

EBERHARD-KARLS-UNIVERSITÄT TÜBINGEN
Wilhelm-Schickard-Institut für Informatik
Lehrstuhl Kognitive Systeme

Master Thesis

Robot Arm Tracking with Random Decision Forests

Felix Widmaier

Reviewer: Prof. Dr. Andreas Zell
*Wilhelm-Schickard-Institute for Computer Science
University of Tübingen*

Prof. Dr. Andreas Schilling
*Wilhelm-Schickard-Institute for Computer Science
University of Tübingen*

Supervisor: Dr. Jeannette Bohg
*Autonomous Motion Department
Max-Planck-Institute for Intelligent Systems
Tübingen*

Started: 4th November 2014

Ended: 1st May 2015

Erklärung

Hiermit erkläre ich, dass ich diese schriftliche Abschlussarbeit selbstständig verfasst habe, keine anderen als die angegebenen Hilfsmittel und Quellen benutzt habe und alle wörtlich oder sinngemäß aus anderen Werken übernommenen Aussagen als solche gekennzeichnet habe.

Tübingen am 1. Mai 2015

Felix Widmaier

Abstract—For grasping and manipulation with robot arms, knowing the current pose of the arm is crucial for successful controlling its motion. Often, pose estimations can be acquired from encoders inside the arm, but they can have significant inaccuracy which makes the use of additional techniques necessary.

In this master thesis, a novel approach of robot arm pose estimation is presented, that works on single depth images without the need of prior foreground segmentation or other preprocessing steps.

A random regression forest is used, which is trained only on synthetically generated data. The approach improves former work by Bohg et al. by considerably reducing the computational effort both at training and test time. The forest in the new method directly estimates the desired joint angles while in the former approach, the forest casts 3D position votes for the joints, which then have to be clustered and fed into an iterative inverse kinematic process to finally get the joint angles.

To improve the estimation accuracy, the standard training objective of the forest training is replaced by a specialized function that makes use of a model-dependent distance metric, called DISP.

Experimental results show that the specialized objective indeed improves pose estimation and it is shown that the method, despite of being trained on synthetic data only, is able to provide reasonable estimations for real data at test time.

Acknowledgements

Many thanks to all the people in the AMD lab who were very kind and made this a nice and interesting half a year that passed just too quickly. Special thanks goes to Jeannette who did a great job in supervising me.

Furthermore, I want to thank Stefan, Conny and Chris for proofreading (or at least trying to do) and Alina for both proofreading and bearing me even in the stressful final spurt.

Last but not least thanks to Prof. Zell for reviewing this work despite of not being happy with my choice.

Contents

1. Introduction	1
1.1. Related Work	2
1.2. Main Contributions	5
1.3. Notation and Nomenclature	5
2. Foundations	7
2.1. Random Decision Forests	7
2.1.1. Decision Trees	7
2.1.2. From Tree to Forest	10
2.1.3. Regression Forest	11
2.1.4. Density Forest	13
2.1.5. Properties and Important Parameters of Random Forests	13
2.2. C-DIST: An efficient Algorithm to compute the DISP metric	15
2.2.1. Motivation	15
2.2.2. Formal Definition	16
2.2.3. The C-DIST Algorithm	17
2.2.4. Modification of the original C-DIST Algorithm	20
2.3. Spectral Clustering	21
2.4. Robot Arm Pose Estimation through Pixel-Wise Part Classification	23
2.4.1. Generating Training Data	23
2.4.2. Depth Image Features	23
2.4.3. Training	24
2.4.4. Prediction	25
3. Methodology	27
3.1. Random Forest Split Criteria	30
3.1.1. Mean Squared Error (MSE)	30
3.1.2. Mean Squared Pairwise DISP (MSPD)	31
3.1.3. Spectral Cluster based Hamming Distance Score	31
3.2. Confidence-weighted Predictions	33
3.2.1. Using the Confidence for Prediction	33
3.2.2. Joint-wise Confidence	34
3.3. Select and Combine Votes of Individual Pixels	34
3.3.1. Histogram Prediction	35
3.3.2. Confidence-weighted Mean	35

4. Results	37
4.1. Experiment Set-up	37
4.1.1. Hardware	37
4.1.2. The ARM Robot	37
4.1.3. Data Sets used for Evaluation	38
4.1.4. Evaluation methods	39
4.2. C-DIST Benchmark	40
4.3. DISP-based Clustering of Arm Poses	41
4.3.1. K-Means Clustering with DISP Distance	41
4.3.2. DISP-based Density Trees	43
4.4. Finding Good Parameters for Training and Estimation	47
4.4.1. RBF-Kernel for Spectral Clustering	47
4.4.2. Finding Forest Parameters with Randomized Grid Search	48
4.4.3. Pixel Confidence Threshold	48
4.5. Comparison of Split Criteria	48
4.5.1. Tests with larger dataset	54
4.6. Prediction on images	54
4.7. Testing on real data	55
5. Discussion	59
5.1. C-DIST	59
5.2. Split Criteria	59
5.3. Use of Confidences	60
5.4. Comparison to the Former Approach	61
5.5. Runtime	61
6. Conclusions and Future Work	65
A. Appendix	67
A.1. Implementation Details	67
A.1.1. Software Libraries	67
A.1.2. Code Optimization	67
A.2. Visualization of Bounding Volume Hierarchies	68
A.3. Note on the enclosed CD-ROM	68
Abbreviations	71
Bibliography	73

1. Introduction

For autonomous grasping and manipulation in robotics, knowing the current pose of the robot’s manipulator is of crucial importance in order to control its movements on the desired trajectories. The actual pose of each link, including the manipulator, can be computed if the kinematic tree of the robot and the angular position for each joint are known. While the kinematic tree is usually given, getting good estimates for the joint angles can be difficult. Encoders in the arm can provide position estimates, but, depending on the robot, they can have significant inaccuracies. This is illustrated in Figure 1.1 for the ARM robot, which is described in Section 4.1.2. In such cases, additional techniques for estimating the arm pose are often necessary.

One way to tackle this problem is to track the arm using visual sensors. This is done, for example, by attaching markers to the arm, which can easily be detected in camera images, or by using the model of the arm to detect it directly without the need for markers. Many recent approaches use depth images, where for each pixel a distance measurement is given. Such depth images can for example be obtained by 3D laser scanners, stereo cameras or RGB-D cameras like the Microsoft Kinect or Asus Xtion. The latter were originally developed for game consoles, but have quickly become very popular in robotics research, as they provide dense point clouds and are very cheap, compared to other range sensors.

Detecting the robot arm in a camera image is one instance of the more general problem of *object detection*. In the past, such object detection tasks were often solved by developing algorithms that are tailored for the specific problem, i.e. models for the detection were constructed manually. In recent years, due to increasing computing power and memory capacity of modern computers, machine learning solutions have become more and more popular, which learn the models automatically. The same learning algorithms can be applied to many different tasks, while only the training data has to be changed for the specific task (leaving one with the problem of acquiring a suitable and large enough training set, though).

In this thesis a new approach of visual robot arm tracking is presented, which uses a machine learning technique called *random decision forest* to estimate joint configurations from depth images provided by a head-mounted RGB-D camera. Random decision forests are ensembles of decision trees that are trained independently of each other using different subsets of the training data. They are especially suitable for real-time critical tasks as they are fast and highly parallelizable.

The approach is an extension of former work by Bohg et al. [BRHS14]. In [BRHS14], a random decision forest was used for pixel-wise part classification of the robot arm based on simple depth features. For non-background pixel the forest also casts 3D position votes for the single joints which are clustered and used

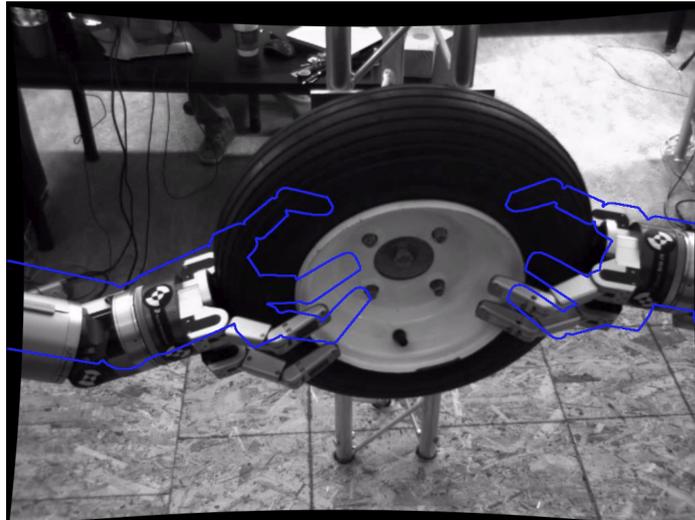


Figure 1.1.: The arms of the ARM robot holding a wheel. The estimated arm poses based on the encoder readings of the arm are visualized by the blue silhouette.

to compute the joint angles with inverse kinematics. For the training, synthetically rendered depth images were used, rather than real sensor data, which allows generation of huge training sets with only little effort.

The new approach of this thesis reuses the synthetic training data of [BRHS14] and improves the method, as it significantly reduces the computational effort for pose estimation. Instead of a classification forest, a regression forest is trained, which directly estimates joint angles from the given depth features. It thus avoids the expensive intermediate step of clustering position votes and applying inverse kinematics to get joint angle estimations. Further the training objective of the random forest was changed: instead of the entropy of the part labels, distances between the arm poses of the training samples are minimized, using a special, model-dependent distance metric for rigid body displacement, called *DISP*.

The thesis is structured as follows: In the remainder of this chapter, related works are discussed and the main contributions are illustrated. In Chapter 2, the foundations for the newly developed methods are given, which are described in chapter 3. Chapter 4 summarizes the experimental results. In Chapter 5 these results are discussed and in Chapter 6 conclusions and an outlook on future work are given.

1.1. Related Work

Robot arm tracking. For doing manipulation with a robot arm, it is of importance to know the actual pose of the manipulator as good as possible. Joint encoders can give an estimate of the manipulators pose, but they can have significant errors which are on the one hand caused by inaccuracy in production and on the other

by limitations of the system itself. An example is the ARM robot (see Section 4.1.2) which has only position encoders in the motors, not in the joints directly. Movement is transferred from the motors to the joints by steel cables, which encounter variable stretch, depending on the actual pose and the load of the arm. This leads to variability in the joint position that cannot be perceived by the encoders in the motors. As the error of the individual joints sums up when going along the kinematic chain, one can end up with an position error of several centimetres at the manipulator (see Figure 1.1). This presents a huge challenge for fine-manipulation tasks such as key in lock or screwing on a lid on a jar.

One attempt to solve this problem is known as *visual servoing*: The manipulator is tracked by a vision system (mono or stereo camera, depth camera, ...). It is then controlled to minimize the error between detected and desired position in the image of the camera.

In such cases, it is common practice to attach markers (e.g. coloured spheres or LEDs) to the robot arm, which are easy to detect in camera images [VWA⁺08, GBBK10]. While this approach is easy to implement, it has the disadvantage, that the markers have to be inside the area that is covered by the camera at any time and must not be occluded.

Another approach is to take the 3D model of the robot into account and detect the arm pose by aligning the observed arm in the image with its reprojection which is rendered based on the model. This is known as *virtual visual servoing* and has been applied successfully in [GRBK12, HHM⁺12, KHRF11]. All three approaches use different variants of the *iterative closest points* (ICP) method to incrementally align observed and rendered points.

A different model-based method was recently proposed in [SNF14]. They use a modified extended Kalman filter that maximizes the log-likelihood of the observed depth image in each iteration. This likelihood is based on a signed distance to the model (positive for points above the surface, negative for points below). Using an optimized GPU implementation they achieved real-time performance on a simple visual servoing problem.

Many of the methods mentioned here track the body over time, that is each new pose estimation is based on the former one. While this approach exploits useful knowledge about the former state, such systems require a good initialization and may not be able to recover, if they loose track of the body, e.g. due to temporary occlusion. The approach presented in this thesis does not depend on former states or initialization, as it estimates the robots pose independently for each frame.

Random Decision Forest in computer vision. In [BRHS14] an approach of robot arm pose estimation is presented, which uses a random decision forest for pixel-wise part classification of the arm in depth images¹. The approach was inspired by [SGF⁺12], where a similar method was used to estimate human poses in depth

¹As this work is an extension of [BRHS14], the methods they developed are described more elaborately later in Section 2.4

images provided by a Kinect, which actually runs in real-time on the Xbox 360 gaming console.

The general idea of using an ensemble of multiple randomly trained decision trees was first reported in the 1990s by Amit and Geman [AG94] who used them for handwritten digit recognition. The term “random decision forest” came up shortly after in work of Ho who also applied a ensemble of decision trees on a digit recognition problem [Ho95]. Further important work on random forests was done in the following years by Breiman [Bre01] who also trademarked the term “random forest”.

Since then, random forests have been successfully applied on a wide range of vision problems like keypoint detection [LF13, SMA⁺11], object detection [GL13], image segmentation [SJC08, DMWG09] or object tracking [GRB13]. Some of their advantages are the ability to naturally handle multi-class problems and the fast run-time compared to other machine learning techniques, which makes them especially suitable for many real-time applications.

Metrics for rigid body displacement. In order to train a random forest on robot arm poses, some distance metric is needed, that computes the distance between two different arm configurations. For this thesis, this distance was measured in the workspace of the robot, given an articulated model of the arm.

The problem of defining a distance metric for transformations on rigid and articulated bodies is not new. It is, for example, important for tasks like path planning and collision detection. The difficulty consists in combining translation and rotation. They have different units and are thus not directly comparable. Furthermore, rotations have the additional challenge of periodicity, which makes defining a distance on rotations alone already a non-trivial task. Accordingly, there are several different attempts to define a metric on rotations [Kuf04, LaV06]. Some approaches to define a metric in $SE(3)$ compute distances on translation and rotation independently and combine the results in a weighted sum [Kuf04]. This is easy to compute, but leaves the user with the task to specify meaningful weights.

Other approaches take the model of the object, that is transformed, into account and define the metric in the workspace rather than the configuration space. One family of such model-dependent metrics measures the swept-volume, that is the volume which is covered when the model is moved (“swept”) from one configuration to the other. [KVL03] present an algorithm to approximate this swept-volume, which is, however, still relatively slow. Another attempt to compute distance based on the swept-volume is made in [Xav97]. While it is exact on translations, the rotational part is only approximated.

Another family of model-dependent metrics looks at the vertices of the mesh model of the object. The so called *DISP distance* [ZKM07, Lat91, LaV06] is defined as the length of the longest displacement vector among all points of the model. [ZKM07] proposed an algorithm for fast DISP computation. Other metrics using model points are for example the *Hausdorff distance* [Lat91] or the one proposed in [HP04] where only the displacement of a set of feature points is computed.

In this thesis, the DISP distance was used to measure the distance between different robot arm configurations, as it is comparatively fast to compute, using the C-DIST algorithm of [ZKM07], and naturally handles both translation and rotation without the need of defining weights.

1.2. Main Contributions

In this thesis, a novel approach of robot arm pose estimation is presented, that uses random decision forests to obtain joint configuration estimates from single depth images. The idea is based on former work by Bohg et al. [BRHS14] (see also Section 2.4) but improves their method as it drops the intermediate steps of classifying pixels as belonging to specific robot arm parts, casting and clustering 3D position votes for the joints and finally computing the joint configurations with inverse kinematics.

Instead, a regression forest was used, that directly estimates joint configurations. To improve the estimation quality of the forest, a model dependent distance metric called DISP [ZKM07] was used to implement a new training objective for the node splits of the decision trees. DISP measures displacement between different arm poses in the workspace of the robot, rather than in the configuration space. Thus it resolves the problem that the movement of a single joint has a different impact on the overall arm displacement, depending on which joint is chosen (see Section 2.2.1).

For each pixel of the depth image, the forest gives a pose estimation for the arm. The estimations of the individual pixels are then combined using a confidence-weighted mean. The use of confidences automatically favours low-error pixels and makes a prior background segmentation unnecessary. In consequence, the forest operates directly on the unmodified depth image using simple, fast computable depth features.

Like in [BRHS14], the forest is trained on synthetically rendered images rather than data from a real sensor. This makes building a large annotated training set easy.

Experimental results show the accuracy of the estimation and that the method can provide meaningful results also on real images, despite being trained only on synthetic data. The implementation for this thesis does not yet achieve real-time performance but with a better optimized implementation, this should generally be possible.

1.3. Notation and Nomenclature

Some notes on notation and nomenclature in this thesis:

- The denominations “random decision forest”, “random forest”, “decision forest” or just “forest” are used interchangeably and all refer to the same thing.

1. Introduction

- When speaking of joint positions, one has to distinguish between “3D positions” $\mathbf{p} \in \mathbb{R}^3$ which denote the position of the joint in a three dimensional Euclidean space, and “angular positions”, which denote the configuration of the joint given as an angle θ .
- The *workspace* of the robot is the Euclidean space \mathbb{R}^3 it operates in. 3D positions of joints and surface points of the robot model are given in this space.
- The *configuration space* denotes the angular space in which the configuration of the joints of the robot arm is given.
- The terms “estimation” and “prediction” are used more or less interchangeably throughout this thesis.
- Scalar variables are denoted by lower case letters (s), vectors by bold lower case letters (\mathbf{v}) and matrices by bold upper case letters (\mathbf{M}).
- θ always denotes a joint angle.
- φ denotes a feature value.
- Data points are denoted by \mathbf{x} and are represented by a vector of feature values, i.e. $\mathbf{x} = (\varphi_1, \dots, \varphi_n)$. For training data, each feature vector \mathbf{x} is paired with an target value y which is a discrete label for classification tasks or a (possibly multi-dimensional) continuous value for regression tasks. In the actual case of joint configurations, $\mathbf{y} = (\theta_1, \dots, \theta_m)$ is a vector of angular joint positions, where θ_i denotes the angular position of the i -th joint.

2. Foundations

In this chapter, the main foundations for this work are presented. Section 2.1 gives an introduction to random forests. In Section 2.2, the DISP distance is defined and the C-DIST algorithm for fast DISP computation is presented. Section 2.3 explains the idea of spectral clustering, which was used to accelerate the random forest training. Finally, Section 2.4 gives a brief overview over the former work of Bohg et al. which is extended in this thesis.

2.1. Random Decision Forests

Over the last two decades ensembles of multiple decision trees, called *random decision forests* (or *decision forests*, *random forests*, *randomized trees*, . . .), have become a popular machine learning technique. They are easy to understand, yield good generalization, are fast and highly parallelizable. Their fast prediction makes them especially suitable for real-time critical tasks for example in computer vision.

This section gives a overview of decision forests by first introducing single decision trees for classification in Section 2.1.1 and then combining them into forests in Section 2.1.2. Section 2.1.3 and 2.1.4 show how the classification model can easily be adapted for regression and density estimation. Finally, Section 2.1.5 summarizes some important properties of the decision forest model.

The forest model described in this section is primarily based on the work of Criminisi and Shotton [CSK12, CS13].

2.1.1. Decision Trees

Decision trees have been a well-known model for classification and regression tasks for many years. Given a set of training samples, they split the samples at each node, based on some feature values. Splits are chosen such that the samples which end up in the same leaf are ideally very similar to each other. At test time, unknown samples descend the tree following the same split rules. Reaching a leaf node, a sample is classified based on the training samples in this leaf. These processes are described in more detail in the following.

Classification

To classify a given data sample x , the sample traverses the tree. At each node one feature of x is compared to a threshold stored in the node. Depending on whether

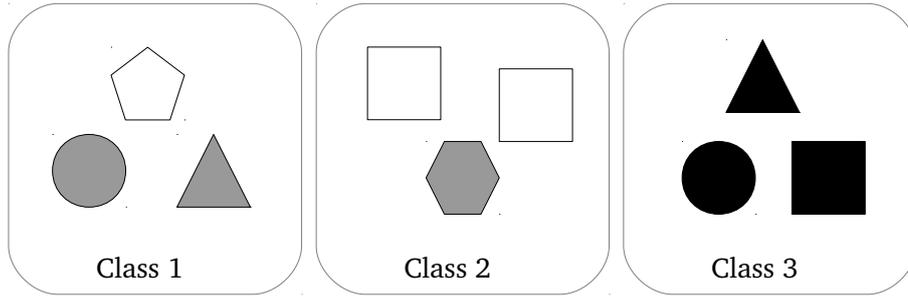


Figure 2.1.: A simple dataset with three classes. Two features are used: number of corners (set to 0 for circles) and brightness (black = 0, grey = 0.5, white = 1).

the feature value is below or above the threshold, the sample descends to the left or the right child. Each leaf holds a class label. When the sample reaches a leaf, it is assigned to the class of this leaf.

Figure 2.1 shows a small exemplary dataset. The data points are geometric shapes that are partitioned into three classes. The used features are “number of corners” (0–6) and “brightness” (black = 0, grey = 0.5, white = 1). For example, the triangle in class 1 is represented by the feature vector (3, 0.5). Figure 2.2 shows a classification tree that is based on this dataset. The test data point (grey square) is represented by the feature vector (4, 0.5). It descends from the root along the highlighted path to a leaf that is assigned to class 1. Thus the grey square is classified as belonging to class 1.

Training of a decision tree

The example tree in Figure 2.2 was constructed by hand, but usually the structure and parameters of the tree are determined automatically based on a set of training data. The basic idea is to split the data at each node, so that the samples ending up in the same leaf node are as *pure* as possible (in terms of class membership).

Let $S = \{(\mathbf{x}_1, y_1), \dots, (\mathbf{x}_n, y_n)\}$ denote a set of training samples. Each sample consists of a d -dimensional feature vector $\mathbf{x} = (\varphi_1, \dots, \varphi_d)$, $\varphi_i \in \mathbb{R}$ and a discrete class label $y \in \mathcal{C}$, where \mathcal{C} denotes the set of all class labels. For ease of notation, for a given S we define $S^{\mathbf{x}}$ and S^y as follows:

$$S^{\mathbf{x}} = \{\mathbf{x} | (\mathbf{x}, y) \in S\} \quad (2.1)$$

$$S^y = \{y | (\mathbf{x}, y) \in S\} \quad (2.2)$$

A split $\chi = (i, \tau)$ consists of a feature index $i \in \{1, \dots, d\}$ and a threshold $\tau \in \mathbb{R}$. Applying a split χ on a sample set S results in a partitioning (S_l, S_r) of S where

$$S_l = \{(\mathbf{x}, y) | (\mathbf{x}, y) \in S, \mathbf{x} = (\varphi_1, \dots, \varphi_d), \varphi_i < \tau\} \quad (2.3)$$

$$S_r = \{(\mathbf{x}, y) | (\mathbf{x}, y) \in S, \mathbf{x} = (\varphi_1, \dots, \varphi_d), \varphi_i \geq \tau\} \quad (2.4)$$

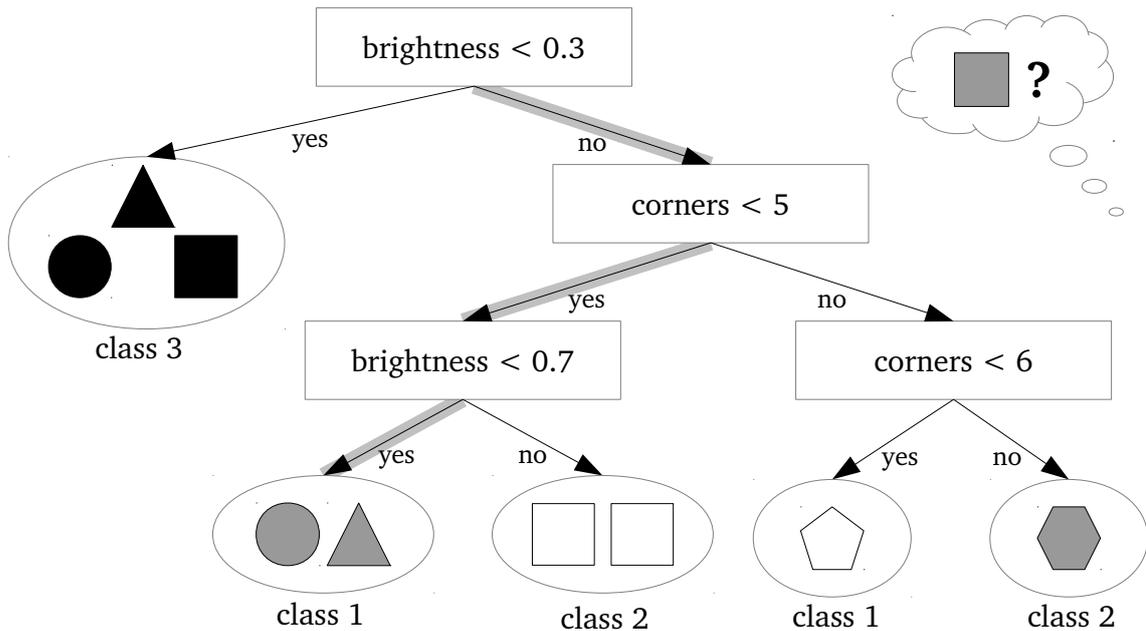


Figure 2.2.: A decision tree, that classifies shapes based on the data in Figure 2.1. The marked edges show the path the new unknown sample (grey square) takes from the root to the leaf.

Note that $S_l \cup S_r = S$ and $S_l \cap S_r = \emptyset$. Literally spoken, all samples with $\varphi_i < \tau$ go to the left and the others to the right side.

Starting at the root node of the tree, a number of different, randomly chosen split candidates χ are tested on S and scored using a scoring function $f_{score}(S, \chi)$. Let X denote the set of all splits that are tested. Let χ^* be the split that maximizes f_{score}

$$\chi^* = \arg \max_{\chi \in X} f_{score}(S, \chi) \quad (2.5)$$

χ^* is used to split S into S_l and S_r . It is further stored with the node and S_l and S_r are pushed to the left and right child node, where the splitting process continues recursively.

How many different splits are tested at each node and how they are chosen depends on the actual implementation. For example, the random forest of the scikit-learn library [PVG⁺11] tests a given number of randomly chosen feature components and does an exhaustive search over all possible thresholds for each of these features.

A node is not split any further, if some termination criterion is met, typically if a maximum tree depth is reached or if the number of samples in the node is below some limit. Such a terminating node becomes a leaf of the tree and the class label that is most prominent in the leaf samples is stored with the leaf.

The split scoring function. The scoring function f_{score} can in general be any function that quantifies the similarity of the samples within the resulting subsets of the split. A very common choice is to use the concept of *information gain*.

The information gain of a node split is defined as “the reduction in uncertainty achieved by splitting the training data arriving at the node into multiple child subsets” [CS13]. For a binary split, this is represented by the following equation:

$$I(S, S_l, S_r) = H(S) - \sum_{i=l,r} \frac{|S_i|}{|S|} H(S_i) \quad (2.6)$$

where $H(S)$ is the *entropy* of the set S . The precise definition of the entropy is dependent of the actual task. For the purpose of classification [CS13] uses the Shannon entropy:

$$H(S) = - \sum_{y \in \mathcal{C}} p(y) \log(p(y)) \quad (2.7)$$

Variations

The decision tree model described above can be extended in various ways.

- The splitting can be done in a more sophisticated manner by taking more than one feature into account and using a hyperplane or even a non-linear function to split the samples.
- Instead of only the most prominent class label, leafs can store distributions over all leaf samples. Doing so, one is able to make probabilistic estimations and take uncertainty into account.
- Dependent on the actual application, the information gain score for finding the best splits can be replaced by other scoring functions that are better fitted for the specific problem domain.

Apart from this, it is also possible to apply decision trees to other tasks like regression or density estimation with only little changes. This is discussed below in Section 2.1.3 and 2.1.4.

2.1.2. From Tree to Forest

A drawback of decision trees is that they are prone to overfitting, especially if they get too deep [Ho95]. Combining multiple different trees into an ensemble—a forest—turned out to lead to a much better generalization capability compared to single trees. More specifically, it has been shown that the accuracy of a forest increases monotonically with the number of trees [CS13]. The changes in the training and prediction procedure when using multiple trees instead of only one are straightforward.

Training.

The trees in the ensemble are trained independently of each other as described above. Instead of training each tree on the full training set, often a technique called *bagging* is applied, where each tree is trained on a different set that is generated by drawing with replacement from the original training set. This reduces the correlation between the individual trees and has been shown to improve generalization [CS13]. Note that due to the replacement, it is possible that one sample appears multiple times in the set of a tree, giving this sample a higher weight. Therefore, this method can be understood as assigning random weights to the training samples.

Prediction.

For predicting the class of a data point x , each tree is applied on x independently. Given that the forest consists of T trees, this results in T individual predictions for x . In the simplest case, when one is only interested in the most likely class, the final classification output of the forest is determined by taking the class that got the most votes from the single trees. If probability distributions are desired, the output distribution can be computed by summing or multiplying the output distributions of the different trees.

As the trees are completely independent of each other, runtime both during training and prediction can be accelerated significantly by processing the single trees in parallel.

So far, only classification was taken into account. However, the forest model is not restricted to this and can be applied to other problem tasks with only small modifications. In the following, modifications necessary for regression and density estimation tasks are described. Both were used in this thesis; density estimation to get a first idea of how well the DISP metric described in Section 2.2 is suited for node split functions, and regression for the actual pose estimation of the robot arm.

2.1.3. Regression Forest

The random forest model described so far is designed for classification problems. There is, however, only little change necessary to adapt the forest for regression problems.

The essential difference between classification and regression is, that the target value of the prediction changes from a discrete class label $y \in \mathcal{C}$ to a continuous value $y \in \mathbb{R}$. While the general structure of the forest stays the same for regression, the data representation at the leaf nodes and the split function for training change a bit, as is described in the following.

Prediction

To get a prediction for a new data point \mathbf{x} , this point is pushed through all trees of the forest, just as for the classification task. The difference shows up at the leaf nodes: Instead of a discrete class label or a distribution over multiple such labels, a probability density distribution over the *continuous* variable y is stored at the leaf. An easy example would be to use Gaussians, but of course, more complex distributions are also possible. The output of the t -th tree for input \mathbf{x} is then the distribution $p_t(y|\mathbf{x})$. The final output of the forest is the average over all T trees:

$$p(y|\mathbf{x}) = \frac{1}{T} \sum_{t=1}^T p_t(y|\mathbf{x}) \quad (2.8)$$

If one is not interested in probabilities, it is sufficient to store only one output value at the leaf, for example the mean of the training samples that ended up in this leaf. The output of the forest is again obtained by averaging the outputs of the individual trees.

Training

The method to find good splits at tree nodes during training changes a bit compared to classification forests. While the general idea of maximizing the information gain as stated by Equations 2.5 and 2.6 can be reused, the definition of the entropy H has to be adapted. Assuming Gaussian distribution of the data, the entropy can be defined as

$$H(S) = \frac{1}{|S|} \sum_{\mathbf{x} \in S^{\mathbf{x}}} \frac{1}{2} ((2\pi e)^2 \sigma_y^2(x)) \quad (2.9)$$

where $\sigma_y^2(x)$ is a conditional variance that is computed by fitting a linear probabilistic model to the data. See [CS13, page 52] for details.

Plugging this into Equation 2.6 and extending it to multidimensional input, the information gain for regression looks as follows:

$$I(S, S_l, S_r) = \sum_{\mathbf{x} \in S^{\mathbf{x}}} \log (|\Lambda_y(\mathbf{x})|) - \sum_{i=l,r} \left(\sum_{\mathbf{x} \in S_i^{\mathbf{x}}} \log (|\Lambda_y(\mathbf{x})|) \right) \quad (2.10)$$

with $\Lambda_y(\mathbf{x})$ the conditional covariance.

Note: The entropy function described above follows [CS13]. The scikit-learn library [PVG⁺11], which was used for the implementation of this thesis, uses a somewhat simpler approach: $H(S)$ is simply defined as the variance of the data in S^y . In the case of multi-dimensional regression, the average variance over all dimensions is used.

2.1.4. Density Forest

A *random density forest* can be used to estimate the underlying probabilistic density function of given *unlabelled* data samples. In [CSK12] the task is defined as follows:

Given a set of unlabeled observations we wish to estimate the latent probability density function from which such data has been generated.

Training

Since this is an unsupervised learning task (i.e. the data is unlabelled), the score function to find the optimal split has to be adjusted again. The training set S consists now only of the feature values, i.e. $S = S^x = (\mathbf{x}_1, \dots, \mathbf{x}_n)$. The basic function that is optimized is still the generic information gain of Equation 2.6. What changes is the definition of the entropy H :

$$H(S) = \frac{1}{2} \log((2\pi e)^d |\Lambda(S^x)|) \quad (2.11)$$

where d denotes the dimensionality of the data, $\Lambda(S^x)$ the covariance matrix of S^x , and $|\cdot|$ the matrix determinant.

This entropy function assumes that the data in the node is Gaussian distributed. This is not satisfied in general, but it can give a sufficiently good approximation nonetheless, as multiple Gaussians are combined in the hierarchical tree structure.

When plugging Equation 2.11 into 2.6, the objective function that is maximized simplifies to

$$I(S, S_l, S_r) = \log(|\Lambda(S^x)|) - \sum_{i=l,r} \frac{|S_i|}{|S|} \log(|\Lambda(S_i^x)|) \quad (2.12)$$

The training samples that end up in the same leaf node are represented by a multivariate Gaussian distribution $\mathcal{N}(\mu, \Lambda)$ where μ is the mean and Λ the covariance of the samples.

Sampling with the trained forest

Once the forest is trained, it can be used to generate new random samples based on the learned distribution, as described in Algorithm 2.1. This sampling algorithm is used in Section 4.3.2 where density forests are trained on robot arm pose samples.

2.1.5. Properties and Important Parameters of Random Forests

Advantages of random forests over many other machine learning techniques are:

- They are fast both during training and prediction and are highly parallelizable. This makes them attractive for real-time critical applications.

Algorithm 2.1: Sampling from a density forest (taken from [CSK12]).

Given a density forest with T trees:

1. Draw uniformly a random tree index $t \in \{1, \dots, T\}$ to select a single tree in the forest.
 2. Descend the tree by randomly choosing a child at each node with probability proportional to the number of samples in the corresponding subtree.
 3. When reaching a leaf, draw a random sample from the Gaussian distribution in this leaf.
-

- The structure of a tree can easily be analysed as they are much more “human readable” than many other models, for example like artificial neuronal networks.
- Random forests can naturally handle multi-class problems (as opposed to support vector machines, for example).
- The same basic model can easily be adapted for different tasks like classification, regression or density estimation.
- Last but not least, the theory behind random forests is intuitive and easy to learn which makes the hurdle for using them relatively low.

Random forests have only a quite small number of important parameters:

Forest Size: The generalization ability of random forests increases monotonically with the number of trees in the forest [CS13]. Choosing the size of the forest is therefore only a trade-off between prediction accuracy and CPU/memory cost.

Tree Depth: The deeper a decision tree, the more it tends to overfitting, so this parameter should be chosen carefully. The depth can either be limited directly by terminating at a given maximum depth, or indirectly by giving a minimum number of samples in each leaf (i.e. splitting is stopped, if this number would be exceeded by another split). The second method has the advantage, that it automatically adapts the tree size to the size of the training set.

Randomness: The higher the randomness of the trees, the more decorrelated are the single trees which tends to result in better generalization. A simple way to influence the randomness is to set the number of different splits (i, τ) that are tested at each node. The higher this number, the less random are the trees. Another way to increase randomness is to use bagging which is described above in 2.1.2.

2.2. C-DIST: An efficient Algorithm to compute the DISP metric

2.2.1. Motivation

To train a random forest on any data, one needs a metric to measure the distance between two data points. In many cases, a simple metric like the Euclidean distance is a good solution, as it is intuitive and easily computed. In the case of robot arm joint configurations, however, the use of the Euclidean distance in the configuration space, that is on the joint angles, can lead to unintended results. The problem is, that changing the angular position of one single joint by a fixed value can result in considerably different displacements of the end effector of the robot arm, depending on the position of the selected joint in the kinematic chain: Due to the geometry of the arm, changes in joints near to the root have usually a higher impact on the position of the end effector than changes in joints further down the kinematic chain. The Euclidean distance in the configuration space is the same, however, independently of which joint is modified. This is illustrated in Figure 2.3a.

One idea that comes to mind, is to compute the pose of each link relative to a fixed reference frame for the given joint configuration. This is known as *forward kinematic* and can easily be done, if the kinematic model of the robot is known.

A pose in 3-dimensional space can be defined by a translation (t_x, t_y, t_z) and a rotation (ϕ_x, ϕ_y, ϕ_z) given in Euler angles. The poses of all n links could be combined into one large vector

$$\mathbf{p} = (t_x^{(1)}, t_y^{(1)}, t_z^{(1)}, \phi_x^{(1)}, \phi_y^{(1)}, \phi_z^{(1)}, \dots, t_x^{(n)}, t_y^{(n)}, t_z^{(n)}, \phi_x^{(n)}, \phi_y^{(n)}, \phi_z^{(n)})$$

which gives a representation of the full arm pose. Now the difference between two poses \mathbf{p}_1 and \mathbf{p}_2 could be defined as the Euclidean distance between these vectors: $\|\mathbf{p}_1 - \mathbf{p}_2\|$. This would solve the problem described above, since a change in a joint somewhere within the kinematic chain also affects the poses of all following joints. Unfortunately, this is still not a good solution as it introduces a new problem: translation and rotation have different units and are therefore not directly comparable.

A metric that can be used without suffering from these problems, is described by [Lat91, LaV06, ZKM07]. There seems to be no commonly recognized name for this metric. In this thesis, it referred to as “DISP” distance, following the nomenclature of [ZKM07].

Literally spoken, the DISP distance is the length of the longest displacement vector of any point on the surface of the robot arm from one configuration to the other. See again Figure 2.3a: In both examples the vector of maximal displacement is marked with a red line. It is obvious that this metric represents the actual difference between the two configurations much better than the Euclidean distance on the joint angles does.

In the following, the DISP distance is formally defined and the C-DIST algorithm for efficient DISP computation is introduced.

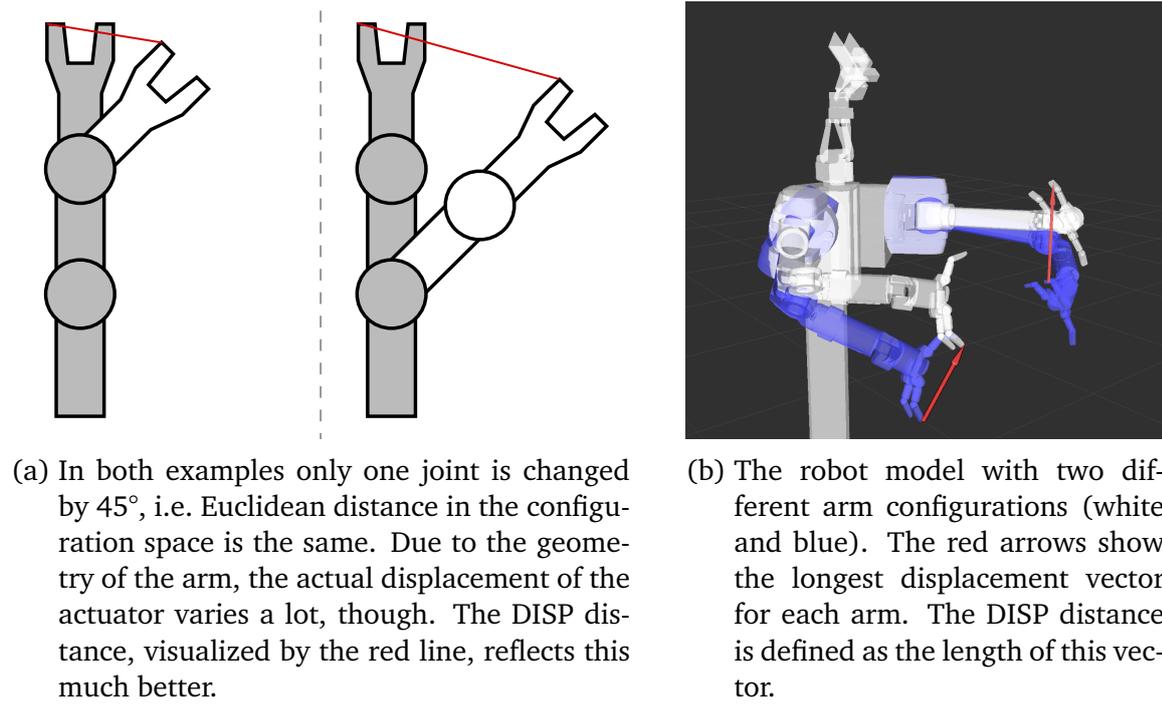


Figure 2.3.: Visualization of the DISP metric

2.2.2. Formal Definition

Given a model A , the DISP distance between two configurations \mathbf{q}_1 and \mathbf{q}_2 of this model is defined as the maximum over the displacements of the individual points of A when going from one configuration to the other:

$$\text{DISP}(\mathbf{q}_1, \mathbf{q}_2) = \max_{\mathbf{p} \in A} \|\mathbf{p}(\mathbf{q}_1) - \mathbf{p}(\mathbf{q}_2)\|_2 \quad (2.13)$$

where $\mathbf{p}(\mathbf{q}_i)$ is defined as the position of the point \mathbf{p} in the configuration \mathbf{q}_i , relative to a fixed reference frame (e.g. the base of the robot).

The authors of [ZKM07] proved, that DISP is a metric, i.e. it satisfies the following properties for all configurations $\mathbf{q}_1, \mathbf{q}_2$:

- Non-negativity: $\text{DISP}(\mathbf{q}_1, \mathbf{q}_2) \geq 0$
- Reflexivity: $\text{DISP}(\mathbf{q}_1, \mathbf{q}_2) = 0 \Leftrightarrow \mathbf{q}_1 = \mathbf{q}_2$
- Symmetry: $\text{DISP}(\mathbf{q}_1, \mathbf{q}_2) = \text{DISP}(\mathbf{q}_2, \mathbf{q}_1)$
- Triangle inequality: $\text{DISP}(\mathbf{q}_1, \mathbf{q}_2) + \text{DISP}(\mathbf{q}_2, \mathbf{q}_3) \geq \text{DISP}(\mathbf{q}_1, \mathbf{q}_3)$

Note that it is negligible here, how the configurations \mathbf{q}_1 and \mathbf{q}_2 are represented, since DISP operates in the workspace of the model rather than its configuration space. The only requirement is, that it is possible to determine the position of each point $\mathbf{p} \in A$ in the given configurations, i.e. $\mathbf{p}(\mathbf{q}_i)$ must be computable.

As an example, in Figure 2.3b the longest displacement vector for the left and right arm from one arm pose to another is visualized.

2.2.3. The C-DIST Algorithm

A trivial “brute force” algorithm to compute DISP for a model A is to simply compute the displacement of every single point and take the maximum of that. While this might be feasible for very simple models, it becomes very expensive when the complexity (i.e. the number of points) of the model increases. Therefore, the C-DIST algorithm for fast DISP computation, proposed by Zhang et al. [ZKM07], was reimplemented for this thesis.

In this section, the algorithm is briefly described and some details are mentioned, where the used implementation diverges from the original algorithm. Benchmark results comparing different variants can be found in Section 4.2.

There are two steps of improvement over the trivial brute force method: First, the authors of C-DIST showed that it is sufficient to consider only the points of the convex hull of the model. Depending on the shape of the model, this can already reduce the number of points to compute by a huge amount. In the second step, they construct a *bounding volume hierarchy* (BVH) for the convex hull which is used to further prune away points.

For a rigid model, both convex hull and BVH do not change and therefore need to be computed only once as a preparation step. Articulated models are treated as a set of rigid models, so the convex hull and BVH are computed separately for each rigid link of the model.

Bounding Volume Hierarchy using Swept Sphere Volumes

A *bounding volume hierarchy* (BVH) is a hierarchical tree structure that recursively splits the points of a model into smaller subsets, holding a bounding volume for points of the corresponding subset at each node [LGLM00].

Given a set P of points, a bounding volume $BV(P)$ is some 3-dimensional geometric shape, that encloses all the points of P . Usually the BV is supposed to be as small as possible, that is it should encapsulate the points without covering more space than necessary. A well known and often used example for such a shape is a rectangular *bounding box*, but in principle any shape can be used.

For a given model A , the root node of the corresponding BVH contains a bounding volume $BV(A)$ for the whole model. Now the points of A are partitioned into two distinct subsets A_1, A_2 , forming the child nodes of the root. This process is continued recursively, that is bounding volumes $BV(A_1)$ and $BV(A_2)$ are computed and the subsets A_1, A_2 , are split further until some termination criterion is met.

For C-DIST, swept sphere volumes (SSVs), as described in [LGLM00], are used as bounding volumes. An SSV is defined by a simple geometric shape C (for example a line), which builds the core of the SSV, and a sphere S . The volume contains all the points that are covered by the sphere when it is swept over C . Formally, this can be defined as the Minkowski sum [Sch13] of C and S .

Three different core shapes are used: *point*, *line* and *rectangle*. The resulting SSVs are referred to as *point swept sphere* (PSS), *line swept sphere* (LSS) and *rectangle swept sphere* (RSS). Figure 2.4 shows how these volumes look like. Note that the

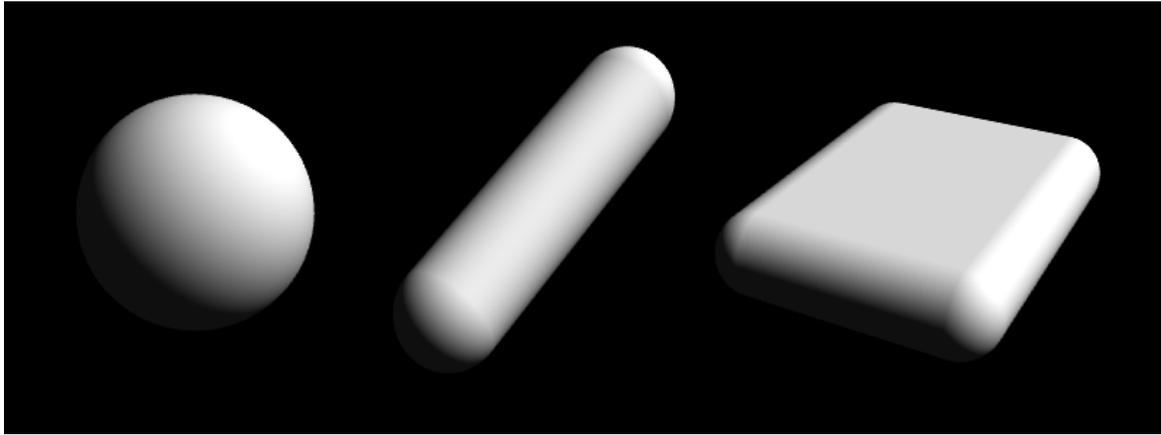


Figure 2.4.: Exemplary shapes of a point, line and rectangle swept sphere. Image taken from [LGLM00].

PSS is in fact just a sphere as the core shape consists of only one point and the sphere cannot be “swept” along it.

When computing the SSV of a set of points, of the three variants the one is chosen which is expected to encapsulate the points most densely. This decision is done heuristically using an oriented bounding box (OBB) that is initially fit to the points. If the three axes of this OBB are approximately of the same length, a PSS is used. If two axes are of the same length and the third is significantly larger, an LSS is used and finally in the case of one small and two large axes, an RSS is used. Like in [LGLM00], an axis is considered as “larger” than an other one if it is more than twice as long.

An SSV is fully defined by the corner points of the core shape C and the radius r of the sphere, so it can be stored with only very low memory cost.

In [ZKM07], when building the BVH, the points are split at each node such that the summed density of the resulting SSVs is maximized and splitting is terminated if the density exceeds a given threshold. This did not work so well in my reimplementation, so the policy was changed a bit, as is described in Section 2.2.4.

Using BVH for accelerated DISP computation Given the BVH of the convex hull of a rigid model¹ and two configurations q_1 and q_2 , the DISP distance for the model between the two configurations is computed by the following algorithm:

1. Traverse the tree in *depth first* order until the first leaf is reached. For each point stored at this leaf, compute its displacement between q_1 and q_2 and determine the maximum displacement d_{\max}

$$d_{\max} = \max_{p \in L} (\|p(q_1) - p(q_2)\|) \quad (2.14)$$

where L denotes the leaf and $p(q_i)$ the point p in configuration q_i .

¹for articulated models see below

2. Continue traversing the tree, whereupon at each...

- ▶ ... **inner node** the maximal possible displacement d_{ssv} within the SSV of this node is computed. The SSV is given by its core shape corner points C and its radius r .

$$d_{\text{ssv}} = \max_{p \in C} (\|p(\mathbf{q}_1) - p(\mathbf{q}_2)\| + 2r) \quad (2.15)$$

This step is very fast, as only 1, 2 or 4 points have to be computed, depending on the core shape C . If $d_{\text{ssv}} \leq d_{\text{max}}$, the whole subtree rooted at the node can be culled away, as no point within this subtree will be able to outrange the current d_{max} .

- ▶ ... **leaf node** the maximal displacement of the points in the leaf is computed like in Equation 2.14 and d_{max} is updated accordingly if one of the points has a greater displacement.

When the tree is fully processed, d_{max} holds the DISP distance between the two configurations.

With this technique, lots of unnecessary computations can be avoided, increasing the speed of the DISP computation. One has to be careful when building the BVH, though, as an ill-shaped BVH can also increase the computation time due to the additional overhead of computing d_{ssv} at every node. This happened with the first BVH that was used, see Section 4.2 for details.

Application on articulated models

When working with articulated models like robot arms, the C-DIST algorithm is applied separately on every link of the model and the maximum of these link displacements is returned as the DISP distance of the overall model.

Algorithm 2.2: Applying C-DIST on an articulated model

Input: Articulated model A , configurations $\mathbf{q}_1, \mathbf{q}_2$

Output: $\text{DISP}_A(\mathbf{q}_1, \mathbf{q}_2)$

maxdisp \leftarrow 0;

foreach rigid link L in A **do**

 maxdisp \leftarrow max($\text{DISP}_L(\mathbf{q}_1, \mathbf{q}_2)$, maxdisp);

return maxdisp;

The algorithm above can be accelerated by extending the idea of the BVH one level up and considering bounding volumes of the links before computing their DISP distance. This way, whole links might be pruned away with the same mechanism that is used within the BVH. The necessary changes are shown in Algorithm 2.3. Ordering the links such that links further away from the base are computed first, increases the chance that links can be pruned, as the outermost links often experience the greatest displacement.

Algorithm 2.3: Accelerated version of Algorithm 2.2

Input: Articulated model A , configurations $\mathbf{q}_1, \mathbf{q}_2$

Output: $\text{DISP}_A(\mathbf{q}_1, \mathbf{q}_2)$

$\text{maxdisp} \leftarrow 0$;

foreach rigid link L in A **do**

Let $V = (C, r)$ be the root SSV of L ;

$\text{maxssv} \leftarrow \max_{\mathbf{p} \in C} (\|\mathbf{p}(\mathbf{q}_1) - \mathbf{p}(\mathbf{q}_2)\| + 2r)$; // Equation 2.15

if $\text{maxssv} > \text{maxdisp}$ **then**

$\text{maxdisp} \leftarrow \max(\text{DISP}_L(\mathbf{q}_1, \mathbf{q}_2), \text{maxdisp})$;

return maxdisp ;

2.2.4. Modification of the original C-DIST Algorithm

The implementation of the C-DIST algorithm that was actually used in this thesis diverges a bit from the one described by [ZKM07].

Splitting policy and termination criterion in BVH construction

To find the optimal split of a node, the authors of [ZKM07] define the density ρ of a node as the number of points in the node over the volume of the corresponding SSV. A partitioning plane is swept along the longest axis of the OBB of the points, to find the split that maximizes the following objective

$$\max(\rho_1 + \rho_2) \tag{2.16}$$

where ρ_1 and ρ_2 are the densities of the emerging subsets. Nodes are recursively split further, until the density exceeds a given threshold. This policy did, however, not behave very well for me for two reasons:

1. The density of the nodes did not always increase monotonically with increasing depth, making this an infeasible termination criterion.
2. Maximizing density often led to a split where only a minimal number of points (which are, however, extremely dense) is split of, resulting in an ill-shaped list-like tree (see Figure A.2 on page 70 in the appendix).

The first issue was tackled by not considering the density but only terminating if the number of points in a node is below a given threshold (this had to be implemented anyway as a fallback).

For the second issue, a first attempt was to replace the objective (2.16) with the following one:

$$\max \left(\frac{\min(\rho_1, \rho_2)}{\max(\rho_1, \rho_2)} \cdot (\rho_1 + \rho_2) \right) \tag{2.17}$$

The idea behind this is to favour splits where both sides have roughly the same density and thereby avoiding super-dense mini-subsets like described above. This

indeed led to much better balanced trees and better runtime. It turned out, however, that comparable results can be achieved by simply placing the partitioning plane at the centre of the longest axis. This last method was preferred in the end, as it is faster and easier to implement.

A comparison of the three different BVHs construction rules can be found in Section 4.2.

Modified link culling on articulated models

For applications on articulated models like robot arms, [ZKM07] proposed a simple optimization to avoid some unnecessary computations: For each link of the model, an *oriented bounding box* (OBB) is computed as a preprocessing step. When computing DISP distance for link L_i , first the DISP distance of the OBB is computed. If DISP for the OBB is less than DISP for all links L_1, \dots, L_{i-1} that have been computed so far, the link L_i can be culled away without computing its actual DISP distance.

There are two minor changes made to this technique:

1. Only compare to the maximum DISP of the so far computed links. This is also valid and simplifies the implementation.
2. Use SSVs instead of OBBs. This has several advantages:
 - SSV is computed anyway for the BVH, so there is no need for the additional effort of storing the OBBs.
 - DISP computation with SSVs is faster than with OBBs as there are less points to be computed (only 1–4 points, while the OBB always has 8 points).
 - Using SSVs makes the link culling more consistent with the usage of the BVH. In a sense, it just adds another level on top of the BVH of the individual links, that covers the whole articulated model.

2.3. Spectral Clustering

Spectral clustering denotes a family of graph based clustering algorithms which cluster data points into a given number of k clusters.

An important difference to other algorithms like k-means is, that spectral clustering requires only a similarity matrix of the data points as input, not the points themselves. This comes in very handy for the application in this thesis, as it means that DISP distances can be used for clustering, without having to modify the algorithm in any way.

This section gives a short overview on how spectral clustering works. For a more elaborate description and a discussion on *why* this actually works, the interested reader is referred to [vL07].

Spectral clustering is a graph based method. As input, it needs an undirected similarity graph of the data, where each node represents one data point and the edge weights denote the similarity of the connected points. As the actual value of the points is irrelevant for the clustering, the graph is fully defined by its weight adjacency matrix $\mathbf{W} \in \mathbb{R}_{\geq 0}^{n \times n}$. The element $w_{i,j}$ of the matrix contains a non-negative measure of the similarity of the i -th and j -th point. Thus \mathbf{W} can also be seen as a *similarity matrix* of the data. The higher such a weight, the more similar are the connected points, while zero means “completely different”. Since the graph is undirected, \mathbf{W} is symmetric, that is $w_{i,j} = w_{j,i}$.

In the following it is assumed that the graph is fully connected, but in principle other graphs are also possible, for example a k -nearest-neighbours graph or one where only edges are kept that have a weight higher than some value ε . Nodes that are not connected are then assumed to have zero similarity.

From the weight matrix \mathbf{W} we can easily derive the *degree matrix* \mathbf{D}

$$\mathbf{D} = \text{diag}(d_1, \dots, d_n), \quad \text{with } d_i = \sum_{j=0}^n w_{i,j} \quad (2.18)$$

With this, the *unnormalized graph Laplacian* is defined as

$$\mathbf{L} = \mathbf{D} - \mathbf{W} \quad (2.19)$$

Using \mathbf{L} , we have all we need for the simplest case of spectral clustering as described by Algorithm 2.4. There are also two other variants presented in [vL07] which use different kinds of normalized graph Laplacians and will not be discussed here.

Algorithm 2.4: Unnormalized Spectral clustering (taken from [vL07])

Input: Similarity matrix $\mathbf{W} \in \mathbb{R}_{\geq 0}^{n \times n}$, number k of clusters to construct.

1. Compute the unnormalized graph Laplacian \mathbf{L} .
2. Compute the first k eigenvectors $\mathbf{u}_1, \dots, \mathbf{u}_k$ of \mathbf{L} .
3. Let $\mathbf{U} \in \mathbb{R}^{n \times k}$ be the matrix containing the vectors $\mathbf{u}_1, \dots, \mathbf{u}_k$ as columns.
4. For $i = 1, \dots, n$, let $\mathbf{y}_i \in \mathbb{R}^k$ be the vector corresponding to the i -th row of \mathbf{U} .
5. Cluster the points $(\mathbf{y}_i)_{i=1, \dots, n}$ in \mathbb{R}^k with the k -means algorithm into clusters C_1, \dots, C_k .

Output: Clusters A_1, \dots, A_k with $A_i = \{j | \mathbf{y}_j \in C_i\}$.

The idea behind this algorithm is, to map the data into another space where similar points end up close together and thus can easily be clustered using a common algorithm like k -means (or any other clustering algorithm).

2.4. Robot Arm Pose Estimation through Pixel-Wise Part Classification

The contribution of this thesis is an extension of the former approach of robot arm pose estimation proposed by Bohg et al. [BRHS14], which is briefly described in this section. The approach they present is inspired by work of Shotton et al. [SGF⁺12] who did pixel-wise part classification of human bodies using the Kinect sensor. Their approach was adopted and, with modifications, applied on the problem of robot arm pose estimation.

For the pose estimation, each pixel of the depth image is pushed through a random decision forest, which then casts votes on joint positions relative to the camera. From this, joint angles are computed using inverse kinematics.

One of the key characteristics is, that both [SGF⁺12] and [BRHS14] use synthetically rendered depth images for the forest training, rather than real ones. By doing so, they get large, automatically annotated training sets with only little effort.

This section is structured as follows: First, the way of rendering synthetic training images is described. Then the depth features are presented, that are extracted from these images. After that, the training of the random forest is described and finally how the forest is used to get pose predictions from new, unknown images.

2.4.1. Generating Training Data

Based on [SGF⁺12], the depth images used for training the random forest were rendered synthetically, rather than using images from a real sensor. This way, huge amounts of training samples with perfect ground truth information can be generated, without the tedious and time-consuming task of labelling the images manually. The synthetic generation of depth images is easy, compared to colour images, as colour and textures of the objects in the scene can be omitted and good CAD models of the robot arm parts were already available.

Using the model of the robot, images with varying arm poses were rendered with a ray-tracer implemented in the CGAL library [cga]. To get more realistic images, the sensor model proposed in [GKUP11] was used and some noise was added to the images. The background varies in each image by adding some furniture (tables and chairs) at randomly chosen positions. Further the position of the head-mounted camera is varied by using a small set of different neck poses. Figure 2.5 shows some of the depth images that were rendered this way.

2.4.2. Depth Image Features

The depth image features used for the random forest training are a simplified version of the ones used in [SGF⁺12]. The value of feature φ_i of a pixel \mathbf{x} in image I is computed by

$$\varphi_i(I, \mathbf{x}) = I(\mathbf{x} + \delta_u) - I(\mathbf{x} + \delta_v) \quad (2.20)$$

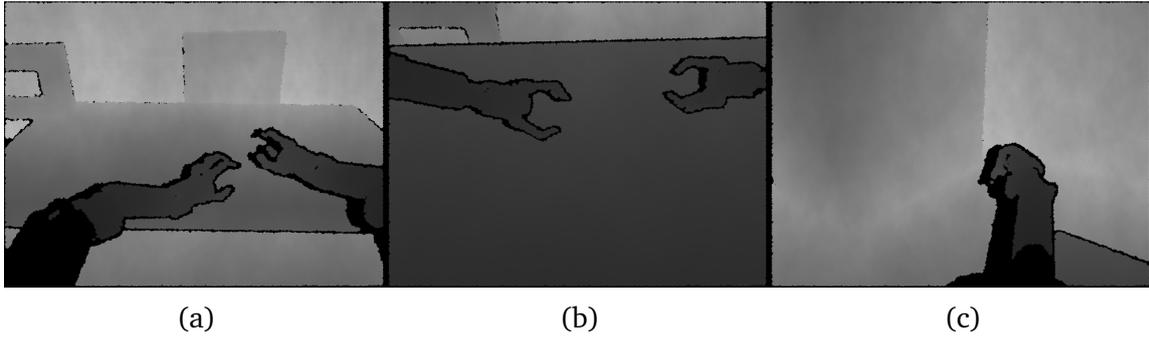


Figure 2.5.: Synthetically rendered depth images that were used to train the random forest.

where $I(x)$ is the depth value of pixel x . δ_u, δ_v are pixel offsets which are different for each feature. Invalid depth measurements and depth values of pixels outside of the image are replaced by a large positive number d_{\max} . In the actual implementation d_{\max} is set to 5 metres.

For the classification, a set of n_f such features is generated by drawing the offsets δ_u and δ_v from a uniform distribution over a quadratic window. This is done only once and the resulting set of features is then used for all training and test data points. In the implementation, n_f is set to 500. The features are illustrated in Figure 2.6.

2.4.3. Training

From each of the rendered depth images, a fixed number of foreground pixels (i.e. pixels showing parts of the robot) as well as a fixed number of background pixels are sampled randomly. For each of these pixels, the features generated in the previous step are computed. As ground truth information a class label, corresponding to the joint the pixel lies on (using an additional “background” class for pixels which are not part of any joint) and the 3D positions of all joints are given. Thus the actual training set is build which is then used for training a random forest.

The training objective that is maximized at the tree nodes is the information gain with respect to the entropy of the class labels. The resulting forest is a classifier that assigns a joint label to every pixel.

Additionally, when reaching a leaf node, all non-background pixels that end in this leaf cast votes for 3D joint position offsets relative to the pixel. A pixel can thereby only vote for the joint it belongs to and for direct neighbours of this joint. In each leaf, the resulting offset votes are clustered independently per joint. The clusters are weighted with their cardinality and for each joint only the position of the cluster with highest weight as well as the weight itself are kept.

The output of a leaf node at test time is then:

1. A class label, corresponding to either a specific joint of the arm or the background.

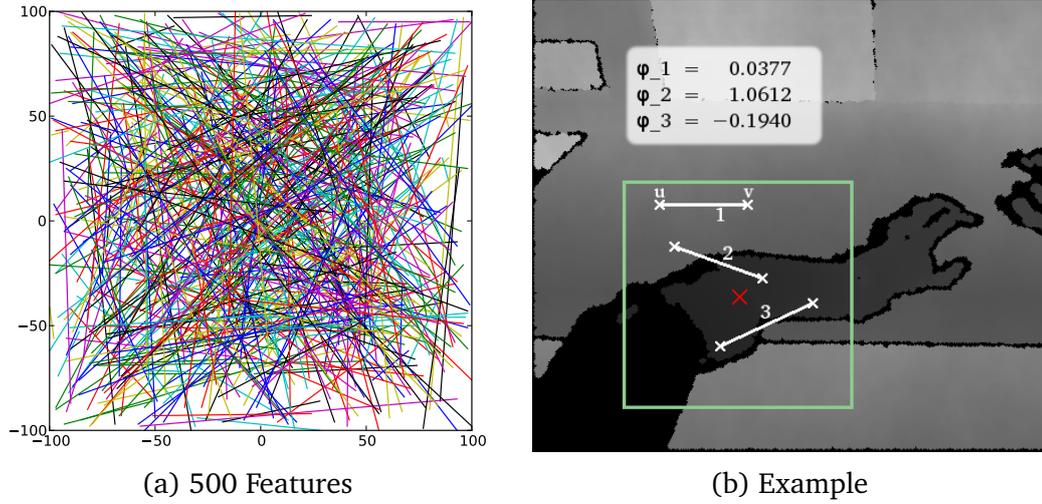


Figure 2.6.: (a) shows a set of 500 features with offsets randomly sampled in a window of 200×200 pixels. Each feature is visualized by a line connecting the two offset points. Different colours are used to make them better distinguishable. (b) shows three exemplary features evaluated for the pixel marked by a red cross. The green box shows the size of the window within which the features were generated.

2. The weighted 3D position offsets for the joints.

2.4.4. Prediction

For prediction, each pixel of the image is pushed through the forest and classified to either one of the links or the background. For pixels that are classified as belonging to the robot, joint position offset votes of the corresponding leaf nodes, together with their weights are cast. Doing so, one gets a bunch of 3D position votes relative to the camera for each joint. To get a single position estimate \hat{p} , for each joint the votes are clustered taking their weights into account. From the resulting clusters, again only the one with highest cardinality is used, storing the cardinality as a confidence measure.

Finally, to get the joint configuration $\hat{\theta}$ that matches best with \hat{p} , the following objective function is minimized incrementally

$$\min_{\theta} \|\hat{p} - p(\theta)\|^2 \quad (2.21)$$

This is basically solving the inverse kinematics problem by aligning a virtual arm with the predicted 3D joint positions. If available, internal joint encoders of the robot can be used to get a good initial estimate $\theta^{(0)}$.

3. Methodology

In this chapter, the methods developed for this thesis are presented. The goal is to improve the method described in Section 2.4 by getting rid of the intermediate steps of first estimating and clustering 3D joint positions and then doing inverse kinematics to get the actual joint configurations. This is achieved by replacing the classification forest with a regression forest, that directly estimates angular joint configurations. Further, the forest training is improved by implementing two new node split functions that optimize objective functions based on the model-dependent DISP metric described in Section 2.2.

The old and the new method are compared in Figure 3.1 (training) and Figure 3.2 (prediction). At training time, the new method simplifies the old one by removing the need of clustering joint position offsets for every joint at every leaf. Only if the spectral clustering (SC) based split criterion (one of the two new split functions, which are described later in this chapter) is used for the random forest training, additional operations on the leafs are necessary. At test time, the new approach drops the inverse kinematics step and also has less work to do to combine the prediction outputs of the individual pixels.

For training the regression forest, the same synthetic images like in Section 2.4 are used. To build the training set, from each image a fixed number of pixels from the foreground (showing the robot) and another fixed number of pixels from the background are randomly sampled. For each of these pixel samples, depth features are computed. From the available ground truth information, only the joint angles are needed, which are used as target value for the regression.

The resulting training set is given as $S = \{(\mathbf{x}_1, \mathbf{y}_1), \dots, (\mathbf{x}_n, \mathbf{y}_n)\}$ where $\mathbf{x}_i = (\varphi_1, \dots, \varphi_{n_f})$ is the feature vector of the i -th data point and $\mathbf{y}_i = (\theta_1, \dots, \theta_d)$ the corresponding ground truth joint configuration with θ_j the angular position of the j -th joint. Once the forest is trained, it expects a feature vector \mathbf{x} as input and returns a pose estimation $\hat{\mathbf{y}}$ with the estimated angle of each joint.

For the experiments, the random forest implementation of scikit-learn [PVG⁺11] was used, a machine learning library for Python, that implements many popular learning algorithms. For regression forests, only one node split criterion, using the mean squared error (MSE) is originally implemented in scikit-learn. This criterion, as well as the two specialized criteria that use the DISP distance, are described in Section 3.1. In Section 3.2 the extension of the random forest to provide prediction confidence values is described and in Section 3.3 it is shown how the pose estimations of the single pixels of an image are combined into the final result.

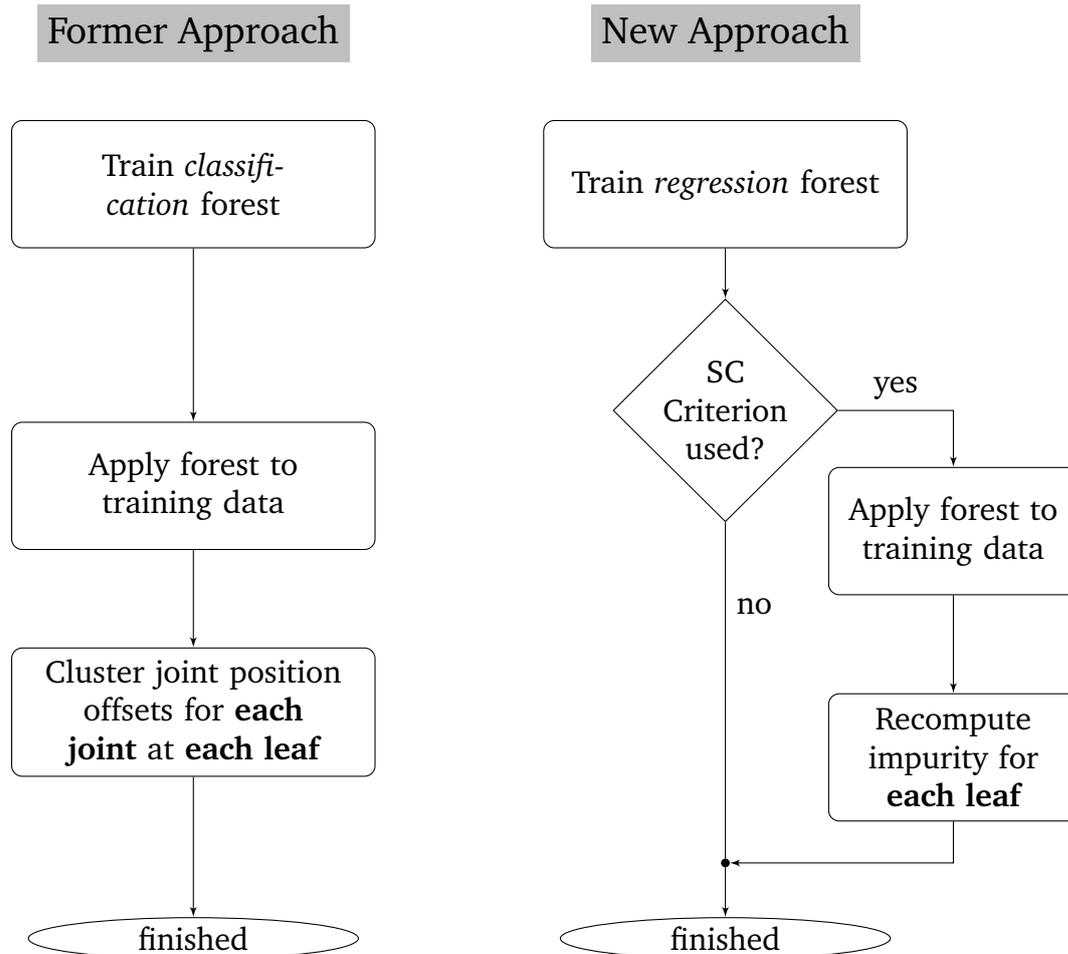


Figure 3.1.: Comparison of the training process of the old and the new method.

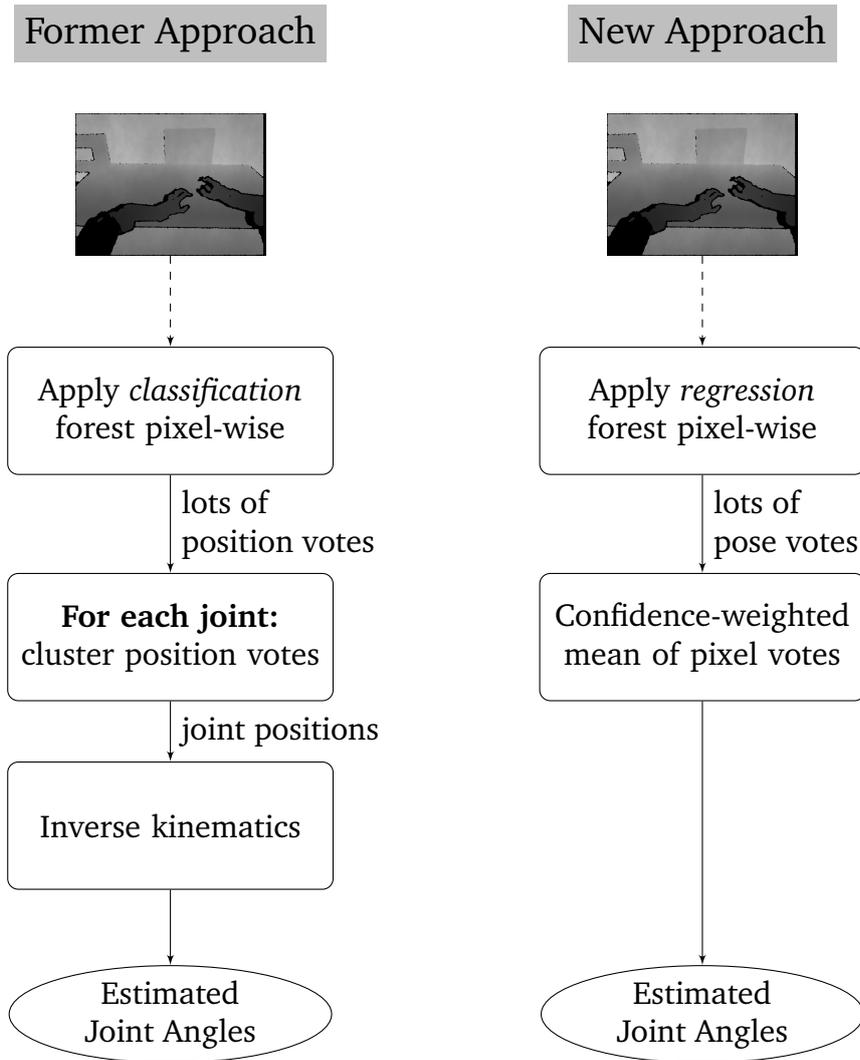


Figure 3.2.: Comparison of the prediction process of the old and the new method.

3.1. Random Forest Split Criteria

For random regression forests, only one split criterion is originally implemented in scikit-learn, using the mean squared error (MSE) in the target space of the regression. In our case, the target values are the joint angles in the configuration space. As we do not want to operate on the joint angles but use the DISP distance for the reason stated in Section 2.2.1, two custom split functions were implemented, one that minimizes the pairwise DISP distance between points and another one that performs a spectral clustering based on DISP distances. These custom criteria, as well as the MSE criterion already implemented in scikit-learn, are presented in this section.

All three split criteria basically maximize the following objective function

$$I(S, S_l, S_r) = H(S) - \sum_{i=l,r} \frac{|S_i|}{|S|} H(S_i) \quad (3.1)$$

that was already defined above in Equation 2.6. If $H(S)$ is defined as the entropy of S , this equation computes the information gain of the split. In the following, H is not restricted to be the entropy, but it is a more general *impurity* term, that is higher, the more “impure” the samples in S are. The exact definition of this impurity, depends on the split criterion. Actually, the impurity is the only thing that differs between the different split criteria, which are described in the following.

As a reminder, $S = \{(\mathbf{x}_1, \mathbf{y}_1), \dots, (\mathbf{x}_n, \mathbf{y}_n)\}$ denotes a set of training samples with feature vectors \mathbf{x}_i and target values \mathbf{y}_i (note that here \mathbf{y}_i is multi-dimensional, as it contains an angle for each joint). To simplify notations, for a given S we define S^x and S^y as follows:

$$S^x = \{\mathbf{x} | (\mathbf{x}, \mathbf{y}) \in S\} \quad (3.2)$$

$$S^y = \{\mathbf{y} | (\mathbf{x}, \mathbf{y}) \in S\} \quad (3.3)$$

3.1.1. Mean Squared Error (MSE)

The MSE split criterion is the one being implemented in scikit-learn. It has not been modified for this thesis and is just mentioned here for completeness, as it is used in the evaluation for comparison with the other criteria.

The impurity is simply defined as the average variance of the dimensions of the target value \mathbf{y} :

$$H(S) = \frac{1}{d} \sum_{i=1, \dots, d} \left(\frac{1}{n} \sum_{\mathbf{y} \in S^y} y_i^2 - \bar{y}_i^2 \right) \quad (3.4)$$

where d is the dimensionality of \mathbf{y} , $n = |S|$ the cardinality of the data and $\bar{\mathbf{y}} = \frac{1}{n} \sum_{\mathbf{y} \in S^y} \mathbf{y}$ is the mean.

3.1.2. Mean Squared Pairwise DISP (MSPD)

The MSE criterion defined above works on the configuration space of the samples, that is, in the context of this thesis, on the joint angles of the arm configuration. Due to the discrepancy between angle error and displacement error (see Figure 2.3a), this may not always choose the best split in terms of minimizing the arm displacement. Therefore, the *mean squared pairwise DISP* (MSPD) criterion was implemented, that chooses the split that minimizes the mean of the squared DISP distance between every pair of two poses in the subsets.

Formally, this is done by defining the impurity term for this criterion as follows:

$$H(S) = \left(\frac{n^2 - n}{2} \right)^{-1} \sum_{\mathbf{y}_1 \in S^y} \sum_{\mathbf{y}_2 \in S^y} \text{DISP}(\mathbf{y}_1, \mathbf{y}_2)^2 \quad (3.5)$$

with $n = |S|$ the cardinality of the data. The term $\frac{n^2 - n}{2}$ equals the number of pairs in S and is used to get the average squared distance of all pairs.

Originally, the DISP distance was intended to be computed on-line during training, but it turned out that the computation is too slow for that, especially since the distance between the same two poses has to be computed a lot of times during training. Instead, the distances are computed only once in advance and stored in a lookup table, which is then used at training time.

As will be shown later in Section 4.5, this criterion performed best in terms of pose estimation, but despite of using a lookup table for DISP distances, it is unfortunately still extremely slow. The reason for this is, that it increases the complexity of the split test considerably due to the double-sum over S^y in Equation 3.5 (in the actual implementation, this increased the complexity of the function that finds the best threshold for a potential split feature from linear to cubic time).

3.1.3. Spectral Cluster based Hamming Distance Score

Since the MSPD achieves considerably better accuracy in the evaluation than MSE, but is far too slow for practical usage, another criterion was developed as an attempt to benefit from the better distance representation of DISP, while considerably accelerating the training compared to MSPD.

To find a split for a set of samples, first these samples are clustered into two clusters, using spectral clustering (see Section 2.3). Spectral clustering is convenient in this case, as it only takes a similarity matrix \mathbf{W} as input, which can easily be generated from the DISP distance matrix \mathbf{D} . \mathbf{D} can be constructed from the pre-computed DISP lookup table without additional cost. The conversion from distance to similarity is done using an RBF kernel [VTS04]:

$$w_{ij} = \exp \left(-\frac{d_{ij}^2}{2 \cdot \delta^2} \right) \quad (3.6)$$

where d_{ij} is the DISP distance between poses i and j . δ is a free parameter, that influences how fast similarity drops with increasing distance. See Section 4.4.1 for experimental results with different values of δ .

This way, the poses can be clustered according to their DISP distance, without having to modify the implementation of the clustering algorithm. All samples in one cluster are assigned to the “cluster label” 0 and the samples in the other cluster to 1. Note that there is no defined order of the clusters and thus the choice, which of the clusters gets the 0 and which the 1 is arbitrary and interchangeable. This has to be kept in mind when using these labels.

The clustering into two clusters can be seen as a split of the data. More precisely, it is the optimal split with respect to \mathbf{W} . All split candidates are tested against the clustering and scored higher, the more they match. This is accomplished by using a measure that is based on the Hamming distance.

Definition (Hamming Distance). The Hamming distance [Ham50] is a measure of the difference between two character strings of equal length. It is defined as the number of positions, where the two strings do not match:

$$\text{hamming_dist}(p, q) = \sum_{i=0}^n [p_i \neq q_i] \quad (3.7)$$

$$\text{where } |p| = |q| = n \text{ and } [cond] = \begin{cases} 1 & \text{if } cond \text{ is true} \\ 0 & \text{else} \end{cases}.$$

Note that by definition, the Hamming distance is bounded in the range 0 to n .

Assuming that the cluster that goes to the left split is labelled with 0 and the one that goes to the right is labelled with 1, the impurity of the subset S_l is set to the number of ones in S_l and the impurity of the subset S_r to the number of zeros in it:

$$\begin{aligned} H(S_l) &= \text{num_ones}(S_l) \\ H(S_r) &= \text{num_zeros}(S_r) \end{aligned} \quad (3.8)$$

The function `num_ones` counts the number of samples in the given set that are assigned to cluster 1 and `num_zeros` does the same for cluster 0. Note that with this definition $H(S_l) + H(S_r)$ equals the Hamming distance of the split to the clustering.

Since it is arbitrary, which cluster is labelled with 0 and which with 1 and further we do not know in advance, which of the clusters should go left and which right, we have to check for both cases and use the one that has a smaller Hamming distance. Algorithm 3.1 shows how this is done. First, one variant is computed and if more than half of the assignments are wrong, labels are swapped to the other variant which yields the smaller Hamming distance in this case.

Why not just use the clustering as split? This question may come to mind here. If the clustering is the best possible split, why not use it directly instead of doing all

Algorithm 3.1: Compute Entropy of Spectral Cluster based Hamming Score

```

 $H(S_l) \leftarrow \text{num\_ones}(S_l)$ 
 $H(S_r) \leftarrow \text{num\_zeros}(S_r)$ 

if  $H(S_l) + H(S_r) > \frac{n}{2}$  then swap cluster labels
  |  $H(S_l) \leftarrow \text{num\_zeros}(S_l)$ 
  |  $H(S_r) \leftarrow \text{num\_ones}(S_r)$ 

```

this work of generating random splits and comparing them to the clustering? The answer is, that the random forest model requires a split criterion, that is only based on the features of the data. As the clustering does not look at the features but at the target values (which are known during training but not later at test time), we do not know how to reproduce this exact split using features—it might even be impossible. Therefore, the clustering can only serve as a “best case example”, which we try to reproduce as good as possible with a split based on the features, but which cannot be used directly.

3.2. Confidence-weighted Predictions

The current regression forest implementation of the scikit-learn library does not support probabilistic predictions. Each leaf simply stores the mean of all training samples, that ended in the leaf. This mean value is then used as the only prediction output of the leaf at prediction time. While this method is easy and fast, it lacks useful information about how “certain” the prediction of a tree is. In consequence, a leaf node with very high variance in its training samples has the same weight than a leaf with zero variance.

Fortunately, extending the existing implementation with confidence-weighted predictions is straightforward. To get confidence values for each leaf, the impurity, which is computed for each node at training time anyway, is used as a measure of uncertainty. An exception is made for the spectral cluster criterion, where the impurity used for training is not suited as an uncertainty measure. Therefore, for this criterion, the leaf uncertainty is computed in an additional step after training, using the impurity function of the MSPD criterion (Equation 3.5). For all three criteria, an RBF-Kernel like in Equation 3.6 is used to convert this uncertainties to confidences that are limited to the interval $[0, 1]$.

3.2.1. Using the Confidence for Prediction

Given a forest with T trees. At prediction time, the t -th tree now returns a prediction value \hat{y}_t just like before and in addition to that a scalar confidence value $c_t \in [0, 1]$.

To combine the single tree predictions into the final forest output, the confidence-

weighted mean is computed:

$$\hat{\mathbf{y}} = \frac{1}{\sum_{t=1,\dots,T} c_t} \sum_{t=1,\dots,T} c_t \cdot \hat{\mathbf{y}}_t \quad (3.9)$$

As a confidence measure of the overall forest prediction, the average confidence of all trees is computed:

$$c = \frac{1}{T} \sum_{t=1,\dots,T} c_t \quad (3.10)$$

3.2.2. Joint-wise Confidence

In the procedure above, only one scalar confidence value is used for the whole pose estimation. Since a specific pixel might provide a very good estimation for some of the joints, but not for all, the method was further extended to work with joint-wise confidences.

Since DISP cannot be used to measure different uncertainties for the single joints, joint-wise variance on the angles was used for all three split criteria. These values are not used for training and therefore have to be additionally computed once the forest is build.

For the prediction, nothing changes, except that c_t becomes a vector of same length as $\hat{\mathbf{y}}_t$ and all operations in Equations 3.9 and 3.10 are done *element-wise*, i.e. separately for each joint.

3.3. Select and Combine Votes of Individual Pixels

The random forest trained with the methods described above takes only the feature vector of a single pixel as input. Given a whole depth image, the forest is applied to each pixel of the image independently, resulting in a separate estimated robot arm pose for each pixel. These independent pixel estimations have to be combined into one final prediction for the whole image.

A first attempt to do this, was to simply compute the mean over all pixel predictions. This led, however, to rather poor results, probably due to the many background pixels that cannot contribute useful information.

Two other methods were tested which are described in the following. First a histogram over the pixel predictions was used to find the most dominant pose among the pixel votes. This slightly improved the prediction in some cases, but a significant breakthrough could not be achieved. The second attempt introduces uncertainties to the prediction (see Section 3.2), and uses only the most confident pixel predictions. This finally led to much better results as can be seen in Section 4.6.

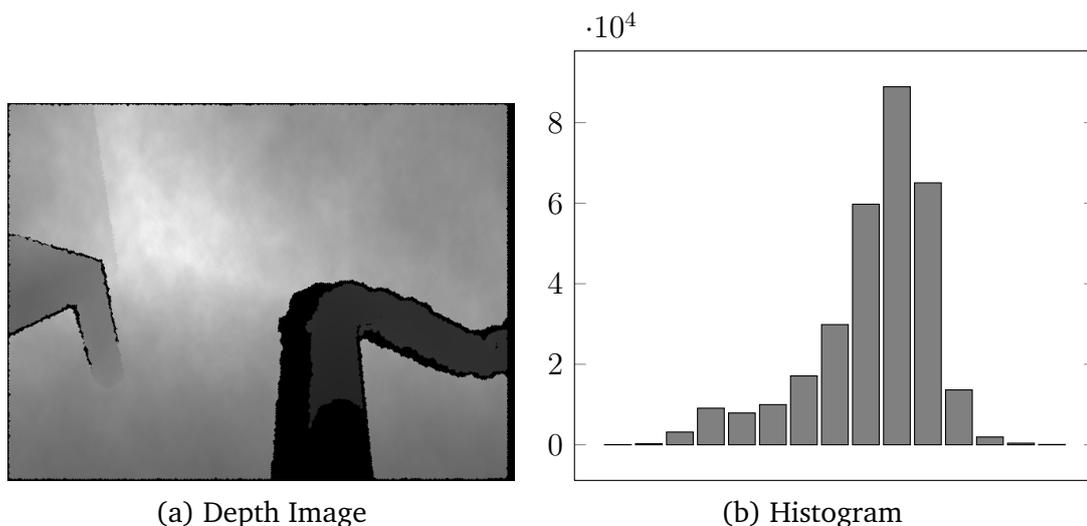


Figure 3.3.: (a) shows a depth image and (b) the histogram over all pixel votes of this image for a specific joint.

3.3.1. Histogram Prediction

Since the votes of the individual pixels are not necessarily Gaussian distributed, simply computing the mean of all votes can be a bad choice. To better catch the actual distribution of the pixel votes, for each joint a histogram over the votes of all pixels is generated. An example is visualized in Figure 3.3. The final pose estimation is then done by selecting for each joint the value that lies in the centre of the largest bin.

3.3.2. Confidence-weighted Mean

Most pixels in the image are part of the background far away from the arm and should provide no or only little useful information about the position of the robot arm (features are computed in a window around the pixel, so background pixels close the arm might still be helpful to some degree as the arm lies within this window). Such pixels are prone to high errors and do thus impair the overall prediction. Ideally, the votes from such pixels should also have a high uncertainty, when using the confidence-based prediction described in Section 3.2.

To exploit this, a confidence-based prediction is done for each pixel $p = 1, \dots, n$, resulting in a prediction \hat{y}_p and an associated confidence value $c_p \in [0, 1]$ for every pixel. Predictions with low confidences are assumed to be bad, so they are completely discarded if the confidence is below a dynamic threshold t :

$$t = \min_p(c_p) + (\max_p(c_p) - \min_p(c_p)) \cdot t_r \quad (3.11)$$

where $t_r \in]0, 1[$ is a free parameter that defines the ratio of the confidence range that is used. This dynamic threshold has some nice properties: It guarantees, that

always at least one pixel is above the threshold and it automatically adapts to images with different average confidence.

To discard a prediction, its confidence is set to zero, so it will effectively be non-existing in the following computations. Doing so gives us the new confidence

$$\tilde{c}_p = \begin{cases} 0 & \text{if } c_p < t \\ c_p & \text{else} \end{cases} \quad (3.12)$$

Finally a confidence-weighted mean over the remaining pixels is computed, to get the final prediction $\hat{\mathbf{y}}$ for the whole image:

$$\hat{\mathbf{y}} = \frac{1}{\sum_{p=0}^n \tilde{c}_p} \cdot \sum_{p=0}^n \tilde{c}_p \cdot \hat{\mathbf{y}}_p \quad (3.13)$$

The parameter t_r in Equation 3.11 can be seen as a “relative threshold”. If it is for example set to $t_r = 0.9$, the lower 90% of the confidence range is discarded and only the upper 10% are used to compute $\hat{\mathbf{y}}$. Experiments in Section 4.4.3 show, that the accuracy of the prediction increases more or less monotonically if t_r is increased.

Joint-wise Confidence

Since a pixel p somewhere on the elbow joint may have good information about this joint, but cannot know anything about the position of the finger joints, the same approach has been tested with joint-wise confidences $c_p \in [0, 1]^n$, where n is the number of joints. This allows a pixel to vote only for joints where the confidence is high, without impairing the estimation of joints, where it has low confidence and thus probably a high error.

The equations look almost the same. Let y_p^j be the prediction of pixel p for joint j and c_p^j the associated confidence. For every joint, a separate threshold t^j is computed:

$$t^j = \min_p(c_p^j) + (\max_p(c_p^j) - \min_p(c_p^j)) \cdot t_r \quad (3.14)$$

Note that the parameter t_r is the same for all joints. The confidence is set to zero only for joints with low confidence, not for the whole pixel:

$$\tilde{c}_p^j = \begin{cases} 0 & \text{if } c_p^j < t^j \\ c_p^j & \text{else} \end{cases} \quad (3.15)$$

It is therefore possible, that a pixel votes only for some of the joints. Due to the different confidences, the weighted mean has to be computed for each joint separately as well:

$$\hat{\mathbf{y}}^j = \frac{1}{\sum_{p=0}^n \tilde{c}_p^j} \cdot \sum_{p=0}^n \tilde{c}_p^j \cdot \hat{\mathbf{y}}_p^j \quad (3.16)$$

4. Results

Different experiments were done to evaluate the performance of the methods of Chapter 3. The results of those experiments are presented in this chapter and are organized as follows:

First some information about the experiment set-up are given and which datasets and evaluation methods were used. In Section 4.2 the performance of the implementation of the C-DIST algorithm is benchmarked and the structure of the constructed bounding volume hierarchies (BVHs) is analysed. In Section 4.3 randomly generated arm poses are clustered using the DISP distance, to get a first insight on how well this metric is suited for the following tasks.

In Section 4.4 experiments were done to find good values for parameters of the random forest estimator. In Sections 4.5 and 4.6 finally the trained random forests for arm pose estimation are evaluated, comparing the different split criteria and showing the accuracy of the prediction. In the end, in Section 4.7, the estimator is applied on data from a real robot, showing qualitative results for a few arm poses.

4.1. Experiment Set-up

4.1.1. Hardware

All experiments were done on a machine running Ubuntu 12.04 with 64 GB RAM and two Intel Xeon CPU E5-2687W v2 with a total of 32 logical cores, running at 3.4 GHz.

4.1.2. The ARM Robot

For the evaluation, the same ARM robot¹ like in [BRHS14] was used (see Figure 4.1a). It consists of two Barret WAM arms with 7 DoF and Barret hands with 4 DoF as well as a head that is mounted on a pan-tilt unit. At the head, an Asus Xtion² RGB-D camera is attached.

To move the arm, rotation of the motors is transferred to the joints via steel cables. There are position encoders only at the motors, not in the joints directly. Joint positions can be inferred from the motor encoder readings, but due to variable stretch of the cables, this gives quite inaccurate results. In [RKP⁺14] a position error of up to 4 cm is reported for the end-effector. This error is qualitatively visualized in

¹<http://thearmrobot.com>

²<https://www.asus.com/Multimedia/Xtion>

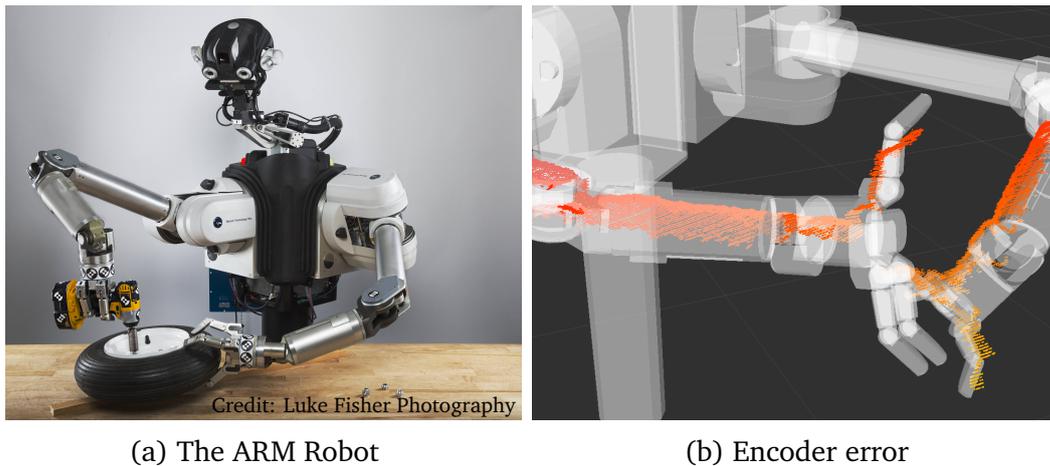


Figure 4.1.: The ARM Robot. (b) shows joint encoder readings versus depth image. The pose of the robot model is based on the joint encoder readings. In red/orange the point cloud recorded by the head-mounted depth camera is overlaid. It can clearly be seen that there is a significant discrepancy between these measurements.

Figure 4.1 where the robot model, showing the pose reported by the encoders, is overlaid with the depth measurements of the Xtion.

4.1.3. Data Sets used for Evaluation

A couple of different datasets were used in the several experiments that are reported in this chapter. They all consist of a set of synthetically rendered depth images of size 640×480 pixels, that are generated as described in Section 2.4.1. Each image shows a different arm pose. As ground truth, only the angular joint configuration of each pose is used.

From each image, 2000 pixels are randomly sampled from the foreground (i.e. pixels that are showing parts of the robot), as well as 1000 pixels from the background. So from each image a total of 3000 data points is used. A set of 500 depth features are randomly generated once using a window of 200×200 pixels for the pixel offsets. These features are then computed for each of the sampled pixels.

For training, the images are randomly partitioned into a training and a test set. For k -fold cross validation, the training images are randomly partitioned into k sets of equal size. For each step in the cross validation, $k - 1$ of these subsets are combined and used as training set and the remaining subset is used for testing.

All these operations are done as a preprocessing step, that is the selected pixels, the features and the splits into training and test set are the same for all forests that are trained on the corresponding dataset. This is especially important for comparing the performance of the different split criteria as it eliminates the noise of different random selections in these steps.

The following datasets were used for the evaluations below:

set2000: The first 2000 images of the dataset that was used in [BRHS14]. Total number of data points: 6 000 000.

set200: Like “set2000” but using only the first 200 images. Total number of data points: 600 000.

small: A very small set, consisting only of the first 79 images of “set2000”. Total number of data points: 237 000.

fixcam200: Another set of 200 images, that was rendered with a fixed camera pose for all images. Total number of data points: 600 000.

fixcam1267 Like “fixcam200” but using all 1267 available images with fixed camera pose. Total number of data points: 3 801 000.

4.1.4. Evaluation methods

For the evaluation of the random forests, a set of different tests were done:

test samples: The forest is tested on the test set, i.e. pixels of the training data that were put aside and not used for the training. For the test, only foreground pixels, showing parts of the robot, are used as background pixels are expected to provide no good estimation in any case.

test images: For this test, not only the sampled pixels but the whole images of the test set were used to estimate the pose like described in 3.3. This is the way, the forest could later be used in a running system, so the performance in this test is actually more important than the “test on samples”.

test images jw: Same as above but using joint-wise confidences instead of pose-wise (i.e. pixels are able to vote only for a part of the joints).

To evaluate the resulting estimations of these tests, two different measures were used, to compute the error between prediction \mathbf{y}_{pred} and ground truth \mathbf{y}_{true} :

- The *mean squared error* (MSE), measuring the error in configuration space, that is on the joint angles

$$\text{MSE} = \frac{1}{n} \sum_{i=1}^n \|\mathbf{y}_{\text{true},i} - \mathbf{y}_{\text{pred},i}\|^2 \quad (4.1)$$

- The *mean squared DISP error* (MSDE), measuring the error in the robots workspace using the squared DISP distance.

$$\text{MSDE} = \frac{1}{n} \sum_{i=1}^n \text{DISP}(\mathbf{y}_{\text{true},i}, \mathbf{y}_{\text{pred},i})^2 \quad (4.2)$$

4.2. C-DIST Benchmark

This section presents the results of some benchmarks that were done with different versions of the C-DIST implementation, to see, how much the several optimizations improve the performance.

All benchmarks were done with two different rigid models and one articulated model. For the rigid models, the famous “Stanford Bunny” and the “Armadillo” model were used, both taken from the *Stanford 3D Scanning Repository* [sta]. The benchmark on the articulated model was done with one arm of the ARM robot, that is also used for the evaluation of the pose estimation.

Three different methods for the DISP computation are evaluated: First a brute force algorithm that simply computes the displacement for every single point of the model to find the maximum. This is easy to implement but also very slow for larger models. The second attempt only computes displacement of every point of the convex hull (CH) which can be build as a preprocessing step. Finally the complete C-DIST algorithm using bounding volume hierarchies (BVHs) is benchmarked. There are three variants of the full algorithm, concerning the BVH construction (see Section 2.2.4):

- *split by density*: Split nodes such, that the summed density of the children is maximal (this is the method described in [ZKM07]).
- *split by scaled density*: Variant of the first, where the density is scaled by the ratio between the densities to avoid huge differences.
- *split by centre*: Simply split at the centre of the longest dimension.

Note that for the articulated robot arm model, link-pruning using SSVs is done. This in fact just extends the BVH one level upwards, adding the roots of all link-BVHs as children of a new root. Further the links where ordered descending by their distance to the base, as proposed by [ZKM07], to increase the pruning efficiency.

The results of all benchmarks as well as the number of vertices of each model are listed in Table 4.1 and plotted in Figure 4.3. It can clearly be seen, that the BVH implementation that splits by density (which, to my best knowledge, matches the algorithm described in the paper), did not work well here. While the authors of C-DIST report an acceleration of about factor 10 on the bunny model, compared to the method using only the convex hull, here the computation got more than 4 times *slower*. This decrease in performance can be explained by the very ill-shaped structure of the BVH: At each node only a very small subset of 3 or 4 points is split of, which has a extremely high density. Thus the tree degenerates to a deep list and the cost of the additional computations on the bounding volumes outweighs the benefit from pruning points away.

One attempt to avoid such splits was to scale the summed density by the ratio of the less dense subset to the denser one (*split by scaled density*). This effectively favours splits where the density on both sides is about the same. Doing so indeed improved the performance, but it turned out, that the results are about the same, if

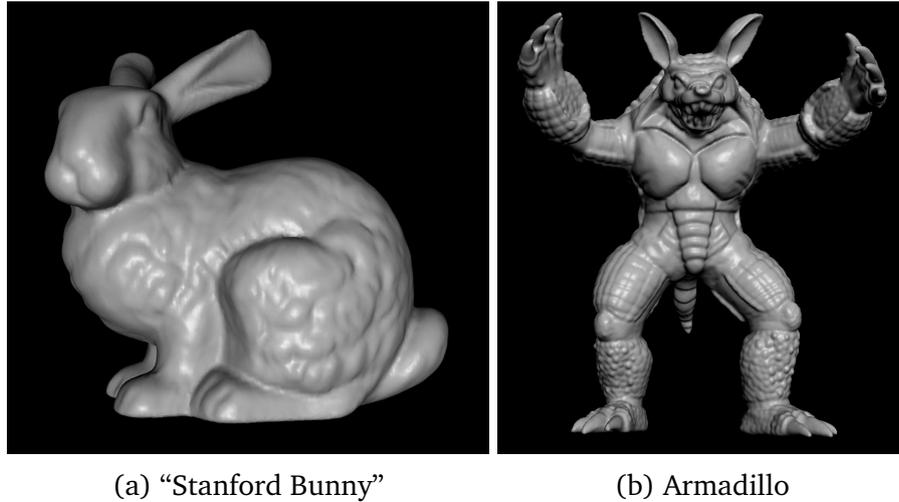


Figure 4.2.: The bunny and the armadillo model of the Stanford 3D Scanning Repository [sta].

the split is simply done at the centre of the axis, rather than optimizing densities. Finally this “centre split” was used as it is faster and much easier to implement. Exemplary BVHs for all three methods are visualized in Figures A.1 and A.2 for one of the links of the robot arm (due to their size, the figures were moved to the appendix, see pages 69 and 70).

The huge difference in performance of the BVH with density split compared to the results of [ZKM07] might be explained by some important detail missing in the paper or perhaps just some misunderstanding when reimplementing it. As the performance of the DISP computation was not of crucial importance any more, after the decision to precompute them and build a lookup table in advance, no further investigation of this issue was done, though.

4.3. DISP-based Clustering of Arm Poses

After implementing the C-DIST algorithm, two experiments were done, to see if the DISP metric is generally suited for clustering similar arm poses together. The results of these experiments are presented in this section. First randomly sampled arm poses were clustered using a modified k-means implementation and second a random decision forest with DISP-based split function was trained to learn the underlying distribution of the random samples.

4.3.1. K-Means Clustering with DISP Distance

To get a first impression of the suitability of DISP for clustering data points in a meaningful way, random arm pose samples were generated and clustered using DISP as distance metric.

Model	Bunny	Armadillo	Robot Arm
number of vertices	34,834	172,974	42,635
number of points on CH	1,497	1,932	4,410
brute force	0.507 ms	2.283 ms	0.561 ms
with convex hull	21.82 μ s	25.93 μ s	53.52 μ s
with BVH (split by density)	95.94 μ s	112.9 μ s	38.51 μ s
with BVH (split by scaled density)	21.16 μ s	17.77 μ s	9.353 μ s
with BVH (split at centre)	22.80 μ s	16.26 μ s	9.608 μ s

Table 4.1.: Average computation time of different versions of the C-DIST implementation. The times for the rigid bunny and armadillo models are averages over 1,000,000 computations with random transformations. For the articulated robot arm model, a set of 5,000 random samples was generated and DISP between every pair of samples was computed (exception: for the slow brute force computation, only 1,000 samples were used to reduce runtime of the benchmark process). A plot of the data is shown in Figure 4.3.

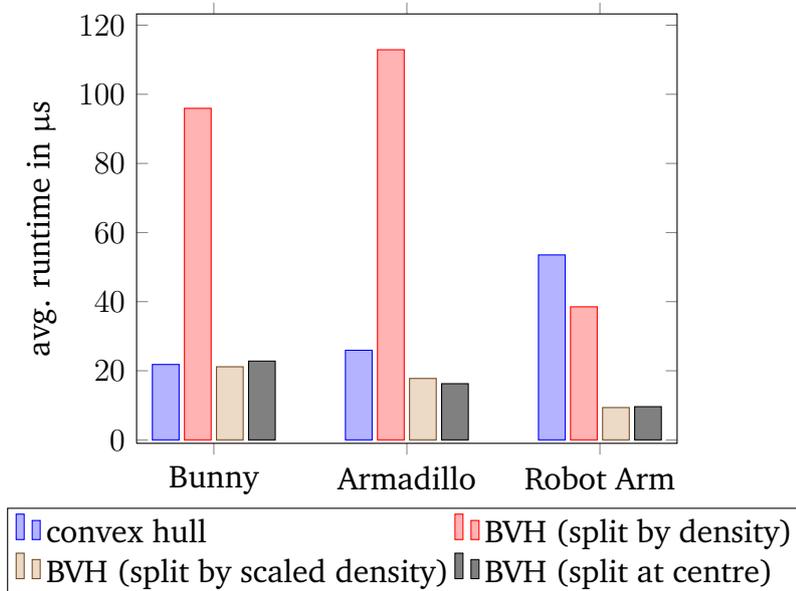


Figure 4.3.: Plot of the data in Table 4.1 (brute force is left out here).

For the clustering, k-means [Mac67] was used, a popular clustering algorithm, that clusters a set of data points into a fixed number of k clusters. It starts by choosing an initial centroid for each cluster and assigning each data point to the cluster with the nearest centroid. After that, new centroids are computed using the mean of the data points in each cluster and all points are reassigned to the nearest cluster, using the new centroids. This process continues iteratively until it converges (i.e. cluster assignments do not change any more).

For this experiment, the metric that is used to find the nearest centroid (typically Euclidean distance) was replaced by the DISP distance.

Arm pose samples s are given in configuration space, that is $s = (\theta_1, \dots, \theta_n)$, where θ_i is the angular position of the i -th joint. Random samples are generated by drawing for each joint a value from a uniform distribution over the range of the joint.

It turned out, that the mean on the joint angles does not always give a good cluster centroid in terms of arm displacement for arbitrarily distributed samples. One extreme example is shown in Figure 4.4a where the mean pose (visualized in blue) is completely off. In consequence, k-means did not converge or at least not within reasonable time. To solve this problem, the k-means implementation was further modified to choose the centroid c of a cluster C in a different manner: Instead of computing the mean of the samples, c is set to the sample in the cluster, that has the smallest summed squared DISP distance to all other samples in the cluster:

$$c = \arg \min_{s \in C} \sum_{s' \in C} \text{DISP}(s, s')^2 \quad (4.3)$$

With this modification the clustering works well as can be seen in Figure 4.4b where a set of random samples was generated by adding Gaussian noise to three initial samples.

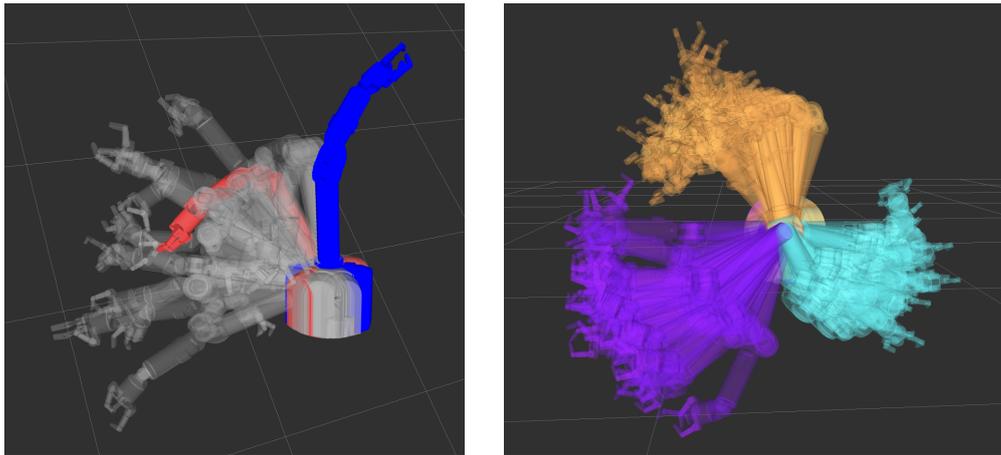
4.3.2. DISP-based Density Trees

After verifying that the DISP distance is able to cluster arm poses in general, the next step was to see, how well it is suited for split functions that are used to construct decision trees.

A random density forest (see Section 2.1.4) was trained to learn the distribution of a randomly sampled training set. This set was generated as follows:

1. Five initial samples are generated by drawing for each joint an angle from a uniform distribution over the range of the joint.
2. More samples are generated by adding Gaussian noise to the initial samples, with standard deviation proportional to the range of the particular joint.

This way, a total of 500 samples was generated for this experiment. The whole set is visualized in Figure 4.5.



- (a) K-Means centroids: The grey arms show one cluster of a set of uniform random arm pose samples. The blue arm shows the mean of this cluster, the red one the sample with the smallest summed squared DISP distance to all other samples in the cluster.
- (b) Result of clustering with the modified k-means. The pose samples were generated by adding Gaussian noise to three initial samples. The cluster assignment is visualized by the different colours.

Figure 4.4.: Cluster poses with k-means using DISP distance.

Using this set, density forests of size 1 (i.e. actually only single density trees) were trained, using different split scoring functions. The reason for using only a single tree, was to get a clearer insight on how well the single trees perform under the different split functions. The joint angles of the pose samples were used as feature values, that is at each node one joint is considered and compared to some threshold.

For the implementation, an existing C++ implementation for random forests was used with some modifications. At each node, a fixed number of splits is tested, that consist of randomly chosen features and thresholds (i.e. no exhaustive search over all possible thresholds is done). Among the tested splits, the one is chosen that maximizes a scoring function f_{score} .

The experiment was done with four different scoring functions, to get a better insight on how well the DISP score behaves:

Minimize DISP This is actually the same score that is used in the MSPD criterion described in Section 3.1.2. It minimizes the average DISP distance between samples within the subsets.

Maximize DISP Same like above but maximizing the distance instead of minimizing. This should lead to very bad splits and is done as a kind of sanity check. It is expected, that this method does *not* work well.

Information Gain Maximize the information gain of each split. This is the scoring function that has been defined above in Section 2.1.4.



Figure 4.5.: The training data used for this experiment. The better the density forest has learned this distribution, the closer should its output be to it.

Random Every split has the constant score 1. This leads to completely random splits and is again used as a sanity check. A good split score should perform better than this, while the *maximize DISP* split is expected to be considerably worse.

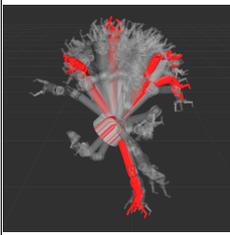
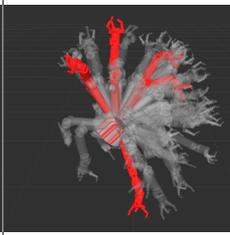
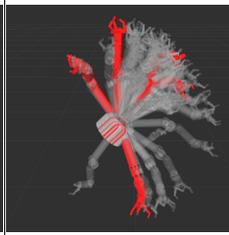
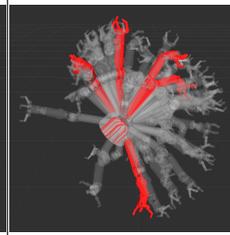
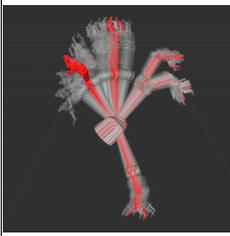
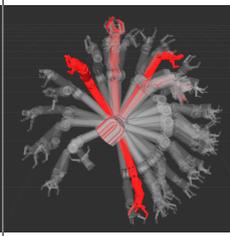
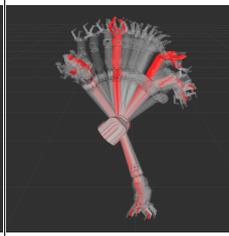
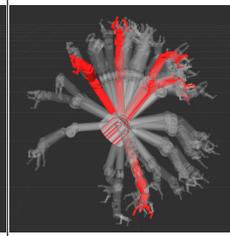
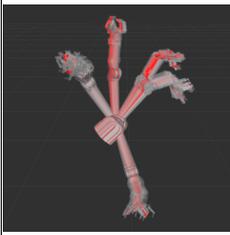
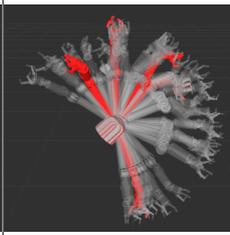
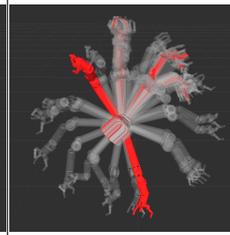
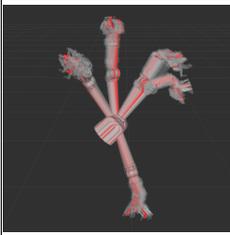
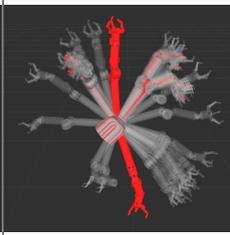
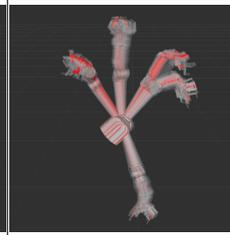
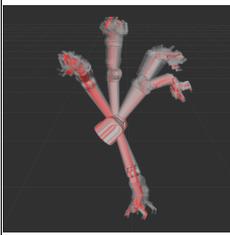
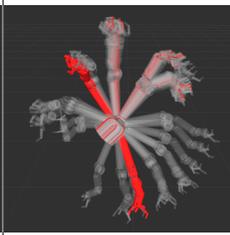
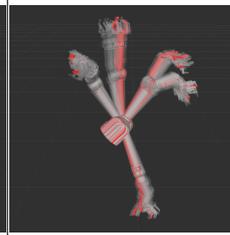
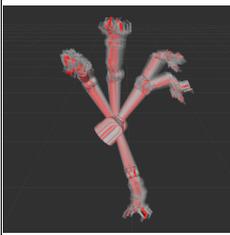
The trained trees were used to generate random samples like described in Section 2.1.4. The better the tree learned the distribution, the better should the distribution of the generated samples match with the one of the training data.

It turned out, that the tree depth has a strong impact on the behaviour of the forest. If trees are too deep, they overfit and learn the training distribution no matter how bad the split policy during training performs. This becomes obvious for the extreme case, where the depth is not limited and nodes are split, until there is only one sample per node. Since each leaf holds only one sample, the distribution within this leaf is pure (i.e. the standard deviation is zero). Sampling from such a tree would always produce samples that are present in the training set. The sampling would degenerate to drawing random samples from the original training data. To make sure not to be fooled by this effect, the experiment was repeated multiple times with different maximum tree depths, i.e. a tree node is not split any further, if the maximum depth is reached.

The result is visualized in Table 4.2. It can be seen, that the score functions *minimize DISP* and *information gain* are performing well already for very small trees. Even the splits of the *random*-score show quite good results when the maximum tree depth is increased enough. As expected, the *maximize DISP*-score behaves very bad. It also gets better, though, if the maximum tree depth is increased and finally becomes comparable to the other scores due to overfitting of deep trees.

4. Results

Table 4.2.: Comparison of different split scores and tree depths. *mtd* is short for “maximum tree depth”.

mtd	Minimize DISP	Maximize DISP	Information Gain	Random
2				
3				
4				
5				
10				
50				

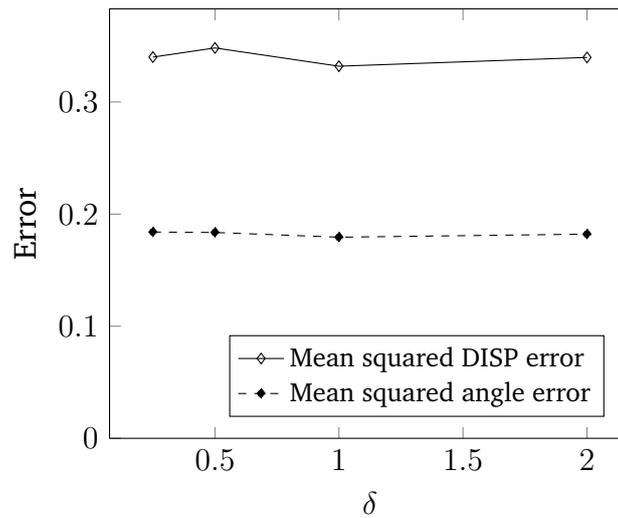


Figure 4.6.: Mean squared error in workspace (using DISP distance) and in configuration space (Euclidean distance on angles) for different values of δ .

4.4. Finding Good Parameters for Training and Estimation

A couple of parameters had to be set for the training of the random forest and the estimation of arm poses using this forest. Experiments were done, to find good values for these parameters, which are presented in this section.

4.4.1. RBF-Kernel for Spectral Clustering

In Section 3.1.3 the usage of an RBF-kernel is described to convert distances between samples to similarity values. These similarity values are used to compute the optimal split of the training samples in a node, using spectral clustering. As a reminder, the RBF-kernel is defined in Equation 3.6 as

$$w_{ij} = \exp\left(-\frac{d_{ij}^2}{2 \cdot \delta^2}\right) \quad (4.4)$$

The *bandwidth* δ of the kernel is a free parameter which influences how fast similarity drops off with increasing distance. A experiment was run to see if the choice of δ influences the performance of the forest. Different values for δ were cross validated on the “small” dataset (see Section 4.1.3). The error was measured using the “test images” method, described in Section 4.1.4.

Figure 4.6 shows a plot of the results. It can be seen, that the choice of δ does not have a significant impact on the regression result, so it can be chosen more or less arbitrarily. For all following experiments, $\delta = 1$ was used.

4.4.2. Finding Forest Parameters with Randomized Grid Search

For the actual training of the random forest, three parameters had to be set:

- Forest size T .
- Minimum number of samples in leaf nodes $n_{l,\min}$.
- Number of split features to test at each node n_{features} .

The forest size was set to 5 for all experiments like in [BRHS14], to get comparable results. The remaining two parameters were determined by doing a randomized grid search using the dataset “set200” and the MSE criterion for forest training. The resulting parameters, which were used for all following experiments, are:

- $n_{l,\min} = 36$
- $n_{\text{features}} = 300$

Note that n_{features} defines only the number of split features that is checked, not the total number of splits candidates. For each randomly selected feature, the full range of possible split thresholds is tested.

4.4.3. Pixel Confidence Threshold

In Section 3.3.2 the concept of confidence-weighted combination of the predictions of individual pixels in an image was described. A free parameter $t_r \in]0, 1[$ has been introduced there, to choose the confidence threshold t dynamically for each image (see Equation 3.11, or 3.14 for joint-wise thresholds).

To see how the choice of t_r influences the estimation accuracy, different values were tested and the results are plotted in Figure 4.7. It can be seen, that for all three split criteria, the error decreases more or less monotonically when t_r is increased. Further, the pose-wise confidence worked better than the joint-wise in all three cases.

4.5. Comparison of Split Criteria

In this section, the estimation accuracy of the three different split criteria described in Section 3.1 is empirically analysed. For each criterion, a 5-fold cross validation was done, that is in each fold four fifths of the images in the dataset are used for training and the resulting forest is tested on the remaining fifth. Due to the very long computation time of the MSPD criterion, only the relatively small dataset “set200” was used³ (see 4.1.3).

³MSPD training still took more than 3 days per fold

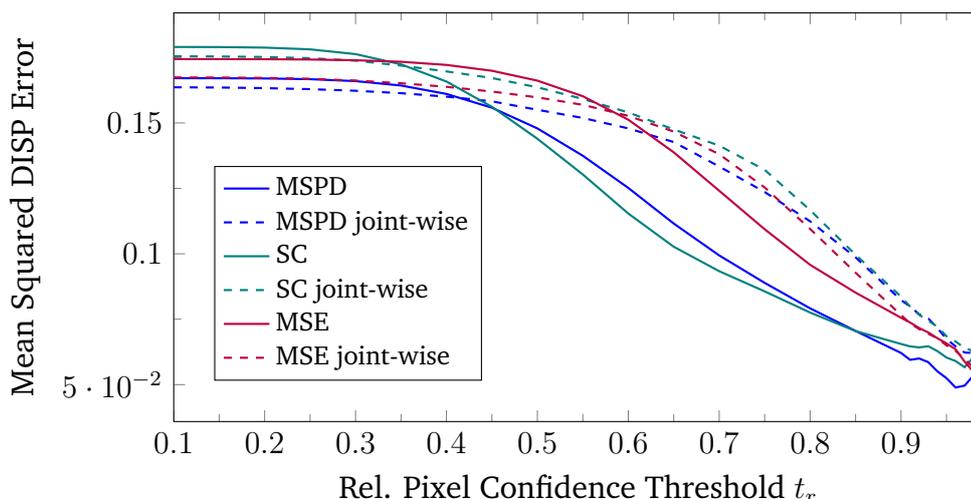


Figure 4.7.: Error for different relative pixel confidence thresholds t_r . Experiments were done on the “set200” dataset. Both pose-wise and joint-wise confidences were evaluated. Different forests were trained with the *mean squared pairwise DISP* (MSPD) split criterion, the *spectral clustering* (SC) based criterion and the *mean squared error* (MSE) criterion. The plot is done with step size 0.05 and a denser step size 0.01 in the more interesting interval of 0.9 to 0.99.

For each forest, the three tests “test samples”, “test images” and “test images jw” described above in Section 4.1.4 were performed whereat for the tests on images two different relative confidence thresholds $t_r = 0.95$ and $t_r = 0.99$ were used.

The average estimation error of the folds and the corresponding standard deviation are plotted in Figure 4.8. Two error metrics are used: The mean squared DISP error (MSDE), measuring the error in the workspace of the robot, and the mean squared error (MSE), measuring the error in the configuration space of the arm (see Section 4.1.4). Table 4.3 shows an analysis of the resulting forests in terms of depth, number of leaf nodes, etc.

The training set used above consists of images that were generated with varying poses of the head-mounted camera. This can lead to ambiguities as different arm poses may look similar in the image, when observed from different camera poses (this problem is discussed in more detail in Chapter 5). To verify, if this leads to errors in the prediction, the same evaluation was done on the “fixcam200” training set, where all images are rendered with the same camera pose. The result of this experiment is shown in Figure 4.9 and the analysis of the forests in Table 4.4.

Note that in the evaluation on the fixcam200 set, for the MSPD criterion, only the first four folds of the 5-fold cross validation could be used, as the training was even slower than on set200 and took more than 6 days per fold. Thereby the last fold did not finish in time, before this thesis was printed. For the other criteria, the full 5 folds are used, though.

4. Results

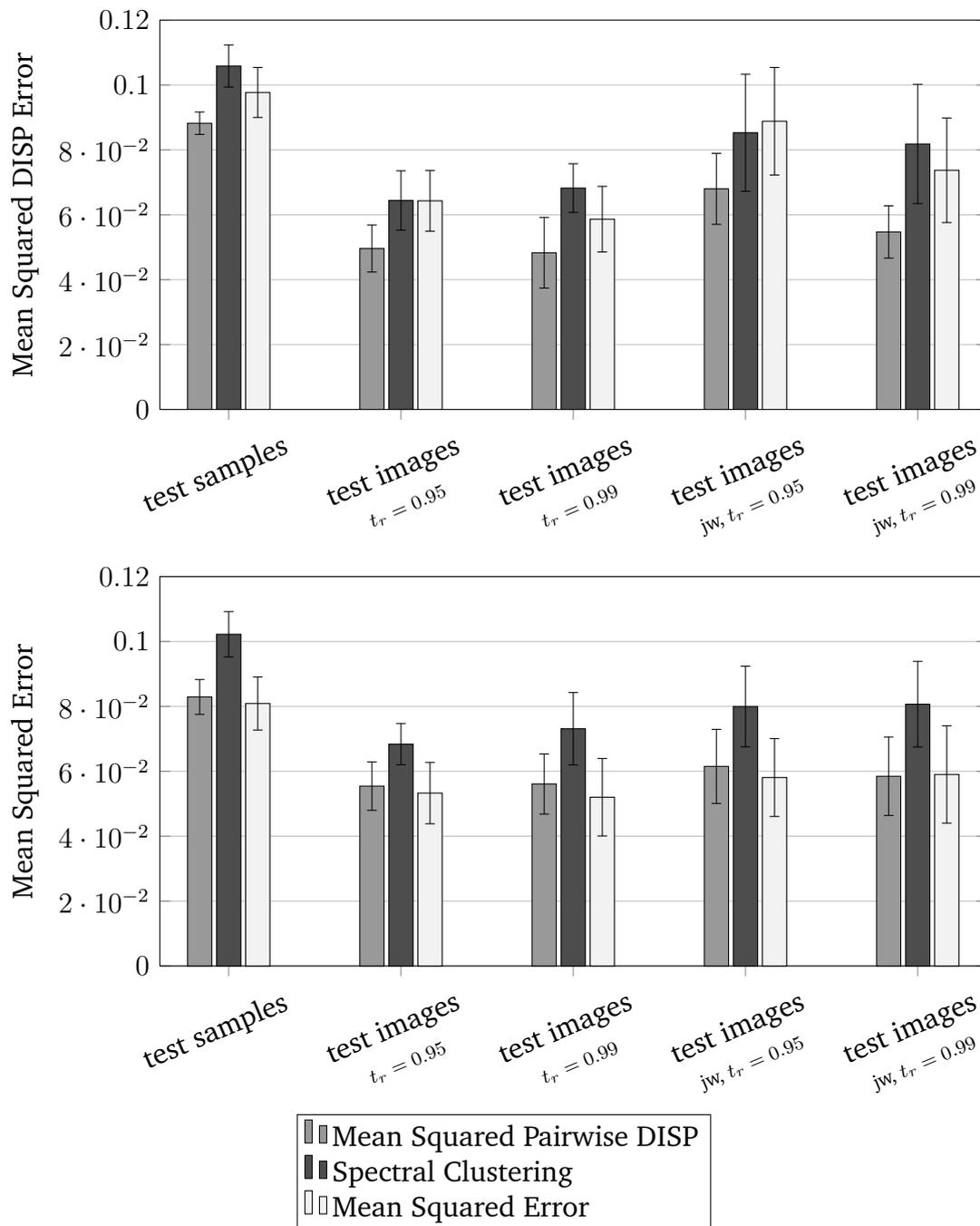


Figure 4.8.: Results of a 5-fold cross validation on the dataset “set200” with standard deviation over the folds. The upper plot shows the mean squared DISP error, the lower one the mean squared error of the joint angles. “jw” is short for joint-wise.

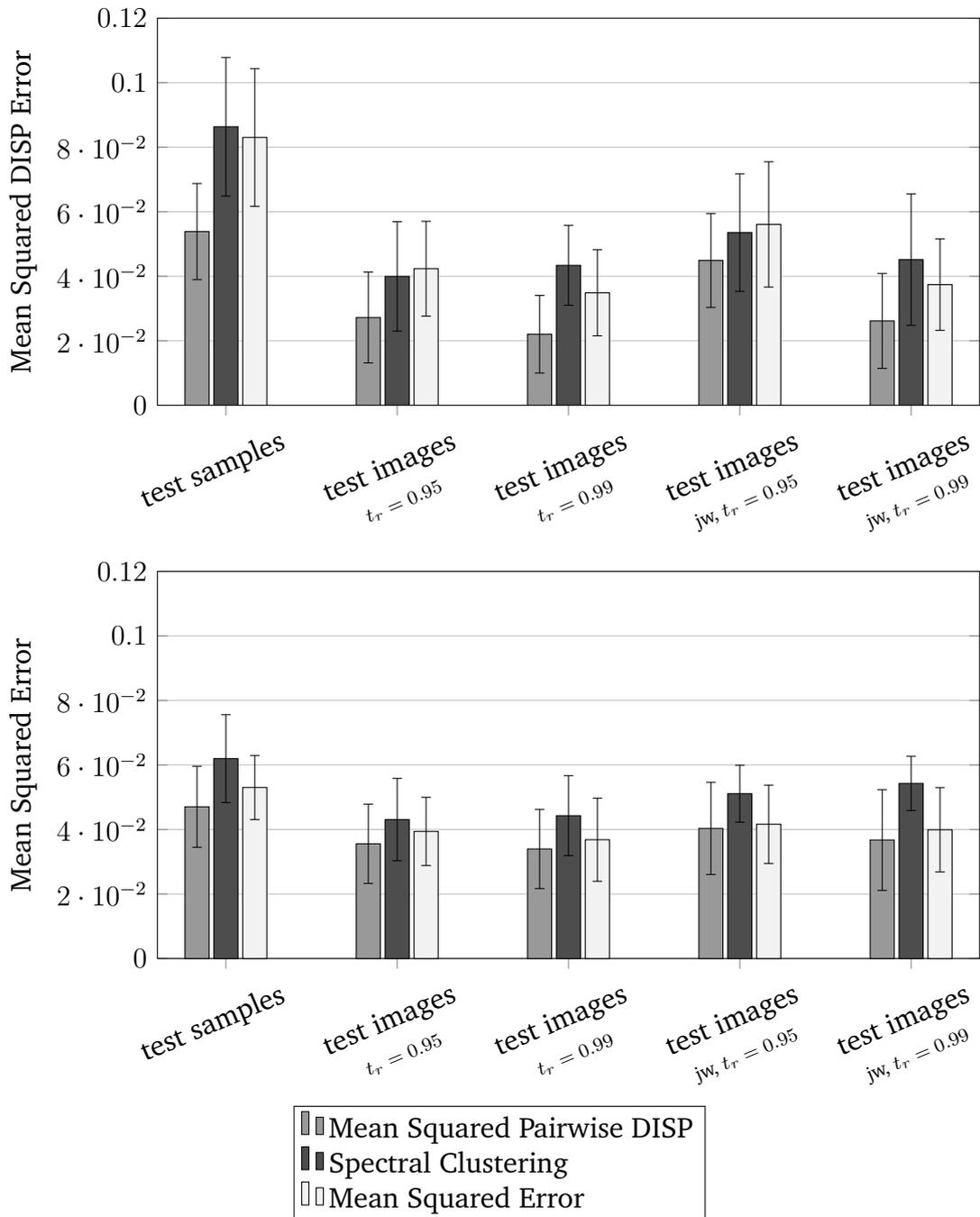


Figure 4.9.: Same as in Figure 4.8 but using the dataset “fixcam200”. Note that for the MSPD criterion only the first four folds were used for evaluation, because the training of all five folds took too long and did not finish before printing this thesis.

4. Results

Table 4.3.: Analysis of the structure of the forest resulting from the first fold of cross validation on “set200”.

(a) MSPD Criterion

Tree	Depth	# Leaves	Highest Leaf	avg. Leaf Depth	avg. MSPD in Leafs
0	40	5945	5	18	0.1216
1	39	5967	5	18	0.1177
2	33	5933	5	18	0.1193
3	41	5976	5	18	0.1192
4	41	6002	5	18	0.1180

(b) Spectral Cluster Criterion

Tree	Depth	# Leaves	Highest Leaf	avg. Leaf Depth	avg. MSPD in Leafs
0	73	5482	9	16	0.1728
1	82	5721	9	16	0.1741
2	74	5483	9	16	0.1713
3	99	5681	10	16	0.1715
4	76	5518	9	16	0.1715

(c) MSE Criterion

Tree	Depth	# Leaves	Highest Leaf	avg. Leaf Depth	avg. MSPD in Leafs
0	44	5966	6	19	0.1433
1	42	5930	6	20	0.1432
2	46	5936	6	20	0.1422
3	54	6053	4	20	0.1448
4	48	6022	5	21	0.1462

Table 4.4.: Analysis of the structure of the forest resulting from the first fold of cross validation on “fixcam200”.

(a) MSPD Criterion

Tree	Depth	# Leaves	Highest Leaf	avg. Leaf Depth	avg. MSPD in Leafs
0	63	6326	5	25	0.1898
1	75	6304	4	27	0.1896
2	79	6324	4	27	0.1856
3	58	6281	5	24	0.1877
4	73	6332	5	28	0.1856

(b) Spectral Cluster Criterion

Tree	Depth	# Leaves	Highest Leaf	avg. Leaf Depth	avg. MSPD in Leafs
0	45	5019	9	15	0.3095
1	45	5047	8	14	0.3107
2	58	5137	9	15	0.3064
3	66	5101	7	15	0.3062
4	61	5148	9	15	0.3091

(c) MSE Criterion

Tree	Depth	# Leaves	Highest Leaf	avg. Leaf Depth	avg. MSPD in Leafs
0	73	6288	6	28	0.2200
1	65	6265	5	25	0.2237
2	76	6288	6	27	0.2175
3	75	6297	5	27	0.2166
4	69	6229	5	29	0.2224

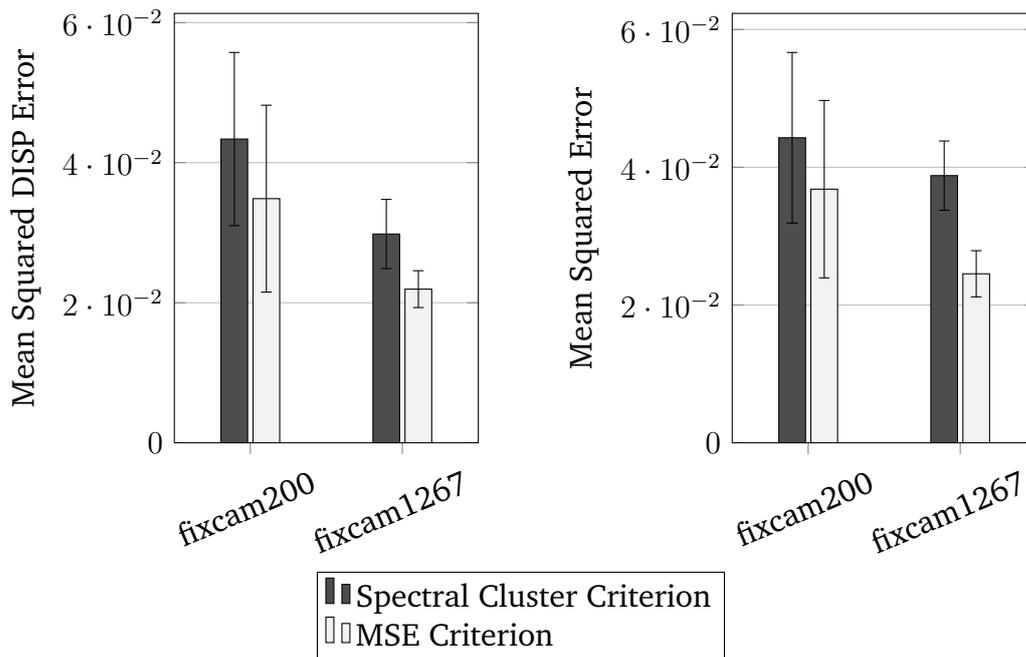


Figure 4.10.: Comparison of forests trained on datasets of different size (200 and 1267 images). The *test on images* using pose-wise confidence threshold with $t_r = 0.99$ was used to measure the estimation accuracy.

4.5.1. Tests with larger dataset

The datasets used in the evaluations above are comparatively small to keep training time of the very slow MSPD criterion within reasonable bounds. The same tests were done on the larger dataset “fixcam1267”, but only using the faster spectral cluster and MSE criteria. The results are compared to the smaller “fixcam200” set in Figure 4.10. It can be seen that increasing the training set considerably improves both estimation accuracy and standard deviation among the cross validation folds.

4.6. Prediction on images

This section goes a bit more into detail about the pose estimation on new depth images as described in Section 3.3. The estimation error on images was already used in the former section (“test image”) to compare the performance of the different split criteria. In this section, different methods to combine the estimations of the single pixels are compared.

Figure 4.11 shows an exemplary depth image and the resulting pose estimations. In the accompanying error image, it can nicely be seen that the prediction error of single pixel predictions is significantly less (marked by darker colour) for pixels that lie directly on the arm, compared to background pixels. For the background the error is still less in the area near to the arm, where some of the features still look at arm pixels, and gets worse in regions where no part of the arm is within the window

that is covered by the features.

The visualization of the arms show the estimated poses, when the single pixel predictions are combined with different methods. The green arm shows the ground truth. Due to the many background pixels with comparatively high estimation errors, the simple mean over all pixels (blue) leads to a very bad estimation. As the actual distribution of joint votes of the single pixels do not differ too much from a Gaussian distribution (compare Figure 3.3 on page 35), the estimation using the histogram (cyan) is very close to the mean estimation. Due to their higher number, the erroneous background pixels dominate over the foreground pixels and thus impair the estimation of both methods.

The confidence-weighted mean in contrast, incorporates only the most confident pixels and thus yields a much better prediction. It can be seen, that these most confident pixels all lie in the area of the arm, where the prediction error is considerably less than in other regions of the image. For $t_r = 0.99$ the number of used pixels significantly decreases compared to $t_r = 0.95$, and only very few pixels are considered. The prediction improves, however, so the chosen pixels seem to have very good estimates.

4.7. Testing on real data

After doing all training and testing on synthetic data, let's see, how the trained forest performs on real data. Since, for real depth images of the robotic arms, no ground truth joint angles are available, only a qualitative test could be made with a few poses recorded on the real robot. The results are shown in Figure 4.12.

While there is a considerable error in Figure 4.12a, in Figure 4.12b the pose estimation of the forest is significantly better than the estimate of the joint encoders. In Figure 4.12c the prediction is completely off, which can be explained by the strange pose of the arm, which is very far from any of the poses in the training data.

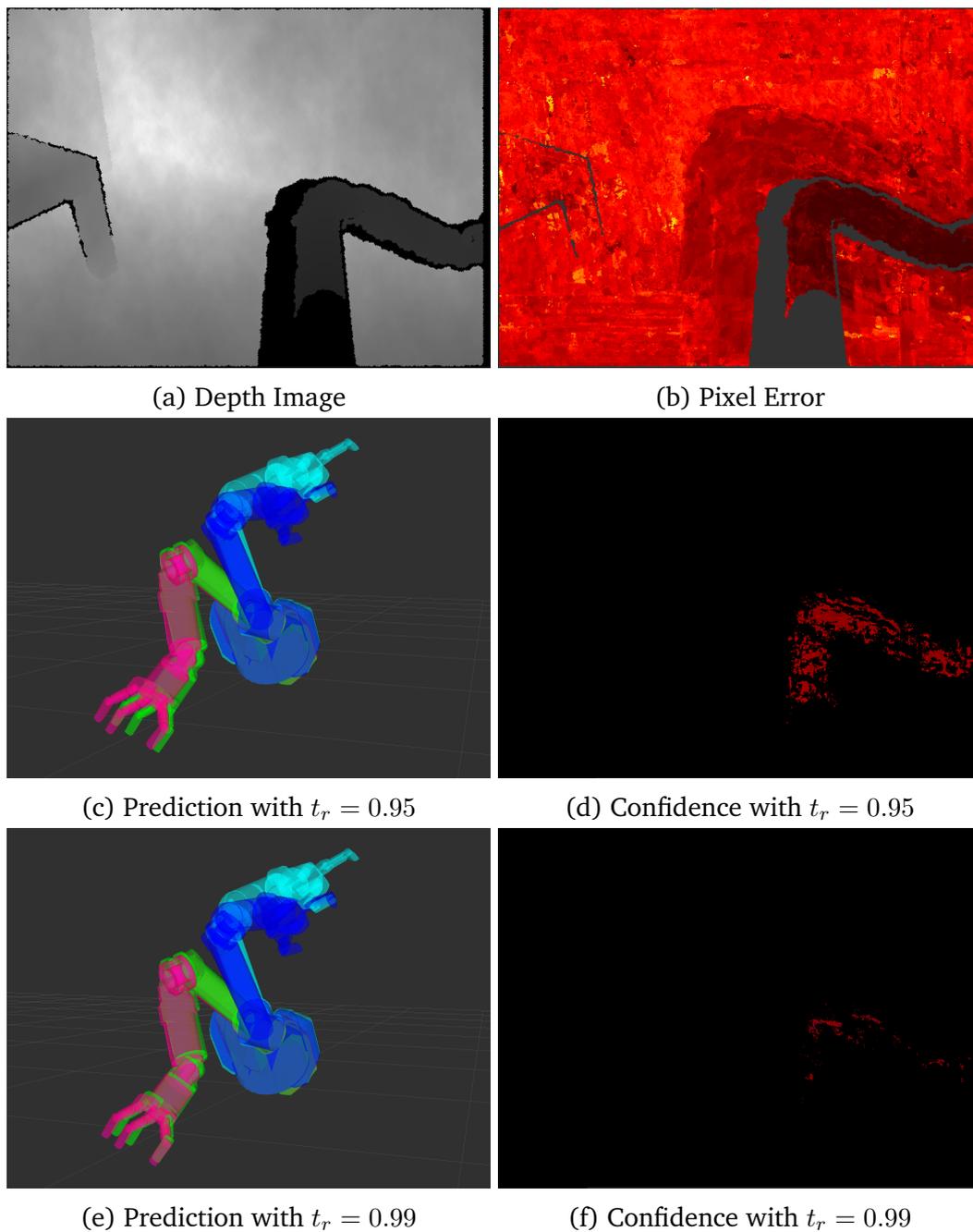


Figure 4.11.: Pose estimation on a depth image. (a) shows the depth image, black pixels denote invalid measurements. In (b) for each pixel the prediction error is visualized. The brighter the pixel, the higher is the error. (c) and (e) visualize the predictions. Green is the ground truth pose, blue the unweighted mean over all pixels, cyan the result of the histogram prediction and red the confidence-weighted mean, using different thresholds on the confidence range. Confidence of pixels above the threshold are visualized in (d) and (f) respectively (brighter pixels have higher confidence).

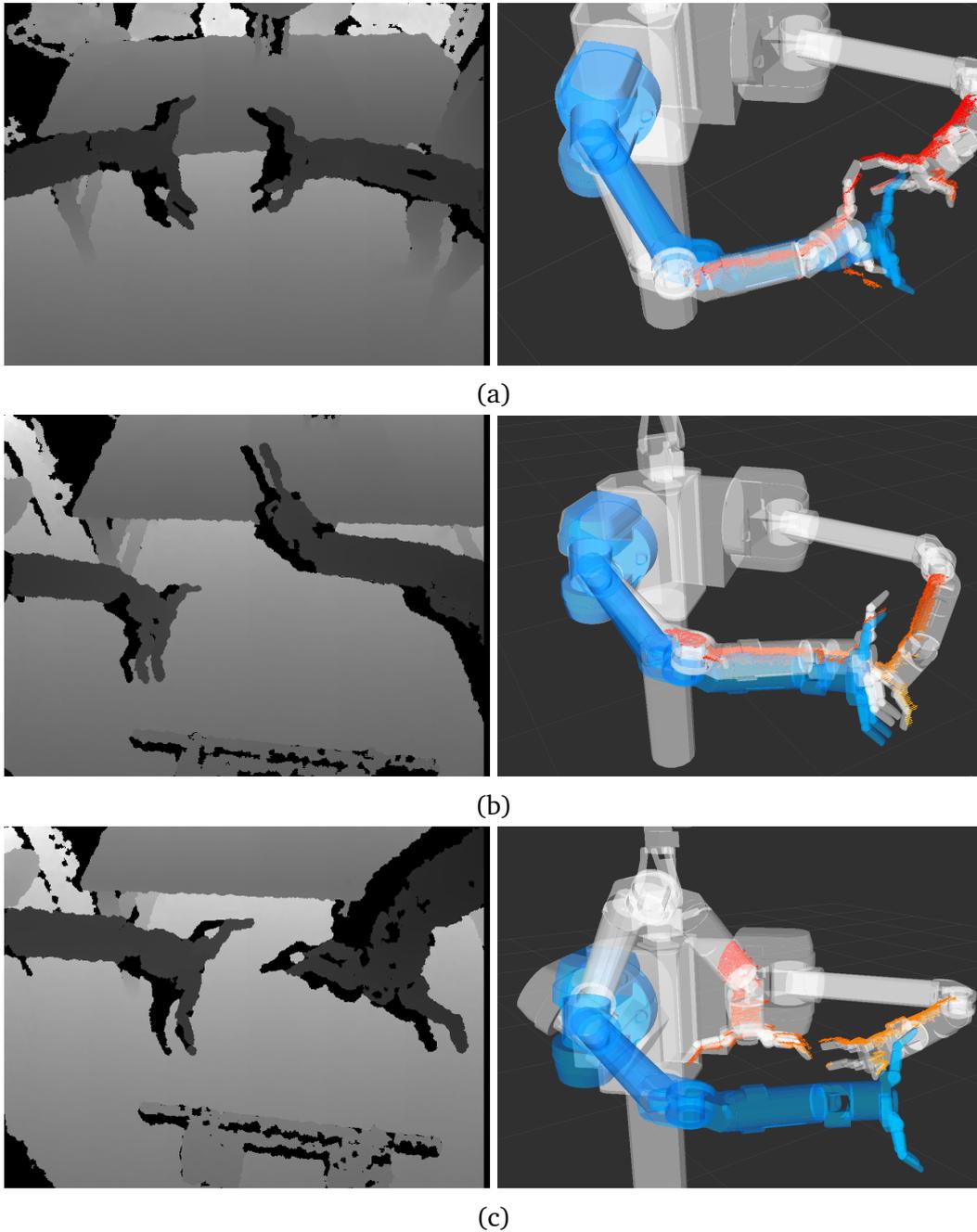


Figure 4.12.: Estimations on real depth images. The left column shows the depth images, the right column visualizes in grey the robot pose calculated from the joint encoder estimates, in orange the point cloud obtained from the depth images and in blue the estimate of the forest for the right arm. A forest with 5 trees trained on “set2000” using the SC criterion and a pose-wise confidence threshold of $t_r = 0.99$ was used.

5. Discussion

5.1. C-DIST

When reimplementing the C-DIST algorithm of [ZKM07], the promising results of the paper could unfortunately not be reproduced completely as is shown in Section 4.2. Building the BVH as described in the paper did actually lead to a significant decrease in performance for the tested rigid models. Since the used objective function for the BVH construction (Equation 2.16) only maximizes the node densities but does not take the cardinality of the subsets into account, in most cases only minimal subsets with three points¹ were split off. The resulting tree structures degenerated more or less to simple lists that hardly branched at all. In consequence, the gain of pruning subtrees away was outweighed by the additional effort of computing the distance bound of the SSV in every node.

Modifying the objective finally led to a better performance, which is, however, still only comparable to the method without the BVH. This discrepancy in the results can probably be explained by some misunderstanding of the method or a missing detail in the description.

On the robot model, the results looked much better than for the rigid models; there the BVH brought indeed a significant increase in performance. Nonetheless, DISP computation was too slow to be done on-line during training. Instead, distances were precomputed only once to build a lookup table for training. With the lookup table, runtime of C-DIST was no crucial issue any more, so the bad performance of the original method was not investigated further and no more precise analysis of the BVH construction was done.

5.2. Split Criteria

The comparison of the different split criteria for forest training showed, that the usage of the DISP distance in the *mean squared pairwise DISP* (MSPD) criterion indeed significantly improved the displacement error in workspace. The *mean squared error* (MSE) criterion performed a bit better in terms of the error in configuration space (i.e. mean squared error on the joint angles) which is no surprise, as this is exactly the value this criterion is optimizing. However, although not explicitly optimized for minimizing the MSE in configuration space, the forest trained with the MSPD criterion achieve similar performance in terms of MSE. A drawback of MSPD

¹The minimal allowed number of points in a node was set to three, as for smaller sets computation of the SSV is not possible.

is its extremely slow performance at training time², which makes it infeasible for practical application. This extreme slowdown can be explained by the increase in complexity for the test of a single split feature from linear to cubic time.

An attempt to benefit from the increased accuracy of the DISP metric while reducing the training time was done with the spectral clustering (SC) based criterion. While this criterion is indeed much faster, the evaluation showed, that it performs only comparable to or even slightly worse than the MSE in terms of accuracy. An explanation could be, that for a split candidate, it is only qualitatively checked whether samples are on the “right” side or not, but no quantitative measure of the amount of “falseness” is done. This means, a sample that lands on the wrong side of the split is always penalized with the same score, no matter if it is completely off from the other samples in the same subset or just somewhere in between of the two clusters so that the error is relatively low. A way to verify this assumption, could be to implement another SC based split criterion, that not just counts samples that are on the wrong side, but also weights them with their distance to the cluster centroid. Unfortunately there was not enough time any more, to implement this as part of the thesis so it has to go to the future work.

When implementing this new criterion, one has to remember the issues that arose with the k-means clustering and showed, that the mean on the joint angles is not always a good cluster centroid (see Figure 4.4a on page 44). The cluster centroid should therefore be computed like in the modified k-means, where the sample of the cluster with minimal distance to all other samples was used. Computing this is, however, expensive again and would slow the training down.

5.3. Use of Confidences

The evaluation shown above in Figure 4.11 showed, that the introduction of confidences for the predictions was crucial for the success of the pose estimation. Without a given segmentation of the foreground, many background pixels are considered, that cannot provide any useful information about the arm pose. Fortunately the confidence of such high error pixels is also relatively low. By considering confidences and only using the most confident predictions, one automatically restricts the estimation to mainly rely on foreground pixels, without the need to actively separate foreground from background. For comparison, the former method of [BRHS14] does also not need a prior segmentation but does this implicitly when classifying the pixels as either a part of the robot or background.

It was a bit of a surprise that the usage of joint-wise confidence led to worse estimation results than the simpler pose-wise confidence. Joint-wise confidence was assumed to work better, as it enables pixels to vote only for some of the joints. This seemed reasonable as for example a pixel somewhere around the elbow joint can have good knowledge of this joint but cannot know anything about the finger joints. In practice this obviously did not work out as expected as throughout all

²training on the rather small “fixcam200”-set takes already about a week on a high-end PC.

tests the pose-wise confidence, where a pixel can only vote for all or for no joint, performed better. So far, no explanation for this behaviour was found, but it could be further analysed in future work.

5.4. Comparison to the Former Approach

As already pointed out in the introduction to Chapter 3, the approach of this thesis considerably reduced the number of necessary computations for the pose estimation, compared to the former work of [BRHS14] (see Figure 3.2 on page 29). In the first instance, the whole step of iteratively computing the inverse kinematics was made unnecessary by directly estimating joint angles with the random forest instead of going the long way round and first estimating 3D joint positions. But also the step of combining the forest votes of the single pixels was significantly simplified. In [BRHS14], 3D position votes are cast, which have to be clustered separately for each joint. In the new approach of this thesis, only the weighted mean of the single pixel votes has to be computed, which is straightforward and has complexity linear in the number of pixels.

In the evaluations done so far, the forest is only trained to estimate the pose of one arm, while in [BRHS14] both arms are tracked at once. In principle it is possible to train the new method on both arms too, without having to change anything, but no evaluations in this direction were done, due to lack of time. In any case, it would be possible to simply run two separate forests, one for each arm. As they could run in parallel and share computed feature values, the additional cost should be relatively low.

There is a limitation compared to [BRHS14]: Right now, the new method is restricted to a fixed camera pose. Since in [BRHS14] 3D joint positions are determined relative to the camera frame, this method automatically adapts to a change in the camera pose. In this work, however, joint angles are estimated, which are independent of the camera pose. This makes the pose estimation prone to ambiguities that can, for example, arise, when arm and camera rotate by the same amount around the same axis. This situation is illustrated in Figure 5.1. The problem is also affirmed by the fact, that the overall prediction error was recognizably less for the “fixcam200” dataset with only one fixed camera pose than for the “set200” which contains images with varying camera poses. Solving it remains a challenge for future work, maybe by adding the camera configuration to the feature vector at training time.

5.5. Runtime

To be actually useful to control the robot arm in a real world application, the pose estimation should be able to run in real-time. This was not yet achieved with the implementation for this thesis. The whole estimation process of computing features and estimating pose for every pixel of the 640×480 pixel depth image and

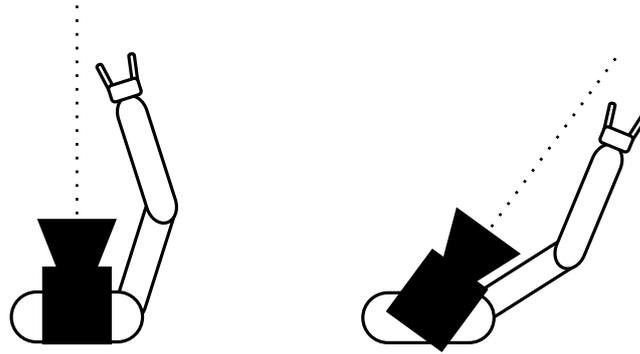


Figure 5.1.: Camera Pose Ambiguity. The figure shows a simplified robot with one arm and a head-mounted camera in two different poses. When arm and camera are rotated about the same amount, the image recorded of the arm will not change significantly, while the actual pose of the arm changes a lot. This ambiguity cannot be resolved by the regression forest and leads to bad pose estimations, when varying camera poses are used.

combining these pixel-votes currently takes about 8–9 seconds on the machine that was used for the evaluation. The current implementation is not considerably optimized, though. There are several ways how it could be accelerated: In general, overall runtime could be improved by moving away from Python, which is relatively slow, compared to C++, and especially in the case of the SC based split criterion prevented fully parallel training of the individual trees³.

There is, however, also room for improvement, independent of the actual language. The computation of feature values, for example, could be improved in various ways. Currently a dense, non-parallelized computation of all 500 features for every pixel is done in advance. This takes roughly 5 seconds and therefore makes the greatest part in the whole estimation process. Since the number of feature-tests that are actually done during prediction is limited by the depth of the tree (for training sets with 200 images, the average leaf depth is in the range of 15–30, see Section 4.5), many of the computed feature values may never be used, though. It would therefore be better to compute the features only on demand instead of precomputing them for the whole image, as it is currently done (scikit-learn unfortunately does not support such lazy evaluation).

Moreover, the number of evaluated pixels could be reduced. Instead of applying the random forest on every single pixel in the image, it may suffice to only consider pixels in the area where the arm is expected. This area could be determined by back projection of the arm into the image, based on given encoder readings.

³The scikit-learn implementation of random forests is done in Cython. Cython allows the usage of Python functions but they require the *global interpreter lock* (GIL) which prevents parallel execution.

By applying these and other possible optimizations on the current implementation, reaching real-time performance should be possible, especially since the approach of [SGF⁺12] is running successfully in real-time on the almost ten years old Xbox 360 gaming console.

6. Conclusions and Future Work

A novel approach of robot arm pose estimation was presented. Based on synthetically generated depth images, a random regression forest was trained, which estimates full arm poses based on simple depth features. It improved [BRHS14] by considerably reducing the computational effort at test time as the forest directly estimates joint angles while in [BRHS14] the forest only cast 3D position votes for the joints, which then have to be clustered and fed into an iterative inverse kinematic process to finally get the joint angles. The pose estimation works directly on raw depth images without the need of prior foreground segmentation or other preprocessing steps.

Despite of being trained on synthetic data only, it was shown that the resulting forest is able to provide reasonable estimations for real data at test time. The current implementation does not yet achieve real-time performance but this should be possible with a better optimized implementation of the same methods, considering that the approach of [SGF⁺12] (which was the basis of [BRHS14]) runs in real-time on customer hardware.

Future Work

First of all, the evaluations done in this thesis, could be extended in several way:

- Due to the lack of reliable ground truth information, no quantitative evaluation on real sensor data was done so far. Such ground truth information could be obtained using a state-of-the-art motion capturing system.
- Tests should be done to see, if the estimator can handle scenes where the robot is interacting with other objects. To cope with such situations, the training set may has to be extended with corresponding images.
- Considering the long training time of the mean squared pairwise DISP (MSPD) split criterion, only relatively small training sets were used for the evaluation. The results presented in Section 4.5.1 showed, however, that increasing the training set can lead to considerably better results. It would therefore be interesting to see, how much the estimation can be improved by increasing the amount of training data. It should also be analysed, how increasing the number of trees in the forest influences the estimation result.
- Unfortunately, it was not possible any more to quantitatively compare the estimation accuracy of the new method with the old one of [BRHS14]. As this

is, besides of runtime complexity, also an important question when comparing the two methods, such an evaluation will be done as soon as possible.

Besides from the evaluation, some issues remain open, that can be tackled in future work: As already stated in Chapter 5, the spectral clustering based split criterion can be improved by not only counting wrong samples in the split subsets, but rather weighting them with their distance to the cluster centroid. This way, estimation accuracy of the forest may be improved while still keeping the training time within reasonable bounds.

The depth features used in this thesis were adopted from [BRHS14] and are a simplified version of those used in [SGF⁺12]. It was not part of this thesis to further analyse these features, but it would be interesting to see if the usage of other features can reduce the estimation error.

Another remaining issue that should be further investigated is inability of the presented method to cope with varying camera poses. For a robot with a moveable head, this is a serious limitation, so solving this problem is of importance for the practicability of the proposed method. One attempt could be, to add the angular joint configuration of the neck joints—which are responsible for the actual camera pose—to the feature vector of the data points that are used for training.

Finally, the random forest implementation could be further extended. The scikit-learn regression forests innately only uses the mean of the samples in a leaf, as the prediction output of this leaf. This was already extended by uncertainty values in Section 3.2 but still only the mean of the data is used as value, which implicitly assumes a Gaussian distribution of the samples in the leaves. This is not necessarily the case, distributions in leaf nodes can be non-Gaussian, even multi-modal. Therefore, a more complex representation of the data in the leaves could be used, where a leaf is able to cast multiple (weighted) votes for multi-modal distributions.

A. Appendix

A.1. Implementation Details

A.1.1. Software Libraries

While implementing the software for this work, I used a bunch of open source software libraries, that made life much easier. While giving a complete list is beyond the scope of this section, I want at least mention the most important ones here:

ROS: A popular framework for robotic systems with good visualization tools [QCG⁺09]. For this thesis version *Groovy Galapagos* was used.

Boost: A large bundle of basic C++ libraries. I mainly used the module `Boost::Python`¹ to port the C-DIST implementation from C++ to Python.

Eigen: An efficient library for linear algebra [GJ⁺10].

mlpack: A C++ machine learning library. I used it for its k-means implementation that provides an interface to use custom distance metrics. [CCS⁺13]

NumPy: Python library that implements high-level matrix/array-types and a lot of mathematical functions [vdWCV].

scikit-learn: A Python machine learning library [PVG⁺11]. Used for its random forest implementation.

A.1.2. Code Optimization

When starting the implementation for this thesis, I originally planned to compute DISP on-line during the forest training. As DISP is computed *very* often during the training, I put quite some effort in optimizing this computation as good as possible. Besides from using a more efficient algorithm, this also included several low-level optimizations of the code. In this section I document some insights I learned from this, that are not of any scientific value, but may be interesting for other people who are implementing real-time critical software.

¹<http://www.boost.org/libs/python>

Partial reduction in Eigen. The Eigen library has a feature called *partial reduction*, which allows column-wise and row-wise operations without a loop. This is very convenient, as the code gets shorter and is easier to understand. For example finding the DISP distance between two poses by computing the displacement of each point can be done in one single line:

```
double disp = (vertices1 - vertices2).colwise().norm().maxCoeff();
```

where `vertices1` and `vertices2` are the model points of the two poses stored column-wise in matrices. While this is very fancy and produces easy readable code, it is unfortunately also considerably slower than using a loop to iterate over the columns. Therefore in functions that are called very often, it is better to use the loop.

Homogeneous transformation matrices in Eigen. In Eigen transformations can handle both simple and homogeneous coordinates. At least for a release build, the simple coordinates were a bit faster (and also made the code simpler...).

NumPy's `vstack/hstack`. NumPy provides two functions `vstack` and `hstack`, that merge arrays vertically and horizontally. Since NumPy arrays are fixed size data structures, these methods internally copy all the data to a new array. This is relatively slow and when working with huge datasets, it can even lead to an out-of-memory error. Therefore, the `*stack`-functions should be avoided if possible, for example by directly loading all data in one array that is initialized to the final size right in the beginning.

Debug Build vs Release Build. It is probably an obvious thing and not worth mentioning at all, that a release build is faster than a debug build due to optimizations done by the compiler. I was, however, surprised how big the difference can be. For the implementation of the C-DIST algorithm, switching to release build resulted in a speed-up of about factor 30.

A.2. Visualization of Bounding Volume Hierarchies

The following Figures A.1 and A.2 were moved to the appendix as they are very big and would break the float of text too much. They show visualizations of the resulting bounding volume hierarchies (BVHs) that were constructed with the rules described in Section 4.2.

A.3. Note on the enclosed CD-ROM

Enclosed to the printed version of this thesis is a CD-ROM containing a PDF-version of this document all the source code that was implemented for this thesis.

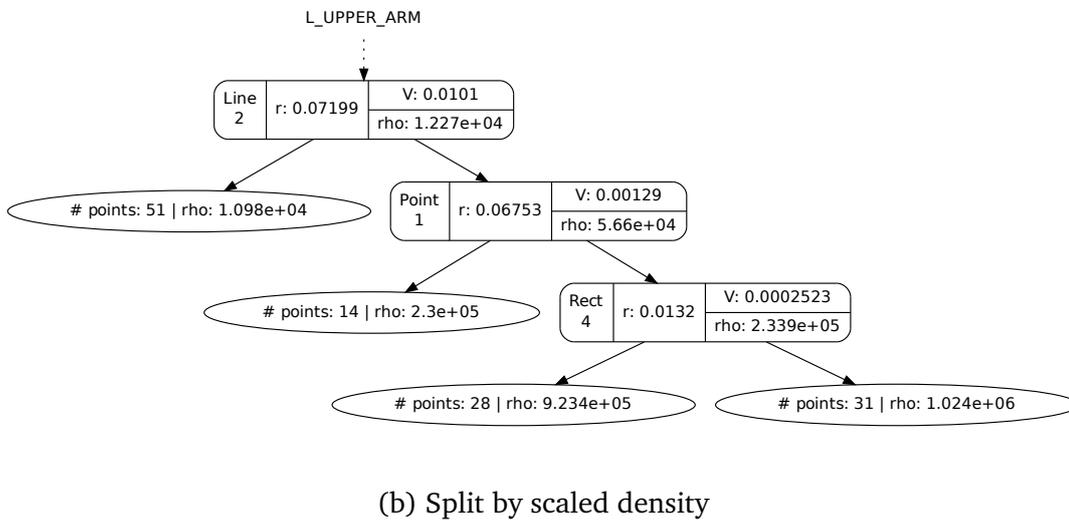
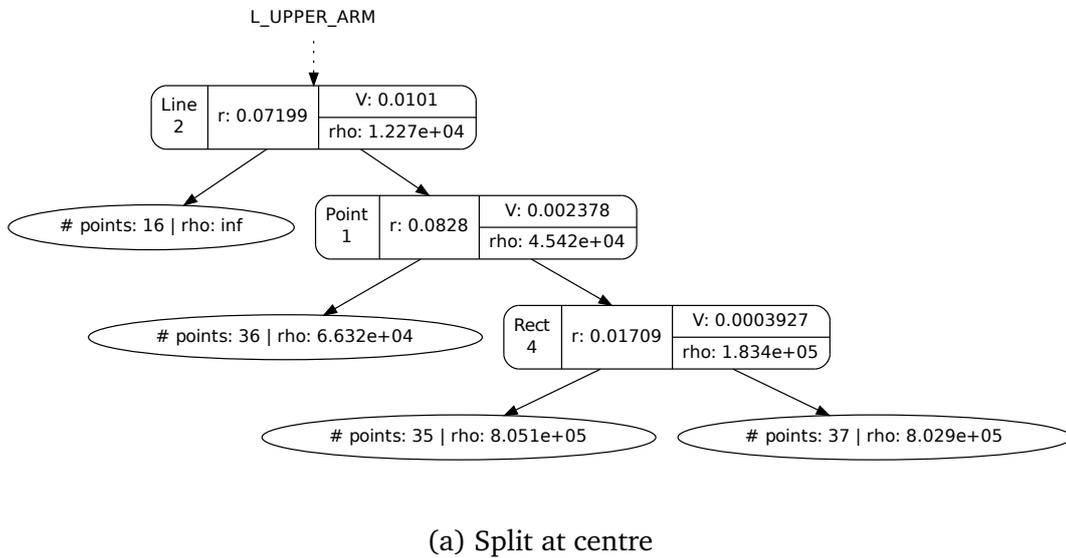


Figure A.1.: Resulting BVH for the left upper arm link using the split policies “split at centre” and “split by scaled density”. At each node, the type of the SSV (point, line or rect), the radius r of the sphere as well as volume V and density ρ are shown.

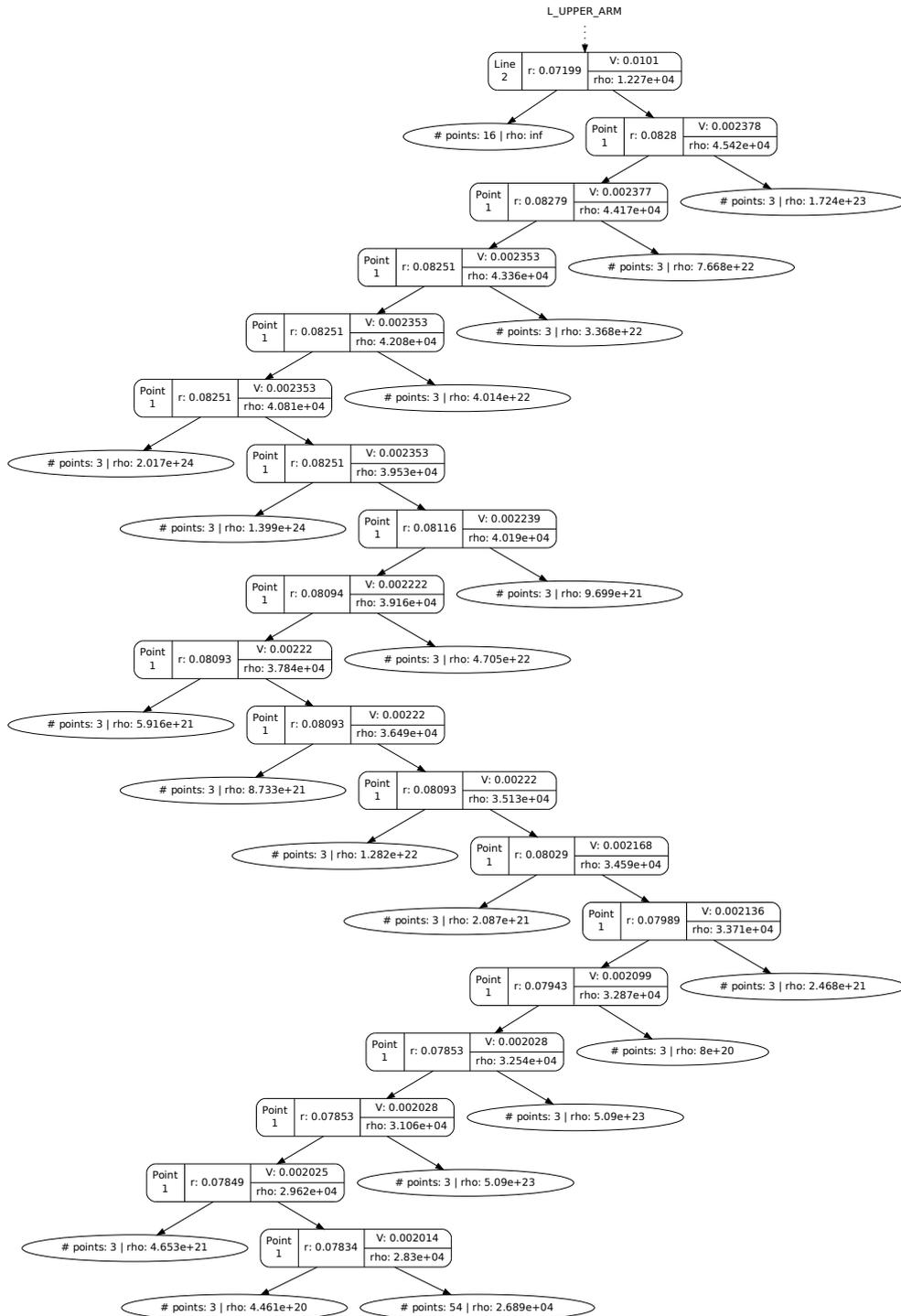


Figure A.2.: Resulting BVH for the left upper arm link using the split policy “split by density”. In most cases only three points with extremely high density are split off, resulting in a list-like tree structure.

Abbreviations

BVH	bounding volume hierarchy
CH	convex hull
ICP	iterative closest points
LSS	line swept sphere
MSDE	mean squared DISP error
MSE	mean squared error
MSPD	mean squared pairwise DISP
OBB	oriented bounding box
PSS	point swept sphere
RBF	radial basis function
RSS	rectangle swept sphere
SC	spectral clustering
SSV	swept sphere volume

Bibliography

- [AG94] Yali Amit and Donald Geman. Randomized Inquiries About Shape: An Application to Handwritten Digit Recognition. Technical report, DTIC Document, 1994.
- [Bre01] Leo Breiman. Random Forests. *Machine Learning*, 45(1):5–32, 2001.
- [BRHS14] J. Bohg, J. Romero, A. Herzog, and S. Schaal. Robot Arm Pose Estimation through Pixel-Wise Part Classification. In *Proceedings of the IEEE International Conference on Robotics and Automation*, 2014.
- [CCS⁺13] Ryan R. Curtin, James R. Cline, Neil P. Slagle, William B. March, P. Ram, Nishant A. Mehta, and Alexander G. Gray. MLPACK: A Scalable C++ Machine Learning Library. *Journal of Machine Learning Research*, 14:801–805, 2013.
- [cga] CGAL: Computational Geometry Algorithms Library. <http://cgal.org>.
- [CS13] Antonio Criminisi and Jamie Shotton. *Decision Forests for Computer Vision and Medical Image Analysis*. Springer Science & Business Media, 2013.
- [CSK12] Antonio Criminisi, Jamie Shotton, and Ender Konukoglu. Decision Forests: A Unified Framework for Classification, Regression, Density Estimation, Manifold Learning and Semi-Supervised Learning. *Foundations and Trends in Computer Graphics and Vision*, 7(2-3):81–227, 2012.
- [DMWG09] Marie Dumont, Raphaël Marée, Louis Wehenkel, and Pierre Geurts. Fast multi-class image annotation with random windows and multiple output randomized trees. In *Proc. International Conference on Computer Vision Theory and Applications (VISAPP) Volume*, volume 2, pages 196–203, 2009.
- [GBBK10] Xavi Gratal, Jeannette Bohg, Mårten Björkman, and Danica Kragic. Scene representation and object grasping using active vision. In *IROS'10 Workshop on Defining and Solving Realistic Perception Problems in Personal Robotics*, 2010.
- [GJ⁺10] Gaël Guennebaud, Benoît Jacob, et al. Eigen v3. <http://eigen.tuxfamily.org>, 2010.

- [GKUP11] Michael Gschwandtner, Roland Kwitt, Andreas Uhl, and Wolfgang Pree. BlenSor: Blender Sensor Simulation Toolbox. In George Bebis et al., editors, *Advances in Visual Computing*, volume 6939 of *Lecture Notes in Computer Science*, pages 199–208. Springer Berlin Heidelberg, 2011.
- [GL13] Juergen Gall and Victor Lempitsky. Class-specific hough forests for object detection. In *Decision Forests for Computer Vision and Medical Image Analysis*, pages 143–157. Springer, 2013.
- [GRB13] M Godec, PM Roth, and H Bischof. Hough-Based Tracking of Deformable Objects. In *Decision Forests for Computer Vision and Medical Image Analysis*, pages 159–173. Springer, 2013.
- [GRBK12] Xavi Gratal, Javier Romero, Jeannette Bohg, and Danica Kragic. Visual servoing on unknown objects. *Mechatronics*, 22(4):423–435, 2012.
- [Ham50] Richard Wesley Hamming. Error Detecting and Error Correcting Codes. *Bell System Technical Journal*, 29(2):147–160, April 1950.
- [HHM⁺12] P. Hebert, N. Hudson, J. Ma, T. Howard, T. Fuchs, M. Bajracharya, and J. Burdick. Combined shape, appearance and silhouette for simultaneous manipulator and object tracking. In *Robotics and Automation (ICRA), 2012 IEEE International Conference on*, pages 2405–2412, May 2012.
- [Ho95] Tin Kam Ho. Random decision forests. In *Document Analysis and Recognition, 1995., Proceedings of the Third International Conference on*, volume 1, pages 278–282 vol.1, Aug 1995.
- [HP04] Michael Hofer and Helmut Pottmann. Energy-minimizing Splines in Manifolds. *ACM Trans. Graph.*, 23(3):284–293, August 2004.
- [KHRF11] Michael Krainin, Peter Henry, Xiaofeng Ren, and Dieter Fox. Manipulator and object tracking for in-hand 3d object modeling. *The International Journal of Robotics Research*, 30(11):1311–1327, 2011.
- [Kuf04] J.J. Kuffner. Effective sampling and distance metrics for 3D rigid body path planning. In *Robotics and Automation, 2004. Proceedings. ICRA '04. 2004 IEEE International Conference on*, volume 4, pages 3993–3998 Vol.4, April 2004.
- [KVLMO3] Young J. Kim, Gokul Varadhan, Ming C. Lin, and Dinesh Manocha. Fast Swept Volume Approximation of Complex Polyhedral Models. In *Proceedings of the Eighth ACM Symposium on Solid Modeling and Applications, SM '03*, pages 11–22, New York, NY, USA, 2003. ACM.

- [Lat91] Jean-Claude Latombe. *Robot Motion Planning*. The Springer International Series in Engineering and Computer Science. Springer US, 1991.
- [LaV06] Steven M. LaValle. *Planning Algorithms*. Cambridge University Press, Cambridge, U.K., 2006. Available at <http://planning.cs.uiuc.edu>.
- [LF13] Vincent Lepetit and Pascal Fua. Keypoint Recognition Using Random Forests and Random Ferns. In *Decision Forests for Computer Vision and Medical Image Analysis*, pages 111–124. Springer, 2013.
- [LGLM00] Eric Larsen, Stefan Gottschalk, Ming C. Lin, and Dinesh Manocha. Fast Proximity Queries with Swept Sphere Volumes. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 3719–3726, April 2000. Also available as tech. rep. TR99-018, Dept. Comput. Sci., Univ. N. Carolina Chapel Hill, 1999.
- [Mac67] J. MacQueen. Some methods for classification and analysis of multivariate observations. In *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability, Volume 1: Statistics*, pages 281–297, Berkeley, Calif., 1967. University of California Press.
- [PVG⁺11] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.
- [QCG⁺09] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully B. Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y. Ng. ROS: an open-source Robot Operating System. In *ICRA Workshop on Open Source Software*, 2009.
- [RKP⁺14] L. Righetti, M. Kalakrishnan, P. Pastor, J. Binney, J. Kelly, R. C. Voorhies, G. S. Sukhatme, and S. Schaal. An autonomous manipulation system based on force control and optimization. (1-2):11–30, 2014.
- [Sch13] Rolf Schneider. *Convex bodies: the Brunn–Minkowski theory*, volume 151. Cambridge University Press, 2013.
- [SGF⁺12] Jamie Shotton, Ross Girshick, Andrew Fitzgibbon, Toby Sharp, Mat Cook, Mark Finocchio, Richard Moore, Pushmeet Kohli, Antonio Criminisi, Alex Kipman, and Andrew Blake. Efficient Human Pose Estimation from Single Depth Images. *Trans. on Pattern Analysis and Machine Intelligence*, 2012.
- [SJC08] Jamie Shotton, Matthew Johnson, and Roberto Cipolla. Semantic texon forests for image categorization and segmentation. In *Computer*

- vision and pattern recognition, 2008. CVPR 2008. IEEE Conference on*, pages 1–8. IEEE, 2008.
- [SMA⁺11] Olivier Stern, Raphaël Marée, Jessica Aceto, Nathalie Jeanray, Marc Muller, Louis Wehenkel, and Pierre Geurts. Automatic Localization of Interest Points in Zebrafish Images with Tree-based Methods. In *Proceedings of the 6th IAPR International Conference on Pattern Recognition in Bioinformatics*, PRIB'11, pages 179–190, Berlin, Heidelberg, 2011. Springer-Verlag.
- [SNF14] Tanner Schmidt, Richard Newcombe, and Dieter Fox. DART: Dense Articulated Real-Time Tracking. In *Proceedings of Robotics: Science and Systems*, Berkeley, USA, July 2014.
- [sta] The Stanford 3D Scanning Repository. <https://graphics.stanford.edu/data/3Dscanrep/>.
- [vdWCV] S. van der Walt, S.C. Colbert, and G. Varoquaux. The NumPy Array: A Structure for Efficient Numerical Computation.
- [vL07] U. von Luxburg. A Tutorial on Spectral Clustering. *Statistics and Computing*, 17(4):395–416, December 2007.
- [VTS04] Jean-Philippe Vert, Koji Tsuda, and Bernhard Schölkopf. A primer on kernel methods. *Kernel Methods in Computational Biology*, pages 35–70, 2004.
- [VWA⁺08] N. Vahrenkamp, S. Wieland, P. Azad, D. Gonzalez, T. Asfour, and R. Dillmann. Visual servoing for humanoid grasping and manipulation tasks. In *Humanoid Robots, 2008. Humanoids 2008. 8th IEEE-RAS International Conference on*, pages 406–412, Dec 2008.
- [Xav97] P.G. Xavier. Fast swept-volume distance for robust collision detection. In *Robotics and Automation, 1997. Proceedings., 1997 IEEE International Conference on*, volume 2, pages 1162–1169 vol.2, Apr 1997.
- [ZKM07] Liangjun Zhang, Young J Kim, and Dinesh Manocha. C-DIST: Efficient distance computation for rigid and articulated models in configuration space. In *Proceedings of the 2007 ACM symposium on Solid and physical modeling*, pages 159–169. ACM, 2007.