Knowledge Extraction and Learning of Significant Robot Task Characteristics from Episodic Memories

Diplomarbeit

Tim Okrongli

Prüfer der Diplomarbeit:	1. Michael Beetz
	2. *fehlt noch*
Supervisor	1. Michael Beetz
	2. Jan Winkler

Eidesstattliche Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig angefertigt, nicht anderweitig zu Prüfungszwecken vorgelegt und keine anderen als die angegebenen Hilfsmittel verwendet habe. Sämtliche wissentlich verwendete Textausschnitte, Zitate oder Inhalte anderer Verfasser wurden ausdrücklich als solche gekennzeichnet.

Bremen, den 1. Mai 2015

_____

Tim Okrongli

Acknowledgements

I would like to thank Prof. Michael Beetz for responding to my vague idea for a topic with one that has actual practical relevance. Furthermore, I would like to thank him and *fehlt noch* for actually reading this thesis.

I would like to thank Dipl.-Inf. Jan Winkler for his invaluable support during my work on this thesis and for familiarizing me with the working group's infrastructure. I would also like to thank PERSON for helping me integrate my code with this infrastructure.

Further thanks go to Dr. des. des. Dominik Wondrousch; Christian Käser, B.Sc. and Björn Mecklenbrauck for moral and logical support, their skill at being sounding boards and for proof-reading this thesis, a duty they shared with Robin Okrongli, whom I also thank at this point.

Lastly, I would like to extend my thanks to my family for supporting me on my not always entirely goal-oriented path towards a vintage university degree. Rocky though it may have been, at last it finally came to an end.

Abstract

Robots are becoming increasingly capable of performing complex tasks in environments they have imperfect knowledge of, whether due to uncertainty about the environment itself or due to the environment changing independently if the robot's actions. Frameworks such as CRAM[1] enable robots to learn and reason about their environment and adapt their actions accordingly. However, planning actions from scratch requires significant time and processing power and leads to unsatisfactory performance in the real world if done constantly.

This thesis will explore means of examining a robot's memories to derive shortcuts which can be used to avoid or minimize the use of costly replanning steps and thus achieve greater performance when carrying out actions. One such attempt, which will be covered in detail, is the learning of stereotypical motions through trajectory clustering and averaging to obviate the need to run a motion planner every time a task similar to the learned one is performed. As will be shown, this can appreciably reduce the planning time needed.

Also covered will be the integration of the code written to preform stereotypical motion learning with an existing robot programming environment.

This thesis will also touch upon the topic of clustering trajectories in three dimensions. While trajectory-clustering is well-understood in two dimensions, with several specialized clustering algorithms having been developed, three-dimensional trajectory clustering is not a popular research subject and lacks specialized algorithms. I will show that the combination of a generic clustering algorithm with Dynamic time warping delivers acceptable results with real-world data.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

1.1. Motivation

Modern robot cognition and learning systems like RoLL[2] and CRAM[1] allow robots to handle everyday tasks in dynamic environments. However, the robot might spend a significant amount of time calculationg plans and correcting for errors during these tasks as a result of uncertainty about the environment and a lack of solid foundational assumptions about how to engage these tasks.

Humans, on the other hand, have little trouble performing tasks in the face of changing environments and rapidly become more proficient at them. For instance, they memorize where objects are likely to be located or how to reach them. In addition, muscle memory allows humans to perform stereotypical motions, such as grasping an object, without having to consciously think about them. **TODO**: Citations for this?

Through analysis of memories of previous experiments it should be possible to identify interesting properties of these experiments that are not apparent when examining each experiment individually. For instance, commonalities between successful attempts to grasp an object might be identified and translated into base parameters for future grasp attempts as an approximation of human muscle memory. Ideally, this will allow the robot to greatly reduce the use of a motion planner for commonly encountered tasks.

1.2. Scenario

Consider the following scenario: A robot operating in a changing environment has the task to locate and retrieve an object sitting on a table. The robot has some basic knowledge of where the object is (ie. which table it is on) and moves towards the table. The robot does not, however, know where exactly on the table the object is. Each time the robot goes to retrieve the object, it will be in a different position relative to the object.

Picking up the object requires the robot to make and execute an appropriate motion plan, which is costly. Due to the changing relative positions this plan cannot simply be recorded and played back. Additionally, such a recorded plan might be very specific to one grasping situation (such as having to avoid other objects on the table) and might be inapplicable to the current situation.

This is where my thesis comes in: By learning about commonalities between different grasping attempts the robot can make better decisions or even bypass costly operations like motion planning partially or entirely.

1.3. Technical Challenges

One challenge lies in the data itself as it may not neccessarily be in a form easily suited for reasoning. For instance, for much of the development of my code I worked with motion data stored as a series of logged CRAM messages, each containing updates to the positional data for some of the robot's components. In order to analyze the motion data I had to reconstruct a model of the robot and track it over time.

Later I progressed towards integrating the relevant parts of my code with a live system, which obviated the need for a complex log parser (as the system can directly be queried for things like logged trajectories) but still required translation between the system's representation of trajectories and that used in my code.

Likewise, the analysis itself is nontrivial as complex motions must be represented in a way that allows analysis methods like clustering to be performed. As will be shown in chapter 2, three-dimensional trajectory clustering has not been thoroughly researched at the moment and still presents some unsolved challenges.

1.4. Basic Approach

The approach taken in this thesis is to examine the logs generated by the CRAM system for groups of similar experiments. CRAM logs all actions taken by the robot in great detail, including the position and heading of the robot, states of individual actuators, time taken for various internal pocesses and outcomes of each step.

Based on this information, stereotypical motions are derived, which can be used to bypass the motion planner and thus save time. (Other ways of learning from logged data will be touched upon but will not be the main focus of this thesis.)

Chapter 2

Technical Foundation

2.1. Choice of Programming Language

All code was written in Java and most of it is compatible with version 1.6 ot the Java Runtime Environment. This was done to make it easier to integrate with the working group's systems.

Some parts were written in Java 1.7 for convenience's sake, but those parts are only intended for local testing use.

2.2. Robot Operating System, Cognitive Robot Abstract Machine, CRAMm

This thesis makes use of the infrastructure provided by several other components.

Robot Operating System (ROS)[3] is a robotics programming framework that provides a variety of useful components, probably the most important of which is inter-process communication via message passing. Messages can be recorded and played back at a later time, which of course is one of the foundations my thesis builds on. In addition, ROS provides the format of the logs I analyze in section 3.4.[4]

Cognitive Robot Abstract Machine (CRAM)[1] is a cognitive reasoning framework for robots built on top of ROS and the *KnowRob*[5] knowledge processing system.

CRAMm[6] is an extension of CRAM that concerns itself with efficient management of robot memories. It provides the "live" data access that supplanted log file parsing during the later stages of development and that is expected to be the main form of interaction between my code and the rest of the system during its productive lifetime.

2.3. Java-ML

The **Java Machine Learning Library (Java-ML)**[7] is a Java library that provides a number of machine learning algorithms as well as common data classes and interfaces to tie them together. It was used to provide some infrastructure, such as clustering algorithms and data classes for trajectories. Changes had to be made to Java-ML to make it usable for the purposes of this thesis; see section 3.3.

2.4. Trajectory Clustering

This thesis puts a heavy emphasis on finding stereotypical motion paths by means of clustering and averaging previously observed motion paths. This necessitates a means of clustering three-dimensional trajectories.

Unfortunately, this is not a particularly well-researched subject yet. Specialized trajectory clustering algorithms such as TRACLUS[8] and its derivatives or vector field k-means[9] are only defined in two dimensions with a three-dimensional adaptation being left to the reader. Since such an adaptation would be nontrivial and lies beyond the scope of this thesis I decided against attempting to perform it.

A likely reason for this preponderance of exclusively two-dimensional approaches among trajectory clustering algorithms is that both use cases and matching data sets are easy to come by in the two-dimensional area; commonly-seen examples are traffic motion profiles and motion prediction for meteorological phenomena[8][9][10].

Other algorithms, like NETSCAN[10] or subtrajectory pattern detection[11], are not applicable to free-form motion paths or do not return the kind of cluster needed in this thesis.

Since no applicable specialized clustering algorithms were available it was decided to use a generic clustering algorithm combined with Dynamic time warping as a dimensionality-agnostic distance measure.

2.5. Dynamic Time Warping

Dynamic time warping[12] (DTW) is an algorithm for aligning two temporal sequences that can also be used as a distance measure. It is tolerant towards sequences being out of phase or having differing speeds or numbers of data points. This makes it useful as a distance measure for clustering real-world trajectories as minor differences like sampling differences do not have a large impact on the calculated distance. Additionally, it handles multidimensional data points well, which makes it attractive for the purposes of this thesis.

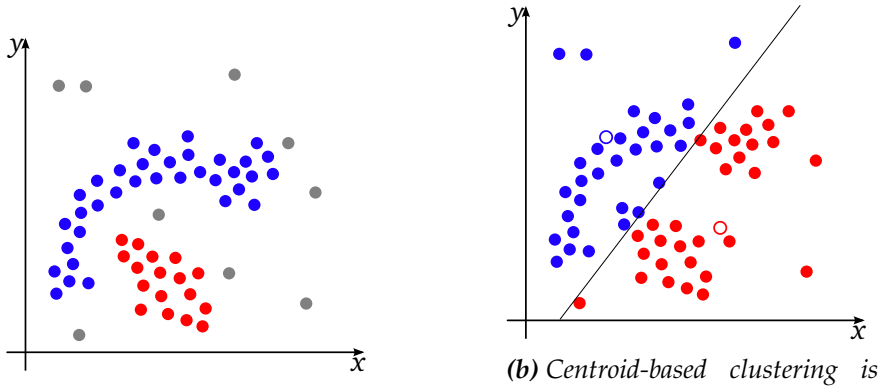
The DTW implementation used is an implementation of FastDTW[13] included in Java-ML. FastDTW is a highly accurate approximate DTW algorithm with linear time and space complexity.

2.6. Density-Based Clustering

The clustering algorithm employed is a form of *density-based clustering*; more precisely, an adapted version of Java-ML's `DensityBasedSpatialClustering` class (see 3.3.2 for further details), which is an implementation of the DBSCAN[14] algorithm, was used.

Density-based clustering is characterized by its definition of clusters as areas of comparatively high data point density. Data points outside these areas are considered to be either noise (and thus unclustered) or outliers of a cluster, at the algorithm's discretion. One of its core strengths lies in being able to handle clusters with complex shapes, such as concave clusters, as illustrated in figure 2.1a.

Another approach towards clustering is *centroid-based clustering*, exemplified by the k-means family of algorithms WHO DO I CITE?, where clusters are defined by proximity to a central vector. This partitions the entire data set into a Voronoi diagram. While k-means clustering is a very popular approach it has problems dealing with clusters that can't be linearly separated; see figure 2.1b. It also has no concept of noise data.



(a) *Density-based clustering can find clusters with complex shapes (Noise shown in gray)*

(b) *Centroid-based clustering is not suitable for all cluster shapes. (Centroids shown as unfilled circles; border between Voronoi cells shown as thin line)*

Figure 2.1: *Comparison of density-based clusters and Centroid-based clustering*

Other approaches would be *hierarchical clustering*, which has similar strengths to density-based clustering, but is often rather slow (with complexities of $\mathcal{O}(n^2)$ or worse[1]), or *distribution-*

[1] SLINK[15]/CLINK[16]

based clustering, which requires certain assumptions about the distribution of the data points to be met in order to be effective.

DBSCAN was chosen because it has a number of positive qualities: Its low complexity of $\mathcal{O}(n \log n)$ allows for fast clustering and while the algorithm is non-deterministic, this will only affect a small number of so-called *border points*. In addition, unlike centroid-based algorithms, density-based clustering algorithms do not require *a priori* knowledge of the number of clusters or expensive iterative schemes to detect the number of clusters. In addition, it was readily available as part of Java-ML.

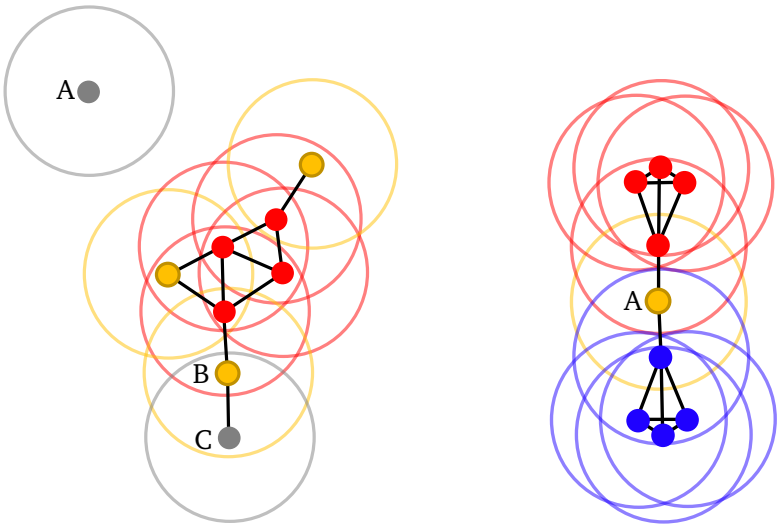
DBSCAN takes two parameters: A distance ϵ and a number $minPts$. All data points are then evaluated and classified as follows:

- A point p is considered a *core point* is at least $minPts$ are within a distance of ϵ; those other points are considered *directly reachable* from p. (A point q is considered reachable from p if there is a)
- A point that is not a core point but is directly reachable from one is considered a *border point*.
- A point p is considered *reachable* from a point q if there is a path $p_1, ..., p_n$ with $p_1 = p$ and $p_n = q$ where each p_{i+1} is directly reachable from p_i.
- A point that is not directly reachable from a core point is considered noise.

A cluster consists of all points reachable from any of its core points, that is the set of its core and border points.

Example: Figure 2.2a shows the result of a DBSCAN run with $minPts = 3$. Data points are shown as solid circles; their respective ϵ is shown as a surrounding unfilled circle and reachability is shown through lines between points. The red points are core points as they are each reachable three other points. The yellow points are border points as they are only reachable from one or two other points. Point **A** is reachable from no other points and is thus noise. Point **C** is reachable from point **B** but since point **B** is not a core point, point **C** is also considered noise.

DBSCAN is non-deterministic (the starting points for cluster generation are chosen at random), however the rules ensure mostly consistent results. Figure 2.2b shows how results can be non-deterministic: Depending on whether the red or the blue cluster are generated first, point **A** could belong to either (but not both) of them. Of course another non-deterministic property is the order in which clusters are detected so two consecutive runs of DBSCAN on the same data with the same parameters will generally result in clusters with the same shapes (modulo border overlaps as in Figure 2.2b) but not neccessarily in the same order.



(a) *DBSCAN results* ***(b)*** *Non-determinism in DBSCAN*

Figure 2.2: *DBSCAN*

DBSCAN has given adequate results with the data it was tested with (see **??**). This may not hold for different input data, however, and a more sophisticated clustering algorithm like OPTICS[17] or EnDBSCAN[18] might give better results in those cases.

2.7. Averaging Trajectories

In order to average trajectories, the fact that they have data points at different time points relative to their respective start must be compensated for. (See figure 2.3a, which shows two one-dimensional time series with data points at different timepoints.)

First, the timepoints at which not all trajectories have a data point are determined (figure 2.3b). For each of these timepoints new data points are synthesized where neccessary by interpolating between the closest data points (figure 2.3c)[2]. Then the values of all data points for each time point are averaged to determine the data points for the new trajectory (figure 2.3d).

While the illustrations only show this for two trajectories, in practice this is done with all trajectories of a cluster at the same time.

Note that all of this operates under the implicit assumption that linear interpolation between data points on the trajectory will not result in significant loss of accuracy. This is true under two conditions: Either if the observed part of the robot is known to make only linear motions between data points or if the data points are sufficiently dense that the difference between a linear and a curved path between them is irrelevant. The latter is true for the data used in this thesis.

2.7.1. Trajectory stretching

In figure 2.3c a new data point is extrapolated beyond the last one by simply copying the last point's x. This is fairly fast but may lead to sub-optimal trajectories if the source trajectories are sufficiently dissimilar in length. A more universally applicable, although more resource-intensive approach would be to stretch all trajectories to the length of the longest one; see figure 2.4.

When stretching a trajectory the data points themselves remain unchanged; only the time points they correspond to are moved. This is done by multiplying the offset between the time point and the first time point in the trajectory with the quotient of the length of the longest trajectory and that of this one:

$$t_{new} = t_{first} + (t_{old} - t_{first}) * \frac{length(longest)}{length(this)}$$

...with $length()$ defined as the difference between the last and the first time point in a trajectory:

[2] Simple linear interpolation is used here, which is deemed acceptable due to the high frequency at which data points are generated. If this approach is used with a sparse data set a more elaborate form of interpolation might be more appropriate.

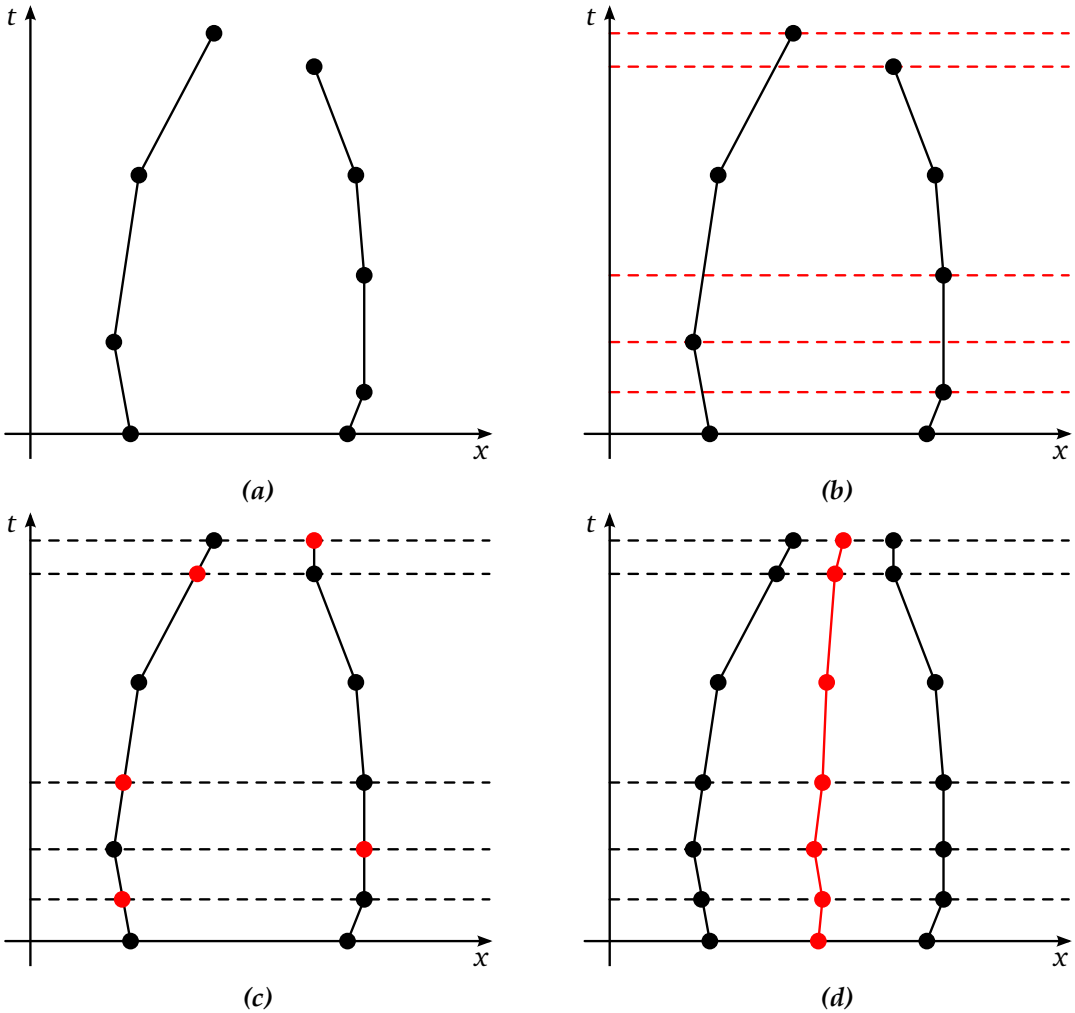


Figure 2.3: *Constructing an averaged trajectory*

$length(x) = t_{last} - t_{first}$

After stretching all trajectories the averaging proceeds as normal.

The choice of scaling direction is an arbitrary one; in principle one could as well normalize towards the shortest length or the mean of all lengths. However, this assumes that when the trajectory is later followed, temporal information is discarded. If the robot tries to match the trajectory in speed as well as in shape, shortening a trajectory might lead to the robot attempting to execute motions at speeds it is not actually capable of supporting.

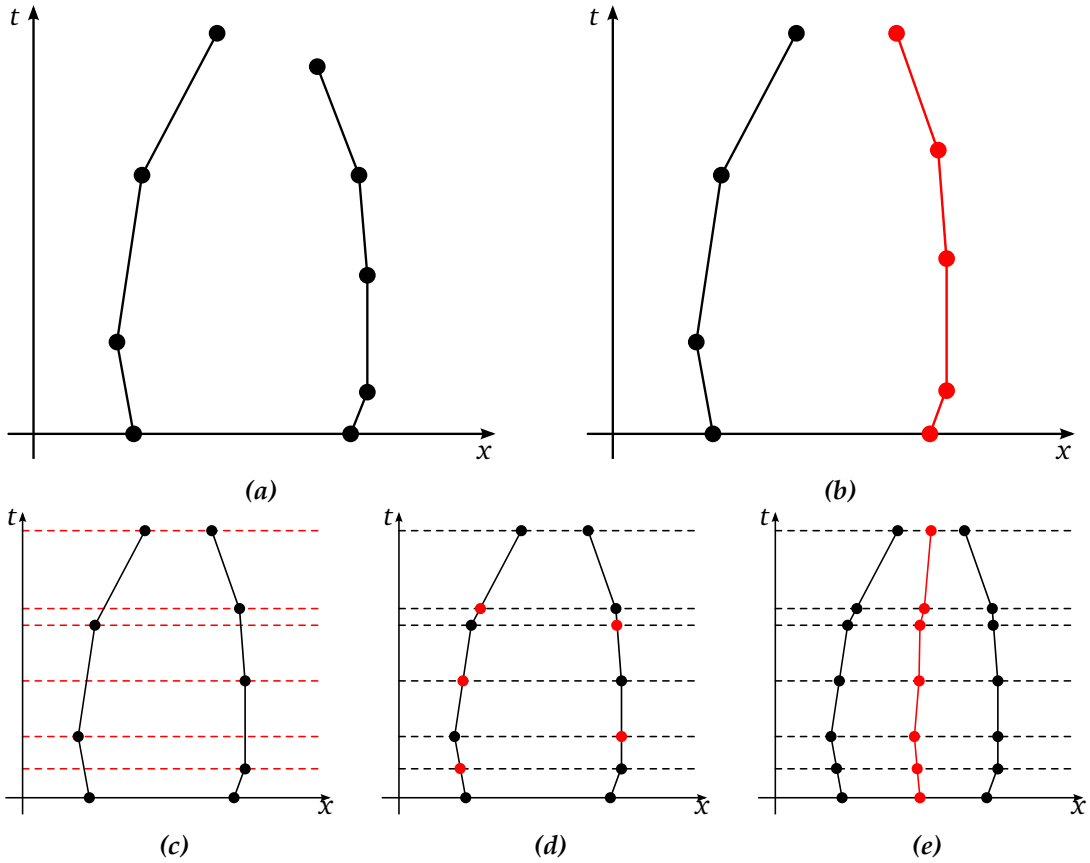


Figure 2.4: *Constructing an averaged trajectory with scaling*

2.8. Summary

This thesis concerns itself with the task of clustering three-dimensional trajectories, which it intends to achieve through a combination of a general-purpose clustering algorithm (DBSCAN) with a distance measure that can robustly handle highly complex data points (Dynamic time warping). Additionally, it touches on deriving averaged trajectories from clusters. The data source is either ROS or CRAMm.

Chapter 3

Approach

3.1. Overview

The approach for implementing the ideas outlined above is split into two major parts: Technical groundwork and actual implementation.

3.2. General Considerations

ROS trajectories are generally series of seven-dimensional time points. Three dimensions are the positional information (translation along the X, Y and Z axis), three are the rotational information (yaw, pitch, roll) and one is the temporal information (timestamp relative to an arbitrary time point). In practice, the temporal component is irrelevant for the most important step, the clustering, due to the way Dynamic time warping works, making the data effectively six-dimensional.

It was decided to only consider the positional component of the robot's motions and discard the rotational component once a trajectory has been generated. This is sufficient for the kind of optimization this thesis mainly explores (reducing planner use during grasping motions by getting the gripper close to its intended position using a stereotypical trajectory; see section 4.1). Even if motion planning has to be performed during the execution of the trajectory it is reduced to determining rotation, dropping it from six dimensions to three, which brings considerable resource savings. (The temporal information is not directly involved in motion planning.)

Simultaneously, the trajectories themselves only have three relevant dimensions instead of six, making the clustering more effective. (The so-called "curse of dimensionality"[19] makes many algorithms, such as most general-purpose clustering algorithms, increasingly ineffective as the number of dimensions in the data increases.)

Additionally, the intended final result of a successful use of a stereotypical trajectory is that the gripper is in close physical proximity to the object, not that the entire gripping motion has been completed. A small motion planning component will remain but the cost of it is expected to be minor compared to that of getting the gripper close to the object in the first place.

3.3. Adapting Java-ML

While Java-ML offers clustering algorithms, an implementation of FastDTW and classes to represent time series (namely the `TimeSeries` class), it does not offer a means of combining all of this, most likely because the developers have not implemented that yet. Since the combination of these two functionalities is crucial to this thesis it was decided to extend Java-ML.

Due to design choices made by Java-ML's authors (such as liberal use of the `private` keyword) there is generally no meaningful way to extend existing classes by means of subclassing them or writing a wrapper class. An attempt to fork and rewrite the library to natively support its `TimeSeries` class as clustering input failed as certain assumptions about the nature of the data the other classes work with don't hold for `TimeSeries` and changing those assumptions would require an extensive rewrite of much of the library.

As such, a smaller fork was made to make it easier to extend Java-ML by introducing my own classes that implemented similar functionality (by means of copying and pasting code from Java-ML and making changes as appropriate) and sometimes inheriting from Java-ML classes. The final version of my Java-ML fork was thus created by downloading the current version of the source code from the project's repository[1] and making the following changes:

- Replacing all instances of `private` with `protected` except where doing so would cause errors (ie. inside of enumerations)
- Removing the `final` keyword from all classes and methods
- Changing package-private constructors to `protected` ones where encountered

This made it possible to extend the library with a relatively small amount of reimplementation.

[1] `http://sourceforge.net/p/java-ml/java-ml-code/ci/master/tree/`

3.3.1. Annotated Time Series

Java-ML's `TimeSeries` class (which represents multidimensional data that changes over time) does not offer a means to link the series to another object. Since that was deemed to be useful the `AnnotatedTimeSeries<T>` class was intriduced. This class extends the existing `TimeSeries` by linking each `TimeSeriesPoint` added to it to an object of type `T`. This makes it possible to link each trajectory to metadata such as its source experiment or whether the task had been completed successfully.

3.3.2. Clustering Annotated Time Series

In order to cluster annotates time series several Java-ML classes involved in clustering had to be duplicated.

The reason for this is that, as already mentioned above, there is no way to cluster a `TimeSeries` with the regular clusterers as the clustering interfaces do not support this. Even though Java-ML's FastDTW implementation offers a ready-to-use distance measure, the design of the interfaces makes it impossible to actually use with the clusterers. (While the distance measure can be used without issue, the clusterers will only accept sets of single data points, not sets of series of data points.)

This problem was solved by duplicating part of Java-ML's clustering infrastructure and changing the copied classes and interfaces to work on `AnnotatedTimeSeries` objects. In the process, support for variable `DistanceMeasure` objects was dropped – the only distance measure that supports `TimeSeries` or `AnnotatedTimeSeries` is DTW, which has been hardcoded into the copied classes.

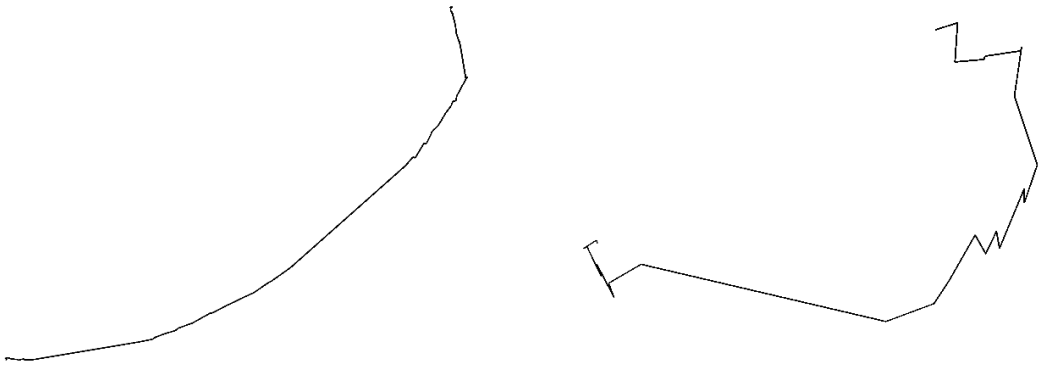
The only clustering algorithms reimplemented in this fashion are `DensityBasedSpatialClustering` (a DBSCAN implementation) and `KMedoids` (a implementation of k-medoids clustering, a variant of k-means clustering).

3.3.3. Generating trajectories for testing

The functionality of the adapted clustering classes was first tested with "bundles" of synthetic trajectories, which gave poor results but allowed for faster development. These will not be explained in detail due to them only being useful to determine code functionality (which can be done with any group of similar trajectories), but to give a rough idea, they were generated by plotting functions in the form of $f(x) = ax^2 + bx + c$ with the trajectories in each "bundle" sharing similar values for a, b and c.

Later, real trajectories were manually generated for testing by physically visiting the robot, enabling its logging infrastructure and then moving its right arm into various positions, mark-

ing the start and end of a "grasping motion" in the system once the arm was in an appropriate position. This produced realistic data for a variety of scenarios without taking much time. The generated trajectories had considerable amounts of jitter compared to "regular" trajectories (see Figure 3.1 for a comparison with a trajectory from a "real" experiment), which fortunately did not seem to affect clustering performance.



(a) *From "real" experiment* ***(b)*** *Hand-made*

Figure 3.1: *Shape comparison between a trajectory from an earlier experiment and a hand-made one. Both trajectories roughy describe an arc.*

3.4. Interpreting Offline Robot Logs

While CRAM offers a way of directly supplying trajectory data by means of the ROS infrastructure, ready access to this infrastructure was not initially available during development and thus a means of working with an offline copy of CRAM activity logs was implemented.

3.4.1. The CRAM Log Format

A CRAM activity log for an experiment consists of several files. The ones most important for this thesis are as follows:

3.4.1.1. cram_log.owl

This file contains a KnowRob-provided OWL[2] ontology describing tasks, subtasks and associated actions performed during the experiment. A simplified example entry might look like this:

[2]Web Ontology Language; see http://www.w3.org/TR/owl2-overview/ and http://www.w3.org/TR/owl2-primer/

```
<owl:namedIndividual rdf:about="&log;CRAMAction_ULaU2YMj">
  <rdf:type rdf:resource="&knowrob;CRAMAction"/>
  <knowrob:taskContext rdf:datatype="&xsd;string">GRASP</knowrob:taskContext>
  <knowrob:startTime rdf:resource="&log;timepoint_1409141798"/>
  <knowrob:endTime rdf:resource="&log;timepoint_1409141798"/>
  <knowrob:subAction rdf:resource="&log;WithFailureHandling_UAkMxifF"/>
  <knowrob:subAction rdf:resource="&log;WithFailureHandling_KE9I0mR9"/>
  <knowrob:objectActedOn rdf:resource="&log;designator_OB3TzSu1Zw1aYq"/>
</owl:namedIndividual>
```

The entities `&log;`, `&knowrob;` and `&xsd;` are shorthands for the first parts of identifier URIs that never change and thus don't require repetition. For instance, `&log;` resolves to `http://ias.cs.tum.edu/kb/cram_log.owl#`. For ease of reading, relevant parts of the entry have been color-coded.

This entry describes the "named individual" `&log;CRAMAction_ULaU2YMj`, which is a `&knowrob;CRAMAction` – an action within a `GRASP` task. It starts at `&log;timepoint_1409141798` (the Unix timestamp 1409141798, which corresponds to 2014-08-27 12:16:38 UTC) and ends at the same timepoint – thus it represents an action that took less than a second to perform. The subactions `&log;WithFailureHandling_UAkMxifF` and `&log;WithFailureHandling_KE9I0mR9` were taken, each of which has an entry comparable to this one. Finally, the object acted on, `&log;designator_OB3TzSu1Zw1aYq`, is given.

3.4.1.2. tf.json

This file contains a list of messages from ROS's `tf` module, which concerns itself with the robot's motions. Each line of the file is its own JSON document representing one such message. Each message contains one or more transforms describing how a part of the robot (or the surrounding room) is transformed (translated and rotated) relative to its parent part. A message with a single transform looks like this (split into multiple lines for readability):

```
{
    "_id" : {
        "$oid" : "53fdcbaf5ae4e112967afdc1"
    },
    "transforms" : [
        {
            "header" : {
                "seq" : 0,
                "stamp" : {
                    "$date" : 1409141679840
                },
                "frame_id" : "/base_link"
            },
            "child_frame_id" : "/base_bellow_link",
            "transform" : {
                "translation" : {
                    "x" : -0.29,
                    "y" : 0,
                    "z" : 0.8
                },
                "rotation" : {
                    "x" : 0,
                    "y" : 0,
                    "z" : 0,
                    "w" : 1
                }
            }
        }
    ],
    "__recorded" : {
        "$date" : 1409141679000
    },
    "__topic" : "/tf"
}
```

The transform concerns a change that occurred on 1409141679840 – the Unix timestamp 1409141679.840 or 2014-08-27 12:14:39.840. (Note that the `tf` module's timestamps use milliseconds instead of the seconds used in the OWL document.) The parent body part (the body part relative to which the transform is) is `/base_link` and the child part (the transformed one) is `/base_bellow_link`. The translation is given as $X/Y/Z$ offsets while the rotation is given as a rotation quaternion.

3.4.2. Data Extraction

In order to obtain data ready for further processing, the state of the robot at each known point in time during a task must first be reconstructed and a representative trajectory (of a certain body part) for the task must be calculated.

In order to reconstruct the robot's state for a certain task, first the beginning and end of that task must be determined. All messages that were logged during these points in time describe potantially relevant changes to the robot's state (or that of the surrounding area). However, merely considering all messages during the task is not enough: As each message may only contain partial information on the experiment's state it is neccessary to gather additional messages to obtain a complete picture.

In practice it is sufficient to additionally consider messages logged up to one second before each task starts as CRAM will log the experiment's complete current status once a second.

Once the start and end timestamps for the task have been determined, state generation can begin. In order to do this, first the messages during the task's time window are examined and discarded until one is encountered that contains all of the parts making up the robot. This is guaranteed to happen before the task's actual start date because of the state dump that happens every second. The first complete message becomes the first working state.

From there, each successive message is parsed and a new working state is generated by copying the last one and applying the changes from the message. After this is done (ie. when the task's end timestamp has been passed) a list of states has been generated, each of which describes the robot in its entirety at each moment something was logged. See Figure 3.2 for an illustration of several such reconstructed states.

Generating trajectories from there is not a difficult task: A relevant part of the robot is chosen (for instance `/r_wrist_roll_link`, the right arm's wrist, which for this robot is often used to represent the position of the robot's "hand") and its location at each point in time is determined by summing up the part's translation with those of each of its parent parts up until the root element. These locations are then used as three-dimensional data points for an `AnnotatedTimeSeries<Task>` object.

The actual algorithm used is shown in subsection A.2.1.

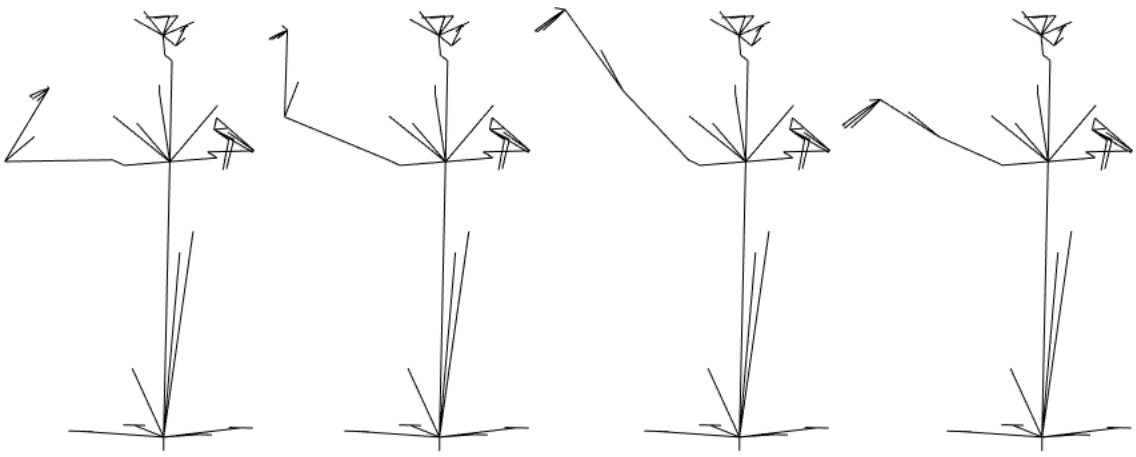


Figure 3.2: *Reconstructed robot states during an arm motion*

For the time being, only the translation component of the individual states is put into the trajectory as per section 3.2.

3.4.3. Dealing With Different Start And End Points

When the system is queried for a trajectory, the provided start and end points do not neccessarily match any known trajectory. In fact, it's virtually guaranteed that no trajectory provides a perfect match. Two steps must be taken to deal with this. (Note that at this point it is assumed that the robot already knows which body part to perform the action with and that appropriate steps have been taken to ensure that the object is actually within range of that body part.)

Firstly, the distance of the given points (Δ_{given}) can be used as an initial clue to determining which trajectories are applicable: If Δ_{given} is similar to the distance between the start and end points of a known trajectory (Δ_{stored}) the trajectory can be assumed to be roughly similar and thus worth further consideration. In other words, all trajectories where $|\Delta_{given} - \Delta_{stored}| \leq \epsilon$ will be considered, where ϵ has to be chosen appropriately.

If each stored trajectory is annotated with the distance of its start and end points it is fast and easy to compare Δ_{given} and Δ_{stored}.

The second step is to select a trajectory from the remaining ones and transform it to lead form the given start point to the given end point. This is partially or entirely skipped if the respective points are close to theie recorded counterparts (with the definition of "close enough" being an implementation detail not covered in this thesis). Figure 3.3 shows the cases that have to be considered.

If the points match the recorded points (case I) no transformation is neccessary. If the start or end points differ (cases II and III) the start and end points of the trajectory have to be moved and the trajectory has to be adapted accordingly. If both points differ (case IV) both of these adjustments must be made.

In practice, cases I, II and III are just variants of case IV where one or both points are moved by a vector of $\begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$. As such, I will only consider case IV from here on since the other cases follow from it.

In order to transform a trajectory, first the transformations of the start and end points (T^{start} and T^{end}, respectively) are determined:

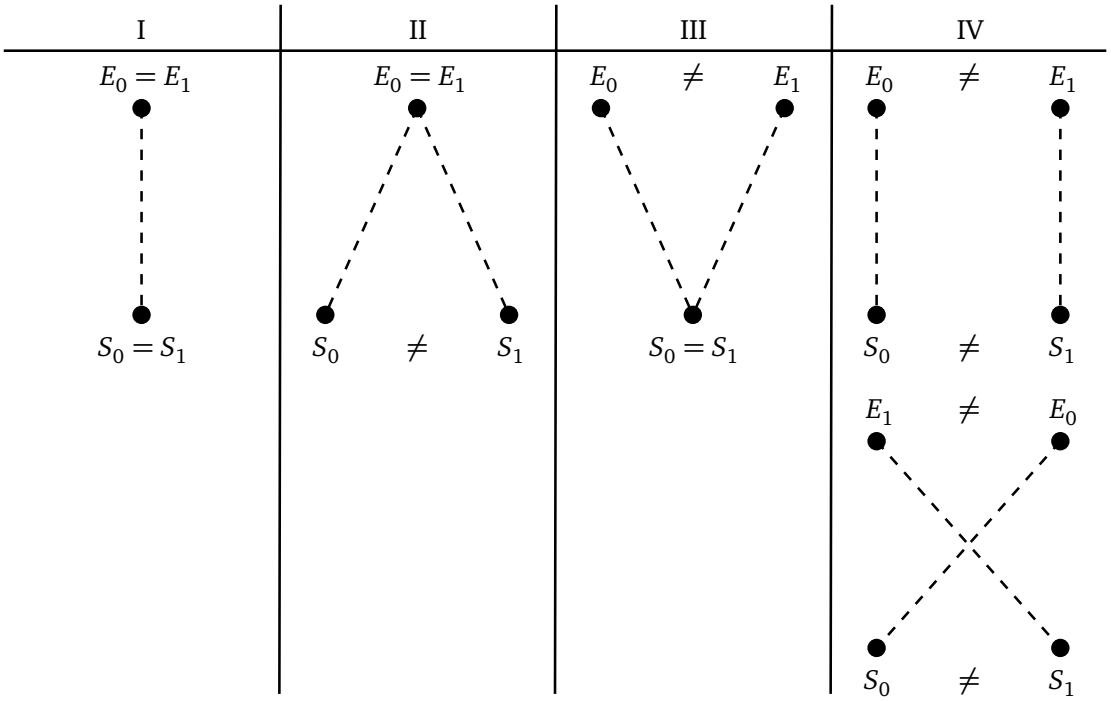


Figure 3.3: *Possible cases for start and end point deviation*

$$T = \left[\begin{array}{ccc|c} 1 & 0 & 0 & x_1 - x_0 \\ 0 & 1 & 0 & y_1 - y_0 \\ 0 & 0 & 1 & z_1 - z_0 \\ \hline 0 & 0 & 0 & 1 \end{array}\right]$$

Once T^{start} and T^{end} are known we iterate over each point on the trajectory and apply the transforms according to how much distance has already been covered.

In order to do this, we need two new related algorithms: One to determine the total length of the trajectory and one to determine how much of that length has been covered at any given point along it. In practice, these collapse to a generic "length up to a given point" algorithm. For further details, please see subsection A.2.2.

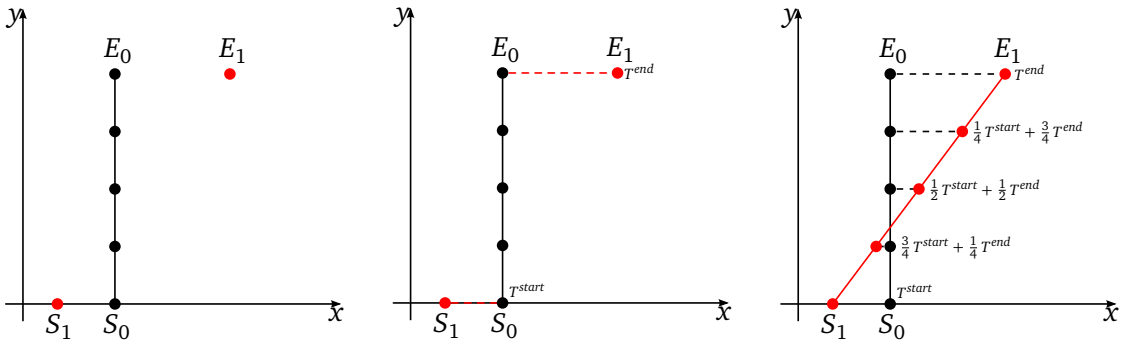
Once the progress along the trajectory at a given point P is known we can determine the transformation T^P for this point by adding the start and end transformations weighted according to the following formula:

$$T^P = progress(P) \cdot T^{end} + (1 - progress(P)) \cdot T^{start}$$

The closer we get towards the end the more $progress(P)$ approaches 1 and thus the more T^{end} dominates; vice versa for T^{start}.

After generating and applying the transformation for each point along the trajectory in this manner, the trajectory is fully transformed (see Figure 3.4 for an example).

This transformation approach has been implemented as part of the *AnnotatedTimeSeries* class.



(a) *Before transformation* ***(b)*** *Determining T^{start} and T^{end}* ***(c)*** *Applying interpolated transformations to all points*

Figure 3.4: *Transforming a trajectory to match new start and end points*

Chapter 4

Experiments

4.1. Gripping Motion Optimization

The main focus of this thesis will be to optimize gripping motions by avoiding (or at least simplifying) motion planning whenever possible. This will be achieved by analyzing past gripping motions and deriving common properties that can be used for future reasoning.

4.1.1. Determining Stereotypical Gripping Motions

A stereotypical motion is a motion that is commonly executed to reach a certain goal and that can be described as the smallest common denominator of such motions. This means that if the robot knows a stereotypical motion with parameters similar to those of the motion it is about to attempt, it can first attempt to execute that motion without having to engage in actual motion planning. This should save a lot of time, since motion planning is rather costly.

This experiment attempts to derive stereotypical gripping motions from past successful gripping attempts. The parameters used to determine whether a gripping motions corresponds to a past gripping motion are the position of the gripper and the position of the object, each relative to the robot's position.

Due to the way this approach operates, it is best suited for offline learning.

A number of trajectories corresponding to recorded successful gripping motions is obtained; see **??**. These trajectories are then clustered (see section 2.6) to determine commonalities. If one or more clusters are found, an average trajectory is generated (see **??**) and stored as the stereotypical trajectory for this cluster. Each such trajectory is annotated with the starting and ending position of the gripper.

Later, when the robot attempts to pick up something, it can consult a motion knowledgebase about known stereotypical trajectories for the kind of gripping motion it wants to perform. The parameters for the query are the current positions of the gripper and the object. The

knowledgebase then returns the trajectories it deems most applicable to the current situation. Applicability is determined based on similarities between the recorded and current starting and ending positions of the gripper and object.

Multiple trajectories may be returned if more than one trajectory with sufficiently similar paramters is known. This can happen if the robot often does similar gripping motions by using different paths, for instance in order to avoid an obstacle.

Before trajectories are returned, they are transformed to reach from the gripper's current position to the object's current position. This means that the robot only has to determine whether one of the trajectories can actually be followed (by trying to calculate a kinematic solution for it, which is reasonably fast) before being able to execute it. This should bring the gripper into relatively close proximity of the object, from where motion planning can do the rest in little time.

Note that not all points on a trajectory are neccessarily reachable. This is an artifact of both the averaging and transformation steps. Unreachable points have to be skipped via interpolation. If a sufficiently large number of points cannot be reached the trajectory may not hold any advantage over plain motion planning and has to be discarded. There is currently not enough data to give a good recommendation but 25% unreachable points might be a good starting point for further refinement.

4.1.2. Results of Trajectory Clustering

Running the hand-crafted trajectories through DBSCAN yielded good results with arbitrarily chosen values of $\epsilon = 3$ and $minPts = 3$, as seen in Figure 4.1a. In fact, it revealed more data than expected: The clusters shown were generated from three data sets intended to result in four clusters. The blue and green trajectories were intended to be one single group but due to inaccurate handling of the robot's arm ended up in two distinct clusters.

Changes to ϵ and $minPts$ did not improve the clustering result.

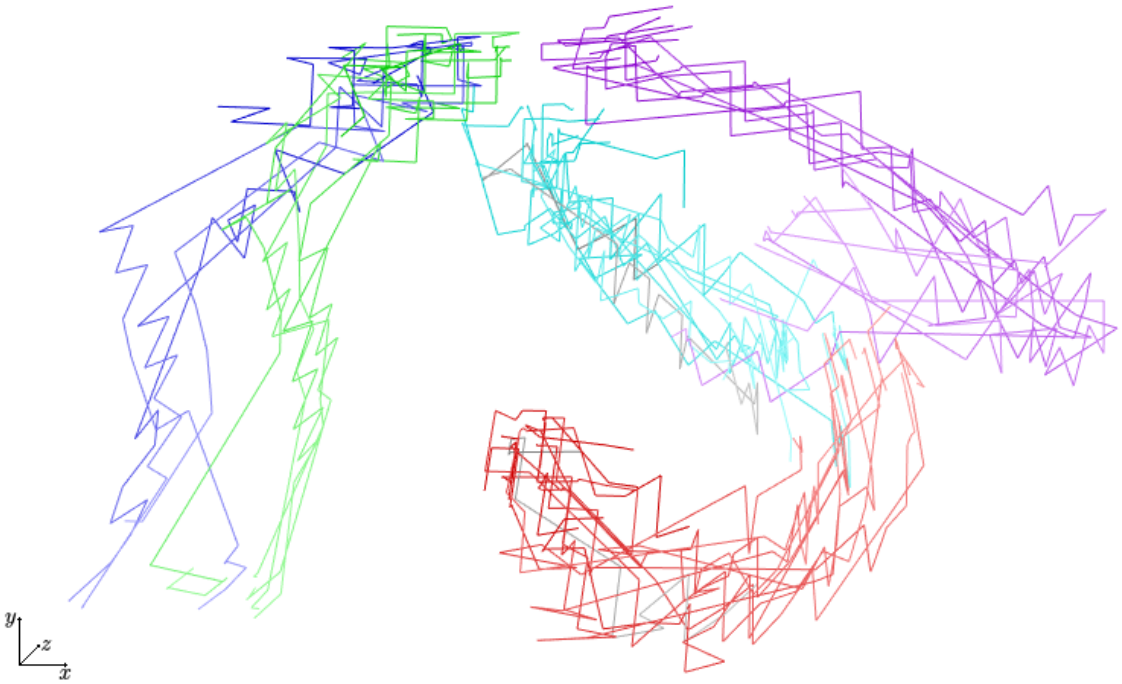
Figure 4.1b shows the averaged trajectories for these clusters. The difference in shape between the blue and cyan clusters becomes even more visible: Both have different start points and the blue one diverges to the left much more trongly than the cyan one. Also visible is how the averaged trajectories contain more points than the "raw" ones.

?? shows the difference that the trajectory stretching discussed in subsection 2.7.1 makes.

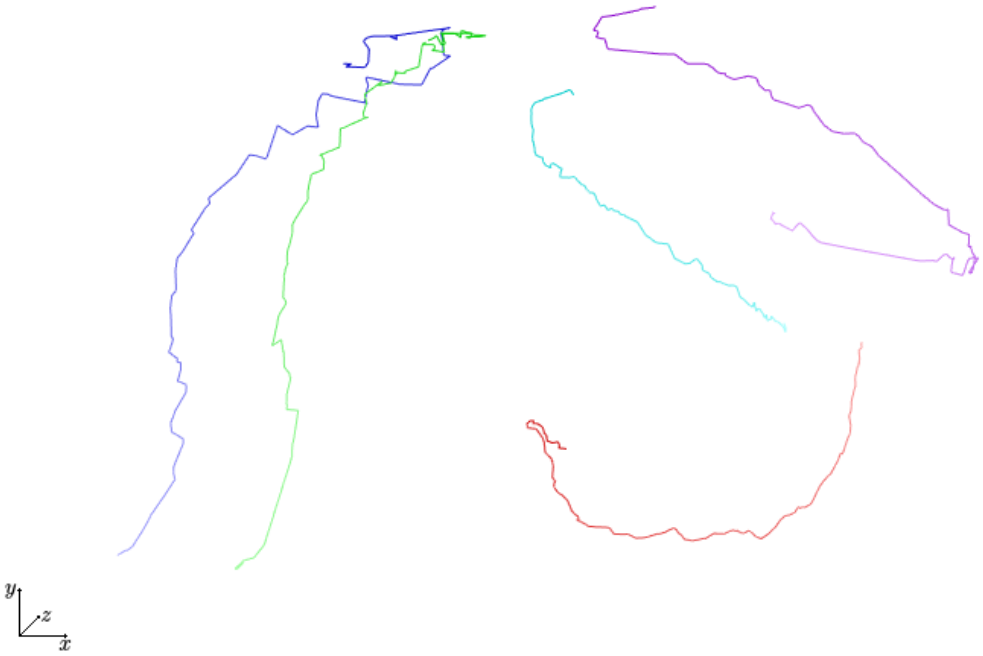
4.1.3. Performance Analysis

TODO: This needs to be written once all the data is there.

Executed: cyan trajectory (see Figure 4.1b). 238 stops total 167 stops reachable; 71 stops unreachable



(a) *Clusters generated from hand-made trajectories*



(b) *Averaged trajectories for these clusters (with stretching applied)*

Figure 4.1: *Clustering output for groups of hand-made trajectories. Each color identifies one cluster. Trajectories start with a light color and get darker towards the end. The red and cyan clusters are each accompanied by one gray trajectory; DBSCAN considered these trajectories to be noise.*

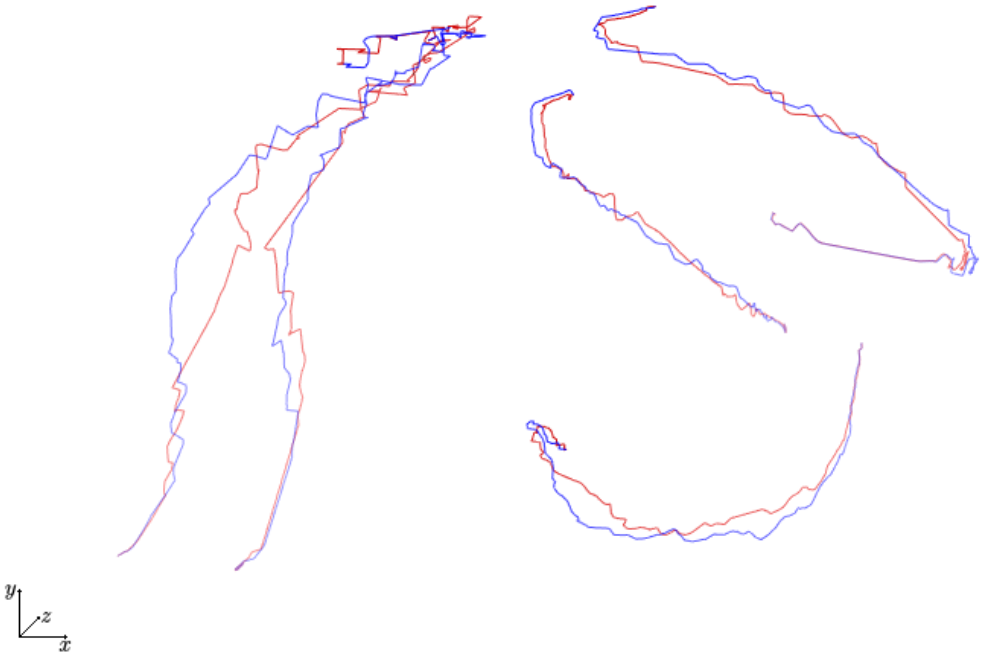


Figure 4.2: *Comparison between unstretched (red) and stretched (blue) averaged trajectories. Note how the leftmost trajectory describes a markedly different path after stretching while the rightmost remains virtually unchanged in the beginning.*

4.2. Further Experiments

While not the main focus of this thesis, there are other ways to use robot memory-based learning to improve efficiency. While these will not be examined in great detail they still deserve mention.

4.2.1. Common Properties of Successful and Unsuccessful Actions

While stereotypical motions allow the robot to save time on calculationg motion plans, there are more things to be learned from clusters of actions.

CRAMm logs contain informations on whether an action or subaction was successful or not (see subsubsection 3.4.1.1; pay particular attention to the names of the subactions). Additionally, they are extensible so success and failure can be directly annotated. (This information is, of course, also available from ROS.) Using this information it is possible to group actions based on whether they were successful (e.g. because an object was grasped) or unsuccessful (e.g. because the robot tipped over the object).

Once actions are grouped, they can be clustered according to a variety of data and the clusters can be analyzed accordingly. For instance, trajectories could be clustered and averaged, giving stereotypical bad motions, which could then be examined to try and see why this kind of trajectory leads to failure and how the robot can be taught not to generate this kind of trajectory in the future.

Another possibility would be to use metadata such as the relative position of the object to the robot (expressed as distance and angle) or even an object's rough shape to generate data points for clustering to identify scenarios in which the robot has a tendency to fail. This data could then be used to generate decision trees that allow the robot a better understanding of whether a certain kind of action is likely to succeed in a given situation.

4.2.2. Object Location Heuristics

Objects may not always be where the robot thinks they are. In a dynamic environment, other actors may move an object from where the robot remembers it. Finding the object can then require a lengthy search of the area or fail entirely. Examining the robot's memories may help in this situation.

The simplest case would be to examine where the object has been seen before, clustering those positions to weed out noise and determine areas with particularly high numbers of sightings and then checking those areas as a heuristic. Of course, care must be taken when recording

object positions in order to avoid counting the same position twice if the robot passes it twice without the object having been moved.

A more advanced approach might be to take note of other noteworthy objects in the vincinity and learn about common object constellations. This would allow the robot to learn that, for instance, a teacup it is looking for is commonly seen together with a kettle and a plate; it could then check its recent memories for incidences of kettles or plates and search those areas for teacups.

Chapter 5

Conclusion

5.1. Summary

In this thesis, ways of learning from robot memories in order to avoid or reduce work were explored. A particular focus was put on the learning of stereotypical motion paths from memories of commonly executed motions. An approach for clustering three-dimensional trajectoried was explored and was shown to deliver satisfying results for the given problem domain. In addition, real-world performance data was collected and it was shown that the approach described in this thesis allows for actual efficiency gains.

Other ways of learning from robot memories were briefly touched upon.

TODO: Have actual results to write about

5.2. Outlook

While this thesis examines one particular problem (making commonly repeated motions more efficient) it does expose a number of opportunities for further research.

Firstly, some work is left to the robot that could be integrated with a future version of the library discussed in this thesis. For instance, obstacles are currently not considered at any point. A possible refinement would be to annotate each source trajectory with the positions and dimensions of relevant obstacles. The obstacles could then be attempted to match prior to clustering and the averaging step could use the aggregated obstacle data of all invlved trajectories to annotate the generated trajectory with data on which areas the trajectoy avoids. Of course this topic requires more consideration than is possible at this point.

Additionally, no attempt is made to determine whether generated trajectories can actually be executed by the robot, with the currently accepted solution being to let the robot run a simulation. Perhaps this could be improved upon.

Trajectory-specific clustering methods might yield even better results than generic clustering algorithms paired with Dynamic time warping. Bringing one of the many powerful two-dimensional clustering algorithms into the third dimension seems like a promising further step (and, of course, holds value of its own).

Averaged trajectories are, by neccessity, more complex than the trajectories they were generated from and can contain many small movements that may not be strictly neccessary. Pairing stereotypical trajectory learning with an appropriate smoothing algorithm (for instance one based on B-splines like [20]) and an appropriate trajectory simplification algorithm (such as one of those shown in [21]) might produce simpler trajectories that could hold advantages both in storage efficiency and execution speed.

Glossary

DTW

Dynamic time warping

offline learning

a form of machine learning in which the dataset is considered static and learning is done by examining the entire dataset at once

Acronyms

CRAM	Cognitive Robot Abstract Machine
DTW	Dynamic time warping
Java-ML	Java Machine Learning Library
ROS	Robot Operating System

Bibliography

[1] Michael Beetz, Lorenz Mösenlechner, and Moritz Tenorth. Cram — a cognitive robot abstract machine for everyday manipulation in human environments. *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2010.

[2] Alexandra Kirsch. *Integration of Programming and Learning in a Control Language for Autonomous Robots Performing Everyday Activities*. PhD thesis, Technische Universität München, 2008.

[3] Open Source Robotics Foundation. Ros.org | powering the world's robots (http://www.ros.org/). [Online; accessed 2015-04-27].

[4] Open Source Robotics Foundation. Ros.org | core components (http://www.ros.org/core-components/). [Online; accessed 2015-04-27].

[5] KnowRob project. Knowrob: Knowledge processing for robots (http://www.knowrob.org/knowrob). [Online; accessed 2015-04-27].

[6] Jan Winkler, Moritz Tenorth, Asil K. Bozcuoğlu, and Michael Beetz. CRAMm — memories for robots performing everyday manipulation activities. In *Proceedings of the Second Annual Conference on Advances in Cognitive Systems*, 2013.

[7] T. Abeel, Y. V. de Peer, and Y Saeys. Java-ml: A machine learning library. *Journal of Machine Learning Research*, 10:931–934, 2009.

[8] Jae-Gil Lee, Jiawei Han, and Kyu-Young Whang. Trajectory clustering: A partition-and-group framework. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 593–604, 2007.

[9] Nivan Ferreira, James T. Klosowski, Carlos E. Scheidegger, and Cláudio T. Silva. Vector field k-means: Clustering trajectories by fitting multiple vector fields. *Computer Graphics Forum*, 32:201–210, June 2013.

[10] Ahmed Kharrat, Iulian Sandu Popa, Karine Zeitouni, and Sami Faiz. Clustering algorithm for network constraint trajectories. *Lecture Notes in Geoinformation and Cartography*, pages 631–647, 2008.

[11] Cynthia Sung, Dan Feldman, and Daniela Rus. Trajectory clustering for motion prediction. *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2012.

[12] Richard Bellman and R. Kalabra. On adaptive control processes. *IRE Transactions on Automatic Control*, 4:1–9, November 1959.

[13] Stan Salvador and Philip Chan. Toward accurate dynamic time warping in linear time and space. *Intell. Data Anal.*, 11(5):561–580, October 2007.

[14] Martin Ester, Hans peter Kriegel, Jörg S, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. pages 226–231. AAAI Press, 1996.

[15] R. Sibson. Slink: An optimally efficient algorithm for the single-link cluster method. *The Computer Journal*, 16(1):30–34, 1973.

[16] D. Defays. An efficient algorithm for a complete link method. *The Computer Journal*, 20(4):364–366, 1977.

[17] Mihael Ankerst, Markus M. Breunig, Hans-Peter Kriegel, and Jörg Sander. Optics: Ordering points to identify the clustering structure. pages 49–60. ACM Press, 1999.

[18] S. Roy and D.K. Bhattacharyya. An approach to find embedded clusters using density based techniques. In Goutam Chakraborty, editor, *Distributed Computing and Internet Technology*, volume 3816 of *Lecture Notes in Computer Science*, pages 523–535. Springer Berlin Heidelberg, 2005.

[19] Richard Bellman. *Dynamic Programming*. Princeton University Press, 1957.

[20] Jia Pan, Liangjun Zhang, and Dinesh Manocha. Collision-free and smooth trajectory computation in cluttered environments. *Int. J. Rob. Res.*, 31(10):1155–1175, September 2012.

[21] Yukun Chen, Kai Jiang, Yu Zheng, Chunping Li, and Nenghai Yu. Trajectory simplification method for location-based social networking services. In *Proceedings of the 2009 International Workshop on Location Based Social Networks*, LBSN '09, pages 33–40, New York, NY, USA, 2009. ACM.

Appendix A

Appendix

A.1. Libraries Used

A.1.1. Java-ML

An adapted version of Java-ML 0.1.7 was used to perform Dynamic Time Warping and clustering of trajectories; see section 3.3.

A.1.2. JSON-simple

The JSON.simple library (`https://code.google.com/p/json-simple/`), version 1.1.1, was used to facilitate the reading of the ROS transform log files.

A.1.3. LWJGL

LW**JGL** version 3 was used as an OpenGL wrapper for visualization purposes.

A.2. Algorithms

A.2.1. Determining a robot part's position

input : The body part to determine the position for (thePart), the robot's root part (root)
output: The part's position relative to the robot's root part coordinates
begin
    position = (0, 0, 0);
    parts = [thePart]; `// stack of body parts`
    currentPart = thePart;
    while currentPart *has a parent* **AND** currentPart *is not* root **do**
        parts.push(currentPart);
        currentPart = thePart.parent;
    end
    `// parts now contains all parts from thePart to root in order`
    while parts *is not empty* **do**
        currentPart = parts.pop();
        partPosition = currentPart.position;
        if currentPart *has a parent* **then**
            partPosition = currentPart.parent.rotation · partPosition;
        end
        position = position + partPosition;
    end
    return position;
end

Algorithm 1: Determining a robot part's position

A.2.2. Trajectory length

The partial trajectory length algorithm simply sums up the Euclidean distance between each pair of successive points on the trajectory until a certain point has been reached. Since the

total length of the trajectory is simply the length up until the last point no special algorithm is needed for that case.

input : A trajectory theTrajectory; an index targetIndex until which to measure
output: The length of the trajectory in whichever unit the robot's spatial data is stored in
begin
    distance = 0;
    lastPoint = null;
    foreach index ⇒ point *in* theTrajectory **do**
        if index > targetIndex **then**
            return distance;
        end
        if lastPoint *is not null* **then**
            squaredDistanceX = $(\text{point}.x - \text{lastPoint}.x)^2$;
            squaredDistanceY = $(\text{point}.y - \text{lastPoint}.y)^2$;
            squaredDistanceZ = $(\text{point}.z - \text{lastPoint}.z)^2$;
            distance += $\sqrt{\text{squaredDistanceX} + \text{squaredDistanceY} + \text{squaredDistanceZ}}$;
        end
        lastPoint = point;
    end
    return distance;
end

Algorithm 2: Determining a trajectory's length

Since we can now determine the distance covered at a given point as well as the total length of the trajectory we can easily determine the progress at a given point:

input : A trajectory theTrajectory; an index targetIndex until which to measure
output: How much of the trajectory has been traversed as a number between 0 and 1
begin
    return length(theTrajectory, targetIndex) / length(theTrajectory, theTrajectory.length - 1);
end
Algorithm 3: Determining how much of a trajectory has been traversed at a given point on it (naïve implementation)

Of course this algorithm is inefficient: It traverses the trajectory twice, performing the exact same calculations. A more efficient approach is to traverse the trajectory once, keeping track of whether the current point is after the given point:

input : A trajectory theTrajectory; an index targetIndex until which to measure
output: How much of the trajectory has been traversed as a number between 0 and 1
begin
    distance = 0;
    totalDistance = 0;
    lastPoint = null;
    foreach index ⇒ point *in* theTrajectory **do**
        if lastPoint *is not null* **then**
            squaredDistanceX = $(\text{point}.x - \text{lastPoint}.x)^2$;
            squaredDistanceY = $(\text{point}.y - \text{lastPoint}.y)^2$;
            squaredDistanceZ = $(\text{point}.z - \text{lastPoint}.z)^2$;
            totalDistance += $\sqrt{\text{squaredDistanceX} + \text{squaredDistanceY} + \text{squaredDistanceZ}}$;
            if index ≤ targetIndex **then**
                distance += $\sqrt{\text{squaredDistanceX} + \text{squaredDistanceY} + \text{squaredDistanceZ}}$;
            end
        end
        lastPoint = point;
    end
    return distance() / totalDistance;
end

Algorithm 4: Determining how much of a trajectory has been traversed at a given point on it

f