Dissertation

# Higher Order Moment Invariants and their Applications

Max Ulrich Langbein

31.7.2014

vom
Fachbereich Informatik
der
Technischen Universität Kaiserslautern
zur Erlangung des akademischen Grades
Doktor der Ingenieurwissenschaften (Dr.-Ing.)
genehmigte Dissertation

D 386

# Summary

This PhD-Thesis deals with the calculation and application of a new class of invariants, that can be used to recognize patterns in tensor fields (i.e. scalar fields, vector fields und matrix fields), and by the composition of scalar fields with delta-functions also to point-clouds. In the first chapter an overview over already existing invariants is given.

In the second chapter the general definition of the new invariants is given: starting with a tensor field a set of moment tensor is created via folding in tensor-product manner with different orders of the tensor product of the positional vector. From these, rotational invariant values are calculated via contraction of tensor products. An algorithm to get a complete and independent set of invariants from a given moment tensor set is described. Furthermore methods to make these sets of invariants invariant against translation, rotation, scaling, and affine transformation.

In the third chapter, a method to optimize the calculation of these sets of invariants is described: every invariant can be modeled as undirected graph comprising multiple sub-graphs representing partially contracted tensor products of the moment tensors. The composition of the sets of invariants is optimized by a clever choice of the decomposition into sub-graphs, all paths creating a hyper-graph of sub-graphs where each node describes a composition step. Finally, C++-source-code is created, which optimized using the symmetry of the different tensors and tensor-products, and a comparison of the effort to other calculation methods of invariants is given.

The fourth chapter describes the application of the invariants to object recognition in point-clouds from 3D-scans. To do this, the invariants of subsets of point-clouds are stored for every known object. Afterwards, invariants are calculated from an unknown point-cloud and tried to find them in the database to assign it to one of the known objects. Benchmarks using three 3D-object databases are made testing time and recognition rate.

## Acknowledgments

I like to thank my supervisor Prof. Dr. Hans Hagen for the opportunity to work in many interesting projects, meeting many different people from all parts of the world, while at the same time giving the time for and the freedom of research. I like to thank our secretary Elisabeth “Mady” Gruys for creating a nice working environment and helping whenever one is in trouble. I like to thank Dr. Inga Scheler who always encouraged me to continue. I like to thank my colleagues for many fruitful discussions and for their support and their time, especially Daniel Engel and Raghed al Tarawneh. And of course I like to thank my parents who taught me many skills which were helpful in my studies.

# Contents

# Introduction

Invariants play an important role in physics and are also import in recognition of objects and other structures in point clouds and vector fields. In the word re-cognition lies another key to understand why invariants are important: you re-cognize things that have not changed in an environment that has changed. Usually, recognition of an object is done in another space, time and rotation, so the important invariants are those which do not change under translation, rotation and time shift. Another thing that is important is their computability. You do not want to do an extensive search for things that are invariant, but want to compute them directly in every position where you want to recognize things. Because one does not want to look at every point in space, taking moments of the vectorfield or pointclouds does some work to keep everything in a form that is compact and easy to process. Rotational invariants of moment tensors are total contractions of the tensors. To get more invariants, also total contractions of tensor products are looked at. To avoid having too many invariants, a set of invariants is taken that is independent, i.e. the elements of the set cannot be computed from each other. Why do you need object recognition ? Important fields of application are :

- Classification of objects in LIDAR scans of terrain surfaces. This is important because in many cases, only the terrain surface without buildings and plants is needed; in other cases, you want to extract street networks.
- Face recognition: in security areas, 3d face recognition could serve as a method to track intruders
- Tracking of cells and fibers in biological settings, with scalarfield data stemming e.g. from confocal laser microscopy , MRT or CT.

I only used the invariants for face recognition. The main contributions of this work are:

- A method to create an independent and complete set of invariants for tensor-valued moments of arbitrary dimension (chapter 2 ) by trying out the total contractions of tensor products of increasing order and taking only those with linear independent first derivatives until the expected number of invariants is reached. A moment tensor set e.g. consisting of the second, third and fourth order symmetric 3d tensor with in total 31 independent components is then transformed into 28 rotational invariants.

- A method to optimize the computation of those invariants using a series of tensor products and contractions described in a hypergraph (chapter 3).
- An algorithm using these invariants for object recognition in point clouds (chapter 4), including a method to transform the invariants to get nice statistical properties (section 4.6).

# Chapter 1

# Basic concepts

## 1.1 Tensor Algebra

This section summarizes the basics of tensor algebra and defines notation used in this document. A good reference is [5], chapter 3:Tensor Algebra. A short reference is given in [2], section 2.4:Multilineare Algebra. subsubsection 2.4.2.2:Kovariante und kontravariante Tensoren. In the following, the italicised part of a section is always a translated excerpt from [2] and the other part a comment.

### 1.1.1 Definition of a tensor

A tensor is a multi-linear mapping, which is a mapping that is linear in each of it's arguments, where the arguments can be vectors: *Be $X$ a finite-dimensional linear space over $\mathbb{K}$* (where $\mathbb{K} \in \{\mathbb{R}, \mathbb{C}\}$).*The set of tensors $\mathfrak{T}^p_q(X)$ consists of the mappings*

$$M : X \times \ldots \times X \times X^T \ldots \times X^T \to \mathbb{K} \qquad ,$$

*where the space $X$ is there $q$ times and the dual space $X^T$ is there $p$ times. The Elements of $\mathfrak{T}^p_k$ are called $k$ times covariant and $p$ times contravariant Tensors over $X$, and $\mathfrak{T}^0_0 = \mathbb{K}$.*

### 1.1.2 Tensor product

If $M \in \mathfrak{T}^p_q(X)$ and $N \in \mathfrak{T}^r_s(X)$ the tensor product is defined as (here using the property of the tensor of being a multilinear mapping):

$$(M \otimes N)(u_1, \ldots, u_{q+s}, v_1, \ldots, v_{p+r}) := M(u_1, \ldots, u_q, v_1 \ldots v_p)N(u_{q+1}, \ldots, u_{q+s}, v_{p+1}, \ldots, v_{p+r})$$

### 1.1.3 Basis representation

*Be $b_1, \ldots, b_n$ a basis of $X$. Every Tensor $M \in \mathfrak{T}_p^q(X)$ with $p + q \geq 1$ has a unique representation*

$$M = t_{i_1 \ldots i_q}^{j_1 \ldots j_p} b^{i_1} \otimes \ldots \otimes b^{i_q} \otimes b_{j_1} \otimes \ldots \otimes b_{j_p} \tag{1.1}$$

*with the coefficients $t_{\ldots}^{\ldots} \in \mathbb{K}$. Here, $b_j$ refers to the linear form $b_j(u^*) = u^*(b_j)$ for all $u^* \in X^T$.* In the following, $M_{i_1 \ldots i_q}^{j_1 \ldots j_p}$ will denote the coefficient $t_{i_1 \ldots i_q}^{j_1 \ldots j_p}$ of the basis representation of $M$. The tensor product written in the coefficients is then simply:

$$(M \otimes N)_{i_1, \ldots, i_{q+s}}^{j_1, \ldots, j_{p+r}} = M_{i_1, \ldots, i_q}^{j_1 \ldots j_p} N_{i_{q+1}, \ldots, i_{q+s}}^{j_{p+1}, \ldots, j_{p+s}}$$

### 1.1.4 Einsteins sum convention

Einstein's sum convention is a notation for summing up indices together: upper indices and lower indices that are denoted with the same name are summed over. $a_l^m = t_{il}^{im}$ for example means $a_l^m = \sum_i t_{il}^{im}$ , $b = t_{il}^{il}$ means $b = \sum_{i,l} t_{il}^{il}$ , and $d_l = a_l^m c_m$ means $d_l = \sum_m a_l^m c_m$ . It will be used below.

### 1.1.5 Basis change

*The coefficients $t_{i_1 \ldots i_q}^{j_1 \ldots j_p}$ of a tensor are transformed in the same way as the basis tensor $b^{i_1} \otimes \ldots \otimes b^{i_q} \otimes_{j_1}^{b} \otimes \ldots \otimes b_{j_p}$ in a basis transform. Example: be $M = t_r^s b^r \otimes b_s$. Then $M = t_{r'}^{s'} b^{r'} \otimes b_{s'}$ with $t_{r'}^{s'} = A_{r'}^{r} A_s^{s'} t_r^s$.* In this document, the possible basis changes will be between orthonormal bases, i.e. consist of rotation and mirroring.

### 1.1.6 Lifting/lowering of indices

To convert contra-variant indices into covariant one, one uses the property that $< a_i, b^j >= \delta_i^j$, where $a^i$ is the co-variant and $b_i$ the contra-variant basis i.Here $< \cdot, \cdot >$ denotes the chosen scalar product. Popular choices for the scalar product are the dot product in $\mathbb{R}^3$ implying an euclidean metric, and $a_1 b_1 + a_2 b_2 + a_3 b_3 \; - \; a_4 b_4$ for spacetime implying a minkowski metric.

One sees that in Matrix notation $a^T b = I \Rightarrow a^T = b^{-1}$ if $b^i$,$a_j$ are column vectors and the scalar product is the usual dot product. Therefore, in order to transform from from $b$ to $a$ one has to multiply by $ab^{-1}$= $aa^T$. For an orthonormal basis $aa^T = I$, so upper and lower indices mean the same and raising or lowering of indices does not require computation.

### 1.1.7 Contraction

*From $M$ in equation 1.1 a new tensor can be constructed by equating one upper and lower index and removing the corresponding basis vectors. This operation is independent of the chosen basis. Example: be $M = t^i_{jk} b_i \otimes b^j \otimes b^k$. Then $N = t^i_{ik} b^k$ is again a tensor in the same basis with the coefficients $s_k = t^i_{ik}$.* In this document, we will refer to this operation on the coefficients by trace-taking as this is what it does to a two-component tensor, i.e. a Matrix. As a computer doesn't know about the names of the indices, we use a contraction operator similar to [5], section 3.13, but do not distinguish between co- and contravariant indices:

$$\sum_{(a,b)} M := \sum_{i_a = i_b = k = 1}^{n} M_{i_1, \dots i_n}$$

so $\sum_{(a,b)}$ means "contract the $a$th and $b$th index" . If the indices are both lower or both upper, one has to be lifted or lowered repectively. It does not matter which of them, as the matrix used for lowering or raising is symmetric, as proved above. Example:

$$\sum_{(1,3)} x_{ijk} = x^i{}_{ji} = \sum_{i,j} A_{ik} x_{ijk} = x_{ij}{}^i = \sum_{i,j} A_{ik} x_{ijk} \qquad ,$$

where $A_{ij} =< a_i, a_j >$ is the matrix used for lifting.

## 1.2 Existing Invariants of Surfaces and Fields

In the following, we give a brief and non-exhaustive listing of invariant definitions:

### 1.2.1 Convolution

In [15] the use of integral invariants in general is studied, (of which our moments are a special case). A stability analysis is given, and it turns out that they are very stable against random noise. A good algorithm for computing them for surfaces in 3d using a mixture of fast fourier transformation and octrees is presented. We use a similar algorithm that is more straight-forward in section 4.7.

In [16] the following method for doing pattern recognition in vectorfields on a 2-dimensional domain is described. It works best for structured rectilinear grids. For every possible patternsize $r$ and position $x_0$, calculate the integrals

$$c_{ij} = \int_{B(r,x_0)} (x-x_0)^i \overline{x-x_0}^j f(x)dx$$

with $x, c \in \mathbb{C}$, $f : \mathbb{C} \to \mathbb{C}$, $0 \leq i+j \leq 2$ and $B(r, x_0)$ a ball with radius $r$ around $x_0$.

From $c_{ij}$ calculate the above mentioned scale/rotational invariants: For vector fields, all products $\prod_k c_{i_k j_k}$ with $\sum_{k=1}^n (j_k - i_k) = n$ are rotationally invariant. for scalar fields (i.e. the image space of $f$ is real), the products with $\sum_{k=1}^n (j_k - i_k) = 0$ are rotationally invariant. The scale invariance is straightforwardly computed by parameter substitution of the radius in the integration. All invariants are stored in a tree which is used for similarity searching. The integrals mentioned are accelerated using the convolution theorem and fast fourier transformation. The complexity of the method is $O(nN^2 logN)$ in time for initialization (n times 2-dimensional FFT) and $O(logN)$ for searching. It uses $O(N^2)$ memory. Here, $N$ refers to the number of grid steps per direction and $n$ to the number of ball radii.

### 1.2.2 Polynomial approximation of surfaces and fields

In [7], a multivariate polynomial is fit to the data and invariants of this polynomial are calculated. It is related to our method because a multivariate polynomial can always be expressed by a set of symmetric tensors. Essentially, they solve a linear equation for the coefficients of a polynomial in the source polynomial coefficients. It is more general in terms of the types of transformations possible, but it will fail if the order of the moments in the tensor components becomes too high. Another problem is that the fitting process is not a very stable operation, as noted by the author.

### 1.2.3 Tensor properties

rotational invariants also include the eigenvalues of second-order tensors (i.e. matrices) . These are of course less robust in computation than traces but can be computed analytically from the characteristic polynomial of a matrix $A$ ( $p(\lambda) = \det(A - \lambda I)$ ). To overcome the robustness issues, the coefficients of that polynomial can be used instead.

In [17], a higher-order structure tensor is defined. An intuitive visualization metaphor for higher-order tensors is given, as well as an eigenvector decomposition algorithm for up to fourth order tensors:

$$T = \sum_i \lambda_i e_i \otimes e_i \otimes e_i \otimes e_i$$

These eigenvalues are rotational invariant tensor properties.

In [13] they define a generalized trace $gentr(f(\mathbf{u})) = \frac{3}{2\pi}\int_\Omega f(\mathbf{u})d\mathbf{u}$ , where $\Omega$ is the unit sphere and give formulas for the computation if $f$ is represented by up to sixth order tensors.

### 1.2.4 Spherical harmonic's properties

Spherical harmonics are a decomposition of the sphere which decompose a 3d field into components depending on latitude and longitude, form an orthonormal basis with respect to the scalar product

$$< a, b >:= \frac{1}{4\pi}\int_0^\pi \int_0^{2\pi} a(\theta,\varphi)b(\theta,\varphi)d\varphi \cos\theta d\theta$$

The basis vectors $Y_l^m(\theta,\varphi)$ , $m = 0, 1, ...\infty$, $l = -m, ..., m$ of spherical harmonics are defined as

$$Y_l^m(\theta,\varphi) = N_l^m P_l^m(\cos\theta)e^{im\varphi} \qquad ,$$

where $P_l^m(x)$ refers to the associated Legendre-polynomials and $N_l^m$ to a normalization constant. The advantages of spherical harmonics are that you can control the level of detail just by restricting $m$, that you can esaily perform a rotation on the coefficients of a decomposition after these basis vectors, and that they can be computed again on local masks using convolution and fast fourier transformation. In [4],[3] they are used to have a rotational invariant similarity measure for a local patch (again the integration goes over a ball-shaped mask). To compare two patches, just try out a set of possible rotations on the spherical harmonic's coefficients and take the one that gives the largest similarity. A problem of the method is the time required to try out the possible rotations for similarity matching.

### 1.2.5 Curvature

For surfaces, nice invariants can be computed from the curvature: the gaussian curvature and the curvatures of the principal curvature directions (which again are computable from a 2d curvature tensor, which can be obtained by fitting a quadratic form to a local pointset, or, if a parametric representation of the surface is given, just by taking the derivatives ). [10] give on overview over Methods to compute the curvature. A series of papers has been published using the curvature Tensor for recognition and reconstruction using Tensor voting: [6],[18],[19],[11],[20]

# Chapter 2

# Moment Invariants on Tensor Fields of Arbitrary Order and Dimension

## 2.1 Preface

Objects in space usually have an arbitrary rotation and position which are the transformations which do not not deform the objects. If you want to recognize patterns which have an arbitrary rotation and scale, one possibility would be to try out each possible rotation and scale of the pattern to be recognized but, this is computationally very expensive if you want to be precise. A better way to do it is to first compute a "fingerprint" of the pattern to find which is independent of the rotation and scale, then compare it to the fingerprints of the patterns you want to analyse.

We will give a type of fingerprints that can be computed from tensorfields of arbitrary order and dimension. We will call them moment invariants of the field because of their relationship to the moment of inertia in physics: If we have a mass density field, from the tensor ${}^2\!A$ we define in section 2.2 the tensor of inertia can be computed : $T = ({}^2\!A)I - {}^2\!A$, and $\frac{1}{2}\,{}^2\!A$ is the moment of inertia in all axes if the moment is isotropic, and is a rotational moment invariant. In this chapter, we introduce a method to compute a complete basis for rotational invariants of moments of tensorfields of – in principle – any order and dimension, that are analytical. We give an example using up to 4th-order structure tensors in 3d. We have published the main contents of this chapter in [8]. We have changed the naming a bit from that paper to have more consistency to existing concepts: We talk about moments when referring to the tensors computed from the tensorfield, and about moment invariants instead of invariant moments when talking about the traces that are rotational invariants.

### 2.1.1 Possibilities for Invariance

The moment invariants $\vec{M}$ can have certain invariance properties, which are

- Rotational invariance

  only in value: $\vec{M}(f(\vec{x})) = \vec{M}(Rf(\vec{x})))$

  only in domain: $\vec{M}(f(\vec{x})) = \vec{M}(f(R^{-1}\vec{x})))$

  in domain and value,

  linked: $\vec{M}(f(\vec{x})) = \vec{M}(Rf(R^{-1}\vec{x}))$, see figure 2.1

  independent: $\vec{M}(f(\vec{x})) = \vec{M}(Rf(R_*^{-1}\vec{x}))$

- Translation invariance $\vec{M}(f(\vec{x})) = \vec{M}(f(\vec{x}-\vec{t})), \vec{t} \in \mathbb{R}^d$

- Scale invariance

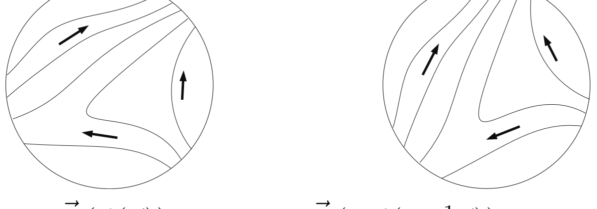
  - In value $(\vec{M}(f(\vec{x})) = \vec{M}(sf(\vec{x})))$ : see figure 2.2

  - In domain ( $M(f(\vec{x})) = \vec{M}(sf(\vec{x})))$ : see figure 2.3

  - In domain and value:

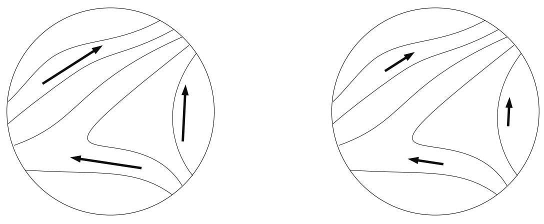
    - Linked : $\vec{M}(f(\vec{x})) = \vec{M}(s^p f(\vec{x}/s))$ , $s \in \mathbb{R}$

    - Independent : $\vec{M}(f(\vec{x})) = \vec{M}(tf(\vec{x}/s))$ , $s, t \in \mathbb{R}$



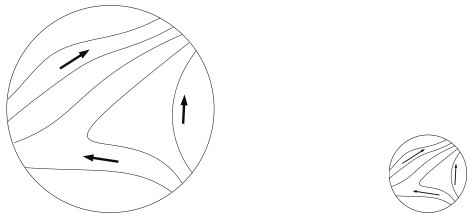
$$\vec{M}(f(\vec{x})) \quad = \quad \vec{M}(Rf(R^{-1}\vec{x}))\ , R \in SO(d)$$

Figure 2.1: Rotational invariance



$$\vec{M}(f(\vec{x})) \quad = \quad \vec{M}(sf(\vec{x}/s)) \ , s \in \mathbb{R}$$

Figure 2.2: Scale invariance in value



$$\vec{M}(f(\vec{x})) \quad = \quad \vec{M}(f(\vec{x}/s)) \ , s \in \mathbb{R}$$

Figure 2.3: Scale invariance in the domain

## 2.2 Rotational Invariance: $Rf(R^{-1}\vec{x}) = f(\vec{x})$

In the following, we always assume that the vectorfield $\vec{f}$ has a limited support.

Rotational invariants $M$ of a vector/tensor-field fulfill:

$$\vec{M}(Rf(R^{-1}\vec{x})) = \vec{M}(f(\vec{x}))$$

To compute them from the field $f(\vec{x})$, we first construct tensors ${}^{m}\!A$

$${}^{m}\!A(f) = \int_{\mathbb{R}^d} \underbrace{\vec{x} \otimes \ldots \otimes \vec{x}}_{m} \otimes f(\vec{x}) dV$$

which explicitly reads:

$${}^{m}\!A_{i_1 \ldots i_m j_1 \ldots j_n} = \int_{\mathbb{R}^d} x_{i_1} \cdots x_{i_m} f_{j_1 \ldots j_n}(\vec{x}) dx_1 \cdots dx_d$$

The tensor rotation $R(T)$ is defined by:

$$T'_{j_1 \ldots j_n} = \sum_{i_1 \ldots i_n} (\prod_{k=1}^{n} R_{i_k j_k}) T_{i_1 \ldots i_n}$$

where $R_{ij}$ is the rotation matrix.

If we compute the tensor ${}^{m}\!A$ for the rotated field $f$, we observe that the rotation of the field $f$ is equivalent to the rotation of the tensor ${}^{m}\!A$:

$$\begin{aligned}
& {}^{m}\!A(R(f(R^{-1}\vec{x}))) \\
= \quad & \int_{\mathbb{R}^d} \underbrace{\vec{x}\otimes\ldots\otimes\vec{x}}_{m} \otimes R(f(R^{-1}\vec{x}))dx_1 ... dx_d \\
=^{x=Ry} \quad & \int_{\mathbb{R}^d} \underbrace{R\vec{y}\otimes\ldots\otimes R\vec{y}}_{m} \otimes R(f(\vec{y}))dy_1 ... dy_d \\
= \quad & \int_{\mathbb{R}^d} \sum_{\alpha_1...\alpha_m} (\prod_{k=1}^{m} R_{\alpha_k i_k} y_{\alpha_k}) \sum_{\beta_1...\beta_n} (\prod_{l=1}^{n} R_{\beta_l j_l}) f_{\beta_1...\beta_n}(\vec{y}) dy_1 \cdots dy_d \\
= \quad & \sum_{\alpha_1...\alpha_m \beta_1...\beta_n} (\prod_{k=1}^{m} R_{\alpha_k i_k})(\prod_{l=1}^{n} R_{\beta_l j_l}) \int_{\mathbb{R}^d} (\prod_{k=1}^{m} y_{\alpha_k}) f_{\beta_1...\beta_n}(\vec{y}) dy_1 ... dy_d \\
= \quad & R({}^{m}\!A(f))
\end{aligned}$$

In the following, we use the convention:

$$\begin{aligned}
\sum_{(a,b)} T_{j_1...j_n} &:= \sum_{i} (T_{j_1...j_n}, j_a = j_b = i) \\
\sum_{(a,b)(c,d)} T_{j_1...j_n} &:= \sum_{ij} (T_{j_1...j_n}, j_a = j_b = i, j_c = j_d = j) \\
& \quad etc.
\end{aligned}$$

We will call those sums traces, and total traces if the result is of 0th order.

Examples:

$$\sum_{(1,3)} T_{j_1 j_2 j_3 j_4} = \sum_{i} T_{i j_2 i j_4} \ , \quad \sum_{(1,3)(2,4)} T_{j_1 j_2 j_3 j_4} = \sum_{ik} T_{ikik} \ .$$

One observes that

$$\sum_{(a,b)} R(T) = R(\sum_{(a,b)} T)$$

( proof see Appendix 2.6 ).

One sees that all total traces are rotation-invariant.

### 2.2.1 Other types of rotational invariance

- rotational-invariant independent in value and domain: you leave out total traces that have pairs between value (the $i$'s) and domain (the $j$'s) indices of the tensor ${}^{m}\!A$.

- rotation-invariant only in value: only take traces between the value indices
- rotation-invariant only in domain: only take traces between the domain indices

### 2.2.2 Construct a basis for the rotational invariants of a tensorset

If one has a set of tensors $A$ with a certain number $L$ of independent components taking their symmetry into account, one wants to find a set of functions whose values are invariant under a common rotation of the tensors in the tensorset. Let $M(d)$ be the number free parameters of a rotation in dimension $d$ ($M(3) = 3, M(2) = 1$ ) Then there will be $L - M(d)$ independent rotationally invariant functions (if you have tensors with order higher than 1 in the set). It turned out that the total traces can be used for these functions. If you want to have a complete basis for the invariants of one tensor $T$, you first take all traces of $T$, then those of $T \otimes T$ , then of $T \otimes T \otimes T$ and so on until you have enough independent invariants.

The number of possibilities for total traces ($\sum_{(a_1,a_2)\ldots(a_{n-1},a_n)} T_{j_1\cdots j_n}$ ) of a tensor with order $n$ are:

$$(n-1) \cdot (n-3) \cdot \ldots \cdot 3 = \frac{n!}{2^{n/2}(n/2)!}$$

The left expression stems from the following: pair the first index with one of the $n-1$ remaining ones, then take the first remaining unpaired index and pair it with one of the now $n-3$ remaining ones, and so on.

| Number of indices | 2 | 4 | 6 | 8 | 10 | 12 |
|---|---|---|---|---|---|---|
| Pairing possibilities | 1 | 3 | 15 | 105 | 945 | 11340 |

You see that there are more summation possibilities than there are independent invariants so one has to eliminate the redundant ones.

**Reducing the number of candidates for invariants in the basis:** One observes that total traces of tensors which are symmetric in certain groups of indices and that have the same number of connections between these groups are equal. Examples for the number of connections: Let $T_{ijk}$ be a total symmetric tensor. then $T \otimes T$ has two groups of indices in which it is total symmetric (the first three and the second three indices). Then the number of connections for $\sum_{(1,2)(3,4)(5,6)} T \otimes T$ is one ( the pair(3,4) ), for $\sum_{(1,3)(2,4)(3,6)} T \otimes T$ it is three (all pairs).

You can exploit this and take only one total trace for every number of connections between groups of total symmetric indices in the tensor.

If you have tensors that are tensorproducts, you so do not take total traces which leave two tensors in the tensorproduct unconnected, because these are computable from traces of lower tensorproduct orders. Example: Let $T_{ij}$ be a total symmetric tensor. Then $\sum_{(1,2)(3,4)} T \otimes T = (\sum_{(1,2)} T)^2$.

**Getting a set of independent invariants:** If the invariants are dependent, then one invariant $M_1$ is a function of the other invariants: $M_1(\vec{a}) = f(M_2(\vec{a}), ...M_n(\vec{a})))$, where $\vec{a}$ is the vector with the independent components of the tensorset $A$. You see then that

$$\frac{\partial M_1(\vec{a})}{\partial \vec{a}} = \frac{df(M_2(\vec{a}), ..., M_n(\vec{a}))}{d\vec{a}} = \sum_{i=2}^{n} \frac{\partial f}{\partial M_i} \frac{\partial M_i}{\partial \vec{a}}$$

So if the invariants are dependent, their derivatives are linear dependent everywhere. To test the linear dependence of the derivatives, it is sufficient to test the dependence of their values at a random test vector. It is possible then that independent invariants will be recognized as dependent, but this is not very harmful because we have enough candidates. To get a complete set of independent invariants, the following algorithm is applied:

1. Create a random test tensor at which the derivatives will be evaluated.

2. Create an upper-triangular matrix with as many columns as independent tensor components and initially zero rows

3. For every candidate (a total trace of a Tensorproduct of the Tensors in the tensorset) do:

   Evaluate the first derivative of it at the test tensor

   Append the derivative as the lowest row to the matrix and „zero out“ by adding multiples of the upper rows (Gaussian elimination)

   If that row is not zero now, the moment is linear independent of the ones already in the basis and we add it to the basis, otherwise the row is removed.

   If the basis is complete(enough entries): stop.

**Remark 1** The calculations have to be done in exact arithmetics (i.e. using fractions of arbitrary precision numbers )

**3D Example: Vectorfield folded with the kernels $1, \vec{x}, \vec{x}\otimes\vec{x}$**

$$ {}^{2}\!A = \int_{\mathbb{R}^3} \vec{x}\otimes\vec{x}\otimes\vec{f}(\vec{x})dV, \quad {}^{1}\!A = \int_{\mathbb{R}^3} \vec{x}\otimes\vec{f}(\vec{x})dV, \quad {}^{0}\!A = \int_{\mathbb{R}^3} \vec{f}(\vec{x})dV $$

Number of Components in A : $6\cdot 3 + 9 + 3 = 30$
Number of independent invariants: $30 - 3 = 27$
The invariants are calculated as total traces from

$$ {}^{0}\!A\otimes{}^{0}\!A, \quad {}^{1}\!A, \quad {}^{1}\!A\otimes{}^{1}\!A, \quad {}^{1}\!A\otimes{}^{1}\!A\otimes{}^{1}\!A, \quad {}^{0}\!A\otimes{}^{2}\!A, \quad {}^{2}\!A\otimes{}^{2}\!A. $$

First, we try to get the maximum number of invariants for every tensor independently. So if a tensor is zero, one still has the contributions of the remaining nonzero tensors.

By the way, the total traces of ${}^{1}\!A$ are correlated to the coefficients of the characteristic polynomial of ${}^{1}\!A$.

### 2.2.3 3D example: Scalar field folded with up to order 4 kernel

$$ {}^{4}\!A = \textstyle\int_{\mathbb{R}^3} \vec{x}\otimes\vec{x}\otimes\vec{x}\otimes\vec{x}\otimes f(\vec{x})dV, \quad {}^{3}\!A = \int_{\mathbb{R}^3} \vec{x}\otimes\vec{x}\otimes\vec{x}\otimes f(\vec{x})dV, $$
$$ {}^{2}\!A = \textstyle\int_{\mathbb{R}^3} \vec{x}\otimes\vec{x}\otimes f(\vec{x})dV, \quad {}^{1}\!A = \int_{\mathbb{R}^3} \vec{x}\otimes f(\vec{x})dV, \quad {}^{0}\!A = \int_{\mathbb{R}^3} f(\vec{x})dV $$

The number of $A$-components is: $1+3+6+10+15 = 35$ . After translating the tensor set $A$ to its gravity center (see section 2.4), ${}^{1}\!A$ get zero and can be ignored. We also dont worry about ${}^{0}\!A$, because it is already rotation invariant. The remaining number of independent invariants is then: $35-4 = 31$. The entries of the moment basis are then calculated as total traces of

$$ \begin{array}{r} {}^{2}\!A, \quad {}^{2}\!A\otimes{}^{2}\!A, \quad {}^{2}\!A\otimes{}^{2}\!A\otimes{}^{2}\!A, \\ {}^{3}\!A\otimes{}^{3}\!A, \ \dots\ , \underbrace{{}^{3}\!A\otimes\cdots\otimes{}^{3}\!A}_{6} \\ {}^{4}\!A, \ \dots\ , \underbrace{{}^{4}\!A\otimes\cdots\otimes{}^{4}\!A}_{4} \\ {}^{2}\!A\otimes{}^{4}\!A, \quad {}^{2}\!A\otimes{}^{3}\!A\otimes{}^{3}\!A, \quad {}^{3}\!A\otimes{}^{3}\!A\otimes{}^{4}\!A \end{array} $$

(see appendix 2.7).

These invariants are also invariant against mirroring of the field. If you want to distinguish between mirrored versions of the field $f$ and original ones, you have to include

$$ \sum_{(1,10),(2,13),(3,16),(4,11),(5,14),(6,17),(7,12),(8,15),(9,18)} {}^{3}\!A\otimes{}^{3}\!A\otimes{}^{3}\!A\otimes\varepsilon\otimes\varepsilon\otimes\varepsilon $$

into the basis where $\varepsilon$ is the total antisymmetric tensor.

| $N$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| $L(N,3)$ | 1 | 4 | 10 | 20 | 35 | 56 | 84 | 120 | 165 |
| # invariants of: | | | | | | | | | |
| scalarfields | 1 | 2 | 7 | 17 | 32 | 53 | 81 | 117 | 162 |
| vectorfields | 1 | 4 | 27 | 57 | 102 | 165 | 249 | 357 | 492 |
| stressfields | 3 | 21 | 57 | 117 | 207 | 333 | 501 | 717 | 987 |

Table 2.1: Number of independent rotational invariants for tensor sets of dimension 3 and maximum order $N$

### 2.2.4 Number of independent invariants for 3D-tensor fields

We first look at the scalar-field case, where the tensors ${}^n\!A$ are total symmetric. The number of components of a total symmetric tensor is equivalent to the number of different words (= index sets) over an alphabet with d characters (=the different index values) without the permutated versions of the words. This number is $\binom{n+d-1}{d-1}$ ( see [2], Word Problem ). The total number of independent components $L$ of a tensorset $A$ with maximum order $N$ is:

$$L(N,d) = \sum_{n=0}^{N} \binom{n+d-1}{d-1} = \binom{N+d}{d}$$

Reasoning for that: The independent components in the whole tensorset of all tensors ${}^0\!A, ..., {}^N\!A$ in dimension $d$ can be expressed as the components of an $N$th-order $(d+1)$-dimensional tensor $T$ where the components of ${}^i\!A$ are the components of $T$ where $N-i$ indices are $(d+1)$ and $i$ indices are in the range $[1, ..., d]$ . The number of components of that new tensor is then accordingly $\binom{N+d}{d}$. For non-scalar tensor fields $f(\vec{x})$, you have to multiply $L$ with the number of independent $f$ components (e.g. 3 for a vectorfield, 6 for a stressfield). If you want the maximum number of independent rotational invariants computed from A, you have to subtract the dimension of the rotation group $SO(3)$ from the number of components of $A$ which is three (or two if the maximum tensor order is one ). If having also translational and value invariance, of course the number is reduced by another four, so for $d = 3, N = 4$, the number of scalar field invariants is 28.

See table 2.1 for the numbers.

## 2.3 Affine Invariance

To achieve affine invariance for scalar fields, a possibility is to apply a basis transform to the tensor set that converts ${}^2\!A$ to a unit matrix. To achieve that, we make an eigenvector decomposition of ${}^2\!A$:

$$ {}^2\!A = VDV^T, D = \begin{pmatrix} D_{11} & & 0 \\ & \ddots & 0 \\ 0 & & D_{nn} \end{pmatrix} $$

and $D_{ii}$ being the $i$-th eigenvalue. $D$ can be expressed as a product $EE^T$, with $E$ being a diagonal matrix with $E_{ii} = \sqrt{D_{ii}}$. It holds :

$$ {}^2\!A = VEIE^TV^T $$

with $I$ being the unit matrix.

If we set $B = VE$ we have ${}^2\!A = BIB^T$. One sees that basis-transformation matrix from a tensor $I$ to the tensor ${}^2\!A$ is $B$. To transform from ${}^2\!A$ to $I$, one has to transform with $B^{-1} = E^{-1}V^{-1}$ with $V^{-1} = V^T$, $E^{-1}$ being diagonal and $E_{ii}^{-1} = 1/\sqrt{D_{ii}}$.

## 2.4 Translational Invariance: $\vec{M}(f(\vec{x})) = \vec{M}(f(\vec{x}-\vec{t}))$

### 2.4.1 General Principle

For translation invariance, one always computes a vector $\vec{c}$ from the field which moves in the same way the field $f(\vec{x}-\vec{t})$ moves (so $\vec{c}+\vec{t}$ is constant). Then one computes the invariants $M$ in a coordinate system with origin in $-\vec{c}$. For scalar fields, a simple choice for $-\vec{c}$ is the center of gravity which can be computed as follows from $A$:

$$ \frac{\int_{\mathbb{R}^d} \vec{x} f(\vec{x}) dV}{\int_{\mathbb{R}^d} f(\vec{x}) dV} = \frac{{}^1\!A}{{}^0\!A} $$

The new $A'$ are computed from $A$ and $\vec{c}$ for scalar fields as following:

$$ \begin{aligned} {}^0\!A' &= {}^0\!A, \qquad {}^1\!A' = {}^1\!A + \vec{c}\,{}^0\!A, \\ {}^2\!A' &= {}^2\!A + \vec{c} \otimes {}^1\!A + {}^1\!A \otimes \vec{c} + \vec{c} \otimes \vec{c}\; {}^0\!A \end{aligned} $$

In the general case, where $f(\vec{x})$ can be a tensor field, and one has $\vec{x}$-powers higher than $(\vec{x}\otimes)^2$ it is more complicated: You have to express

$$\int_{\mathbb{R}^d} \underbrace{(\vec{x}+\vec{c})\otimes\cdots\otimes(\vec{x}+\vec{c})\otimes}_{n} f(\vec{x})dV$$

As a combination of $A$ and $\vec{c}$. The tensors ${}^{n}\!A'$ of the translated tensorset $A'$ are then computed as follows:

$$\begin{aligned} {}^{n}\!A' &= \sum_{k=0}^{n} \binom{n}{k} B(k,n) \\ \text{with} & \\ B(k,n) &= C(k,n) \text{ symmetrized in the first } n \text{ indices} \\ C(k,n) &= \underbrace{\vec{c}\otimes\cdots\otimes\vec{c}\otimes}_{k} {}^{n-k}\!A \end{aligned}$$

To be more efficient, we look at it componentwise and go to a certain dimension:

**Translation in 3D:** We use the following notation in 3-dimensional space: $a_{ijk}$ denotes a tensor component of the total symmetric tensor ${}^{i+j+k}\!A$ which which has $i$ indices that are 1, $j$ indices that are 2, and $k$ indices that are 3. The translation of this tensorset is computed by:

$$a'_{IJK} = \sum_{i=0}^{I}\sum_{j=0}^{J}\sum_{k=0}^{K} \binom{I}{i}\binom{J}{j}\binom{K}{k} a_{ijk} d_{I-i,J-j,K-k}$$

where $d_{ijk} := c_1^i c_2^j c_3^k$ denotes the components of a tensor constructed from the translation vector $\vec{c}$. The formula above results from

$$a'_{IJK} = \int_{\mathbb{R}^3} (x_1+c_1)^I (x_2+c_2)^J (x_3+c_3)^K dx_1 dx_2 dx_3$$

If you look at all components to compute, you see that the total number of summands for the previous formula applied for all components is:

$$L_s := \sum_{I=0}^{N}(I+1)\sum_{J=0}^{N-I}(J+1)\sum_{K=0}^{N-I-J}(K+1) \qquad , N = \text{max. order in } A$$

One will then additionally have $L_c := (N+1)(N+2)-4$ multiplications you need for the calculation of $d$ : one can compute every higher order component by the multiplication of a lower-order component with a component of

$\vec{c}$. Effort versus tensor order (effort = number multiplications):

| Maximum Order(N) | 2 | 3 | 4 | 5 | 6 |
|---:|---|---|---|---|---|
| Approximated effort: $2L_s + L_c$ : | 66 | 188 | 455 | 980 | 1932 |
| Measured by implementation of the program: | 27 | 101 | 287 | 689 | 1467 |

### 2.4.2 Vectorfields $\vec{f}$ and alternate choices for $-\vec{c}$

One does not always want to center the coordinate system in the center of gravity. For higher-order fields, a center of gravity is not defined. So you have to look for other choices of $-\vec{c}$.

At the center of gravity in the scalar case the moment $\sum_{(1,2)} {}^{1}\!A' \otimes {}^{1}\!A'$ takes its minimum with respect to $\vec{c}$. So a good choice for $\vec{c}$ in the other cases would be the minimum of a moment. To have a unique minimum, we choose a moment which is quadratic in $\vec{c}$. Then we get then the minimum from a linear equation. For vectorfields, let's look at the invariants

$$m_1' := \sum_{(1,2)(3,4)} {}^{1}\!A' \otimes {}^{1}\!A' \quad \text{and} \quad m_2' := \sum_{(1,4)(2,3)} {}^{1}\!A' \otimes {}^{1}\!A'$$

At the minimum we have $\partial m_i' / \partial \vec{c} = 0$. In the following formulas we use matrix calculus, where ${}^{1}\!A$ is a matrix, ${}^{0}\!A$ a column vector, $I$ the unit matrix.

$\partial m_1' / \partial \vec{c} = 0$ gives

$$\left( -\,{}^{0}\!A\,{}^{0}\!A^T + 2 \begin{pmatrix} ({}^{0}\!A_1)^2 & & 0 \\ & \ddots & \\ 0 & & ({}^{0}\!A_d)^2 \end{pmatrix} \right) \vec{c} = -\left( \sum_{(1,2)} {}^{1}\!A \right) \; {}^{0}\!A$$

$\partial m_2' / \partial \vec{c} = 0$ gives

$$\left( -\,{}^{0}\!A^T\,{}^{0}\!A \;\; I + 2 \begin{pmatrix} ({}^{0}\!A_1)^2 & & 0 \\ & \ddots & \\ 0 & & ({}^{0}\!A_d)^2 \end{pmatrix} \right) \vec{c} = -\,{}^{1}\!A\,{}^{0}\!A$$

The above calculation is not possible if the vector ${}^{0}\!A$ vanishes which often occurs for the interesting vectorfields $\vec{f}(\vec{x})$ All invariants with order 2 in $\vec{c}$ rely on ${}^{0}\!A$. So you use either minima of higher order invariants or extract the translation vector directly from the vectorfield and not $A$. Here, one choice for the translation vector $-\vec{c}$ is the gravity center of the vector-lengths $\frac{\int_{\mathbb{R}^d} \vec{x} ||\vec{f}(\vec{x})|| dV}{\int_{\mathbb{R}^d} ||\vec{f}(\vec{x})|| dV}$.

## 2.5 Scale Invariance

In this section we will describe how to make the rotational invariants described above scale-invariant.

### 2.5.1 Scale Invariance in Value: $\vec{M}(f(\vec{x})) = \vec{M}(sf(\vec{x}))$

For the scale invariance , you have to look in which order the scaling factor $s$ appears in the invariants. This $s$-order is just the number of A-tensors from which $\vec{M}$ is calculated ($\sum_{(1,4)(2,5)(3,6)} {}^3\!A \otimes {}^3\!A$ for example has an $s$-order of 2). We denote it $o_v$ (o̲rder in v̲alue). Let $\vec{M}(k)$ denote the subset of invariants with $o_v = k$. Let $c_k := ||\vec{M}(k)||_\infty = max_i|M(k)_i|$ denote a norm for these invariant sets. You want to find a factor s so that for every set of invariants $c_k s^{-k} \leq 1$ and for one set $c_k s^{-k} = 1$. Then multiply $\vec{M}(k)$ by $s^{-k}$. The resulting invariant set is scale invariant. Let $l$ be the order of the set with $c_l s^{-l} = 1$ so $s = \sqrt[l]{c_l}$. A consequence is $c_k / \sqrt[l]{c_l}^k \leq 1 \Rightarrow \sqrt[k]{c_k} \leq \sqrt[l]{c_l}$. So $s$ will be set to $\max_k \sqrt[k]{c_k}$.

### 2.5.2 Scale Invariance in Domain: $\vec{M}(f(\vec{x})) = \vec{M}(f(\vec{x}/s))$

We define ${}^m\!A'$ as the $A$-tensor computed from the domain-scaled field:

$${}^m\!A' := {}^m\!A(f(\vec{x}/s)) = \int_{\mathbb{R}^d} (\vec{x}\otimes)^m f(\vec{x}/s) dx_1 ... dx_d$$

Substitution of $\vec{x}$ with $\vec{y} = \vec{x}/s$ gives

$${}^m\!A' = \int_{\mathbb{R}^d} (s\vec{y}\otimes)^m f(\vec{y})(sdy_1)...(sdy_d) = s^{m+d} \ {}^m\!A$$

You see that the order the scaling factor in ${}^3\!A$ is the number of $\vec{x}$-powers plus the dimension of the field. We will call that order the *domain order* $o_d$ of the tensor $A$. So $o_d({}^n\!A) = n + d$ if $f$ is defined on $\mathbb{R}^d$. If you want to compute the domain order $o_d$ of the invariants, you refer to the order of the source tensors of the total traces. Example in 3D: $o_d(\sum_{(1,4)(2,5)(3,6)} {}^3\!A \otimes {}^3\!A) = o_d({}^3\!A \otimes {}^3\!A) = 2o_d({}^3\!A) = 12$. If $\vec{M}(l)$ denotes the subset of invariants with $o_d = l$, the moment invariants are $\vec{M}(l)s^{-l}$ with $s = \max_l \sqrt[l]{||\vec{M}(l)||_\infty}$. Proof is in analogy to section 2.5.1.

### 2.5.3 Scale Invariance in Domain and Value: $\vec{M}(f(\vec{x})) = \vec{M}(s^p f(\vec{x}/s))$

Here, the combined order $o_c = o_v + o_d$ is of importance. Example in 3D with $p = 1$: The trace of ${}^3\!A$ has combined order 6+1=7, the trace of ${}^3\!A \otimes {}^3\!A$ has combined order $7 \cdot 2 = 14$. Again, you look for the order by looking at the order of source tensors of the total traces. If $\vec{M}(l)$ now denotes the subset of invariance with $o_c = l$, the invariant moments are $\vec{M}(l)s^{-l}$ with $s = \max_l \sqrt[l]{||\vec{M}(l)||_\infty}$. Proof is in analogy to section 2.5.1.

### 2.5.4 Scale Invariance Independently in Domain and Value: $\vec{M}(f(\vec{x})) = \vec{M}(sf(\vec{x}/t))$

We construct the moment invariants in the following way: Let $c_{kl} := ||\vec{M}(k,l)||_\infty$ be the norms of the invariant's subsets with domain order $o_d = k$ and value order $o_v = l$. You construct the quotient of two norms $q_k := c^{l_2}_{k_1 l_1}/c^{l_1}_{k_2 l_2}$ with the property $k = o_d(q_k) = l_2 k_1 - l_1 k_2 > 0$ and $o_v(q_k) = l_1 l_2 - l_2 l_1 = 0$ (so the value scaling factor is eliminated). Of course,one should choose $c^{l_1}_{k_2 l_2} \neq 0$. Now from $s = \max\{\sqrt[k]{q_k}\}$ one constructs the domain-scale-invariant moments $\vec{M}(k,l)s^{-l}$. Subsequently, you apply the method for scale invariance in value as previously described in section 2.5.1.

**Example: Scalar case with up to order 4 kernel, 3D Point-Cloud setting** For this case, the invariants $\{\sum_{(0,1)} {}^2\!A\}$ with the norm $c_{21}$ and $\{\sum_{(0,1),(2,3)} {}^4\!A\}$ with the norm $c_{41}$ are a good choice for the construction of the quotients. We have then $k_1 = 4, k_2 = 2, l_1 = l_2 = 1$, $k = 4 - 2 = 2$, $q_2 = c_{41}/c_{21}$, $s = \sqrt[2]{q_2} = \sqrt{c_{41}/c_{21}}$. We also have then (as experiments showed) always $t = \sqrt[1]{s^{-2}c_{21}} = c^2_{21}/c_{41}$. The scale-invariants are then $\vec{M}(k,l)t^{-k}s^{-l}$.

## 2.6 Proof of $\sum_{(a,b)} R(T) = R(\sum_{(a,b)} T)$

We show that trace taking is a rotationally invariant operation (Of course this is a known fact from tensor algebra). In terms of components, we have to prove the relation

$$\sum_{k=j_a=j_b=1}^{d} \sum_{i_1,\ldots,i_n=1}^{d} \left(\prod_{l=1}^{n} R_{i_l j_l}\right) T_{i_1,\ldots,i_n} \overset{?}{=} \sum_{\substack{i_1,\ldots,i_n \\ \text{w.o. } i_a,i_b}=1}^{d} \left(\prod_{\substack{l\in\{1,\ldots,n\} \\ \setminus\{a,b\}}} R_{i_l j_l}\right) \sum_{m=i_a=i_b=1}^{d} T_{i_1,\ldots,i_n}$$

We start with the left side and transform it to the right side. (without loosing generality, we set $a = 1$ and $b = 2$).

$$\begin{aligned}
& \sum_{k=j_1=j_2=1}^{d} \sum_{i_1,\ldots,i_n=1}^{d} \prod_{l} R_{i_l j_l} T_{i_1,\ldots,i_n} \\
\text{separate the factors with } l=1,2\text{, replace } j_{1,2} \text{ by } k \quad = & \sum_{k=1}^{d} \sum_{i_1,i_2=1}^{d} \sum_{i_3,\ldots,i_n=1}^{d} R_{i_1 k} R_{i_2 k} \left(\prod_{l=3}^{n} R_{i_l j_l}\right) T_{i_1,\ldots,i_n} \\
\text{reorder it} \quad = & \sum_{i_3,\ldots,i_n=1}^{d} \sum_{i_1,i_2=1}^{d} \underbrace{\sum_{k=1}^{d} R_{i_1 k} R_{i_2 k}}_{\text{use orthonormality of R: } \left\{\begin{matrix} 1:\ i_1=i_2 \\ 0:\ i_1\neq i_2 \end{matrix}\right\}} \left(\prod_{l=3}^{n} R_{i_l j_l}\right) T_{i_1,\ldots,i_n} \\
\text{only summands with } i_1 = i_2 \text{ contribute} \quad = & \sum_{i_3,\ldots,i_n=1}^{d} \sum_{m=i_1=i_2=1}^{d} \left(\prod_{l=3}^{n} R_{i_l j_l}\right) T_{i_1,\ldots,i_n} \\
\text{reorder it} \quad = & \sum_{i_3,\ldots,i_n=1}^{d} \left(\prod_{l=3}^{n} R_{i_l j_l}\right) \sum_{m=i_1=i_2=1}^{d} T_{i_1,\ldots,i_n}
\end{aligned}$$

This proves the rotational invariance.

## 2.7 Invariant moments for scalarfields with up to 4th order in $\vec{x}$

The tables 2.2 and 2.3 each represent a basis for the rotation-invariant moments of a scalarfield in 3D. Every row in these tables corresponds to an invariant moment. On the left side, the tensor is written from which the total traces are taken. On the right, the index pairs are written which are summed over. The index indices start with 0 (are zero-based).

| | |
|---|---|
| ${}^2\mathbb{A}:$ | (0 1) |
| ${}^2\mathbb{A}\otimes{}^2\mathbb{A}:$ | (0 2)(1 3) |
| ${}^2\mathbb{A}\otimes{}^2\mathbb{A}\otimes{}^2\mathbb{A}:$ | (0 2)(1 4)(3 5) |
| ${}^3\mathbb{A}\otimes{}^3\mathbb{A}:$ | (0 3)(1 4)(2 5) |
| | (0 1)(3 4)(2 5) |
| ${}^3\mathbb{A}\otimes{}^3\mathbb{A}\otimes{}^3\mathbb{A}\otimes{}^3\mathbb{A}:$ | (0 6)(1 9)(2 10)(3 7)(4 8)(5 11) |
| | (0 3)(1 6)(2 9)(4 7)(5 10)(8 11) |
| | (9 10)(0 3)(1 6)(2 11)(4 7)(5 8) |
| | (3 4)(6 7)(9 10)(0 5)(1 8)(2 11) |
| ${}^3\mathbb{A}\otimes{}^3\mathbb{A}\otimes{}^3\mathbb{A}\otimes{}^3\mathbb{A}\otimes{}^3\mathbb{A}\otimes{}^3\mathbb{A}:$ | (0 12)(1 15)(2 16)(3 9)(4 13)(5 17)(6 10)(7 11)(8 14) |
| ${}^4\mathbb{A}:$ | (0 1)(2 3) |
| ${}^4\mathbb{A}\otimes{}^4\mathbb{A}:$ | (0 4)(1 5)(2 6)(3 7) |
| | (0 1)(4 5)(2 6)(3 7) |
| ${}^4\mathbb{A}\otimes{}^4\mathbb{A}\otimes{}^4\mathbb{A}:$ | (0 4)(1 5)(2 8)(3 9)(6 10)(7 11) |
| | (8 9)(0 4)(1 5)(2 6)(3 10)(7 11) |
| | (4 5)(8 9)(0 6)(1 7)(2 10)(3 11) |
| | (0 1)(4 5)(8 9)(2 6)(3 10)(7 11) |
| ${}^4\mathbb{A}\otimes{}^4\mathbb{A}\otimes{}^4\mathbb{A}\otimes{}^4\mathbb{A}:$ | (0 8)(1 12)(2 13)(3 14)(4 9)(5 10)(6 11)(7 15) |
| | (0 8)(1 9)(2 12)(3 13)(4 10)(5 11)(6 14)(7 15) |
| | (0 4)(1 8)(2 12)(3 13)(5 9)(6 10)(7 14)(11 15) |
| | (12 13)(0 4)(1 8)(2 14)(3 15)(5 9)(6 10)(7 11) |
| | (12 13)(0 4)(1 8)(2 9)(3 14)(5 10)(6 11)(7 15) |
| ${}^2\mathbb{A}\otimes{}^4\mathbb{A}:$ | (2 3)(0 4)(1 5) |
| ${}^2\mathbb{A}\otimes{}^3\mathbb{A}\otimes{}^3\mathbb{A}:$ | (2 3)(0 5)(1 6)(4 7) |
| | (2 3)(5 6)(0 4)(1 7) |
| | (0 2)(1 5)(3 6)(4 7) |
| ${}^3\mathbb{A}\otimes{}^3\mathbb{A}\otimes{}^4\mathbb{A}:$ | (0 1)(3 4)(6 7)(2 8)(5 9) |
| | (0 1)(2 6)(3 7)(4 8)(5 9) |

Table 2.2: a set of total traces that forms a basis for the rot.inv. moments in the 3d scalar case

| | |
|---|---|
| ${}^2A$ : | (0 1) |
| ${}^2A \otimes {}^2A$ : | (0 2)(1 3) |
| ${}^2A \otimes {}^2A \otimes {}^2A$ : | (0 2)(1 4)(3 5) |
| ${}^3A \otimes {}^3A$ : | (0 3)(1 4)(2 5) |
| | (0 1)(3 4)(2 5) |
| ${}^3A \otimes {}^3A \otimes {}^3A \otimes {}^3A$ : | (0 6)(1 9)(2 10)(3 7)(4 8)(5 11) |
| | (0 3)(1 6)(2 9)(4 7)(5 10)(8 11) |
| | (9 10)(0 3)(1 6)(2 11)(4 7)(5 8) |
| | (3 4)(6 7)(9 10)(0 5)(1 8)(2 11) |
| ${}^4A$ : | (0 1)(2 3) |
| ${}^4A \otimes {}^4A$ : | (0 4)(1 5)(2 6)(3 7) |
| | (0 1)(4 5)(2 6)(3 7) |
| ${}^4A \otimes {}^4A \otimes {}^4A$ : | (0 4)(1 5)(2 8)(3 9)(6 10)(7 11) |
| | (8 9)(0 4)(1 5)(2 6)(3 10)(7 11) |
| | (4 5)(8 9)(0 6)(1 7)(2 10)(3 11) |
| | (0 1)(4 5)(8 9)(2 6)(3 10)(7 11) |
| ${}^2A \otimes {}^4A$ : | (2 3)(0 4)(1 5) |
| ${}^2A \otimes {}^2A \otimes {}^4A$ : | (0 4)(1 5)(2 6)(3 7) |
| | (4 5)(0 2)(1 6)(3 7) |
| ${}^2A \otimes {}^2A \otimes {}^2A \otimes {}^4A$ : | (0 6)(1 7)(2 4)(3 8)(5 9) |
| ${}^2A \otimes {}^4A \otimes {}^4A$ : | (2 3)(0 6)(1 7)(4 8)(5 9) |
| | (2 3)(6 7)(0 4)(1 8)(5 9) |
| | (0 2)(1 6)(3 7)(4 8)(5 9) |
| ${}^2A \otimes {}^3A \otimes {}^3A$ : | (2 3)(0 5)(1 6)(4 7) |
| | (2 3)(5 6)(0 4)(1 7) |
| | (0 2)(1 5)(3 6)(4 7) |
| ${}^2A \otimes {}^2A \otimes {}^3A \otimes {}^3A$ : | (4 5)(0 7)(1 8)(2 6)(3 9) |
| ${}^3A \otimes {}^3A \otimes {}^4A$ : | (0 1)(2 6)(3 7)(4 8)(5 9) |

Table 2.3: a second set of total traces that forms a basis for the rot. inv. moments in the 3d scalar case

The computation of the total traces of tensors with order $\geq 12$ in table 2.2 is very expensive. One can save computing time by including more traces from combined tensors into the basis, which is done in table 2.3. However, this basis will be incomplete once some of the source tensors ${}^m\!A$ get zero. The tensors ${}^m\!A$ with odd $m$ get zero if the field fulfils $f(\vec{x}) = f(-\vec{x})$, the tensors with even $m$ get zero once $f(\vec{x}) = -f(-\vec{x})$. The source tensors ${}^m\!A$ can have zeros even for other reasons.

**Results and Conclusions**

We have presented a method to compute moments of a finite tensorfield invariant to rotation, scale and translation which are computed in the following steps: At first a set of structure tensors is computed, then we it is made invariant to translation, then the basis for rotationally invariant moments of it is computed, and finally the basis is made invariant to scale. We have also given some figures on the efficiency of the different steps. Then we show our method for the case of a 3d scalar field with tensor orders up to 4 in the tensorset. The resulting moments are given in table 2.2. The evaluation of one set of the corresponding polynomials in the ${}^m\!A$-components, optimized by a quasi-Horner-scheme in double precision (in the paper, I wrongly assumed long double, as my compiler interpreted long double as double) needs $2.6\mu s$ on an Intel Xeon 3GHz CPU. Further optimization can be achieved by computing intermediate tensors along a tensor hypergraph (see next chapter).

## 2.8 Comparison to other tensor invariants

Here, a comparison to the invariants mentioned in section 1.2 is given:

Compared to the method of [17] which computes Eigenvalues and eigenvectors of up to fourth order structure tensors, our invariants are more efficient to compute (as you will see in the following chapter 3). We could replace our ${}^m\!A$-tensors described in section 2.2 by these structure tensors, but the disadvantage is that they are not additive for different spacial domains, which is used in our octree summing algorithm for speeding up computations described in Appendix 4.7.1. Another disadvantage is that you can't easily compute versions for a translated origin for them which is important when computing translation-invariant properties of tensorfields.

# Chapter 3

# Efficient Computation of Higher-Order Moment Invariants Using a Graph Representation

This chapter is about methods to compute the moment invariants introduced in the previous chapter more efficiently.

To look at the structure of the invariants in a different way, we give the set of invariants in the so-called Einstein notation (table 3.1): The type of the tensor is indicated here by the number of its indices, and the pairs of the paired sums in the previous chapter are now indicated by an index with same letter, once as upper and once as lower index. Because our tensors live in euclidean space with a unit metric tensor, it doesn't matter which of the two indices is an upper index (co-variant) or a lower index (contra-variant).

$M_0 = A_i A^i$ , $M_1 = A_j A_i{}^i{}_k A_l A_{mn}{}^l A^m A^{knj}$ , $M_2 = A_i A_j A^{ji}$ , $M_3 = A_i{}^i$ , $M_4 = A_{ij} A^{ij}$ , $M_5 = A_{ij} A^{ji}$ , $M_6 = A_{ij} A_k{}^j A^{ki}$ , $M_7 = A_{ij} A^j{}_k A^{ki}$ , $M_8 = A_{ij} A_{kl} A^{lj} A^{ki}$ , $M_9 = A_{kl} A_i{}^{il} A_j{}^{jk}$ , $M_{10} = A_{jk} A_i{}^{ik} A^j{}_l{}^l$ , $M_{11} = A_{jk} A^k{}_l{}^l A_i{}^{ij}$ , $M_{12} = A_i{}^i{}_k A_j{}^{jk}$ , $M_{13} = A_i{}^i{}_j A^j{}_k{}^k$ , $M_{14} = A_{ij}{}^i A^j{}_k{}^k$ , $M_{15} = A_{ijk} A^{ijk}$ , $M_{16} = A_{ijk} A^{ikj}$ , $M_{17} = A_i{}^i{}_l A_j{}^j{}_m A_k{}^k{}_n A^{mnl}$ , $M_{18} = A_i{}^i{}_k A_j{}^j{}_l A_{mn}{}^m A^{lnk}$ , $M_{19} = A_i{}^i{}_k A_j{}^j{}_l A_{mn}{}^l A^{mnk}$ , $M_{20} = A_i{}^i{}_k A_j{}^j{}_l A^l{}_{mn} A^{mnk}$ , $M_{21} = A_i{}^i{}_j A_{klm} A^k{}_n{}^n A^{lmj}$ , $M_{22} = A_i{}^i{}_j A_{klm} A^m{}_n{}^n A^{klj}$ , $M_{23} = A_i{}^i{}_j A_{klm} A^k{}_n{}^m A^{lnj}$ , $M_{24} = A_i{}^i{}_j A_{klm} A^{kl}{}_n A^{mnj}$ , $M_{25} = A_i{}^i{}_j A_{klm} A^{km}{}_n A^{lnj}$ , $M_{26} = A_i{}^i{}_j A_{klm} A^m{}_n{}^k A^{lnj}$

Table 3.1: set of invariants to be computed for a vector field in Einstein notation

This notation shows an important property of the contractions: associativity. To compute the final moments using some arbitrary computation order using intermediate tensors, just put braces into the product of indexed tensors. This "brace-putting" will be the base of the graph-based optimization done in this chapter. I tried out the following possibilities for computation of the moment invariants:

- Polynomials in DNF in the source tensor components
- Horner scheme for the resulting polynomials
- The polynomials optimized for common sub-expressions
- Graph-based optimization

The graph-based optimization performed best, see tables ( 3.2,3.3 ).

## 3.1 Notes on the visualizations used in the following sections

### 3.1.1 The tensor graph

It is common usage as mentioned e.g. by Penrose in [14],section 12.8, to visualize a set of tensors with contracted indices as a graph. To visualize graphs of tensors where the with two sets of symmetric indices stemming from moments of vector- or tensorfields, I use graphs where every basic tensor is visualized by two circles directly attached to each other, each circle sized according to the number of indices.

Each contracted index pair is then indicated by a line between the circle of the corresponding symmetric index set and the corresponding index of the other basic component.

The circle is used to make clear that it makes no difference to which position of the circle , i.e. to which index in the set of symmetric indices an edge representing a contraction is attached to. Uncontracted indices are indicated by open edges attached to only one circle(s). If having only one set of symmetric indices of course only one circle is needed.

### 3.1.2 The tensor hypergraph

To visualize which tensors are contracted together to form a new tensor, I use a (directed) graph where each node is a tensor represented by a tensor graph inside a circle. We will call it a *tensor hypergraph.* The tensor graphs of final results, i.e. tensors with no open indices, will be drawn with a black background. The direction is indicated by the direction on the page, that means that the base tensors (the moments) will be draw on the bottom and the results of a contraction one level above. To indicate that a tensor is computed by a contraction of two instances of the same base tensor, the result tensor is connected by two lines bent a little bit outwards to its base tensor (see figures on pages 34 ff ).

### 3.1.3 Image creation

To create the actual tensor hypergraph images in this document I wrote a small tool in C++ that automatically creates the tensor hypergraph images in postscript from the graph datastructures I use to create the code for the moment Invariant computation.

For the tensor graphs it places all base moments in a circle in the order they are in the graph and then connects them through bent lines going through the middle. Self-contractions are indicated on the outside. A dynamic spring layout would give nicer graph images, but with my method it is made sure that the tensor graphs always have the same size.

For the tensor hypergraph the image is created level by level: In the bottom level, the base tensors(the moments) are drawn, in the level above all tensors that are calculated only from the tensors in the level(s) below and so on.

To prevent image cluttering at least a little bit I ordered the tensors in one level by the number of the leftmost tensor of the tensor in the level below they are connected to.

## 3.2 Cost estimate for invariant computation using contraction of tensor products

To asses the costs for a certain "brace configuration" which we will use to find an optimal one, we need some formulas to describe the cost of a tensor product and contraction in one step. For simplicity I used only formulas for product+contractions of non-symmetric tensors which is an upper bound.

$$\begin{aligned} \text{number of additions: } n_a &= d^f(d^c - 1) \\ \text{number of multiplications: } n_m &= d^f d^c(t-1) \end{aligned}$$

where

- $d$ is the dimension of the tensors
- $f$ is the number of free indices of the result
- $c$ is the number of indices that are contracted in one step
- $t$ is the number of tensors the new tensor is computed from

## 3.3 Optimization by decomposition

Let's take $M_{17} = A_i{}^i{}_l A_j{}^j{}_m A_k{}^k{}_n A^{mnl}$ as an example. If you don't decompose the moment invariant computation at all and don't mind symmetry, you would have

$$f = 0,\ c = 6,\ t = 4 \quad\Longrightarrow\quad n_a = 728,\ n_m = 2916$$

If you calculate the DNF of the polynomial which minds tensor symmetry, you still have $n_a = 299$, $n_m = 1155$.

We use associativity of tensor products and contractions to calculate them in an optimized way.

if you would decompose it like this:

$$\underbrace{\underbrace{\underbrace{\underbrace{A_i{}^i{}_l}_{=B_l}\underbrace{A_j{}^j{}_m}_{=B_m}}_{=C_{lm}}\underbrace{A_k{}^k{}_n}_{=B_n}}_{=D_{lmn}}A^{mnl}}_{=M_{17}}$$

You would use in three dimensions:

| | $f$ | $c$ | $t$ | $n_a$ | $n_m$ |
|---|---|---|---|---|---|
| $B$ | 1 | 1 | 1 | 6 | 0 |
| $C$ | 2 | 1 | 2 | 0 | 9 |
| $D$ | 3 | 1 | 2 | 0 | 27 |
| $M_{17}$ | 0 | 3 | 2 | 26 | 27 |
| *sum* | | | | 32 | 63 |

You see that by decomposition, you save a lot ( You use 63 instead of 2916 multiplications )

If you decompose it in an other way:

$$\underbrace{\underbrace{A_i{}^i{}_l}_{=B_l}\,\underbrace{\underbrace{A_j{}^j{}_m}_{=B_m}\,\underbrace{\underbrace{A_k{}^k{}_n}_{=B_n}\,A^{mnl}}_{=C^{ml}}}_{=D^l}}_{=M_{17}}$$

You would use in three dimensions:

| | $f$ | $c$ | $t$ | $n_a$ | $n_m$ |
|---|---|---|---|---|---|
| $B$ | 1 | 1 | 1 | 6 | 0 |
| $C$ | 2 | 1 | 2 | 18 | 27 |
| $D$ | 1 | 1 | 2 | 6 | 9 |
| $M_{17}$ | 0 | 1 | 2 | 2 | 3 |
| *sum* | | | | 37 | 39 |

You see that by re-ordering the computations you achieve further savings (here 39 instead of 63 multiplications have to be used)

## 3.3.1 The tensor graph

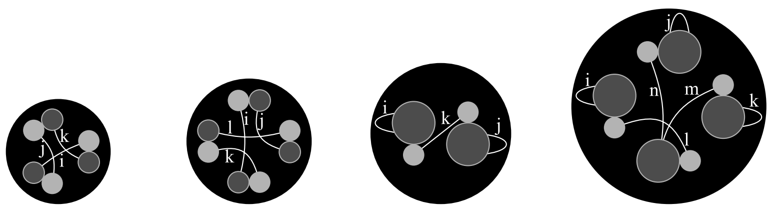
Further savings are achieved by re-using the intermediate tensors $B, C, D$ in other moments and minding the symmetry of the tensors. To recognize the tensors which are the same if index permutations are allowed, we convert them into graphs and compute their normal forms respective to index permutations.

formally speaking, our tensor graph consists of:

- a list of nodes $S$ that represent the tensors. One tensor can be in there multiply. The nodes are labeled by their tensor.

- a list of edges. Each edge consists of 2 node ids and 2 endpoint ids, the lower node id coming first. The endpoint ids denote the group of interchangeable indices in the tensor the edge is connected to. The list of edges is always lexicographically sorted by (1) it's node ids and (2) the according labels.

### 3.3.2 Representation of moment invariants as graphs

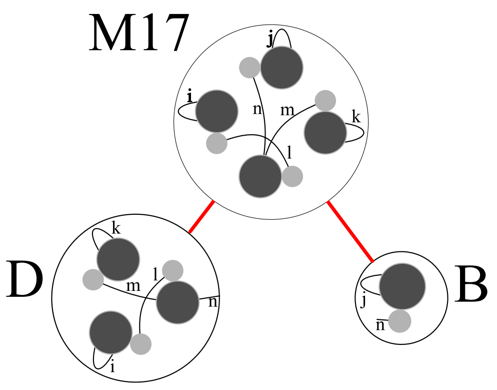


$A_{ij}A_k{}^jA^{ki}$ $M_6$ $A_{ij}A_{kl}A^{lj}A^{ki}$ $M_8$ $A_i{}^i{}_kA_j{}^{jk}$ $M_{12}$ $A_i{}^i{}_lA_j{}^j{}_mA_k{}^k{}_nA^{mnl}$ $M_{17}$

An edge here represents a contracted index, a group of directly attached balls represents a tensor, and a ball represents a group of interchangeable indices inside a tensor. In this example, tensors computed from vector fields are used which have special symmetry properties: The order-1 first indices are always interchangeable.

## 3.4 Representation of tensors as as graphs

here, the final step of composition as described previously is drawn as a graph. The tensors B and D are represented as a graph with one open edge each which represents a free(=uncontracted) index.



### 3.4.1 Normalization of the tensor graph

To normalize the graph, we first sort the nodes by their tensor. Then we we try out each permutation of the nodes that does not interchange nodes with different tensors and take that one where the graph is smallest in our special order: The graphs are compared by lexicographically comparing the list of edges. of course , the node ids of the edges have to be updated in each permutation and the edges have to be re-sorted.

### 3.4.2 Creation of the hypergraph

To optimize the computation of all moment invariants in the set, a directed hypergraph is created whose nodes are graphs. Every node again represents a tensor. Every node is linked to the (max. two) sub-nodes it was created from.

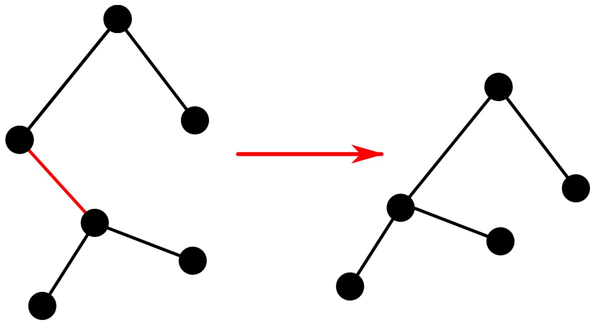
In the creation process, a sorted list of tensor graphs is maintained.

The process starts at the moment tensor graphs and tries to remove each edge recursively. (in one step, only one edge is removed)

It assesses each split by the cost function $n_m = d^f d^c t$ of the split and its sub-splits. Tensor graphs that are re-used are counted only once.

The permutations of nodes that lead to the normal forms of the sub nodes and the edges that were split are saved for later use.

### 3.4.3 Join nodes



Afterwards the nodes that have only one parent and whose parents have only one child are joined with their parent. This is done because the child has unnecessary free indices that makes its computation expensive: if you contract two indices, the number of free indices $f$ is decreased by two, the number of contracted indices $c$ only increased by 1. If you look at the cost function this means a saving of a factor of $d$.

In that join of course you have to be careful that you convert the permutations and split edges correctly.

## 3.5 Create the tensors from the graph

After the hypergraph has been created, the computation of the intermediate tensors and the moment invariants(the black graphs) has to be created from it. To do this, we sort the graph topologically into different levels: The first level consists of the nodes that don't have any subnodes, the upper levels of

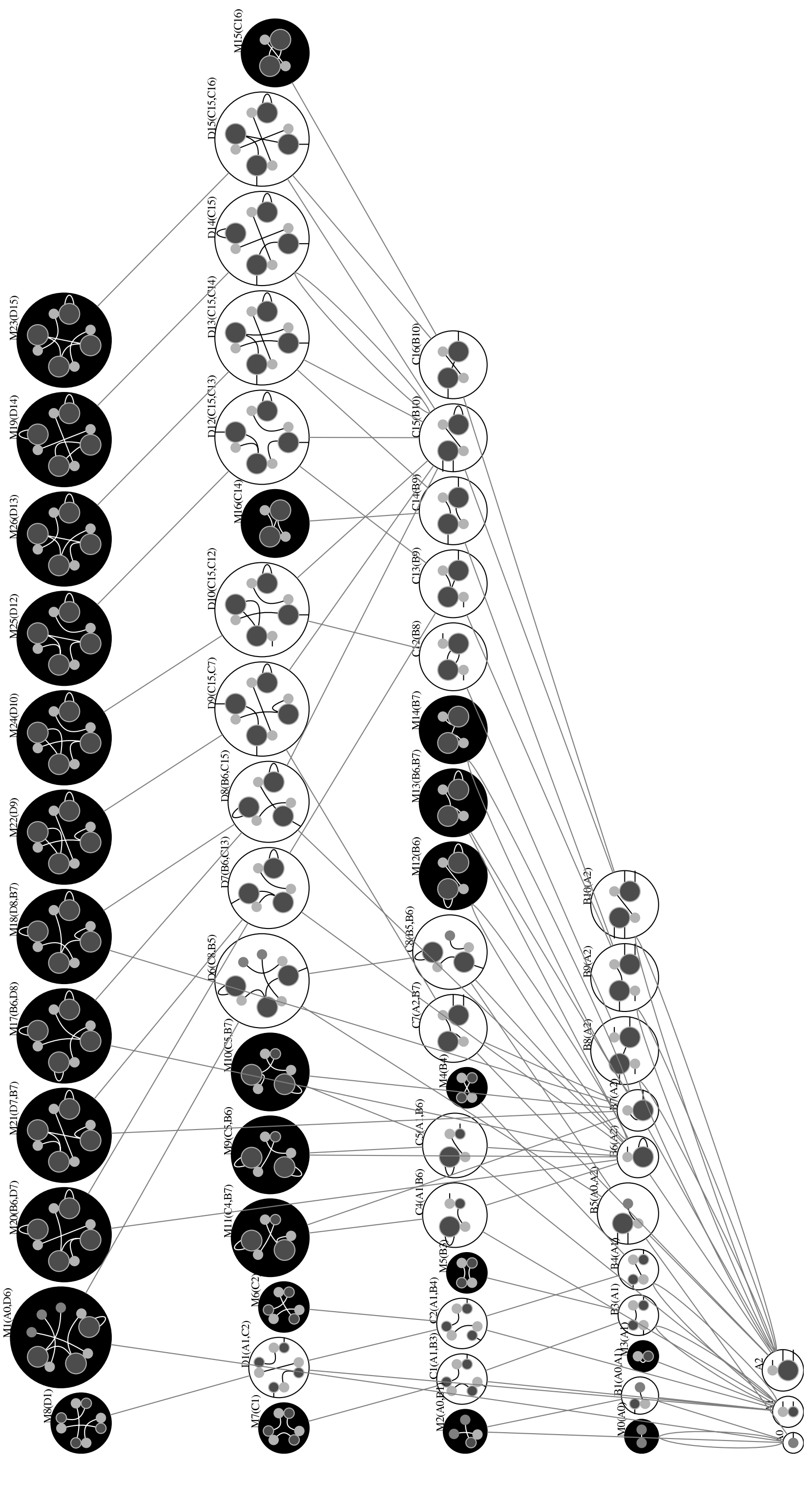


Figure 3.1: The tensor hypergraph before join (invariant set 1)

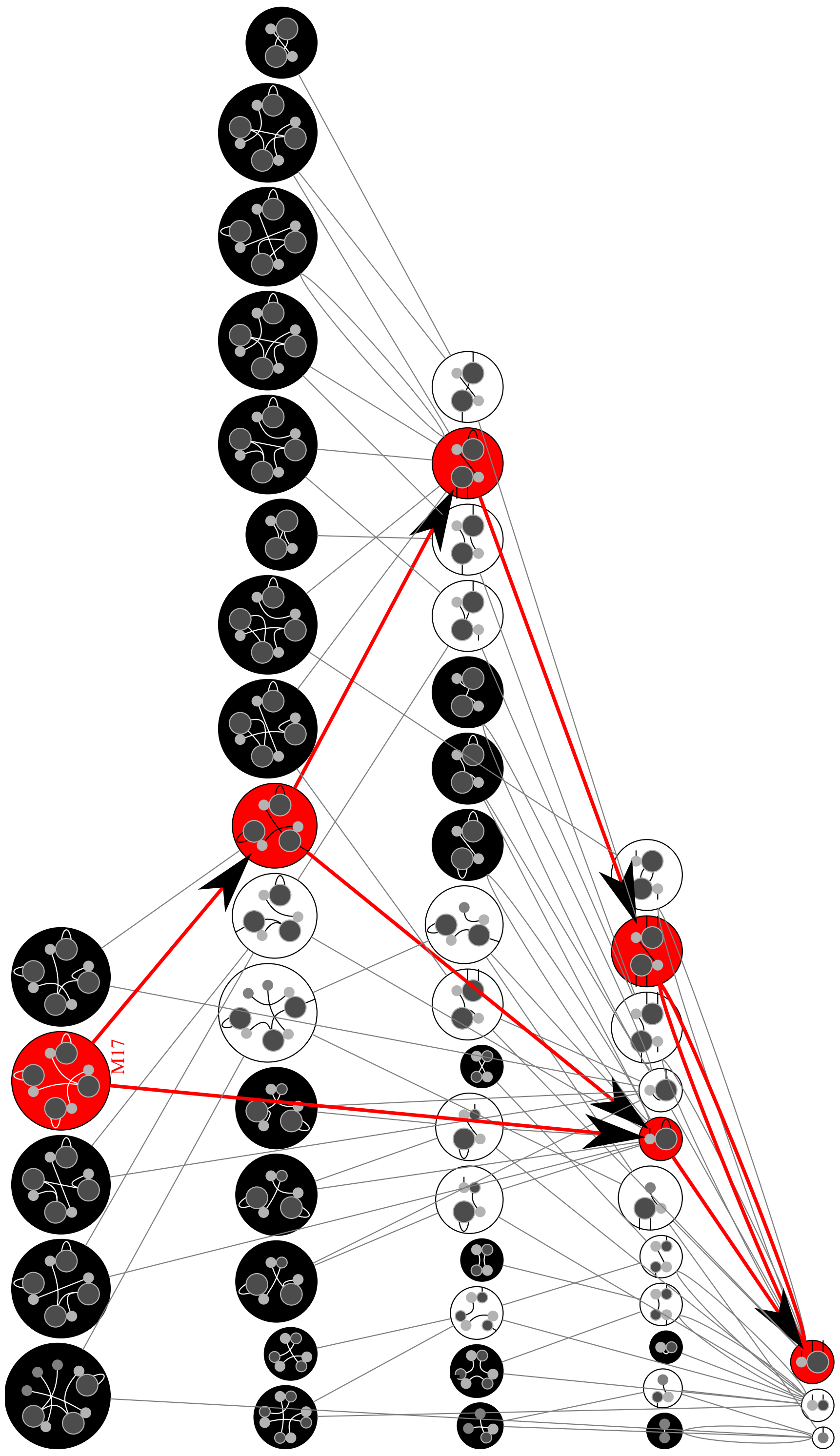


Figure 3.2: The tensor hypergraph (invariant set 1)

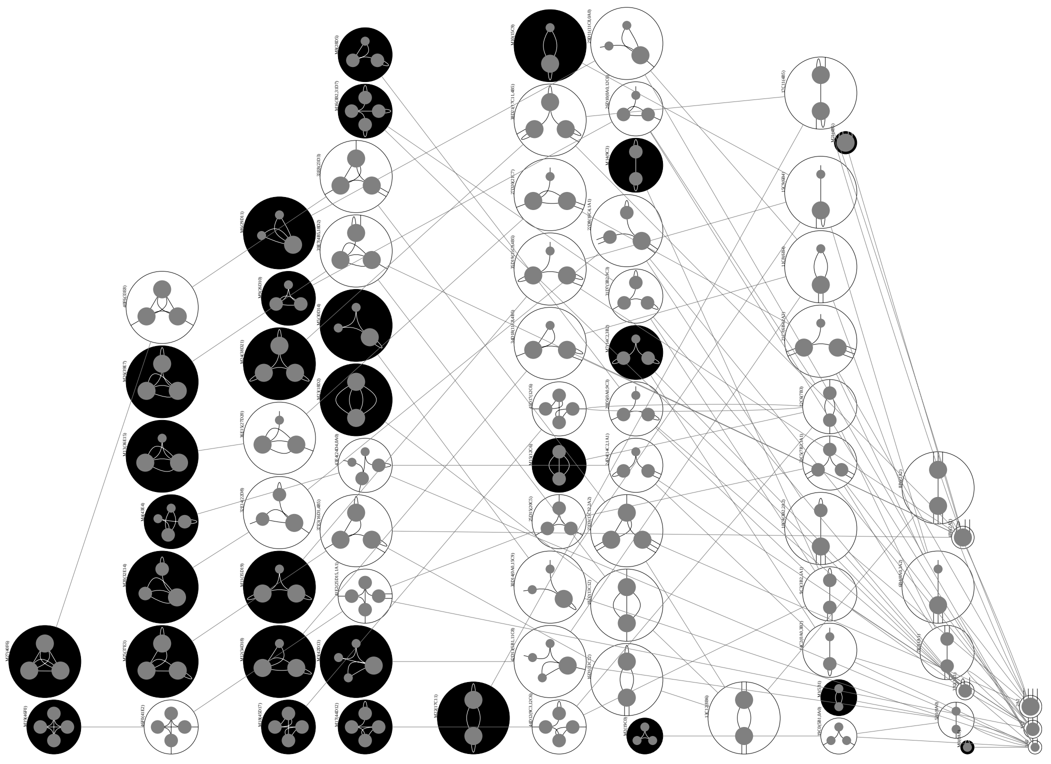


Figure 3.3: The tensor hypergraph before join (tensor set 2)

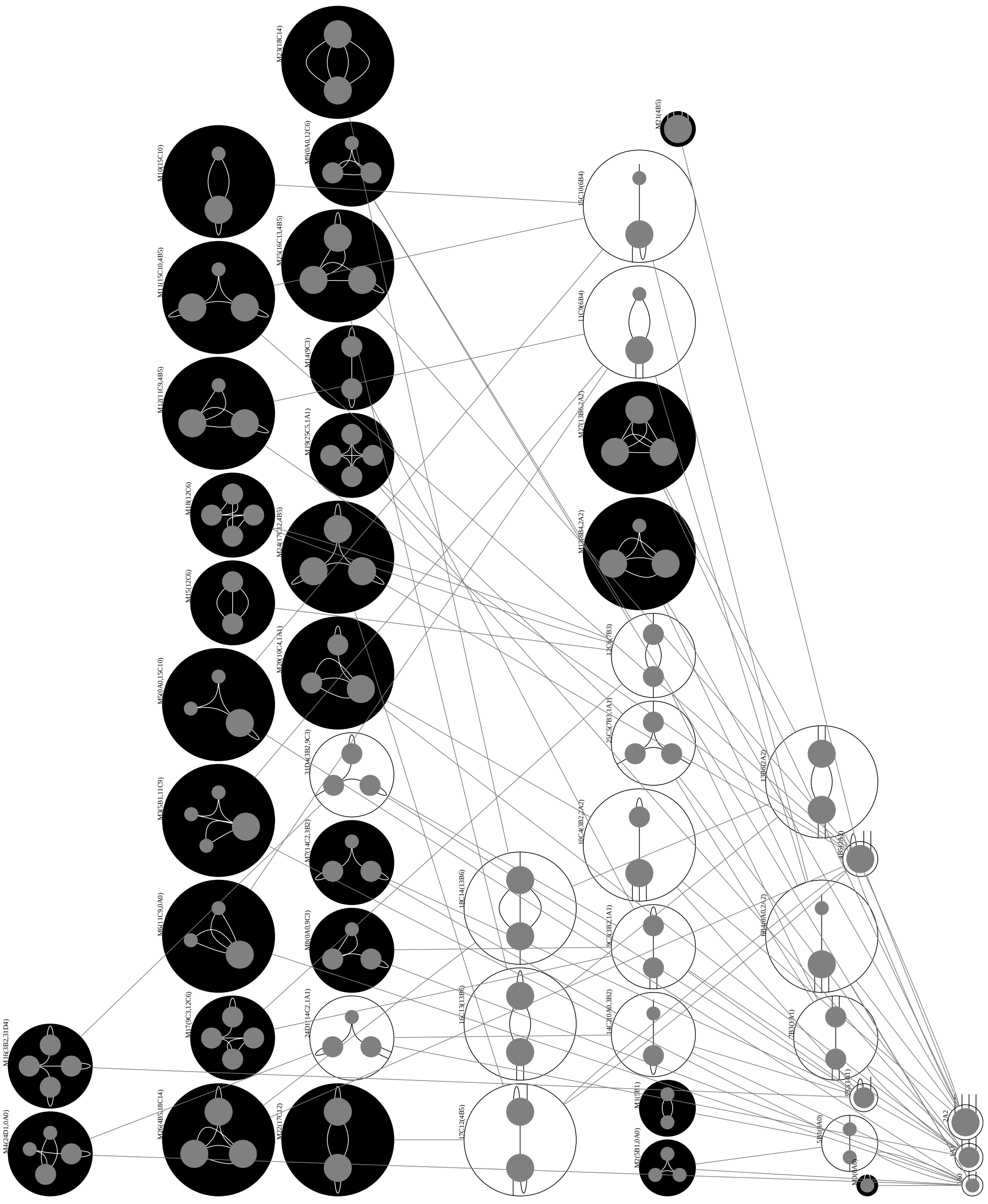


Figure 3.4: The tensor hypergraph (tensor set 2)

those nodes that only have subnodes in levels below. Then, the tensors are computed in that order.

### 3.5.1 Use symmetry

The symmetry should play a role here, so to compute one tensor, we use a special type that accounts for repeated entries because of symmetry (`repeatTensor`): For every unique component, we store a polynomial

- in the unique tensor components of the tensor(s) corresponding to the immediate subnodes, and
- in the unique tensor components of the overall source tensors (those on the lowest level of our topological order), to determine if they are duplicated.

For every possible tensor multi-index, we store the id of the unique component. From the outside, this type can just be accessed as a tensor with polynomials as components.

### 3.5.2 Example for repeatTensor

If you would have the tensor product $B_{ij} = a_i a_j$ with $\vec{a} = (x+y, x)$ you have the tensor

$$B = \begin{pmatrix} x^2+y^2+2xy & x^2+xy \\ x^2+xy & x^2 \end{pmatrix}$$

This is expressed with a repeatTensor as

$$\begin{pmatrix} 0 & 1 \\ 1 & 2 \end{pmatrix}, \quad \begin{pmatrix} a_0a_0 \\ a_0a_1 \\ a_1a_1 \end{pmatrix}, \quad \begin{pmatrix} x^2+y^2+2xy \\ x^2+xy \\ x^2 \end{pmatrix}$$

### 3.5.3 Compute the new tensor from its subnodes

For the new tensor we first create an object which represents the tensor that results from the specific tensor product. It is composed of objects that represent

- a tensor contraction. For this, the edge endpoints have to be converted to tensor indices using the symmetry groups in the tensor.
- a tensor product.

- a tensor with permuted indices. The index permutation has to be computed from the permutation of the nodes the indices are attached to.

To keep the memory usage low (the polynomials easily use a large portion of memory), the actual components of these objects are only computed if accessed. This tensor is then in turn converted to a repeatTensor with the properties as described above.

### 3.5.4 Create the actual C++-code

Every repeatTensor object gets assigned a unique id in computation order. If we have computed the repeatTensor objects, we have to convert their computation into C++-code: Every repeatTensor is represented by an array in its unique components with a short unique alphabetic name computed from its id. For each unique component, the polynomial in the unique components of it's sub-tensors is printed out. Further optimization could be achieved by leaving out unused components of the intermediate tensors and computing common products only once. This is not done yet.

## 3.6 Statistics for the computation of the moment invariants

Here, statistics for the computation of the numeric values after having converted the formulas to C++-code are given. In table 3.2 the statistics for an invariant set for Vector fields and in 3.3 the statistics for an invariant set for scalar fields and pointclouds are given. All times were measured on an Intel Xeon 3 Ghz, 32 bit , double precision. For every optimization method, the number of additions (Adds),muliplications (Mults),the runtime of the optimized code (Time), and the number of buffer float variables (Vars) needed is given. To assess the optimization, it is compared against three other methods to compute the invariants:

- A polynomial in DNF in the source tensor components (the components of ${}^0\!A$, ${}^1\!A$, ${}^2\!A$ )
- That polynomial ,but evaluated using a quasi-horner scheme (common factors are factored out)
- The factored-out polynomial optimized by computing common subexpressions of the different invariants first

Table 3.2: Statistics for invariants of moments computed from 3D vector fields

| Optimization Method | Adds | Mults | Time[$\mu s$] | Vars |
|---|---|---|---|---|
| none(polyn. in DNF) | 5378 | 19176 | 18 | 0 |
| factoring out | 5378 | 5946 | 6.8 | 0 |
| common subexpressions | 3676 | 2228 | 4.8 | 830 |
| tensor hypergraph | 624 | 780 | 0.68 | 201 |

Table 3.3: Statistics for invariants of moments computed from 3D scalar fields or point clouds

| Optimization Method | Adds | Mults | Time[$\mu$s] | Vars |
|---|---|---|---|---|
| none(polyn. in DNF) | 1779 | 5611 | 4.5 | 0 |
| factoring out | 1779 | 2833 | 2.6 | 0 |
| common subexpressions | 1513 | 1317 | 2.1 | 419 |
| tensor hypergraph | 596 | 686 | 0.92 | 198 |

# Chapter 4

# Application of Moment Invariants To Object Recognition in 3D Point Clouds

One can do object recognition 3D point clouds with our moment invariants. The point clouds stem e.g. from a stereo image or a laser scan.

To use these moments which are defined for scalarfields, you construct a scalar field consisting of a set of delta functions centered at the points. The tensors ${}^{m}\!\mathcal{A}$ are calculated as:

$$ {}^{m}\!\mathcal{A} = \int_{\mathbb{R}^3} (\vec{x}\otimes)^m \underbrace{\sum_i \delta(\vec{x} - \vec{x}_i)}_{f(\vec{x})} \, dV = \sum_i \underbrace{\vec{x}_i \otimes \ldots \otimes \vec{x}_i}_{m} \tag{4.1} $$

In our case, we use for the domain order of the source tensors defined in section 2.5.2 only the $\vec{x}$-order $m$.

## 4.1 Principle of our object recognition algorithm

Our algorithm will recognize objects saved in a reference database in an unknown pointcloud and compute a unlikelihood value for every object in our database to be contained in the unknown pointcloud. The unlikelihood values are all initialized to 1.

## 4.2 Build the database

For every object, you calculate a large number of invariant sets ( in our examples up to 50000). Each moment set is computed from a ball centered on the object's surface. For the centers, we use the gravity centers of the leaves of the octree structure we used for acceleration (see section 4.7) After having computed the invariant sets you do some coordinate transformations in the invariants' space to have a better distance function. (see section 4.6 ). To reduce the number of moment sets, you can cluster them (see section 4.6.1 ). After that, you insert them into a 28-dimensional adaptive kd-tree and store for each leaf the id of the object it belongs to.

## 4.3 Analyze an unknown pointcloud

If you want to analyze a certain unknown pointcloud of a surface, you calculate the moments sets of some randomly chosen balls centered on points of the pointcloud. Each of these invariant sets can be a witness that the pointcloud contains a certain object, so we call them witnesses. For each witness, you do the following:

- Get all invariant sets of the database which are inside a hypercube of max-distance 0.5 to the witness.
- Compute the distance $d_{\text{min}}$ of the closest of these invariant sets to the witness. Let $i_{\text{min}}$ be the id of the object this moment set is belonging to.
- Compute the distance $d_{\text{other}}$ of the closest moment to the witness which is inside the hypercube and not belonging to object $i_{\text{min}}$.
- If $1.2d_{\text{min}} < d_{\text{other}}$, we multiply the unlikelihood value for $i_{\text{min}}$ by $1 - (d_{\text{other}} - d_{\text{min}})$.

If one of the unlikelihood values drops below a certain threshold, we consider an object as recognized and stop. Usually 100 witnesses are enough to recognize the pointcloud reliably.

## 4.4 Probabilistic Reasonings

If we assume that the unknown pointcloud is a noised subset of one of the pointclouds in the database it follows that the invariant sets of the unknown pointcloud are a noised subset of the invariant sets of the pointclouds in the database. Let $o$ be the index of the objects and $i$ be the index of a moment set of an object. We can then compute the probability distribution of the 28-dimensional invariant sets with the following assumptions:

- We have for every component of the moment set noise $v_i$ that has a Gaussian distribution with $p(v_i) = \frac{1}{\sigma\sqrt{2\pi}} e^{-v_i^2/(2\sigma^2)}$ which gives a distribution of
$$p(\vec{v}) = \prod_{i=1}^{28} p(v_i) = ce^{-\vec{v}^2/(2\sigma^2)}, \quad c := \frac{1}{(\sigma\sqrt{2\pi})^{28}}$$
for the 28-dimensional vector of moment invariants.
- All objects are equally probable to be the sample.

We then have the basis set $\Omega = \{1..n\} \times \{1..m_k\} \times \mathbb{R}^{28}$ with the probability distribution

$$p(a) = \frac{p(\vec{x} - \vec{x}_i)}{nm_k}, \qquad a = (k, i, x) \in \Omega \qquad ,$$

where $n$ is the number of objects, $k$ the object id, $m_k$ the number of invariant sets of object $k$, $i$ the moment set id, and $x$ the value of an invariant set.

Then the probability of a set of invariants sets $x_j \in \mathbb{R}^{28}, j = 1..l$ stemming from object $k$ is

$$\begin{aligned}
&P(k' = k | x'_1 = x_1, ..., x'_l = x_l) \\
=&\frac{p(k' = k, x'_1 = x_1, ..., x'_l = x_l)}{p(x'_1 = x_1, ..., x_l = x'_l)} = \frac{P(k' = k)p(x'_1 = x_1, ..., x'_l = x_l | k' = k)}{p(x'_1 = x_1, ..., x_l = x'_l)} \\
=&P(k' = k)\frac{\prod_j p(x'_j = x_j | k' = k)}{\prod_j p(x_j = x'_j)} = P(k' = k)\prod_j \frac{p(x'_j = x_j | k' = k)}{p(x_j = x'_j)}
\end{aligned}$$

With $y_{ki} \in \mathbb{R}^{28}$ being the original invariant set number $i$ of object number $k$, the parts of the above equation are computed as:

$$p(x'_j = x_j | k' = k, i' = i) = \prod_{i=1}^{28} P(x'_i = x_i) \quad = ce^{-(x_j - y_{ki})^2/(2\sigma^2)} \tag{4.2}$$

$$p(x'_j = x_j | k' = k) = \frac{1}{m_k} \sum_{i=1}^{m_k} ce^{-(x_j - y_{ki})^2/(2\sigma^2)} \tag{4.3}$$

$$p(x'_j = x_j) = \frac{1}{n} \sum_{k=1}^{n} \frac{1}{m_k} \sum_{i=1}^{m_k} ce^{-(x_j - y_{ki})^2/(2\sigma^2)} \tag{4.4}$$

### 4.4.1 Approximation of probabilities for efficient computation

Because we don't want to inspect too many points, we limit the exact computation to a certain distance $d$ of the sample points and approximate the probability that the sample is from outside the cut-off-distance by the gaussian distribution of all object's distribution.

$$\begin{aligned} p(x'_j = x_j | k' = k) &= \frac{c}{m_k} \sum_{i=1}^{m_k} e^{-(x_j - y_{ki})^2//(2\sigma^2)} \\ &= \frac{c}{m_k} \Big( \sum_{i:||x_j - y_{ki}|| < d} e^{-(x_j - y_{ki})^2/(2\sigma^2)} + \underbrace{\sum_{i:||x_j - y_{ki}|| > d} e^{-(x_j - y_{ki})^2/(2\sigma^2)} \Big)}_{=:E_{kj}} \end{aligned}$$

We now want to approximate $E_{kj}$ without looking at the single points. You see that

$$E_{kj} = \sum_{i=1}^{m_k} \theta(d - ||x_j - y_{ki}||) e^{-(x_j - y_{ki})^2/(2\sigma^2)}$$

We approximate the point density of point of object $k$ in the neighborhood of $x_j$ by the density given by a multidimensional gaussian distribution with the mean and mean square of the invariant sets in the object $k$ at point $x_j$ and call it $p_k$. $E_{kj}$ is then Integral of $p_k e^{-(x_j - y_{ki})^2/(2\sigma^2)}$.

The distribution function of the euclidean distance of a vector, whose 28 components are independent and $N(0, \sigma^2)$ - gaussian distributed is

$$c' r^{27} e^{-r^2/(2\sigma^2)}, c' := \frac{1}{51011754393600\sigma^{28}}$$

(the term $r^{27}$ is proportional to the surface of a a ball in 28 dimensions). It takes its maximum at $r_{max} = \sigma 3\sqrt{3}$

### 4.4.2 A nice approximation of the bell function

Because we want to use it the bell function $e^{-r^2/2}$ can be nicely approximated by

$$b_p(r) = k(r)^p \theta(k(r)) \quad , k(r) = 1 - (r^2/2)/p$$

This function has a $C^{(p-1)}$-smooth transition to 0 at $r = \sqrt{2p}$ : The $l$-th derivative on the left side is

$$\frac{r}{p} \frac{p!}{(p-l)!} k^{p-l}$$

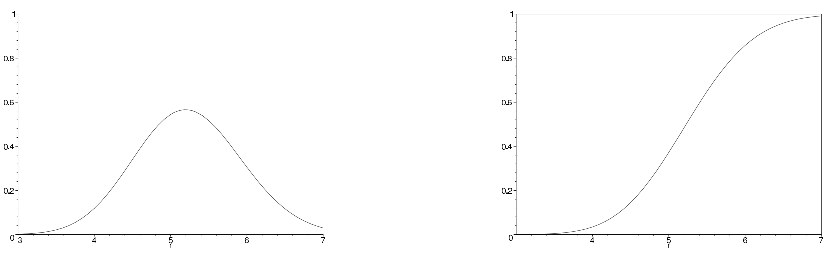


Figure 4.1: $c' r^{27} e^{-r^2/(2\sigma^2)}$ and its integral

and $k(\sqrt{2p}) = 0$, so the derivatives $0 \ldots p-1$ are all zero at the transition point. Because of the well-known limit $\lim_{n\to\infty}(1+x/n)^n = e^x$ this converges to the bell function for high $p$ values. By choosing $p = 2^n$ the exponentiation can be calculated by squaring $n$ times. For $n = 7$ you only need 7+2 = 9 multiplications and one subtraction to calculate it.

To get an approximation of the error of $b_p(r)$ relative to $e^{-r^2/2}$ in the range $0 ... \sqrt{2p}$,we first look for an approximation of the error of $(1 + x/p)^p$ relative to $e^x$ for $x/p << 1$: Using $e^{x/p} = 1 + x/p + (x/p)^2/2 + O((x/p)^3$ the error is:

$$\begin{aligned}
(1 + x/p)^p - e^x &= \left(e^{x/p} - (x/p)^2/2 - O((x/p)^3)\right)^p - e^x \\
&= e^x\left(1 - e^{-x/p}(x/p)^2/2 - O(e^{-x/p}(x/p)^3)\right)^p - e^x \\
&\approx e^x\left(1 - p\, e^{-x/p}(x/p)^2/2\right) - e^x \\
&\approx -\frac{x^2}{2p}e^x
\end{aligned}$$

Replacing $x$ by $-r^2/2$ , the approximation error of our formula is approximately $-\frac{r^4}{8p}e^{-r^2/2}$ , which takes its maximum negative value at 2, so a good approximation for the approximation error is $-0.3/p$.

## 4.5 Compute the moments and their invariants

In the following, will use $f(\vec{x}) = \sum_i \delta(\vec{x} - \vec{x}_i)\theta(r - ||\vec{x} - \vec{c}||)$, where $\theta$ is the Heaviside function $\theta(x) = (x > 0?1 : 0)$. Inserted into the equation 4.1, this

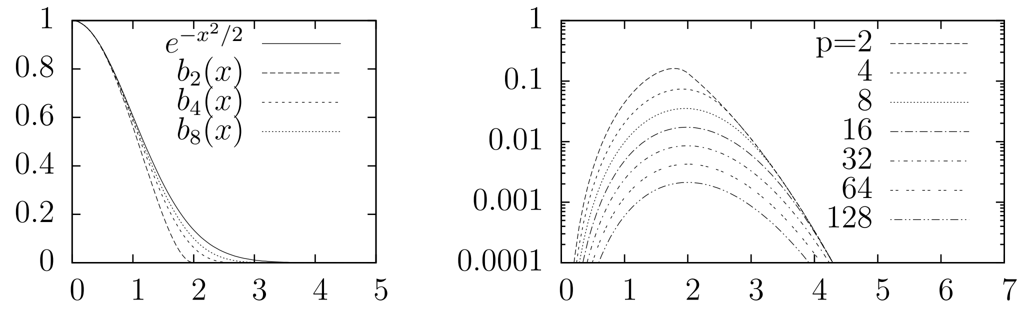


Figure 4.2: approximation of the bell curve $e^{-x^2/2}$ with $b_p(x)$: to the left the approximation curves, to the right the approximation errors for different $p$ values

leads to

$$ {}^{m}\!A(\vec{c}, r) = \sum_{\{i:||\vec{x}_i - \vec{c}|| < r\}} \underbrace{\vec{x}_i \otimes \dots \otimes \vec{x}_i}_{m} $$

where $\vec{c}$ is a point on an object's surface and $r$ is a fixed radius, preferably being a power of two. $r$ will be set to e.g. $2^{\lfloor \ln_2 \frac{\sigma_r}{16} \rfloor}$, with $\sigma_r$ being the standard deviation of the pointcloud containing the reference object. We use a power of two for the radius so we have a set of fixed radii which are independent of the objects. $\theta(r - ||\vec{x} - \vec{c}||)$ is rotationally invariant around $\vec{c}$, so we don't introduce a dependency of rotation by using this cut-off function. For every center $\vec{c}$, we compute the tensorset ${}^{0}\!A, ..., {}^{4}\!A$.

This tensorset will be divided by the number of points it is computed from given in ${}^{0}\!A$, so the point density on the surface is allowed to vary. ${}^{0}\!A$ is 1 after that.

It is also translated to it's center of gravity given in ${}^{1}\!A$ (see 2.4), so it is not too bad if there is much noise in the dataset and the center point $\vec{c}$ is not exactly on the surface. ${}^{1}\!A$ is 0 after that.

After that, you domain-scale the tensors by the inverse radius of the ball ( ${}^{m}\!A(\vec{c}, r)' = {}^{m}\!A(\vec{c}, r) r^{-m}$ ).

Then, you can compute the rotational invariants from the moments in $A$ (see table 2.3)

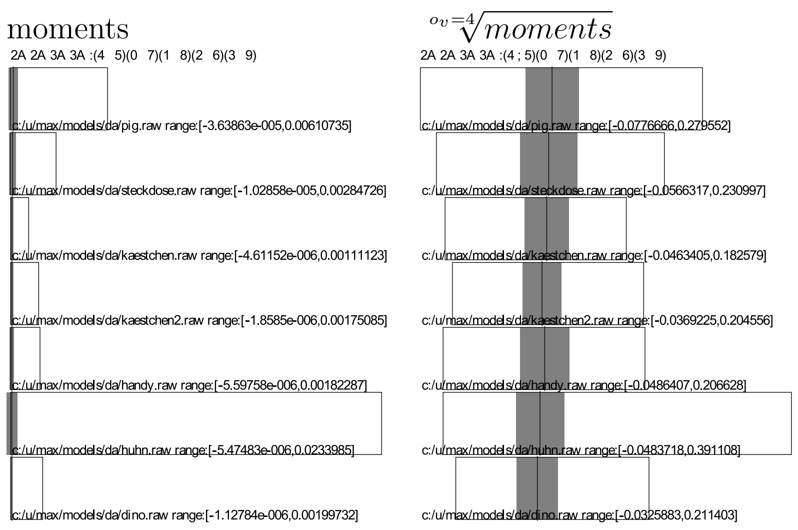


Figure 4.3: moment statistics, min-max range with a grey bar of 1 standard deviation to both sides of the average

## 4.6 Coordinate transformations of the moments to have a nice distance measure

If you have a large set of invariant sets, you still don't know how to measure the distance between these high-dimensional values. We do the following here:

- We take the $o_v$-th root of the moments, so that the distribution of the higher-order moments is less imbalanced (see figure 4.3), with $o_v$ being the value order of the moment defined in section 2.5.1.

- We divide the moments by the standard deviation of the moments from all invariant sets in the database, so the distance gives a measure of how unlikely a certain value is.

If you look at figure 4.4, you see that there are two principal directions in the set of invariant sets.

**Making the invariant set components statistically independent** It proved to be good to transform the invariant sets with a linear transform where the covariance matrix

$$C_{ij} = 1/n \sum_{k=1}^{n} (x_{ki} - \overline{x_i})(x_{kj} - \overline{x_j})$$

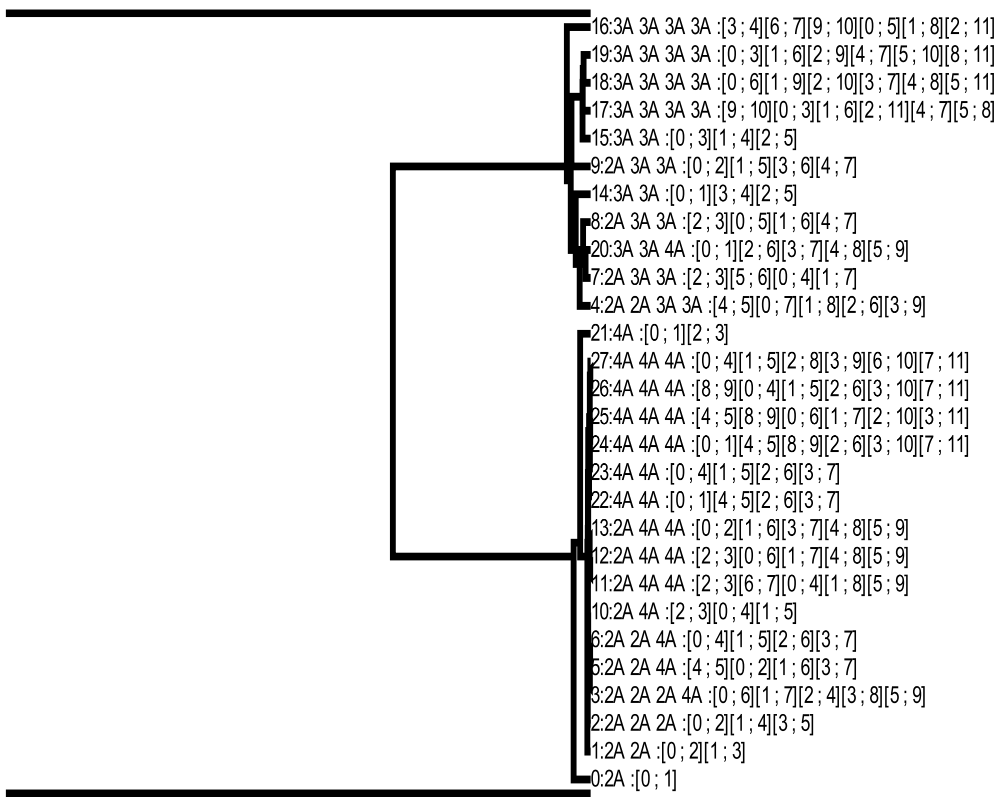

16:3A 3A 3A 3A :[3 ; 4][6 ; 7][9 ; 10][0 ; 5][1 ; 8][2 ; 11]
19:3A 3A 3A 3A :[0 ; 3][1 ; 6][2 ; 9][4 ; 7][5 ; 10][8 ; 11]
18:3A 3A 3A 3A :[0 ; 6][1 ; 9][2 ; 10][3 ; 7][4 ; 8][5 ; 11]
17:3A 3A 3A 3A :[9 ; 10][0 ; 3][1 ; 6][2 ; 11][4 ; 7][5 ; 8]
15:3A 3A :[0 ; 3][1 ; 4][2 ; 5]
9:2A 3A 3A :[0 ; 2][1 ; 5][3 ; 6][4 ; 7]
14:3A 3A :[0 ; 1][3 ; 4][2 ; 5]
8:2A 3A 3A :[2 ; 3][0 ; 5][1 ; 6][4 ; 7]
20:3A 3A 4A :[0 ; 1][2 ; 6][3 ; 7][4 ; 8][5 ; 9]
7:2A 3A 3A :[2 ; 3][5 ; 6][0 ; 4][1 ; 7]
4:2A 2A 3A 3A :[4 ; 5][0 ; 7][1 ; 8][2 ; 6][3 ; 9]
21:4A :[0 ; 1][2 ; 3]
27:4A 4A 4A :[0 ; 4][1 ; 5][2 ; 8][3 ; 9][6 ; 10][7 ; 11]
26:4A 4A 4A :[8 ; 9][0 ; 4][1 ; 5][2 ; 6][3 ; 10][7 ; 11]
25:4A 4A 4A :[4 ; 5][8 ; 9][0 ; 6][1 ; 7][2 ; 10][3 ; 11]
24:4A 4A 4A :[0 ; 1][4 ; 5][8 ; 9][2 ; 6][3 ; 10][7 ; 11]
23:4A 4A :[0 ; 4][1 ; 5][2 ; 6][3 ; 7]
22:4A 4A :[0 ; 1][4 ; 5][2 ; 6][3 ; 7]
13:2A 4A 4A :[0 ; 2][1 ; 6][3 ; 7][4 ; 8][5 ; 9]
12:2A 4A 4A :[2 ; 3][0 ; 6][1 ; 7][4 ; 8][5 ; 9]
11:2A 4A 4A :[2 ; 3][6 ; 7][0 ; 4][1 ; 8][5 ; 9]
10:2A 4A :[2 ; 3][0 ; 4][1 ; 5]
6:2A 2A 4A :[0 ; 4][1 ; 5][2 ; 6][3 ; 7]
5:2A 2A 4A :[4 ; 5][0 ; 2][1 ; 6][3 ; 7]
3:2A 2A 2A 4A :[0 ; 6][1 ; 7][2 ; 4][3 ; 8][5 ; 9]
2:2A 2A 2A :[0 ; 2][1 ; 4][3 ; 5]
1:2A 2A :[0 ; 2][1 ; 3]
0:2A :[0 ; 1]

Figure 4.4: The different invariants clustered using the correlation coefficients as inverse distance function. The x-coordinate of the vertical connection lines in the tree give the correlation coefficient between the clusters connected

becomes the unit matrix, as used e.g. in principal component analysis. After the transform the euclidean distances are proportional to the standard deviation in that direction. The resulting distance metric on the untransformed data is known as Mahalanobis distance.

Here a short description how it is done: The covariance Matrix of a set of vectors transformed with matrix $M$ is $C = MC'M^T$, if $C'$ is the covariance matrix of the untransformed ones. With an eigenvector decomposition $C$ can be expressed as $VDV^{-1}$, with $V$ being the orthonormal eigenvector matrix and $D$ being a diagonal matrix holding the eigenvalues on the diagonals. $D$ can be expressed as a product $EE^T$, with E being a diagonal matrix with $E_{ii} = \sqrt{D_{ii}}$ so it holds : $C = VEIE^TV^T$ , with $I$ being the unit matrix. If we set $M = VE$ and $I = C'$ then we have $C = MIM^T$. So, to transform from vectors with cov.-mat. $C' = I$ to vectors with cov. mat $C$ you have to multiply with $M$. To transform from vectors having cov.mat. $C$ to ones having cov.mat $I$, you have to transform with $M^{-1} = E^{-1}V^{-1}$. Here, $V^{-1} = V^T$, $E^{-1}$ is diagonal and $E_{ii}^{-1} = 1/\sqrt{D_{ii}}$.

### 4.6.1 Thin out the invariant sets

If you have too many invariant sets it helps the efficiency to keep only one invariant set for each box of a certain size. To achieve that with 28-dimensional data where keeping an entry for each grid cell inside the bounding hypercube is impossible, you can do the following:

- Scale all values so that the desired cluster distance is 1.
- Round all values in the moments sets to integers.
- Insert the integer invariant sets into a sorted list where the ordering criterion is the lexicographical order of the integer value sets.
- For every new moment set look up it's rounded version in the list. If it is already in there, discard it.

With this clustering technique it is still possible that some very close invariant sets have different rounded versions and are not discarded. To overcome this, one can use the following method: use $d + 1 = 29$ mappings

$$s_k : \mathbb{R}^d \to \mathbb{Z}^d, \quad s_k(x_1, ..., x_d) = (s_k(x_1), ..., s_k(x_d))$$

with

$$s_k : \mathbb{R} \to \mathbb{Z}, \quad s_k(x) = \lfloor x_1 - tk \rfloor \quad , t = 1/(d+1), k \in \{0, ..., d\}, d = 28$$

Insert a new invariant set only if none of its mappings is contained in the existing sets of integer vectors corresponding to a mapping. This method uses the following theorem:

**Theorem 1** *Each pair of points $\vec{x}, \vec{y}$ which is closer than $t$ in all dimensions ($||\vec{x} - \vec{y}||_\infty < t$), has got at least one mapping $s_k$ where it is mapped to the same integer vector.*

PROOF Let $z : \mathbb{R} \to \mathbb{Z}$ be the mapping $z(x) = \lfloor x/t \rfloor$, and $x \equiv_k y \Leftrightarrow s_k(x) = s_k(y)$. Obviously $|x_i - y_i| < t \quad \Rightarrow \quad |z(x_i) - z(y_i)| \leq 1$. One sees that $s_k(x) = \lfloor (z(x) - k)/(d+1) \rfloor$. It follows $z(x_i) = z(y_i) \forall i \quad \Rightarrow \vec{x} \equiv_k \vec{y}, k = 0..d$. One also sees that $s_k(x)$ does its jumps where $z(x_i)$ jumps from $(d+1)n + k - 1$ to $(d+1)n + k$ ($n \in \mathbb{N}$), and the numbers $x$ where $z(x)$ is in the range $k + n(d+1)...k + d + n(d+1)$ are all mapped to $n$ by $s_k$. So, if $|z(x_i) - z(y_i)| = 1$, $x_i$ and $y_i$ are only *not* equivalent in the relation $=_{\max\{z(x_i),z(y_i)\} \mod (d+1)}$ because there, $k \equiv \max\{z(x_i), z(y_i)\} \mod (d+1)$ and $k - 1 \equiv \min\{z(x_i), z(y_i)\} \mod (d+1)$ So, for every dimension $i$ there is at most one mapping where $x_i$ and $y_i$ are not equivalent. As we have $d+1$ mappings, but only $d$ dimensions, there is always at least one mapping remaining where $\vec{x}$ and $\vec{y}$ are equivalent in all dimensions.

## 4.7 Optimized computation of the moment tensor set $A$

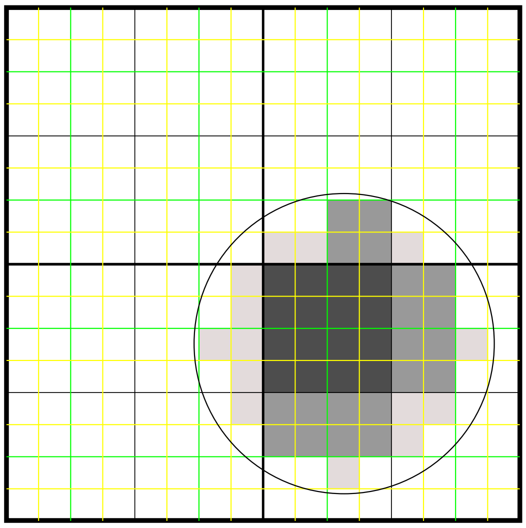
We want to compute

$$A(\vec{c}, r) = \left\{ \ {}^m\!A(\vec{c}, r) = \sum_{\{i:||\vec{x}_i - \vec{c}|| < r\}} \underbrace{\vec{x}_i \otimes ... \otimes \vec{x}_i}_{m} \qquad , \qquad m = 0...4 \ \right\}$$

for many different $\vec{c}$ in an optimized and possibly approximative way so we do not have to sum up the tensorsets for all points inside the ball $\{\vec{x} : ||\vec{x} - \vec{c}|| < r\}$ individually. The idea for that is to build an octree that stores in every node the sum of tensorsproducts computed from the points inside it's box. The summation process inside the ball is then a recursive one as shown in figure 4.5.

### 4.7.1 Building the octree from grid cells

To build the octree for a pointcloud, we first sort the points into the cells of a regular grid, with a cell size of $r_b = 2^{\lfloor \ln_2 \frac{\sigma_r}{128} \rfloor}$ with $\sigma_r$ being the standard deviation of the distance of the points to the point-cloud center. We use a



```
sum_inside_ball(A,node,ball)
{
    if(node.box.is_inside(ball) )
        A+= node.A;
    else if(!node.box.is_outside(ball) && ! node.is_leaf() )
        //box is not outside ball, so it intersects the ball:
        for(int i=0;i<8;++i)
            sum_inside_ball(A , node.children[i], ball);
}
```

Figure 4.5: A picture and C++ pseudo-code for the hierarchical summation process

power of two for the cell size to avoid rounding problems when converting from the source coordinates to grid coordinates. Then we build a regularly subdivided octree with these cells as leaves. The root node of the octree will have a power of two as edge length that is greater or equal to the maximum dimension of the points' bounding box.

For each leaf of the octree, we compute the tensorset

$$A(B) = \Big\{ \sum_{\{i:x_i \in B\}} \underbrace{\vec{x}_i \otimes ... \otimes \vec{x}_i}_{m}, \quad m = 0...4 \Big\}, B = \text{Box of the leaf}$$

and store it there. This tensorset consists of 35 numbers and is efficiently computed with 31 multiplications per point. After that, we store in each node of the octree the sums of the tensorsets of it's children.

### Sort the points into the grid cells

To sort the points into the grid cells, we build a 3-dimensional array $v_{ijk}$ of integers of size

$$\underbrace{\lceil (x_{\max} - x_{\min})/r_b \rceil}_{:=n_x} \times \underbrace{\lceil (y_{\max} - y_{\min})/r_b \rceil}_{:=n_y} \times \underbrace{\lceil (z_{\max} - z_{\min})/r_b \rceil}_{:=n_z}$$

with each entry corresponding to a grid cell. Then we count the points in each grid cell by applying the following algorithm:

1. Set all entries of $v_{ijk}$ to 0

2. For all points $\vec{x}_i = (x_i, y_i, z_i)$,
   add 1 to to $v_{\lfloor (x_i - x_{\min})/r_b \rfloor \lfloor (y_i - y_{\min})/r_b \rfloor \lfloor (z_i - z_{\min})/r_b \rfloor}$

After that, we sort the points into the cells by applying the following algorithm:

1. Create an integer array $w$ which contiguously stores all non-zero entries of $v$ and set each corresponding entry in $v$ to it's index in array $w$.

2. Create an array of coordinates $c$ with the same size as the number of points

3. Convert the entries in $w$ to offset indices in $c$: $w'_0 = w_0$ , $w'_{i+1} = w'_i + w_{i+1}$.

4. For every point $\vec{x}_i = (x_i, y_i, z_i)$ do the following:

   - Calculate it's index $l$ in the array w:

   $$l := v_{\lfloor (x_i - x_{\min})/r_b \rfloor \lfloor (y_i - y_{\min})/r_b \rfloor \lfloor (z_i - z_{\min})/r_b \rfloor}$$

   - Set $\vec{c}_{w_l - 1}$ to $\vec{x}_i$, set $w_l$ to $w_l - 1$.

After that procedure, the point coordinates in the cell $(i, j, k)$ are

$$\{\vec{c}_p, \quad p \in [w_{v_{ijk}}, w_{v_{ijk}+1})\}$$

#### Build the octree

To avoid traversing the octree for each cell we insert, we apply a bottom-up strategy, where we only traverse the tree without reading any data. This can be done because the octree is uniformly subdivided and the children's bounding boxes can be computed without any additional memory access. We start with a box of edge length $2^{\lceil \ln_2 \max\{n_x,n_y,n_z\}\rceil}$ in grid coordinates and recursively subdivide until we reach an edge length of 1. There we create a leaf if the corresponding grid cell is not empty. In the recursion level above, we create a node if the recursion into the children returned any leaves and directly add the leaves, and so on. The complexity of this procedure is linear in the total number of grid cells, and the complexity of the memory access is linear in the number of tree nodes which in most cases is linear dependant to the number of filled leaves. Because memory access is always the most expensive part of an algorithm, we prefer this bottom-up algorithm to a top-down algorithm with a memory access complexity which is log-linear in the number of leaves.

### 4.7.2 Alternative: Building the Octree with z-index

#### The z-index

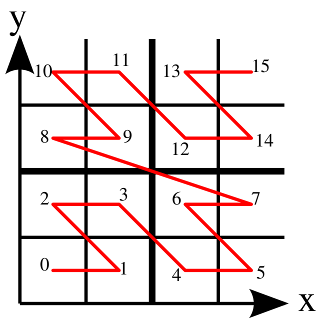


Figure 4.6: The z-curve in 2d with two levels. You see that the ranges 0-3,4-7,8-11,12-15 correspond to a quad, a node in a quadtree

The well-known z-index is a number that is calculated from the integer coordinates and resembles the position in the fractal z-curve (see figures 4.6,4.7). It has the advantage that the membership in a regular octree's node corresponds to an interval in the z-index. In contrast to the hilbert curve which could also used, it is discontinuous, but it is much easier and faster to compute. Let $x_i, y_i, z_i$ denote the i'th bit of the x,y and z coordinate

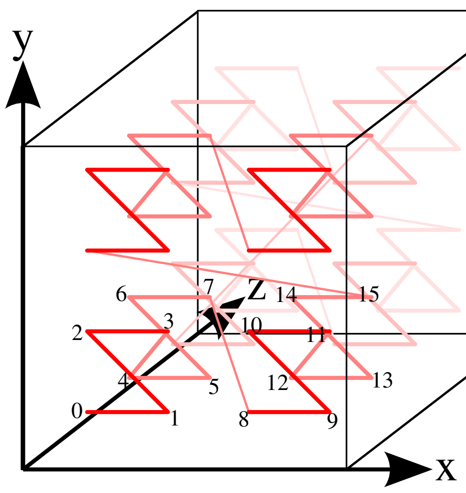


Figure 4.7: The z-curve in 3d with two levels. You see that the range 0-7 corresponds to the front lower left cube and the range 8-15 corresponds to the front lower right cube, i.e. a node of an octree.

respectively. then the z-index is in binary notation $...z_2y_2x_2z_1y_1x_1z_0y_0x_0$ or, more formally $\sum_{i=0}^{n}(2^{3i}x_i + 2^{3i+1}y_i + 2^{3i+2}z_i)$.

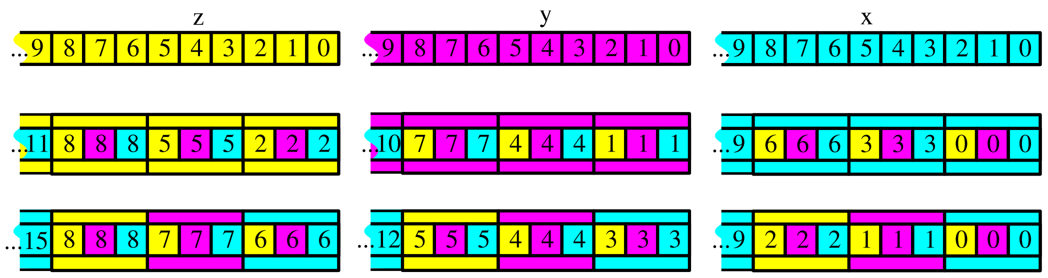


Figure 4.8: z-index computation using bit parallelity: first 3 steps

To compute it using bit-parallelity we use the following iteration using buffer variables $b_j^{(k)}, j \in \{0,1,2\}$ and mask constants $m_j^{(k)}$ with & being the bitwise-and-operator, and $n$ the number of bits in the number format:

$$b_0^{(0)} := x, \quad b_1^{(0)} := y, \quad b_2^{(0)} := z \tag{4.5}$$

$$m_j^{(k)} := 2^{j3^k} \sum_{i=0}^{\lfloor n/(3^{k+1}) \rfloor} 2^{i3^{k+1}}(2^{3^k} - 1) \tag{4.6}$$

$$b_j^{(k+1)} := \sum_{l=0}^{2} 2^{(l-j)3^k}(b_l^{(k)} \& m_j^{(k)}) \tag{4.7}$$

The sums in the formulae are computed with bitwise ors (because the 1-bits of the summands don't overlap) and the multiplication with powers of two with bit-shifts. The logic behind it is that you put always 3 blocks of size $3^k$ together in correct order: in $b_0^{(k+1)}$ the zeroth,third, sixth .. block is stored, in $b_1^{k+1}$ the first, fourth, seventh block is stored and so on. The size of the blocks with consecutive bits in correct order is multiplied by 3 in every step. In $b_0^{(k)}$ there are always $3^k$ bits of the correct result. As soon as $3^k$ is greater than three times the maximum number number of trailing bits of the three input variables, the computation can be stopped. Example: if $x, y, z < 2^{10}$ (10 trailing bits) the correct result is returned in $b_0^{(4)}$ because $3^4 = 81 >= 3 \cdot 10$. In table 4.1 you see a c++-implementation with loops unrolled, and in figure 4.8 you see how the bits are moved around.

### Build the tree

To compute the octree from the point cloud using the z-index and a desired maximum number $n_{pt}$ of points per octree leaf, the following steps are done:

1. Compute a bounding box $[x_1, x_2] \times [y_1, y_2] \times [z_1, z_2]$ for the pointcloud
2. Compute the zindex for all points using a base length of $2^{p-10}$, where $p = \lceil \ln_2 max\{x_2 - x_1, y_2 - y_1, z_2 - z_1\} \rceil$, and a coordinate origin of $(x_1, y_1, z_1)$. This is done so no integer coordinate is $\geq 2^{10}$ and so its z-index fits into a 32-bit integer.
3. Create an array with tuples of z-index and coordinate index and sort it by the z-index (using e.g. the c++ stl). This is very quick because sorting is a standard operation that is optimized well.
4. Calculate for every pair of consecutive z-indices in that array the number of the highest bit where they differ divided by three and rounded up, and store it in an array $h$. This gives the height of the lowest octree node both points corresponding to the indices belong to. For the following steps to work, the first and last entry in the array is set to the maximum tree height (most probably 10 here).
5. Calculate offsets $o$ in the sorted coordinate array that correspond to the octree's leaves and at the same time calculate the number of upper octree nodes by iterating through $h$. In every iteration do the following:
   (a) Add an entry in an offsets array $o$ pointing to the current index in $h$. Each entry of the offset array corresponds to a leaf in the octree to be built. Save the height at the current index in a variable $h_c$.

(b) Walk right through $h$ while the height is lower than $h_c$ and either less than $n_{pc}$ entries have been visited or the the z-index did not change (i.e. the height is zero).

Take the index of the highest entry visited in that walk to be the next current index. Also increase the octree's number of parent nodes by the height difference between the new entry and the previous one, which corresponds to the number of parent nodes that are above the current leaf and not present in the previous leaves.

6. Initialize the leaf and parent node arrays to the sizes as calculated in the previous step.

7. Initialize the leaves from the point sets as given by the offset indices (Each pair of consecutive offsets describes a set of points), i.e. create the tensorsets from the points. Initialize the parent nodes: create so many parent nodes as the height difference indicates. Maintain an array of node pointers that represents the current path to the leaf. Set one of the eight child pointers of the newly created parent nodes as indicated by the corresponding 3 bits in the z-index to the corresponding child.

8. Initialize the tensor sets stored in the parent nodes as sum of the children's tensor sets.

## 4.8 Results

Three databases of objects were tested:

1. DENKER: A database scanned by a colleague of mine, Klaus Denker. It consists of seven pointclouds with 500k to 4M points, which were laserscans of some small objects: a piggy bank, two boxes, a power socket, a chicken puppet, a dinosaur model, and a handy. Results in figures 4.10,4.11 and table 4.2.

2. 3DRMA: A database of faces scanned at the Royal Military Academy in Belgium[1] . It consists of 120 faces with 5000 - 15000 points per face. Each face was scanned in different angles. You see the results in figures 4.12,4.13.

3. GAVAB: A database of faces scanned at the Groupo Investigation GAVAB, published in [12]. I used the vertices of the vrml meshes they created from laser scans for recognition here. There were 60 faces

in it scanned at different angles. Each face consisted of 5000-15000 points. In figures 4.12,4.13 you see the recognition results.

After building the database, we tested the recognition of the objects with a version of an object with some random noise applied.The recognition happened to find the correct object in the database until the length of the noise vectors exceeded the radius of our balls that we used to compute the moments. Then we tested if we could cut away part of the object and would still recognize it. We have cut away half of the points with a random cuttingplane on the center of gravity and still recognized the objects correctly. In figure 4.9 you see the noise applied to a face pointcloud.

The two face databases had multiple versions of each face scanned from different directions. For my benchmarks, I put one version of each face into my moment database. Afterwards, I tested for each face if my algorithm finds the correct face in the database if given it's pointcloud. Firstly I tested it for the pointcloud version of the face that was in the database and secondly for a pointcloud version of the face that was not in my database.

I have the following conclusions:

- Given enough points, the system always finds the correct result (as you see for the DENKER database in figure 4.10 ).
- To reduce scanning artifacts that arise e.g. from the direction of the laser lines, it is good to add some random displacements, as you see in figure 4.14 on the left where noiselevel 0.01 gives better recognition results than no noise at all.

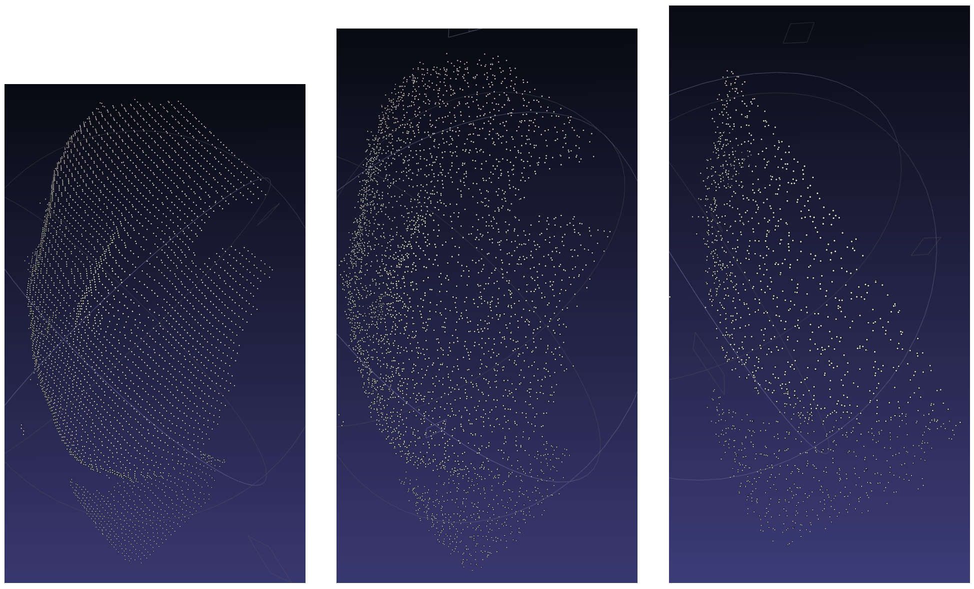


Figure 4.9: The noise that was applied for testing.

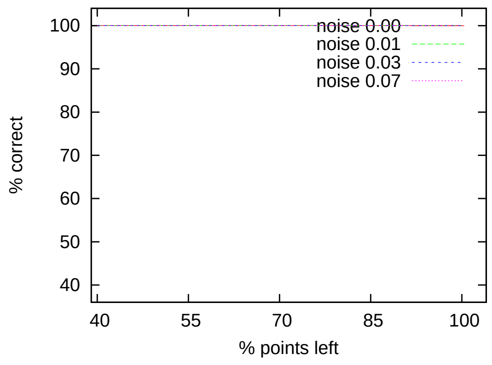


Figure 4.10: Database Denker: Accurracy under noise for the dino

```
template<class uint>
inline uint zord(uint x, uint y, uint z)
{
uint xx,yy,zz;
uint xyz = x|y|z;
//binary ...01001001001001001001001001001001
static const uint A0=0111111111u;
static const uint a0=(((A0<<27)|A0)<<27)|A0;
xx = (x&a0) | ((y&a0)<<1) | ((z&a0)<<2);
if( (xyz>>1)==0 ) return xx;
static const uint a1=a0<<1;
yy = ((x&a1)>>1) | (y&a1) | ((z&a1)<<1);
if( (xyz>>2)==0 ) return xx|(yy<<3);
static const uint a2=a0<<2;
zz = ((x&a2)>>2) | ((y&a2)>>1) | (z&a2);
//binary ...00111000000111000000111000000111
static const uint B0=07007007;
static const uint b0=(((B0<<27)|B0)<<27)|B0;
x =  (xx&b0) | ((yy&b0)<<3) | ((zz&b0)<<6);
if( (xyz>>3)==0 ) return x;
static const uint b3=b0<<3;
y =  (xx&b3)>>3 | ((yy&b3)) | ((zz&b3)<<3);
if( (xyz>>6)==0 ) return x|(y<<9);
static const uint b6=b0<<6;
z =  (xx&b6)>>6 | ((yy&b6)>>3) | ((zz&b6));
//binary ...11000000000000000000000111111111
static const uint C0=0777;
static const uint c0=(((C0<<27)|C0)<<27)|C0;
xx =  (x&c0) | ((y&c0)<<9) | ((z&c0)<<18);
if( (xyz>>9)==0 ) return xx;
static const uint c9=c0<<9;
yy =  (x&c9)>>9 | ((y&c9)) | ((z&c9)<<9);
if( (xyz>>18)==0 ) return xx | (yy<<27);
static const uint c18=c0<<18;
zz =  ((x&c18)>>18) | ((y&c18)>>9) | ((z&c18));
//binary ...00111111111111111111111111111111
static const uint d0=0777777777;
if((xyz>>27)==0 ) return xx | (yy<<27) | (zz<<54);
else return ~0Lu;
}
```

Table 4.1: z-index computation in c++. $m_i^{(k)}$ from eq.4.6 is here denoted as {a,b,c}{0,3,6,9,18,27} , $b_i^{(k)}$ from eq.4.7 denoted alternatingly as x,y,z and xx,yy,zz

| Name | dino | chicken | handy | box2 | box | plug | pig |
|---|---|---|---|---|---|---|---|
| # points | 1.27M | 0.87M | 0.58M | 0.66M | 1.20M | 1.59M | 3.84M |
| Insertion | 3.2s | 3.8s | 1.9s | 2.0s | 2.3s | 4.6s | 8.4s |
| # Invariant sets | 12485 | 14889 | 8815 | 8887 | 9445 | 22757 | 49796 |
| Recognition | 0.65s | 0.44s | 0.52s | 0.75s | 0.67s | 0.53s | 1.14s |
| Noise level 1 | 0.77s | 0.50s | 0.47s | 0.65s | 0.69s | 0.72s | 1.22s |
| Noise level 7 | 0.80s | 0.52s | 0.55s | 0.77s | 0.78s | 0.77s | 1.42s |

Table 4.2: Times for Object recognition

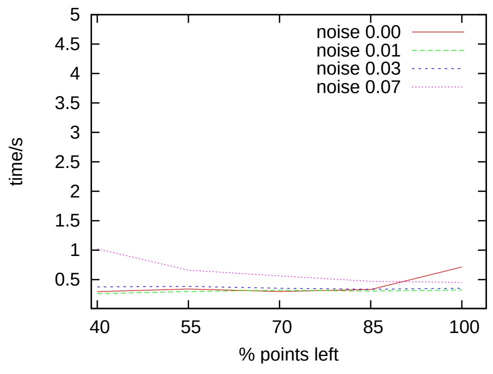
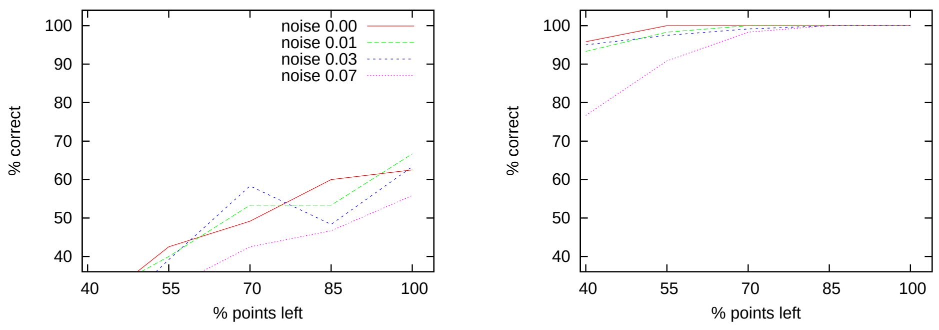
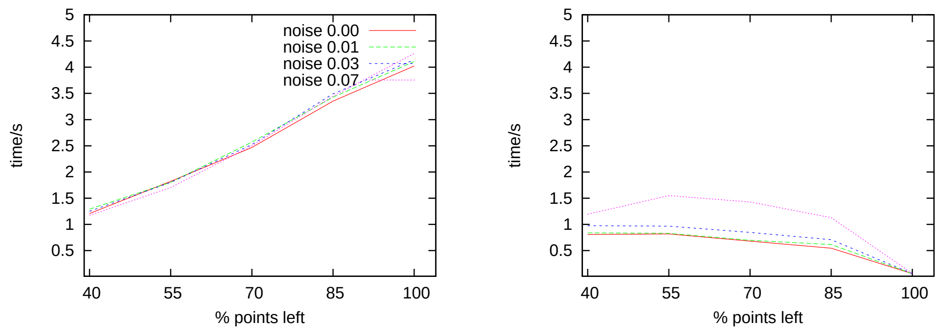


Figure 4.11: Database DENKER: Time under noise for the dino



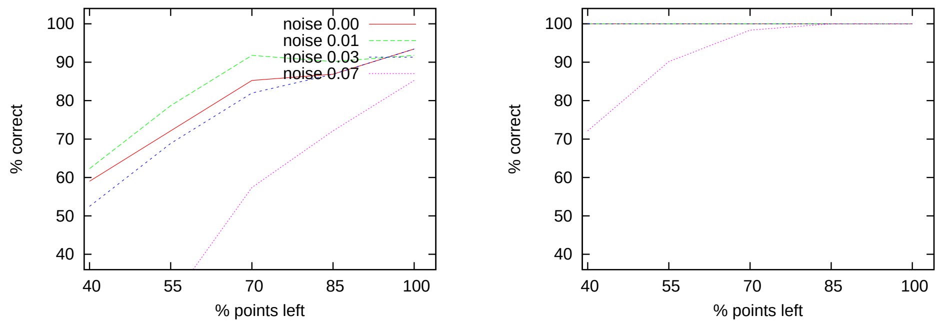
different version same version

Figure 4.12: Database 3DRMA: Accurracy



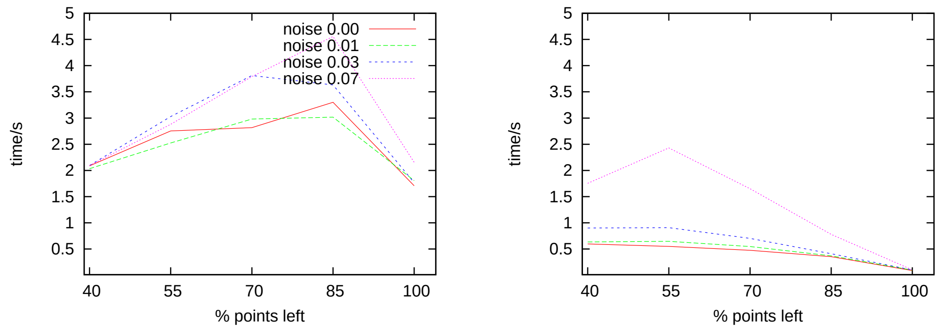
different version same version

Figure 4.13: Database 3DRMA: time used for recognition



different version same version

Figure 4.14: Accurracy Database GAVAB



different version same version

Figure 4.15: Database GAVAB: time used for recognition

# Bibliography

[1] Charles Beumier. http://www.sic.rma.ac.be/ beumier/db/3d_rma.html, 2008.

[2] I.N. Bronstein, K.A. Semendjajew, Günter Grosche, and Eberhard Zeidler. *Teubner-Taschenbuch der Mathematik*, volume I. Teubner Verlagsgesellschaft, 1996. ISBN 3-8154-2001-6.

[3] Janis Fehr, Marco Reisert, and Hans Burkhardt. Cross-correlation and rotation estimation of local 3d vector field patches. In *Proceedings of the 5th International Symposium on Advances in Visual Computing: Part I*, ISVC '09, pages 287–296, Berlin, Heidelberg, 2009. Springer-Verlag.

[4] Janis Fehr, Alexander Streicher, and Hans Burkhardt. A bag of features approach for 3d shape retrieval. In *Proceedings of the 5th International Symposium on Advances in Visual Computing: Part I*, ISVC '09, pages 34–43, Berlin, Heidelberg, 2009. Springer-Verlag.

[5] 1925-1991 Greub, Werner Hildbert. *Multilinear algebra*. Springer, New York [u.a.], 2nd edition, 1978.

[6] G. Guy and G. Medioni. Inference of surfaces, 3d curves, and junctions from sparse, noisy, 3d data. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 19(11):1265 –1277, nov 1997.

[7] D. Keren. Using symbolic computation to find algebraic invariants. *IEEE Trans. Pattern Anal. Mach. Intell.*, 16(11):1143–1149, 1994.

[8] Max Langbein and Hans Hagen. A generalization of moment invariants on 2d vector fields to tensor fields of arbitrary order and dimension. In George Bebis, Richard D. Boyle, Bahram Parvin, Darko Koracin, Yoshinori Kuno, Junxian Wang, Renato Pajarola, Peter Lindstrom, André Hinkenjann, Miguel L. Encarnação, Cláudio T. Silva, and Daniel S. Coming, editors, *ISVC (2)*, volume 5876 of *Lecture Notes in Computer Science*, pages 1151–1160. Springer, 2009.

[9] Max Langbein, Gerik Scheuermann, and Xavier Tricoche. An efficient point location method for visualization in large unstructured grids. In *Proc. 8th Int. Worksh. Vision, Modeling, and Visualization*, pages 27–35, 2003.

[10] Evgeni Magid, Octavian Soldea, and Ehud Rivlin. A comparison of gaussian and mean curvature estimation methods on triangular meshes of range image data. *Computer Vision and Image Understanding*, 107(3):139 – 159, 2007.

[11] Gérard Medioni, Chi-Keung Tang, and Mi-Suen Lee. Tensor voting: Theory and applications. *Proceedings of RFIA, Paris, France*, 2000.

[12] A.B. Moreno and A.Sanchez. Gavabdb: A 3d face database. In *2nd COST Workshop on Biometrics on the Internet: Fundamentals, Advances and Applications, C. Garcia et al (eds): Proc. 2nd COST Workshop on Biometrics on the Internet: Fundamentals, Advances and Applications, Ed. Univ. Vigo*, pages 77–82, 2004.

[13] Evren Özarslan, Baba C. Vemuri, and Thomas H. Mareci. Generalized scalar measures for diffusion mri using trace, variance, and entropy. *Magnetic resonance in Medicine*, 53:866–867, 2005.

[14] Roger Penrose. *The Road to Reality.* Vintage Books, 2004. ISBN 978-0-679-77631-4.

[15] Helmut Pottmann, Johannes Wallner, Qi-Xin Huang, and Yong-Liang Yang. Integral invariants for robust geometry processing. *Computer Aided Geometric Design*, 26:37–59, 2009.

[16] Michael Schlemmer, Manuel Heringer, Florian Morr, Ingrid Hotz, Martin Hering-Bertram, Christoph Garth, Wolfgang Kollmann, Bernd Hamann, and Hans Hagen. Moment invariants for the analysis of 2d flow fields. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1743–1750, 2007.

[17] Thomas Schultz, Joachim Weickert, and Hans-Peter Seidel. A higher-order structure tensor. Research Report MPI-I-2007-4-005, Max-Planck-Institut für Informatik, Stuhlsatzenhausweg 85, 66123 Saarbrücken, Germany, July 2007.

[18] Chi-Keung Tang and G. Medioni. Inference of integrated surface, curve and junction descriptions from sparse 3d data. *Pattern Analysis and*

*Machine Intelligence, IEEE Transactions on*, 20(11):1206 –1223, nov 1998.

[19] Chi-Keung Tang and G. Medioni. Robust estimation of curvature information from noisy 3d data for shape description. In *Computer Vision, 1999. The Proceedings of the Seventh IEEE International Conference on*, volume 1, pages 426 –433 vol.1, 1999.

[20] Wai-Shun Tong and Chi-Keung Tang. Robust estimation of adaptive tensors of curvature by tensor voting. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 27(3):434–449, 2005.

# List of Figures

# List of Tables

# Tools used

The following tools have been used:

- MS Visual Studio for C++ source code editing and compiling
- Postscript (together with C++) to create the hypergraphs and the statistics' images
- osgviewer (OpenSceneGraph) to create 3D visualizations
- Inkscape to edit the hypergraphs and create small explanation figures
- Maple for integrals
- Gnuplot to create diagrams
- LaTeX to create this whole document

# Own Publications

- M. Langbein, I. Scheler, A. Ebert, and H. Hagen. *B:3D - visualize land-use plans interactively.* In *Proceedings of the The Eurographics Conference on Visualization, Leipzig, Germany*, 2013.

- Max Langbein and Hans Hagen. *A generalization of moment invariants on 2d vector fields to tensor fields of arbitrary order and dimension.* In George Bebis, Richard D. Boyle, Bahram Parvin, Darko Koracin, Yoshinori Kuno, Junxian Wang, Renato Pajarola, Peter Lindstrom, André Hinkenjann, Miguel L. Encarnação, Cláudio T. Silva, and Daniel S. Coming, editors, *ISVC (2)*, volume 5876 of *Lecture Notes in Computer Science*, pages 1151–1160. Springer, 2009.

- Max Langbein, Gerik Scheuermann, and Xavier Tricoche. *An efficient point location method for visualization in large unstructured grids.* In *Proc. 8th Int. Worksh. Vision, Modeling, and Visualization*, pages 27–35, 2003.

- Peter-Scott Olech, Ariane Middel, Max Langbein, Sebastian Thelen, Achim Ebert, Jörg Meyer, and Hans Hagen. *Enhancing the planner's toolkit - new display technologies for planning support.* In *International Urban Planning and Environment Association (IUPEA) 8th International Symposium (UPE8), Kaiserslautern, Germany*, 2009.

## To be published:

- Ragaad AlTarawneh, Max Langbein, Shah Rukh Humayoun, and Hans Hagen. *TopoLayout-DG: A topological feature-based framework for visualizing inside behaviour of large directed graphs.* In E. Sintorn M. Obaid, D. Sjölie and M. Fjeld, editors, *SIGRAD*, 2014. Won Best Student Paper Award.

## Possible publication, under review:

(a short version of chapters 3 and 4:)

- Max Langbein and Hans Hagen. *Visualization and Processing of Higher Order Descriptors for Multi-Valued Data*, chapter *Moment Invariants for Pattern Recognition.* Mathematics and Visualization. Springer.

# Curriculum vitae

| | |
|---|---|
| 1998-2007 | Computer Science with minor physics<br>at the University of Kaiserslautern |
| 2003 | *Projektarbeit* and publication [9] titled<br>"A Memory-Efficient Point Location Method<br>for the In-core Visualization of Tensor Fields on<br>Unstructured Finite Element Meshes"<br>at the University of Kaiserslautern |
| 2004 | *Diplomarbeit* titled<br>"Entwurf und Implementierung einer verteilten<br>out-of core Datenstruktur für die Visualisierung<br>großer unstrukturierter Datensätze"<br>at the University of Kaiserslautern |
| 2007 | Diploma degree in Computer Science<br>at the University of Kaiserslautern |
| 2007- | Doctoral studies at TU Kaiserslautern<br>Cooperation in various industry projects of TU KL |