Ludwig-Maximilians-Universität München
Mathematisches Institut

Diplomarbeit

Approximation of probability density functions on the Euclidean group parametrized by dual quaternions

Autorin: Muriel Lang
Themenstellung und Betreuung:
Dr. Vitali Wachtel, Dr. Wendelin Feiten

Eingereicht am 25.01.2011

Abstract

Perception is fundamental to many robot application areas especially in service robotics. Our aim is to perceive and model an unprepared kitchen scenario with many objects. We start with the perception of a single target object. The modeling relies especially on fusing and merging of weak information from the sensors of the robot in order to localize objects. This requires the representation of various probability distributions of pose in $S_3 \times \mathbb{R}^3$ as orientation and position have to be localized. In this thesis I present a framework for probabilistic modeling of poses in $S_3 \times \mathbb{R}^3$ that represents a large class of probability distributions and provides among others the operations of the fusion and the merge of estimates. Further it offers the propagation of uncertain information data. I work out why we choose to represent the orientation part of a pose by a unit quaternion. The translation part is described either by a 3-dimensional vector or by a purely imaginary quaternion. This depends on whether we define the probability density function or whether we want to represent a transformation which consists of a rotation and a translation by a dual quaternion. A basic probability density function over the poses is defined by a tangent point on the hypersphere and a 6-dimensional Gaussian distribution. The hypersphere is embedded to the $\mathbb{R}^4$ which is representing a unit quaternions whereas the Gaussian is defined over the product of the tangent space of the sphere and of the space of translations. The projection of this Gaussian to the hypersphere induces a distribution over poses in $S_3 \times \mathbb{R}^3$. The set of mixtures of projected Gaussians can approximate the probability density functions that arise in our application. Moreover it is closed under the operations introduced in this framework and allows for an efficient implementation.

Acknowledgments

Especially I want to thank the participants of the "Diplomandenseminar" for the helpful remarks they gave to me throughout the whole period I was at Siemens and to Thilo Grundmann for the pictures.
Further thanks to Dr. Wendelin Feiten, Dr. Gisbert Lawitzky and Dr. Vitali Wachtel for the good mentoring and to Darren Murph for allowing me to use the pictures of the web page www.Engadget.com.

My family, Judith Gampe, Thomas Haugg, Florian Schäfer, Simeon Schmied and all the others that were so patient with me I want to thank for their support.

Contents

1. Introduction

1.1. Motivation

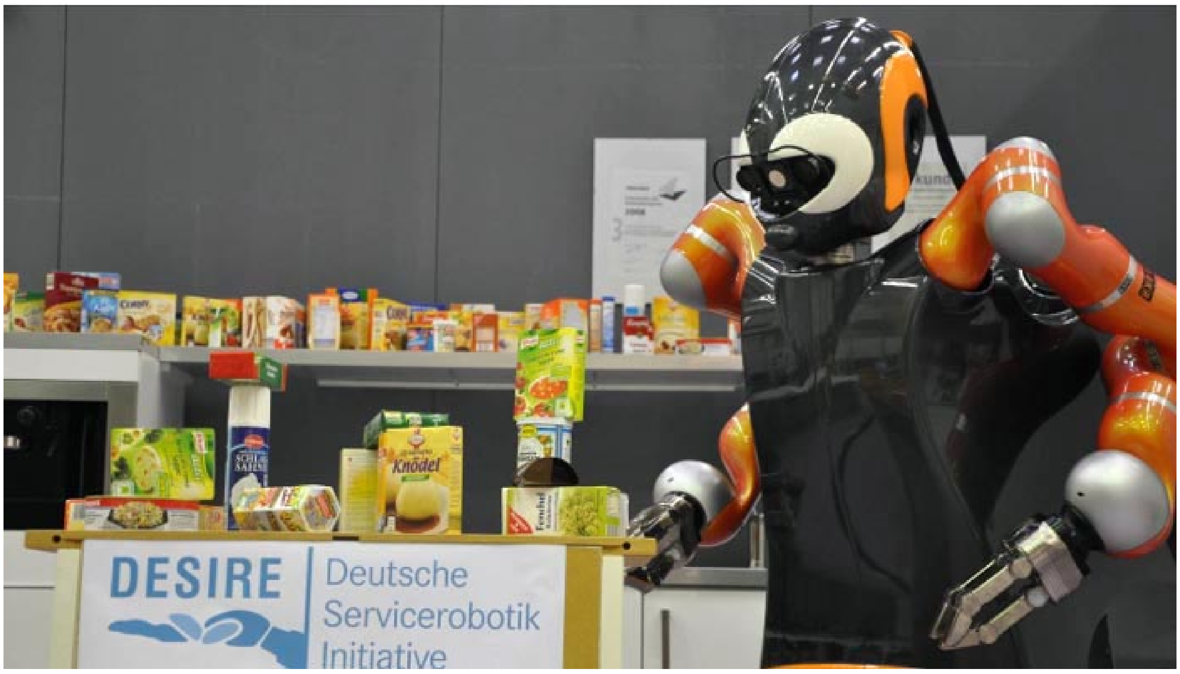


Figure 1.1.: The figure shows the robot of the DESIRE project (which stands for 'Deutsche Servicerobotic Initiative') observing a kitchen scenario.

Imagine a robot that has the task to get a specified object from a table. Several objects are arranged on the table like in an arbitrary kitchen scenario. This kind of task is fundamental for many applications of service robotics like mobile manipulation.
The robot knows the pose of the table and notices there is something on top of it. Height and position of the table usually are given to the robot to receive better results in experiments. For the algorithm it is not necessary to know the pose of the table. It could also be estimated. That something is placed on top of it, the robot can see through its stereo camera system. Often a 3D sensor like a laser scanner or ultra sonic sound sensor also is part of the robot's sensor system.

Robots are not able to identify unknown objects through the process established in our framework. They just can recognize known objects. Therefore a database of 3D models of objects is implemented to the robot. The robot makes several independent attempts to determine orientation and position of the target object exact enough to be able to grasp it. These localization attempts might be with different methods, from different points of view or under different conditions like brightness and shadows for instance. Thus we get plenty of more or less suitable information data which the robot needs to handle in an appropriate way. The robot repeats making independent attempts to localize the object until the resulting pose is determined *well enough* i.e. the uncertainty is below a given threshold and the robot can pick up the object with a failure probability below a resulting value $\varepsilon > 0$.

In this context rotation and orientation are often used synonymously as well as translation and position. This comes from the fact that the orientation of any object results from the rotation that moves it to this alignment and likewise the position results from the translation. Together orientation and position describe the **pose** of an object as a pose is the result of any transformation consisting of rotation and translation.
Furthermore I won't handle object classes in this work. I suppose that the objects are part of the robot's database and that it recognizes them.

1.2. Problem Statement

For the whole present work I assume that the data association problem is solved. This means that we know which object the data describes we receive from the individual localization attempts. I claim this to reduce the uncertainty and thus to simplify the situation we have to deal with. We work with a single target object and its observation data.
After having reduced the complexity of the task that far we have to estimate the state of the target object. As in a kitchen scenario like illustrated in figure 1.1, three dimensions for each, the rotation and translation can occur, we have to deal with a 6-dimensional space, namely the special Euclidean group, which will be introduced in 2.1.2. Such a state of the target object in the special Euclidean group is called object pose and thus I will refer to pose estimation instead of state estimation.
To be able to estimate a pose at first the representation of the pose and parametrization of rotation and translation have to be defined. In section 2.1 I introduce several can-

didates for the pose representation and then select the most suitable one for our topic. After analyzing the problem from an algebraic point of view some probability theory has to be introduced. To estimate the pose of a target object, its probability distribution has to be known. Section 2.2 is concerned with some distribution functions on the special Euclidean group which can either have a parameterized density or consist of a particle set. Finally I justify in section 2.3 my decision to choose mixtures of projected Gaussians as preferable distribution function to estimate the pose of the target object and introduce the algorithms for basic operations. One of them is the fusing of two mixtures i.e. deriving a common estimate from two independent estimates, another one is to merge components in a mixture.
In chapter 3 which is the body of this work I handle approximations of mixtures of projected Gaussians. In detail I give an upper bound for the approximation error that occurs on dropping summands of a mixture in section 3.2.1. Further I explain the difficulties that arise on applying the operations to fuse and to merge to mixtures of projected Gaussians. If different information data shall be applied at the same time a weighting factor has to be introduced to evaluate the compatibility of the single mixture elements. Section 3.2.2 is concerned with that whereas 3.2.3 is concerned with the merge of a mixture of projected Gaussians. Merging a mixture means reducing the number of summands through iteratively merging the two least dissimilar summands of the mixture. Therefore a kind of dissimilarity measure basing on an appropriate distance measure has to be defined.
I pull the analysis of distance and convergence measures out to the beginning of chapter 3 as I need it for severals aspects of approximations. It is desired to keep the exactness that the robot is able to grasp the target object with a high probability despite any approximation of the mixture. To study the accuracy of the approximation of a pose estimation I examine the failure probability that occurs on trying to grasp an object. Thus the introduction of a grasp criterion given in section 3.1 is required. The criterion also base on distance measures between the pose of the gripper and the estimated pose of the target object.
We require from the approximation of a mixture of projected Gaussians to optimize the necessary computational effort with minimized loss of accuracy of the pose estimation. It is known that the mixture describing the pose estimation after evaluating more and more data from the localization attempts converges to the mixture describing the true object's pose. I study further convergence criteria that hold for approximations of mixtures of projected Gaussians and examine the properties that are passed on from the original mixture to the approximated one in section 3.2.

The last section 3.3 of this chapter is concerned with approximation algorithms. Given a mixture of projected Gaussian that describes the probability distribution of the pose of a target object, but might be complicated to calculate or might contain parameters which are unknown to us. Then a second mixture using a reduced number of base elements should be fitted to the first mixture. I explain the expectation maximization algorithm 3.3.1 and the Monte Carlo algorithm 3.3.2 which are two different ways how this fit can be done.
The practical application of the results from chapter 3 is introduced in chapter 4. As already mentioned the robot makes several localization attempts to determine the pose of the target object. Then it estimates the uncertainty of each of these tries and fuses and merges the weighted measurements according to an evaluation algorithm. With this evaluation algorithm the uncertainty shall be reduced until the distribution of the estimated pose is peaked enough so that the grasp criterion is fulfilled with failure probability smaller that a given threshold. In chapter 4 I explain how the robot draws conclusions from single 3D points it detects on the surface of the object, about the pose of the target object itself where the 3D points are so-called point features. The only features I treat in the framework are SIFT features what stands for scale invariant feature transform, even though the framework also would allow the handling of other features like edge detection for instance. In section 4.1 I give a documentation of the code I wrote to program a framework for the precise estimation of an object's pose and in section 4.2 I demonstrate how the framework works. In the outlook 5 a picture 5.1 shows the result that can be achieved by a similar approach of pose estimation to the one in this work even though I also have to remark that some aspects are left over to be treated in future work.

1.3. Related Work

The representation of rigid motions and especially of orientation in three dimensions is a central issue in various disciplines of arts, science and engineering. Rotation matrix, Euler angles, Rodrigues vector and unit quaternions are the most popular representations of a rotation in three dimensions. Rotation matrices have many parameters, Euler angles are not invariant under transforms and have singularities and Rodrigues vectors do not allow for an easy composition algorithm. Stuelpnagel [29] points out that unit quaternions are a suitable representation of rotations on the hypersphere S_3 with few parameters, but does not provide probability distributions. Choe [5] represents the probability distribution of rotations via a projected Gaussian on a tangent space. He only deals with concentrated distributions and does not take translations into account. Goddard and Abidi [12, 11] use dual quaternions for motion tracking. In their observations the correlation between rotation and translation is captured also. The probability distribution over the parameters of the state model is a unimodal normal distribution. If the initial estimate is sufficiently certain and if the information that shall be fused to the estimate is sufficiently well focused this is an appropriate model. As can be seen in [16] from Kavan et al. dual quaternions provide a closed form for the composition of rigid motions, similar to the transform matrix in homogeneous coordinates. Antone [30] suggests to use the Bingham distribution in order to represent weak information even though he does not give a practical algorithm for fusion of information or propagation of uncertain information. By now it is known that propagated uncertain information only can be approximated by Bingham distributions. Further Love [22] states that the renormalization of the Bingham distribution is computationally expensive. Glover [13] also works with a mixture of Bingham distributions and recommends to create a precomputed lookup table of approximations of the normalizing constant using standard floating point arithmetic. Mardia et al. [23] use a mixture of bivariate von Mises distributions. They fit the mixture model to a data set using the expectation maximization algorithm because this allows for modeling widely spread distributions. Translations are not treated by them. To propagate the covariance matrix of a random variable through a nonlinear function, the Jacobian matrix is used in general. Kraft et al. [18] use therefore an unscented Kalman Filter [15]. This technique would have to be extended to the mixture distributions.

2. Pose Estimation

In robotic perception the six dimensional pose of the object of interest has to be estimated. The input information is weak as it comes from imperfect sensors. These uncertainties arise from not exactly tuned 3D sensors or cameras and from the joints in the robot's body. Moreover the three dimensional models which the robot uses to recognize the objects are not perfect. Altogether one has to deal with uncertain information which is modeled by widely spread probability density functions ($pdfs$) to estimate position and orientation in the six dimensional space. In order to be able to represent and process this weak information, the density functions are formulated on the bases of suitable parametric representations of the pose.

2.1. Pose Representation

2.1.1. Representation of Orientation

The more critical part in the pose representation is the rotation. There are several requirements concerning the parametrization of the rotation which we wish to be fulfilled the best.

- For each position there should be just one representation to avoid wrong choice of representation.

- To minimize the computational effort we desire the rotation to be represented by few parameters. If a representation uses more than the minimal number of three parameters, some additional condition needs to be satisfied to reduce the number of independent parameters to three. After calculations or estimations of parameters these conditions may have to be re-established in the *intuitively best possible way* although we do not formally define what this is. We call it 'renormalization' and we desire this step to be easy to perform.

- There should be an easy way to derive the parameters of the composed rotation from the parameters of two input rotations of the composition. The composability of rigid motions ins needed for instance if sensor data taken from two different sensor system poses shall be fused in a common coordinate system.
- We wish the rotation to be a differentiable function of the parameters to assure smoothness. At least it should be continuous.
- Finally a desired characteristic of the parameterization is to be area and distance preserving under rotation and translation. This is important when we deal with probability density functions over rigid motions.

A short overview about the parameterizations of orientations is given in [28].

Rotation Matrices

These are the probably most wide spread representations, especially in the homogeneous coordinate formulation.
A rotation matrix R is constrained to be orthogonal $R^\top \cdot R = R \cdot R^\top = I$ and the restriction $\det(R) = 1$ is to avoid it to be a reflection.
The set of all rotation matrices forms a group, known as the rotation group or the special orthogonal group $SO(n)$. It is a subset of the orthogonal group $O(n)$, which includes reflections and consists of all orthogonal matrices with determinant 1 or -1.

In $\mathbb{R}^3$ the matrix for a rotation by an angle θ about the axis v where $v = (x, y, z)^\top$ a unit vector, is given by:

$$R = \begin{bmatrix} \cos\theta + x^2(1-\cos\theta) & xy(1-\cos\theta) - z\sin\theta & xz(1-\cos\theta) + y\sin\theta \\ yx(1-\cos\theta) + z\sin\theta & \cos\theta + y^2(1-\cos\theta) & yz(1-\cos\theta) - x\sin\theta \\ zx(1-\cos\theta) - y\sin\theta & zy(1-\cos\theta) + x\sin\theta & \cos\theta + z^2(1-\cos\theta) \end{bmatrix}$$

What characteristics do rotation matrices have?

- The representation is unique.
- It is well known and there are wide spread applications.
- It consists of too many parameters. The constraints arise from the need of nine values for the rotation matrix to represent three independent variables of a 3D rotation.

- The renormalization is difficult.
- The rotation matrix is differentiable with resect to its parameters and preserves area and distance.

Euler Angles

The Euler angles are three angels Ψ, Θ and Φ which describe rotations around specified axes in $\mathbb{R}^3$ usually. Together they define a transformation between two coordinate systems. There are different possibilities to choose the rotation axes. One of the most common ones is the following convention:

1. Rotate the coordinate system at first about Ψ around the z-axis. Then new coordinate axes x' and y' are obtained.
2. Then rotate it about the angel Θ around the new x-axis x'. The new coordinate axes which are obtained after the rotation are denoted with y'' and z''.
3. Finally rotate the system about Φ around the new z-axis z''.

Thus the whole rotation is obtained by the rotation matrix $R_{zx'z''}$:

$$R_{zx'z''} = \begin{pmatrix} \cos\Psi & -\sin\Psi & 0 \\ \sin\Psi & \cos\Psi & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\Theta & -\sin\Theta \\ 0 & \sin\Theta & \cos\Theta \end{pmatrix} \begin{pmatrix} \cos\Phi & -\sin\Phi & 0 \\ \sin\Phi & \cos\Phi & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

$$= \begin{pmatrix} \cos\Psi\cos\Phi - \sin\Psi\cos\Theta\sin\Phi & -\cos\Psi\sin\Phi - \sin\Psi\cos\Theta\cos\Phi & \sin\Psi\sin\Theta \\ \sin\Psi\cos\Phi + \cos\Psi\cos\Theta\sin\Phi & \cos\Psi\cos\Theta\cos\Phi - \sin\Psi\sin\Phi & -\cos\Psi\sin\Theta \\ \sin\Theta\sin\Phi & \sin\Theta\cos\Phi & \cos\Theta \end{pmatrix}$$

Euler angles and translation vector are natural for robotics, for instance where the angles are the motor positions like in the wrist, and common for representation of small angle ranges of the $SO(3)$ like for cars and ships. But there are twelve different choices for the rotation axes which is much room for confusion.

Characteristics of the Euler angles:

- Minimal number of parameters, namely three.

- The composition is not straight forward and there is no easy algorithm for finding the Euler angles of a composition given the Euler angles of two individual rotations.
- The parameterization is periodic with 2π and dependent on the choice of axes, moreover the Euler angles are not invariant under transformations and have singularities.
- Gimbal Lock (loss of one degree of freedom in a 3D space that occurs when the axes of two of the three gimbals are driven into a parallel configuration)

Rodrigues Vector

Rodrigues found the quaternions three years before Hamilton and derived the Rodrigues rotation formula from them. This rotation formula describes an algorithm to rotate a vector in the 3-dimensional Euclidean space, given an axis $\hat{\mathbf{e}}$ and angle θ of rotation. The paper he wrote appeared in *Annales de Gergonne* in 1840 [26].

Definition 2.1.1. *Let $v = (v_1, v_2, v_3)^\top$ be a vector in $\mathbb{R}^3$ and $\hat{\mathbf{e}} = (e_x\ e_y\ e_z)^\top$ the 3D unit vector describing an axis of rotation about which we want to rotate v by the angle θ.*
The Rodrigues formula is defined as:

$$v_{\text{rot}} = v\cos\theta + (\hat{\mathbf{e}} \times v)\sin\theta + \hat{\mathbf{e}}(\hat{\mathbf{e}} \cdot v)(1 - \cos\theta)$$

And in matrix notation:

$$v_{\text{rot}} = v\cos\theta + \begin{pmatrix} 0 & -e_z & e_y \\ e_z & 0 & -e_x \\ -e_y & e_x & 0 \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \\ v_3 \end{pmatrix} \sin\theta + \hat{\mathbf{e}} \cdot \hat{\mathbf{e}}^\top \cdot v(1 - \cos\theta)$$

In the definition $\hat{\mathbf{e}} \times v$ means the cross product of the two vectors $\hat{\mathbf{e}}$ and v. It is defined as $(|\hat{\mathbf{e}}| \cdot |v| \sin\alpha) \cdot n$ where $0 \leq \alpha \leq 180°$ is the smallest angle between the vectors and n is the normal vector of the plane containing $\hat{\mathbf{e}}$ and v.

Rodrigues parameters can be expressed in terms of Euler axis and angle as $u = \hat{\mathbf{e}} \cdot \tan(\theta/2)$, a non-normalized 3D vector. The direction of u specifies the axis, and its magnitude is $\tan(\theta/2)$. Thus the angle θ of rotation is given by $\|u\| = \tan(\theta/2)$.

Since $\hat{\mathbf{e}} \in S_3$ and $-\hat{\mathbf{e}} \in S_3$ define the same rotation, each orientation is uniquely determined by a point on the unit hemisphere of S_3.

Characteristics of rotation by Rodrigues vectors:

- They have the minimal number of three parameters.
- There is no or at least no easy composition algorithm.
- The parametrization is periodic with 2π and computations are not efficient.

Unit Quaternions

Quaternions are a generalization of the complex numbers to the $\mathbb{R}^4$ as can be seen in [33]. Instead of one imaginary unit, they have three, i, j and k. With real coefficients a, b, c, d a quaternion q is defined as

$$q := a + i \cdot b + j \cdot c + k \cdot d$$

or $[a, b, c, d]$ in vector notation.

Definition 2.1.2. *The set $\mathbb{H} := \{q = a + i \cdot b + j \cdot c + k \cdot d : a,\ b,\ c,\ d \in \mathbb{R}\}$, named after Hamilton, is the* ***skew field of quaternions*** *with component wise summation and quaternionic multiplication which are defined in the following and the neutral elements:*

$$0 = 0 + i \cdot 0 + j \cdot 0 + k \cdot 0 \quad \text{and} \quad 1 = 1 + i \cdot 0 + j \cdot 0 + k \cdot 0$$

Further it has the properties:

$$\begin{aligned} i \cdot j &= k \\ j \cdot k &= i \\ k \cdot i &= j \\ i \cdot j \cdot k &= i^2 = j^2 = k^2 = -1 \end{aligned}$$

Component wise summation of quaternions:

$$\begin{aligned} q_1 + q_2 &= (a + i \cdot b + j \cdot c + k \cdot d) + (e + i \cdot f + j \cdot g + k \cdot h) \\ &= a + e + i \cdot (b + f) + j \cdot (c + g) + k \cdot (d + h) \end{aligned}$$

Multiplication of a quaternion with a scalar $\lambda \in \mathbb{R}$ is defined as:

$$\begin{aligned} \lambda \cdot q &= \lambda \cdot (a + i \cdot b + j \cdot c + k \cdot d) \\ &= \lambda \cdot a + \lambda \cdot j \cdot b + \lambda \cdot j \cdot c + \lambda \cdot k \cdot d \end{aligned}$$

Quaternionic multiplication follows from the properties of imaginary units:

$$\begin{aligned} q_1 * q_2 &= (a + i \cdot b + j \cdot c + k \cdot d) * (e + i \cdot f + j \cdot g + k \cdot h) \\ &= (ae - bf - cg - dh) + i \cdot (af + be + ch - dg) \\ &+ j \cdot (ag - bh + ce + df) + k \cdot (ah + bg - cf + de) \end{aligned}$$

For the multiplication of quaternions the associative and the distributive law hold, but not the commutative law. Easy calculations show that $\mathbb{H}$ is an associative and not commutative algebra see [3] and 1 is the identity element of $\mathbb{H}$.
Like for complex numbers, there is a conjugate quaternion: $\overline{q} := a - i \cdot b - j \cdot c - k \cdot d$ and the norm of a quaternion is the 2-norm in $\mathbb{R}^4$: $\|q\| = \sqrt{a^2 + b^2 + c^2 + d^2} = \sqrt{q * \overline{q}}$. Every non-zero quaternion has an inverse: $q^{-1} = \frac{1}{(\|q\|)^2}\overline{q}$ and for any two quaternions q_1 and q_2 we have the formula $\overline{(q_1 * q_2)} = \overline{q_2} * \overline{q_1}$.

We can restrict the quaternion to be imaginary what means without real part:

Definition 2.1.3. *The set* $\mathbb{H}_{Im} := \{q = 0 + i \cdot b + j \cdot c + k \cdot d : b, c, d \in \mathbb{R}\}$ *is called the set of imaginary quaternions.*

The 3-sphere $S_3 \subset \mathbb{H}$ in quaternionic calculus is similar to the unit circle $S_1 \subset \mathbb{C}$ in complex calculus. In fact:

$$S_3 = \{q \in \mathbb{H} : \|q\| = 1\}$$

$\|q\| = 1$ defines exactly the **unit quaternions** which obey the following constraint:

$$a^2 + b^2 + c^2 + d^2 = 1$$

In terms of the Euler axis $\hat{\mathbf{e}} = [e_x \; e_y \; e_z]^\top$ and angle θ the elements of the quaternion in vector notion can be expressed as follows:

$$\begin{aligned} a &= \cos(\theta/2) \\ b &= e_x \sin(\theta/2) \\ c &= e_y \sin(\theta/2) \\ d &= e_z \sin(\theta/2) \end{aligned}$$

Thus the rotation can be represented with the quaternion $q = [\cos(\frac{\theta}{2}), \hat{\mathbf{e}} \cdot \sin(\frac{\theta}{2})]$. To rotate a point in $\mathbb{R}^3$ we identify it with the quaternion $p = [0, b, c, d]$. Then the rotation about q can be applied as:

$$q * p * \overline{q}$$

The composition of rotations q_1 and q_2 easily is the product of them:

$$q_1 * (q_2 * p * \overline{q_2}) * \overline{q_1} = (q_1 * q_2) * p * (\overline{q_1 * q_2})$$

From $q * p * \overline{q} = (-q) * p * \overline{(-q)}$ the antipodal symmetry of the quaternions can be seen.

Comparing the performance of rotation with quaternions and matrices gives the following result:
Rotating a point, is easier done by matrix operation than with quaternions, but chaining operations like composition of rotations is less costly with quaternions. Matrix multiplication needs 27 operations in opposition to quaternion multiplication that just needs 16. For summation and subtraction 18 operation for matrices and just 12 for quaternions are needed. Thus the computation with quaternions is less expensive.

In summary the characteristics of rotation representation by unit quaternions are:

- Unit quaternions have few parameters, four instead of the minimal number three.
- They are unique except for antipodal symmetry.
- The rotation from one state to another on the great circle gives a smooth movement avoiding unnatural angular moves that occur for Euler angles.
- Computations are highly efficient.
- They are easy to deal with.

Pluecker line coordinates and Complex mapping

These are rotation representations which I just want to name for completeness. They don't fulfill the desired requirements and thus I won't handle them any further.

2.1.2. Pose

Pose representation is equivalent to representation of rigid motion. The group of rigid motions in $\mathbb{R}^n$ is sometimes called **special Euclidean group** $SE(n)$.

Euclidean Group

A transformation is said to be rigid if it preserves **relative distances** what means it is angle and distance preserving.

- The composition of rigid transformations is rigid.
- The inverse of a rigid transformations is rigid.

The subgroup of rigid transformations which additionally is orientation preserving, is called the **group of rigid motions** and just contains rotations and translations.

Definition 2.1.4. *$E(n) := \mathbb{R}^n \times O(n)$ is the n-dimensional Euclidean Group, where $O(n)$ is the n-dimensional orthogonal group.*

$E(n)$ is the symmetry group of n-dimensional Euclidean space and consists of bijective, distance and angle preserving affine transformations.

Definition 2.1.5. *$SE(n) := \mathbb{R}^n \times SO(n)$ is the n-dimensional special Euclidean Group, where $SO(n)$ is the n-dimensional special orthogonal group and S_n is the n-sphere.*

$SE(n)$ is a subgroup of $E(n)$ which just contains direct isometries. Direct means orientation preserving. $SE(n)$ is also called the subgroup of rigid motions.

The 3-dimensional special Euclidean Group $SE(3) = \mathbb{R}^3 \times SO(3)$ thus represents all possible rotations and translations in the three dimensional Euclidean space.

Dual Quaternions

Definition 2.1.6. *The ring of the dual quaternions with the dual unit ϵ, which has the property $\epsilon^2 = 0$, is defined as:*

$$\mathbb{H}_D = \{dq \mid dq = q_1 + \epsilon \cdot q_2 \,\&\, q_1, q_2 \in \mathbb{H}\}$$

This ring can also be written as $\{a_D + i\cdot b_D + j\cdot c_D + k\cdot d_D : a_D,\ b_D,\ c_D,\ d_D$ dual numbers$\}$ see [3]. The dual numbers $\{a_D, b_D, c_D, d_D\}$ are called components of a dual quaternion then. Just as the quaternions, the dual quaternions have the basis $\{1, i, j, k\}$ of the 4-dimensional linear space over the dual numbers.

Summation of dual quaternions is component wise:

$$\begin{aligned} dq_1 + dq_2 &= (q_{1,1} + \epsilon \cdot q_{1,2}) + (q_{2,1} + \epsilon \cdot q_{2,2}) \\ &= (q_{1,1} + q_{2,1}) + \epsilon \cdot (q_{1,2} + q_{2,2}) \end{aligned}$$

Multiplication of a dual quaternion with a scalar $\lambda \in \mathbb{R}$ is defined as:

$$\begin{aligned} \lambda \cdot dq &= \lambda \cdot (q_1 + \epsilon \cdot q_2) \\ &= \lambda \cdot q_1 + \lambda \cdot \epsilon \cdot q_2 \end{aligned}$$

The product of two dual quaternions is defined as:

$$\begin{aligned} dq_1 ** dq_2 &= (q_{1,1} + \epsilon \cdot q_{1,2}) ** (q_{2,1} + \epsilon \cdot q_{2,2}) \\ &= (q_{1,1} * q_{2,1}) + \epsilon \cdot (q_{1,2} * q_{2,1} + q_{1,1} * q_{2,2}) \end{aligned}$$

As for quaternions the associative and the distributive law hold, but not the commutative law.

For dual quaternions there are three different conjugates:

- *Conjugation of the quaternions:* $\overline{dq} := \overline{q_1} + \epsilon \cdot \overline{q_2} \quad \forall\, dq \in \mathbb{H}_D$
- *Dual conjugation:* $dq^{\epsilon} := q_1 - \epsilon \cdot q_2 \quad \forall\, dq \in \mathbb{H}_D$
- *Total conjugation:* $\overline{dq}^{\epsilon} := \overline{q_1} - \epsilon \cdot \overline{q_2} \quad \forall\, dq \in \mathbb{H}_D$

For the quaternion conjugate, the definition of dual quaternion multiplication yields dq_1 and dq_2 that $\overline{dq_1 ** dq_2} = \overline{dq_2} ** \overline{dq_1}$. The 2-norm of a dual quaternion is given by $\|dq\| := \sqrt{dq ** \overline{dq}}$ and the inverse of a dual quaternion is $dq^{-1} = \frac{\overline{dq}}{\|dq\|^2}$. In all three cases the quaternion conjugate is meant.

Dual quaternions can be used for the representation of pose in the three dimensional Euclidean Group. The quaternion q_r representing the rotation is chosen to lay on the unit sphere S_3. To represent the translation $(t_1, t_2, t_3)^\top \in \mathbb{R}^3$ let $q_t := [0, t_1, t_2, t_3]$ be a second quaternion. Thus the dual quaternion

$$dq := q_r + \epsilon \frac{1}{2} \cdot q_t * q_r$$

represents the transformation in $S_3 \times \mathbb{R}^3$.
Any point $p = (u, v, w)^\top$ can be embedded to $\mathbb{H}_D$ by the dual quaternion $p_d = [1, 0, 0, 0] + \epsilon \cdot [0, u, v, w]$. The transformation of this point about dq is then $dq ** p_d ** \overline{dq}$.
This pose representation contains the important property that the composition of motions or of a pose followed by a motion is represented easily by the product of dual quaternions:

$$\begin{aligned} p_{new} &= dq_2 ** dq_1 ** p_{old} ** \overline{dq_1} ** \overline{dq_2} \\ &= dq_2 ** dq_1 ** p_{old} ** \overline{dq_2 ** dq_1} \end{aligned}$$

The rotation and translation a dual quaternion describes can also be expressed in terms of a rotation matrix R and a translation vector t by the formula:

$$\begin{pmatrix} R & t \\ 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} p \\ 1 \end{pmatrix}$$

which is equivalent to the transformation of the point p, corresponding to $\begin{pmatrix} p \\ 1 \end{pmatrix}$, by the dual quaternion $dq = [q_r, \frac{1}{2} q_t * q_r]$ in ordinary form:

$$dq ** p ** \overline{dq}$$

Remember the restriction that $q_r \in S_3$ and q_t is an imaginary quaternion.
The equivalence is proven by the fact that rigid motion is equivalent to the one by rotation matrix and translation vector.
Let dq_1 and dq_2 be two dual quaternions with $\|q_{r_1}\| = \|q_{r_2}\| = 1$ and $\operatorname{Re} q_{t_1} = \operatorname{Re} q_{t_2} = 0$

$$dq_1 = q_{r_1} + \epsilon q_{d_1} = q_{r_1} + \epsilon \frac{1}{2} \cdot q_{t_1} * q_{r_1}$$

$$dq_2 = q_{r_2} + \epsilon q_{d_2} = q_{r_2} + \epsilon \frac{1}{2} \cdot q_{t_2} * q_{r_2}$$

which represent the following transformations:

- q_{r_1} corresponds to the rotation matrix R_1
- q_{r_2} corresponds to R_2

- q_{t_1} corresponds to the translation vector t_1
- q_{t_2} corresponds to t_2

Then the rigid motion $\begin{pmatrix} R_3 & t_3 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} R_2 & t_2 \\ 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} R_1 & t_1 \\ 0 & 1 \end{pmatrix}$ is the same as:

$$\begin{aligned} dq_3 &= dq_2 * *dq_1 \\ &= (q_{r_2} + \epsilon q_{d_2}) * *(q_{r_1} + \epsilon q_{d_1}) = q_{r_2} * q_{r_1} + \epsilon(q_{r_2} * q_{d_1} + q_{d_2} * q_{r_1}) \\ &= q_{r_2} * q_{r_1} + \epsilon \frac{1}{2} \cdot (q_{r_2} * q_{t_1} * q_{r_1} + q_{t_2} * q_{r_2} * q_{r_1}) \\ &= q_{r_2} * q_{r_1} + \epsilon \frac{1}{2} \cdot (q_{r_2} * q_{t_1} * \overline{q_{r_2}} + q_{t_2}) * q_{r_2} * q_{r_1} \\ &= q_{r_3} + \epsilon q_{d_3} \end{aligned}$$

with $q_{r_3} = q_{r_2} * q_{r_1}$ and $q_{d_3} = \frac{1}{2} \cdot q_{t_3} * q_{r_3}$ where $q_{t_3} = q_{r_2} * q_{t_1} * \overline{q_{r_2}} + q_{t_2}$.
By matrix multiplication we get:

$$\begin{pmatrix} R_3 & t_3 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} R_2 * R_1 & R_2 t_1 + t_2 \\ 0 & 1 \end{pmatrix}$$

Thus $R_3 = R_2 * R_1$ and $t_3 = R_2 t_1 + t_2$.

2.2. Distribution Functions

To estimate the pose of an object or equivalently a feature of the object in the special Euclidean group $SE(3)$ we have to choose a probability density function (*pdf*). Most of the *pdf*s depend on some parameters that describe the function but they can be particle based as well. The case of a particle based description of the distribution will be handled in 2.2.4. We want the *pdf*s to satisfy several characteristics:

- The density function has to be independent from the coordinate system. Then a coordinate change causes just a change of the arguments of the *pdf* but not a change of the structure of the parameters.
- The fusion of two probability density informations shall be supported. This is needed for maximum likelihood estimation for instance.
- The uncertain information of an objects pose or the relation of joints in a robots arm where each link has pose uncertainty with respect to the previous link for instance shall be propagated.

- The representation of the *pdf* shall use a reasonably small set of parameters, much fewer parameters than needed for a particle set. Thus computations can be done efficiently.

In the following I will introduce some candidates for the probability distributions and their density functions.

2.2.1. Projected Gaussian

An intuitively good choice on the translation part is the multivariate normal distribution $\mathscr{N}(\mu, \Sigma)$. Now we would like to have something similar on the hypersphere S_3 embedded to $\mathbb{R}^4$ because that would make it easy to deal with correlations between rotation and translation.
An obvious approach for the rotation is the projection of a three dimensional Gaussian distribution from the tangent space to the S_3. We will do this by central projection (i.e. the center of the projection is the midpoint of the 3-dimensional unit sphere in $\mathbb{R}^4$ and the intersections of S_3 with the straight line through any point on the tangent space and the center of projection get the value of the normal distribution of the corresponding point on the three dimensional tangent space).

Definition 2.2.1. *Let S_3 be the 3-sphere and q_0 be an arbitrary point on S_3. Further, let $TS_{q_0} \sim \mathbb{R}^3$ be the 3-dimensional tangent space to S_3 at the point q_0, with a local coordinate system that has the tangent point as origin. Now, let $\mathscr{N}(\mu, \Sigma)$ be a multivariate normal distribution on TS_{q_0} which has p_{TS} as corresponding probability density function.*
Then the central projection

$$\Pi_{q_0} : TS_{q_0} \longrightarrow S_3$$

provides a density function

$$p_{S_3}(x) := \frac{1}{C} \cdot p_{TS}\left(\Pi_{q_0}^{-1}(x)\right)$$

on S_3, with $C = \int_{S_3} p_{TS}\left(\Pi_{q_0}^{-1}(x)\right) \, \mathrm{d}x$.

As there are always two antipodal projected points on the sphere which represent the same point in the tangent space, this captures correctly the topology of the quaternion rotation space.

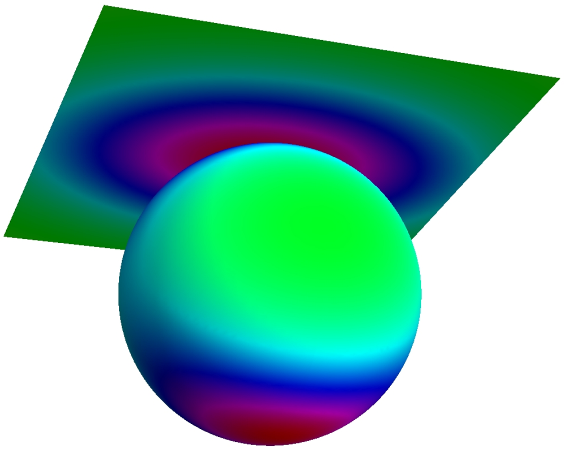


Figure 2.1.: The figure visualizes the central projection of a Gaussian distribution from the tangential plane to a unit sphere S_2.

To define a **base element** in the special Euclidean group $SE(3)$ a tangent point on $S_3 \subset \mathbb{R}^4$, a 6D mean vector μ and a 6×6 covariance matrix Σ are required. In $\mathbb{R}^4$ the quaternions offer a canonical way to create a non-vanishing continuous tangential vector field on the unit sphere S_3. Thus we can define a basis B of the 3-dimensional tangent space TS_{q_0} in $\mathbb{R}^4$ at the tangent point q_0. Further we complete B to be a basis B_0 of $\mathbb{R}^4$ by concatenating as first vector the tangent space normal, which is the tangent point q_0.

How do we get the orthogonal vectors q_1, q_2 and q_3 of the basis B?
Rotations in $\mathbb{R}^4$ can be represented by pairs of unit quaternions q_l, q_r, so that the rotated quaternion is given by $\operatorname{rot}(q) = q_l * q * \overline{q_r}$. Selecting $q_r = e_1$ and $q_l = q_0$, the canonical basis of $\mathbb{R}^4$ is rotated to the tangent point $q_0 = [c_1, c_2, c_3, c_4]$ in quaternion writing. Thus the other vectors of the basis B can be calculated by

$$q_i = q_0 * e_{i+1} * \overline{e_1} = q_0 * e_{i+1} \quad \text{for } i = 1,\, 2,\, 3$$

where e_1 and e_{i+1} are the following unit quaternions $e_1 = \overline{e_1} = [1, 0, 0, 0]$, $e_2 = [0, 1, 0, 0]$, $e_3 = [0, 0, 1, 0]$ and $e_4 = [0, 0, 0, 1]$.
Than the basis is given by the following matrix:

$$B_0 = \left(q_0,\, \underbrace{q_1,\, q_2,\, q_3}_{=B} \right) = \left(\begin{pmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \end{pmatrix} \begin{pmatrix} -c_2 & -c_3 & -c_4 \\ c_1 & -c_4 & c_3 \\ c_4 & c_1 & -c_2 \\ -c_3 & c_2 & c_1 \end{pmatrix} \right) = \begin{pmatrix} c_1 & -c_2 & -c_3 & -c_4 \\ c_2 & c_1 & -c_4 & c_3 \\ c_3 & c_4 & c_1 & -c_2 \\ c_4 & -c_3 & c_2 & c_1 \end{pmatrix}$$

Anyway, the basis of the tangent space can be created randomly in all dimensions through orthogonalization by the Gram-Schmidt process followed by normalization.

Let us come back to the problem to define a base element in the $SE(3)$. I want it to be similar to a Gaussian distribution. Thus I will refer to the distribution function consisting of a Gaussian distribution for the rotation part which can be projected to the S_3 by central projection as introduced in 2.2.1 and another Gaussian distribution for the translation part of the rigid motion with *base element*.
Subsuming the requirements, a base element with specified tangent point q_0 to the hypersphere S_3 and a basis B_0 of the tangent space $TS(q_0, B_0)$ is defined as:

$$\mathscr{N}\left(TS(q_0, B_0), \mu, \Sigma\right)$$

As mentioned above in case of four dimensions the basis can be skipped, as we know then the canonical way of constructing the basis out of the tangent point.
The set of projected probability distributions with the projected density of the normal distribution as density function p_{S_3} on the sphere S_3 is called the set of projected Gaussians (short PG). The subset of PG for which $\mu = 0$ in the corresponding normal distribution on the tangent space TS_{q_0} is denoted as PG_0.
Note that points $r^\perp \in S_3$ that are orthogonal to q_0 are not in the image of the central projection, and by consequence not in the domain of the inverse central projection. Since for each normal distribution the density goes to 0 as the argument goes to infinity, 0 is a continuous completion and we define: $p_S\left(r^\perp\right) := 0$

If the probability density of a given pose shall be evaluated, the vector consisting of the first three entries of the mean vector, which is the mean of a Gaussian kernel in the tangent space, has to be projected to the 3-sphere by central projection as it represents a rotation. Thus the mean becomes a four dimensional vector with length 1 that represents the mean of the rotations on S_3. The last three entries just remain the way they are and correspond to the mean vector of the Gaussian distribution of the translations in $\mathbb{R}^3$. To introduce intuitively values to the covariance matrix it can be written in for of a block matrix:

$$\Sigma = \begin{pmatrix} \text{covMatRotation} & 0 \\ 0 & \text{covMatTranslation} \end{pmatrix}$$

where

$$\text{covMatRotation} = \begin{pmatrix} var(X_1) & cov(X_1, X_2) & cov(X_1, X_3) \\ cov(X_2, X_1) & var(X_2) & cov(X_2, X_3) \\ cov(X_3, X_1) & cov(X_3, X_2) & var(X_3) \end{pmatrix}$$

is the covariance matrix on the tangent space that has to be projected to S_3 to represent the covariance of the rotations in three dimensions and covMatTranslation is the covariance matrix on the other three dimensions that represent the translations.
Thus mean μ and covariance matrix Σ together define a six dimensional Gauss kernel with density:

$$p(x) = \frac{1}{\sqrt{\det(2\pi\Sigma)}} \exp\Big(-\tfrac{1}{2}(x-\mu)^\top \Sigma^{-1}(x-\mu)\Big)$$

A projected Gaussian on a 3-sphere needs to be normalized to 1 to define a probability density as mentioned already in the definition 2.2.1. The normal renormalization constant $\sqrt{\det(2\pi\Sigma)}$ is not sufficient. Now I will explain how the correction weight for the parameterization in the integration to obtain the renormalization factor is calculated. The closed form of the renormalization factor $1/C$ itself involves confluent hypergeometric functions of a matrix argument and is quite complicated to calculate.

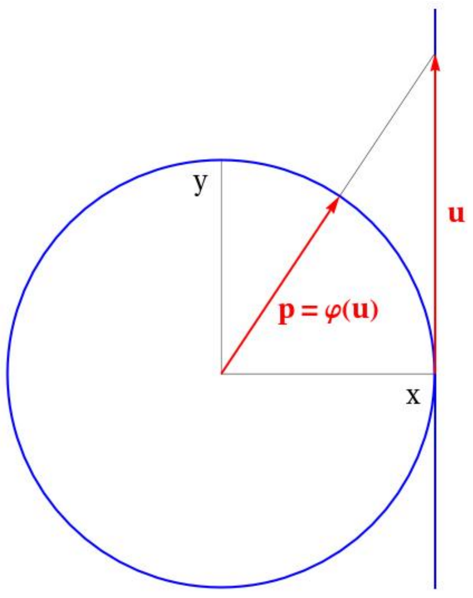


Figure 2.2.

To calculate the surface integral we have to integrate in the directions of the coordinate axes with the Jacobian matrix as a factor for stretching the volume elements. We get this statement from the substitution rule for multiple variables:

Theorem 2.2.2. *Let U be an open set in $\mathbb{R}^n$ and $\varphi : U \to \mathbb{R}^n$ an injective differentiable function with continuous partial derivatives. The Jacobian J_φ of φ is nonzero for every $\mathbf{u} \in U$. Then for any compactly supported, continuous function f with values in $\mathbb{R}$ and with support contained in $\varphi(U)$ it holds that:*

$$\int_{\varphi(U)} f(q)\, \mathrm{d}q = \int_U f(\varphi(\mathbf{u}))\, |\det(\mathrm{D}\,\varphi)(\mathbf{u})|\, \mathrm{d}\mathbf{u}$$

In our case $U = \mathbb{R}^3$, $\mathbf{u} = (u, v, w)^\top$, $f = 1$ and φ has to be defined as the following parametrization of the hemisphere around the x-axis of the hypersphere. See also figure 2.2

$$\begin{aligned} \varphi : \mathbb{R}^3 &\to S_3 \\ \begin{pmatrix} u \\ v \\ w \end{pmatrix} &\mapsto \tfrac{1}{\sqrt{1+u^2+v^2+w^2}} \cdot \begin{pmatrix} 1 \\ u \\ v \\ w \end{pmatrix} \end{aligned}$$

It is sufficient to integrate over a hemisphere as the density of the projected Gaussian is antipodally symmetric and thus the overall integral easily is twice the integral over one half sphere.
The equation from theorem 2.2.2 reduces to:

$$\int_{S_3} 1 \,\mathrm{d}q = \int_{\mathbb{R}^2} \sqrt{\det(J_\varphi^\top \cdot J_\varphi)}(\mathbf{u}) \,\mathrm{d}\mathbf{u}$$

where we replace $|\det(\mathrm{D}\,\varphi)(\mathbf{u})|$ by $\sqrt{\det(\mathrm{D}\,\varphi^\top \cdot \mathrm{D}\,\varphi)}(\mathbf{u})$ as the Jacobi matrix is not symmetric.
For $(J_\varphi)(\mathbf{u})$ we get:

$$\frac{1}{(1+u^2+v^2+w^2)^{\frac{3}{2}}} \cdot \begin{pmatrix} 1+v^2+w^2 & -uv & -uw \\ -uv & 1+u^2+w^2 & -vw \\ -uw & -vw & 1+u^2+v^2 \\ -u & -v & -w \end{pmatrix}$$

and $(J_\varphi)^\top(\mathbf{u})$ is:

$$\frac{1}{(1+u^2+v^2+w^2)^{\frac{3}{2}}} \cdot \begin{pmatrix} 1+v^2+w^2 & -uv & -uw & -u \\ -uv & 1+u^2+w^2 & -vw & -v \\ -uw & -vw & 1+u^2+v^2 & -w \end{pmatrix}$$

Now we can calculate $\sqrt{\det(J_\varphi^\top \cdot J_\varphi)}(\mathbf{u}) = 1/(1+u^2+v^2+w^2)^2$ and thus obtain $\frac{1}{(1+u^2+v^2+w^2)^2}$ as correction weight for the parameterization in the integration.
As the calculation of a closed form of the renormalization factor $1/C = \int_{S_3} p_{S_3}(x)\mathrm{d}x$ is too complicated and very costly in calculation time we do this numerically with Monte Carlo integration which I will introduce in 3.3.2.

An important property of projected Gaussians is the transformation invariance of its pose density. That means the density is independent from the coordinate system. I will show that the density at a certain pose in the euclidean space equals the density of the moved pose by dual quaternions. Every pose and motion in $SE(3)$ can be represented by a base element and the corresponding dual quaternion can be extracted from it. More explications are given in 4.1 and the concrete calculations are given in the appendix A.1.

Definition 2.2.3. *Define the embedding $\mathscr{E}_Q$ of the $\mathbb{R}^4$ into the quaternions $\mathbb{H}$:*

$$\mathscr{E}_Q : \mathbb{R}^4 \to \mathbb{H}$$
$$(a, b, c, d)^\top \mapsto a + ib + jc + kd$$

This function can be used to embed any 4-dimensional vector to the quaternions, but from now on, we will just embed unit vectors on the $S_3 \subset \mathbb{R}^4$ to the quaternions by $\mathscr{E}_Q$.

Definition 2.2.4. *Furthermore define the embedding $\tilde{\mathscr{E}}_Q$ of the $\mathbb{R}^3$ into the imaginary quaternions $\mathbb{H}_{Im}$:*

$$\tilde{\mathscr{E}}_Q : \mathbb{R}^3 \to \mathbb{H}_{Im}$$
$$(b, c, d)^\top \mapsto 0 + ib + jc + kd$$

The vectors in $\mathbb{R}^3$ that shall be embedded are not necessarily unit vectors.
Then it follows that there is an embedding of $\mathbb{R}^4 \times \mathbb{R}^3$ into the dual quaternions $\mathbb{H}_{DQ}$.

Definition 2.2.5. *Let $v_1 = (a, b, c, d)^\top \in \mathbb{R}^4$ be a vector with length one, i.e. $\|v_1\| = \sqrt{a^2 + b^2 + c^2 + d^2} = 1$ and $v_2 \in \mathbb{R}^3$. Then $q_r = \mathscr{E}_Q(v_1)$ and $q_t = \tilde{\mathscr{E}}_Q(v_2)$. Thus we can define*

$$\mathscr{E}_{DQ} : \mathbb{R}^4 \times \mathbb{R}^3 \to \mathbb{H}_{DQ}$$
$$(v_1, v_2) \mapsto q_r + q_d = q_r + \epsilon\frac{1}{2} \cdot q_t * q_r = \epsilon\frac{1}{2} \cdot \tilde{\mathscr{E}}_Q(v_2) * \mathscr{E}_Q(v_1)$$

Now the aim is to prove that the density at the original pose is equal to the moved density with respect to the new pose.

Proof:
Let $dq_0 \in \mathbb{H}_D$ be a dual quaternion describing a starting pose in the Euclidean group.

$$dq_0 = [q_{r0}, q_{d0}] = [q_{r0}, 1/2 \cdot q_{t0} * q_{r0}]$$

with base $B_0 = [b_1, b_2, b_3, b_4]$ where $b_1 = (a_0, b_0, c_0, d_0)^\top$ has the same entries as the tangent point $q_{r0} = [a_0, b_0, c_0, d_0]$. Then we obtain the base:

$$B_0 = \begin{pmatrix} a_0 & -b_0 & -c_0 & -d_0 \\ b_0 & a_0 & -d_0 & c_0 \\ c_0 & d_0 & a_0 & -b_0 \\ d_0 & -c_0 & b_0 & a_0 \end{pmatrix}$$

Define $dq_c \in \mathbb{H}_D$ a constant rigid motion:

$$dq_c = [q_{rc}, q_{dc}] = [q_{rc}, 1/2 \cdot q_{tc} * q_{rc}]$$

Then dq_c applied to dq_0 defines the resulting pose dq_1:

$$\begin{aligned} dq_1 &= dq_c ** dq_0 \\ &= [q_{rc}, q_{dc}] ** [q_{r0}, q_{d0}] \\ &= [q_{rc} * q_{r0}, q_{rc} * q_{d0} + q_{dc} * q_{r0}] \\ &= [q_{rc} * q_{r0}, 1/2 \cdot q_{rc} * q_{t0} * q_{r0} + 1/2 \cdot q_{tc} * q_{rc} * q_{r0}] \\ &= [q_{rc} * q_{r0}, 1/2 \cdot (q_{rc} * q_{t0} * \overline{q}_{rc} + q_{tc}) * q_{rc} * q_{r0}] \end{aligned}$$

where the conjugate $\overline{q}_{rc} = q_{rc}^{-1}$ as q_{rc} is a unit quaternion.

Now let $rp = (u, v, w)^\top \in \mathbb{R}^3$ be a point in the tangent space of the tangent point q_{r0}. Then $q_{rp} = \mathscr{E}_Q(\frac{1}{\sqrt{1+u^2+v^2+w^2}} \cdot B_0 \cdot (1, u, v, w)^\top)$ is the corresponding quaternion to rp projected to the unit sphere S_3. The point tp defined as $(x, y, z)^\top \in \mathbb{R}^3$ represents a translation in the 3-dimensional space. $q_{tp} = \tilde{\mathscr{E}}_Q((x, y, z)^\top) = [0, x, y, z]$ is the embedded point tp in the imaginary quaternions $\mathbb{H}_{Im}$. Then $dq_p = [q_{rp}, q_{dp}] = [q_{rp}, 1/2 \cdot q_{tp} * q_{rp}]$ has a certain density in the system of the original pose dq_0.

If I can show that the by dq_c moved point $dq_c ** dq_p$ corresponds to the same 6-dimensional point in the system of the new pose dq_1 than it holds that the density remains the same under motion by dual quaternions.

$$\begin{aligned} dq_c ** dq_p &= [q_{rc}, q_{dc}] ** [q_{rp}, q_{dp}] \\ &= [q_{rc} * q_{rp}, q_{rc} * q_{dp} + q_{dc} * q_{rp}] \\ &= [q_{rc} * q_{rp}, 1/2 \cdot q_{rc} * q_{tp} * q_{rp} + 1/2 \cdot q_{tc} * q_{rc} * q_{rp}] \\ &= [q_{rc} * q_{rp}, 1/2 \cdot (q_{rc} * q_{tp} * \overline{q}_{rc} + q_{tc}) * q_{rc} * q_{rp}] \end{aligned}$$

Name $dq_{pNew} := dq_c * * dq_p$ the moved point, then $q_{rpNew} = q_{rc} * q_{rp}$ and $q_{tpNew} = q_{rc} * q_{tp} * \overline{q}_{rc} + q_{tc}$.

First I will show that the back projected point $\mathscr{E}_Q^{-1}(q_{rpNew})$ corresponds to the same point $rp = (u, v, w)^\top$ in the new tangent space with base B_1 at the tangent point q_{r1}. This is equivalent to showing that $q_{rpNew} = \mathscr{E}_Q(\frac{1}{\sqrt{1+u^2+v^2+w^2}} \cdot B_1 \cdot (1, u, v, w)^\top)$.
Let be $q_{rc} = [a_c, b_c, c_c, d_c]$ than $q_{r1} = q_{rc} * q_{r0} = [a_0a_c - b_0b_c - c_0c_c - d_0d_c, a_cb_0 + a_0b_c + c_cd_0 - c_0d_c, a_cc_0 + a_0c_c - b_cd_0 + b_0d_c, b_cc_0 - b_0c_c + a_cd_0 + a_0d_c]$ and thus

$$B_1 = \begin{pmatrix} a_1 & -b_1 & -c_1 & -d_1 \\ b_1 & a_1 & -d_1 & c_1 \\ c_1 & d_1 & a_1 & -b_1 \\ d_1 & -c_1 & b_1 & a_1 \end{pmatrix}$$

where $a_1 = a_ca_0 - b_cb_0 - c_cc_0 - d_cd_0$, $b_1 = a_cb_0 + b_ca_0 + c_cd_0 - d_c1c_0$, $c_1 = a_cc_0 - b_cd_0 + c_ca_0 + d_cb_0$ and $d_1 = a_cd_0 + b_cc_0 - c_cb_0 + d_ca_0$.
Now we can calculate:

$$\mathscr{E}_Q(\frac{1}{\sqrt{1+u^2+v^2+w^2}} \cdot B_1 \cdot (1, u, v, w)^\top)$$

$$\begin{aligned} = \tfrac{1}{\sqrt{1+u^2+v^2+w^2}} \quad \cdot \quad & [a_0a_c - b_0b_c - c_0c_c - d_0d_c + (-a_cb_0 - a_0b_c - c_cd_0 + c_0d_c)u \\ + \quad & (-a_cc_0 - a_0c_c + b_cd_0 - b_0d_c)v + (-b_cc_0 + b_0c_c - a_cd_0 - a_0d_c)w, \\ & a_cb_0 + a_0b_c + c_cd_0 - c_0d_c + (a_0a_c - b_0b_c - c_0c_c - d_0d_c)u \\ + \quad & (-b_cc_0 + b_0c_c - a_cd_0 - a_0d_c)v + (a_cc_0 + a_0c_c - b_cd_0 + b_0d_c)w, \\ & a_cc_0 + a_0c_c - b_cd_0 + b_0d_c + (b_cc_0 - b_0c_c + a_cd_0 + a_0d_c)u \\ + \quad & (a_0a_c - b_0b_c - c_0c_c - d_0d_)v + (-a_cb_0 - a_0b_c - c_cd_0 + c_0d_c)w \\ & b_cc_0 - b_0c_c + a_cd_0 + a_0d_c + (-a_cc_0 - a_0c_c + b_cd_0 - b_0d_c)u \\ + \quad & (a_cb_0 + a_0b_c + c_cd_0 - c_0d_c)v + (a_0a_c - b_0b_c - c_0c_c - d_0d_c)w] \end{aligned}$$

On the other side:

$$\begin{aligned} q_{rpNew} &= q_{rc} * q_{rp} \\ &= q_{rc} * \mathscr{E}_Q(\tfrac{1}{\sqrt{1+u^2+v^2+w^2}} \cdot B_0 \cdot (1, u, v, w)^\top) \\ &= \tfrac{1}{\sqrt{1+u^2+v^2+w^2}} \cdot q_{rc} * \mathscr{E}_Q(B_0 \cdot (1, u, v, w)^\top) \\ &= \tfrac{1}{\sqrt{1+u^2+v^2+w^2}} \cdot [a_c, b_c, c_c, d_c] * [a_0 - b_0u - c_0v - d_0w, b_0 + a_0u - d_0v + c_0w, \\ & \quad c_0 + d_0u + a_0v - b_0w, d_0 - c_0u + b_0v + a_0w] \end{aligned}$$

On evaluating this, we get $q_{rpNew} = \mathscr{E}_Q(\frac{1}{\sqrt{1+u^2+v^2+w^2}} \cdot B_1 \cdot (1, u, v, w)^\top)$ as we wanted.

For the translation part there is no such property as for the rotation because the coordinate system remains unchanged. But we can calculate the vector $v_{tp,t0}$ between the point tp and the point corresponding to the quaternion that represents the position of the system:

$$v_{tp,t0} = tp - \tilde{\mathcal{E}}_Q^{-1}(q_{t0})$$

Now we need to show that $v_{tp,t0} = v_{tpNew,t1}$.

$$\begin{aligned}
v_{tNew,t1} &= \mathcal{E}_Q^{-1}(q_{rc} * q_{tp} * \overline{q}_{rc} + q_{tc}) - \mathcal{E}_Q^{-1}(q_{rc} * q_{t0} * \overline{q}_{rc} + q_{tc}) \\
&= \tilde{\mathcal{E}}_Q^{-1}(q_{rc} * q_{tp} * \overline{q}_{rc} + q_{tc} - q_{rc} * q_{t0} * \overline{q}_{rc} - q_{tc}) \\
&= \tilde{\mathcal{E}}_Q^{-1}(q_{rc} * q_{tp} * \overline{q}_{rc} - q_{rc} * q_{t0} * \overline{q}_{rc}) \\
&= \tilde{\mathcal{E}}_Q^{-1}(q_{rc} * (q_{tp} - q_{t0}) * \overline{q}_{rc}) \\
&= \tilde{\mathcal{E}}_Q^{-1}(q_{tp} - q_{t0}) \\
&= v_{tp,t0}
\end{aligned}$$

as we know that $\mathrm{rot}(q) = q_{rc} * q * \overline{q}_{rc}$ is the rotation of q about q_{rc} in quaternionic writing and rotation is a length and orientation preserving operation.

$\square$

I recapitulate that we model a distribution at some point and then know the parameters of the distribution under some rigid motion. For the rotation part we just rotate the tangent points of the mixture elements, for the translation part we just translate the translation part of the parameter vector. Further remember that the "zero mean" requirement of PG_0 only concerns the rotation part of the parameter vector.

The projected Gaussians fulfill all of the upper named desired requirements of the distribution function:

- This density is independent from the coordinate system.
- The fusion of two probability density informations is supported as well as propagation of uncertain information. I will explain it later on in the more general case of mixtures of projected Gaussians 2.3.
- The representation just needs the parameters $TS(q_0, B_0)$, μ and Σ.

2.2.2. Bingham

The Bingham distribution is an antipodally symmetric probability distribution on a unit hypersphere S_d.

Definition 2.2.6. *The probability density function of a Bingham distribution is defined as:*

$$f(x, \Lambda, V) = \frac{1}{F} e^{\sum_{i=1}^{d} \lambda_i (v_i^\top x)^2} = \frac{1}{F} e^{x^\top C x}$$

where the first expression is the standard form for Bingham distributions. F is the normalization constant of the distribution, Λ is a vector of concentration parameters, the columns of the $(d+1) \times d$ matrix V are orthogonal unit vectors and C is a $(d+1) \times (d+1)$ orthogonal matrix.

As can be seen from the second form, the Bingham distribution is derived from a zero-mean Gaussian on $\mathbb{R}^{d+1}$. It is conditioned to lie on the surface of the unit hypersphere S_d and thus models rotational probability densities the best. This property is utilized by Alexander and Buxton in their work [1]. Just as the projected Gaussian, the Bingham distribution fits the antipodal symmetry of the quaternions, since the unit quaternions q and $-q$ represent the same rotation in the 3D space.
A big disadvantage of the Bingham distribution is the computationally expensive renormalization constant F which does not have a closed form in general. Since the distribution must integrate to 1 over S_d, this constant can be written as

$$F(\Lambda) = \int_{x \in \mathbb{S}^d} e^{\sum_{i=1}^{d} \lambda_i (v_i^T x)^2} \mathrm{d}x$$

Glover [13] solved this efficiency problem by using a precomputed lookup table to approximate F.

The main advantage of using a Bingham distribution to model the probabilities of three dimensional orientations on the quaternion hypersphere, is to handle distributions with rotational symmetry in a compact way. In a personal communication Glover told me that they can be used without undue linear approximations like necessary for distributions such as projected Gaussians, which could cause distortions and require more mixture components. The Bingham distribution is well suited to hyperspherical distributions with high variance. However, it remains to be seen whether the benefits of using the Binghams outweigh their added complexity compared to projected Gaussians. Moreover, in the case of a peaked probability density, the difference in accuracy are supposed to vanish.

In summary which of the desired properties does the Bingham distribution fulfill?
The density function is independent from the coordinate system as we wanted. The procedure to merge two Bingham kernels $f_1(X)$ and $f_2(X)$ with its weights λ and

$(1-\lambda)$ by a single one, is described by J. Glover in his paper. The idea behind this merge is to find a Bingham distribution f that fits the joint inertia matrix S_f which is easy to calculate:

$$S_f = \lambda \mathrm{E}_1[xx^\top] + (1-\lambda)\mathrm{E}_2[xx^\top]$$

where $\mathrm{E}_1[xx^\top]$ is the inertia matrix of $f_1(X)$ and $\mathrm{E}_2[xx^\top]$ is the one of $f_2(X)$. The Inertia matrices can be derived from the exponent of the Bingham density:

$$\frac{1}{N}\sum_{i=1}^{d} \lambda_i (v_i^\top x_i)^2 = v_j^\top S v_j$$

The propagation of uncertain information can't be done straight forward with this density function. The composition of Bingham kernels does not provide another Bingham density. At least by now there is no method known to obtain another Bingham kernel. The true distribution can just be Bingham approximated. Moreover it is difficult to represent correlation between rotation and translation. And finally I want to repeat that Bingham distributions are more complex than projected Gaussians and the little gains in accuracy compared to projected Gaussians don't justify this inefficiency.

2.2.3. von Mises-Fisher

The von Mises distribution can be thought of as the spherical analogue of the normal density [20]. It is a continuous probability distribution on the $(n-1)$-dimensional sphere in $\mathbb{R}^n$. For $n = 2$ the distribution reduces to the so called circular normal distribution found by von Mises. In the case $n = 3$ it is called Fisher distribution.

Definition 2.2.7. *In general the density of the von Mises-Fisher distribution for v, a n-dimensional random unit vector, is given by:*

$$p(v,\mu,\kappa) = \frac{\kappa^{n/2-1}}{(2\pi)^{n/2} I_{n/2-1}(\kappa)} \cdot \mathrm{e}^{\kappa\mu^T v}$$

where $I_{n/2-1}$ denotes the modified Bessel function of first kind and of order $\frac{n}{2}-1$, $\|\mu\| = 1$ and $\kappa \geq 0$. Then μ is called the mean direction and κ is the concentration parameter.

The parameters μ and κ determine the shape of the density. As κ increases, the concentration of the distribution around the mean direction becomes higher and the density approaches a normal density.

Back to the case $n = 2$, the von Mises density reduces to:

$$p(v, \mu, \kappa) = \frac{1}{2\pi I_0(\kappa)} e^{\kappa \cos(v-\mu)}$$

where I_0 is the modified Bessel function of order 0.
We want to define a probability density on the special orthogonal group $SO(3)$ from the von Mises distribution in matrix form. Let $R \in SO(3)$, then R is said to have the von Mises-Fisher matrix density, if:

$$p(R) = \frac{1}{c_F} \mathrm{e}^{tr\left[F \cdot R^\top\right]}$$

with respect to the uniform distribution $\mathscr{U}(SO(3))$. F is a 3×3 parameter matrix containing the concentration and $1/c_F$ is the normalization constant depending on F.

Does the von Mises-Fisher distribution satisfy the desired characteristics?
It is independent from the coordinate system but as in matrix form the density is dependent of the matrix R the necessary number of parameters is nine to represent a 3-dimensional density.
In definition 2.2.7 the distribution is defined for any unit sphere, so it could be used for the unit quaternions as well and thus four parameters would be required for the rotation. This distribution would not have the antipodal symmetry, though.
Within the limits of this work it was not feasible to check for the applicability of information fusion and propagation of uncertain information.

2.2.4. Sample Based Description

Instead of choosing a probability density function to approximate the true distribution of the pose in $SE(3)$, we could also work directly with samples. This has the advantage that sampling is extremely general and flexible. Moreover there is a direct proportionality between the numbers of samples and the accuracy. But to get significant results, a large number (the square of what is needed in $\mathbb{R}^3$) is necessary to accurately describe a probability distribution in a six dimensional space.

Thus the desired feature of few parameters is not satisfied and due to the number of necessary samples, the computation gets very slow.

On the other hand the value of the sample based representation as universally usable for all distributions, with arbitrary precision should be honored. Further it can at least be used as a vehicle to obtain experimental results at least in off line calculations.

2.3. Mixture of Projected Gaussians

A typical application for mixtures of projected Gaussians is the following example:

> Given the height of each person in a mixed group of people. As the average height of women and men both can be modeled by a Gaussian kernel the mixture of these two kernels models the distribution of heights of the whole group. Now one could calculate the probability of single persons to be male or female just from the information how tall they are.

In our framework we want to model an approximation of an unknown probability distribution instead of finding disjoint classes. As the probability distributions that can be represented by a single base element are limited, we want to combine several of them to describe more complex distributions as introduced in [6]. This specific use of the concept of mixtures of Gaussian distributions is not as common as the one given in the example above.

Definition 2.3.1. *Let* $\mathrm{PG_i} := \mathscr{N}(TS_i, \mu_i, \Sigma_i)$ *be a sequence of projected Gaussians for* $i \in \{1, \dots, n\}$ *with* μ_i *a d-dimensional mean vector,* Σ_i *a* $d \times d$ *covariance matrix and* $TS_i := TS(q_i, B_i)$ *a corresponding tangent space consisting of a tangent point* q_i *and a basis* B_i*. Furthermore we require* $\sum_{i=1}^{n} \lambda_i = 1$ *and* $0 \leq \lambda_i \leq 1\ \forall\, i \in \{1, \dots, n\}$*.*
Then we define a mixture of projected Gaussians MoPG *as follows:*

$$\begin{aligned} \mathrm{MoPG} &= \lambda_1 \cdot \mathrm{PG_1} + \lambda_2 \cdot \mathrm{PG_2} + \ldots + \lambda_n \cdot \mathrm{PG_n} \\ &= \textstyle\sum_{i=1}^{n} \lambda_i \cdot \mathrm{PG_i} \\ &= \textstyle\sum_{i=1}^{n} \lambda_i \cdot \mathscr{N}(TS_i, \mu_i, \Sigma_i) \end{aligned}$$

In four dimensions it is sufficient to specify a tangent point q_i instead of a whole tangent space TS_i as we know the instruction to construct the canonical basis out of the tangent point.

Such a mixture of projected Gaussians is similar to the multivariate normal distribution. It behaves like a normal distribution on the tangent space and can be treated as one. Another advantage is the low number of parameters. In the case that the tangent point is situated on the hypersphere $S_3 \subset \mathbb{R}^4$ we only have a four dimensional vector in addition to the parameters of the normal distribution μ and Σ.
A MoPG can describe a wide range of distributions from highly peaked ones to wide spread ones. Furthermore it can easily represent correlation between rotation and translation, what is important to model object features properly. I conjecture that a mixture of projected Gaussians (including mixtures with zero variance) can approximate any antipodally-symmetric density. But to approximate a uniform distribution or step function, which doesn't coincidentally form the shape of a bell, a large number of Gaussian kernels are required, to receive fine results.

Of course one has to keep in mind that a MoPG is just an approximation of the true distribution function that would be appropriate for pose estimation on the special Euclidean group. The more base elements the mixture contains, the better the true distribution can be approximated, what is conflicting the wish about efficient computability.

2.3.1. Probabilistic Inferences on MoPGs

From the transformation invariance of the pose density of single base elements it is easy to deduce that a complete mixture of projected Gaussians is independent from the coordinate system as well, because we restrict the mixture to consists of a finite sum of base elements.

MoPGs support data fusion. This means the component wise fusion of each element of one mixture with every element of another mixture. Fusing two base elements PG_i and PG_j means calculating a combined base elements PG_{ij} out of the original ones. Therefore a new tangent point p_{ij} on the sphere is determined which lies in the middle between the tangent points p_i and p_j of the base elements to be fused. At p_{ij} the new tangent space is created and the base elements PG_i and PG_j are transformed to this new tangent space. Then the fused covariance matrix Σ_{ij} has to be calculated from the covariance matrices Σ_i and Σ_j:

$$\Sigma_{ij} = \Sigma_i \cdot (\Sigma_i + \Sigma_j)^{-1} \cdot \Sigma_j$$

The mean vector μ_{ij} is obtained from μ_i and μ_j by the formula:

$$\mu_{ij} = \Sigma_j \cdot (\Sigma_i + \Sigma_j)^{-1} \cdot \mu_i + \Sigma_i \cdot (\Sigma_i + \Sigma_j)^{-1} \cdot \mu_j$$

A disadvantage of this procedure is the quickly growing number of elements of the mixture. Therefore a reduction algorithm of the number of summands of a MoPGs is introduced, namely the merge of similar base elements to keep the number small. Actually to merge base elements PG_i and PG_j, the weights λ_i and λ_j of these projected Gaussians in the mixture need to be known. That's why I require the input of λ_i and λ_j to the function. Likewise in data fusion, the new tangent point p_{ij} is calculated from p_i and p_j and PG_i and PG_j are transformed to the new tangent space $TS_{p_{ij}}$ at p_{ij}. Now the new mean μ_{ij} and the new covariance matrix Σ_{ij} can be calculated:

$$\mu_{ij} = \frac{\lambda_i}{\lambda_i + \lambda_j}\mu_i + \frac{\lambda_j}{\lambda_i + \lambda_j}\mu_j$$

$$\Sigma_{ij} = \frac{\lambda_i}{\lambda_i + \lambda_j}\Sigma_i + \frac{\lambda_j}{\lambda_i + \lambda_j}\Sigma_j + \frac{\lambda_i \cdot \lambda_j}{\lambda_i + \lambda_j}(\mu_i - \mu_j)(\mu_i - \mu_j)^\top$$

The weight λ_{ij} of the new base element PG_{ij} is the sum of the weights of the input base elements $\lambda_i + \lambda_j$.
I distinguish clearly between fusing and merging base elements or mixtures as fusion of two MoPGs means examining the information of both of them, in contrast to merging base elements or whole mixtures what stands for joining it to a single one.

Further the composition of a certain or uncertain motion mixture and a pose modeling mixture is supported. To compose a motion base element with another base element describing a pose, the dual quaternions of the base elements are extracted and executed one to the other. Thereby the new mean μ_{ij} is obtained automatically. The covariance matrix Σ_{ij} is calculated by applying the Jacobian matrix J from both sides to the block covariance matrix:

$$\Sigma_{ij} = J \cdot \begin{pmatrix} \Sigma_i & 0 \\ 0 & \Sigma_j \end{pmatrix} \cdot J^\top$$

As the composition usually contains uncertainties which are included in the covariance matrix of the composed base element, we call this instruction random pose transformation. For the case that the motion is secure, the covariance matrix of the rigid motion can easily be set to zero.

That MoPGs consist of a reasonably small set of parameters is an advantage that allows efficient computations with this distribution function.

2.3.2. Comparison of Mixtures of Binghams with MoPG

Both kinds of mixtures are used in similar applications. Some part of the natural scientists prefers mixtures of Binghams as they are the appropriate distribution to model probabilities on sphere surfaces. The other part is persuaded that the modeling error made by the use of mixtures of projected Gaussians is negligibly small and that the facility of working with this kind of distribution outweighs this error by far. Further as in our context we use the mixture of projected Gaussians to approximate the uncertain pose of an object the accuracy that can be reached is limited.
However I want to point out the differences and similarities of the two mixtures.

- For the renormalization constant of both the Bingham distribution and the projected Gaussian no closed form of the integral exists and thus the factor must be approximated. Glover suggests to transform the series expansion of the factor in a way that an approximation of the normalizing constant of the Bingham distribution can be done using standard floating point arithmetic. As this procedure still is very slow he uses a precomputed lookup table in his framework.
  We approximate the renormalization factor of the projected Gaussian distribution via Monte Carlo integration. For the number of samples $n = 1000$ the error ranges in order of magnitude 10^{-3} and the process time is less than half a minute for each base element. For $n = 5000$ the error ranges in order of magnitude 10^{-4} and the process time is about one and a half minutes. We don't need more precise results.

- In the case that a widely spread distribution similar to an uniform distribution shall be modeled it is undoubted that a mixture of Bingham distributions requires a lower number of elements of the mixture than a MoPG. If a distribution with peaked shape shall be modeled, it stands out to proof whether the computation with mixtures of Binghams or with MoPGs is more efficient as than the number of necessary elements of the mixture approximates.

- In both distribution functions the low number of parameters is convincing. This could just be depreciated if the number of mixture elements becomes big.
- How the merge of two single Gaussian kernels is defined I already introduced in 2.3.1. For a mixture of Bingham distributions consisting of two kernels $f(X) = \alpha f_1(X) + (1-\alpha)f_2(X)$ Glover describes an algorithm to approximate $f(X)$ by a single Bingham $g(X)$ in the preprint [10]. With maximum likelihood parameter estimation the sufficient statistic is the sample inertia matrix $\hat{S} = 1/N \sum_{i=1}^{N} x_i \cdot x_i^\top$ for x_i from the sample set $\{x_1, \ldots, x_N\}$. $\hat{S}$ goes to the true inertia matrix $S = \mathrm{E}[x \cdot x^\top]$ as $N \to \infty$. Thus the Bingham distribution $g(X)$ which fits the inertia matrix $S_f = \alpha \mathrm{E}_1[x \cdot x^\top] + (1-\alpha)\mathrm{E}_2[x \cdot x^\top]$ is the maximum likelihood fit of a single Bingham to the mixture $f(X)$. By the way this Bingham distribution $g(X)$ has minimal KL divergence to $f(X)$.
- Let q and r be two Bingham distributed random variables. The composition of the Binghams $p(q)$ and $p(r)$ can be done by the use of the method of moments. This yields a Bingham approximation to the true distribution, $p(qr)$, by computing $\mathrm{E}[qr(qr)^\top]$. Glover plans to develop the composition algorithm for mixtures of Binghams in a future paper. By now it is already secure that the result of composing two Bingham distribution doesn't give a Bingham and thus better results can be achieved by the composition of MoPGs or base elements of the mixture like introduces in 2.3.1.
- By now there is no operation known for data fusion of mixtures of Binghams.

3. Approximation

The quintessence of this chapter shall be to find suitable approximations of mixtures of projected Gaussians to cut down computational effort with minimal loss of accuracy. At least I want to determine an upper bound for the impreciseness resulting from the approximation.
We know that by an infinite mixture of projected Gaussians the pose of a target object could be described user-defined precisely. Of course such a mixture doesn't exist in practice and thus the finite mixtures we use just approximate the true pose. In section 3.2 I introduce criteria to reduce the number of base elements of a MoPG without loosing much from the preciseness of the mixture. Further the section handles a convergence criterion that improves the approximation by fusion of information data. Section 3.3 is concerned with two approaches how a MoPG can be fitted to another. The expectation maximization algorithm fits a mixture to a set of samples drawn from the other mixture by iteratively increasing the log likelihood function of the sample set. In the other approach the L^p norm between the densities of the mixtures is minimized.

Moreover an aim is to study the coherence between particle sets and MoPGs. Therefore let's denote the *composition*, the *fusion* and the *merge* of mixture densities as modification operations.
Let p_1, p_2 and p_3 be the density functions of the mixtures M_1, M_2, $M_3 \in$ MoPG. p_3 shall be generated by one of the modification operations out of p_1 and p_2. If we draw a set of samples $Z_1 = \{z_{1,n}\}_n$ respectively $Z_2 = \{z_{2,n}\}_n$ from each of the densities p_1 and p_2 and apply the same modification operation to them, we obtain the particle set $Z_3 = \{z_{3,n}\}_n$. The probability density $\tilde{p}_3$ is obtained by fitting a mixture of projected Gaussians to the sample set Z_3.

$$\begin{tikzcd}
 & (p_1, p_2) \arrow[rr, "\text{modification operation}"] & & p_3 & \\
 & & & \tilde{p}_3 & \\
\text{draw samples} & \downarrow \ \downarrow & & \uparrow & \text{fit mixture} \\
 & (Z_1, Z_2) \arrow[rr, "\text{modification operation}"] & & Z_3 &
\end{tikzcd}$$

Now we want to examine the dissimilarity between p_3 and $\tilde{p}_3$. We expect it to disappear for a growing number of samples. I introduce several distance measures and study convergence measures in section 3.1 as it is not clear by now which measure for the dissimilarity of probability density functions is appropriate in out topic. In the context of this work it was not possible to solve the distance problem between p_3 and $\tilde{p}_3$ completely. Hence I just introduce some considerations about convergences of approximations in general in 3.2.
We also need to know the similarity respectively the dissimilarity of density functions to evaluate their importance for the accuracy of the mixture and to be able to decide whether an object is graspable for the robot. In the following section these distance measures will be examined further.

3.1. Grasp Criterion

To decide whether an object is graspable for a robot one has to consider several aspects. For instance the relations of the joints in a robots arm contains weak information what has the effect that the pose of the gripper contains little uncertainties. Further the imperfect sensors of the robot produce weak information data and thus the pose of the target object is uncertain. Hence the probability for failure of any grasping task is composed of several failure probabilities. If for an attempt to grasp a failure threshold of $\varepsilon > 0$ is allowed, it has to be split up into the part $\varepsilon' > 0$ for the sensor and camera uncertainty, or the pose uncertainty of the gripper $\varepsilon'' > 0$ and so on.
The part ε''' determines the impreciseness of the MoPG. Thus as many base elements of the mixture M with low weight can be discarded as the remaining approximated mixture M_{app} still has a total preciseness $\geq 1 - \varepsilon'''$.

Mainly I found two different criteria to measure whether an object is graspable for a robot.
The one of them is a kind of *box criterion* because a box B describes the size the gripper can encompass. Thus the robots hand needs to be navigated to a pose where the object fits inside the box which represents the robots hand. In other words the right box B in the set of boxes $\mathbb{B} \subset \mathbb{R}^6$ with defined size has to be selected to enable the robot to grasp the object. The box selection is done by an arg max function f_p that

chooses the box, which contains the maximum of the probability mass to represent the pose of the object, with respect to the probability density p of the mixture distribution.

$$f_p := \arg\max_{B \in \mathbb{B}} \int_B p(x) d\mu_B$$

where $p(x)$ is the probability density at the random point $x \in B$. Then of course the mean of the pose of the gripper has to be navigated to the center of the box selected by f_p.
Even though this criterion is intuitively the correct one, the problem arises to find an $\arg\max$ function that is efficient in practical use and determines the correct box containing the maximum of probability mass. This can become difficult in the case of a non-symmetric probability distribution.

The other criterion concerns the distance between the probability density that describes the gripper and the one that describes the objects pose. The probability that the robot will succeed on grasping the object has its maximum at the point where the distance of the values of the densities of the random variables is minimal, which describe the pose of the robots hand and the estimated pose of the object. Hence the distance between gripper and object pose needs to be small in terms of rotation and translation. Further we require the pose density of the gripper to be focused. As a result, also the density describing the object's pose will have to be focused.

Definition 3.1.1. *Let p be the density of the random variable describing the objects pose and g the reasonably strong focused density of the one that estimates the pose of the gripper. Define a threshold G when the distributions are close enough such that the excepted probability for failure is smaller than $\varepsilon > 0$.*
Then the object is called graspable with error $< \varepsilon$, when:

$$\mathrm{P}\left(dist(g, p) \leq G\right) > 1 - \varepsilon$$

where $dist(\cdot, \cdot)$ is an appropriate distance measure of probability density functions.

The grasp criterion basing on distances of densities of random variables is in most cases the preferable one for our topic.

3.1.1. Distance Measures

In this section I want to examine some distance measures and their characteristics. This is not only necessary for the grasp criterion but also to evaluate similarity of

Gaussian kernels in a mixture.

At first I want to recall some basics about metrics and topologies in general [9]:

Definition 3.1.2. *For any point x in a metric space M we define the open ball of radius $r\,(> 0)$ around x as the set:*

$$B(x, r) = \{y \in M : d(x, y) < r\}$$

A subspace of M is a neighborhood of x if it contains an open ball about x.

Each of these neighborhoods fulfill the axioms of a topology and therefore the union defines a topology on M - the *induced topology.* Every metric space is a topological space in a natural manner, i.e. all definitions and theorems about topological spaces also apply to all metric spaces.

Definition 3.1.3. *A function $f : M_1 \longrightarrow M_2$, from one topological space M_1 to another M_2, is continuous if and only if the inverse image of every open set is open:*
$\forall V$ open, $V \subseteq M_2$, the inverse image $f^{-1}(V) = \{x \in M_1 \mid f(x) \in V\}$ is open.

Theorem 3.1.4. *(ε-δ)-continuity of maps:*
Let (M_1, d_1) and (M_2, d_2) be metric spaces and $f : M_1 \longrightarrow M_2$ a map. f is continuous if $\forall x \in M_1$ and $\forall \varepsilon > 0\ \exists \delta > 0$ such that $\forall y \in M_1$:
$d_1(x, y) < \delta \Rightarrow d_2(f(x), f(y)) < \varepsilon$

In general a *norm* determines a metric and all metrics induce topologies, but the inverse is not true. A metric defines a norm only if it is *translation invariant*, i.e. $d(x, y) = d(\alpha + x, \alpha + y)$ and *homogeneous*, i.e. $d(\alpha x, \alpha y) = |\alpha| \cdot d(x, y)$.

Now I will check the following distance measures for satisfying the desired features:

- Does the measure define a metric or at least a pre-metric?
  Pre-metric means that it generates a topology on the space of probability distributions.
- Is it easy to calculate analytically or is there an efficient numerical calculation?
- Is this a measure that is appropriate to measure distances between probability density functions?

L^p Norm (especially L^2)

Definition 3.1.5. *The Euclidean distance between two points x and y in the n-dimensional space $\mathbb{R}^n$ is defined as:*

$$d_{\mathrm{E}}(x,y) = \sqrt{(x_1-y_1)^2+\cdots+(x_n-y_n)^2}$$

and $d_{\mathrm{E}}(x,0) = \|x\|_2 = \sqrt{x_1^2+\cdots+x_n^2} = \sqrt{x^Tx}$ is the Euclidean norm (also called 2-norm) of x.

Definition 3.1.6. *If p is a real number, $p \geq 1$, define the L^p norm and L^p distance of $x \in \mathbb{R}^n$ by:*

$$\|x\|_p = \left(|x_1|^p+|x_2|^p+\cdots+|x_n|^p\right)^{1/p}$$

$$d_{\mathrm{L}^p}(x,y) = \left(|x_1-y_1|^p+|x_2-x_2|^p+\cdots+|x_n-y_n|^p\right)^{1/p}$$

(while the L^2 norm is the familiar Euclidean norm, the distance in the L^1 norm is known as the Manhattan distance or taxicab norm).

One extends this to $p=\infty$ via

$$\|x\|_\infty = \max\{|x_1|,|x_2|,\ldots,|x_n|\}$$

which is in fact the limit of the p norms for finite p. The L^∞ norm is also known as the *maximum norm.*

It turns out that for all $p \geq 1$ this definition indeed satisfies the following characteristics $\forall x,\ y \in \mathbb{R}^n$ (for $0<p<1$ the triangle inequality is violated):

- $d_{\mathrm{L}^p}(x,y) = d_{\mathrm{L}^p}(y,x)$ (symmetry)
- $d_{\mathrm{L}^p}(x,y) \leq d_{\mathrm{L}^p}(x,z) + d_{\mathrm{L}^p}(z,y)$ (triangle inequality)
- $d_{\mathrm{L}^p}(x,y) \geq 0$ and $d_{\mathrm{L}^p}(x,y) = 0 \iff x=y$ (non-negativity and identity of indiscernibles)
- The length of the vector is positive homogeneous with respect to multiplication by a scalar.

Furthermore the L^p norm is easy to calculate analytically provided that the integrand is easy to calculate.

To define the L^p norm of a function, in our case the density function of a random variable, let $1 \leq p < \infty$ and (Ω, Σ, μ) be a measure space. Consider the set of all measurable functions from Ω to $\mathbb{R}$ whose absolute value raised to the p-th power has finite integral, i.e. $\|f\|_p := \left(\int |f|^p \,\mathrm{d}\mu\right)^{1/p} < \infty$. Thus we define the L^p norm for the difference of two random variables X and Y with the densities pdf_X and pdf_Y as follows:

$$d_{\mathrm{L}^p}(pdf_X, pdf_Y) = \left(\int_\Omega |pdf_X(q) - pdf_Y(q)|^p \,\mathrm{d}q\right)^{1/p}$$

It is well known that the L^p norm is distance preserving under rigid motions.

Mahalanobis Distance

Definition 3.1.7. *The Mahalanobis distance of a vector $x \in \mathbb{R}^n$ from a set of points with mean μ and covariance matrix Σ is defined as:*

$$d_{\mathrm{M}}(x) = \sqrt{(x-\mu)^T \Sigma^{-1} (x-\mu)}$$

The Mahalanobis distance with respect to the covariance matrix Σ between two n-dimensional points x and y in the space $\mathbb{R}^n$ is defined as:

$$d_{\mathrm{M}}(x,y) = \sqrt{(x-y)^T \Sigma^{-1} (x-y)}$$

Moreover the Mahalanobis norm of x is $d_{\mathrm{M}}(x,0) = \|x\|_{\mathrm{M}} := \sqrt{x^T \Sigma^{-1} x}$.

Characteristics of this distance are:

- $d_{\mathrm{M}}(x,y) = d_{\mathrm{M}}(y,x)$ (symmetry)
- $d_{\mathrm{M}}(x,y) \leq d_{\mathrm{M}}(x,z) + d_{\mathrm{M}}(z,y)$ (triangle inequality)
- $d_{\mathrm{M}}(x,y) \geq 0$ and $d_{\mathrm{M}}(x,y) = 0 \iff x = y$ (non-negativity and identity of indiscernibles)

This shows that the Mahalanobis distance is a metric and the analytical calculation is easy.

Definition 3.1.8. *The Mahalanobis distance between the densities of two (multivariate) normal distributed random variables X_1 and X_2 with distributions $\mathcal{N}(\mu_1, \Sigma_1)$ and $\mathcal{N}(\mu_2, \Sigma_2)$ is defined as:*

$$d_{\mathrm{M}}(X_1, X_2) = \sqrt{(\mu_1 - \mu_2)^T (\Sigma_1 + \Sigma_2)^{-1} (\mu_1 - \mu_2)}$$

In this case one has to take care that it suffices to have $\mu_1 = \mu_2$ to get $d_{\mathrm{M}}(X_1, X_2) = 0$ even though the distributions might be different with $\Sigma_1 \neq \Sigma_2$.

I just defined the Mahalanobis distance for single Gaussian kernels. For this framework one would need to expand the definition to distances of arbitrary random variables or at least to the density function of a mixture of projected Gaussians. The problem that arises hereby is that there is no Mahalanobis distance known by now for such a mixture.

Kullback-Leibler Divergence (or Kullback-Leibler Discrimination)

I will use the denotation *Kullback-Leibler divergence* though Kullback and Leibler themselves used this term to refer to $d_{KL}(P\|Q) + d_{KL}(Q\|P)$. To me it seems that the denotation divergence is the most common one.

An informal motivation for Kullback-Leibler (KL) divergence is given in [27]:
Imagine we can draw independent samples x_1, $x_2, \ldots$ which we assume to be either from the probability density function $p(x)$ or from $q(x)$. Now we wish to decide which density is the correct one. An approach might be to continue drawing samples until the likelihood ratio $\prod_i \frac{p(x_i)}{q(x_i)}$ exceeds some predefined threshold $G := 100 : 1$ in favor on one candidate or the other. Equivalently, we could aim to achieve a sample large enough that the logarithm of the likelihood ratio falls outside the bounds $\pm \log(100)$. We don't know where the data stream actually is coming from, but we suppose it to be from $p(x)$. Then the expected value of the log-likelihood-ratio for a single sample is $\mathrm{E}[\log(\frac{p(x)}{q(x)})]$ what defines the KL divergence. Thus the expected log-likelihood-ratio for the full sample will exceed $\log(100)$ if the sample size becomes larger than $\frac{\log(100)}{\mathrm{E}[\log(\frac{p(x)}{q(x)})]}$.

The KL divergence is a non-symmetric measure as explained in [17] of the difference between two probability distributions P and Q. Typically P represents the 'true' distribution of data, Q typically represents a theory, model, description, or approximation of P.

Definition 3.1.9. *For probability distributions P and Q of a discrete random variable the KL divergence is defined to be:*

$$d_{\mathrm{KL}}(P\|Q) = \sum_i P(i) \log \frac{P(i)}{Q(i)}$$

($P > 0, Q > 0 \forall i$)

Definition 3.1.10. *For distributions P and Q of a continuous random variable on $\mathbb{R}$ the KL divergence is defined to be the integral:*

$$d_{\mathrm{KL}}(P\|Q) = \int_{-\infty}^{\infty} p(x) \log \frac{p(x)}{q(x)} \, \mathrm{d}x$$

where p and q denote the densities of P and Q.

More generally, if P and Q are probability measures over a set X, and Q is absolutely continuous with respect to P, then the KL divergence from P to Q is given by:

$$d_{\mathrm{KL}}(P\|Q) = -\int_X \log \frac{dQ}{dP} \, \mathrm{d}P$$

where $\frac{dQ}{dP}$ is the Radon-Nikodym derivative of Q with respect to P, and provided the expression on the right-hand side exists. $\mathrm{d}P$ and $\mathrm{d}Q$ are the densities of the measures P and Q.

A definition of the Radon-Nikodym derivative can be given by the following:

Definition 3.1.11. *Let ν be a σ-finite measure on (X, Σ) that is absolutely continuous with respect to a σ-finite measure μ on (X, Σ). Then it holds that $\exists f : X \longrightarrow (0, \infty)$ measurable such that:*

$$\nu(A) = \int_A f \mathrm{d}\mu$$

*f is usually written as $\frac{\mathrm{d}\nu}{\mathrm{d}\mu}$ and is called the **Radon-Nikodym derivative**.*

Unfortunately the KL divergence is not a metric. It is non-symmetric, i.e. $d_{\mathrm{KL}}(P\|Q) \neq d_{\mathrm{KL}}(Q\|P)$ and it does not satisfy the triangle inequality. At least it is a pre-metric: If $\{P_i\}_{i=1}^n$ is a sequence of distributions such that $\lim_{n\to\infty} d_{\mathrm{KL}}(P_n\|Q) = 0$ then one says $P_n \longrightarrow Q$.
In the discrete case the KL divergence further has the property to be non-negative, i.e. $d_{\mathrm{KL}}(P\|Q) \geq 0$ and $d_{\mathrm{KL}}(P\|Q) = 0 \Longleftrightarrow P = Q$.

Kullback and Leibler themselves defined a symmetric version of the divergence:

$$d_{\mathrm{sKL}}(P, Q) := d_{\mathrm{KL}}(P\|Q) + d_{\mathrm{KL}}(Q\|P)$$

It is symmetric and nonnegative but still does not satisfy the triangle inequality. Further the symmetrized version of the KL divergence can be calculated without big computational effort.

Convergence in Measure

Let E be a set of finite measure and $E_n(\varepsilon) = \{x \in X :| f_n(x) - f(x) |\geq \varepsilon\}$ a set where the values of f_n are at least ε away from f. f and f_n ($n \in \mathbb{N}$) are real valued measurable functions on E. As almost everywhere (a.e.) convergence is the weakened version of point wise convergence, one can say that $\{f_n\}_n$ converges a.e. to f if and only if

$$\lim_{n\to\infty} \mu\left(E \cap \bigcup_{m=n}^{\infty} E_n(\varepsilon)\right) = 0$$

for every $\varepsilon > 0$.
Hence convergence in measure over a set of finite measure is equal to a.e. convergence over sets of finite measure. In general this is not true.

Definition 3.1.12. *Let f, f_n ($n \in \mathbb{N}$) $: X \to \mathbb{R}$ be measurable functions on a measure space (X, Σ, μ). The sequence $\{f_n\}$ is said to converge globally in measure to f if for every $\varepsilon > 0$:*

$$\lim_{n\to\infty} \mu(\{x \in X : |f(x) - f_n(x)| \geq \varepsilon\}) = 0$$

and to converge locally in measure to f if for every $\varepsilon > 0$ and every $F \in \Sigma$ with $\mu(F) < \infty$:

$$\lim_{n\to\infty} \mu(\{x \in F : |f(x) - f_n(x)| \geq \varepsilon\}) = 0$$

There is no metric which includes this sense of convergence, i.e. there are no such properties like triangle inequality for convergence in measure. At least a kind of Cauchy criterion can be defined.
A sequence of functions is called *Cauchy in measure* if for every $\varepsilon > 0$:

$$\mu\left(\{x \in X : |f_m(x) - f_n(x)| \geq \varepsilon\}\right) \longrightarrow 0$$

for $n, m \longrightarrow \infty, n, m \in \mathbb{N}$.

Mutual Information

What does mutual information mean intuitively?
It measures the information that the random variable X and Y share. Suppose I know something about one of these variables. How much reduces this the uncertainty about the other?

For example, if X and Y are independent, then knowing X does not give any information about Y and vice versa. That means their mutual information is zero. On the other extreme, if X and Y are identical then all information known about X is shared with Y completely. In this case the mutual information is the same as the entropy of any of the random variables.

Definition 3.1.13. *The entropy H of a discrete random variable X is defined by:*

$$H(X) = \mathrm{E}[I(X)]$$

with E *the expected value, and $I(X)$ the information content or self-information of X.*

The information content of an event x with probability $\mathrm{P}(x)$ is given by $I(x) = -\log(\mathrm{P}(x))$ and hence $I(X)$ is a random variable. If p denotes the probability mass function of X then the entropy can explicitly be written as:

$$H(X) = \sum_{i=1}^{n} p(x_i)I(x_i) = -\sum_{i=1}^{n} p(x_i)\log p(x_i)$$

Definition 3.1.14. *The mutual information, also transinformation, of two discrete random variables X and Y can be defined as:*

$$I(X,Y) = \sum_{y\in Y}\sum_{x\in X} p(x,y)\log\left(\frac{p(x,y)}{p_1(x)p_2(y)}\right)$$

where $p(x,y)$ is the joint probability distribution function of X and Y, and $p_1(x)$ and $p_2(y)$ are the marginal probability distribution functions of X and Y respectively.

Definition 3.1.15. *In the continuous case, the mutual information of X and Y can be defined as:*

$$I(X,Y) = \int_Y\int_X p(x,y)\log\left(\frac{p(x,y)}{p_1(x)p_2(y)}\right)\mathrm{d}x\mathrm{d}y$$

where $p(x,y)$ is the joint probability density function of X and Y, and $p_1(x)$ and $p_2(y)$ are the marginal probability density functions of X and Y respectively.

As the base of the log function is not specified, these definitions are ambiguous. To change these functions to become unique, the function I could be parameterized as $I(X,Y,b)$ with b the base. An alternative would be to specify the base to be 2, since one bit is the most common unit of measure of mutual information.

Characteristics of the mutual information:

- Mutual information can be expressed as a Kullback-Leibler divergence:

  $$I(X,Y) = d_{\mathrm{KL}}(p(x,y)\|p(x)p(y))$$

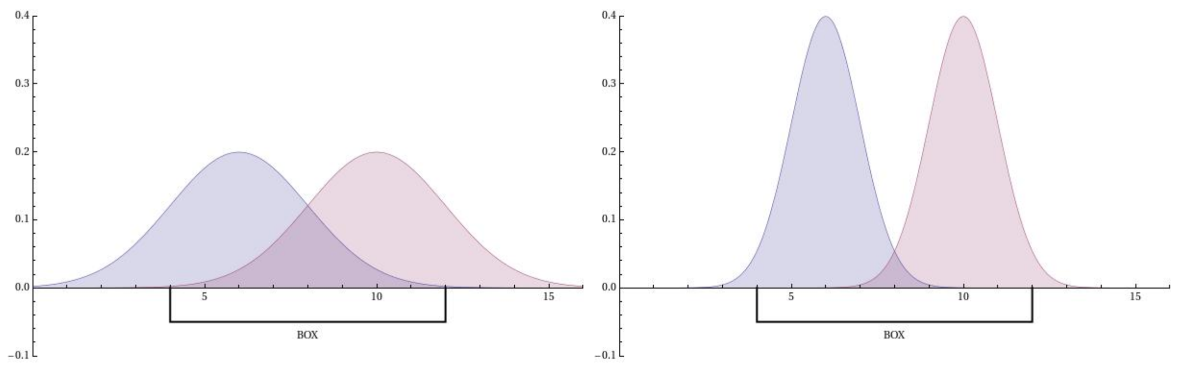
  where $p(x,y)$ is the joint distribution of the random variables X and Y.

- It is a measure of dependence in the following sense:
  $I(X,Y)=0$ if and only if X and Y are independent random variables.

- The measure is non-negative, i.e. $I(X,Y) \geq 0$ for all random variables X and Y and symmetric, i.e. $I(X,Y) = I(Y,X)$.

- Even though mutual information does not define a metric, $d(X,Y) = H(X,Y) - I(X,Y)$ does, where $H(X,Y)$ is the joint entropy of X and Y.

In this framework we will need to evaluate distances between more than two random variables. Thus one of the various extensions of mutual information has to be chosen. The most established extensions are the conditional mutual information and interaction information.
But instead of going more to detail, I want to mention, that this measure doesn't fit the problem properly about the ability to grasp.
Imagine a robot estimates the pose of a target object at least two times and receives the random variable X_1 and X_2 with the probability distributions of the base elements $\mathscr{N}(TS_1,\mu_1,\Sigma_1)$ and $\mathscr{N}(TS_2,\mu_2,\Sigma_2)$ with densities p_1 and p_2. In the case that these Gaussians are far away from each other and thus have a small overlap, the mutual information $I(X_1,X_2) = \int_{X_2}\int_{X_1} p(x_1,x_2)\log\left(\frac{p(x_1,x_2)}{p_1(x_1)p_2(x_2)}\right)\mathrm{d}x_1\mathrm{d}x_2$ gives the right indication on becoming very small that at least one of the measures is unusable and another measure is required to receive a proper pose estimation.
For the case that the Gaussian kernels are close enough for the gripper being able to encompass a high percentage of both of them, we would like the distance measure to show this. For instance in figure 3.1(a) 84% of the probability mass of both of the distributions is inside the box, the robots hand can grasp. Now recall that the more peaked the kernels are, the higher the proportion of the distribution, that is enclosed in the box. This means in case of strongly peaked density functions like an approximation of the Dirac delta function $\delta_\varepsilon(x) = \frac{1}{\sqrt{2\pi\varepsilon}}\,\mathrm{e}^{-x^2/(2\varepsilon)}$ where $\varepsilon > 0$, one can be almost sure that the object has its pose somewhere in the box, that contains the probability mass of both of the Gaussians. In the figure 3.1(b) below 97,8% of the probability mass is inside the box.



(a) $I(X_1, X_2) = k$ (b) $I(X_1, X_2) = n < k$

Figure 3.1.

Unfortunately the mutual information just shows whether the kernels have big or small overlap, and thus in the case of peaked Gaussian kernels it gives a small value. That's why I arrived at the conclusion that this distance measure is not suitable for our problem.

3.1.2. Convergence Measures

This is an attempt to grade the different convergences in order of strength.
The properties of sequences of functions (or random variables) can vary a lot for growing indices. Hence one needs quite different kinds of convergences, which usually are with respect to various norms or topologies even though there are sometimes other kinds of convergences like convergence in measure as well.

The classical types of convergence are the

- **pointwise convergence:**
  Let f_n be a sequence of functions on the same domain D. One says f_n converges pointwise to the limit f if $f(x) = \lim_{n\to\infty} f_n(x)$
- and the **uniform convergence:**
  A sequence f_n converges uniform to f if maximal differences between f_n and f converge to zero. This is a kind of convergence in terms of the maximum norm. The limit function f has the property that if the sequence is continuous, then the uniform limit also is continuous. Furthermore it holds that the integral of the uniform limit is the limit of the integral of the sequence, i.e. $\lim_{n\to\infty} \int_a^b f_n \mathrm{d}x =$

$\int_a^b f \mathrm{d}x$ and the derivative of the uniform limit is the limit of the derivative of the sequence, i.e. $\lim_{n\to\infty} f_n' = f'$.

In measure theory these types of convergence usually are unambiguous and thus one can only define the convergence *almost everywhere.*

- **pointwise convergence almost everywhere (a.e.)**
  The convergence is not true at the most on a set with zero measure.
- **convergence in measure**
  If a sequence converges almost everywhere in a space with finite measure $\mu(\Omega) < \infty$ then it converges in measure. Thus convergence in measure is weaker than convergence a.e.
- **L^p convergence**
  A sequence converges in L^p if
  $\lim_{n\to\infty} \|f_n - f\|_p = \lim_{n\to\infty} \left(\int_\Omega \|f_n(x) - f(x)\|^p \,\mathrm{d}\mu(x)\right)^{1/p} = 0$. Hence from L^p convergence follows convergence in measure.
- **almost uniform convergence**
  A sequence converges almost uniform if $\forall \varepsilon > 0 \; \exists A \in \Sigma : \mu(A) < \varepsilon$ and the sequence converges uniformly on $\Sigma \backslash A$.
- **convergence in probability (weak convergence)**
  It is related to convergence in measure. There are several equivalent definitions of weak convergence of a sequence of measures (see Portmanteau).

There are two hierarchies of convergences $f_n \to f$ in spaces with finite measure $\mu(\Sigma) < \infty$:
uniform convergence
$\Rightarrow$ pointwise convergence
$\Rightarrow$ pointwise convergence almost everywhere $\Leftrightarrow$ almost uniform convergence
$\Rightarrow$ convergence in measure

uniform convergence
$\Rightarrow$ L^∞ convergence
$\Rightarrow$ L^p convergence, for all real $0 < p < \infty$
$\Rightarrow$ convergence in measure
$\Rightarrow$ convergence in probability

3.2. Behavior and Properties of Approximations of MoPGs

On approximating mixtures of projected Gaussians one surely wants to know what properties remain. There are various theoretical questions to answer:

- *What kinds of properties are passed on from the original* MoPG*s to the new* MoPG*, one obtains after applying one of modification operations fusing, merging or composing to it?*
- *What do we know about the accuracy of the approximation?*
  Let's assume to have the pdf of a known MoPG *that fulfills the grasp criterion with a certain level for failure $\varepsilon > 0$. On approximating the pdf we want to preserve that the resulting pdf still fulfills the criterion.*

About the approximation and simplification step of the mixture:

- *Which kernels can be omitted? Can an upper bound be given for the error that arises from dropping kernels? Which threshold for the weights is appropriate to discard the ones below?*
- *Can various kernels be replaced by a single one, if they have similar mean and covariance? And what means 'similar'? Is there a kind of pdf that might replace kernels with equal mean, but different covariances?*
- *How many kernels are required and are reasonable to model different kinds of distributions like an identical distribution for instance with a mixture of projected Gaussians?*

3.2.1. Error Estimation on Dropping Base Elements of a Mixture

Let $M \in$ MoPG be a mixture of projected Gaussians with the density $p_M(x) = \sum_{i=1}^{n} \lambda_i \cdot p(\mu_i, \Sigma_i, x)$. Lets denote $p_i = p(\mu_i, \Sigma_i, x)$. Now we want to approximate M by the mixture $M_{app} \in$ MoPG. This approximated mixture M_{app} can easily be achieved by discarding the less relevant base element with smallest weight λ_{i_0}. For easier notation renumber the weights λ_i and densities p_i such that λ_{i_0} becomes the last one. As we know that the λ_i have to sum to $1 = \sum_{i=1}^{n} \lambda_i$, we can renormalize the remaining weights by:

$$\lambda'_i := \lambda_i + \frac{\lambda_i \lambda_n}{1-\lambda_n} = \frac{\lambda_i}{1-\lambda_n}$$

for all $i \in \{1, \ldots, n-1\}$.

Now an upper bound for the error of the approximation can be given. I will calculate the total variance which is the maximal error that can occur on trying to grasp an object by using the box criterion. Remember that this uses the arg max of an integral over the probability density enclosed in any box B of the set of boxes $\mathbb{B}$.

$$\begin{aligned}
|\mathrm{P}(B) - \mathrm{P}_{app}(B)| &= \left| \int_B \sum_{i=1}^{n} \lambda_i p_i \, \mathrm{d}\mu_B - \int_B \sum_{i=1}^{n-1} \lambda'_i p_i \, \mathrm{d}\mu_B \right| \\
&= \left| \int_B \sum_{i=1}^{n-1} (\lambda_i - \lambda'_i) p_i + \lambda_n p_n \, \mathrm{d}\mu_B \right| \\
&\leq \left| \int_B \sum_{i=1}^{n-1} \left(- \frac{\lambda'_i \lambda_n}{1-\lambda_n} \right) p_i \, \mathrm{d}\mu_B \right| + \lambda_n \underbrace{\int_B p_n \, \mathrm{d}\mu_B}_{\leq 1} \\
&\leq \int_B \left(\frac{\lambda_n}{1-\lambda_n} \right) \sum_{i=1}^{n-1} \lambda'_i p_i \, \mathrm{d}\mu_B + \lambda_n \\
&= \frac{\lambda_n}{1-\lambda_n} \underbrace{\sum_{i=1}^{n-1} \lambda'_i}_{1-\lambda_n} \underbrace{\int_B p_i \, \mathrm{d}\mu_B}_{\leq 1 \; \forall i} + \lambda_n \\
&\leq 2\lambda_n \qquad \forall B \in \mathbb{B}
\end{aligned}$$

Of course the approximation $\left| \int_B p_i \, \mathrm{d}\mu_B \right| \leq 1$ is very rough but it suffices to show that the difference between the approximated and the original probability at most is $2\lambda_{i_0}$.

3.2.2. Fusing MoPGs

Robots commonly have a stereo system of cameras and make several localization attempts of the target object from different points of view. Thus in general a couple of mixtures are obtained that seem to be reliable and all describe the same object. If we

believe in these observations, we need to think of a solution to join the information we get from the single mixtures.
Our approach to utilize all the information, is to *fuse the mixtures* in order to receive the best possible probability distribution for the pose.

Let M_1, $M_2 \in$ MoPG be two mixtures of projected Gaussians with densities p_{M_1} and p_{M_2}. To obtain the base elements of the fused mixture $p_{M_3} = fuse(p_{M_1}, p_{M_2})$ each of the base elements of M_1 has to be fused with all of the base elements of M_2. How this fusion works is briefly introduced in [8] and I explained it in section 2.3.1.
It doesn't make sense to fuse widely separated base elements as then a systematic overestimation of the concentration of the covariance matrix results. Thus we require all covariance matrices of the mixtures to be sufficiently well peaked. There are further things that shall be payed attention to:

- The tangent points q_i and q_j of the the base elements $\mathrm{PG}_i \in M_1$ and $\mathrm{PG}_j \in M_2$ to be fused need to be sufficiently close. In practice it turned out to make no sense to allow a bigger angle than 15° between the point q_i and q_j on the hypersphere S_3. To assure that the base elements are compatible a weighting factor

$$\alpha_{ij} = \mathrm{e}^{-5 \cdot \arccos((q_i \cdot q_j)^2)}$$

  is introduced. The angle θ between q_i and q_j can be calculated with $\theta = \arccos(q_i \cdot q_j)$. By taking the square of the scalar product of the tangent points $(q_i \cdot q_j)^2$ it is secured that the exponent of the function and thus also the function is antipodal symmetric on the sphere. The factor -5 was obtained by heuristics and has the effect that the whole function goes to 0 reasonably quick.

- If both base elements PG_i and PG_j shall be applied at the same moment, the dissimilarity of the distribution functions has to be small as well. Note that I require the base elements to be projected to the same tangent space already. We use the Mahalanobis distance to weight the fused base element $\mathrm{PG}_{i,j}$:

$$\delta_{ij} = \mathrm{e}^{-1/2 \cdot (\mu_i - \mu_j)(\Sigma_i + \Sigma_j)^{-1}(\mu_i - \mu_j)^\top}$$

  This expresses that even if the base elements share the same tangent space, they could be incompatible because the distributions might be too different. A disadvantage of this weighting function is that for base elements that by accident have the same mean $\mu_i = \mu_j$ the maximal weight is returned.

All together each of the summands of the fused mixture has the form:

$$C \cdot \lambda_i \lambda_j \alpha_{ij} \delta_{ij} \cdot \mathrm{PG}_{ij}$$

where $C = 1/(\sum_{i=1}^{n_1} \sum_{j=1}^{n_2} \lambda_i \lambda_j \alpha_{ij} \delta_{ij})$ is the normalizing constant for p_{M_3} to be a probability density function, λ_i and λ_j are the weights of the base elements in their original mixtures and PG_{ij} is the fused base element with density p_{M_3}.

The problem that arises with this approach to fuse mixtures is that the resulting mixture M_3 consists of $n_1 \cdot n_2$ summands for M_1 consisting of n_1 and M_2 consisting of n_2 elements. To reduce this rapidly increasing number of base elements, kernels with low weight can be omitted as I mentioned already in 3.2.1. Another strategy is to merge similar kernels what I will explain in the following.

3.2.3. Merging MoPGs

The moment-preserving merge [7] is a common procedure to substitute two elements of a MoPG by a new one, matching the zeroth, first and second-order moments of the original mixture. I described this merge already in section 2.3.1.
The more interesting point of merging elements of a mixture is the best choice of the projected Gaussians. The Mahalanobis distance of two Gaussian kernels, I introduced in definition 3.1.8 might match the problem to check the compatibility of single kernels, but for this topic I have to chose another dissimilarity measure that fits whole mixtures of projected Gaussians. Actually I don't need to know *which kernels of the mixture are the most similar*, I want to find the kernels such that after merging them, *the whole approximated mixture is the least dissimilar from the mixture before the merge.*

Williams searches in his master's thesis [34] for a scalar cost function which measures the difference between the density of the original mixture and the approximated mixture in order to evaluate whether one merge is 'better' than another. He proposes to use the square of the L^2 norm

$$d_{ISD}(f_1, f_2) = \int (f_1(x) - f_2(x))^2 \, \mathrm{d}x$$

which he refers to as integral square difference measure (ISD).
Further he introduces the Kolmogorov variational distance $d_K(f_1, f_2) = \int |f_1(x) - f_2(x)| \, \mathrm{d}x$ which has an intuitively appealing probability mass interpretation and the

Maximum Likelihood measure $d_{ML}(f_1, f_2) = \int f_1(x) log(f_2(x))\, \mathrm{d}x$ which would fit the requirements of a cost function the best. But only the ISD measure provides the advantage to be computable in closed form. Anyway the Williams criterion is disadvantageous for the reason that the optimization often finds local minima and Runnalls constructed an example that showed the scale-dependency of the ISD cost measure which also can lead to anomalies.
Salmond with his criterion reduces the number of components by repeatedly choosing the two most similar components and merging them. The similarity is derived from a statistical analysis of the variance. For any two mixture elements $G_i = \lambda_i \cdot \mathscr{N}(\mu_i, \Sigma_i)$ and $G_j = \lambda_j \cdot \mathscr{N}(\mu_j, \Sigma_j)$ the dissimilarity measure proposed by Salmond is defined as:

$$d_S(G_i, G_j) = \mathrm{tr}(\Sigma^{-1} \frac{\lambda_i \lambda_j}{\lambda_i + \lambda_j} (\mu_i - \mu_j)(\mu_i - \mu_j)^\top)$$

where Σ is the 'overall variance' of the mixture, $\Sigma = \sum_{i=1}^n \lambda_i \Sigma_i + \sum_{i=1}^n \lambda_i (\mu_i - \mu)(\mu_i - \mu)^\top$ and $\mu = \sum_{i=1}^n \lambda_i \mu_i$ is the 'overall mean' of the mixture.
Major drawbacks are that the measure just depends on the means of the components, not on their individual covariances and that adding a new component might alter the merge order of existing components. Thus in various cases unfavored behavior of the merging algorithm arises.

A more promising criterion is the dissimilarity measure based on Kullback-Leibler (KL) divergence. I developed a variant based on the symmetrized version of the Kl divergence which I will refer to as sKL divergence.

From Runnalls [27] paper we know that the following holds:

Theorem 3.2.1. *Let $p_1(x)$ be the density of a d-dimensional Gaussian distribution $\mathscr{N}(\mu_1, \Sigma_1)$ and $p_2(x)$ be the density of a d-dimensional distribution $\mathscr{N}(\mu_2, \Sigma_2)$. Then:*

$$2 d_{KL}(p_1, p_2) = \mathrm{tr}\left(\Sigma_2^{-1}(\Sigma_1 - \Sigma_2 + (\mu_1 - \mu_2)(\mu_1 - \mu_2)^\top)\right) + \log \frac{\det(\Sigma_2)}{\det(\Sigma_1)}$$

This implies:

$$\begin{aligned} d_{sKL}(p_1, p_2) &= \tfrac{1}{2}\mathrm{tr}\left(\Sigma_2^{-1}(\Sigma_1 - \Sigma_2 + (\mu_1 - \mu_2)(\mu_1 - \mu_2)^\top)\right) \\ &+ \tfrac{1}{2}\mathrm{tr}\left(\Sigma_1^{-1}(\Sigma_2 - \Sigma_1 + (\mu_2 - \mu_1)(\mu_2 - \mu_1)^\top)\right) \\ &= \tfrac{1}{2}\mathrm{tr}\left(\Sigma_2^{-1}\Sigma_1 + \Sigma_1^{-1}\Sigma_2 + (\Sigma_1^{-1} + \Sigma_2^{-1})(\mu_1 - \mu_2)(\mu_1 - \mu_2)^\top\right) - d \end{aligned}$$

which can be calculated much faster as the logarithm cancels out.

Regrettably there is no closed form expression neither for the KL divergence of two mixtures of projected Gaussians, nor for the sKL divergence of two mixtures of projected Gaussians, as the mixture density $p_M = \sum_{i=1}^{n} \lambda_i \, p_i$ consists of a sum, where the p_is are the densities of the single projected Gaussian kernels.
For this reason Runnalls thought of an upper bound of the KL divergence between the mixture before the merge and the mixture after the merge of two similar Gaussian kernels. He denominated this upper bound $B(i, j)$. Analogously I will refer to my upper bound of the *symmetrized KL divergence* as $B_s(i, j)$, which I will derive now.

Theorem 3.2.2. *If $f_1(x)$, $f_2(x)$ and $h(x)$ are any pdfs over d dimensions, $0 \leq \omega \leq 1$ and writing $\overline{\omega}$ for $1 - \omega$, then:*

$$d_{sKL}(\omega f_1 + \overline{\omega} h, \omega f_2 + \overline{\omega} h) \leq \omega \, d_{sKL}(f_1, f_2)$$

Proof:

$$\begin{aligned}
\omega d_{sKL}(f_1, f_2) &- d_{sKL}(\omega f_1 + \overline{\omega} h, \omega f_2 + \overline{\omega} h) = \\
&= \omega \int_{\mathbb{R}^d} f_1 \log \frac{f_1}{f_2} \mathrm{d}x + \omega \int_{\mathbb{R}^d} f_2 \log \frac{f_2}{f_1} \mathrm{d}x \\
&- \int_{\mathbb{R}^d} (\omega f_1 + \overline{\omega} h) \log \frac{\omega f_1 + \overline{\omega} h}{\omega f_2 + \overline{\omega} h} \mathrm{d}x - \int_{\mathbb{R}^d} (\omega f_2 + \overline{\omega} h) \log \frac{\omega f_2 + \overline{\omega} h}{\omega f_1 + \overline{\omega} h} \mathrm{d}x \\
&= \omega \int_{\mathbb{R}^d} f_1 \log \frac{f_1(\omega f_2 + \overline{\omega} h)}{f_2(\omega f_1 + \overline{\omega} h)} \mathrm{d}x + \omega \int_{\mathbb{R}^d} f_2 \log \frac{f_2(\omega f_1 + \overline{\omega} h)}{f_1(\omega f_2 + \overline{\omega} h)} \mathrm{d}x - 0 \\
&\overset{*}{\geq} \omega \int_{\mathbb{R}^d} f_1 \left(1 - \frac{f_2(\omega f_1 + \overline{\omega} h)}{f_1(\omega f_2 + \overline{\omega} h)}\right) \mathrm{d}x + \omega \int_{\mathbb{R}^d} f_2 \left(1 - \frac{f_1(\omega f_2 + \overline{\omega} h)}{f_2(\omega f_1 + \overline{\omega} h)}\right) \mathrm{d}x \\
&= \int_{\mathbb{R}^d} \left(\frac{(\omega f_1 - \omega f_2) \overline{\omega} h)}{\omega f_2 + \overline{\omega} h} + \frac{(\omega f_2 - \omega f_1) \overline{\omega} h)}{\omega f_1 + \overline{\omega} h} \right) \mathrm{d}x \\
&= \int_{\mathbb{R}^d} \frac{(\omega f_1 - \omega f_2)^2 \overline{\omega} h}{(\omega f_1 + \overline{\omega} h)(\omega f_2 + \overline{\omega} h)} \mathrm{d}x \\
&\geq 0
\end{aligned}$$

□

That $*$ holds can be seen from the following:

Lemma 3.2.3. *For all a, $b \in \mathbb{Z}$ it holds that:*

$$\log \frac{a}{b} \geq 1 - \frac{b}{a}$$

Proof:

$$\log\frac{a}{b} \geq 1 - \frac{b}{a} \quad \Longleftrightarrow \quad \frac{a}{b}\log\frac{a}{b} \geq \frac{a}{b} - 1$$

Substitute $x := \frac{a}{b}$

$$x\log x \geq x - 1 \quad \Longleftrightarrow \quad 1 - x + x\log x \geq 0$$

Define $f : \mathbb{Q} \to \mathbb{Q}$:

$$\begin{aligned} f(x) &= 1 - x + x\log x \\ f'(x) &= -1 + \log x + 1 = \log x \end{aligned}$$

Now it can be seen that $f(x)$ has its global minimum at $x = 1$ and $f(1) = 0$. Thus $1 - x + x\log x \geq 0$ is true.

$\square$

Let f_1 be the density of the normalized mixture M_{i+j} consisting of the two base elements PG_i and PG_j, $i \neq j$, that shall be merged. If the base elements do not share the same tangent space anyway I project them to a common one TS_{ij} at the tangent point $p_{ij} = \frac{p_i+p_j}{\|p_i+p_j\|}$ by the double projection: Central projection $\prod_{p_i}$ respectively $\prod_{p_j}$ to the sphere followed by the inverse of the central projection $\prod_{p_{ij}}^{-1}$ to the common tangent space. For easier notation I name these new base elements with PG_i and PG_j as well, as the postulation that they have the same tangent space, doesn't involve further changes. Hence from now on it can be assumed for the whole remainder of this section that the rotation part of PG_i and PG_j live in the same tangent space.
Let f_2 be the density of the mixture M_{ij} consisting of the single kernel PG_{ij} that is obtained by merging the elements PG_i and PG_j with the moment-preserving merge. Note that PG_{ij} also has the same tangent space as PG_i and PG_j. Further let h be the remaining mixture except the two particular kernels of f_1.
Then we get from theorem 3.2.2 that the divergence of the whole mixture after merging the components PG_i and PG_j from the mixture before the merge will not exceed $\omega \cdot d_{sKL}(f_1, f_2)$, where $\omega = \lambda_i + \lambda_j$ and $d_{sKL}(f_1, f_2)$ is the divergence of the normalized mixture M_{i+j} and the mixture of the merged single Gaussian M_{ij}.

Theorem 3.2.4. *If $f(x)$, $h_1(x)$ and $h_2(x)$ are any pdfs over d dimensions, $0 \leq \omega \leq 1$ and writing $\overline{\omega}$ for $1 - \omega$, then:*

$$d_{sKL}(\omega h_1 + \overline{\omega} h_2, f) \leq \omega\, d_{sKL}(h_1, f) + \overline{\omega}\, d_{sKL}(h_2, f)$$

Proof:

$$\begin{aligned}\omega d_{sKL}(\omega h_1 + \overline{\omega} h_2, f) &= \int_{\mathbb{R}^d} (\omega h_1 + \overline{\omega} h_2) \log \frac{\omega h_1 + \overline{\omega} h_2}{f} \mathrm{d}x + \int_{\mathbb{R}^d} f \log \frac{f}{\omega h_1 + \overline{\omega} h_2} \mathrm{d}x \\ &= \omega \int_{\mathbb{R}^d} h_1 \log \frac{\omega h_1 + \overline{\omega} h_2}{f} \mathrm{d}x + \overline{\omega} \int_{\mathbb{R}^d} h_2 \log \frac{\omega h_1 + \overline{\omega} h_2}{f} \mathrm{d}x \\ &+ \int_{\mathbb{R}^d} f \log \frac{f}{\omega h_1 + \overline{\omega} h_2} \mathrm{d}x\end{aligned}$$

$$\begin{aligned}\omega\, d_{sKL}(h_1, f) + \overline{\omega}\, d_{sKL}(h_2, f) &= \omega \int_{\mathbb{R}^d} h_1 \log \frac{h_1}{f} \mathrm{d}x + \omega \int_{\mathbb{R}^d} f \log \frac{f}{h_1} \mathrm{d}x \\ &+ \overline{\omega} \int_{\mathbb{R}^d} h_2 \log \frac{h_2}{f} \mathrm{d}x + \overline{\omega} \int_{\mathbb{R}^d} f \log \frac{f}{h_2} \mathrm{d}x\end{aligned}$$

If we can show that:

$$\omega \int_{\mathbb{R}^d} h_1 \log \frac{\omega h_1 + \overline{\omega} h_2}{f} \mathrm{d}x \leq \omega \int_{\mathbb{R}^d} h_1 \log \frac{h_1}{f} \mathrm{d}x$$

it follows directly:

$$\overline{\omega} \int_{\mathbb{R}^d} h_2 \log \frac{\omega h_1 + \overline{\omega} h_2}{f} \mathrm{d}x \leq \overline{\omega} \int_{\mathbb{R}^d} h_2 \log \frac{h_2}{f} \mathrm{d}x$$

and just remains to check whether:

$$\int_{\mathbb{R}^d} f \log \frac{f}{\omega h_1 + \overline{\omega} h_2} \mathrm{d}x \leq \omega \int_{\mathbb{R}^d} f \log \frac{f}{h_1} \mathrm{d}x + \overline{\omega} \int_{\mathbb{R}^d} f \log \frac{f}{h_2} \mathrm{d}x$$

Now calculate:

$$\begin{aligned}\omega \int_{\mathbb{R}^d} h_1 \log \frac{h_1}{f} \mathrm{d}x - \omega \int_{\mathbb{R}^d} h_1 \log \frac{\omega h_1 + \overline{\omega} h_2}{f} \mathrm{d}x &= \omega \int_{\mathbb{R}^d} h_1 \left(\log \frac{h_1}{f} - \log \frac{\omega h_1 + \overline{\omega} h_2}{f} \right) \mathrm{d}x \\ &= \omega \int_{\mathbb{R}^d} h_1 \left(\log \frac{h_1}{\omega h_1 + \overline{\omega} h_2} \right) \mathrm{d}x \\ &\overset{3.2.3}{\geq} \omega \int_{\mathbb{R}^d} h_1 \left(1 - \frac{\omega h_1 + \overline{\omega} h_2}{h_1} \right) \mathrm{d}x \\ &= \omega \int_{\mathbb{R}^d} h_1 - (\omega h_1 + \overline{\omega} h_2)\, \mathrm{d}x \\ &= \omega \underbrace{\int_{\mathbb{R}^d} h_1\, \mathrm{d}x}_{=1} - \omega \underbrace{\int_{\mathbb{R}^d} \omega h_1 + \overline{\omega} h_2\, \mathrm{d}x}_{=1} \\ &= 0\end{aligned}$$

$$\begin{aligned}
&\int_{\mathbb{R}^d} f \log \frac{f}{\omega h_1 + \overline{\omega} h_2} \mathrm{d}x - \omega \int_{\mathbb{R}^d} f \log \frac{f}{h_1} \mathrm{d}x - \overline{\omega} \int_{\mathbb{R}^d} f \log \frac{f}{h_2} \mathrm{d}x \\
&\quad \leq \int_{\mathbb{R}^d} f \left(1 - \frac{\omega h_1 + \overline{\omega} h_2}{f}\right) \mathrm{d}x - \omega \int_{\mathbb{R}^d} f \left(1 - \frac{h_1}{f}\right) \mathrm{d}x - \overline{\omega} \int_{\mathbb{R}^d} f \left(1 - \frac{h_2}{f}\right) \mathrm{d}x \\
&\quad = \int_{\mathbb{R}^d} \left(f - (\omega h_1 + \overline{\omega} h_2) - \omega f + \omega h_1 - \overline{\omega} f + \overline{\omega} h_2\right) \mathrm{d}x \\
&\quad = \int_{\mathbb{R}^d} \left(f - (\omega + \overline{\omega}) f\right) \mathrm{d}x \\
&\quad = 0
\end{aligned}$$

□

Now let h_1 be the density of PG_i and h_2 the one of PG_j. Thus we have the normalized mixture:

$$M_{i+j} = \frac{\lambda_i}{\lambda_i + \lambda_j} \mathrm{PG}_i + \frac{\lambda_j}{\lambda_i + \lambda_j} \mathrm{PG}_j$$

and $M_{ij} := 1 \cdot \mathrm{PG}_{ij}$.
Then we get from theorem 3.2.4 that $d_{sKL}(f_1, f_2)$ will never raise above:

$$\frac{1}{\lambda_i + \lambda_j} \left(\lambda_i \, d_{sKL}(h_1, f) + \lambda_j \, d_{sKL}(h_2, f)\right)$$

What can equivalently be written in terms of mixtures and base elements:

$$d_{sKL}(M_{i+j}, M_{ij}) \leq \frac{1}{\lambda_i + \lambda_j} \left(\lambda_i \, d_{sKL}(\mathrm{PG}_i, \mathrm{PG}_{ij}) + \lambda_j \, d_{sKL}(\mathrm{PG}_j, \mathrm{PG}_{ij})\right)$$

Putting this together with the upper result, from theorem 3.2.2, it follows:

Theorem 3.2.5. *The sKL divergence of the whole mixture of projected Gaussians* MoPG *before the merge from the mixture* MoPG_{app} *after the merge of the components* PG_i *and* PG_j *of the mixture* MoPG*,* $i \neq j$*, will not exceed:*

$$B_s(i, j) := \lambda_i \, d_{sKL}(\mathrm{PG}_i, \mathrm{PG}_{ij}) + \lambda_j \, d_{sKL}(\mathrm{PG}_j, \mathrm{PG}_{ij})$$

This formula can still be simplified using the result from theorem 3.2.1. Further substitute $\lambda_{ij} := \lambda_i + \lambda_j$ and $\mu_{ij} := \frac{\lambda_i}{\lambda_{ij}}\mu_i + \frac{\lambda_j}{\lambda_{ij}}\mu_j$.
Then the upper bound for the sKL divergence can be written as:

$$\begin{aligned}
B_s(i,j) &= \lambda_i \, d_{sKL}(\mathrm{PG}_i, \mathrm{PG}_{ij}) + \lambda_j \, d_{sKL}(\mathrm{PG}_j, \mathrm{PG}_{ij}) \\
&= \frac{1}{2}\lambda_i \operatorname{tr}\left(\Sigma_{ij}^{-1}\Sigma_i + \Sigma_i^{-1}\Sigma_{ij} + (\Sigma_i^{-1} + \Sigma_{ij}^{-1})(\mu_i - \mu_{ij})(\mu_i - \mu_{ij})^\top\right) - d\,\lambda_i \\
&+ \frac{1}{2}\lambda_j \operatorname{tr}\left(\Sigma_{ij}^{-1}\Sigma_j + \Sigma_j^{-1}\Sigma_{ij} + (\Sigma_j^{-1} + \Sigma_{ij}^{-1})(\mu_j - \mu_{ij})(\mu_j - \mu_{ij})^\top\right) - d\,\lambda_j \\
&= \frac{1}{2}\lambda_i \operatorname{tr}\left(\Sigma_i^{-1}\Sigma_{ij} + \left(\frac{\lambda_j}{\lambda_{ij}}\right)^2 (\Sigma_i^{-1} + \Sigma_{ij}^{-1})(\mu_i - \mu_j)(\mu_i - \mu_j)^\top + \left(\Sigma_i^{-1}\Sigma_{ij}\right)^{-1}\right) \\
&+ \frac{1}{2}\lambda_j \operatorname{tr}\left(\Sigma_j^{-1}\Sigma_{ij} + \left(\frac{\lambda_i}{\lambda_{ij}}\right)^2 (\Sigma_j^{-1} + \Sigma_{ij}^{-1})(\mu_i - \mu_j)(\mu_i - \mu_j)^\top + \left(\Sigma_j^{-1}\Sigma_{ij}\right)^{-1}\right) \\
&- d\,\lambda_{ij}
\end{aligned}$$

where $\Sigma_{ij} := \frac{1}{\lambda_{ij}} \cdot \left(\lambda_i\Sigma_i + \lambda_j\Sigma_j + \lambda_i\,\lambda_j(\mu_i - \mu_j)(\mu_i - \mu_j)^\top\right)$ and d is the dimension of the base elements.
This means $B_s(i,j)$ can be calculated directly from the densities of the base elements PG_i and PG_j.

3.2.4. Formulation of Conjectures

For the whole section let g be the density of the mixture of projected Gaussians that describes the pose of the robots gripper in the $SE(3)$. Further let p, p_∞ respectively p_{app} be the densities of the mixtures of projected Gaussians $M \in$ MoPG, $M_\infty \in$ MoPG respectively $M_{app} \in$ MoPG. M is any mixture that estimates the pose of the target object. M_∞ is the infinitely long mixture that just exists in theory and which would determine the pose of the target object precisely. Finally M_{app} is a mixture approximating M that consists of less summands than M and can be achieved by merging or dropping base elements of M.

Coherence of the Introduced Grasp Criteria

For an appropriate distance measure *dist* there is always a threshold $G \geq 0$ for which it holds that if $dist(g - p) \leq G$ the box the gripper can encompass at its pose close to the estimated pose of the object is the one which contains the maximum of the probability mass.

Of course one aims to find a strictly positive thresholds $G \gneqq 0$ to have a bigger range of tolerance for the action of grasping. Further I suppose that in this case the L^p norm should be chosen as distance measure instead of the KL divergence, as it determines the absolute difference of the densities.

Convergence of Approximation

Theorem 3.2.6. *Let p and p_{app} be mixture densities like defined above. Define a small threshold $G \geq 0$. If we know that through the uncertainty of the approximation we just get a small error $\delta \geq 0$ i.e. $\mathrm{P}(\|p - p_{app}\| > G) \leq \delta$, then it holds:*

$$\mathrm{P}(\|g - p\| \leq G) > 1 - \varepsilon \Longrightarrow \mathrm{P}(\|g - p_{app}\| \leq 2G) > 1 - \tilde{\varepsilon}$$

where $\tilde{\varepsilon} := \varepsilon + \delta$.

Proof:

$$\begin{aligned}
\mathrm{P}\left(\|g - p_{app}\| > 2G\right) &= \mathrm{P}\left(\|g - p + p - p_{app}\| > 2G\right) \\
&\leq \mathrm{P}\left(\|g - p\| + \|p - p_{app}\| > 2G\right) \\
&\leq \underbrace{\mathrm{P}\left(\|g - p\| > G\right)}_{\leq \varepsilon} + \underbrace{\mathrm{P}\left(\|p - p_{app}\| > G\right)}_{\leq \delta} \\
&\leq \tilde{\varepsilon}
\end{aligned}$$

This is equivalent to: $\mathrm{P}\left(\|g - p_{app}\| \leq 2G\right) > 1 - \tilde{\varepsilon}$

□

Theorem 3.2.7. *Cauchy convergence*
Let $\{p_n\}_n$ be a family of densities of mixtures of projected Gaussians consisting of n summands. The first mixture density p_1 just consists of the density of a single base element. The other mixture densities are achieved by recursively concatenating another projected Gaussian density to the existing mixture density p_n in each step $n \to n + 1$. The weights need to be renormalized each time.
On repeating the recursion infinitely long concatenating base elements that estimate the pose of a target object, one would determine the pose of the target by p_∞.
We assume that $\forall \varepsilon > 0$ $\exists N_1 \in \mathbb{N}$ such that $\forall n > N_1$:

$$P(\|g - p_n\| \leq G) > 1 - \varepsilon$$

Further we know that we have a Cauchy sequence $\{p_n\}_n$, i.e.: $\forall\delta > 0\ \exists N_2 \in \mathbb{N}$ $\forall m_0, m_1 > N_2 : \|p_{m_0}, p_{m_1}\| < \delta$. Allow δ to be big enough for that $N_2 + 1 < N_1$. Then choose an arbitrary m with $N_2 < m < N_1$ and it holds:

$$P(\|g - p_m\| \leq G + \delta) > 1 - \varepsilon$$

Proof: Let be $n > N_1 > N_2 + 1$ and $N_2 < m < N_1$ like above.

$$\begin{aligned}
\mathrm{P}(\|g - p_m\| > G + \delta) &= \mathrm{P}(\|g - p_n + p_n - p_m\| > G + \delta) \\
&\leq \mathrm{P}(\|g - p_n\| + \underbrace{\|p_n - p_m\|}_{<\delta} > G + \delta) \\
&\leq \mathrm{P}(\|g - p_n\| > G + \delta - \delta) \\
&= \mathrm{P}(\|p - p_n\| > G) \\
&\leq \varepsilon
\end{aligned}$$

□

3.3. Algorithms for Approximation

Let M_0, $M_{app} \in$ MoPG be two mixtures of projected Gaussians. M_0 is a mixture that describes the probability distribution of the pose of a target object, but might be complicated to calculate or might contain parameters which are unknown to us. The second mixture M_{app} using a reduced number of base elements should be fitted to the first mixture. This can be done in a number of different ways. One is the fit of a set of samples drawn from the first distribution by use of the expectation maximization algorithm. Another possibility is the numerical minimization of the Euclidean norm of the difference of the two probability density functions *pdf*s, as a function of the parameters λ_i, μ_i and Σ_i. In this case the 2-norm is chosen as it is easy to deal with and the absolute difference between the mixtures shall be minimized. In a next step these approaches to reduce the number of elements of a mixture would have to be compared. This stands out to be done in future work.

3.3.1. Expectation Maximization

In the following I will explain how a mixture of projected Gaussians can be fitted to a set of samples drawn from another mixture. This algorithm is well known for mixtures of Gaussians and can be applied to the projected Gaussians in the same manner because we can easily switch between tangent spaces by the double projection, central projection to the sphere and back to another tangent space which is reasonably close to the original one.

A mixture $M_0 \in$ MoPG is defined as $\sum_{i=1}^{n} \lambda_i \cdot \mathscr{N}(TS_i, \mu_i, \Sigma_i)$ as we know from definition 2.3.1 and has the density

$$p(x) = \sum_{i=1}^{n} \lambda_i \cdot \varphi(x|TS_i, \mu_i, \Sigma_i)$$

where $\varphi(x|TS_i, \mu_i, \Sigma_i) = \varphi_{TS_i,\mu_i,\Sigma_i}(x)$ is the density of the i-th base element.

Let us introduce now the latent variable z. In this context latent means to be hidden. z is a n-dimensional binary random variable consisting of a 1-of-n representation what means a certain element $z_i = 1$ and all the other $n-1$ elements equal 0. Together the values of z_i thus satisfy $\sum_{i=1}^{n} z_i = 1$ and there are n possible states which element of the vector z is nonzero. We define the marginal distribution $\mathrm{P}(z)$ over z in terms of the weighting coefficients λ_i corresponding to the weights of the mixture M:

$$\mathrm{P}(z_i = 1) := \lambda_i \quad \text{for } i = 1, \ldots, n$$

As z has a 1-of-n representation we can write the probability distribution of z in the form

$$\mathrm{P}(z) = \prod_{i=1}^{n} \lambda_i^{z_i}$$

In the same way the conditional probability of x given a particular value for z is the distribution function of the corresponding base element

$$\mathrm{P}(x|z_i = 1) = \mathscr{N}(x|TS_i, \mu_i, \Sigma_i)$$

which can also be written in the form

$$\mathrm{P}(x|z) = \prod_{i=1}^{n} \mathscr{N}(x|TS_i, \mu_i, \Sigma_i)^{z_i}$$

Then the joint distribution $\mathrm{P}(x, z)$ is given by $\mathrm{P}(z) \cdot \mathrm{P}(x|z)$ and thus the distribution of the whole mixture of projected Gaussians is obtained by summing over all possible states of z:

$$\mathrm{P}(x) = \sum_{i=1}^{n} \mathrm{P}(z) \cdot \mathrm{P}(x|z) = \sum_{i=1}^{n} \lambda_i \cdot \mathscr{N}(x|TS_i, \mu_i, \Sigma_i)$$

This means MoPGs can be interpreted in terms of *discrete latent variables.* And a general technique for finding maximum likelihood estimators in latent variable models is the **expectation maximization (EM) algorithm** which is an algorithm that has brought applicability [14]. At first I will give a more informal motivation of the EM algorithm and describe explicitly the steps of this algorithm afterwards.

The log likelihood function for a data set $X = \{x_1, \dots x_N\}$ of independently drawn samples from a distribution $\mathrm{P}(X)$ is given by:

$$\ln \mathrm{P}(X|TS, \mu, \Sigma, \lambda) = \sum_{j=1}^{N} \ln \left(\sum_{i=1}^{n} \lambda_i \mathscr{N}(x_j|TS_i, \mu_i, \Sigma_i) \right)$$

It expresses how probable the observed data set is for different settings of the parameters TS, μ, Σ and λ. Note that the likelihood function is not a probability distribution.

Theorem 3.3.1. *Bayes' theorem*
For two events A and B with positive probability $\mathrm{P}(B) > 0$ it holds:

$$\mathrm{P}(A|B) = \frac{\mathrm{P}(B|A)\mathrm{P}(A)}{\mathrm{P}(B)}$$

For a segmentation of the sample space Ω into a finite number of disjunct events A_i, $i = 1, \dots, N$ and an event B with $\mathrm{P}(B) > 0$ it holds:

$$\mathrm{P}(A_i|B) = \frac{\mathrm{P}(B|A_i)\mathrm{P}(A_i)}{\sum_i \mathrm{P}(B|A_i)\mathrm{P}(A_i)} = \frac{\mathrm{P}(B|A_i)\mathrm{P}(A_i)}{\mathrm{P}(B)}$$

Set the *posterior probability* which is also called *responsibility*:

$$\gamma(z_i) \equiv \mathrm{P}(z_i = 1|x) = \frac{\lambda_i \mathscr{N}(x|TS_i, \mu_i, \Sigma_i)}{\sum_{k=1}^{n} \lambda_k \mathscr{N}(x|TS_k, \mu_k, \Sigma_k)}$$

where the values can be found using the Bayes' theorem.
Thus we obtain:

$$\gamma(z_{j,i}) := \frac{\lambda_i \mathscr{N}(x_j|TS_i, \mu_i, \Sigma_i)}{\sum_{k=1}^{n} \lambda_k \mathscr{N}(x_j|TS_k, \mu_k, \Sigma_k)}$$

Note that the samples $x_i \in S_3 \times \mathbb{R}^3 \; \forall i \in \{1, \dots, N\}$ are drawn from the special Euclidean group such that the rotation part lies on the 3-sphere. To assign the appropriate responsibilities to these samples they have to be reprojected by $\prod_{p_k}^{-1}(x_i)$ to the tangent space of the Gaussian kernel with tangent point p_k for $k = 1, \dots, n$.

Maximizing the log likelihood function for a projected Gaussian mixture model turns out to be a more complex problem than for the case of a single projected Gaussian. Fortunately the EM algorithm is an elegant and powerful method for finding maximum likelihood solutions for models with latent variables. It consists of the two following steps between which we alternate until the algorithm converged as stated in [4]. Initially there are arbitrary values chosen for the means, covariances and weighting coefficients.

- **Expectation step (E step):**
  The current values are used for the parameters to evaluate the posterior probabilities or responsibilities.

- **Maximization step (M step):**
  The probabilities obtained in the E step are used to reestimate the means, covariances and mixing coefficients. Then the tangent spaces are changed by double projection so that we obtain Gaussian kernels with zero mean for the rotation.

It can be shown that each update to the parameters resulting from an E step followed by an M step is guaranteed to increase the log likelihood function $\ln \mathrm{P}(X|TS, \mu, \Sigma, \lambda)$. In practice, one expects the algorithm to have converged when the change in the log likelihood function, or alternatively in the parameters, falls below some fixed threshold. It is well known that the EM algorithm needs comparatively many iteration steps and that each cycle is computationally expensive. Thus it is common to run other algorithms like the n-means algorithm first to achieve better initial values than randomly chosen ones. Further I want to mention that another disadvantage of the EM algorithm arises from the fact that it might get stuck in some local maxima of the log likelihood function instead of finding the global maximum. This is a second indication for the need to choose the initial values carefully.

Summary of the EM algorithm

1. Set the initial value for the means μ_i, covariance matrices Σ_i and weighting coefficients λ_i and evaluate the log likelihood with these values.

2. E step:
   Evaluate the responsibilities $\gamma(x_{n,i})$ using the current parameter values
   $$\gamma(z_{j,i}) := \frac{\lambda_i \mathscr{N}(x_j|TS_i, \mu_i, \Sigma_i)}{\sum_k \lambda_k \mathscr{N}(x_j|TS_k, \mu_k, \Sigma_k)}$$

3. M step:
   Reestimate the parameters using the current responsibilities

$$\mu_i^{new} = \frac{1}{N_i} \sum_{j=1}^{N} \gamma(z_{j,i}) \cdot x_j$$

$$\Sigma_i^{new} = \frac{1}{N_i} \sum_{j=1}^{N} \gamma(z_{j,i})(x_j - \mu_i^{new})(x_j - \mu_i^{new})^\top$$

$$\lambda_i^{new} = \frac{N_i}{N}$$

   where $N_i = \sum_{j=1}^{N} \gamma(z_{j,i})$

4. Evaluate the log likelihood:

$$\ln \mathrm{P}(X|TS, \mu, \Sigma, \lambda) = \sum_{j=1}^{N} \ln \left(\sum_{i=1}^{n} \lambda_i \mathscr{N}(x_j|TS_i, \mu_i, \Sigma_i) \right)$$

   and check for convergence of either the parameters or the log likelihood. If the convergence criterion is not satisfied return to the E step.

3.3.2. Monte Carlo

Let p_0 be the density of the mixture of projected Gaussians $M_0 \in$ MoPG and p_{app} be the density of $M_{app} \in$ MoPG. The square of the L^2 norm of these probability density functions p_0 and p_{app} is defined as:

$$\|p_0 - p_{app}\|_2{}^2 := \int_{S_3 \times \mathbb{R}^3} (p_0(q) - p_{app}(q))^2 \, \mathrm{d}q$$

The minimization of this integral is equivalent to the minimization of the Euclidean norm. The calculation of it can be done using the Monte Carlo algorithm which relies on summation instead of the costly integration.

What is Monte Carlo (MC) integration in general?
MC integration is a numerical integration that randomly chooses the points at which the integrand is evaluated. First I will specify the region A to integrate over. To estimate the area of interest D, pick a simple area A which is easy to calculate and which contains D. Then pick a sequence of random points that fall within A. Some fraction of these points will also fall within D. The area of D is then estimated as this

fraction multiplied by the area of A.
As we handle a mixture of projected Gaussians, we know for each kernel PG_i the center of mass D_i and thus easily can deduce an approximation of the center of mass of the mixture. Of course we require the area $A \subset S_3 \times \mathbb{R}^4$ to contain $D_i \, \forall i$.
Now the algorithm has to be defined:

- $\{a_n\}_n,\ n \in \mathbb{N}$ is a random sequence of identically distributed points in the integration area A.
- g is the integrand. For a function of one variable the average value of $g(x)$ can be estimated by:
$$\tilde{g_N} \approx \frac{1}{N}\sum_{n=1}^{N} g(a_n),\ N \geq 1$$
- M_A is the mass of the whole integration area.
- Iteration:
  1. $V_1 = g(a_1)$ is the value of the first point
  2. $V_{n+1} = \frac{n}{n+1} \cdot V_n + \frac{1}{n+1} \cdot g(a_{n+1})$ defines the steps from $n\,(\geq 1)$ to $n+1$
- $V := \lim_{n\to\infty} V_n$
- The value of the integral I is then:
$$\int_A g(x)\,\mathrm{d}M_A = V \cdot M_A$$
and an approximation of the integral can be given by $I \approx M_A \cdot \tilde{g}_N$

An estimate for the error is given by:

$$err = M_A \cdot \sqrt{\frac{\tilde{g^2}_N - \tilde{g_N}^2}{N}}$$

where $\tilde{g^2}_N := \frac{1}{N}\sum_{n=1}^{N} g^2(a_n)$.

On integrating with this algorithm the values converge with order $o(\frac{1}{\sqrt{N}})$ regardless of the smoothness of the integrand. MC integration is not competitive in one or two dimensions, but in higher dimensions. Further keep in mind that each time the MC algorithm is implemented using the same sample size N, it will come up with a slightly different value as the integration points are picked randomly. Obviously larger values of N produce more accurate approximations.

I suppose the iteration can be stopped when the difference $|V_{n+1} - V_n|$ remained sufficiently long under a majorant with sufficiently small sum. Then we assume the algorithm to converge significantly to the true value of the integral I.
The most important advantage of this approximation of the integral is that the algorithm is easy and fast.

The traditional Monte Carlo algorithm distributes the evaluation points uniformly over the integration region like mentioned above. But there are also adaptive algorithms such as VEGAS and MISER:

- **MISER Monte Carlo**
  This algorithm of Press and Farrar [25] is based on recursive stratified sampling. This technique aims to reduce the overall integration error by concentrating integration points in the regions of highest variance.

- **VEGAS Monte Carlo**
  The algorithm of G. P. Lepage [21] is based on importance sampling. It samples points from the probability distribution described by the absolute value of the function $|g|$, so that the points are concentrated in the regions that make the largest contribution to the integral.

We would like to approximate ${\|p_0 - p_{app}\|_2}^2$ by:

$$\frac{1}{N}\sum_{i=1}^{N}(p_0(a_i) - p_{app}(a_i))^2 \cdot M_A$$

where $a_i \in A$ are the elements of the sample set $\{a_1, \dots, a_N\}$ and $A \subset S_3 \times \mathbb{R}^4$ is the region to integrate. It turned out that the specification of the region A lacks an easy solution but we found a possibility to elegantly eludes this specification which I will introduce in the following.

Importance Sampling

We know that the Monte Carlo estimator of $\mathrm{E}[g(X)]$ is $\tilde{g_n}(X) = \frac{1}{n}\sum_{i=1}^{n} g(x_i)$ for X being a uniformly distributed continuous random variable. Furthermore this estimator is unbiased, what means $\mathrm{E}[\tilde{g_n}(X)] = \mathrm{E}[g(X)]$.

An important thing to note is that there is *no restriction* that says that the random variables must be uniformly distributed. It is obvious that the choice of distribution from which to draw the random variables will affect the quality of their Monte Carlo estimator. This implies *importance sampling* [2] is choosing a good distribution from which to simulate the random variables.

Now consider X to be a continuous random variable with any probability density function $f_X(x) > 0 \; \forall x \in \mathbb{R}$. Then the expected value of a function g of X is:

$$\mathrm{E}_{f_X}[g(X)] = \int_{x\in\mathbb{R}} g(x) f_X(x)\,\mathrm{d}x$$

This is deduced from the following:
If X is a continuous random variable defined on a probability space (Ω, Σ, P), then the expected value of X is defined as:

$$\mathrm{E}_{f_X}[X] = \int_{\Omega} X\,\mathrm{d}P$$

When this integral converges absolutely, it is called the expectation of X. If the probability distribution of X admits a probability density function $f_X(x)$ on Ω, then the expected value can be computed as:

$$\mathrm{E}_{f_X}[X] = \int_{-\infty}^{\infty} x f_X(x)\,\mathrm{d}x$$

And the expected value of an arbitrary function of X, $g(X)$, with respect to the probability density function $f_X(x)$ is given by the inner product of f_X and g.

Then we can estimate the value of $\mathrm{E}_{f_X}[\frac{g(x)}{f_X(x)}]$ by generating a number of random samples according to f_X, computing $\frac{g}{f_X}$ for each sample, and finding the average of these values. As more and more samples are taken, this average is guaranteed to converge to the expected value, which is also the value of the integral I.

Definition and Theorem 3.3.2. *Let $f_X(x)$ be a density for a continuous random variable X which this time only takes values in A so that $\int_{x\in A} f_X(x)\,\mathrm{d}x = 1$ and E_{f_X} denotes the expectation with respect to the density f_X:*

$$\int_{x\in A} g(x)\,\mathrm{d}x = \int_{x\in A} g(x)\cdot\frac{f_X(x)}{f_X(x)}\,\mathrm{d}x = \int_{x\in A} \frac{g(x)}{f_X(x)}\cdot f_X(x)\,\mathrm{d}x = \mathrm{E}_{f_X}\left[\frac{g(x)}{f_X(x)}\right]$$

so long as $f_X(x) \neq 0$ for any $x \in A$ for which $g(x) \neq 0$.
From this follows that the Monte Carlo estimator is:

$$\tilde{g}_{n,f_X}(X) := \frac{1}{n}\sum_{i=1}^{n} \frac{g(x_i)}{f_X(x_i)}$$

where $x_i \sim f_X(x)$.

Proof:

$$\begin{aligned}
\mathrm{E}_{f_X}\left[\tilde{g}_{n,f_X}(X)\right] &= \frac{1}{n}\sum_{i=1}^{n} \mathrm{E}_{f_X}\left[\frac{g(x_i)}{f_X(x_i)}\right] \\
&= \frac{1}{n}\sum_{i=1}^{n} \int_A \frac{g(x)}{f_X(x)} f_X(x)\mathrm{d}x \\
&= \frac{n}{n}\int_A g(x)\frac{f_X(x)}{f_X(x)}\mathrm{d}x \\
&= \int_A g(x)\mathrm{d}x \\
&= I
\end{aligned}$$

given that $\frac{g(x)}{f_X(x)}$ is finite $\forall\, x$.

$\square$

Finding a MC estimator that provides good estimates in a reasonable amount of computing time is not a trivial task.
An assessment for MC estimators can be given by the variance [31] which is defined by:

$$\mathrm{Var}[\tilde{g}_{n,f_X}(X)] = \frac{1}{n}\int_{x\in\mathbb{R}} \left(g(x) - \mathrm{E}[g(X)]\right)^2 f_X(x)\mathrm{d}x$$

The smaller the variance for the same amount of computational effort the better the estimator in comparison to its competitors. Thus we are looking for an importance sampling function $f_X(x)$ that has the following properties:

- $f_X(x) > 0$ whenever $g(x) = 0$
- $f_X(x)$ should be close to being proportional to $|g(x)|$
- It should be easy to simulate values from $f_X(x)$.
- It should be easy to compute the density $f_X(x)$ for any value x one might realize.

I want to point out that serious difficulties arise if $f_X(x)$ gets small much faster than $g(x)$ out in the tails. Though drawing a sample from the tail of the distribution is unlikely the MC estimator will give a big error if it occurs. $\frac{g(x_i)}{f_X(x_i)}$ for such an unlikely x_i may be orders of magnitude larger than the typical values of $\frac{g(x_i)}{f_X(x_i)}$.

To estimate the absolute error of the MC integration with importance sampling the central limit theorem can be used. It states that $\tilde{g}_{n,f_X}(X)$ converges to the normal distribution as $n \to \infty$. Let's denote $Y_i := \frac{g(x_i)}{f_X(x_i)}$ and $Y := Y_1$. In particular the central limit theorem gives for $t \in \mathbb{R}$ and $\sigma(Y)$ the standard deviation of Y:

$$\lim_{n\to\infty} \mathrm{P}\left[\frac{1}{n}\sum_{i=1}^{n} Y_i - \mathrm{E}[Y] \le t \cdot \frac{\sigma(Y)}{\sqrt{n}}\right] = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{t} \mathrm{e}^{-x^2/2}\,\mathrm{d}x$$

Hence we receive the following equation for the error:

$$\mathrm{P}\left[|\tilde{g}_{n,f_X} - I| \ge t \cdot \sigma(\tilde{g}_{n,f_X})\right] = \sqrt{\frac{2}{\pi}} \int_{t}^{\infty} \mathrm{e}^{-x^2/2}\,\mathrm{d}x$$

where the standard deviation of $\tilde{g}_{n,f_X}$ is $\sigma(\tilde{g}_{n,f_X}) = \frac{1}{\sqrt{n}}\sigma(Y)$.

By the way I want to bring up that the treatment of higher order error estimation is not just an academic point. Lazopoulos deals in his paper [19] with first-order errors of MC integration that is the error directly on the integral estimate and second-order errors that is the error on the error estimate of the integration. A mis-estimate of the integration error can lead to a serious under and over estimate of the confidence level and thus it's estimation should be done carefully.

Finally I want to explain why for Monte Carlo integration with importance sampling over mixtures of Gaussians as distribution function no limits of integration are needed. Let's calculate the MC estimator $\tilde{g}_{n,f_X}(X)$ for n samples of a mixture of projected Gaussians with length d with a set of normally distributed samples $\{x_1, \dots, x_n\}$. We define the fraction

$$\begin{aligned}
\frac{g(x)}{f_X(x)} &:= \sum_{i=1}^{d} \frac{g_i(x)}{f_{X,i}(x)} \\
&= \sum_{i=1}^{d} \frac{\lambda_i \cdot 1/C(x) \cdot 1/\sqrt{\det(2\pi\Sigma_i)} \cdot \exp(-\frac{1}{2}(x-\mu_i)^\top \Sigma_i^{-1}(x-\mu_i))}{1/\sqrt{\det(2\pi\Sigma_i)} \cdot \exp(-\frac{1}{2}(x-\mu_i)^\top \Sigma_i^{-1}(x-\mu_i))} \\
&= \sum_{i=1}^{d} \lambda_i \cdot 1/C(x)
\end{aligned}$$

where $1/C(x)$ is the correction weight for the parameterization in the integration. This means for $x = (x_1, x_2, x_3, x_4, x_5, x_6)^\top \in \mathbb{R}^6$ it is defined as $1/C(x) = 1/(1+x_1^2+x_2^2+x_3^2)$ as I already showed at the end of section 2.2.1. For all $i = 1, \dots, d$ we know that $f_{X,i}(x)$ will never be smaller than $g_i(x)$ as $\lambda_i \leq 1$ and $1/C(x) \leq 1 \, \forall x \in \mathbb{R}^6$. Thus there is no risk for abnormal behavior in the tails of the probability density. As we know from theorem 3.3.2 it holds that $I = \mathrm{E}_{f_X}\left[\tilde{g}_{n,f_X}(X)\right]$. Hence we can calculate an approximation of the integral by the following formula:

$$I \approx \sum_{i=1}^{d} \lambda_i \frac{1}{n} \left(\sum_{j=1}^{n} \frac{1}{C(x_{i_j})} \right)$$

where x_{i_j} is the jth element of the $\varphi(\mu_i, \Sigma_i)$ distributed sample set with $\varphi(\mu_i, \Sigma_i) := 1/\sqrt{\det(2\pi\Sigma_i)} \cdot \exp(-\frac{1}{2}(x-\mu_i)^\top \Sigma_i^{-1}(x-\mu_i))$.

Formally we would have to pick the samples x_{i_j} out of a box shaped integration area A including the area of interest D and than let the side length of the box go to infinity always normalizing with the density of the underlying sample distribution. But it can be seen directly that each of the summands of the mixture and thus the whole mixture itself does not contain significant mass in the tails as we just work with finite mixtures of projected Gaussians.

What stands out to be done in future work is the minimization of the square of the L^2 norm of the densities of the mixtures M_0 and M_{app}:

$$h(\lambda_i, TS_i, \mu_i, \Sigma_i) := \min \|p_0 - p_{app}\|_2^2$$

By now the Monte Carlo integration can at least be used to validate the fit of a mixture achieved with the expectation maximization algorithm.

4. Implementation and Experimental Verification

Recall that a robot makes several localization attempts to estimate the pose of a target object. In the following I will explain how the robot draws conclusions from the separated 3D SIFT features it detects to the object pose. This procedure is called sensor model.

- Every object of the robots data base of 3D models is given a Cartesian coordinate system CS_O. To systematize the arbitrary choice of origin and axis, we postulate the origin of the coordinate system to be the midpoint of the bottom of the object. Then we define the x- and y-axes to be the main axes in the basement of the object the way that together with the z-axis, which is straight up, they produce a right-handed coordinate system. Now any point feature on the objects surface can be described by a 3-dimensional position and an orientation in 2 dimensions similar to the description in [24].

- Each camera or 3D sensor of the robot has a normalized coordinate system CS_C which we define the way that the viewing direction equals the z-axis. As we want the coordinate system to be a right-handed Cartesian one, the other axes are determined on claiming the x-axis to point to the right from viewing direction and thus the y-axis points down.

- If the robot detects a feature with its camera, a new Cartesian coordinate system CS_F for the feature has to be defined. CS_F has its origin at the mean of the estimation for the features pose. To take into account that the most likely hypothesis is frontal perspective to the feature we define the z-axis to point to the origin of CS_C. As features have an orientation on the locally planar objects surface, the x- and y-axes are predefined through the 3D object model and the orthogonality to the z-axis.

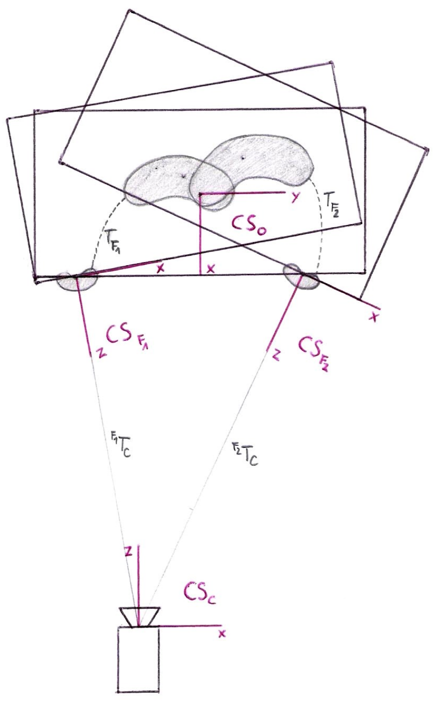


Figure 4.1.

Let ${}^{F}T_C : \mathrm{CS}_C \rightarrow \mathrm{CS}_F$ be the transformation from the camera coordinate system to the feature coordinate system. As I create the mixture that models the distribution of the feature in camera coordinates, I have to change the coordinate system to represent the feature at its true pose. As mentioned already we have the 3D model of the object and can describe any feature on the object by the function T_O. But to grasp the object, we need to model the object pose by knowing the feature. The inverse of the function T_O is the transformation $T_F = (T_O)^{-1}$ that lets us draw conclusions from the feature pose on the object's surface about the object pose.
As we represent any rigid motion by a rotation matrix and a translation vector or equivalently by a dual quaternions, the inverse of the dual quaternion represents exactly the inverse rigid motion.

Features have a predefined orientation and thus the possible rotation around the z-axis is very small, in contrary around the x- and y-axis I allowed as uncertainty rotations in the interval $[-15°, 15°]$. We know about the translation that there can be little shifting in the x-y-plane, but as the scale of the feature also is uncertain, the dislocation in direction of the z-axis can be fairly big. Hence we need a mixture of 6D projected Gaussians MoPG_0 to model the pose of the feature the robots camera detected. This model now has to be shifted to the pose, where the feature was detected by the base element $\mathrm{PG}_1 = {}^{F}T_C$.
As the 3D models of the objects are imperfect, one has to calculate with little uncertainties in position and orientation of the feature on the objects surface. Thus another 6D projected Gaussian $\mathrm{PG}_2 = T_F$ is required.

Now the distribution of the object's pose can be estimated in camera coordinates by:

$$\mathrm{PG}_1 \circ \mathrm{MoPG}_0 \circ \mathrm{PG}_2$$

Of course a single feature is not sufficient to determine the pose of an object. In reality about 50 features on one taget object are needed to estimate the pose precisely enough.

4.1. Python Code

Now I want to give a brief documentation about the code I wrote to create the framework. As Python is a object oriented programming language I structured the system in mainly five object classes. These are:

- MoPG_tangentSpace
- MoPG_baseElement
- MoPG_mixture
- Quaternion
- DualQuaternion

A tangent space consists of the tangent point p on the (hyper-)sphere surface and a basis B of the tangent space in world coordinates. That means if the point $p \in \mathbb{R}^d$, the basis is a $d \times (d-1)$ matrix, that is completed to a basis of the d-dimensional space by concatenating the vector p as first column.

The class **MoPG_tangentSpace** contains several functions. *_init_* always is the first method of a class and creates a representative. Furthermore *equal* tests whether two tangent spaces are equal and *display* changes the tangent space to a printable formate on the display.
The method *tangentSpaceToSphereCentralProjection* projects a 3-dimensional vector in the tangent space by central projection to the sphere whereas *sphereToTangentSpace-CentralProjection* projects any point on the sphere surface to any given tangent space except for the case that the tangent point and the point on the sphere to be projected have an angle of $\pi/2$ between themselves. With *transformFromSelfToTS* the tangent space can be changed. Therefore a vector v in the first tangent space with tangent point p_1 is projected to the sphere and then is back projected to the second tangent space with tangent point p_2. In these methods the translational part remains unchanged and can be inputted to the function also.

Finally this class has the method *poseTransformationTS* which transforms a 6-dimensional vector consisting of a rotation and a translation part with a given tangent space by a vector with its appropriate tangent space to another 6D vector in a third tangent space.

A base element is a projected Gaussian consisting of a tangent space, like defined above, the mean value of the gauss kernel and the appropriate covariance matrix. The mean value is a vector with dimension $d-1$ for the rotation part if the tangent point on the sphere is d-dimensional.

The class **MoPG_baseElement** also contains the methods *_init_*, *equal* and *display*. Moreover I wrote a function *dimensions* to determine the dimensions of the rotation and the translation part. *computeMassMonteCarlo6D* computes the mass of the base element by using Monte Carlo integration with importance sampling. It is needed for renormalization.
Any base element contains information about orientation and position in the special Euclidean group which can also be represented by a dual quaternion. This pose is calculated by *extractDualQuaternion*. *changeTS* changes the tangent space of a base element by falling back on the tangent space method *transformFromSelfToTS*. The method *mahalanobisDistance* determines the Mahalanobis distance between two base elements. Furthermore I implemented the functions *fuse*, *merge* and *randomPoseTransformation* which fuse, merge and compose base elements respectively.
If it is desired to obtain the mean vector $\mu_3 = 0$ for the fused respectively merged base element, the 'modeFlag' has to be set to 1.
density is a method that returns the density of a point which has a 4D rotation part already projected to the sphere and a 3D translation part. If the point is close to the equator (for the tangent point being a pole) the back projected point to the tangent space gets to infinity at least in one dimension. Then we set the density to be 0.
Finally the methods *draw1Sample*, *draw1SampleMat*, *drawNSamples* and *paintCS* all sample from the distribution of the base element and are needed for visualization.

MoPG_mixture denotes the class of mixtures of projected Gaussians. Each mixture consists of a list of base elements, each with its weight i.e. it is an array with two columns:

$$\text{Mixture} = [[\text{PG}_1, \lambda_1], [\text{PG}_2, \lambda_2], \ldots, [\text{PG}_\text{n}, \lambda_n]]$$

The methods of this class are *_init_* and *equal* like in the other classes, but instead of *display*, this class has the function *toList* what changes the mixture into a list which

is printable and has all list features implemented in Python, but does not give a reasonable output on the display.
The method *density* calculates the mixture density of a point by using the base element method of same name. The other methods *computeMassMonteCarlo6D*, *fuseMoPG* and *randomMoPGTransformation* also just fall back to the corresponding methods for base elements and apply them on each entry of the mixture. Furthermore I want to mention that the renormalization constant C of the function *randomMoPGTransformation* just consists of the weights λ_i and λ_j. That means $C = 1/(\sum_{i,j} \lambda_i * \lambda_j)$ as there is no question of whether two base elements can be applied at the same time, because the probability distributions are assumed to be independent.
drawNSamples, *drawNSamplesV* and *paint* are used for visualization of the mixture and sample from each of its base elements.

The class of **quaternions** is well known in algebra, but the implemented module in Python is not structured well. Therefore I wrote the code for my own class of quaternions with the following methods:

- *_init_()* makes a quaternion out of a list with four entries [a,b,c,d].
- *norm()* calculates the norm of a quaternion whereas *norm2()* gives the square of the norm
- *normalize()* and *normalizeD()* normalize the quaternion. The first method has a copy as output, the second one is destructive on the input quaternion
- *conj()* and *conjD()* conjugate the input quaternion and give it back in copy or destructive.
- *toList()* prints the quaternion to a list to receive a readable output.
- *inv()* and *invD()* calculate the inverse of a quaternion (copy and destructive version)
- *equal()* tests whether two quaternions are equal.
- *copy()* creates a copy of the input quaternion.
- *plus()* adds two quaternions: $Quat1 + Quat2 = Quat1.plus(Quat2)$
- *scalar()* multiplies a scalar $\lambda \in \mathbb{R}$ to a quaternion: $\lambda * Quat = Quat.scalar(\lambda)$

- *times()* multiplies one quaternion Quat1 with another quaternion Quat2: $Quat1 * Quat2 = Quat1.times(Quat2)$
- *toMatrix()* returns the 3×3 rotation matrix of a quaternion.
- *randomUnit()* creates a unit quaternion randomly from an equal distribution on S_3 whereas *randomImaginary()* creates an imaginary quaternion randomly.
- *radAxis()* returns a unit quaternion corresponding to a rotation by radian measure around a given axis, *degreeAxis()* does the same with degree.

The **dualQuaternions** are the last class I created to complete the framework. They have the corresponding methods:
init(), *plus()*, *equal()*, *copy()* and *toList()*.
times() multiplies two dual quaternions by the multiplication defined in 2.1.2 whereas *conjQuat()* returns the quaternion conjugate, *conjDual()* returns the dual conjugation and *conjTotal()* the total conjugation. These conjugations are also defined in 2.1.2. Because every dual quaternion describes a rigid motion it contains a rotation and a translation which can also be written as rotation matrix and translation vector. That does the method *transformationMatrix()*. Finally *inv()* calculates the inverse of the input dual quaternion.

Furthermore there exist the following functions which are not related to any class:

- *computeJacobian* calculates the Jacobian from the tangent space method *transformFromSelfToTS* at the point given as input to the function.
- *rottransVector6D* returns a 6D vector that contains the information about rotation and translation, not in quaternion style, but in a quite normal 6D vector.
- *makeSamples* creates a given number of samples which fulfill special requirements like the distribution where they are drawn from.
- *makeInitMixture* makes a mixture out of a list of samples.
- *rotToQuat* is a function that returns the quaternion which represents the same rotation like a given rotation matrix.
- *transformationToDQ* changes the input of a tuple of quaternions q_r and q_t to a dual quaternion with dual part $q_d = 1/2 \cdot q_t * q_r$.

- *DQToTransformation* makes the inverse transformation from a dual quaternion to a tuple of quaternions.
- The function *transformPose* returns a dual quaternion which is obtained after applying a transformation dual quaternion to a pose dual quaternion.

4.2. Visualization

We have a range of options introduced to improve the pose estimation like weighting the measurement results, fusing, merging, dropping base elements and making another localization attempt. An algorithm shall evaluate what to apply to the distribution function describing the object pose to improve it and when the distribution is peaked enough for that a grasp criterion is fulfilled. For this framework it stands out to define such an evaluation algorithm. A possible approach is the further development of the MC-SOPE algorithm Glover introduced in [13]. This algorithm to solve the single object pose estimation uses importance sampling to generate a weighted set of pose samples of the target distribution and then returns the top n samples ranked by weight. We would need to sample from the target distribution that is achieved after applying different options like the modification operations and then compare the distributions fitted to the resulting sample sets.

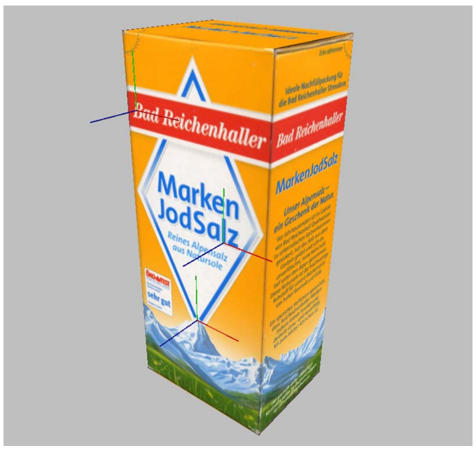


Figure 4.2.

In the following I describe a simulated experiment how such an algorithm might proceed. The programming language Python offers a Coin binding to enable visualizations like the ones I made. [32] explains how to write applications using the Open Inventor toolkit in this Coin binding.
Imagine a robot that has the task to grasp the salt box of figure 4.2. Let's assume that the robot detects the features 'B' of the word *Bad* and 'l' of *Salz*. Later on I will also work with the mountain top as a third feature.

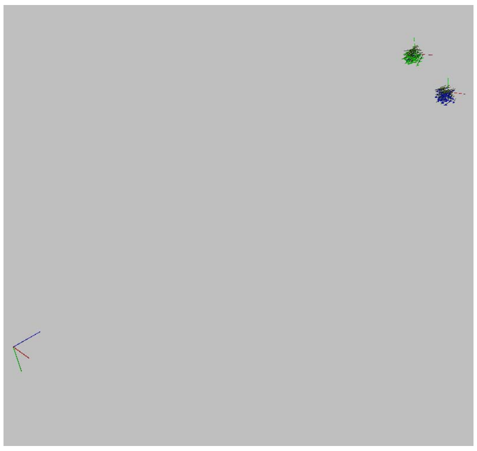


Figure 4.3.

The flags represent samples out of the mixture describing the pose of the features in $S_3 \times \mathbb{R}^3$. The most likely hypothesis is that the robot detects features which are frontal to the camera and thus we model the probability distribution as represented in figure 4.3. The green flags represent the 'B' and the blue flags the 'l'. Further the coordinate system on the left is the camera coordinate system CS_C.

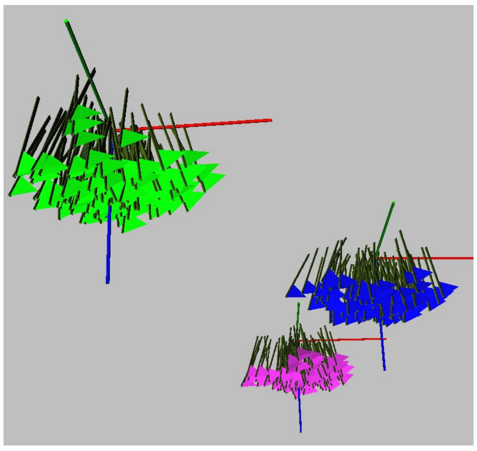


Figure 4.4.

As mentioned already I assume that the the robot will detect the feature mountain top on the salt box as well. This feature will be modeled by the pink samples drawn from the mixture of projected Gaussians describing the probability distribution of the pose of the mountain top.

The figure 4.4 shows the sample sets in a view from above the salt box.

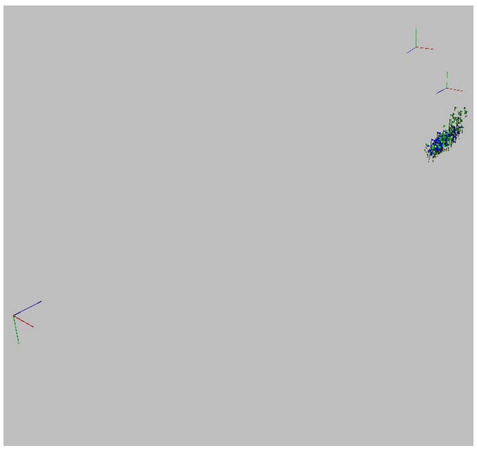


Figure 4.5.

In figure 4.5 the green and blue samples are drawn from the mixture distribution describing the pose of the object. The result is what we get on drawing conclusions from the feature pose to the pose of the object.

For easier orientation again the coordinate system of the camera can be seen in the the lower left corner of the figure as well.

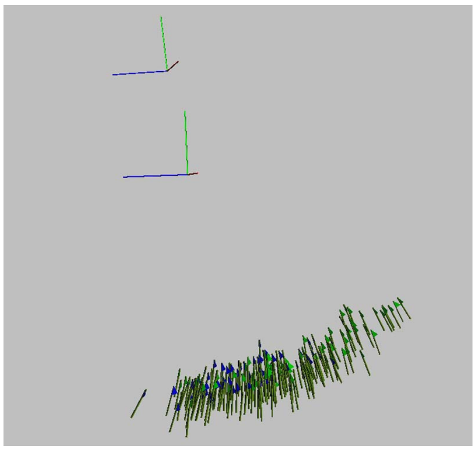


Figure 4.6.

The coordinate systems which are located at the position of the mean vectors of the distribution describing the poses of the features 'B' and 'l' are rotated the way that the z-axes point to the camera. Further figure 4.6 shows the big variance in direction of the z-axis of the camera which equals the viewing direction of the camera.

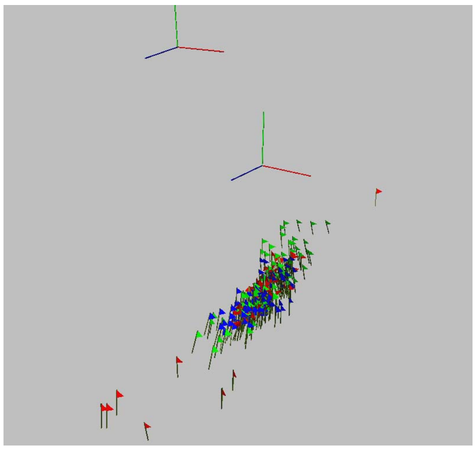


Figure 4.7.

Now we apply the modification operation fusion to the green and the blue mixture. As both distribution functions provide reasonable sample sets this is an ordinary step of the evaluation algorithm. Figure 4.7 shows the new red sample set which is obtained after fusing the other mixtures. The adjustment of the weights which are used in the operation of fusing might be improved to avoid scattering of the red samples.

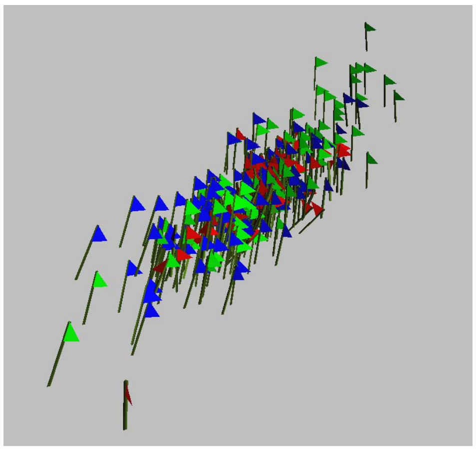


Figure 4.8.

Through this operation the number of elements of the mixture grew to 49 as the blue and the green mixture consisted of 7 kernels each. Many summands of the mixture have low weight but scatter and thus disturb the evaluation algorithm. This is the reason why I decided to drop 10 kernels with low weight. Figure 4.8 shows that as result the red samples become more concentrated.

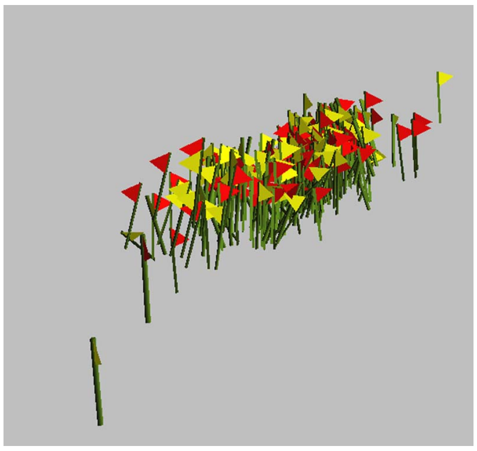


Figure 4.9.

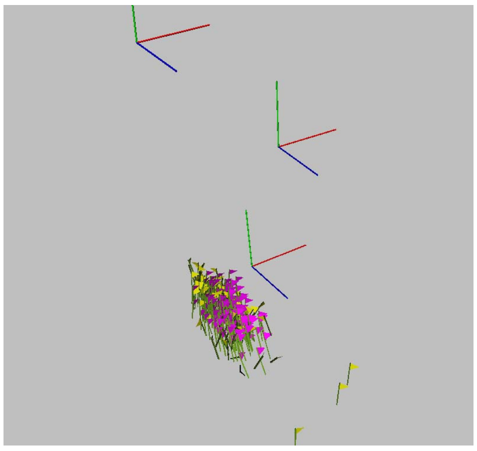


Figure 4.10.

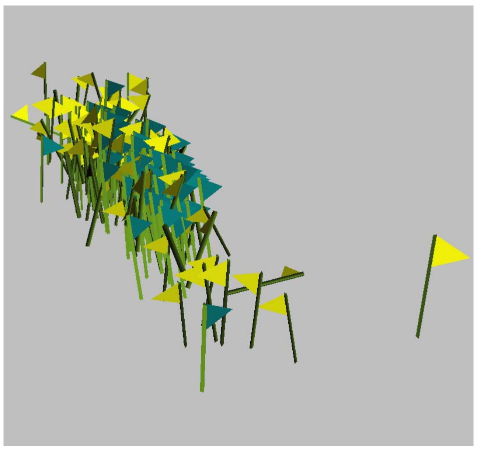


Figure 4.11.

In figure 4.9 the red samples still represent the fused and reduced mixture which is obtained already in the figure before. The yellow sample set are drawn from the mixture after merging the mixture down until the number of base elements that remain is 10. This smaller mixture is computationally efficient and doesn't seem to lack any information. That the flags are slightly different comes from the fact that the sample points are chosen randomly.

As the results we got by now seem reliable we will go on with the yellow mixture we obtained after the modification operation of mering. An appropriate next step for the evaluating algorithm would be to make a new localization attempt to get more information data. Figure 4.10 shows the pink samples which are drawn from the mixture one obtains from drawing conclusion from the feature pose of the mountain top to the object pose.

Figure 4.11 shows the turquoise sample set of the mixture which is obtained after fusing the pink and the yellow mixtures and further dropping the irrelevant base elements with low weights as well. This mixture still consists of 36 base elements.

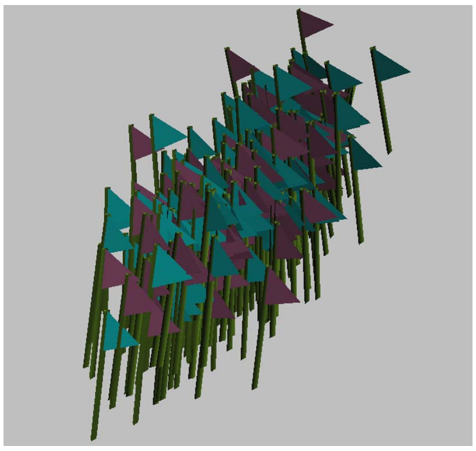


Figure 4.12.

A remarkable effect of making the new localization attempt is that the fused sample set is way better ordered in orientation than before. It is seen in figure 4.12 that from merging the turquoise mixture down to the new purple mixture consisting of 10 summands no loss of information arises. The mixture is strongly peaked already and estimates the true pose of the salt box in $S_3 \times \mathbb{R}^3$.

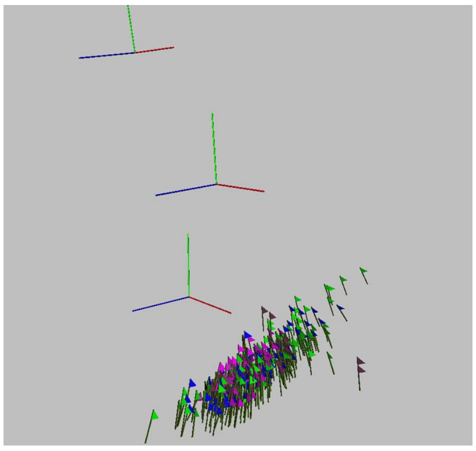


Figure 4.13.

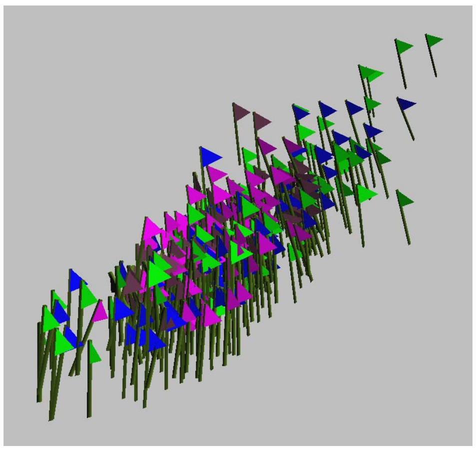


Figure 4.14.

Finally the last two figures 4.13 and 4.14 show the distribution of the object pose that is estimated directly from the three features represented by the green, blue and pink sample set and the distribution after simulating an evaluation algorithm. The purple sample set containing all information data fits the center of mass of the other mixtures and furthermore provides a peaked distribution function for the pose estimation.

From this example one gets a sense of how the evaluation algorithm should work. The steps I introduced need to be repeated until a stop criterion is reached, like i.e. that a grasp criterion is fulfilled. If no information gain can be registered any more possible further steps might be to involve other kinds of features like edge extraction or to try a new localization attempt from another side.

5. Outlook

5.1. Result

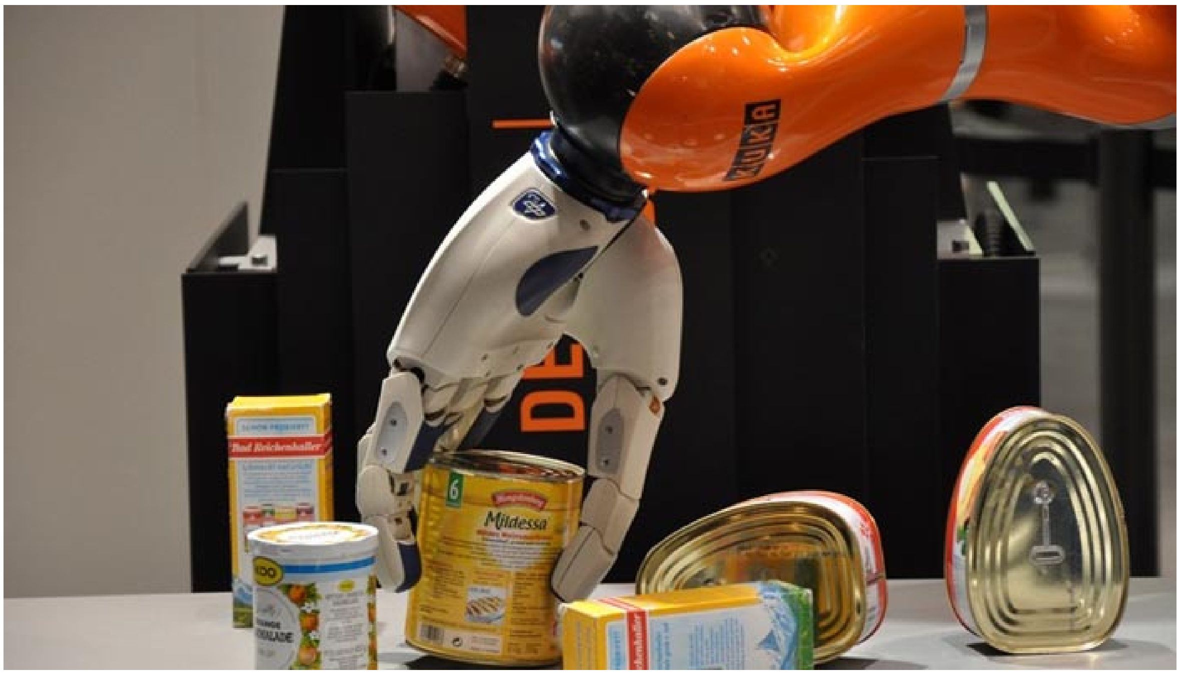


Figure 5.1.: The picture shows the robot of the DESIRE project succeeding to grasp the target object after having identified it on the CeBIT in March 2009.

5.2. Improvements

- In 2.2.1 Projected Gaussian I explain how a basis B_0 of the $\mathbb{R}^4$ can be calculated in a canonical way. I guess that there is also a canonical way to construct a basis over any odd dimension. This stands out to be proven.

- It would be nice to implement a variation of our framework using the Bingham distribution to compare the performance and the time needed for the computations. Further the accuracy of the modeling by Binghams and projected Gaussians stands out to be compared. The composition of Bingham distributions doesn't give another Bingham but can be approximated by one. The result of this approximation should be compared with a composed projected Gaussian distribution, as well.

- By now I can't constitute which dissimilarity measure is appropriate for specific applications. Experiments are necessary to get informations about what grasp criterion and which distance measure should be applied.

- It remains to check whether the KL divergence or the sKL divergence is faster to calculate. As in

  $$B(i,j) = \frac{1}{2}\left((\lambda_i + \lambda_j)\log\det(\Sigma_{ij}) - \log\det(\Sigma_i) - \log\det(\Sigma_j)\right)$$

  one has to calculate three logarithms the computing time is very slow. For that reason I assume that the much longer expression $B_s(i,j)$ might be slightly faster. To verify this the number of necessary operations need to be determined for each formula.

- The studies of the coherence between particle sets and MoPGs should be finished. This means the comparison of the results one achieves by applying a modification operation to a mixture and to a particle set.

- The approaches to reduce the number of elements of a mixture introduced in 3.3 need to be developed and a check for the accuracy of the results has to be implemented.

- In future work experimental results would be desirable. Therefore the evaluation algorithm has to be developed. Then further simulated experiments are necessary as well as real world experiments to validate the truth of the algorithm.

- The implemented framework was planned to be open to any dimensions but in the course of the work I realized that this requires too many case-by-case analysis. Thus I decided to implement a part of the functions just for the 4-dimensional case. This could be universalized.

A. Appendix

In A.1 the complete Python code is appended usually including the input and output format for each function. Except in the obvious case.

A.1.

For the code the following conventions were made:

- $(\cdot, \dots, \cdot)$ denotes vectors
- $[\cdot, \cdot, \cdot, \cdot,]$ stands for quaternions and lists
- $[\cdot, \cdot] = [[\cdot, \cdot, \cdot, \cdot,], [\cdot, \cdot, \cdot, \cdot,]]$ denotes a dual quaternion and
- $[[\cdot, \cdot], \dots, [\cdot, \cdot]]$ is a mixture of base elements or equivalently a double list

Further the following modules need to be installed and imported:

- numpy
- numpy.linalg
- numpy.random
- scipy
- math
- pyviblib.calc.common
- scipy.integrate.quadpack

The following code consists of auxiliary functions that don't belong to any specific class. These are basic rules that are needed for the methods of the classes I introduce already in 4.1.

```
def makeVector(list):
    "the input is a d dimensional list"
    pt = matrix(list)
    return pt.T

def makeBase(tangentPoint):
    """tangentPoint is a d dimensional vector (so if you have a list do
    makeVector(tangentPoint))."""
    "A point on the sphere surface."
    pl = tangentPoint.tolist()
    dim = len(tangentPoint)
    if dim == 4: #make a canonical tangent space
        c = list(flatten(pl))
        B = matrix([[-c[1], -c[2], -c[3]],
                    [c[0], -c[3], c[2]],
                    [c[3], c[0], -c[1]],
                    [-c[2], c[1], c[0]]])
        return B
    else: #create a random base
        n = 1
        B = tangentPoint
        while (n < dim):
            arr = random_integers(-99, 99, (dim, 1))
            B_test = concatenate((B, arr), axis = 1)
            #join the array to the matrix T = [B,arr]
            rk = matrixrank(B_test) #calculate the new rank of B
            if rk == n+1:
                B = B_test
                n = n+1
        basis = array(B)
        basis_orthogonal = orthogonalize_set_my_version(basis,
                                                        make_orthonormal
                                                        = True)
        mat = matrix(basis_orthogonal)
        #der erste vektor wird weggelassen
        z = zeros((dim, dim-1), float)
        for i in range(dim):
            for j in range(dim-1):
                z[i][j] = mat[i, j+1]
        base = matrix(z)
        return base

def flatten(lst):
    for elem in lst:
        if type(elem) in (tuple, list):
            for i in flatten(elem):
                yield i
        else:
            yield elem

def drange(start, stop, step):
    r = start
    ret = []
    while r < stop:
        ret.append(r)
        r += step
    return ret

def matrixrank(A):
    tol=1e-8
    s = svd(A,compute_uv=0)
    #print sum(where(s>tol,1,0))
    return sum(where(s>tol,1,0))

def orthogonalize_set_my_version(set_in, make_orthonormal=True):
    """As the original function couldn't write in the 'set_out',
    I modified the function.
    Orthogonalize a set of vectors using the Gram-Schmidt algorithm.
    Positional arguments :
    set_in: vectors to be orthogonalized (threes-dimensional ndarray)
            their base is given by the keyword argument of the same name
    Keyword arguments :
    set_out: where the result is to be placed (default None) unless
             given, use set_in
             the caller is responsible for memory allocation
    make_orthonormal:  whether the result set is to be normalized
    base:  base index (default 1)"""
    base = 0
    if not isinstance(set_in, ndarray) or 2 > len(set_in.shape):
        raise InvalidArgumentError('Invalid set_in argument')
    set_u = zeros(set_in.shape    , 'd')
    sum_  = zeros(set_in.shape[1:], 'd')
    # u1 = v1
    set_u[base] = set_in[base].copy()
    for i in xrange(1 + base, set_in.shape[0]) :
        sum_ *= 0.
        for j in xrange(base, i) :
            ujvi = contract(set_u[j], set_in[i])
            ujuj = contract(set_u[j], set_u[j])
            if 0. != ujuj :
                sum_ += (ujvi / ujuj) * set_u[j]
        set_u[i] = set_in[i] - sum_
    # orthonormalize if necessary
    if make_orthonormal :
        normalize_set(set_u, set_out=None, base=0)
    return set_u

def func(x1,x2):
    "Weightening function"
    "The input has to be two vectors x1 and x2."
    #Funktion, die sicherstellen soll, dass die Gaussverteilungen
    #wirklich die gleiche versteckte Variable beschreiben.
    prod = (x1.T)*x2
    prodList = prod.tolist()
    if prod > 1+1e-4:
        print 'The scalar product is bigger than 1'
    elif prod > 1:
        return 1.0
    else:
        valueAcos = math.acos(prodList[0][0])
        valueExp = -5*(valueAcos)**2
        valueTotal = math.exp(valueExp)
        return valueTotal

def id(x1,x2,x3,x4,x5,x6):
    return [x1,x2,x3,x4,x5,x6]

def makeMatTransInput(rotMat, transVec):
    """Input has to be a matrix and a vector.
    The rotation matrix is 3x3 and also the vector is a 3x1 matrix.
    The output of the function is a matrix: [[R,t],[0,1]]"""
    #Needed as the format known by INVENTOR.
    vec = transVec
    vec = concatenate((vec,array([[1]])),axis = 0)
    vec = array(vec)
    vec = list(flatten(vec))
    vecarr = array(vec)
    arr = array([[0,0,0]])
    #rotMat = rotMat.T
    value = concatenate((rotMat,arr),axis = 0)
    value = concatenate((value,vecarr),axis = 1)
    value = matrix(value)
    #value = value.T
    #value = value.tolist()
    #value = list(flatten(value))
    return value

def removeEl(li, numelem):
    assert(numelem < len(li))
    li1 = li[:numelem]
    li2 = li[numelem+1:]
    li1.extend(li2)
    return li1
```

```
def rottransVector6D(value, theta, axis, transpoint):
    """This function creates a 6D vector that contains the information
    about rotation and translation not in quaternion style, but in a
    quite normal 6D vector s. The tangent space point is a 4D point on
    the unit sphere: value, a vector.
    Theta is the angle to rotate, axis is the axis to rotate around and
    transpoint is the translation part."""
    a = math.cos(theta/2)
    th = math.sin(theta/2)
    axis = axis/sqrt(axis[0]**2 + axis[1]**2 + axis[2]**2)
    b = th*axis[0]
    c = th*axis[1]
    d = th*axis[2]
    p1 = transpoint[0]
    p2 = transpoint[1]
    p3 = transpoint[2]
    point = [a, b, c, d, p1, p2, p3]
    TS = tangentSpace(value)
    rottrans = TS.sphereToTangentSpaceCentralProjection(point)
    rottrans = list(flatten(rottrans.tolist()))
    return rottrans

def computeJacobian(point,tangentSpace1,tangentSpace2):
    "Input point has to be a list."
    if isinstance(tangentSpace1,tangentSpace) == True:
        if isinstance(tangentSpace2,tangentSpace) == True:
            T1 = tangentSpace1
            T2 = tangentSpace2
            dim = len(point)
            J = zeros((dim,dim), float)
            h = 0.1e-6
            I = eye(dim) #Einheitsmatix
            # i Zeilen und j Spalten der Matrix
            for j in range(dim):
                z = I[j]
                z = z*h
                pointplus = point+z
                lpp = list(flatten(pointplus))
                pointminus = point-z
                lpm = list(flatten(pointminus))
                transformpointplus = T1.transformFromSelfToTS(lpp, T2)
                transformpointminus = T1.transformFromSelfToTS(lpm, T2)
                difference = transformpointplus - transformpointminus
                qj = difference/(2*h)
                qjlist = qj.tolist()
                qjlist = list(flatten(qjlist))
                for i in range(dim):
                    qij = qj[i]
                    J[i][j] = qij
            #print type(J)
            return J
        else:
            print 'Second argument is not a tangentSpace.'
    else:
        print 'First argument is not a tangentSpace.'

def makeInitMixture(samples):
    """ make a mixture with a kernel for each sample
    """
    nElem   = len(samples)
    mixlist = []
    for sampleCol in samples:
        sample  = sampleCol[0]
        col     = sampleCol[1] # will not be used here
        tanpt   = makeVector(sample[:4])
#         cov     = 0.00001*matrix(eye(6))
        cov = matrix([[ 0.01,  0.0,   0.0,   0.0,   0.0,   0.0],
                      [ 0.0,   0.02,  0.0,   0.0,   0.00,  0.0],
                      [ 0.0,   0.00,  0.04,  0.00,  0.0,   0.0],
                      [ 0.0,   0.0,   0.0,   0.000001,   0.0, 0.0],
                      [ 0.0,   0.00,  0.0,   0.0,   0.000001, 0.0],
                      [0.0,    0.0,    0.0,    0.0,    0.0, 0.000001]])
        comp = [baseElement(tangentSpace(tanpt),
                            makeVector([0,0,0]+sample[4:7]),cov ,2000),
                            1.0/float(nElem)]
        mixlist.append(comp)
    return mixture(mixlist)

def makeSamples(numberOfSamples, spec, params):
    """spec is a string that specifies which kind of samples should be
    created. This is kind of open ended. The first one could be just
    equally distributed over the unit quaternions and a box of
    translations"""
    samples  = []
    # make a quaternion to derive samples from
    rootquat = quaternion(0,0,0,0)
    for sampleIdx in range(numberOfSamples):
        if spec == 'equalOnQuatAndBox':
            'The box is given as [[lowx,hix],[lowy,hiy],[lowz,hiz]].'
            rotparams   = [random.uniform(-1,1) for idx in range(4)]
            norm        = sqrt(sum([rotparams[idx]*rotparams[idx] for
                                  idx in range(4)]))
            rotparams   = [1/norm*rotparams[idx] for idx in range(4)]
            transparams = [random.uniform(params[idx][0], params[idx][1])
                           for idx in range(3)]
            sample      = rotparams + transparams
#            samples.append(sample)
        elif spec == 'equalRotationXYNormalRotationZ':
            """a normally distributed rotation around Z ,sigma=params[0],
            followed by an equally distributed rotation around an axis in
            the x-y plane, combined with translation in the new direction
            of z (i.e. first translate, then rotate)"""
            zrad    = random.normal(0.0,params[0])
            rqz     = rootquat.radAxis(zrad, [0,0,1])
            rvecrad = random.uniform(0.0,2*pi)
            rvec    = [cos(rvecrad),sin(rvecrad),0]
            xyrad   = random.uniform(0.0,params[1])
            rqxy     = rootquat.radAxis(xyrad, rvec)
            rq       = rqxy.times(rqz)
            sample   = rq.toList() + [0,0,0]
        elif spec == 'siftReferredToObject':
            """the distribution of object pose derived from one SIFT
            feature:
            params[0]     sigma of normal distribution around z axis
            params[1]     visibility angle
            params[2]     sigma of x-y-offset
            params[3]     mean value of z-offset
            params[4]     sigma of z-offset
            params[5:9]   rotation quat for feature in object cs
            params[9:12]  translation for feature in object cs
            a normally distributed rotation around Z ,sigma=params[0],
            followed by an equally distributed rotation around an axis in
            the x-y plane, combined with translation in the new direction
            of z (i.e. first translate, then rotate)"""
            zrad     = random.normal(0.0,params[0])
            rqz      = rootquat.radAxis(zrad, [0,0,1])
            rvecrad = random.uniform(0.0,2*pi)
            rvec     = [cos(rvecrad),sin(rvecrad),0]
            xyrad    = random.uniform(0.0,params[1])
            rqxy     = rootquat.radAxis(xyrad, rvec)
            rqf      = rqxy.times(rqz)
            tx       = random.normal(0.0,params[2])
            ty       = random.normal(0.0,params[2])
            tz       = random.normal(params[3],params[4])
            tqf      = quaternion(0.0, tx, ty, tz)
#            sample   = rqf.toList() + [tx, ty, tz, 2]
#            samples.append(sample)
            dqf      = transformationToDQ([rqf,tqf])
            dqfi     = dqf.inv()
            #turn the description of the feature pose in object CS
            #into a dq
            qrfo     = quaternion(params[5],params[6],params[7],params[8])
            qtfo     = quaternion(0.0,params[9],params[10],params[11])
            dqfo    = dualQuaternion(qrfo,qtfo)
            dqof    = dqfo.inv()
            dqr     = dqf.times(dqof)
            rottr   = DQToTransformation(dqr)
            qrr     = rottr[0]
            rrlst   = qrr.toList()
            qrt     = rottr[1]
            rtlst   = qrt.toList()
            rtlst   = rtlst[1:]
        elif spec == 'MoPG':
            # params[0] has the weights - they're renormalized, so don't
            # worry about the sum
            weights = params[0]
            # params[1] gives the range for cov and mean of rot and
            # trans this implicitly tells us the number of elements
        else:
            print 'this sample specification is not supported'
            break
        samples.append([sample, 1])
    return samples

def rotToQuat(r):
    """returns a unit quaternion that represents the rotation given by
    the rotation matrix r. The input matrix has to be a 3x3 rotation
    matrix."""
    diff =  flatten((r*r.T-eye(3)).tolist())
    if max([abs(el) for el in diff]) > 10**(-4):
        print r
    else:
        a2 = (1 + r[0, 0] + r[1, 1] + r[2, 2])/4.0
        min = 0.25
        if(a2 >= min):
            a = sqrt(a2)
            b = 1/4.0*(r[2, 1] - r[1, 2])/a
            c = 1/4.0*(r[0, 2] - r[2, 0])/a
            d = 1/4.0*(-r[0, 1] + r[1, 0])/a
        else:
            b2 = a2 - 1/2.0*(r[1, 1] + r[2, 2])
            if(b2 >= min):
                b = sqrt(b2)
                a = 1/4.0*(r[2, 1] - r[1, 2])/b
                c = 1/4.0*(r[0, 1] + r[1, 0])/b
                d = 1/4.0*(r[0, 2] + r[2, 0])/b
            else:
                c2 = a2 - 1/2.0*(r[0, 0] + r[2, 2])
                if(c2 >= min):
                    c = sqrt(c2)
                    a = 1/4.0*(r[0, 2] - r[2, 0])/c
                    b = 1/4.0*(r[0, 1] + r[1, 0])/c
                    d = 1/4.0*(r[1, 2] + r[2, 1])/c
                else:
                # If we arrive here, d2 is big enough
                    d2 = a2 - 1/2.0*(r[0, 0] + r[1, 1])
                    d = sqrt(d2)
                    a = 1/4.0*(r[1, 0] - r[0, 1])/d
                    b = 1/4.0*(r[0, 2] + r[2, 0])/d
                    c = 1/4.0*(r[1, 2] + r[2, 1])/d
        return quaternion(a, b, c, d)

def transformationToDQ(RotTrans):
    """RotTrans = [qr,qt] is a tuple of quaternions that describe a
    transformation in 6D; the first quaternion describes the rotation,
    the second one describes the translation recall that the first entry
    of the translation quaternion is zero: qt = [0,t1,t2,t3]"""
    if isinstance(RotTrans[0],quaternion) == True:
        if isinstance(RotTrans[1],quaternion) == True:
            quat1 = RotTrans[0] #rotationsteil q_r des dualen Quaternions
            quat2 = RotTrans[1] #q_t
            quatresdu = quat2.scalar(0.5)
            quatresdu = quatresdu.times(quat1)
            value = dualQuaternion(quat1,quatresdu)
            #q = q_rot+0.5*e*q_trans*q_rot
            #print value
            return value
        else:
            print 'translation of the dual quaternion is no quaternion'
    else:
        print 'rotation of the dual quaternion is no quaternion'


def DQToTransformation(dualQuat):
    if isinstance(dualQuat,dualQuaternion):
        re = dualQuat.Real
        du = dualQuat.Dual
        dqlist = dualQuat.toList()
        rot = quaternion(dqlist[0][0],dqlist[0][1],dqlist[0][2],
                         dqlist[0][3])
        dubbledu = du.scalar(2)
        trans = dubbledu.times(re.conj())
        #warum conj und nicht inv? - weil einheitsquaternion!
        #print [rot,trans]
        return [rot,trans]
    else:
        print 'input is not a dual Quaternion'


def transformPose(pose, transformation):
    """input pose and transformation have to be two dual quaternions"""
    if isinstance(pose,dualQuaternion) == True:
        if isinstance(transformation,dualQuaternion) == True:
            pose_new = transformation.times(pose)
            #print pose_new
            return pose_new
        else:
            print 'second input is not a dual quaternion'
    else:
        print 'first input is not a dual quaternion'
```

The class defining the tangent space contains the following methods.

```
class tangentSpace:
    """tangentSpace is of the from [point, basis]. point is a vector and
    has the dimension d and basis is a d x (d-1)-matrix. Basis consists
    of column vectors but as the input in Python in line wise one has to
    make a vector in the common sense out of the input list."""
    def __init__(self, tangentPoint):
        assert isinstance(tangentPoint, matrix)
        pl = tangentPoint
        one = sqrt(pl[0]**2 + pl[1]**2 + pl[2]**2 + pl[3]**2)
        assert abs(1-one) < 10**(-4)
        self.p = pl
        basis = makeBase(tangentPoint)
        self.b = basis

    def equal(self, newTS):
        sp = self.p
        sb = self.b
        n = len(sp)
        k = 0
        for i in range(n):
            if sp[i][0] == (newTS.p)[i][0]:
                for j in range(n-1):
                    sl = sb.tolist()
                    nb = newTS.b
                    nl = nb.tolist()
                    if sl[i][j] == nl[i][j]:
                        k = k+1
        if k == n*(n-1):
            print True
            return True
        else:
            print False
            return False

    def display(self):
        """Attention: THE OUTPUT HERE IS LINE WISE!
        Thus the first row of the matrix is the first column vector
        of the basis."""
        sp = self.p
        listp = sp.tolist()
        sb = self.b
        listb = sb.tolist()
        #print [list(flatten(listp)),listb]
        return [list(flatten(listp)), listb]

    #Achtung: Es wird eine ORTHONORMALBASIS benoetigt!
    def tangentSpaceToSphereCentralProjection(self, tangentSpacePoint):
        """tangentSpacePoint = [x1,x2,...,x(n-1)] is the point in the
        tangential (hyper-) plane whose value shall be projected on the
        (p-1)sphere. The first r entries of tangentSpacePoint are the
        ones to describe the rotation, the last ones t = n-1-r describe
        the translation and stay unchanged. tangentSpace = [p,B] is the
        tangential (hyper-) plane with p the tangent point and B the
        basis of the tangential (hyper-) plane. p has n dimensions.
        The input of tangentSpacePoint has to be a list and tangentSpace
        really has to be a tangentSpace!
        The output is the wanted vector."""
        p = self.p
        B = self.b
        k = len(p)-1
        rvec = tangentSpacePoint[:k]
        rtvec = matrix(rvec).T
        rtWorldCoord = B*rtvec + p
        normalizationFactor = linalg.norm(rtWorldCoord)
        svec = rtWorldCoord/normalizationFactor
        lsvec = svec.tolist()
        tvec = tangentSpacePoint[k:]
        value = [lsvec, tvec]
        value = list(flatten(value))
        #print value
        value = makeVector(value)
        return value
```

```
def sphereToTangentSpaceCentralProjection(self, pointOnSphere):
    """pointOnSphere = [q1,q2,...,qn] is the point to be projected
    consisting of rotation and translation part.
    tangentSpace = [p,B] is the tangential (hyper-) plane with p the
    tangent point and B the basis of the tangential (hyper-) plane.
    The input of pointOnSphere has to be a list and tangentSpace
    really has to be a tangentSpace! The output is a vector in the
    tangent space (including translation)."""
    p = self.p
    B = self.b
    Bt = B.T
    n = len(p)
    qr = pointOnSphere[:n]
    qrT = matrix([qr]).T
    scalar1 = qr*p
    scalar2 = scalar1.tolist()
    scalar3 = scalar2[0][0]
    #um aus der 1x1 matrix wirklich ein skalar zu machen....
    qrInTangentSpace = Bt*(1.0/(scalar3)*qrT-p)
    qt = pointOnSphere[n:]
    lqr = qrInTangentSpace.tolist()
    value = [lqr, qt]
    value = list(flatten(value))
    #print value
    value = makeVector(value)
    return value


def transformFromSelfToTS(self, point, tangentSpace_new):
    """point = [x1,x2,...,xn] is the point in the old tangentSpace1
    that shall be projected to a new tangentSpace.
    self = [p1,B1] is the old tangent space.
    tangentSpace_new = [p2,B2] is the new tangent space.
    In the input 'point' has to be of the type list but the output
```

```
        it is a vector."""
        assert isinstance(tangentSpace_new, tangentSpace)
        Tn = tangentSpace_new
        pointOnSphere = self.tangentSpaceToSphereCentralProjection(point)
        listpoint = pointOnSphere.tolist()
        lp = list(flatten(listpoint))
        value =tangentSpace_new.sphereToTangentSpaceCentralProjection(lp)
        #print value
        return value


    def poseTransformationTS(self, pointold, transpoint,
                             tangentSpace_transform, tangentSpace_new):
        """Input shall be vectors and tangent spaces.
        I want the output to consist of vectors."""
        assert isinstance(tangentSpace_transform, tangentSpace)
        assert isinstance(tangentSpace_new, tangentSpace)
        assert isinstance(pointold, matrix)
        assert isinstance(transpoint, matrix)
        T_trans = tangentSpace_transform
        T_new = tangentSpace_new
        pt_list = list(flatten(pointold.tolist()))
        tp_list = list(flatten(transpoint.tolist()))
        pointsphere = self.tangentSpaceToSphereCentralProjection(pt_list)
        tpsphere = T_trans.tangentSpaceToSphereCentralProjection(tp_list)
        l = list(flatten(pointsphere.tolist()))
        rot = quaternion(l[0], l[1], l[2], l[3])
        trans = quaternion(0, l[4], l[5], l[6])
        rotTrans = [rot, trans] #entsteht aus pointold
        pointquat = transformationToDQ(rotTrans)
        li = list(flatten(tpsphere.tolist()))
        trot = quaternion(li[0], li[1], li[2], li[3])
        ttrans = quaternion(0, li[4], li[5], li[6])
        trotTrans = [trot, ttrans] #entsteht aus transpoint
        transpointquat = transformationToDQ(trotTrans)
        pointquat_new = transformPose(pointquat, transpointquat)
        ps_new = DQToTransformation(pointquat_new)
        rot = pointsphere_new[0].toList()
        trans = pointsphere_new[1].toList()
        pointsphere_new = [rot[0],rot[1],rot[2],rot[3],trans[1],trans[2],
                           trans[3]]
        point_new = T_new.sphereToTangentSpaceCentralProjection(ps_new)
        #print point_new
        return point_new
```

The next section is the code describing the base element.

```
class baseElement:
    """baseElement consists of a tangent space, the mean value and the
    covariance matrix of a Gaussian. The tangent space has to be of the
    class tangentSpace, the mean value is a vector and the covariance
    matrix is of the type matrix."""
    def __init__(self, tanSpace, mean, covMatr, n = 2000):
        #n is the rate of exactness for the calculation of the mass
        #by default n = 2000 samples
        assert isinstance(mean,matrix)
        assert isinstance(covMatr,matrix)
        assert all(linalg.eigvals(covMatr)>0),"not all eigenvalues positive"
        assert isinstance(tanSpace,tangentSpace)
        dm = len(mean)
        "usually the mean is 6 dimensional"
        listcovMat = covMatr.tolist()
        dcMr = len(listcovMat)
        dcMc = len(listcovMat[0])
        assert dcMr == dcMc
        assert dm == dcMr
        self.tanSp = tanSpace
        self.mean = mean
        self.covMat = covMatr
        #Calculation of the mass of the base element:
        sigma = 0
        X = zeros((n,6))
        def f(u,v,w,x,y,z):
            den = (1 + u*u + v*v + w*w) #denominator
            psi = 1.0 / (den*den)
            return psi
        mu = self.mean
        cov = self.covMat
        mulist = list(flatten(mu.tolist()))
        for j in range(n):
            X[j] = multivariate_normal(mulist,cov)
            sigma = sigma + f(X[j][0],X[j][1],X[j][2],X[j][3],X[j][4],
                              X[j][5])
        self.mass = sigma/n
        self.maDist = 1

    def computeMassMonteCarlo6D(self,n = 2000):
        #n = 1000 streuung ab 3ter stelle hinterm komma
        #n = 5000 streuung ab 4ter stelle
        Sum = 0
        sigma = 0
        X = zeros((n,6))
        def f(u,v,w,x,y,z):
            den = (1 + u*u + v*v + w*w) #denominator
            psi = 1.0 / (den*den)
            return psi
        mu = self.mean
        cov = self.covMat
        mulist = list(flatten(mu.tolist()))
        weight = 1
        for j in range(n):
            X[j] = multivariate_normal(mulist,cov)
            sigma = sigma + f(X[j][0],X[j][1],X[j][2],X[j][3],X[j][4],
                              X[j][5])
        Sum += weight * sigma
        #print Sum/n
        return Sum/n

    def equal(self,newBE):
        st = self.tanSp
        sm = self.mean
        sc = self.covMat
        nt = newBE.tanSp
        nm = newBE.mean
        nc = newBE.covMat
        k = 0
        if st.equal(nt):
            k = k+1
        else:
            print 'TangentSpace not equal.'
        km = 0
        if len(sm) == len(nm):
            sml = sm.tolist()
            nml = nm.tolist()
            for i in range(len(sml)):
                if sml[i] == nml[i]:
                    km = km+1
        if km == len(sm):
            k = k+1
        else:
            print 'Mean not equal.'
        kc = 0
        scl = sc.tolist()
        ncl = nc.tolist()
        if (len(scl) == len(ncl)) & (len(scl[0]) == len(ncl[0])):
            for i in range(len(scl)):
                for j in range(len(scl[0])):
                    if scl[i][j] == ncl[i][j]:
                        kc = kc+1
        if kc == (len(scl)*len(scl[0])):
            k = k+1
        else:
            print 'CovMat not equal.'
        if k == 3:
            return True
        else:
            return False

    def display(self):
        #st = self.tanSp
        stl = self.tanSp.display()
        sm = self.mean
        sml = sm.tolist()
        sml = list(flatten(sml))
        sc = self.covMat
        scl = sc.tolist()
        #print [stl,sml,scl]
        return [stl,sml,scl]

    def dimensions(self):
        sm = self.mean
        st = self.tanSp
        point = st.p
        dimRot = len(point)-1
        dimTrans = len(sm)-dimRot
        print 'The dimension of the rotation is', dimRot,
        print ', and of the translation', dimTrans, '.'
        return dimRot and dimTrans

    def extractDualQuaternion(self):
        mean = self.mean
        meanlist = list(flatten(mean.tolist()))
        TS = self.tanSp
        pointonsphere = TS.tangentSpaceToSphereCentralProjection(meanlist)
        l = list(flatten(pointonsphere.tolist()))
        rot = quaternion(l[0],l[1],l[2],l[3])
        trans = quaternion(0,l[4],l[5],l[6])
        rotTrans = [rot,trans]
        dualQuat = transformationToDQ(rotTrans)
        return dualQuat
```

```
def changeTS(self,newTanSpace):
    """baseElement consists of a tangent space (=[[point],[matrix]]),
    the mean value of the Gaissian kernel (=[point]) and the
    covariance matrix (=[matrix]).
    The output is a base element."""
    assert isinstance(newTanSpace, tangentSpace)
    TS_old = self.tanSp #Tangentenraum
    Mean_old = self.mean
    Mean_oldlist = Mean_old.tolist()
    Mean_oldlist = list(flatten(Mean_oldlist))
    CovMat_old = self.covMat
    TS_new = newTanSpace #in der Form: [[point],[matrix]]
    Mean_new = TS_old.transformFromSelfToTS(Mean_oldlist,TS_new)
    Jac = computeJacobian(Mean_oldlist,TS_old,TS_new)
    JacT = Jac.T
    CovMat_new = Jac*CovMat_old*JacT
    baseElem = baseElement(TS_new,Mean_new,CovMat_new)
    return baseElem

def mahalanobisDistance(self,bE):
    mu1 = self.mean
    mu2 = bE.mean
    cm1 = self.covMat
    cm2 = bE.covMat
    value = math.exp((-0.5)*(mu1-mu2).T*inv(cm1+cm2)*(mu1-mu2))
    return value

def fuse(self,baseElement_new,modeFlag):
    """modeFlag == 0 or modeFlag ==1.
    In case 0 no adjustment of the tangent space necessary.
    In case 1 the tangent space is transformed to another tangent
    space that is centered in 'mean_new'."""
    assert isinstance(baseElement_new,baseElement)
    sp = self.tanSp.p
    np = baseElement_new.tanSp.p
    angle1 = math.acos(sp.T*np / (sqrt(sp.T*sp) * sqrt(np.T*np)))
```

```
angle2 = math.acos(sp.T*(-np) / (sqrt(sp.T*sp) * sqrt(np.T*np)))
if angle1 <= angle2:
    np = -np
else:
    np = np
tp = (sp + np)/(linalg.norm(sp + np))
T3 = tangentSpace(tp)
baseElem1T3 = self.changeTS(T3)
baseElem2T3 = baseElement_new.changeTS(T3)
CovMat1 = baseElem1T3.covMat
CovMat2 = baseElem2T3.covMat
#less matrix inversions: CovMat3 = inv(inv(CovMat1) + inv(CovMat2))
CovMat3 = CovMat1*inv(CovMat1 + CovMat2)*CovMat2
mean1 = baseElem1T3.mean
mean2 = baseElem2T3.mean
mean3 = CovMat2*inv((CovMat1+CovMat2))*mean1
mean3 = mean3+CovMat1*inv((CovMat1+CovMat2))*mean2
baseElem3 = baseElement(T3,mean3, CovMat3)
MaDi2 = baseElem2T3.mahalanobisDistance(baseElem1T3)
baseElem3.maDist = baseElem3.maDist*MaDi2
#print baseElem3.maDist
if modeFlag == 1:
    mean3list = mean3.tolist()
    mean3list = list(flatten(mean3list))
    mean3center = T3.tangentSpaceToSphereCentralProjection(mean3list)
    l = len(sp)
    list3 = mean3center.tolist()
    list3 = list(flatten(list3))
    mean3center_rot = list3[:l]
    mean3center_rot = makeVector(mean3center_rot)
    T3center = tangentSpace(mean3center_rot)
    baseElem3 = baseElem3.changeTS(T3center)
    return baseElem3
elif modeFlag == 0:
    return baseElem3
else:
```

```
            print 'wrong modeFlag'
            return NONE

    def merge(self,baseElem,lam1,lam2,modeFlag):
        assert isinstance(baseElem,baseElement)
        sp = self.tanSp.p
        np = baseElem.tanSp.p
        if sp.T*np<0:
            np = -np
        tp = (sp + np)/(linalg.norm(sp + np))
        #print tp
        T3 = tangentSpace(tp)
        baseElem1T3 = self.changeTS(T3)
        baseElem2T3 = baseElem.changeTS(T3)
        mean1 = baseElem1T3.mean
        mean2 = baseElem2T3.mean
        l12 = lam1/(lam1+lam2)
        l21 = lam2/(lam1+lam2)
        mean3 = l12*mean1 + l21*mean2
        CovMat1 = baseElem1T3.covMat
        CovMat2 = baseElem2T3.covMat
        CovMat3 = l12*CovMat1+l21*CovMat2
        CovMat3 = CpvMat3 + l12*l21*(mean1-mean2)*(mean1-mean2).T
        baseElem3 = baseElement(T3,mean3, CovMat3)
        if modeFlag == 1:
            mean3list = mean3.tolist()
            mean3li = list(flatten(mean3list))
            mean3center = T3.tangentSpaceToSphereCentralProjection(mean3li)
            l = len(sp)
            list3 = mean3center.tolist()
            list3 = list(flatten(list3))
            mean3center_rot = list3[:l]
            mean3center_rot = makeVector(mean3center_rot)
            T3center = tangentSpace(mean3center_rot)
            baseElem3 = baseElem3.changeTS(T3center)
            return baseElem3
        elif modeFlag == 0:
            return baseElem3
        else:
            print 'wrong modeFlag'
            return NONE


    def density(self,point):
        """If the point is close to the equator the projected point
        value gets to infinity and thus has the density = 0.
        point has to be type list 7D (4 dimensions for the rotation on
        the sphere and 3 dimensions for the translation).
        The output is a scalar."""
        TS = self.tanSp
        n = len(TS.p)
        rotpoint = matrix(point[:n])
        h = 0.1e-6
        print (rotpoint)*TS.p
        if (abs(math.acos((rotpoint)*TS.p))>=math.pi*0.5-h)&
           (abs(math.acos((rotpoint)*TS.p))<=math.pi*0.5+h):
            density = 0
        else:
            mean = self.mean
            lmean = mean.tolist()
            lmean = list(flatten(lmean))
            dim = len(lmean)
            V = self.covMat
            VT = V.T
            CM = 0.5*(V+VT)
            value = TS.sphereToTangentSpaceCentralProjection(point)
            exponent = -0.5*(value-mean).T*inv(CM)*(value-mean)
            exponent = exponent.tolist()
            exponent = exponent[0][0]
            #normalize = 1/math.sqrt(det(2*math.pi*CM))
            density = math.exp(exponent)/self.mass
        #print density
        return density

    def randomPoseTransformation(self, baseElement2):
        """Composition of two base elements. If the covariance matrix is
        chosen to be zero, the pose transformation is secure, without
        random part."""
        if isinstance(baseElement2,baseElement) == True:
            smean = self.mean
            smeanlist = list(flatten(smean.tolist()))
            sT = self.tanSp
            spoint = sT.tangentSpaceToSphereCentralProjection(smeanlist)
            l = list(flatten(spoint.tolist()))
            srot = quaternion(l[0],l[1],l[2],l[3])
            strans = quaternion(0,l[4],l[5],l[6])
            rotTrans = [srot,strans]
            spoint = transformationToDQ(rotTrans)
            #PROBLEM:
            #for the transformation matrix q_t is needed not 1/2 q_t*q_r!
            #assert isinstance(spoint,dualQuaternion)
            nmean = baseElement2.mean
            nmeanlist = list(flatten(nmean.tolist()))
            nT = baseElement2.tanSp
            npoint = nT.tangentSpaceToSphereCentralProjection(nmeanlist)
            li = list(flatten(npoint.tolist()))
            nrot = quaternion(li[0],li[1],li[2],li[3])
            ntrans = quaternion(0,li[4],li[5],li[6])
            nrotTrans = [nrot,ntrans]
            npoint = transformationToDQ(nrotTrans)
            new_point = transformPose(spoint, npoint)
            new_point = DQToTransformation(new_point)#tupel of quaternions
            new_point_rot = new_point[0] #new_point_rot is quaternion
            point3list = new_point_rot.toList()
            point3trans = new_point[1].toList()
            point3 = makeVector(point3list)
            T3 = tangentSpace(point3)
            #print T3.display()
```

```
input = [point3list[0],point3list[1],point3list[2],
         point3list[3],point3trans[1],point3trans[2],
         point3trans[3]]
mean3 = T3.sphereToTangentSpaceCentralProjection(input)
J = zeros((6,12),float)
#see 3.5.2. in wends paper:
#\Sigma_c is a 12x12 matrix and thus J is a 6x12 matrix to
#receive a 6x6 matrix for \Sigma_3
h = 0.1e-3
I = eye(6)
# i Zeilen und j Spalten der Matrix
for j in range(6):
    z = I[j].T
    z = z*h
    pt = smean
    tp = nmean
    pointplus = pt+z #vector
    pointminus = pt-z #vector
    transformpointplus = sT.poseTransformationTS(pointplus,
                                                tp,nT,T3)
    transformpointminus = sT.poseTransformationTS(pointminus,
                                                 tp,nT,T3)
    difference = transformpointplus - transformpointminus
    qj = difference/(2*h)
    qjlist = list(flatten(qj.tolist()))
    for i in range(6):
        qij = qjlist[i]
        J[i][j] = qij
for j in range(6):
    z = I[j].T
    z = z*h
    pt = smean
    tp = nmean
    transplus = tp+z #vector
    transminus = tp-z #vector
    transpluspoint = sT.poseTransformationTS(pt,transplus,
                                                    nT,T3)
                transminuspoint = sT.poseTransformationTS(pt,transminus,
                                                     nT,T3)
                difference = transpluspoint - transminuspoint
                qj = difference/(2*h)
                qjlist = list(flatten(qj.tolist()))
                for i in range(6):
                    qij = qjlist[i]
                    J[i][j+6] = qij
            J = matrix(J)
            #print J.round()
            VT = zeros((12,12),float)
            for i in range(6):
                for j in range(6):
                    cMat1 = self.covMat
                    cMat2 = baseElement2.covMat
                    VT[i][j] = cMat1[i,j]
                    VT[i+6][j+6] = cMat2[i,j]
            VT = matrix(VT)
            CV3 = J*VT*J.T
            baseElement3 = baseElement(T3,mean3,CV3)
            return baseElement3

    def sKLDivBound(self, otherBE, slam, olam):
        smu = self.mean
        lensmu = len(smu)
        omu = otherBE.mean
        lenomu = len(omu)
        assert(lensmu == lenomu)
        scm = self.covMat
        ocm = otherBE.covMat
        G = (smu-omu)*(smu-omu).T
        solam = slam+olam
        somu = slam/solam*smu + olam/solam*omu
        socm = 1/solam*(slam*scm + olam*ocm + slam*olam*G)
        ssocm = inv(scm)*socm
        ssocminv = inv(ssocm)
        osocm = inv(ocm)*socm
        osocminv = inv(osocm)
        value = 0.5*slam*trace(ssocm+ssocminv+(slam/solam)**2
                              *(inv(scm)+inv(socm))*G)
                +0.5*olam*trace(osocm+osocminv+(olam/solam)**2
                               *(inv(ocm)+inv(socm))*G)-lensmu*solam
        return value

    def draw1Sample(self):
        """the output of this function isn't the image, but the matrix,
        needed for drawing it in pivy."""
        mu = self.mean
        mulist = list(flatten(mu.tolist()))
        sigma = self.covMat
        sample = multivariate_normal(mulist,sigma)
        samplelist = sample.tolist()
        TS = self.tanSp
        value = TS.tangentSpaceToSphereCentralProjection(samplelist)
        value = list(flatten(value.tolist()))
        rot = quaternion(value[0],value[1],value[2],value[3])
        trans = quaternion(0,value[4],value[5],value[6])
        rotTrans = [rot,trans]
        dualQuat = transformationToDQ(rotTrans)
        transform = dualQuaternion.transformationMatrix(dualQuat)
        matrix = makeMatTransInput(transform[0],transform[1])
        matrix = matrix.T
        limat = matrix.tolist()
        limat = list(flatten(limat))
        return limat

    def draw1SampleMat(self):
        """the output of this function isn't the image, but the matrix,
        needed for drawing it in pivy."""
        mu = self.mean
        mulist = list(flatten(mu.tolist()))
        sigma = self.covMat
        sample = multivariate_normal(mulist,sigma)
        samplelist = sample.tolist()
        TS = self.tanSp
        value = TS.tangentSpaceToSphereCentralProjection(samplelist)
        value = list(flatten(value.tolist()))
        rot = quaternion(value[0],value[1],value[2],value[3])
        trans = quaternion(0,value[4],value[5],value[6])
        rotTrans = [rot,trans]
        dualQuat = transformationToDQ(rotTrans)
        transform = dualQuaternion.transformationMatrix(dualQuat)
        matrix = makeMatTransInput(transform[0],transform[1])
        return matrix

    def drawNSamples(self,N):
        """the output of this function isn't the image, but the matrix,
        needed for drawing it in pivy."""
        mu = self.mean
        mulist = list(flatten(mu.tolist()))
        sigma = self.covMat
        sample = multivariate_normal(mulist,sigma,N)
        sampleli = sample.tolist()
        L = []
        for i in range(N):
            TS = self.tanSp
            value = TS.tangentSpaceToSphereCentralProjection(sampleli[i])
            value = list(flatten(value.tolist()))
            rot = quaternion(value[0],value[1],value[2],value[3])
            trans = quaternion(0,value[4],value[5],value[6])
            rotTrans = [rot,trans]
            dualQuat = transformationToDQ(rotTrans)
            transform = dualQuaternion.transformationMatrix(dualQuat)
            matrix = makeMatTransInput(transform[0],transform[1])
            matrix = matrix.T
            limat = matrix.tolist()
            limat = list(flatten(limat))
            L.append(limat)
        return L

    def paintCS(self):
        mu = self.mean
        mulist = list(flatten(mu.tolist()))
        TS = self.tanSp
        value = TS.tangentSpaceToSphereCentralProjection(mulist)
        value = list(flatten(value.tolist()))
        rot = quaternion(value[0],value[1],value[2],value[3])
        trans = quaternion(0,value[4],value[5],value[6])
        rotTrans = [rot,trans]
        dualQuat = transformationToDQ(rotTrans)
        transform = dualQuaternion.transformationMatrix(dualQuat)
        matrix = makeMatTransInput(transform[0],transform[1])
        matrix = matrix.T
        limat = matrix.tolist()
        limat = list(flatten(limat))
        return limat
```

In the following section mixtures of projected Gaussians are introduced.

```
class mixture:
    """Mixture is a double array. Consisting of a list of baseElements
    with a weight for each one. Mixture is an array with two columns.
    One for the baseElements and one for the weights."""
    def __init__(self, liste):
        assert isinstance(liste,list)
        n = len(liste)
        k = 0
        for j in range(n):
            baseElem = liste[j][0]
            weight = liste[j][1]
            assert isinstance(baseElem,baseElement)
            k = k+weight
        assert round(k, 2) == 1.00, 'the weights do not sum to 1'
        self.l            = liste
        self.nBaseElem    = len(liste)
        self.CSum         = 1e-30
        self.CSumOld      = 1e-30
        self.CSumVect     = []
        self.logCSumVect  = []
        self.iteration    = 0

    def toList(self):
        sl = self.l
        return sl

    def equal(self,new_mixture):
        n = len(self.l)
        sl = self.l
        ml = new_mixture.l
        k = 0
        for i in range(n):
            bE = sl[i][0]
            nbE = ml[i][0]
            sweight = sl[i][1]
            mweight = ml[i][1]
            bE.equal(nbE)
            if (sweight == mweight) & (bE.equal(nbE) == True):
                k = k+1
        if (k-n) == 0:
            print 'mixtures are equal'
            return True
        else:
            print 'mixtures are NOT equal'
            return False

#    def display(self):
#        l = self.l
#        n = len(l)
#        z = zeros((n,1))
#        lis = [l[i][0].display() for i in range(len(self.l))]
#         for i in range(len(self.l)):
#             weight = l[i][1]
#             z[i][0] = weight
#         print lis
#         print z
#         return z

    def mixedDensity(self,point):
        "Input is a list, output a scalar."
        sl = self.l
        n = len(sl)
        z = zeros((1,n),float)
        for i in range(n):
            bE = sl[i][0]
            density = bE.baseDensity(point)
            weight = sl[i][1]
            z[0][i] = weight*density
        s = z.sum()
        print 'the mixed density is:', s
        return s

    def computeMassMonteCarlo6D(self,n):
        Sum = 0
        sigma = 0
        liste = self.l
        d = len(self.l)
        X = zeros((n,6))
        def f(u,v,w,x,y,z):
            den = (1 + u*u + v*v + w*w) #denominator
            psi = 1.0 / (den*den)
            return psi
        for i in range(d): #fuer jedes baseElement:
            baseElem = liste[i][0]
            mu = baseElem.mean
            cov = baseElem.covMat
            mulist = list(flatten(mu.tolist()))
            weight = liste[i][1]
            for j in range(n):
                X[j] = multivariate_normal(mulist,cov)
                sigma = sigma + f(X[j][0],X[j][1],X[j][2],X[j][3],
                                  X[j][4],X[j][5])
            Sum += weight * sigma
        #print Sum/n
        return Sum/n


    def fuseMoPG(self,other_mixture):
        """Mixture consists of a list of base elements, each with its
        weight. That means:
        Mixture1=[[PG_1,lambda_1],[PG_2, lambda_2],...,[PG_n,lambda_n]]"""
        assert isinstance(other_mixture,mixture)
        sl = self.l
        ml = other_mixture.l
        #n = samples
        n1 = len(sl) #Laenge der Mixture
        n2 = len(ml)
        n3 = n1*n2
        #Dimension der zusammengefuegten Mixture aus self u. other_mixture
        "a = 5" #willkuerliche Festlegung basierend auf Erfahrungswerten
        #Eintraege fuer Mixture3 berechnen.
        #an die jeweils erste Stelle der Tupel kommt:
        fused_bE = [[sl[i][0].fuse(ml[j][0],1) for j in range(n2)]
                    for i in range(n1)]
        fbE = array(list(flatten(fused_bE)))
        z = zeros(n3)
        fm = column_stack((fbE,z))
        for i in range(n1):
            for j in range(n2):
                #Laufindex laeuft durch von 0 bis n3-1.
                #an die jeweils zweite Stelle der Tupel kommt:
                weight = sl[i][1]*ml[j][1]
                #Produkt der einzelnen Gewichte in der entsprechenden
                #Reihenfolge
                PGi = sl[i][0]
                t1 = PGi.tanSp
                q1 = t1.p
                PGj = ml[j][0]
                t2 = PGj.tanSp
                q2 = t2.p
                f = func(q1,q2)
                m = fbE[i*n2+j].mass
                madist = fbE[i*n2+j].maDist
                fm[i*n2+j][1] = 1/m*weight*f*madist -
        "Achtung: fm ist im moment ein ndarray."
        s = 0
        for k in range(n3):
            s = s + fm[k][1]
        for l in range(n3):
            fm[l][1] = fm[l][1]/s
        fused_mixture = mixture(list(fm))
#         print fm
#         mass = fused_mixture.computeMassMonteCarlo6D(n)
#         print mass
        mix = mixture(list(fm))
        #print mix.l[4][0].display()
        return mix

    def randomMoPGTransformation(self,other_mixture):
        #NICHT GETESTET
        assert isinstance(other_mixture,mixture)
        sm = self.l
        om = other_mixture.l
        n1 = len(sm)
        n2 = len(om)
        n3 = n1*n2
        composed_mixture = []
        for i in range(n1):
            PGi = sm[i][0]
            for j in range(n2):
                PGj = om[j][0]
                PG = PGi.randomPoseTransformation(PGj)
                composed_mixture.append([PG,sm[i][1]*om[j][1]])
        s = 0
        for k in range(n3):
            s = s + composed_mixture[k][1]
        for l in range(n3):
            composed_mixture[l][1] = composed_mixture[l][1]/s
        return mixture(composed_mixture)

    def sKLDivBound(self,other_mixture):
        """This function calculates the symmetrized KL divergence between
        each base element of one mixture and each base element of the
        other mixture."""
        sml = self.l
        oml = other_mixture.l
        k = 1000
        for i in range(len(sml)):
            PGi = sml[i][0]
            smlam = sml[i][1]
            for j in range(len(oml)):
                PGj = oml[j][0]
                omlam = oml[j][1]
                div = PGi.sKLDivBound(PGj,smlam,omlam)
                if div < k:
                    k = div
                    selfmix = i
                    othermix = j
        print 'k = ', k, ', i = ',selfmix,', j = ', othermix
        return [selfmix, othermix]

    def merge(self, nummixelem):
        sml = self.l
        nself = len(sml)
        num = nself-nummixelem
        if num<=0:
            print 'mixture has less elements'
            return self
        else:
            for l in range(num):
                bound = 1000
                for i in range(len(sml)):
                    PGi = sml[i][0]
                    lam1 = sml[i][1]
                    for j in range(len(sml)-(i+1)):
                        PGj = sml[j+i+1][0]
                        lam2 = sml[j+i+1][1]
                        div = PGi.sKLDivBound(PGj,lam1,lam2)
                        if div < bound:
                            bound = div
                            selfmix = i
                            othermix = j+i+1

                print 'divergence = ',bound,',i=',selfmix,',j=',othermix
                lamself = sml[selfmix][1]
                lamother = sml[othermix][1]
                baseElem = sml[selfmix][0].merge(sml[othermix][0],
                                                 lamself,lamother,0)
                sml = removeEl(sml,othermix)
                sml = removeEl(sml,selfmix)
                sml.append([baseElem,lamself+lamother])
            return mixture(sml)

    def paint(self,N):
        samples = self.drawNSamplesV(N)
#        showSamplesThread(samples)
        return samples

    def drawNSamples(self,N):
        """the output of this function isn't the image, but the matrix,
        needed for drawing it in pivy."""
        sl = self.l
        L  = []
        for n in range(N):
            lam = 0
            i   = 0
            rn  = random_sample((1,1))[0][0]
            while lam<rn:
                weight = sl[i][1]
                lam = lam + weight
                i = i+1
            #print i-1
            baseElem = sl[i-1][0]
            mu = baseElem.mean
            mulist = list(flatten(mu.tolist()))
            sigma = baseElem.covMat
            sample = multivariate_normal(mulist,sigma)
            samli = sample.tolist()
            v=baseElem.tanSp.tangentSpaceToSphereCentralProjection(samli)
            value = list(flatten(v.tolist()))
            rot = quaternion(value[0],value[1],value[2],value[3])
            trans = quaternion(0,value[4],value[5],value[6])
            rotTrans = [rot,trans]
            dualQuat = transformationToDQ(rotTrans)
            transform = dualQuaternion.transformationMatrix(dualQuat)
            matrix = makeMatTransInput(transform[0],transform[1])
            L.append(matrix)
            L.append(i-1)
        return L

    def drawNSamplesV(self,N):
        """the output of this function isn't the image, but the matrix,
        needed for drawing it in pivy."""
        sl = self.l
        L = []
        for n in range(N):
            lam = 0
            i = 0
            rn = random_sample((1,1))[0][0]
            while lam<rn:
                weight = sl[i][1]
                lam = lam + weight
                i = i+1
            #print i-1
            baseElem = sl[i-1][0]
            mu = baseElem.mean
            mulist = list(flatten(mu.tolist()))
            sigma = baseElem.covMat
            sample = multivariate_normal(mulist,sigma)
            samli = sample.tolist()
            v = baseElem.tanSp.tangentSpaceToSphereCentralProjection(samli)
            value = list(flatten(v.tolist()))
            L.append([value, i-1])
        return L


    def fitMixture(self, sampleSet, maxIterations, minIncrement, eps):
        # This implementation follows the one in Mathematica
        """ @1: the table of samples (size of table: 7xN)
            @2: number of MAG kernels
            @3: translation range
            @4: the maximim number of iterations the algorithm is allows
                to make the relative improvement of total likelihood that
                is sufficient to stop the algoithm
            @6: the number that will be added to all eigenvalues of all
                the estimated covariance matrices (prevents them from
                becoming singular)
            @r: resulting Mag"""
        numberOfSamples = len(sampleSet)
        increment       = float('infinity')
        for idx in [1]:
            # E Step
            # unnormalizedC collects for each sample the densities of
```

```
# the base elements multiplied by weight
# numberOfSamples x nBaseElements
# the rows contain the base elements
# the columns contain the samples
unnormalizedC = [[self.l[elementIdx][1]
                  *(self.l[elementIdx][0]).
                  density(sampleSet[sampleIdx][:7])
                  for elementIdx in range(self.nBaseElem)]
                      for sampleIdx in range(numberOfSamples)]
# get the sums of the rows
CWeights       = [sum(row) for row in unnormalizedC]
# these sums of the row are actually the likelyhoods of a
#sample given the mixture
self.CSum      = sum(CWeights)
logCWeights    = [log(weight) for weight in CWeights]
logCSum        = sum(logCWeights)
# up to the log this is the log likelihood from page 439,
#(9.28), in Bishop
self.CSumVect.append(self.CSum)
self.logCSumVect.append(logCSum)
increment      = abs(self.CSum/self.CSumOld - 1.0)
self.CSumOld   = self.CSum
# when CSum becomes stationary, assume the algorithm has
# converged
# normalizedC collects the 'responsibilities' gamma(znk)
normalizedC    = [[unnormalizedC[sampleIdx][elementIdx] /
                  CWeights[sampleIdx]
                  for elementIdx in range(self.nBaseElem)]
                      for sampleIdx in range(numberOfSamples)]
# M step
# now we work on the columns of normalizedC, so we use a
#transposed version
nCmat          = matrix(normalizedC)
nCmatT         = nCmat.T
normalizedCT   = nCmatT.tolist()
nCTSampleSum   = [sum(row) for row in normalizedCT]
```

```
# the Nk from the book
# re-project each sample to each tangent space
reproSamples=[[self.l[elementIdx][0].tanSp.
               sphereToTangentSpaceCentralProjection
               (sampleSet[sampleIdx])
               for elementIdx in range(self.nBaseElem)]
                   for sampleIdx in range(numberOfSamples)]
weightedRS=[[normalizedC[sampleIdx][elementIdx]
             *reproSamples[sampleIdx][elementIdx]
             for elementIdx in range(self.nBaseElem)]
                 for sampleIdx in range(numberOfSamples)]
# the mean is the normalized sum over the weighted
# reprojected samples, one for each base element
meanValues=[matrix([[0],[0],[0],[0],[0],[0]])
            for elementIdx in range(self.nBaseElem)]
for sampleIdx in range(numberOfSamples):
    meanValues=[meanValues[elementIdx]
                +weightedRS[sampleIdx][elementIdx]
                for elementIdx in range(self.nBaseElem)]
meanValues=[1/nCTSampleSum[elementIdx]*meanValues[elementIdx]
            for elementIdx in range(self.nBaseElem)]
# the covariance is estimated as the weighted sample covariance
covMatrices   = [matrix(eye(6)) for elementIdx in
                 range(self.nBaseElem)]
for elementIdx in range(self.nBaseElem):
    for sampleIdx in range(numberOfSamples):
        covMatrices[elementIdx] =
        (normalizedC[sampleIdx][elementIdx]*
        (reproSamples[sampleIdx][elementIdx]
         -meanValues[elementIdx])*
        (reproSamples[sampleIdx][elementIdx]
         -meanValues[elementIdx]).T)
covMatrices=[1/nCTSampleSum[elementIdx]
             *covMatrices[elementIdx]
             for elementIdx in range(self.nBaseElem)]
newWeights=[nCTSampleSum[elementIdx]
```

```
                        /numberOfSamples for elementIdx in
                        range(self.nBaseElem)]
            # write the results back to the mixture
            for elementIdx in range(self.nBaseElem):
                self.l[elementIdx][1] = newWeights[elementIdx]
                self.l[elementIdx][0].tanSp.mean
                =meanValues[elementIdx]
                self.l[elementIdx][0].tanSp.covMat
                =covMatrices[elementIdx]
            # and finally, shift the tangent spaces towards PG0,
            # i.e. zero mean
            for elementIdx in range(self.nBaseElem):
                tanSpPt  = meanValues[elementIdx].T.tolist()[0]
                newTanPt = self.l[elementIdx][0].tanSp.
                tangentSpaceToSphereCentralProjection(tanSpPt)[:4]
                newTanSp = tangentSpace(newTanPt)
                self.l[elementIdx][0].changeTS(newTanSp)
            print self.iteration
            self.iteration     = self.iteration + 1

    def resetFitting(self):
        self.CSumOld      = 1e-30
        self.CSumVect     = []
        self.logCSumVect  = []
        self.iteration    = 0
```

From here on the quaternions and their methods are defined.

```
class quaternion:
    """ definition of the quaternion and its properties
    """
    def __init__(self, a, b, c, d):
        self.R = float(a)  # real part
        self.I = float(b)  # first imaginary part
        self.J = float(c)  # second imaginary part
        self.K = float(d)  # third imaginary part
```

```
def norm(self):
    return sqrt(self.R*self.R + self.I*self.I + self.J*self.J +
                self.K*self.K)

def norm2(self):
    return self.R*self.R+self.I*self.I+self.J*self.J+self.K*self.K

def normalizeD(self): # destructively normalizes the quaternion
    n       = self.norm()
    if n == 0:
        print 'cant normalize, is 0'
    else:
        self.R = self.R/n  # real part
        self.I = self.I/n  # first imaginary part
        self.J = self.J/n  # second imaginary part
        self.K = self.K/n  # third imaginary part

def normalize(self):
    # non-destructively returns the normalized quaternion
    n       = self.norm()
    qret    = quaternion(self.R, self.I, self.J, self.K)
    if n == 0:
        print 'cant normalize, is 0'
    else:
        qret.R = self.R/n  # real part
        qret.I = self.I/n  # first imaginary part
        qret.J = self.J/n  # second imaginary part
        qret.K = self.K/n  # third imaginary part
    return qret

def conjD(self):
    # destructively turns the quaternion into its conjugate
    self.R =   self.R  # real part
    self.I = - self.I  # first imaginary part
    self.J = - self.J  # second imaginary part
    self.K = - self.K  # third imaginary part
```

```
        return self

    def conj(self): # non-destructively returns the conjugate quaternion
        qret   = quaternion(self.R, self.I, self.J, self.K)
        qret.R =   self.R  # real part
        qret.I = - self.I  # first imaginary part
        qret.J = - self.J  # second imaginary part
        qret.K = - self.K  # third imaginary part
        return qret

    def times(self,factor): # multiplication with another quaternion
        qret   = quaternion(0, 0, 0, 0)
        qret.R = self.R*factor.R - self.I*factor.I - self.J*factor.J
                 - self.K*factor.K
        qret.I = self.I*factor.R + self.R*factor.I - self.K*factor.J
                 + self.J*factor.K
        qret.J = self.J*factor.R + self.K*factor.I + self.R*factor.J
                 - self.I*factor.K
        qret.K = self.K*factor.R - self.J*factor.I + self.I*factor.J
                 + self.R*factor.K  # third imaginary part
        return qret

    def toList(self):
        return([self.R, self.I, self.J, self.K])

    def invD(self):
        n2 = self.norm2()
        if n2 == 0:
            print 'cant invert, is 0'
        else:
            self.R =   self.R/n2  # real part
            self.I = - self.I/n2  # first imaginary part
            self.J = - self.J/n2  # second imaginary part
            self.K = - self.K/n2  # third imaginary part

    def inv(self):
        qret    = quaternion(0,0,0,0)
        n2      = self.norm2()
        if n2 == 0:
            print 'cant invert, is 0'
        else:
            qret.R =   self.R/n2  # real part
            qret.I = - self.I/n2  # first imaginary part
            qret.J = - self.J/n2  # second imaginary part
            qret.K = - self.K/n2  # third imaginary part
            #print qret
            return qret

    def equal(self,same): #tests whether is equal to another quaternion
        if (self.R == same.R) & (self.I == same.I) & (self.J == same.J)
           & (self.K == same.K):
            return True
        else:
            return False

    def copy(self):
        qret    = quaternion(0,0,0,0)
        qret.R =    self.R  # real part
        qret.I =    self.I  # first imaginary part
        qret.J =    self.J  # second imaginary part
        qret.K =    self.K  # third imaginary part
        return qret

    def plus(self,otherq):
        qret = quaternion(0,0,0,0)
        qret.R = self.R + otherq.R
        qret.I = self.I + otherq.I
        qret.J = self.J + otherq.J
        qret.K = self.K + otherq.K
        return qret

    def scalar(self,num):
        qret = quaternion(0,0,0,0)
        qret.R = num*self.R
        qret.I = num*self.I
        qret.J = num*self.J
        qret.K = num*self.K
        return qret

    def toMatrix(self):
        a = self.R
        b = self.I
        c = self.J
        d = self.K
        mat=matrix([[1 - 2*c*c - 2*d*d, 2*b*c - 2*a*d, 2*(a*c + b*d)], \
                    [2*(b*c + a*d), 1 - 2*b*b - 2*d*d, -2*a*b + 2*c*d],\
                    [-2*a*c + 2*b*d, 2*(a*b + c*d), 1 - 2*b*b - 2*c*c]])
        return mat

    def randomUnit(self):
        """this method returns a random unit quaternion from an equal
        distribution on S3"""
        qret = quaternion(1,1,1,1)
        while qret.norm() > 1.0:
            qret.R = random.uniform(-1.0, 1.0)
            qret.I = random.uniform(-1.0, 1.0)
            qret.J = random.uniform(-1.0, 1.0)
            qret.K = random.uniform(-1.0, 1.0)
        qret = qret.normalize()
        return qret

    def randomImaginary(self,min = [-5,-5,-5],max = [5,5,5]):
        """this method returns a random imaginary quaternion from an
        equal distribution on S3"""
        qret = quaternion(0,0,0,0)
        qret.I = random.uniform(min[0],max[0])
        qret.J = random.uniform(min[1],max[1])
        qret.K = random.uniform(min[2],max[2])
        return qret

    def radAxis(self, rad, axis):
        """
        this method returns a unit quaternion corresponding to a
        rotation by rad around axis
        """
        # axis need not be normalized, we'll take care of that
        qret = quaternion(1,0,0,0)
        a1   = float(axis[0])
        a2   = float(axis[1])
        a3   = float(axis[2])
        axisnorm = sqrt(a1*a1 + a2*a2 + a3*a3)
        if axisnorm > 0:
            qret.R = cos(rad/2.)
            sr2    = sin(rad/2.)/axisnorm
            qret.I = sr2*a1
            qret.J = sr2*a2
            qret.K = sr2*a3
        return qret

    def degreeAxis(self, deg, axis):
        """this method returns a unit quaternion corresponding to a
        rotation by deg around axis"""
        # axis need not be normalized, we'll take care of that
        rad = deg * pi/180.
        qret = self.radAxis(rad, axis)
        return qret

    def display(self):
        print [self.R, self.I, self.J, self.K]
```

The final section is concerned with dual quaternions.

```
class dualQuaternion:
    """Quaternions have 4 entries. Quat = [q1,q2,q3,q4] the first one is
    real, the others are imaginary."""
    def __init__(self,Quat1,Quat2):
        assert isinstance(Quat1,quaternion), '1st argument no quaterion'
        assert isinstance(Quat2,quaternion), '2nd argument no quaternion'
        self.Real = Quat1.copy() #Realteil des dualen Quaternions
        self.Dual = Quat2.copy() #Dualteil des dualen Quaternions
        #ein duales Quaternion hat dann die Form: Quat1 + E*Quat2

    def plus(self,otherDuQu):
        dqret = dualQuaternion(quaternion(0,0,0,0),quaternion(0,0,0,0))
        sR = self.Real
        oR = otherDuQu.Real
        sD = self.Dual
        oD = otherDuQu.Dual
        dqret.Real = sR.plus(oR)
        dqret.Dual = sD.plus(oD)
        return dqret

    def equal(self,otherDuQu):
        #tests whether is equal to another quaternion
        sR = self.Real
        oR = otherDuQu.Real
        sD = self.Dual
        oD = otherDuQu.Dual
        if (sR.equal(oR)) & (sD.equal(oD)):
            return True
        else:
            return False

    def copy(self):
        dqret = dualQuaternion(quaternion(0,0,0,0),quaternion(0,0,0,0))
        dqret.Real = self.Real  # real part
        dqret.Dual = self.Dual  # dual part
        return dqret

    def toList(self):
        Q1 = self.Real
        Q2 = self.Dual
        return [[Q1.R, Q1.I, Q1.J, Q1.K],[Q2.R, Q2.I, Q2.J, Q2.K]]

    def times(self,otherDuQu): # multiplication with another quaternion
        dqret = dualQuaternion(quaternion(0,0,0,0),quaternion(0,0,0,0))
        qR1 = self.Real
        qR2 = otherDuQu.Real
        qD1 = self.Dual
        qD2 = otherDuQu.Dual
        qu1 = qR1.times(qD2)
        qu2 = qD1.times(qR2)
        dqret.Real = qR1.times(qR2)
        dqret.Dual = qu1.plus(qu2)
        return dqret

    def conjTotal(self):
        dqret = dualQuaternion(quaternion(0,0,0,0),quaternion(0,0,0,0))
        sR = self.Real
        sD = self.Dual
        dqret.Real = sR.conj()
        cD = sD.conj()
        dqret.Dual = cD.scalar(-1)
        return dqret
    def conjQuat(self):
        dqret = dualQuaternion(quaternion(0,0,0,0),quaternion(0,0,0,0))
        sR = self.Real
        sD = self.Dual
        dqret.Real = sR.conj()
        dqret.Dual = sD.conj()
        return dqret
    def conjDual(self):
        dqret = dualQuaternion(quaternion(0,0,0,0),quaternion(0,0,0,0))
        sR = self.Real
        du = self.Dual
        dqret.Real = sR
        dqret.Dual = du.scalar(-1)
        return dqret

    def transformationMatrix(self):
        R = self.Real
        D = self.Dual
        rotmat = R.toMatrix() #matrix
        DD = D.scalar(2)
        DD = DD.times(R.conj())
        DDlist = DD.toList()
        transvec = DDlist[1:]
        transvec = makeVector(transvec)
        #print rotmat
        #print transvec
        return [rotmat,transvec]

    def inv(self):
        ret     = dualQuaternion(quaternion(0,0,0,0),quaternion(0,0,0,0))
        real    = self.Real
        realc   = real.conj()
        ret.Real= realc
        dual    = self.Dual
        retd    = dual.times(realc)
        retd    = realc.times(retd)
        retd    = retd.scalar(-1.0)
        ret.Dual= retd
        return ret

    def display(self):
        print self.Real.toList() + self.Dual.toList()
```

A.2.

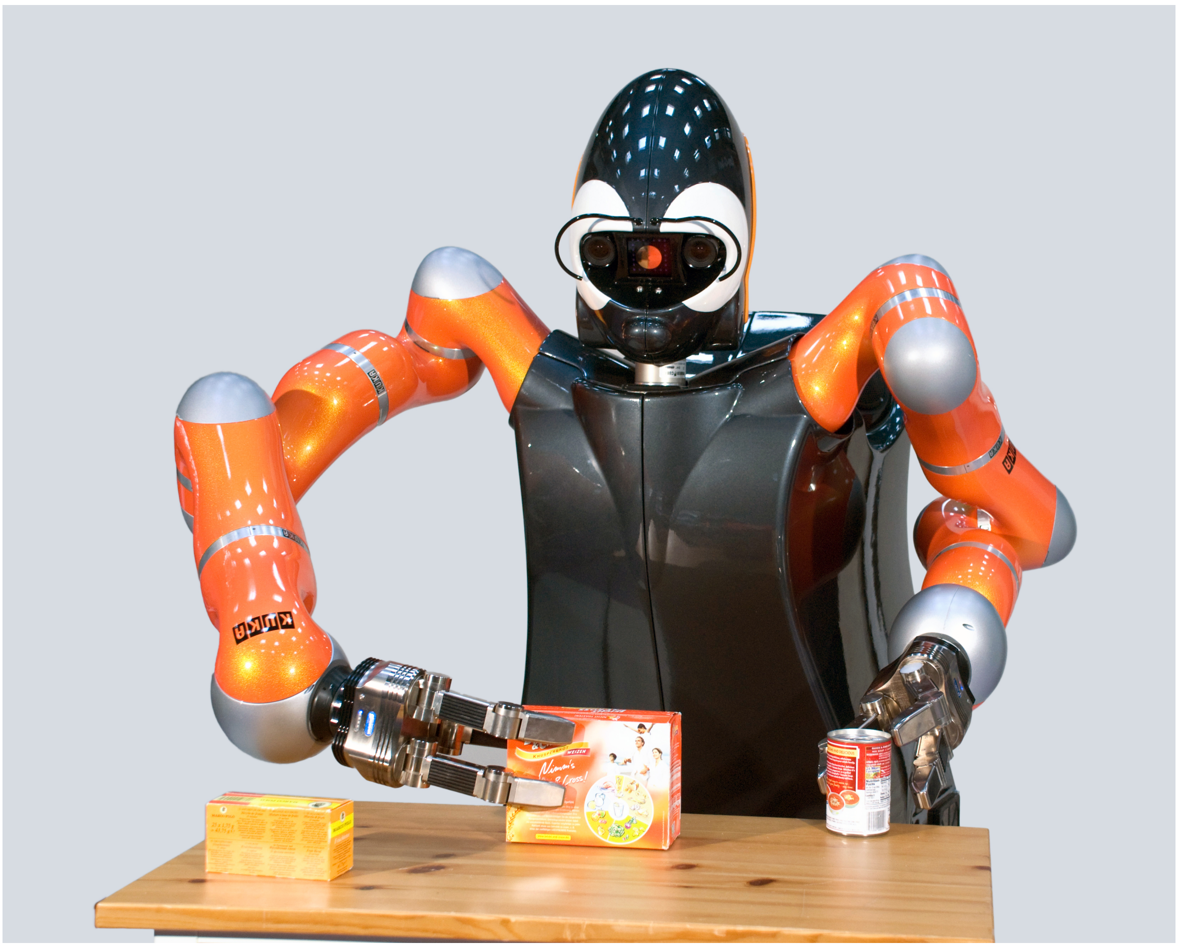


Figure A.1.: The picture shows the DESIRE robot on picking up two objects of a simple scenario at the same time.

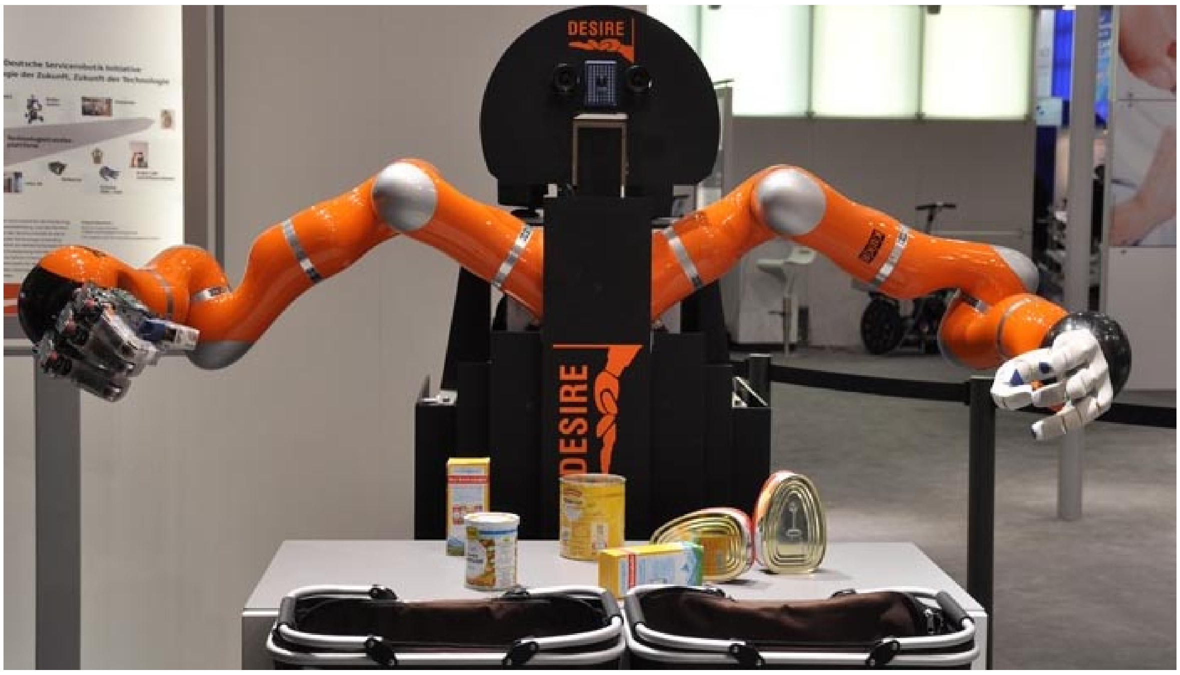


Figure A.2.: The picture shows an older model of the DESIRE robot. This one was presented on the CeBIT 2009.

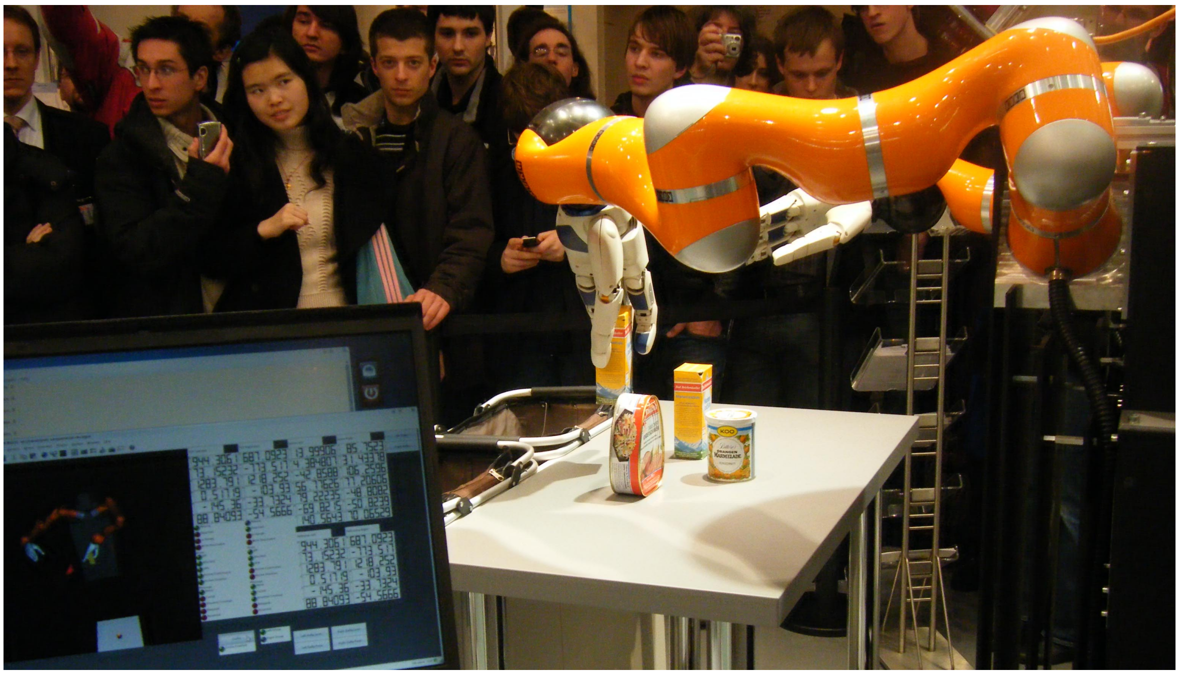


Figure A.3.: The picture shows the DESIRE robot sorting trash. The robot picks the objects up and throws them into one of the bags on the back side of the table. Which bag is chosen depends on whether the object is empty or not.

Bibliography

[1] D. C. Alexander and B. F. Buxton. Modelling of single mode distributions of colour data using directional statistics, 1997.

[2] Eric C. Anderson. Monte carlo methods and importance sampling. Lecture Notes, 2001.

[3] Erhan Ata and Yusuf Yayli. Dual unitary matrices and unit dual quaternions. *Differential Geometry - Dynamical Systems*, 10:1–12, 2008.

[4] Christopher M. Bishop. *Pattern Recognition and Machine Learning.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[5] Su Bang Choe. *Statistical analysis of orientation trajectories via quaternions with applications to human motion.* PhD thesis, UNIVERSITY OF MICHIGAN, 2006.

[6] Sanjoy Dasgupta. Learning mixtures of gaussians. Technical report, University of California at Berkeley, Berkeley, CA, USA, 1999.

[7] Robert Eidenberger et al. Fast parametric viewpoint estimation for active object detection. Part of Roberts thesis, August 2008.

[8] Wendelin Feiten, Pradeep Atwal, Robert Eidenberger, and Thilo Grundmann. 6d pose uncertainty in robotic perception. In Torsten Kröger and Friedrich M. Wahl, editors, *Advances in Robotics Research*, pages 89–98. Springer Berlin Heidelberg, 2009.

[9] Alison L. Gibbs and Francis E. Su. On choosing and bounding probability metrics. *Internat. Statist. Rev.*, pages 419–435, 2001.

[10] Jared Glover. Bingham mixture models of quaternions for object orientation estimation. preprint of "Monte Carlo Pose Estimation with Quaternion Kernels and the Bingham Distribution", 2009.

[11] J. S. Goddard and M. A. Abidi. Pose and motion estimation using dual quaternion-based extended kalman filtering. *SPIE Conf. on Three-Dimensional Image Capture and Applications*, 3313:189–200, January 1998.

[12] James Samuel Goddard. *Pose And Motion Estimation From Vision Using Dual Quaternion-Based Extended Kalman Filtering.* PhD thesis, University of Tennessee, Knoxville, 1997.

[13] *Monte Carlo Pose Estimation with Quaternion Kernels and the Bingham Distribution*, Los Angeles, 2011. Submitted to Robotics: Science and Systems (RSS 2011).

[14] Hyoung joo Lee and Sungzoon Cho. Combining gaussian mixture models. *Lecture Notes in computer Science*, 3177:666–671, 2004.

[15] *A New Extension of the Kalman Filter to Nonlinear Systems*, 1997.

[16] L Kavan, S Collins, Carol O'Sullivan, and J Zara. Dual quaternions for rigid transformation blending. *Technical report TCDCS200646 Trinity College Dublin*, 2009.

[17] *Computing the Kullback-Leibler Divergence Between Probabilistic Automata Using Rational Kernels*, 2006.

[18] E. Kraft. A quaternion-based unscented kalman filter for orientation tracking. *Information Fusion*, 1:47–54, 2003.

[19] Achilleas Lazopoulos. Error estimates in monte carlo and quasi-monte carlo integration. *Acta Physica Polonica B*, 35(11), October 2004.

[20] Taeyoung Lee, Melvin Leok, and N. Harris McClamroch. Global sympletic uncertainty propagation on so(3). *Decision and Control*, 47th IEEE Conference:61–66, December 2008.

[21] G. Peter Lepage. A new algorithm for adaptive multidimensional integration. *Journal of Computational Physics*, 27(2):192–203, May 1978.

[22] J. J. Love. Bingham statistics. In *Encyclopedia of Geomagnetism & Paleomagnetism*, pages 45–47. Springer, Dordrecht, The Netherlands, 2007.

[23] K. V. Marida, C. C. Taylor, and G.K. Subramaniam. Protein bioinformatics and mixtures of bivariate von mises distributions for angular data. *Biometrics*, 63(2):505–512, June 2007.

[24] *High Accuracy Optical Flow Serves 3-D Pose Tracking: Exploiting Contour and Flow Based Constraints*, 2006.

[25] William H. Press and Glennys R. Farrar. Recursive stratified sampling for multidimensional monte carlo integration. *Comput. Phys.*, 4(2):190–195, February 1990.

[26] Olinde Rodrigues. Transformation groups. *Annales de mathématiques pures et appliquées*, 5, 1810.

[27] Andrew R. Runnalls. A kullback-leibler approach to gaussian mixture reduction. *IEEE Transactions on Aerospace and Electronic Systems*, 43(3):989–999, 2007.

[28] H. Schaeben. Parametrizations and probability distributions of orientations. *Textures and Microstructures*, 13(1):51–54, 1990.

[29] J. Stuelpnagel. On the parameterization of the three-dimensional rotation group. In *NASA Technical Documents*. NASA, January 1948.

[30] Seth Teller and Matthew E. Antone. Robust camera pose recovery using stochastic geometry. Technical report, Proceedings of the AOIS-2001, 2001.

[31] Eric Veach. *Robust Monte Carlo Methods for Light Transport Simulation*. PhD thesis, Stanford University, December 1997.

[32] Josie Wernecke. *The Inventor Mentor: Programming Object-Oriented 3d Graphics with Open Inventor, Release 2*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1994.

[33] M.D. Wheeler and Katsushi Ikeuchi. Iterative estimation of rotation and translation using the quaternion. Technical Report CMU-CS-95-215, Computer Science Department, Pittsburgh, PA, 1995.

[34] Jason L. Williams. Gaussian mixture reduction for tracking multiple maneuvering targets in clutter. Master's thesis, Air Force Inst Of Tech Wright-Patterson AFB OH School Of Engeneering And Management, March 2003.

List of Figures

Declaration

Hiermit erkläre ich ehrenwörtlich, dass ich die hier vorliegende Diplomarbeit selbständig verfasst und nur die ausgewiesenen Quellen verwendet habe.

....

Muriel Lang, München, den 24.01.2011