a_y*rho_yy*a_zz-12*rho_x^2*rho_z*a_x*a_y*
rho_yyz*a_z-12*rho_x^2*rho_z*a_x*a_z*rho_yz*a_yy+12*rho_x^2*
rho_z*a_x*a_z*a_yz*rho_yy+12*rho_x^2*rho_z*a_z*a_y*rho_xz*
a_yy-12*rho_x^2*rho_z*a_z*a_y*a_xz*rho_yy+12*rho_x*rho_y^2*
a_x*a_y*a_xz*rho_zz-12*rho_x*rho_y^2*a_x*a_y*a_z*rho_xzz-12*
rho_x*rho_y^2*a_x*a_y*rho_xz*a_zz+12*rho_x*rho_y^2*a_x*a_z*
rho_xy*a_zz-12*rho_x*rho_y^2*a_x*a_z*a_xy*rho_zz-6*rho_x*
rho_y^2*a_z*a_y*a_xx*rho_zz+6*rho_x*rho_y^2*a_z*a_y*a_zz*
rho_xx-12*rho_x*rho_z*rho_y*a_x^2*a_y*rho_yzz+12*rho_x*rho_z*
rho_y*a_x^2*rho_yyz*a_z-6*rho_x*rho_z*rho_y*a_x^2*rho_zz*a_yy
+6*rho_x*rho_z*rho_y*a_x^2*rho_yy*a_zz+12*rho_x*rho_z*rho_y*
a_x*rho_xzz*a_y^2-12*rho_x*rho_z*rho_y*a_x*a_z^2*rho_xyy-6*
rho_x*rho_z*rho_y*a_y^2*a_zz*rho_xx-12*rho_x*rho_z*rho_y*a_y
^2*a_z*rho_xxz+6*rho_x*rho_z*rho_y*a_y^2*a_xx*rho_zz+12*rho_x
*rho_z*rho_y*a_z^2*a_y*rho_xxy+6*rho_x*rho_z*rho_y*a_z^2*a_yy
*rho_xx-6*rho_x*rho_z*rho_y*a_z^2*rho_yy*a_xx+12*rho_x*rho_z
^2*a_x*a_y*a_z*rho_xyy-12*rho_x*rho_z^2*a_x*a_y*rho_xz*a_yy
+12*rho_x*rho_z^2*a_x*a_y*a_xz*rho_yy+12*rho_x*rho_z^2*a_x*
a_z*rho_xy*a_yy-12*rho_x*rho_z^2*a_x*a_z*rho_yy*a_xy-6*rho_x*
rho_z^2*a_y*a_z*a_yy*rho_xx+6*rho_x*rho_z^2*a_y*a_z*rho_yy*
a_xx+6*rho_z*rho_y^2*a_x*a_y*a_zz*rho_xx+12*rho_z*rho_y^2*a_x
*a_y*a_z*rho_xxz-6*rho_z*rho_y^2*a_x*a_y*a_xx*rho_zz+12*rho_z
*rho_y^2*a_x*a_z*a_yz*rho_xx-12*rho_z*rho_y^2*a_x*a_z*rho_yz*
a_xx+12*rho_z*rho_y^2*a_z*a_y*a_xx*rho_xz-12*rho_z*rho_y^2*
a_z*a_y*a_xz*rho_xx+12*rho_z^2*rho_y*a_x*a_y*rho_yz*a_xx-12*
rho_z^2*rho_y*a_x*a_y*a_z*rho_xxy-12*rho_z^2*rho_y*a_x*a_y*
a_yz*rho_xx-6*rho_z^2*rho_y*a_x*a_z*a_yy*rho_xx+6*rho_z^2*
rho_y*a_x*a_z*rho_yy*a_xx+12*rho_z^2*rho_y*a_y*a_z*a_xy*
rho_xx-12*rho_z^2*rho_y*a_y*a_z*a_xx*rho_xy+2*rho_x^3*rho_zzz
*a_y^3-2*rho_x^3*a_z^3*rho_yyy-2*rho_y^3*rho_zzz*a_x^3+2*
rho_y^3*a_z^3*rho_xxx+2*rho_z^3*rho_yyy*a_x^3-2*rho_z^3*a_y
^3*rho_xxx+12*rho*rho_x*rho_y*a_x*a_xz*rho_zz*a_yy-12*rho*
rho_x*rho_y*a_x*rho_xz*a_yy*a_zz-24*rho*rho_x*rho_y*a_y*a_z*
a_xz*rho_xyz+12*rho*rho_x*rho_y*a_y*a_z*rho_xxy*a_zz+12*rho*
rho_x*rho_y*a_y*a_z*rho_yzz*a_xx-12*rho*rho_x*rho_y*a_y*a_xy*
rho_xz*a_zz+12*rho*rho_x*rho_y*a_y*a_xy*a_xz*rho_zz+12*rho*
rho_x*rho_y*a_y*a_xz*rho_xz*a_yz-12*rho*rho_x*rho_y*a_y*a_xx*
a_yz*rho_zz+12*rho*rho_x*rho_y*a_y*a_xx*rho_yz*a_zz+12*rho*
rho_x*rho_y*a_z*a_xy*rho_xz*a_yz-12*rho*rho_x*rho_y*a_z*a_xy*
rho_yz*a_xz-12*rho*rho_x*rho_y*a_z*rho_xz*a_xz*a_yy+12*rho*
rho_x*rho_y*a_z*a_yz*a_xx*rho_yz+12*rho*rho_x*rho_y*a_z*a_zz*
a_yy*rho_xx-12*rho*rho_x*rho_y*a_z*a_zz*rho_yy*a_xx-24*rho*
rho_x*rho_z*a_x*a_y*rho_xyz*a_yz+12*rho*rho_x*rho_z*a_x*a_y*
rho_xzz*a_yy+12*rho*rho_x*rho_z*a_x*a_y*rho_xyy*a_zz-12*rho*
rho_x*rho_z*a_x*a_xy*rho_yy*a_zz+12*rho*rho_x*rho_z*a_x*a_xy*
a_yz*rho_yz-12*rho*rho_x*rho_z*a_x*a_xz*rho_yz*a_yy+12*rho*
rho_x*rho_z*a_x*a_xz*a_yz*rho_yy+12*rho*rho_x*rho_z*a_x*
```

```
rho_xy*a_yy*a_zz-12*rho*rho_x*rho_z*a_y*a_z*rho_xxz*a_yy-12*
rho*rho_x*rho_z*a_y*a_z*rho_yyz*a_xx+24*rho*rho_x*rho_z*a_y*
a_z*rho_xyz*a_xy+12*rho*rho_x*rho_z*a_y*a_xy*rho_yz*a_xz+12*
rho*rho_x*rho_z*a_y*a_xy*rho_xy*a_zz-12*rho*rho_x*rho_z*a_y*
a_yz*rho_xy*a_xz-12*rho*rho_x*rho_z*a_y*a_yz*a_xx*rho_yz-12*
rho*rho_x*rho_z*a_y*a_zz*a_yy*rho_xx+12*rho*rho_x*rho_z*a_y*
a_yy*a_xx*rho_zz-12*rho*rho_x*rho_z*a_z*a_xy*a_xz*rho_yy-12*
rho*rho_x*rho_z*a_z*a_xy*a_yz*rho_xy+12*rho*rho_x*rho_z*a_z*
rho_xy*a_xz*a_yy-12*rho*rho_x*rho_z*a_z*a_xx*rho_yz*a_yy+12*
rho*rho_x*rho_z*a_z*a_xx*a_yz*rho_yy-12*rho*rho_z*rho_y*a_x*
a_y*rho_yzz*a_xx-12*rho*rho_z*rho_y*a_x*a_y*rho_xxy*a_zz+24*
rho*rho_z*rho_y*a_x*a_y*a_xz*rho_xyz+12*rho*rho_z*rho_y*a_x*
a_z*rho_yyz*a_xx-24*rho*rho_z*rho_y*a_x*a_z*rho_xyz*a_xy+12*
rho*rho_z*rho_y*a_x*a_z*rho_xxz*a_yy-12*rho*rho_z*rho_y*a_x*
a_xy*rho_xz*a_yz-12*rho*rho_z*rho_y*a_x*a_xy*rho_xy*a_zz+12*
rho*rho_z*rho_y*a_x*rho_xz*a_xz*a_yy+12*rho*rho_z*rho_y*a_x*
a_yz*rho_xy*a_xz-12*rho*rho_z*rho_y*a_x*a_yy*a_xx*rho_zz+12*
rho*rho_z*rho_y*a_x*a_zz*rho_yy*a_xx-12*rho*rho_z*rho_y*a_y*
a_xy*rho_xz*a_xz+12*rho*rho_z*rho_y*a_y*a_xy*a_zz*rho_xx-12*
rho*rho_z*rho_y*a_y*rho_xx*a_yz*a_xz-12*rho*rho_z*rho_y*a_y*
a_xx*rho_xy*a_zz+12*rho*rho_z*rho_y*a_y*a_xx*rho_xz*a_yz+12*
rho*rho_z*rho_y*a_z*a_xy*rho_xy*a_xz+12*rho*rho_z*rho_y*a_z*
a_xy*a_yz*rho_xx-12*rho*rho_z*rho_y*a_z*rho_xx*a_yy*a_xz+12*
rho*rho_z*rho_y*a_z*a_xx*rho_xz*a_yy-12*rho*rho_z*rho_y*a_z*
a_xx*a_yz*rho_xy+24*rho_x*rho_z*rho_y*a_x*a_y*rho_xz*a_yz-24*
rho_x*rho_z*rho_y*a_x*a_y*rho_yz*a_xz+24*rho_x*rho_z*rho_y*
a_x*a_z*rho_yz*a_xy-24*rho_x*rho_z*rho_y*a_x*a_z*a_yz*rho_xy
+24*rho_x*rho_z*rho_y*a_z*a_y*rho_xy*a_xz-24*rho_x*rho_z*
rho_y*a_z*a_y*a_xy*rho_xz+12*rho_x*rho_y^2*a_z^2*a_xy*rho_xz
+6*rho_x*rho_z^2*a_x^2*rho_yz*a_yy+6*rho_x*rho_z^2*a_x^2*
rho_yyz*a_y-6*rho_x*rho_z^2*a_x^2*a_yz*rho_yy-6*rho_x*rho_z
^2*a_x^2*a_z*rho_yyy-12*rho_x*rho_z^2*a_x*rho_xyz*a_y^2+6*
rho_x*rho_z^2*a_y^2*a_yz*rho_xx+12*rho_x*rho_z^2*a_y^2*a_xy*
rho_xz-6*rho_x*rho_z^2*a_y^2*a_z*rho_xxy-12*rho_x*rho_z^2*a_y
^2*rho_xy*a_xz-6*rho_x*rho_z^2*a_y^2*rho_yz*a_xx+6*rho_y^3*
a_z*a_x*a_xx*rho_zz-6*rho_y^3*a_z*a_x*a_zz*rho_xx-12*rho_z*
rho_y^2*a_x^2*rho_xz*a_yz-6*rho_z*rho_y^2*a_x^2*a_y*rho_xzz
-6*rho_z*rho_y^2*a_x^2*rho_xy*a_zz+6*rho_z*rho_y^2*a_x^2*a_xy
*rho_zz+12*rho_z*rho_y^2*a_x^2*rho_yz*a_xz-12*rho_z*rho_y^2*
a_x^2*a_z*rho_xyz+6*rho_z*rho_y^2*a_x*a_z^2*rho_xxy-6*rho_z*
rho_y^2*a_y*a_z^2*rho_xxx-6*rho_z*rho_y^2*a_z^2*a_xy*rho_xx
+6*rho_z*rho_y^2*a_z^2*a_xx*rho_xy-12*rho_z^2*rho_y*a_x^2*
rho_yz*a_xy-6*rho_z^2*rho_y*a_x^2*a_xz*rho_yy+6*rho_z^2*rho_y
*a_x^2*rho_xz*a_yy+12*rho_z^2*rho_y*a_x^2*a_yz*rho_xy+12*
rho_z^2*rho_y*a_x^2*a_y*rho_xyz+6*rho_z^2*rho_y*a_x^2*a_z*
rho_xyy-6*rho_z^2*rho_y*a_x*rho_xxz*a_y^2+6*rho_z^2*rho_y*a_y
^2*a_z*rho_xxx-6*rho_z^2*rho_y*a_y^2*a_xx*rho_xz+6*rho_z^2*
rho_y*a_y^2*a_xz*rho_xx+6*rho_z^3*a_y*a_x*a_yy*rho_xx-6*rho_z
^3*a_y*a_x*rho_yy*a_xx+6*rho_z^3*a_y^2*a_xx*rho_xy:
```

B Unskewed Polynomials $\dot{a}'$ and $\dot{\rho}'$

```
adotUnskew := 6*a_x^3*a_yzz*rho_y^2*rho_z+12*a_x^2*a_xy*a_yzz*
    rho*rho_y*rho_z+6*a_x^2*a_xz*a_yzz*rho*rho_y^2+6*a_x^2*a_y*
    a_yyz*rho_x*rho_z^2+6*a_x^2*a_y*a_zzz*rho_x*rho_y^2+6*a_x^2*
    a_yy*a_zzz*rho*rho_x*rho_y+12*a_x^2*a_yyz*a_yz*rho*rho_x*
    rho_z+12*a_x^2*a_yyz*a_z*rho_x*rho_y*rho_z+6*a_x^2*a_yyz*a_zz
    *rho*rho_x*rho_y+6*a_x*a_xx*a_y*a_yyz*rho*rho_z^2+6*a_x*a_xx*
    a_y*a_zzz*rho*rho_y^2+6*a_x*a_xx*a_yy*a_zzz*rho^2*rho_y+12*
    a_x*a_xx*a_yyz*a_yz*rho^2*rho_z+12*a_x*a_xx*a_yyz*a_z*rho*
    rho_y*rho_z+6*a_x*a_xx*a_yyz*a_zz*rho^2*rho_y+6*a_x*a_xy^2*
    a_yzz*rho^2*rho_z+12*a_x*a_xy*a_xz*a_yzz*rho^2*rho_y+12*a_x*
    a_xyz*a_xz*a_yz*rho^2*rho_y+24*a_x*a_xyz*a_yz*a_z*rho*rho_x*
    rho_y+12*a_x*a_xyz*a_z^2*rho_x*rho_y^2+6*a_x*a_y^2*a_yzz*
    rho_x^2*rho_z+12*a_x*a_y*a_yz*a_yzz*rho*rho_x^2+12*a_x*a_y*
    a_yzz*a_z*rho_x^2*rho_y+6*a_x*a_yy*a_yzz*a_z*rho*rho_x^2+12*
    a_xx*a_xyz*a_yz*a_z*rho^2*rho_y+6*a_xx*a_y^2*a_yzz*rho*rho_x*
    rho_z+12*a_xx*a_y*a_yz*a_yzz*rho^2*rho_x+12*a_xx*a_y*a_yzz*
    a_z*rho*rho_x*rho_y+6*a_xx*a_yy*a_yzz*a_z*rho^2*rho_x+6*a_xxx
    *a_yz^2*a_z*rho^2*rho_y+6*a_xxx*a_yz*a_z^2*rho*rho_y^2+12*
    a_xy*a_xz*a_yyz*a_z*rho^2*rho_x+6*a_xy*a_yyz*a_z^2*rho*rho_x
    ^2+6*a_xyz*a_y^2*a_zz*rho*rho_x^2+6*a_xz^2*a_y*a_yyz*rho^2*
    rho_x+12*a_xz*a_y*a_yyz*a_z*rho*rho_x^2+2*a_y^3*a_zzz*rho_x
    ^3+6*a_y*a_yyz*a_z^2*rho_x^3:
rhodotUnskew := 12*a_x^2*a_yz*rho*rho_x*rho_yyz*rho_z+6*a_x^2*
    a_yz*rho*rho_xzz*rho_y^2+6*a_x^2*a_yz*rho_x*rho_y^2*rho_zz
    +12*a_x^2*a_yz*rho_xy*rho_y*rho_z^2+6*a_x^2*a_zz*rho*rho_x*
    rho_y*rho_yyz+6*a_x^2*a_zz*rho*rho_xyy*rho_y*rho_z+6*a_x^2*
    a_zz*rho_x*rho_y*rho_yy*rho_z+6*a_x^2*a_zz*rho_xz*rho_y^3+12*
    a_x*a_xx*a_yz*rho^2*rho_yyz*rho_z+12*a_x*a_xx*a_yz*rho*rho_y
    ^2*rho_zz+12*a_x*a_xz*a_yz*rho^2*rho_xyz*rho_y+12*a_x*a_xz*
    a_yz*rho*rho_x*rho_yy*rho_z+12*a_x*a_xz*a_yz*rho*rho_xy*rho_y
    *rho_z+12*a_x*a_y*a_yz*rho*rho_x^2*rho_yzz+12*a_x*a_y*a_yz*
    rho*rho_xxy*rho_z^2+24*a_x*a_y*a_yz*rho_x*rho_xz*rho_y*rho_z
    +12*a_x*a_y*a_z*rho_x^2*rho_y*rho_yzz+12*a_x*a_y*a_zz*rho*
    rho_x*rho_xyy*rho_z+6*a_x*a_y*a_zz*rho*rho_xxz*rho_y^2+12*a_x
    *a_y*a_zz*rho_x^2*rho_y*rho_yz+6*a_x*a_y*a_zz*rho_xx*rho_y^2*
    rho_z+6*a_x*a_yy*a_zz*rho^2*rho_xxz*rho_y+12*a_x*a_yy*a_zz*
    rho*rho_x*rho_xy*rho_z+6*a_x*a_yz^2*rho^2*rho_xxy*rho_z+12*
    a_x*a_yz^2*rho*rho_x*rho_xz*rho_y+24*a_x*a_yz*a_z*rho*rho_x*
    rho_xyz*rho_y+12*a_x*a_yz*a_z*rho_x^2*rho_yy*rho_z+12*a_x*
    a_yz*a_z*rho_xx*rho_y^2*rho_z+6*a_x*a_z^2*rho_x^2*rho_yyy*
    rho_z+12*a_x*a_z^2*rho_x*rho_xyz*rho_y^2+6*a_x*a_z^2*rho_xxy*
    rho_y^2*rho_z+6*a_x*a_z*a_zz*rho*rho_x^2*rho_yyy+6*a_x*a_z*
    a_zz*rho*rho_xxy*rho_y^2+12*a_x*a_z*a_zz*rho_x*rho_xy*rho_y
    ^2+12*a_xx*a_y*a_yz*rho^2*rho_x*rho_yzz+12*a_xx*a_y*a_yz*rho*
    rho_xy*rho_z^2+12*a_xx*a_y*a_yz*rho*rho_xz*rho_y*rho_z+12*
```

```
a_xx*a_yz*a_z*rho^2*rho_xyz*rho_y+12*a_xx*a_yz*a_z*rho*rho_x*
rho_y*rho_yz+12*a_xx*a_yz*a_z*rho*rho_x*rho_yy*rho_z+12*a_xz*
a_y*a_yz*rho^2*rho_xxy*rho_z+12*a_xz*a_y*a_yz*rho*rho_x^2*
rho_yz+12*a_xz*a_y*a_yz*rho*rho_x*rho_xz*rho_y+12*a_xz*a_yz*
a_z*rho^2*rho_x*rho_xyy+12*a_xz*a_yz*a_z*rho*rho_xx*rho_y
^2+12*a_y^2*a_yz*rho*rho_x*rho_xxz*rho_z+6*a_y^2*a_yz*rho_x
^3*rho_zz+6*a_y^2*a_yz*rho_x*rho_xx*rho_z^2+6*a_y^2*a_zz*rho*
rho_x^2*rho_xyz+6*a_y^2*a_zz*rho*rho_xxx*rho_y*rho_z+6*a_y^2*
a_zz*rho_x^2*rho_xy*rho_z+6*a_y^2*a_zz*rho_x^2*rho_xz*rho_y
+6*a_y*a_yy*a_zz*rho^2*rho_xxx*rho_z+12*a_y*a_yy*a_zz*rho*
rho_x^2*rho_xz+6*a_y*a_yz^2*rho^2*rho_x*rho_xxz+12*a_y*a_yz
^2*rho*rho_x*rho_xx*rho_z+6*a_y*a_z^2*rho_x^3*rho_yyz+12*a_y*
a_z^2*rho_x*rho_xxy*rho_y*rho_z+6*a_y*a_z^2*rho_x*rho_xxz*
rho_y^2+12*a_y*a_z*a_zz*rho*rho_x*rho_xxy*rho_y+6*a_y*a_z*
a_zz*rho_x^3*rho_yy+6*a_y*a_z*a_zz*rho_x*rho_xx*rho_y^2+6*
a_yy*a_z*a_zz*rho^2*rho_x*rho_xxy+12*a_yy*a_z*a_zz*rho*rho_x*
rho_xx*rho_y+6*a_yz^2*a_z*rho^2*rho_xxx*rho_y+12*a_yz^2*a_z*
rho*rho_x^2*rho_xy+6*a_yz*a_z^2*rho*rho_x^2*rho_xyy+6*a_yz*
a_z^2*rho*rho_xxx*rho_y^2+12*a_yz*a_z^2*rho_x^2*rho_xy*rho_y
+6*a_z^3*rho_x^2*rho_xyy*rho_y+2*a_z^3*rho_xxx*rho_y^3:
```