8-26-2010

Automated Construction of Robotic Manipulation Programs

Rosen Diankov
Carnegie Mellon University

Automated Construction of Robotic Manipulation Programs

Rosen Diankov

CMU-RI-TR-10-29

Submitted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Robotics

The Robotics Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

26 August 2010

Thesis Committee:
Takeo Kanade, Co-chair
James Kuffner, Co-chair
Paul Rybski
Kei Okada, University of Tokyo

Keywords: robotics, autonomous manipulation, object recognition, planning, grasping, simulation, inverse kinematics, motion control, camera calibration

Abstract

Society is becoming more automated with robots beginning to perform most tasks in factories and starting to help out in home and office environments. One of the most important functions of robots is the ability to manipulate objects in their environment. Because the space of possible robot designs, sensor modalities, and target tasks is huge, researchers end up having to manually create many models, databases, and programs for their specific task, an effort that is repeated whenever the task changes. Given a specification for a robot and a task, the presented framework automatically *constructs* the necessary databases and programs required for the robot to reliably execute manipulation tasks. It includes contributions in three major components that are critical for manipulation tasks.

The first is a geometric-based planning system that analyzes all necessary modalities of manipulation planning and offers efficient algorithms to formulate and solve them. This allows identification of the necessary information needed from the task and robot specifications. Using this set of analyses, we build a planning knowledge-base that allows informative geometric reasoning about the structure of the scene and the robot's goals. We show how to efficiently generate and query the information for planners.

The second is a set of efficient algorithms considering the visibility of objects in cameras when choosing manipulation goals. We show results with several robot platforms using grippers cameras to boost accuracy of the detected objects and to reliably complete the tasks. Furthermore, we use the presented planning and visibility infrastructure to develop a completely automated extrinsic camera calibration method and a method for detecting insufficient calibration data.

The third is a vision-centric database that can analyze a rigid object's surface for stable and discriminable features to be used in pose extraction programs. Furthermore, we show work towards a new voting-based object pose extraction algorithm that does not rely on 2D/3D feature correspondences and thus reduces the early-commitment problem plaguing the generality of traditional vision-based pose extraction algorithms.

In order to reinforce our theoric contributions with a solid implementation basis, we discuss the open-source planning environment OpenRAVE, which began and evolved as a result of the work done in this thesis. We present an analysis of its architecture and provide insight for successful robotics software environments.

Contents

1 Toward A New Level of Automation **1**
- 1.1 Need for Automated Construction . . . 2
- 1.2 Framework Design . . . 4
- 1.3 Computational Approach . . . 7
- 1.4 OpenRAVE . . . 8
- 1.5 Thesis Outline . . . 9
- 1.6 Major Contributions . . . 11
- 1.7 Publication Note . . . 12

2 Manipulation System **13**
- 2.1 Problem Domain . . . 14
  - 2.1.1 Task Specification . . . 14
  - 2.1.2 Robot Specification . . . 16
- 2.2 System Modules . . . 19
- 2.3 Component Relationships . . . 22
- 2.4 Execution Process . . . 25
- 2.5 General Guidelines for Autonomy . . . 30
- 2.6 Discussion . . . 32

3 Manipulation Planning Algorithms **33**
- 3.1 Planning in Configuration Spaces . . . 34
- 3.2 Planning to a Goal Space . . . 37
- 3.3 Planning to a Grasp . . . 40
- 3.4 Planning with Nonlinear Grasping Constraints . . . 46
  - 3.4.1 Relaxed Formulation . . . 47
  - 3.4.2 Discretized Algorithm Formulation . . . 49
  - 3.4.3 Randomized Algorithm Formulation . . . 51
  - 3.4.4 Experimental Validation . . . 53

3.5 Planning with Free-Joints . . . 57
3.6 Planning with Base Placement . . . 60
3.6.1 Base Placement Sampling . . . 61
3.6.2 Two-Stage Planning with Navigation . . . 63
3.6.3 BiSpace Planning . . . 65
3.7 Discussion . . . 73

4 Manipulation Planning Knowledge-base **75**
4.1 Inverse Kinematics . . . 78
4.1.1 Basic Formulation of Inverse Kinematics . . . 80
4.1.2 Evaluating Equation Complexity . . . 82
4.1.3 Solving 3D Translation IK . . . 84
4.1.4 Solving 3D Rotation IK . . . 89
4.1.5 Solving 6D Transformation IK . . . 91
4.1.6 Solving 4D Ray IK . . . 92
4.1.7 Handling Redundancies . . . 95
4.1.8 IKFast Results . . . 96
4.2 Grasping . . . 98
4.2.1 Force Closure Squeezing Strategy . . . 99
4.2.2 Caging Strategy . . . 104
4.2.3 Insertion Strategy . . . 106
4.3 Kinematic Reachability . . . 108
4.3.1 Uniform Discrete Sampling . . . 111
4.4 Inverse Reachability . . . 113
4.5 Grasp Reachability . . . 119
4.6 Convex Decompositions . . . 121
4.6.1 Padding and Collisions . . . 122
4.6.2 Advantages of Volume Representations . . . 123
4.6.3 Configuration Distance Metrics . . . 124
4.7 Object Detectability Extents . . . 127
4.8 Discussion . . . 130

5 Planning with Sensor Visibility **133**
5.1 Sampling Visibility Configurations . . . 135
5.1.1 Sampling Valid Camera Poses . . . 137
5.1.2 Detecting Occlusions . . . 140
5.1.3 Sampling the Robot Configuration . . . 142

5.2 Planning with Visibility Goals . . . 143
5.3 Integrating Grasp Selection and Visual Feedback . . . 145
5.3.1 Stochastic-Gradient Descent . . . 146
5.4 Humanoid Experiments . . . 149
5.5 Industrial Bin-Picking Experiments . . . 155
5.6 Discussion . . . 157

6 Automated Camera Calibration 159
6.1 Problem Statement . . . 160
6.2 Problem Formulation . . . 161
6.3 Process Outline . . . 163
6.3.1 Application to an Environment Camera . . . 169
6.4 Calibration Quality and Validation . . . 170
6.5 Discussion . . . 174

7 Object-Specific Pose Recognition 175
7.1 Pose Recognition Algorithms . . . 176
7.1.1 Classifying Image Features . . . 178
7.2 Building the Object Database . . . 179
7.2.1 Gathering Training Data . . . 180
7.2.2 Processing Feature Geometry . . . 181
7.2.3 Feature Stability Analysis . . . 183
7.2.4 Geometric and Visual Words . . . 186
7.2.5 Relational Database . . . 188
7.3 Pose Extraction using Induced Pose Sets . . . 189
7.3.1 Generating Induced Pose Sets . . . 190
7.3.2 Pose Evaluation and Classification . . . 191
7.3.3 Pose Extraction Process . . . 194
7.3.4 Experiments . . . 197
7.4 Discussion . . . 201

8 Conclusion 203
8.1 Contributions . . . 204
8.2 Future of Robotics: Robot-Task Compilers . . . 207

A OpenRAVE - The Open Robotics Automation Virtual Environment 209
A.1 Architecture . . . 210
A.1.1 Environment . . . 212

A.1.2 Validating Plugins . . . 214
A.1.3 Parallel Execution . . . 215
A.1.4 Exception and Fault Handling . . . 215
A.1.5 Hashes for Body Structure . . . 216
A.2 Interfaces . . . 217
A.2.1 Kinematics Body Interface . . . 218
A.2.2 Robot Interface . . . 218
A.2.3 Collision Checker Interface . . . 219
A.2.4 Physics Engine Interface . . . 220
A.2.5 Controller Interface . . . 220
A.2.6 Inverse Kinematics Interface . . . 221
A.2.7 Planner Interface . . . 221
A.2.8 Trajectory Interface . . . 224
A.2.9 Sensor Interface . . . 224
A.2.10 Sensor System Interface . . . 224
A.2.11 Viewer Interface . . . 225
A.2.12 Modular Problem Interface . . . 225
A.3 Working with Real Robots . . . 226
A.3.1 Padding . . . 226
A.3.2 Jittering . . . 227
A.4 Discussion . . . 227

References **229**

List of Figures

1.1 A system that grasps cups from a bartender robot's tray and puts them in a dish rack for washing. The bottom shows the minimal set of components that have to be considered to create a functioning autonomous bartender system. Each component takes a lot of time to construct making the entire system development time on the order of a year. Our goal is to reduce this time by automating the construction of the components related to manipulation planning and target object recognition. 3

1.2 Some of the robot platforms we have tested basic manipulation on. Each robot's internal world continuously models the geometric state of the environment so planners can make the most informed decisions. 5

1.3 Given a set of robot and task specifications, we construct the databases necessary for the robot to robustly execute the task. The specifications feed into the planning and vision knowledge-bases. Each knowledge-base analyzes the specifications and constructors models to help drive the manipulation planning, sensor visibility analysis, and pose recognition algorithms. These basic set of algorithms are used in the run-time phase to calibrate the robot sensors and execute the task. 6

1.4 The OpenRAVE Architecture is composed of four major layers and is designed to be used in four different ways. 9

2.1 A basic task is composed of a set of manipulable objects, their goal criteria, and the real-world training data used to measure sensor noise models and recognition programs. 15

2.2 A robot specification should provide a CAD model with annotations of which parts of the geometry serve what purpose. 17

2.3 The modules forming a manipulation system that this thesis concentrates on. The knowledge-bases and goal configuration generators are automatically generated from the task and robot specifications. 21
2.4 The robot and task knowledge database stores relationships between two or more components. The relationships are color coded where green represents task-related relations while blue represents robot-related relations. 24
2.5 The levels of an execution architecture and a description of their inputs and outputs. 25
2.6 The steps necessary for executing the target tasks. 27
2.7 Example of localization in a 3D map using a camera. 28

3.1 Shows the connection between the robot databases when used for manipulation and grasp planning. The most involved and important process is sampling the goal configurations while taking into account all constraints and task priorities. 34
3.2 Shows the environment distances for every point on the object surface (blue is close, red is far). 35
3.3 An example of a breakdown of the manipulation configuration space of a task. 36
3.4 Grasp planning involving just the arm and gripper can sometimes require both the grasp and release goals to be considered before a grasp is chosen.. 37
3.5 Shows examples of collision-free, stable grasps that are invalid in the environment. 42
3.6 Divides the grasps that pass GraspValidator into the set of reachable (has inverse kinematics solutions) and unreachable grasps for one target. 44
3.7 Shows all the valid, reachable grasps that can be sampled from simultaneously. The grasp set size is 341 and there's 6 target objects for a total of 136 valid grasps. 45
3.8 Several robot arms performing grasp planning. 46
3.9 A robot hand opening a cupboard by caging the handle. Space of all possible caging grasps (blue) is sampled (red) along with the contact grasps (green). . 47
3.10 The basic framework used for planning discrete paths $\{q_i\}|_1^n$ in robot configuration space to satisfy paths $\{q_{target,i}\}|_1^n$ in object configuration space. . . . 51
3.11 Comparison of fixed feasibility regions (left) and relaxed feasibility regions (right) for the Manus and Puma robot arms. 53
3.12 Example simulations using the relaxed grasp-set planning formulation. . . . 55
3.13 WAM arm mounted on a mobile base autonomously opening a cupboard and a fridge. 56

3.14 WAM arm autonomously opening a cupboard, putting in a cup, and closing it. Wall clock times from start of planning are shown in each frame. 57
3.15 Shows a gripper whose fingers need to open, but cannot due to the table. Planning by setting the free joints to the arm can move the gripper to a safe location to open its fingers. 58
3.16 The planner is also applied to moving the torso joints of the robot such that the arms do not hit anything. 59
3.17 When a robot is far away from its goal, it must also plan for moving its body along with its arm. The robot should first find the possible grasps from which it can sample robot goal locations. The planning algorithms will then join the two configurations. 60
3.18 Several configurations returned from GoalSampler_BasePlacement considering both robot arms, the humanoid's torso, and multiple objects. 62
3.19 Humanoid upper body structure used to extract *arm* chains. 63
3.20 Having just a static navigation configuration $q_{navigation}$ is not enough to safely go into navigation mode, sometimes the target object might collide with the robot. 64
3.21 Robot needs to plan to a configuration such that all its limbs are within the base navigation footprint. 65
3.22 BiSpace Planning: A full configuration space tree is grown out from the robot's initial configuration (1). Simultaneously, a goal back tree is randomly grown out from a set of goal space nodes (2). When a new node is created, the configuration tree can choose to *follow* a goal space path leading to the goal (3). Following can directly lead to the goal (4); if it does not, then repeat starting at (1). 66
3.23 Comparison of how the distance metric can affect the exploration of the arm. The top image shows the search trees (red/black) generated when the distance metric and follow probability is weighted according to Equation 3.14. The bottom image shows the trees when the distance metric stays uniform across the space; note how it repeatedly explores areas. The goal space trees are colored in blue. 70
3.24 Scenes used to compare BiSpace, RRT-JT, and BiRRTs. 72
3.25 Hard scene for BiSpace. The forward space tree (red) does not explore the space since it is falsely led over the table by the goal space tree (blue). . . . 73

4.1 Computational dependency graph of all the components extracted from the relational graph in Figure 2.4. 76

4.2 An outline of the parameterizations **ikfast** can solve along with how the equations are decomposed. . . . 79
4.3 The labeled joints (black) of the PR2 robot's right arm. . . . 85
4.4 The labeled joints (black) of the Kuka KR5 R850 industrial robot. . . . 86
4.5 Two types of separable kinematic chains, which allow rotation joints (red) to be solved separately from translation joints (blue). T_0 is the base link frame, T_{ee} is the end effector link frame. . . . 92
4.6 The parameterization of the ray (green) with the last two axes interesting. . . . 93
4.7 The parameterization of the ray (green) using a sub-chain of the Barrett WAM arm. . . . 94
4.8 Several robot platforms whose inverse kinematics can be solved by **ikfast**. . . . 96
4.9 The gripper first approaches the target object based on a preferred approach direction of the gripper. Once close, the fingers close around the object until everything collides, then contact points are extracted. . . . 100
4.10 A simple way of parameterizing the grasp search space. . . . 100
4.11 Examples of the types of robot hands that can be handled by the grasp strategy. 101
4.12 Grasps for the HRP2 gripper with a wrist camera mounted. Grasps contacting the wrist camera are rejected automatically. As the gripper approach varies the grasps change from power grasps to pinch grasps. . . . 102
4.13 Fragile grasps that have force closure in simulation (contacts shown). . . . 102
4.14 Example caging grasp set generated for the Manus Hand. . . . 104
4.15 A fragile grasp that was rejected by the random perturbation in the grasp exploration stage, even though it mathematically cages the handle. . . . 106
4.16 Goal is for both manipulator tips to contact the surface of the target object. The left box shows failure cases that are pruned, the right box shows the final grasp set. . . . 107
4.17 Idustrial robot using the magnet tip grasps to pick parts out of a bin. . . . 107
4.18 Barrett WAM and Yaskawa SDA-10 kinematic reachability spaces projected into 3D by marginalization of the rotations at each point. . . . 109
4.19 The HRP2 reachability space changes when the chest joint is added (left vs right). Also, the HRP2 wrist has a unique joint limits range, which can increase the reachability density by 2x if handled correctly. . . . 112
4.20 Shows one of the extracted equivalence sets for the Barrett WAM (869 sets), HRP2 7 DOF arm (547 sets), and HRP2 8 DOF arm+chest (576 sets). Each was generated with $\Delta\theta_I = 0.15$, $\Delta z_I = 0.05$. . . . 114

4.21 Base placements distributions for achieving specific gripper locations; darker colors indicate more in-plane rotations. 117

4.22 The grasp reachability map using validated grasps to compute the base distribution (overlayed as blue and red densities). The transparency is proportional to the number of rotations. 120

4.23 Examples of convex decompositions for the geometry of the robots. 121

4.24 Examples of convex decompositions for the geometry of the robots. 122

4.25 Convex decomposition makes it possible to accurately prune laser points from the known geometry. 123

4.26 The swept volumes of each joint, where the parent joint sweeps the volume of its children's joints and its link. 125

4.27 Example scenes used to test the effectiveness of the configuration metric. . . 126

4.28 Camera locations that can successfully extract the object pose are saved. . . 128

4.29 The probability density of the detection extents of several textured objects using a SIFT-based 2D/3D point correspondence pose detection algorithm. . 129

5.1 Comparison of a commonly used grasping framework with the visibility-based framework. Because the grasp selection phase is moved to the visual feedback step, our framework can take into account a wider variety of errors during execution. The robot platforms used to test this framework (bottom). . . . 134

5.2 Object initially hidden from the robot, so its pose is not known with much precision. It takes the robot three tries before finding it. 136

5.3 The real robots with the a close-up simulation of the wrist cameras are shown in the top two rows. Given the real camera image, the silhouette of the gripper (bottom) is extracted (black) and sampled (red), then the biggest convex polygon (blue) not intersecting the gripper mask is computed. 139

5.4 For every camera location, the object is projected onto the camera image and rays are uniformly sampled from its convex polygon. If a ray hits an obstacle, the camera sample is rejected. The current estimate of environment obstacles is used for the occlusion detect. As the robot gets closer and new obstacles come into view and a better location of the target is estimated, the visibility process will have to be repeated. 140

5.5 The two stage visibility planning method is as efficient as the one-stage grasp planning method because it divides and conquers the free space. 144

5.6 When initially detecting objects, the robot could have a wrong measurements. If it fixes the grasp at the time of the wrong measurement, then when it gets close to the object, the grasp could be infeasible. This shows the necessity to perform grasp selection in the visual feedback phase. 145
5.7 The scenes tested with the visibility framework. The images show the robots at a configuration such that the visibility constraints are satisfied. 149
5.8 The calibration error from the head camera can be very inaccurate when detecting over large distances. 150
5.9 Can efficiently prune out all the visibility candidates before sampling by using reachability. 151
5.10 The full process of mobile manipulation using visibility, reachability, and base placement models. 152
5.11 Manipulation with dynamic obstacles from range data (red boxes) using visibility configurations. 153
5.12 The full process of mobile manipulation using visibility, reachability, and base placement models. In order for the robot to grasp the object, it has to move to the other side. But at the other side, the only way to see object is with its wrist camera. 154
5.13 When sampling base placements with visibility, the reachability of grasp sets also have to be considered, otherwise the robot will need to move again before it can grasp the object. 154
5.14 Industrial bin-picking scene has a ceiling camera looking at a bin of parts and a gripper camera. 155
5.15 Collision obstacles around target part need to be created for modeling unknown regions because the robot does not have the full state of the world. . 155
5.16 Grasp Planning is tested by simulating the unknown regions. 156

6.1 Frames used in hand-eye camera calibration. 162
6.2 Describes the automated process for calibration. Assuming the pattern is initially visible in the camera image, the robot first gathers a small initial set of images inside a cone around the camera (green) to estimate the pattern's location. Then the robot uses visibility to gather a bigger set training images used in the final optimization process. 164
6.3 The detectability extents of the checkerboard pattern (show in black). 166
6.4 Several configurations of the robot moving to a visible pattern location. . . . 168

6.5 Example environment camera calibration environment. The pattern is attached to the robot gripper and robot uses moves it to gather data. Sampled configurations are shown on the right. 169

6.6 A plot of the intrinsic and absolute error (y-axis) vs the gradients of the reprojection error (x-axis). The top plots were generated by taking all combinations of 5 images from a bigger database, the bottom plots were with combinations of 10 images. This shows that gradients do not hold any information about the confidence of result. Left side is projection error used to optimize the intrinsic parameters while right side is the absolute error of f_x. 171

6.7 Graphs the second smallest eigenvalue of the matrix used to estimate the intrinsic camera parameters. As the eigenvalue increases, the deviation f_x becomes smaller and more stable. The left graphs are done for combinations of 5 images while the right graphs are with more than 11 images. The pattern is more apparent for smaller number of observations. More observations decrease the standard deviation of the solution, but the stability is still dependent on the second eigenvalue. 172

7.1 A well established pose extraction method using 2D/3D point correspondences to find the best pose. Works really well when the lighting and affine invariant features can be extracted from the template image. 177

7.2 Database is built from a set of real-world images, a CAD model, and a set of image feature detectors. It analyzes the features for stability and discriminability. 179

7.3 A 3DOF motorized stage and a checkerboard pattern for automatically gathering data. 180

7.4 Example rendering a blue mug from novel views using only one image in the database. 181

7.5 Examples CAD models (first row), the training images (second row), and the extracted depth maps (third row) of several objects. 182

7.6 Orientations from image features are projected onto the object surface and are clustered. By re-projecting the raw directions on the surface plane and converting to angles (right graph), we can compute all identified angles (in this case 0° and 180°) and the standard deviation of the particular mean direction $\pm 15.5^\circ$. 183

7.7 Shows the stability computation for a hole detector and the stability detected locations (red). 184

7.8 Stability computation for a line detector (upper right is marginalized density for surface). The lower images show the lines from the training images that are stably detected (red) and those that are pruned (blue). 184
7.9 Shows the statistics of a hole detector on industrial metallic parts. Each part was trained with 300 images around the sphere. The right side shows the final filtered features, which are used for extracting geometric words. 185
7.10 Shows the effect of poorly labeled data sets on the raw features (blue) and the final filtered features (red). The filtered clusters for the correctly labeled set become much more clear. 186
7.11 The histogram and labels of a feature's descriptor distribution on the object (in this case, it is the size of the holes). 187
7.12 The major shape clusters extracted. The clusters on the left (simpler shapes) are more frequent than the clusters on the right (more complex shapes). . . . 187
7.13 PostgreSQL database that stores the relationships and dependencies of all offline processes used for pose extraction and object analysis. 188
7.14 An example of the induced poses for a corner detector. The corner confidence $\psi(f)$ (top) is used to prune out observations with low confidence. The final induced poses $\mathcal{T}(f)$ for a corner detector are stored as a set (bottom). 189
7.15 Several induced poses for a particular curve (red). 190
7.16 Mean images of the induced poses for an edge detector of a paper cup (in simulation). 191
7.17 The individual detector scores for good and bad poses for the object in Figure 7.3. Using Adaboost, the misclassified good poses and misclassified bad poses are marked on the data points. 193
7.18 For an object that has four holes, there are many valid poses of the object in the real image that can hit all four holes. In fact, using hole features is not enough to extract a confident pose, 1D curve features are absolutely necessary. 194
7.19 The search process first matches image features to a database, then randomly selects a set of features and tests for consistency by intersecting their induced pose sets. Each pose candidate is validated based on positively supporting and negatively supporting features lying inside the projected object region. . 195
7.20 Several examples of chosen image features (blue) and their pose hypotheses (in white). The bottom row shows the positively (green) and negatively (red) supporting features of the pose hypothesis (blue). 196
7.21 Results of pose extraction using induced pose sets. 198
7.22 Ground-truth accuracy results gathered for each DOF of the pose. 199

7.23 Some geometric words cluster very densely around specific points on the object surface. These stable points can be used for alignment. 200

8.1 The construction process can be treated as an offline compiler that converts the specifications into run-time programs while optimizing the speed of its execution. 207

A.1 The OpenRAVE Architecture is composed of four major layers and is designed to be used in four different ways. 211
A.2 Distance queries. 219
A.3 Physics simulations. 220
A.4 Several simulated sensors. 224
A.5 Simple functions a viewer should support. 225

List of Tables

2.1 Task Specification Parameters 15
2.2 Robot Specification Parameters 17

3.1 Statistics for the scenes tested showing average planning times (in seconds) and size of the grasp sets used. 54
3.2 Increase in Feasibility Space when using relaxed planning compared to fixed-grasp planning. 54
3.3 Average planning time in seconds for each scene for the scenes in Figure 3.24. 72

4.1 Average time for calling the ikfast solver with the success rate of returning correct joint values for transformation queries. 97
4.2 Analytic Inverse Kinematics Parameters 97
4.3 Force-closure Grasping Parameters 104
4.4 Kinematic Reachability Parameters 111
4.5 Inverse Reachability Parameters 118
4.6 Grasp Reachability Parameters 121
4.7 Statistics and final distance metric weights for the Barrett WAM joints. . . . 126
4.8 Statistics and final distance metric weights for the Barrett WAM joints. . . . 126
4.9 Convex Decomposition Parameters 127
4.10 Detectability Extents Parameters 129

5.1 Average processing times for the first visibility stage for thousands of simulation trials. 143
5.2 Average planning times (with success rates) of the visual feedback stage. . . 149

6.1 Initial intrinsic calibration statics showing the mean and standard deviation of calibration results using 5 images. Note how the standard deviation is much smaller when the principal is fixed to the center of the image. 165

6.2 Reprojection and 3D errors associated with calibrating the extrinsic parameters of the gripper camera. Both intrinsic error and reprojection errors are in pixel distances, with reprojection error using only one pose for the checkerboard by computing it from the robot's position. 169

Acknowledgments

First of all, I would like to thank my two advisors Takeo Kanade and James Kuffner. Thank you both for the amount of patience with my work and the time you have put into explaining the intracacies of research presentation and the power of language. Kanade-sensei, it has been a great honor working with you for four years. I have learned more about doing solid research and its meaning from you than anyone else. You have taught me skills that few graduate students ever get a chance to experience during their time of study. James, from the beginning of starting my graduate studies, I have felt a deep connection between us. Both of us have come from a very mathematics- and graphics-heavy background, and we eventually decided that our skills are better spent on solving serious robotics problems. Our conversations about planning, AI, and software architectures have always been exhilarating. Both of you have been very intrumental in shaping my thoughts about the robotics field and the attitude I should approach all challenges with.

I would like to thank the rest of my committee members Kei Okada and Paul Rybski for giving invaluable feedback throughout the thesis writing. It has been an amazing learning experience and your comments and criticisms really helped clear up the content and final thesis message.

Throughout my graduate studies, I had the honor of working with some of the best researchers in the robotics field on many robot platforms. I would like to thank Satoshi Kagami and Koichi Nishiwaki from AIST Japan who supported the initial research on autonomous manipulation on the HRP2 humanoid robot. I owe many thanks to our initial robotics team at Intel Research Pittsburgh: Sidd Srinivasa, Dave Ferguson, Mike vande Weghe, and Dmitry Berenson. The numerous projects on autonomous manipulation, the contant demos, and the great conversations will be always remembered. I would like to thank everyone at Willow Garage for working hard to have the PR2 arms ready for testing the manipulation planners. Special thanks to Morgan Quigley, Brian Gerkey, Ken Conoly, and Josh Faust, and the other ROS (Robot Operating System) developers for making an awesome distributed robotics platform. I would like to thank Ryohei Ueda, Yohei Kakiuchi, Manabu Saito, and Kei Okada for the tremendous help with HRP2 humanoid experiments at University of Tokyo. I didn't

think it was possible to stay and sleep in a lab without going home for a week straight until I worked with you guys. I would like to specially thank Furukawa Makoto coming from the industry. All the countless conversations we had on automating factory processes and the current state of the art really helped put this thesis into perspective. You've been a great friend and teacher, and it was always fun to bounce new ideas off of you. Finally, I would like to thank my lab mates Jun-sik Kim and Jeronimo Rodrigues for the great work we have done together.

I would like to thank my office mate Tomoaki Mashimo. It has been really fun discussing with you the state of the robotics, how it will impact society. You have taught me a lot about Japan and were always there when I had something on my mind to say. I also thank Aki Oyama for the great conversations we've had about robotics venture companies and how we can meet the demands of today's consumers. Your passion for starting communities that achieve great things as a whole is contagious, keep it up.

Finally, I would like to thank the entire OpenRAVE community, which rapidly grown over the past years. All your great feedback has made OpenRAVE a very powerful tool for planning research. Thank you for all the patience at times of major changes, it was all for the best. Your help has let OpenRAVE become a success story for open-sourced research.

To my parents, who showed how to make dreams come true

Chapter 1

Toward A New Level of Automation

Today robots can be programmed to perform specific, repetitive tasks both in the industrial setting and at the home. The continuous demand of new products and new services in our society implies that there will be no shortage of new tasks that robots can perform without human intervention. However, the current set of programming tools put a limit on the rate at which new programs can be developed to meet the changing task specifications. For most companies today, it takes on average six months to a year to develop a new assembly line for mass producing one product. Considering the current trends of automated production and services using robots, the amount of human resources available will continue to limit growth until we can find a process that can automatically construct all the necessary programs for completing a designated task. Because of increasing lifestyle demands in the 21st century, a more efficient automation technology has to step in to compensate for the world's limited resources and fill the growing labor gap [Ausubel (2009)]

Today more than 1,000,000 industrial robots are operational worldwide, but very few, if any, of them actually employ complex planning systems and geometrically reason about their environments. The problem is that working with robots in the real world is plagued with uncertainties like positioning error, which prevents an autonomous robot to completely trust the output of the automatically constructed programs. A robot's behavior needs to be confirmed by a person before running on the real assembly line, and this requires a lot of engineering manpower and detailed adjustments of the programs. To be able to trust a completely automated behavior, the errors due to robot manufacturing, sensors acquisition, and simulation modeling need to be quantified by *calibrating* the robot, a crucial process for robot automation. Calibration corrects the internal models of the robot sensors and its geometry such that the robot can begin to compute an accurate measure of its surroundings at all times.

A system that can automatically generate programs should handle common industrial environments, reason about the kinematic and sensing capabilities of the robot, learn to recognize the target objects, and minimize uncertainty due to calibration, perception, and execution errors by using sensing and planning in a feedback loop. As part of the manipulation process, we dedicate a significant part of this thesis to explicitly consider the information camera sensors provide when planning robot movement.

1.1 Need for Automated Construction

The reasoning components required for developing robust robot behavior can be complex, and require a deep understanding of the geometric, computational, and physical phenomena that underlies the problem. Although robotic automation has been gaining popularity in factory settings and robotic systems have started becoming robust enough to be able to autonomously navigate across city streets [Urmson et al (2008)], the amount of development and robot programming required to setup a robot to reason about its surroundings and autonomously perform a task is immense. Even the simplest and most common task of robustly picking up an object in the environment and placing it in another place involves an amount of autonomy and system infrastructure whose development cost is higher than the economics of most products.

For exapmle, consider the bartender-arm system we developed in Figure 1.1 to explore the possibilities and requirements of autonomous robots [Srinivasa et al (2008)]. Figure 1.1 shows the outline of the task and the components of the system. The bartender first builds a map of its indoor environment and moves from place to place allowing people to place their cups on it. After a short period of time, the bartender drives to a designated location where an arm unloads the cups from it into a dish rack. Once the arm has picked up all cups, the bartender continues to collect cups from people. Creating such a system involves constructing the following robot system components:

- Motion controllers for the base, arm, and hand, the controllers run on dedicated hardware guaranteeing real-time low-level command execution.
- Perception module for reading laser range data and camera sensors and updating the environment model with where the robot is and where the cups are.
- Navigation module for avoiding obstacles and moving around designated locations in the indoor environment.
- Simulation software to compute robot kinematics and geometric information.
- Manipulation planning infrastructure that allows the arm to reason about grasping the cups and putting them in the dish rack.

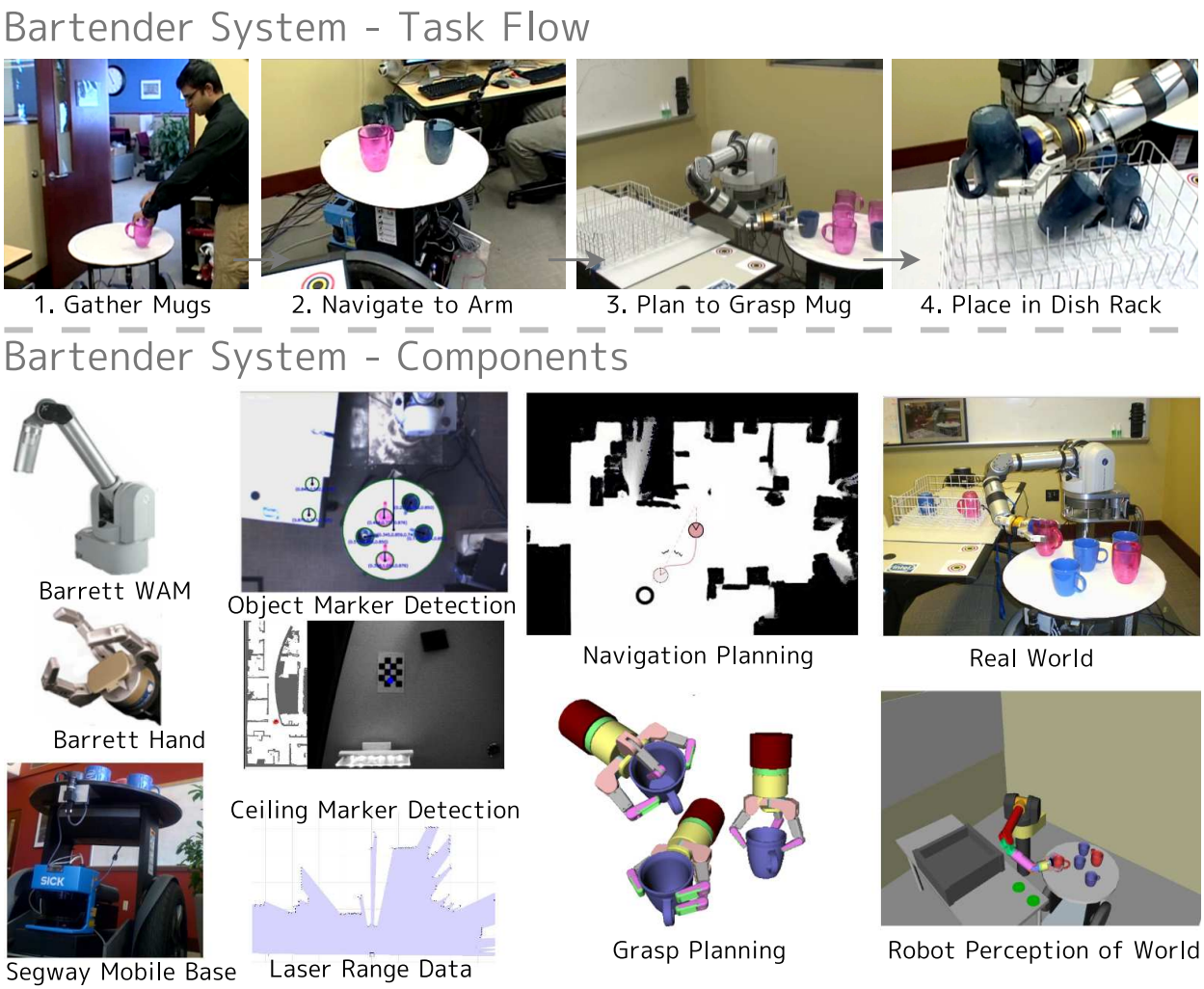


Figure 1.1: A system that grasps cups from a bartender robot's tray and puts them in a dish rack for washing. The bottom shows the minimal set of components that have to be considered to create a functioning autonomous bartender system. Each component takes a lot of time to construct making the entire system development time on the order of a year. Our goal is to reduce this time by automating the construction of the components related to manipulation planning and target object recognition.

Possibly the weakest link in the entire chain of components is the manipulation system and how it interacts with the perception system to complete its tasks. Manipulation requires the highest precision to localize and detect the object and is the source of most of the failures in the final system. Even for the simple bartender system, the interplay between the perception and planning systems has a very big impact on overall task performance. Therefore,

the automated construction process has to explicitly consider the quality of information each sensor is capable of producing in order to make the best decisions about robot placement. For more complex manipulation systems involving mobile manipulators, multiple arms, onboard camera sensors, and a plethora of target object types, it becomes an even bigger challenge to analyze all the pieces to achieve working reliable robot behavior.

We target the manipulation framework for industry professionals. Although professionals thoroughly understand their working environment and constraints, they are not experts in motion planning, geometric analysis, and the intricacies of computational theory. Therefore it is most advantageous to give them an input space that represents the minimal domain knowledge necessary to define the task. This domain knowledge is separated from the rest of the information like algorithmic parameters, which can be automatically generated by logical reasoning and analysis. In the presented manipulation framework, professionals specify their domain knowledge in the form of specifications that define the CAD models of the robot and the task, semantic identification of the robot parts and environment locations, task constraints, and training data. Using the specifications as inputs, the framework should use its tool-set of generic algorithms to *construct* the knowledge-bases and execution programs so that the robot can automatically calibrate itself, perceive it surroundings, plan in the environment to pick-and-place objects, and reliably execute the plans for the task.

1.2 Framework Design

In this thesis, we analyze the most common manipulation tasks in industrial and smart-home environments: have a robot reliably move a target object from one place to another. Pursuing this goal has led us to test manipulation algorithms on more than eight different robot systems, some of them are shown in Figure 1.2. As work toward this thesis progressed, each successful demo has required significantly less time than the previous demos to construct; towards the final humanoid experiments performed in this thesis, it took only a few weeks to construct, test, and employ the framework. At its current state, the framework has grown to a point that is starting to see applicability outside of the research sector. Through experiences with all these platforms, several patterns in the development processes have begun to appear, which we formally define and describe in the later chapters. The most important design decision has been in separating the manipulation framework into two parts: a set of generic algorithms running on the robot, and a construction process that can *adapt* the generic algorithms to a wide variety of robots and tasks.

The thesis tackles several areas associated with the manipulation problem where it is still not clear what components to use, how to automate their generation, and how to efficiently

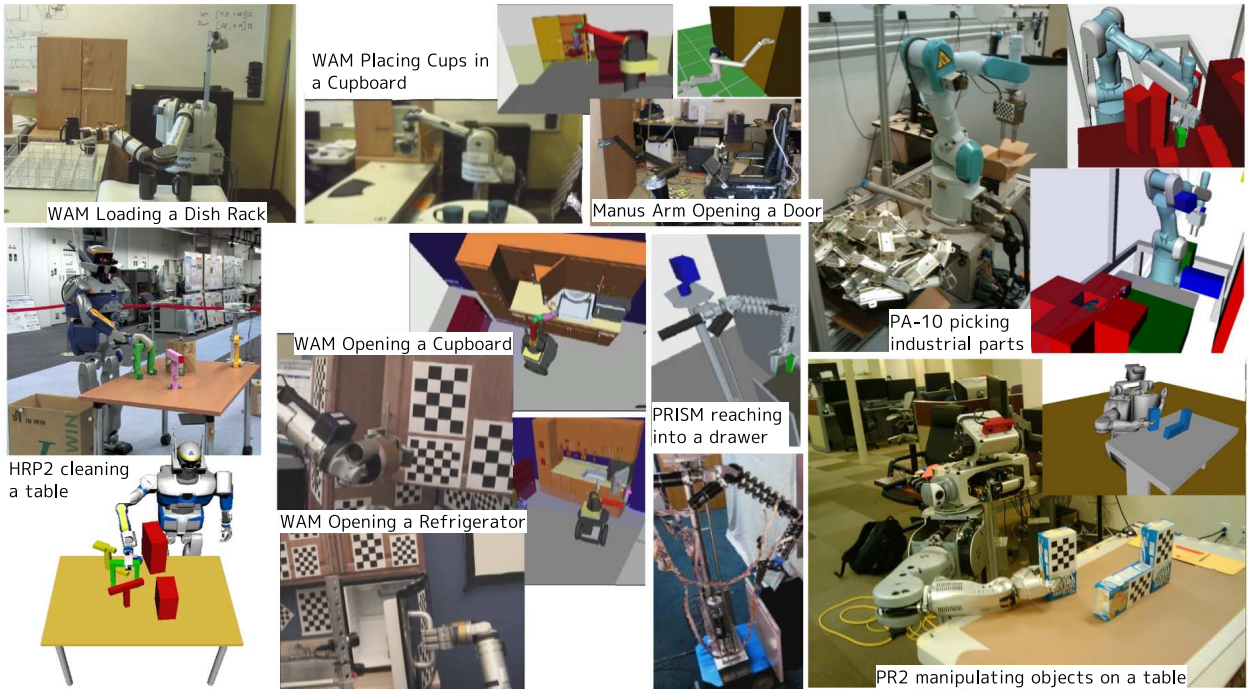


Figure 1.2: Some of the robot platforms we have tested basic manipulation on. Each robot's internal world continuously models the geometric state of the environment so planners can make the most informed decisions.

combine the algorithms into a final working system. The areas we specifically focus on are geometric-based analyses of the robot and the task, whose outcome leads to a more automated construction process. With our framework, it is possible to construct programs requiring minimal explicit user input, which we quantitatively define as the robot and task specification. The framework has the ability to *adapt* to new tasks and robot kinematics so even people with minimal planning and vision knowledge can quickly get a robot to automatically complete a basic manipulation task.

The goal is a minimal set of algorithms such that the system components mentioned so far can come together into a coherent system that achieves the domain of tasks we define. Figure 1.3 shows a breakdown of the construction process and the components that we play a key role in the culmination of the framework. The task and robot specifications are the domain knowledge input into the system, they provide the CAD models and several basic semantic definitions like where the robot arms are; their complexity has been kept to a minimum so that non-experts can easily define their own configurations. We define a manipulation planning knowledge-base that organizes all databases, models, and heuristics necessary for manipulation tasks. The knowledge-base is generated from the task and robot specifications,

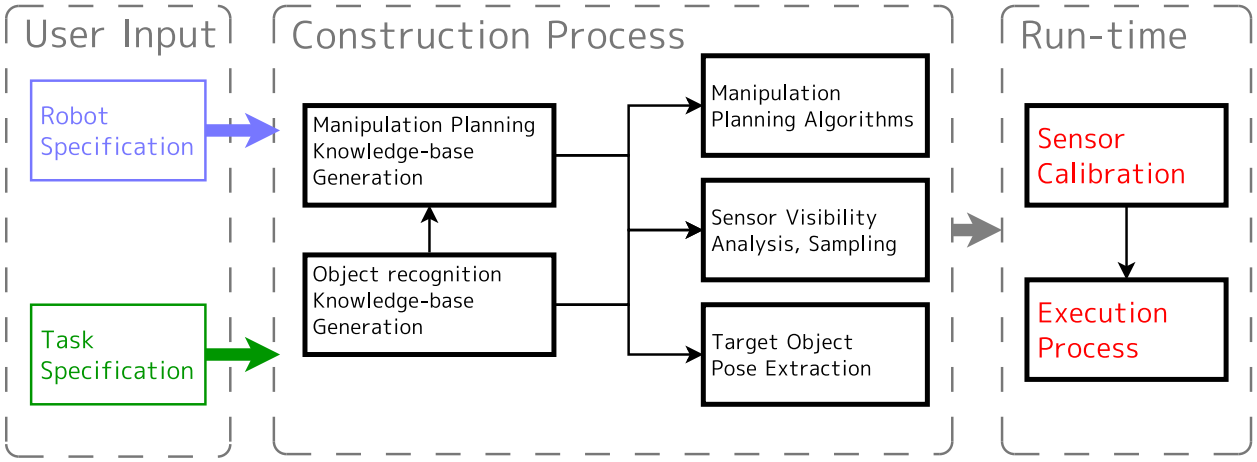


Figure 1.3: Given a set of robot and task specifications, we construct the databases necessary for the robot to robustly execute the task. The specifications feed into the planning and vision knowledge-bases. Each knowledge-base analyzes the specifications and constructors models to help drive the manipulation planning, sensor visibility analysis, and pose recognition algorithms. These basic set of algorithms are used in the run-time phase to calibrate the robot sensors and execute the task.

and represents frequently used information that can be computed offline and is dependent on robot and task structure. The object recognition programs are trained at this point using the target object CAD model and a set of training images showing the appearance of the object. Similar to the planning knowledge-base, a vision-based database can help in organizing the object-specific information for pose extraction programs. Using the knowledge-bases, the generic planning and sensor visibility algorithms responsible for robot movement are selected depending on the task and instantiated with the specific knowledge-base models. We motivate the need for a minimal set of motion planners and goal configuration samplers that use this knowledge-base to complete all required manipulation tasks. After all databases and planners are trained and before the robot can execute its plans, the system has to calibrate the robot's sensors so its internal representation matches the real world. This calibration method should be completely automated when considering the manipulation system as a whole so that it can be performed at any time. Once the calibration phase is done, it the task can be executed using a simple execution pipeline.

Although we pose no time constraints on the planner and sensing algorithm computation, we loosely aim for the entire robot system to complete its target tasks on the order of minutes. We stress here that the planning algorithms and metrics we analyze are meant to prioritize reliable execution of the task rather than generating smooth, time-optimal, energy-optimal, safety-optimal, or natural-looking robot motions. Most of these heuristics would require

a different set of algorithms that also incorporate velocities and time into the planning process, which are not covered in this thesis. Because of the complexity of dynamics and lack of accurate dynamic simulators for robotics, the thesis only analyzes the kinematics and goemetry processes of a manipulation task. We treat all dynamics and force-feedback processes as part of the robot controller that moves a robot. In the context of industrial robots, most task can be automated without requiring programs with complex force-feedback loops. Furthermore, execution failures come at the highest cost where an assembly line has to be completely stopped in order to recover from the error. Given that most problems plaguing today's autonomous robots are failures due to perception and calibration errors, we dedicate all goals of the thesis to achieving reliability.

1.3 Computational Approach

When designing the system architecture, we followed the driving principle that **offboard computation and memory are freely available**. This assumption allows us to concentrate on designing more powerful algorithms, store more flexible models, and use scripting languages rather than spending time on optimizations for onboard embedded computers. Many of the final robot systems presented in this thesis have gigabytes of preprocessed knowledge-bases and distribute their computation across many computers. There are very few programs that need to run on the robot like: drivers for the robot hardware, communication software for allowing access to robot systems to offboard devices, and reactive controllers that where the robot has to respond on the order of several milliseconds. Any other program that communicates at a low-bandwidth and sends at approximately 30Hz or less can be easily moved offboard where the computational and power constraints are much less restricted.

Today's advances in quickly accessing large databases has allowed a paradigm shift in how models and information is stored and processed in computers. Previously, researchers relied on parameterizations for capturing the essence and patterns of a sub-problem. Using parameterizations, it becomes possible use numerical methods and gradient-descent techniques to quickly search for the best solutions. Parameterizations also allow very compact storage of the data since most of the work for a particular sub-problem is encoded into the type of parameterization. However, parameterization comes at the cost of flexibility in representing more complex spaces that have not been considered at the design phase of the algorithm. This implies that newer parameterizations have to be studied and developed as the complexity of tasks and demands of users continue to grow. Fortunately, there exist kernel modeling methods that can provide the flexibility in modeling any complex space without the need

for explicitly parameterizing it, which comes at the cost of computation and memory. In order to design a manipulation framework capable of handling a wide variety of usage cases without limiting users to particular patterns, this thesis models spaces using **kernels** and **probability distributions**, and it efficiently queries these spaces during runtime processes using **sampling-based** methods. In each area of manipulation, we analyze algorithms for efficiently sampling hypotheses that abide by the constraints of the task and the goals of the robot. These analyses have allowed us to *shift* the representational complexities of the domain tasks into computational complexities, which have converted manipulation problems into problems of architecture design and speed optimizations. Today's hardware allows us to leverage such modeling tools while providing flexible modeling of any domain space without sacrificing the required planning speed.

Algorithm Performance Measurements

We present many timing experiments throughout the thesis in order to give a general insight on how fast samplers are. We record all timings of an algorithm on a single thread using an average CPU core, despite the fact that the samplers and planners presented in this thesis can be ridiculously parallelized. Furthermore, all planning experiments are measured from the time the planner starts settings up structures and caching its models to the time a path is generated; no post-processing operations like smoothing are performed.

1.4 OpenRAVE

For facilitating the development of this framework, we developed a planning environment titled OpenRAVE, the Open Robotics Automation Virtual Environment. OpenRAVE forms the foundation of all the results presented in this thesis, and its open-source nature has allowed over a hundred researchers across the world to quickly run manipulation programs. The OpenRAVE architecture shown in Figure 1.4 simplifies the implementation and testing of planning algorithms and allows new modules to easily inter-operate with other modules through a well-defined Application Programming Interface (API). The structure of operating systems, computer languages, and debugging tools has motivated the division of OpenRAVE into four major layers: a **core layer** for the API, a **plugins layer** for extending functionality, a **scripting layer** for run-time analysis, and a **database layer** for generation and quick retrieval of the knowledge-bases. These layers combine into a set of tools non-experts can use in analyzing their problem without requiring an in-depth knowledge of software systems, manipulation planning, and physics.

The OpenRAVE architecture began as a result of this thesis work and has evolved through

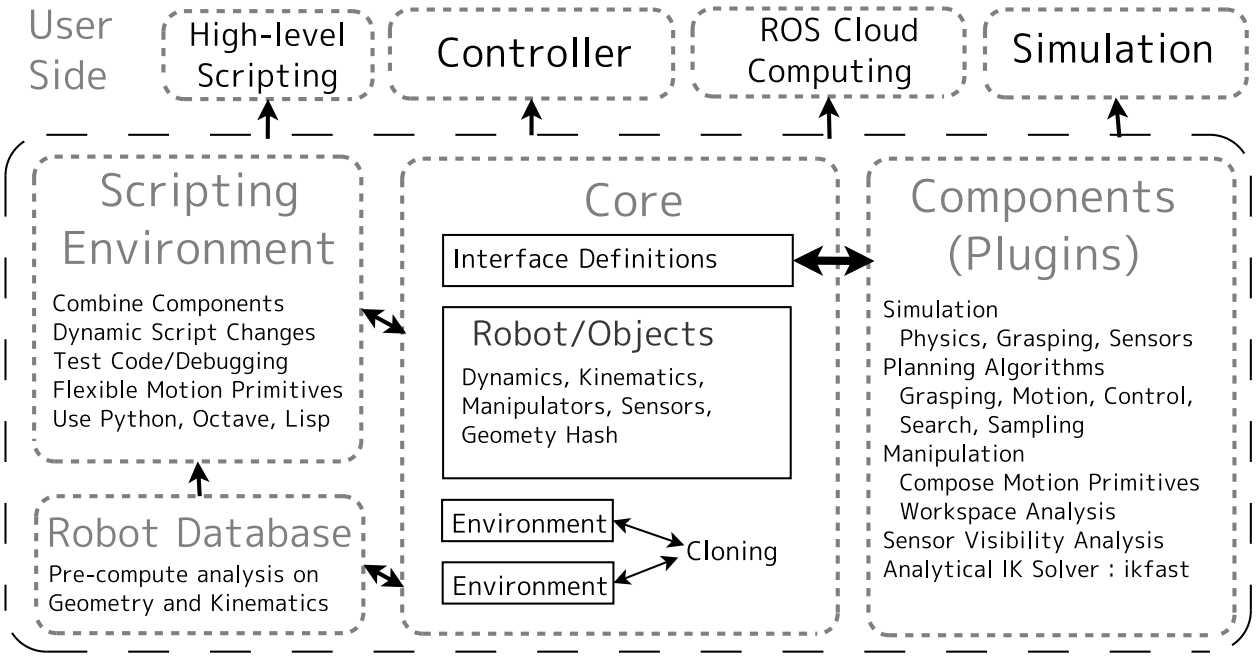


Figure 1.4: The OpenRAVE Architecture is composed of four major layers and is designed to be used in four different ways.

the testing of many robot platforms and the feedback of a rapidly growing OpenRAVE community. Having a large user base composed of planning researchers and non-exerts like mechanical engineers has provided invaluable feedback into how a manipulation framework should be designed and how the user should interact with it. As a consequence, OpenRAVE has gone through a re-design phase two times, each new design has automated more basic manipulation functions while removing functions that spread the environment too thin and don't provide much value. Such a process has allowed OpenRAVE to concentrate only in manipulation and geometric analyses, which has allowed new innovations beyond any capabilities of existing planning environments.

1.5 Thesis Outline

Chapter 2 is responsible for motivating the algorithms and contributions for the bulk of the content presented in the thesis. It begins with the definition of the manipulation problem and the robot and task specifications that serve as inputs to it. Section 2.2 presents a component-based execution architecture that combines all the processes into one coherent system. Section 2.3 formulates the inter-relations between all components as a relational

graph, which can be used to motivate what information can be computed offline and what is online. In order to simplify the system design and assumptions we make on each of the algorithms, Section 2.5 sets up several guidelines for autonomy for the execution architecture. These guidelines are used to define the environment, robot, and task complexity and the typical scenes in which we will evaluate our framework.

Chapter 3 introduces the fundamental planning algorithms necessary for safely moving the robot across the scene. We describe the general formulation of how to search a robot configuration space using an arbitrary goal space. This formulation is then applied to sub-problems of manipulation like planning to grasp, planning with the robot base, planning to opening doors, and planning with free-joints. We stress importance in defining goal samplers that can quickly analyze the scene and give a direction for the robot to search.

Chapter 4 presents the structure and efficient generation of a manipulation planning knowledge-base that stores information dependent on the current robot and task specifications. The knowledge-base analyzes object grasping, inverse kinematics, kinematic reachability, grasp reachability, robot base placement, correct collision and volume modeling, distance metrics, and vision-based models of object detectability. For each component, it provides automated algorithms for its generation and discusses usage in the context of the manipulation framework.

Chapter 5 deals with the relationship between vision sensors, visibility capabilities, and planners. It defines a visibility sampler that allows the robot to reason about where to place its sensors in order to provide better estimates of the target objects. The visibility framework is tested on several real robots and results are discussed. Furthermore, we present a combination of visibility and visual feedback methods for continuous validate of the plan.

To make the entire system complete, Chapter 6 delves into algorithms to calibrate the position of cameras with respect to the robot coordinate system. While most algorithms usually require several manual processes to sample and gather good calibration data, we formulate a method that allows the robot to automatically reason about the calibration pattern detectability and how to achieve a good sampling of training data for solving the parameters. Using the planning theories developed in this thesis, we show how to calibrate the robot's onboard camera sensors from a pattern anywhere in the world.

On the perception side, Chapter 7 discusses object pose recognition programs. To provide the extraction programs with better features we present a capturing method that allows automatic analysis of any image feature's stability and discriminability on the object's surface by taking advantage of the availability of the object's geometry. We discuss the practical trade-offs with the analysis and experiences with creating a database for this information. Furthermore, we present a novel voting-based algorithm based on induced pose sets that

does not require 3D object features or feature correspondences.

As a closure to this thesis, Chapter A focuses on the implementation and integration issues of the presented framework using the OpenRAVE planning environment. We discuss the practical issues and enabling technologies of OpenRAVE.

1.6 Major Contributions

The theoretical major contributions of this thesis are:

- An explicit construction process that takes domain knowledge into the form of specifications, and generates a knowledge-base that allows quick runtime execution of plans.
- Proposed several new manipulation planning algorithms that allow efficient planning with goal spaces, mobile manipulators, and sensor visibility. We present a generalized goal configuration sampler that encodes everything about the robot and task when prioritizing goals.
- A planning knowledge-base that automated analyses for object grasping, inverse kinematics generation, kinematics reachability, and object detectability. The algorithms are described in the context of previous work and their contributions lie in their non-parametric formulations and flexibility in handling different tasks and objects.
- Methods that efficiently plan with the camera sensor to provide reliable execution. We present a two-stage approach to manipulation planning and prove its necessity for easily working with sensors without tight feedback loops.
- An automated camera calibration method that can compute the camera model and camera placement on the robot surface with a single button click. The method using motion planning to handle environment collisions and guarantee visibility with the calibration pattern. Furthermore, we contributed a way to measure to quality of the calibration data.
- A method to analyze the a feature detector's distribution on the object surface to compute stable and discriminable features, which allows for extraction of geometric and visual words for the specific object.
- The *induced-set* pose extraction algorithm that can accurately extract pose under very cluttered scenes and works under no assumptions on the object surface and the feature detectors. The algorithm is voting-based and uses a novel learning-based process for evaluating pose hypotheses in the image.

1.7 Publication Note

Part of the work done in this thesis has appeared in previous publications. The system outline in Chapter 2 was first introduced in the basic manipulation systems in [Srinivasa et al (2008, 2009)] where we combined navigation and manipulation planning to form one coherent system. Part of the grasp planning theory in Chapter 3 [Diankov and Kuffner (2008)] was first covered in [Diankov and Kuffner (2007); Berenson et al (2007); Diankov et al (2008a)]. Door opening and caging grasp construction was first published in [Diankov et al (2008b)]. The visibility theory covered in Chapter 5 was first presented in [Diankov et al (2009)]. Finally, the OpenRAVE architecture of Chapter A was first published in [Diankov and Kuffner (2008)], since then the architecture has evolved at a great pace and the hope is for this thesis to serve as the more current OpenRAVE publication.

Chapter 2

Manipulation System

The system architecture dictates how all the individual components in a system interact with each other to form a collective system that can perform its target tasks. The most basic systems center around a **Sense→Plan→Execute** loop where the information flow is strictly defined between the three components; the formulation allows a divide-and-conquer approach to solving robotics problems [Russell and Norvig (1995)]. Although several complex and capable systems have been designed with this structure [Srinivasa et al (2009)], the robot execution needs to consider feedback processes in several hierarchy levels in order to recover from errors and changes in the environment. Each hierarchy level can represent sub-loops of Sense→Plan→Execute depending on the time-constraints of the solution and at what semantic level the planning is being done at. Recent research in autonomous vehicles [Baker et al (2008)] has shown that a critical component to robust execution is an execution monitor that continually validates the current path the robot follows and takes a corresponding action when an error is detected. The ALIS architecture [Goerick et al (2007)] further generalizes the concept of hierarchies and monitoring across the system by mathematically modeling the relationships between information flow. A common observable pattern is that all successful robotics systems have many sensor feedback loops beyond low-level controllers that each cooperate to move the robot.

The goal of this chapter is to setup the flow of the execution system and dictate the structure of the rest of the thesis in explaining the manipulation processes. We begin by defining the manipulation problem to be solved and covering all the system elements involved in producing the final robot behavior and computational infrastructure. We cover robot and task specifications that clearly state what domain knowledge is required in the problem, which helps define the extensibility of the system. In the simplest form, the robot specification defines the CAD model and kinematics of the robot along with several semantic

labels for what geometry corresponds to attached sensors and arms. We then provide the system map of all the functional modules and how they pass information across each other. Compared to other systems, there are two new modules in our system which play a key role: the goal configuration generators and the knowledge-base of precomputed, planning-specific information. Furthermore, we analyze the manipulation system from the angle of individual components that play key roles in the planning algorithms and help us formulate and design the planning knowledge-base. For example, the arm component and gripper components are relate to each other through the kinematics. Using the basic module and component definitions, we formulate an execution process that defines the general flow in which the robot will move in order to complete its task. Towards the end, we briefly introduce set of autonomy guidelines to formally define the requirements of each of the modules in our execution system.

2.1 Problem Domain

The complexity of the algorithms we use is directly related to the assumptions and expectations made on the framework. In this section we define a set of loose requirements for the types of tasks and types of robots our framework should handle. These requirements will allow us to simplify the design of the algorithms while still providing a useful and convenient framework for manipulation.

2.1.1 Task Specification

In this thesis we are interested in tasks that involve *moving a single rigid object* to achieve these set of goals:

- Place the object in designated locations with respect to another object or the map of the environment.
- Act on the object by moving it a relative distance from its initial configuration, placing it a designated location, or pushing it like a button.

Figure 2.1 shows the required information the user should specify to the system. We require the CAD model of every *manipulable* object be specified. We are interested in manipulation specific objects that have a fixed geometric and material properties, and are not interested in generalizing manipulation algorithms semantic classes of objects. Because there is a lot of research on how to automatically acquire the geometry of an object using LIDAR and vision sensors [Curless et al (1995); Zhang et al (2002, 2003)], we can safely

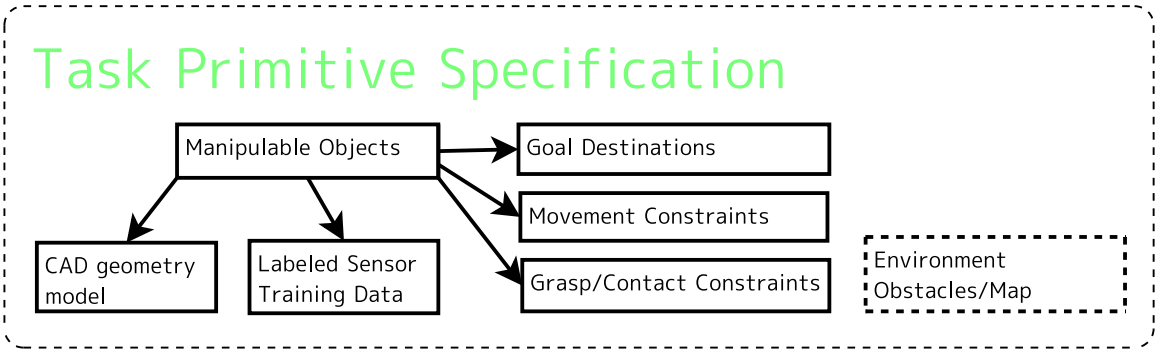


Figure 2.1: A basic task is composed of a set of manipulable objects, their goal criteria, and the real-world training data used to measure sensor noise models and recognition programs.

Parameter Name	Parameter Description
Target Geometry	a set of rigid links (using convex decompositions) connected by joints
Target Training Data	labeled images containing the appearance for operating conditions
Target/Robot Goals	a goal sampler or explicit goal positions to move the target/robot to
Target Constraints	A function defining the valid configurations of the target
Avoid Regions	All geometric regions of the target that should not be contacted
Environment	Geometry, the up direction, and the regions of possible robot movement

Table 2.1: Task Specification Parameters

assume the geometry is known. Furthermore, the task specification should provide real world images or other sensor readings in order to learn how the object's appearance is associated with its pose, which is used to build more robust object-detection programs. Because the training data comes from sensors, it is possible to measure the uncertainty introduced by the sensor acquisition process. Having fixed geometry and appearance greatly simplifies the object pose extraction and grasping programs because the framework does not have to generalize or learn this information. This allows the generated programs to be more accurate and less prone to modeling error.

Other information in the task specification deals with defining the goal of the task and any constraints on the target object or robot motion. For example, one of the simplest task constraints would be to carry a bucket full of water while maintaining its orientation with respect to the ground so as to avoid spilling the water. Finally, an optional specification is a geometric map of the environment the robot moves in, which can be used for navigating and collision avoidance. Although there exists many algorithms to acquire environment

semantic information during run-time [Rusu et al (2008)], knowing the map can avoid the extra information-gathering step and make execution faster. Table 2.1 lists the parameters and their descriptions.

High-level Task Planning and Behavior

There are many flavors of task planning involving higher-level reasoning that can be added on top of this framework to perform more complex tasks. High-level reasoning can involve the possibility of moving obstacles out of the way for a planner to succeed [Stilman et al (2007b)], or can involve symbolic task planning that reasons about the optimal order of actions [Marthi et al (2008); Okada et al (2008)]. Harnessing the power of symbolic planners requires domain knowledge to be inserted into the system to help interpret and extract symbols from sensor data and the environment [Fikes and Nilsson (1971)]. A different approach to completing tasks are cognitive systems that have very abstract goals of learning, exploration, and task reward. Such systems are meant to mimic how the human mind works, and therefore prioritize sensor-feedback and reactive behaviors over explicit modeling of the environment [Stoytchev and Arkin (2004)]. Other systems more formally encode the task goals as robot behavior such that the collective behaviors eventually achieve the task [Stolarz and Rybski (2007)].

Although high-level task planning and cognitive behaviors are important fields, we specifically focus on basic tasks in order to separate the problems associated with automated manipulation from those of language and intelligence. Because the framework we present is deeply grounded on geometry, environment models, and physics sensor readings, it has very little semantic associations to language and function. This makes it possible to leverage the presented manipulation framework to serve as a buffer between the symbolic and behavior worlds and low-level physical manifestations of the environment state. Most high-level intelligence frameworks require symbols to be grounded [Russell and Norvig (1995)]. The framework presented in this thesis can serve as a way the basis for a set of motion primitives which can be later combined in a high-level intelligence system; the goal of this system is to create an autonomous robot, not an intelligent robot.

2.1.2 Robot Specification

A robot is composed of a set of sensors to take measurements of the environment and a set of actuators that move the robot from one place to another. Figure 2.2 shows a diagram of the information necessary for a robot specification, basically a specification should provide:

- The kinematic and geometric models of how the actuators affect the physical robot.

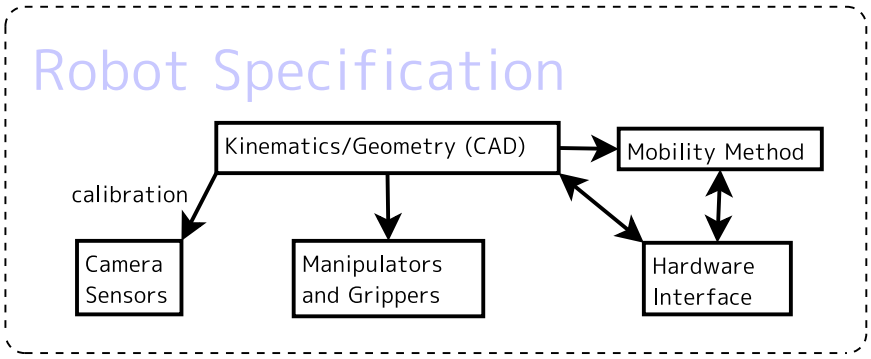


Figure 2.2: A robot specification should provide a CAD model with annotations of which parts of the geometry serve what purpose.

Parameter Name	Parameter Description
Geometry	a set of rigid links precisely defining the robot shape
Padded Geometry	convex decompositions of padded links used for environment collisions
Kinematics	Joints connecting links that define the configuration space and its limits
Dynamics	the dynamic properties of all the links and joints, parameter limits
Arms	a chain of joints defining the arm and the type of inverse kinematics to use
Grippers	a set of joints controlling the fingers/mechanism, joint directions for closing fingers
Manipulators	each manipulator consists of an arm and a gripper
Sensors	type of sensor, robot link and location attached to, sensing volume
Robot Base	how robot base moves across environment. Moves all manipulators and sensors
Avoid Regions	parts of robot like sensors that should be avoided from all collisions
Control	expected positioning errors on physical movement from inputs, and hardware limits

Table 2.2: Robot Specification Parameters

- Constraints due to the mobility method of the robot base,
- A separation of the links composing the arms and grippers of the robot. A gripper is responsible for making contact with the environment, and an arm is responsible for moving the gripper.
- The attached sensors.
- A hardware interface for controlling the robot.

Table 2.2 describes all the robot parameters in detail.

Sensing is divided into estimating the state of the robot (proprioception) and estimating the state of the environment (extero-ception). Both sensing modalities have errors associ-

ated with them that must be accounted for during the knowledge database construction and task execution processes. Proprioception errors typically take the form of fixed calibration offsets while perception errors take the form of wrong and missing measurements and misclassification of target objects. The lowest level of control in a robotic system is a tight feedback loop between the actuators and proprioception module that cancels out dynamics and physical mechanism properties, thus giving the higher level components a simpler kinematic view of the entire robot. Because this thesis mostly focuses on the interplay between the planning, vision, and execution components, we assume a simple hardware interface that communicates with the robot hardware is specified. The hardware interface should provide:

- The current position and velocity of the robot's joints at high communications rates.
- A service to control velocity at high communication rates.
- A service to follow a time-stamped trajectory of joint positions and velocities.

The rate at which communication occurs between the robot hardware greatly impacts the robot's responsiveness to the environment and should be greater than the fastest feedback loops in the execution architecture. In our case, the controller loops should be faster than the visual servoing, environment sensing, and execution monitoring loops.

The only perception sensor we examine deeply in this thesis is the camera sensor. Dense information can be extracted from camera sensors, which allows more robust object detection and object pose computation as compared to other sensors. Once a user specifies what robot links the cameras are attached to, our framework can automatically calibrate the sensor using the method in Chapter 6. We can also start computing the sensor visibility models of objects (Section 4.7) necessary for reliable planning and execution.

Robot Kinematics

The robot kinematics should be divided into:

- a mobile base that controls the in-plane 2D translation and 1D orientation of the robot,
- a robot arm that controls the 3D translation and 3D orientation of the gripper,
- and a gripper that is used to make contact with the environment and manipulate objects.

The separation of the mobile base is because navigation planners commonly treat the robot directly in the 2D workspace. Furthermore, the mobile base is responsible for moving across the environment where there are uncertainties in the composition and shape of the terrain. Such uncertainties mean that the mobile base can be very inaccurate in moving

to its desired goal position, therefore it is treated as a separate component that receives goal commands and roughly moves the robot to the destination. Depending on the mobility method and target environments, the control inputs can become arbitrary complex, so we refer to the vast controls research literature on providing simple interfaces to complicated humanoid and nonholonomic robots [Chestnutt et al (2006); Chestnutt (2007); Li and Canny (1993); An et al (1988)].

For the robot arm, we plan directly in the joint configuration space and build the necessary heuristics to speed up the planning process. Although not a requirement, the execution performance will greatly increase for robot arms that lend themselves to easy computation of the analytical inverse kinematics equations[1]. Most common robot arms today [Wyrobek et al (2008); Kaneko et al (2004); Albu-Schaffer et al (2007); Barrett-Technologies (1990-present); Exact-Dynamics-BV (1991-present)] do satisfy these properties, so the thesis concentrates on making planning easier and faster for such kinematic structures. However, inverse kinematics solvers only speed up the planning process, and are not required for robots with low number of joints such as the Katana arm [Neuronics (2001-present)].

For the gripper, the algorithms we cover are capable of handling most kinematic structures. Because, we can simulate all possible gripper preshapes and directions of approach, there are no particular restrictions on geometry or degrees of freedom. However, the entire process will greatly speed up if the following information is specified for each gripper:

- A preferable set of approach directions for the gripper; typically close to the normal vector of the *gripper* palm.
- The direction that joints should move when applying force on a grasped object. Like the human hand, gripper mechanisms are optimized for applying forces in designated directions and withstanding contact forces at them.

2.2 System Modules

We identify several key tasks for a robot to achieve basic pick-and-place manipulation:

- Calibrating the robot sensors to predict the robot state in the real world.
- Finding the target objects and sensing the environment obstacles.
- Breaking down the task into robot goal configurations.
- Moving the robot throughout the environment while avoiding obstacles.
- Considering sensor visibility for better information extraction.

[1] This requires having six or more degrees of freedom

- Recovering from errors due to perception and execution.

Implementations satisfying these functions can become arbitrarily complex depending on the assumptions researchers make on the problem. For example, creating a program to find the locations of all cars or cups in an image is an unbelievably complicated process [Li et al (2009)]. However, finding the translation and orientation of a *specific* cup is a much more well-defined problem that has an exact and unique solution. In the case of object recognition, we can generate as many object-specific vision recognition programs as the task requires without having to solve the much harder general problem of object recognition [Malisiewicz and Efros (2008); Pantofaru and Hebert (2007); Torralba et al (2007); Schneiderman and Kanade (2004)], which also brings in issues of language semantics and context [Divvala et al (2009)]. If we can develop such programs for all manipulable target objects, then the rest of the environment can just be treated as one obstacle with no extra semantics attached. Then to successfully avoid the obstacles, the robot just has to continually measure the distances to them. Exploiting the fact that the robot and task specifications contain everything necessary to describe the problem, we can use similar arguments to simplify the assumptions imposed on the technologies mentioned above, thus making the automatic construction of manipulation programs a feasible problem.

Steps for Completing a Task

We introduce the typical modules composing an autonomous manipulation system. Figure 2.3 shows how the modules pass information to each other. Before the robot starts the task, we compute a robot and task knowledge database independent on runtime information like environment obstacles. This knowledge database includes:

- Grasping models for target object and hand,
- Analytical inverse kinematic solvers,
- Robot reachability models,
- Information to perform faster and more accurate collisions,
- Robot configuration distance metrics,
- Robot sensor visibility capabilities,
- Programs to find target objects in images and extract their poses.

An execution monitor that constantly monitors the current robot state for possible obstacle collisions or unexpected sensor readings is a necessary module for robust execution [Okada et al (2006); Baker et al (2008)]. Each phase of the planning outputs a path of robot positions that are then be executed by the controller. The controller should have the

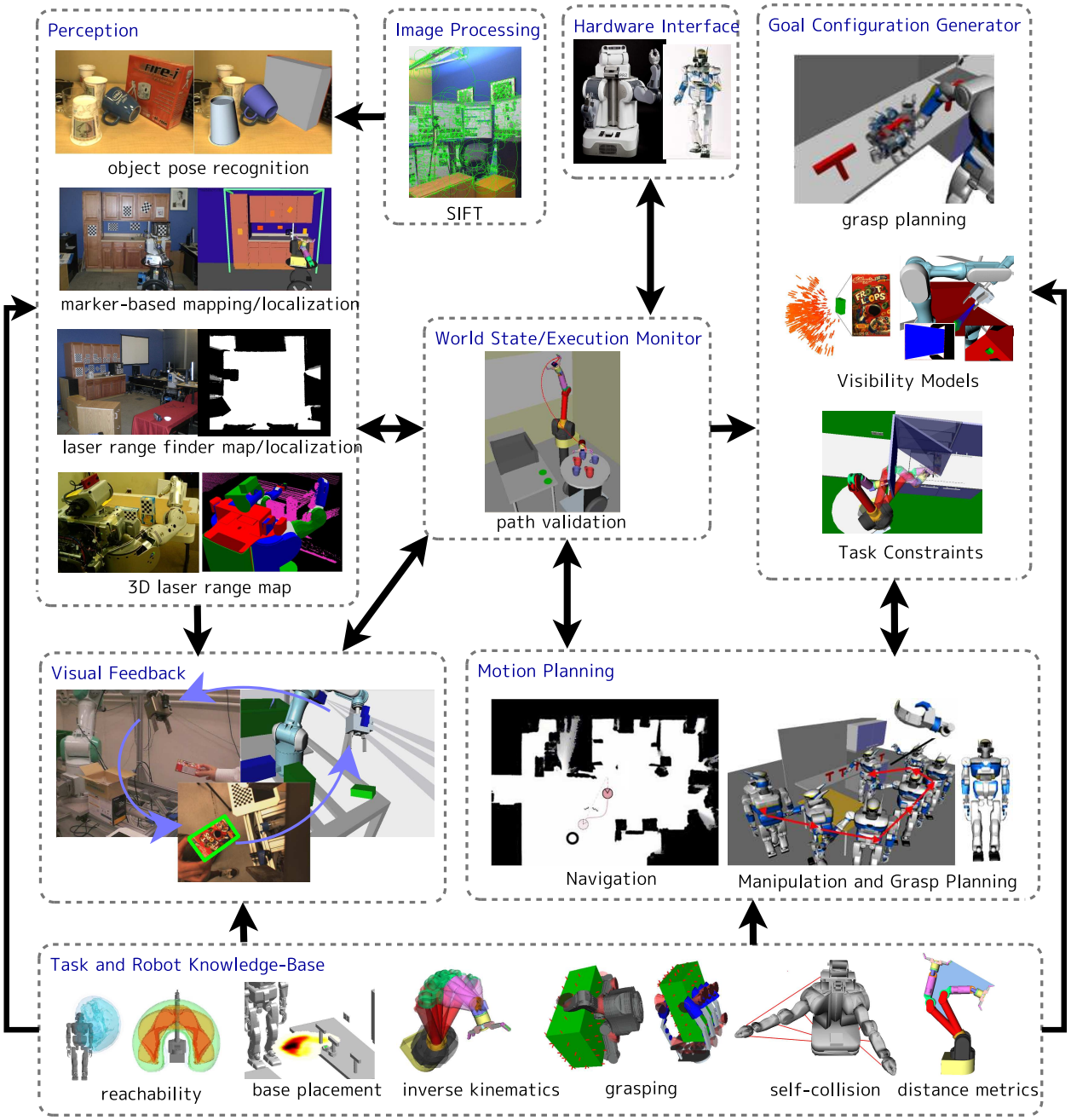


Figure 2.3: The modules forming a manipulation system that this thesis concentrates on. The knowledge-bases and goal configuration generators are automatically generated from the task and robot specifications.

capability to stop a trajectory at any point in time and restart a new trajectory avoiding the possible error condition. This ability allows us to implement the entire spectrum of error recovery conditions.

A fundamental module is the ability to safety move the robot around the environment without hitting any obstacles. This requires the robot maintain up-to-date knowledge of the obstacles in the environment. Besides obstacles, the system also needs a module to find the locations of all the target objects that must be manipulated by the robot. Combining this information with sensor noise models, we can construct an accurate environment representation in simulation where the robot can geometrically start reasoning about how to avoid obstacles and manipulate the target objects. Once a robot has found the target object, using knowledge of physics and the object geometry, we can automatically build models of how to stably grasp the object and manipulate it. Combining the grasping models and robot kinematics with the task constraints defined by the problem specification, we can compute the space of all possible robot goals and movements that will satisfy the task. Goal configuration selection should also consider sensor visibility so that the robot can guarantee the quality of the target object information extracted from the sensor data.

Robot movement is divided into planning for the base of the robot and planning for the manipulators responsible for achieving contact with the target object. Base movement, also called navigation, deals with moving the robot on a 2D map such that dynamics constraints, real-time constraints, and collision-avoidance constraints are met [Ferguson (2006); Kuffner et al (2003)]. Manipulator movement is slightly more complex because of its high number of degrees of freedom, so randomized algorithms are introduced that trade-off optimality for algorithm tractability. Once a global robot trajectory is computed from the planners, it is sent to the low level robot controller that filters the movements through the robot dynamics and commands the actuators. During execution of the trajectories, an execution monitor must continually validate the robot target path with the new environment information to guarantee the robot is still collision-free, the target object is still in the expected position, and the goal requirements can still be satisfied. Some task motions require very careful positioning of the robot gripper with respect to the target object. In this case, a different class of planning algorithms is used that form a tight feedback loop with the vision sensor measuring the object location.

2.3 Component Relationships

We identify nine components part of the robot system:

- **Environment** - Environment snapshot storing the current state of the obstacles and target objects..
- **LIDAR** - Distance data to environment using laser range finders.
- **Camera(s)** - The camera sensors and produced images.
- **Target Object(s)** - The target objects of the specified task
- **Robot Base** - The base link of the robot specifying the location of the robot.
- **Manipulator(s)** - The manipulator links and joints used to move the grippers around the environment.
- **Gripper(s)** - The grippers used to make contact with the environment.
- **Control** - Controller used to execute trajectories.
- **Planning** - The planner used to plan paths from an initial configuration to a goal configuration.

Figure 2.4 shows a detailed relational graph of the nine components; each edge links two or more components and specifies the databases constructed from those components. This relational graph can greatly help in figuring out the necessary databases to construct and the dependencies between components when using the knowledge database. Some of the relationships between these components can be computed offline directly from the robot and task specifications. It is these runtime-independent relationships that form the basis for the planning knowledge-base covered in Chapter 4.

As a simple example of traveling the relational graph, let us compute all the necessary databases needed for planning to grasp an object. In order to connect the **target object** and **planning** components, we start at the **target object** and use the grasp sets to relate to the **gripper**. Given the **gripper** locations we can use the inverse kinematics to relate to the **manipulator**. Finally, we need to use the self-collision and distance metric modules to start **planning**. However, it is clear that the **planning** component requires an **environment** that holds the state of the current obstacles which the planner will use to guarantee collision-free paths in the robot configuration space. As discussed in Chapter 3, the planning process is mostly parameterized by the robot goal-configuration function, so all the relations that have been gathered starting at the **target object** will be used to set the correct goal configurations for the planner to search a path to.

As Figure 2.4 shows, there exist many different paths when linking one component to another. In fact, all the databases along any path that relate two components should be used together to increase performance. For example, another relationship between the **manipulator** and **gripper** is the reachability, which stores an easily indexable map of which parts of the robot's workspace can be reached by the gripper (Section 4.3). We also define a reverse-indexable map that relates where the **robot base** should be placed in order to

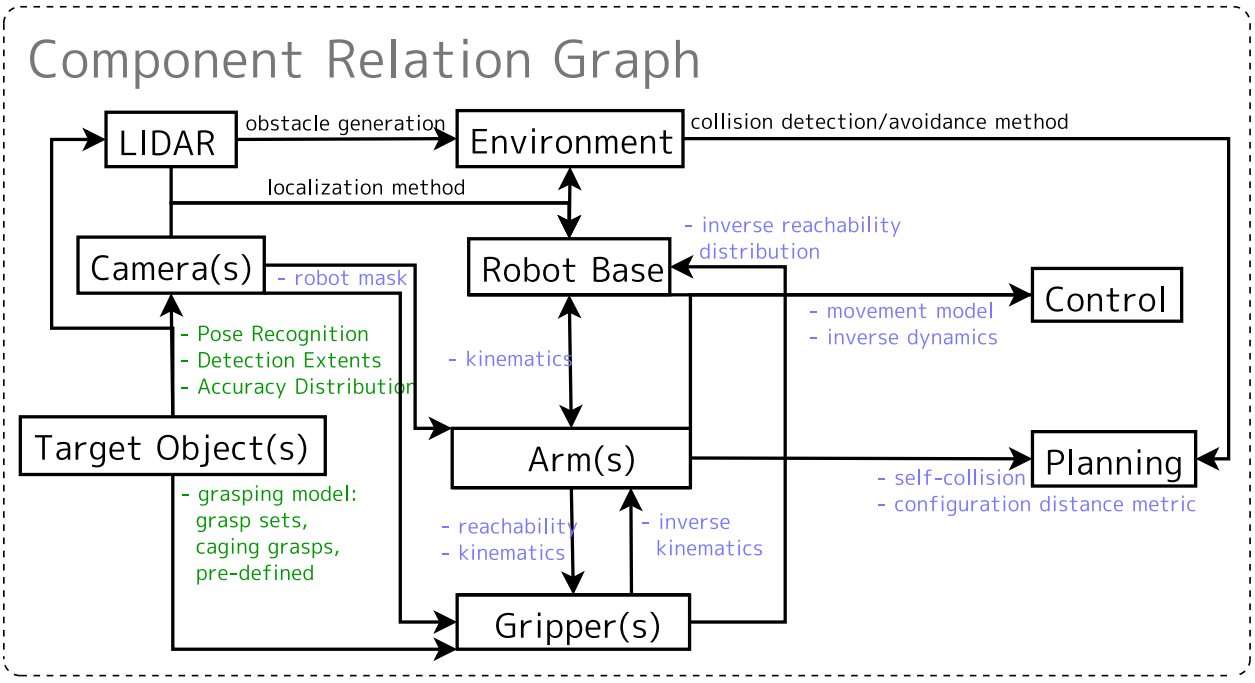


Figure 2.4: The robot and task knowledge database stores relationships between two or more components. The relationships are color coded where green represents task-related relations while blue represents robot-related relations.

achieve a particular **gripper** configuration (Section 4.4).

The task specification is mostly responsible for relating the **target object** to the **gripper** and **camera**. It allows us to construct the following relations:

- A pose recognition program taking camera images and producing a list of object locations within it, further discussed in Chapter 7.
- A map showing how the object detection relates with camera direction and distance, discussed in Section 4.7.

The main purpose of the models we identify and build for the manipulation framework is to make the planning and vision problems computationally tractable during runtime. Furthermore, we show how gathering statistics about these relations can allow us to prioritize the search spaces in the planners, and can lend themselves to optimizing the design and placement of a workplace for industrial settings.

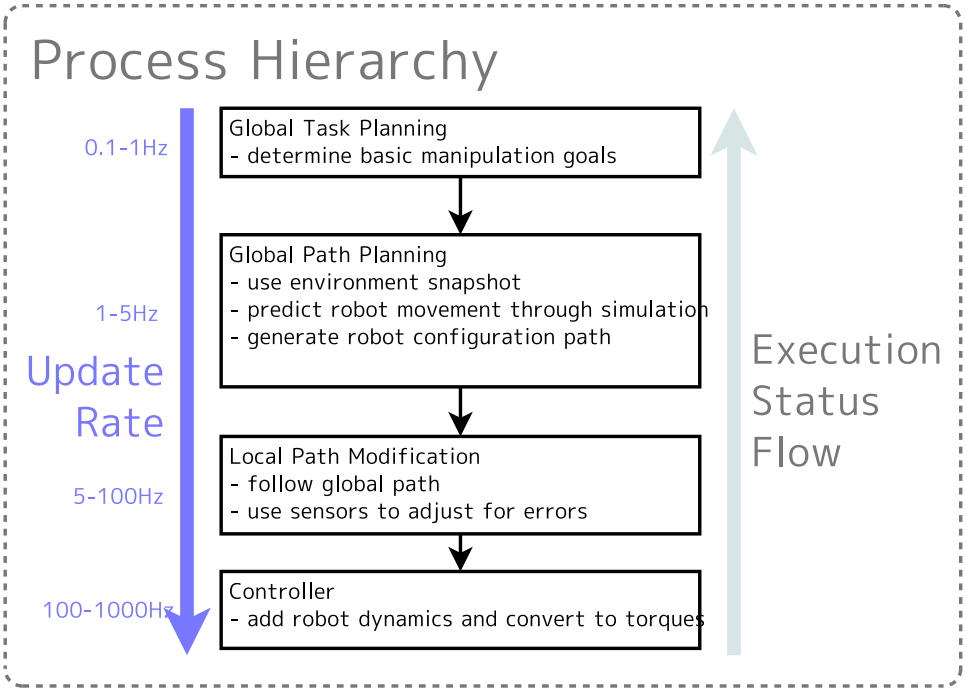


Figure 2.5: The levels of an execution architecture and a description of their inputs and outputs.

2.4 Execution Process

A common pattern for systems is to divide the execution of a plan into a global path planning component and a local feedback component that follows the global path while adjusting for local errors. In fact, a hierarchical arrangement of processes is very important when considering low-level reactive behaviors and high-level goal completion behaviors. Figure 2.5 shows an outline of the four-level hierarchy used to organize the processes. The highest task planning component concentrates on finding the order to perform manipulation primitives, so it has the slowest update rate of the system. Inside each motion primitive is a process to capture a snapshot of the environment and search for a feasible path that achieves the goal. The global paths are usually rough and use simulation to predict the future. Because such plans last up to a couple of seconds, the update rate is typically on the order of 1-5Hz. Then the global path is inserted into a sensor feedback loop which directly outputs robot position and velocity commands on the order of 5-100Hz. The purpose for the local modification phase is to maintain robot safety and stability while following the given path. The lowest level is the controllers that convert the position and velocity commands into motor torques while considering the robot dynamics and tight sensor feedback loops.

One very popular approach to manipulation is to tightly integrate dynamics and sensory feedback processes into the execution processes [Drake (1989)]. Such feedback is especially important when making contact with the environment where it is easy to get jammed due to undesired contacts and unknown friction properties [Mason (2001)]. Furthermore, machine learning can play a major role in quickly mapping the sensor inputs to torque outputs for compliant and operational-space control [Peters and Schaal (2008)]. Each of these sensory feedback modules constitute the lowest level in Figure 2.5 where torques are directly produced from the inputs. Even though these research directions have produced many unique and human-like robot behaviors, we argue that geometric analyses are still crucial to automation and unavoidable regardless of how many sensor feedback loops there are. The presented manipulation framework can quickly determine the environment topology and compute explicit goals for the entire robot configuration. Having such a toolset allows the operational space control to continuously compute the best goals while maintaining its given constraints.

It is possible to view the execution process from a task-level perspective. Figure 2.6 shows a flow chart for completing a manipulation problem by defining four types of planning situations that control the execution flow:

1. **Searching for the object**. In the beginning, the robot first searches for the object and once found, plans for a closer robot configuration where the object pose can be better estimated. Methods can involve moving the robot randomly until an object is found or moving to maximize information [Hollinger et al (2009); Saidi et al (2007)]. Regardless of the method, the output of the module should be a rough location of the object.

2. **Approaching the object**. Once a rough location is found, the robot should plan to move toward the object so that its sensors can observe the object unobstructed (discussed in Chapter 5). As the robot gets closer to its target, the new environment information could change the visibility of the target, so the planner should be called again. This step requires knowing the properties of the object detector, the (inverse) kinematics of the robot arm, and the location of the visual feedback sensor.

3. **Grasping the object**. Once a better estimate of the target object pose is achieved, the system should start considering potential collision-free ways to grasp the object (Section 3.3). Since the robot is already close to the object and the arm is directly looking at it. As an alternative to open-loop grasping, we present a stochastic gradient-descent algorithm to approach the object while considering changing grasps (Section 5.3.1). The planner should simultaneously consider the possible grasps and maintain visibility constraints necessary for continuing to track the object. If the goal position

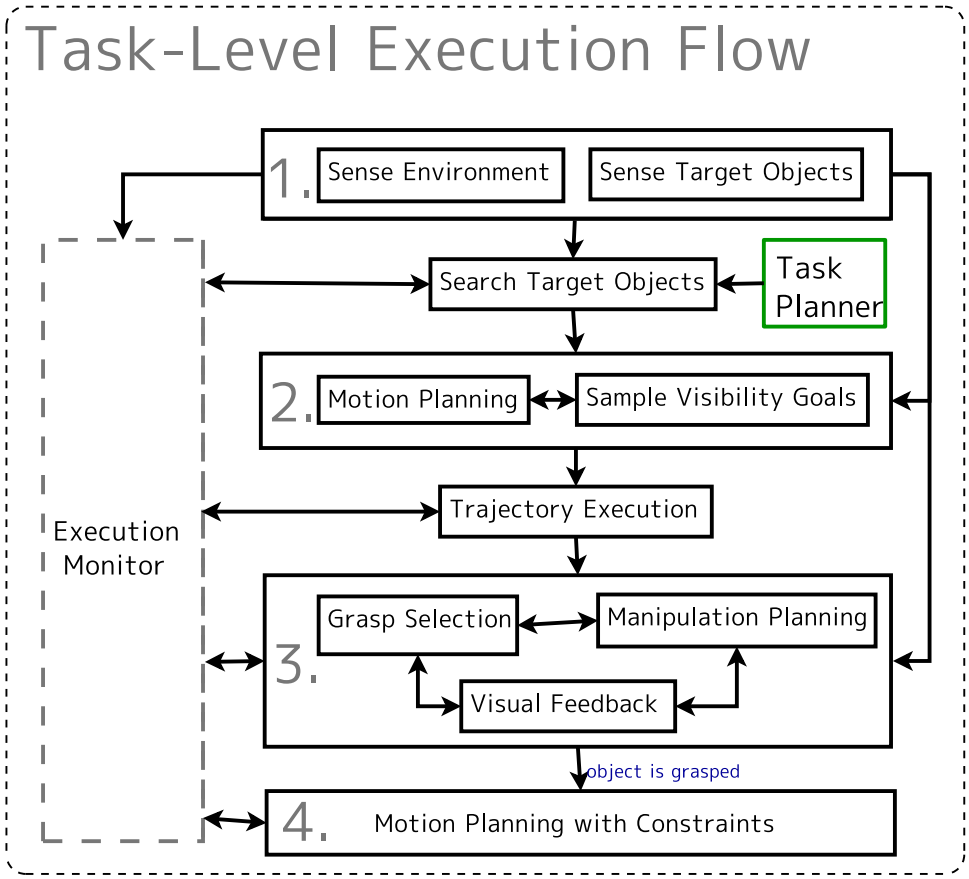


Figure 2.6: The steps necessary for executing the target tasks.

is known at this time, each grasp can be also validated by checking for the existence of a feasible goal robot configuration.

4. **Moving the object**. Once the object is successfully grasped, the planning framework switches to maintaining any specific task constraints and move the object to its goal region. Task constraints can include not tipping the object [Stilman (2007)] or moving the object along a hinge [Diankov et al (2008b)]. During robot movement, an execution monitor validates the robot configuration with the current environment estimate.

Each of these steps requires the perception modules to constantly run and generate a map of the environment and target objects. In order to take a snapshot of the environment state at a particular instance in time, each sensor needs to timestamp its data. All perception algorithms maintain this timestamp when publishing their data out to other modules. For every time instance, all sensor data are combined with the robot position at that time and

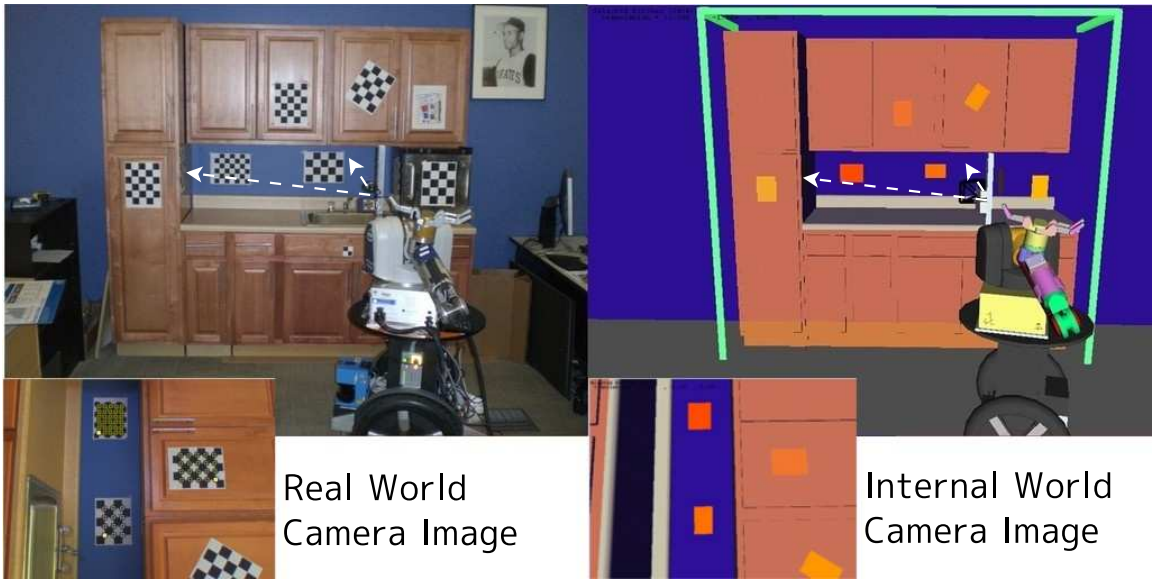


Figure 2.7: Example of localization in a 3D map using a camera.

transformed into the world coordinate system. This allows sensors to be attached anywhere on the robot without any coordinate system confusion. The should be constantly publishing their data regardless of how it is being processed, this independence on the system state simplifies many start and stop events and making it much easier for users and robots to work with. Most sensor algorithms publish at 10Hz or lower, therefore most of their computation can be offloaded onto a server of computers, which greatly reduces the requirements of load balancing.

We rely on range data from stereo or laser range finders for sensing the environment obstacles. The laser range data is used to accumulate a run-time obstacle map. This data is then processed in the world coordinate system to fill holes and create collision obstacles like boxes that are easier to use than point clouds [Rusu et al (2008)]. Processing the data in the context of manipulation is discussed in Section 4.6.2.

Very frequently in robotics, there is a map of the environment that the robot can use to navigate from place to place. Sometimes generating the 3D geometry from raw sensor data is too time consuming, so a 3D map is present that the robot can localize to. For example, Figure 2.7 shows how a robot localizes with respect to the hand-crafted 3D kitchen model using patterns randomly placed inside the real kitchen environment. The localization module should publish the best expected robot position and its mean error.

For each different object type specified in the task, a pose recognition program will be generated and will be continually running during the execution phase. Using this program,

we can gather in an offline phase all the directions from which the pose can be detected by the camera (Section 4.7). This allows the planning process to prioritize robot configurations that generate preferable views for the camera. In Chapter 7, we analyze object-specific recognition programs and present several analyzes for automatic detection of good features along with a new pose extraction algorithm. During runtime, each pose recognition program will continuously publish the detected object types, poses, and the expected error on the extracted pose. Because there can be a lot of overlap in the low-level feature sets used for the object detectors, we can separately run programs for each feature detector for each camera.

Navigation planning for indoor environments consists of creating an offline map with SLAM and planning on the 2D plane considering velocities and the non-holonomic nature of the robot base. Navigation and robot localization have been deeply studied in many other works [Likhachev and Ferguson (2009); Ferguson (2006); Urmson et al (2008); Rybski et al (2008)], therefore for this thesis we use already available navigation planning libraries from Player/Stage [Gerkey et al (2001)] and ROS [Quigley et al (2009)]. On a segway mobile base using a standard 2D laser range finder, they can usually move the base within 5cm of the intended goal. The robot localization is within 2cm when using range data and 0.5cm when using vision-based localization (Figure 2.7). Even when treating navigation as a black box, finding the correct goal to give the navigation planner so that it can handle errors of 5cm is vital to robust mobile manipulation. Nevertheless, the target objects always have to be re-detected after the base stops. In Chapter 5, we motivate the use of cameras attached to the gripper for the final reconfirmation phase. Gripper cameras allow very accurate results of the entire process.

There are two main ways to achieve robust grasping, and we choose a combination of the best of both worlds in the manipulation framework. The first way is to make the grasps themselves as robust as possible to errors in perception. In Section 4.2 we discuss a *repeatability* metric that allows us to measure if a grasp can slip or not based on simulated noise. The second way requires careful positioning of the gripper with respect to the object using visual servoing techniques, or otherwise the gripper might push the object out of the way and fail the grasp [Prats et al (2008a); Jain and Kemp (2008)]. However, visual servoing first requires the camera to view the object. In Chapter 5 motivate the need for sampling robot configurations that can achieve object visibility in the camera field of view before we begin to grasp it. Our technique is just as effective as visual servoing, except it does not rely on fixed grasps and to maintain visibility constraints as the robot moves throughout the scene. In most cases the object is static in the environment, and so viewing the object once as close is possible is more than enough to get a good accurate statement. By combining the repeatability metrics in grasping and the visibility sampling, we can assure

very reliable grasping without needing to analyze perfect sensor-less grasping algorithms or always maintain visibility.

2.5 General Guidelines for Autonomy

Our goal is to define a very conservative set of guidelines for autonomy that help us decide the necessary level of sensing and feedback required for manipulation systems. Because the level of autonomy of a system is dependent on the assumptions made on the problem, *autonomy* has become a very ambiguous term in robotics literature. If autonomy is considered to be the ability to deal with changes in a problem/environment/system, then we can look at the autonomy of the our framework as a whole in being able to deal with any robots and tasks as defined in Section 2.1, or we can look at the autonomy of our presented execution architecture in dealing with runtime changes. To eliminate the ambiguity, we only talk about autonomy guidelines with respect to the execution system of the robot in performing the specified tasks.

Environment Complexity

We define the maximal complexity on the environment geometry and its changes.

1. The system should be able to handle most living and factory environments as long as the obstacle regions can be confidently extracted and are well defined. Environments where it is not possible for the robot to reach its target objects without having to move or touching anything should not be considered. Interacting in such environments requires the robot to have a higher level reasoning module, which is not part of the task specification (Section 2.1.1).

2. The environment lighting should be similar to the training data given by the task specification. The system should not have to adjust to lighting changes or unexpected sensor data from difficult environments.

3. The robot should be able to compensate for changes in the environment within several seconds of the change.

Sensor Uncertainty

We define the extent to which the robot should monitor and recover from execution and calibration errors.

1. Robot should reason about target object observability and guarantee extents on pose uncertainty.

2. The noise for sensors measuring distances to obstacles should be modeled. A robot should treat any unexplored region as an obstacle and should give obstacles safe boundaries.

3. Robot should automatically detect when sensor calibration is off.

4. Robot should continually validate its current position with the expected goal position. If goal position is defined with respect to an object's coordinate system, that object has to be continually detected.

Error Recovery

We define the extent in which a robot should be able to recover from errors. In all cases, the robot hardware and computational framework should be fully functional at all times; we do not expect the robot to perform introspection [Morris (2007)] and detect and recover from internal problems.

1. If an obstacle or other object blocks another target object, wait for the target object to become available or choose another target.

2. If an obstacle or other object blocks the robot path, restart the planning process, locally modify the trajectory, or wait.

3. If the robot becomes stuck during a trajectory because of an obstacle collision, restart the planning process.

4. If a target object moves before grasped, the robot should compensate for the change or restart the planning process.

5. While the object is grasped, continually monitor the grasp and restart the planning process if object is lost.

6. Confirm the object is at destination once placed there.

2.6 Discussion

In this chapter we covered the outline of the manipulation architecture and discussed how all elements relate to each other and where our contributions lie within the framework. We defined a set of robot and task specifications that represent all domain knowledge being injected into the system. Then we presented a set of functional modules of a system that can achieve all requirements. Unlike other systems, we specifically make the goal generation process a module. It analyzes the scene and determines where a robot should go in order to achieve its higher level goals. In previous systems, most of these goals came from a person manually specifying *handles* or other workspace parametriztions. We also presented a robot component-based view of the manipulation system. Each component like the robot arm serves a particular purpose in the plan. Relations between these component allow us to formalize what information a particular task relies on. This analysis helps us define the planning knowledge-base in Chapter 4 that organizes and tracks all possible offline relations between the robot components we defined.

We also discussed the rough execution outline and the steps the robot takes in order to complete the manipulation plans. Each of these high level steps are decomposed into a set of primitive planner calls whose theory is covered in Chapter 3. We discussed many of the practical issues encountered when executing the manipulation pipeline. Through this discussion, we revealed the requirements for each of the components and discussed the trade-offs that have to made in order to get them to work.

We closed this section with several general guidelines of robot autonomy. For many different research groups, the definition of an *autonomous robot* differs, which causes a lot of confusion when comparing results between frameworks. The guidelines on autonomy defines the basic assumptions of the scene and environment, all algorithm developed to support this thesis are designed with them in mind. Eventually, we hope that such simple prerequisites can serve as the basis for a formal definition of levels of autonomy. Such a taxonomy will greatly help in comparing robotics architectures. However, it will only become possible once the process of combining all components of a robot system becomes as simple as installing an operating system and setting up a computer network on it. But even before such a time will come, it is necessary to understand all the elements in the most basic and fundamental level.

Chapter 3

Manipulation Planning Algorithms

Motion planning is responsible for finding a path in the robot configuration space between an initial configuration to a goal-satisfying configuration while maintaining constraints on the search space. The primary contributions of this chapter are to introduce a set of planning algorithms specific to manipulation planning and define how a planning knowledge database can be used within the framework of these planners. We divide a planner into two components: a generator for feasible, goal-satisfying robot configurations, and a search algorithm that explores the feasible space and composes the robot path. Even though the space of all search algorithms is infinite, we only concentrate on the Rapidly Exploring Random Trees [LaValle and Kuffner (2001); Kuffner et al (2003); Oriolo et al (2004)] and A* [Ferguson and Stentz (2004, 2005); Diankov and Kuffner (2007)]. RRTs have been shown to explore the feasible solution space quickly, so they are ideal for high dimensional search spaces like manipulator arms. On the other hand, A* algorithms give more power to heuristics to guide the search and are used for problems whose state space is much smaller. In order to specialize a planner to the task and robot specification, we present modifications to the goal space definitions and state space samplers. The driving principle in the planning framework is that giving a search algorithm informed configurations that get the robot closer to its goal is a much more important capability than being able to efficiently search through many uninformed configurations.

In this chapter we define the planning framework and specifically delve into the search components of planning algorithms. The search components are the static code that stays the same throughout different tasks and robots. We start with the overall algorithmic structure and show the role of the robot knowledge database as discussed in Chapter 4. Furthermore, we cover the general case of mobile manipulation when considering the complexities of grasping and visibility goals. A pattern that becomes quickly apparent in the planning algorithms

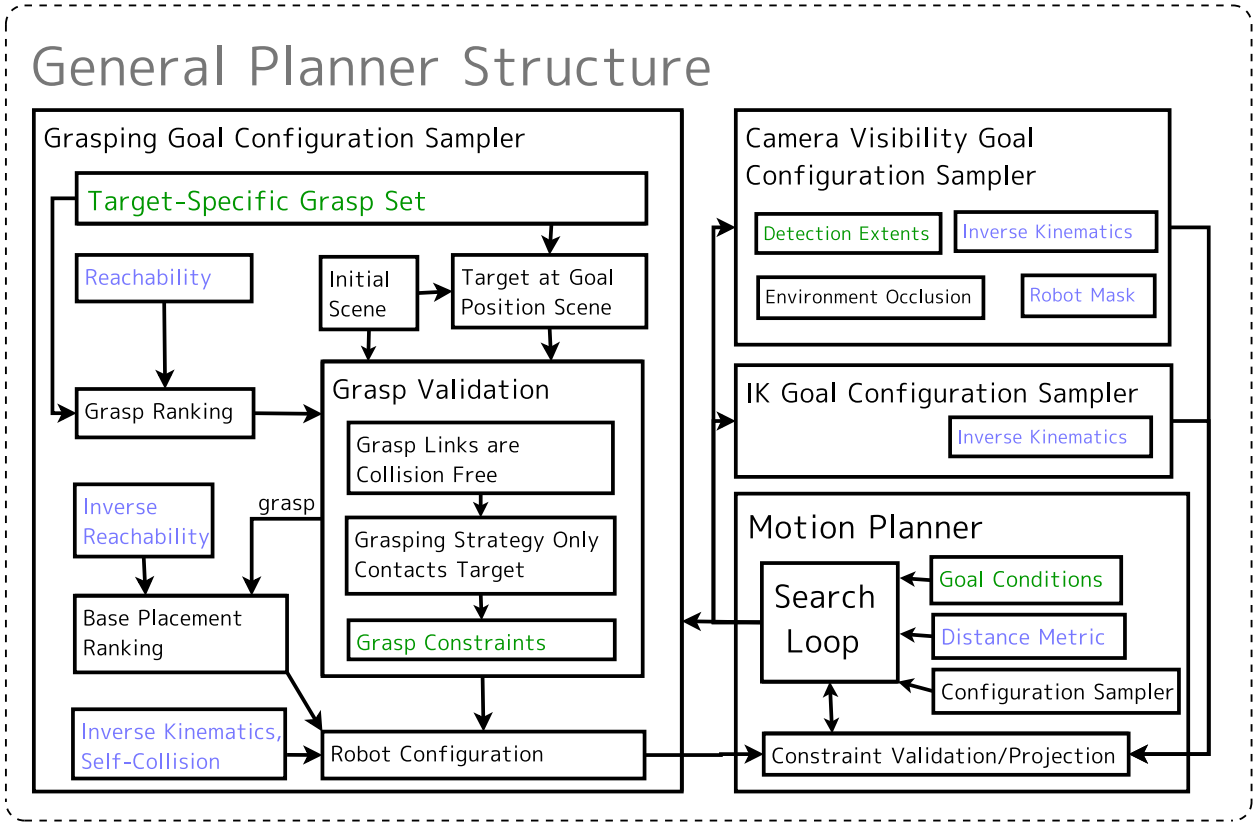


Figure 3.1: Shows the connection between the robot databases when used for manipulation and grasp planning. The most involved and important process is sampling the goal configurations while taking into account all constraints and task priorities.

we cover is that the way goals and heuristics are defined and used can have a bigger effect on the planning process than search algorithms themselves. In fact, all task-specific information is encoded in the databases, so it would be natural to expect databases to take a lead role in influencing the search.

3.1 Planning in Configuration Spaces

Figure 3.1 shows the structure and information considered in the goal samplers and how the role the planning algorithm plays in. We heavily rely on goal configuration samplers that give hints to the search system as to where to focus exploration. There are three major goal configuration samplers shown in Figure 3.1:

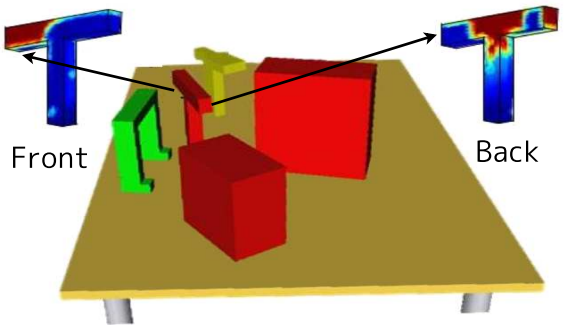


Figure 3.2: Shows the environment distances for every point on the object surface (blue is close, red is far).

- a generalized grasp sampler discussed in this section,
- an inverse kinematics sampler used to relate the work and configuration spaces, and
- a sensor visibility sampler discussed in Chapter 5.

In order to make the most informed decisions, the following elements should be considered in the manipulation planning process:

1. **Reachability** - Stores a density map measuring how close the gripper is to a position that the robot arm can actually move to (Section 4.3).

2. **Grasp Ranking** - Ranks grasps depending on how probable they are to be reached by the robot. The ranking involves evaluating the environment around the object (Figure 3.2) along with using the reachability density map.

3. **Target at Goal Scene** - If the task involves placing the object in a particular location, then we must ensure that the gripper can actually fit in this location. By having the a second scene with the predicted object goal, we can validate if a grasp is achievable.

4. **Grasp Validation** - Once a grasp is picked, it should be checked for collisions and consistency of task constraints. Checking if any links on the gripper collide with the environment is the quickest check. Next we perform the grasp strategy and check to make sure that the gripper *only* collides with the target object. If the gripper collides with anything else, we cannot guarantee the object is grasped as it was predicted, so the grasp should be rejected.

5. **Base Placement Ranking** - By inverting the reachability map, we can acquire a density map of the base locations with respect to a canonical set of grasps (Section 3.6.1). This can be used to rank the base placements when searching for valid solutions.

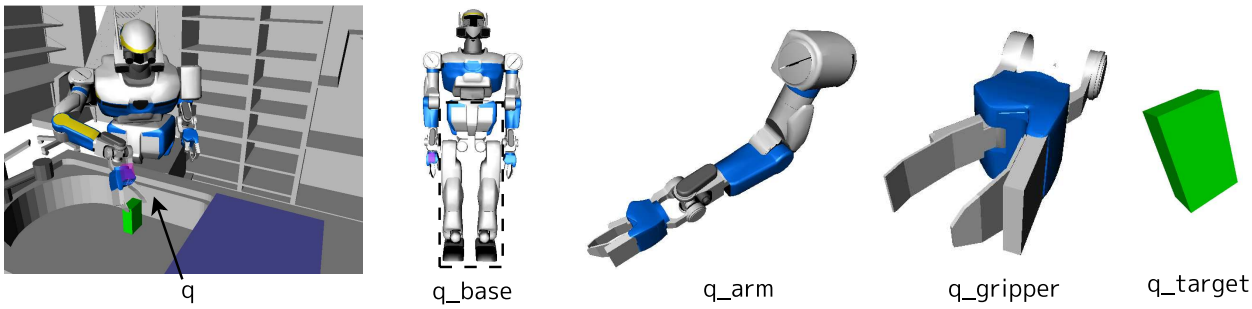


Figure 3.3: An example of a breakdown of the manipulation configuration space of a task.

6. **Final Robot Configuration** - Base placements are sampled according to their ranking and we check for the existence of collision-free inverse kinematics solutions.

Considering the above elements, we decompose the entire manipulation configuration space $\mathcal{C}$ into four parts:

$$q = (q_{base}, q_{arm}, q_{gripper}, q_{target}) \tag{3.1}$$

where the gripper is responsible for interacting with the object, the arm is responsible for precisely moving the gripper to its desired orientation, and the base is responsible for moving the arm and gripper around the environment (Figure 3.3). For robots moving on a flat floor, q_{base} is 3 DOF and represents the translation and in-plane rotation. If the target is attached to the environment and has a door, then q_{target} is the angle of the door; otherwise, q_{target} represents the workspace of the target. q_{arm} and $q_{gripper}$ are the joint values. Usually the gripper is treated as in independent articulated body, whose transformation is the referred to as the end-effector of the arm. Even before defining the goal of the plan, the decomposition of the space immediately raises questions about:

- what are all inter-relationships between the four components,
- what is the order in which the four components should be searched, and
- how these spaces connect to the real world execution of the plan.

For example, the forward kinematics relationship between the arm and gripper is clear, but it is common for planners to use inverse relationships: given the gripper position, compute the configuration q_{arm} placing the gripper there.

If planning for mobile manipulation, then the question can turn out to involve all components: given a target location q_{target}, what are all possible configurations q_{base} such that the target is graspable.

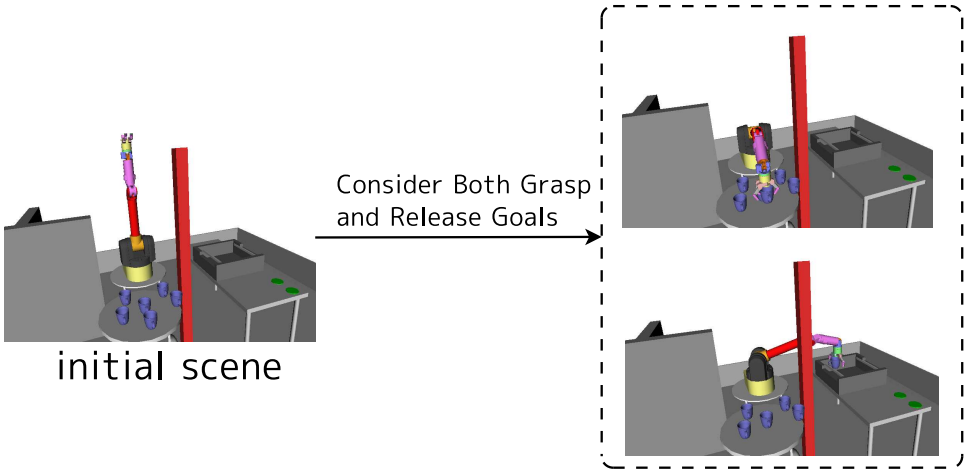


Figure 3.4: Grasp planning involving just the arm and gripper can sometimes require both the grasp and release goals to be considered before a grasp is chosen..

Because the search space is very big, it becomes very difficult to guarantee an optimal solution; therefore samplers, heuristics, and hierarchical subdivision of the problem have great influence over the quality and computability of a solution. For example when finding a feasible path to a graspable location, should the robot base be sampled first, which affects the grasps that can be achieved, or should the grasp be sampled first, which affects the base placements? If the goal is also to place the grabbed target to a specified location as shown in Figure 3.4, then it becomes necessary to consider the feasibility of the path and goal configuration of the release process so that a robot doesn't choose a bad grasp to begin with only to discover that it cannot achieve its final goal. Instead of delving into the search complexities of planning horizons as in [Stilman et al (2007b)], we perform most of the computation inside the goal-configuration samplers, which use the inter-relations of the configuration space to make the most informed decisions. The theory and generation of these relations are covered in the robot knowledge database.

3.2 Planning to a Goal Space

A planning algorithm takes an initial robot configuration and attempts to find a path to a goal-satisfying configuration while maintaining constraints. These constraints define a subset of the configuration space $\mathcal{C} = \{q\}$ that represents the set of feasible and free configurations, commonly denoted by $\mathcal{C}_{\text{free}}$. The goal satisfying configurations can be similarly denoted by a sub-space $\mathcal{C}_{\text{goal}} \in \mathcal{C}_{\text{free}}$. Because $\mathcal{C}_{\text{free}}$ and $\mathcal{C}_{\text{goal}}$ can be very complex, most of the planning literature focuses on how to efficiently represent and search this space for a solution path.

Algorithm 3.1: RRTConnect*(q_{init})

```
/* ρ ∈ [0,1] - uniform random variable                                */
1  Init(T_a, q_init); Init(T_g, Sample(C_goal)); T_b ← T_g
2  for iteration = 1 to N do
3      if ρ ≤ GoalSampleProbability() then
4          AddRoot(T_g, Sample(C_goal))
5      q_new ← Extend(T_a, Sample(C_free))
6      if q_new ≠ ∅ then
7          if Connect(T_b, q_new) then
8              return Path(T_a, T_b)
9      Swap(T_a, T_b)
10 end
11 return ∅
```

It is also possible to fit dynamics and time-sensitive information into this definition [Harada et al (2008); Kuffner et al (2003); Hauser et al (2008); Stilman et al (2007a); Berenson et al (2009a)].

Although, several works have shown the possibility of planning without sampling explicit goal configurations [Bertram et al (2006); Diankov and Kuffner (2007)] by using gradients and cost functions, the planning process becomes much slower thus making it a worthwhile effort to develop explicit goal configuration generators. Traditionally, planning algorithms have assumed $\mathcal{C}_{\text{goal}}$ is either one specific goal configuration, or a set of configurations with a distance threshold defining how close to the goal a robot should get in order to terminate with success. However, many researchers cite the possibility of $\mathcal{C}_{\text{goal}}$ being so complex that the only way to accurate use it in a planner is by random sampling. For example, the goal of manipulation planning is to have the gripper interact with the target object, meaning the goals are specified in the workspace of the gripper and not directly in the configuration space being searched for. It is this type of discrepancy that motivates a periodic random sampling of goals during the search process.

RRTConnect* (Algorithm 3.1) shows the modifications to the RRT-Connect algorithm [Kuffner and LaValle (2000)] that samples goals. The algorithm maintains a tree of configurations, with edges representing the ability for the robot to move from one configuration to the other. In every step the RRT randomly grows trees simultaneously from both the initial and goal configurations by first sampling a random configuration and then extending both trees to it using a step size ϵ. After every extension operation, the planner checks if the two

Algorithm 3.2: SAMPLEWITHCONSTRAINTS($\mathcal{C}$)

```
1 while true do
2     q ← PROJECTCONSTRAINTS(f_C,SAMPLE(C))
3     if q ∈ C then
4         return q
5 end
```

Algorithm 3.3: EXTENDWITHCONSTRAINTS($\mathcal{T}$,q)

```
  /* Uses definitions from [Kuffner and LaValle (2000)]        */
1 q_near ← NEARESTNEIGHBOR(q,T)
2 q_new ← PROJECTCONSTRAINTS(f_C,NEWCONFIG(q,q_near))
3 if q_new ∈ C_free then
4     ADDVERTEX(T,q_new)
5     ADDEDGE(T,q_near,q_new)
6     return q_new
7 return ∅
```

trees can be connected and a path is computed. With probability GOALSAMPLEPROBABILITY(), a goal configuration is sampled and inserted into the goal tree. Planning with this modification is commonly done when the goal space is complex, with varying names given to the algorithm depending on how the SAMPLE function is defined [Stilman et al (2007b); Brock et al (2008); Berenson et al (2009b); Diankov et al (2009)]. It should be noted that GOALSAMPLEPROBABILITY() can change with respect to how many goals have already been sampled and how well the planner is progressing, a constant probability might not be the best decision. The resulting paths from an RRT planners are notorious for being jagged and unnecessarily long. Almost all researchers add a *path-smoothing* phase as a post-processing step to the resulting paths. The linear short-cut method of joining two points on the path with a straight segment is the most commonly used. Recent results show that it is possible to also apply this to piecewise-quadratics [Hauser and Ng-Thow-Hing (2010)].

Some problems require the motion of the robot or the object to follow specific geometric and kinematic constraints. Although the simplest method of sampling valid configurations is to reject those that do not meet the constraints, it takes a long time. If the constraints are parameterizable by a function $f_C(q) = 0$, then gradient descent using $\frac{\delta f_C}{\delta q}$ allows projection of any invalid configuration q to a valid configuration. Such projection operators can be inserted in planners to allow a robot to open doors [Kojima et al (2008)], maintaining orientation of

target objects [Stilman (2007)], slide objects along a surface, or compliant whole-body humanoid control [Sentis (2007)]. Although we will not delve further into projection operators and trade-offs with sampling, we do go over two modifications required for RRTs to work with constraints. The first is the sampling function that uses a projection operator defined in SAMPLEWITHCONSTRAINTS (Algorithm 3.2). Because the projection does not take into account the entire free space of the problem, we check that the configuration is still in $\mathcal{C}$. The second modification is in the extend operation defined in EXTENDWITHCONSTRAINTS (Algorithm 3.3). Before any configuration can be added to the tree, it always makes an explicit check on the free space.

Using this structure, there are four components that play a key role in the search algorithms we discuss:

- A sampler on the feasible configuration space SAMPLE($\mathcal{C}_{\text{free}}$). The free space describes where the system can safely go, but it doesn't provide any information about its neighboring locations. By defining a state space cost function, we can use it to prioritize sampling of regions that will most likely contribute in finding the path.
- A distance metric $\delta(q_0, q_1)$ for $\mathcal{C}$ used to methodically discretize and step through the configuration space. Distance metrics are commonly used for two purposes: finding nearest neighbors and finding the appropriate step resolution for RRT extensions and line collision checking [Cameron (1985); Schwarzer et al (2003)]. We present an automated way of setting distance metrics in Section 4.6.3.
- A sampler on the goal space SAMPLE($\mathcal{C}_{\text{goal}}$). Defining goals usually involves the workspace and the reachability properties of the robot. We present a generalized manipulation sampler in Section 3.1 involving both grasping locations and the robot's arm and base.
- A parameterized constraint function f_C that allows the robot to continuously move from one configuration to another. A PROJECTCONSTRAINTS(f_C,q) function modifies the input configuration to the closest configuration that meets all the constraints.

Of the four, we primarily concentrate on goal samplers SAMPLE($\mathcal{C}_{\text{goal}}$) for the target task types described in Section 2.4.

3.3 Planning to a Grasp

The key to successful manipulation planning is to choose a grasping strategy and move to observe the target of the task so that execution is informed of the surroundings. Reaching and grasping the target object requires multiple pieces of information, including: knowledge

of the robot kinematics, robot geometry, the gripper capabilities, the grasping strategy of the target, and the environment obstacles that must be avoided. For example, the grasping strategy can be manually specified through *task frames* [Prats et al (2007a,b)] or *grasp handles* [Gravot et al (2006); Okada et al (2004)], or can be automatically constructed by considering the geometry of the gripper/target pair and searching for a stable grasp [Ciocarlie et al (2007); Goldfeder et al (2007); Berenson et al (2007)] or a caged grasp [Sudsang et al (1997); Diankov et al (2008b)]. These grasps can be parameterized by simple linear models [Prats et al (2008b)] or by storing all instances of successful grasps in a set [Berenson et al (2007); Diankov et al (2008a)] and planning with them simultaneously. In this section we cover the theory on how to use grasp sets in the context of the entire plan, the computation of grasps from a task description is covered in Section 4.2.

We define a *grasp* as:

- transformation of the gripper $T_{gripper}^{target}$ in the target's coordinate system,
- starting joint values[1], $q_{gripper}$
- direction of approach $v_{gripper} = \begin{bmatrix} v_x & v_y & v_z & 0 \end{bmatrix}^T$ in the target's coordinate system, and
- minimum distance along negative direction to get out of collision $\delta_{gripper}$.

We represent the space of all stable grasps as $\mathcal{G}$, which includes the 6D pose of the end-effector in $SE(3)$, the grasp preshape, and the approach direction. From that space, we extract a discretized ordered set $\mathcal{G}_{stable}$ that represents all stable grasps of the object. $\mathcal{G}_{stable}$ is used for finding goal configurations of the end effector and only concerns itself with how the gripper links interact with the target object. At any target configuration q_{target}, we denote the grasp set in the world frame by

$$T^{world}(q_{target}) \circ \mathcal{G} \tag{3.2}$$

where the gripper transform and direction of approach are transformed by $T_{target}^{world}(q_{target})$. The world grasp set allows inverse kinematics and other robot maps to be applied to the object.

Pre-ordering grasps in the set is one way of prioritizing which grasps get tested first; however, we should make it clear that the grasp order is only dependent on stability and not on the environment or the robot arm/base. Potential ordering methods are discussed in Section 4.2.

Before a grasp from $\mathcal{G}$ can be determined valid in the current environment, it needs to be collision-free from other obstacles, it needs to be tested for successful completion of the

[1] Commonly referred to as the gripper *preshape*.

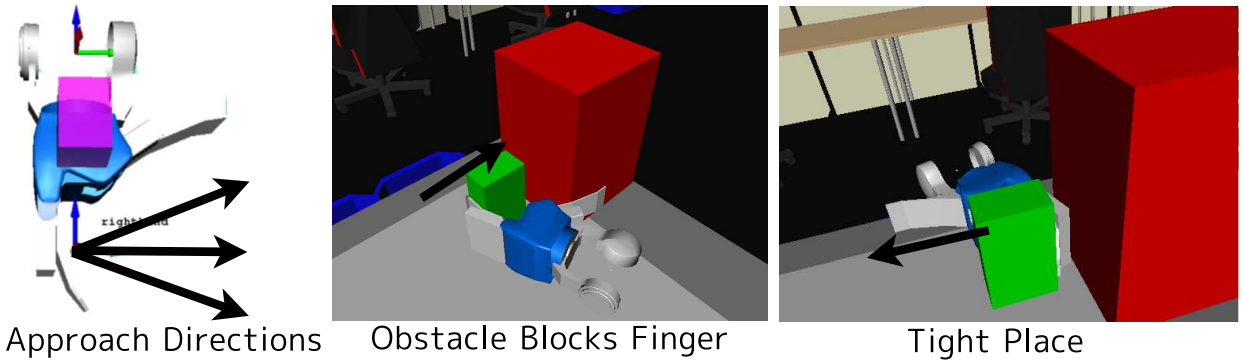


Figure 3.5: Shows examples of collision-free, stable grasps that are invalid in the environment.

Algorithm 3.4: GRASPVALIDATOR($T^{world}_{gripper}$,q,v^{world},δ)

```
1  SETCONFIGURATION(gripper,q)
   /* Check collision with gripper along negative direction.   */
2  GripperTransforms ← ∅
3  for d ∈ [δ, δ + ε] do
4      T ← [ I_3x3   −dv^world ] T^world_gripper
           [   0         1    ]
5      SETTRANSFORM(gripper,T)
6      if CHECKCOLLISION(gripper) then
7          return ∅
8      ADD(GripperTransforms,T)
9  end
10 SETTRANSFORM(gripper, T^world_gripper)
11 q_new ← GRASPSTRATEGY()
12 SETCONFIGURATION(gripper,q_new)
13 if ALLCOLLISIONS(gripper) ⊆ {target} then
14     return GripperTransforms
15 return ∅
```

grasp strategy GRASPSTRATEGY($q_{gripper}$)), and it needs to have small room to maneuver along its direction of approach. Figure 3.5 shows examples of collision-free grasps that are not valid and all must be pruned.

All grasps in $\mathcal{G}_{stable}$ are validated using the GRASPVALIDATOR function as described in GRASPVALIDATOR (Algorithm 3.4). First the gripper is set to its preshape and checked for

environment collisions as it moves along its negative direction of approach. ϵ is proportional to the error expected on the object's pose and the robot execution[2]. Then the gripper is set to the final transformation and the grasp strategy is executed in simulation. Because the grasp strategy guarantees stability only if the gripper's contacts are from the target object, we check that all collisions with the final gripper configuration are only with the target using AllCollisions and the space inclusion operator $\subseteq$. It is possible to just check the gripper collisions since we know its joint values q_{new} and its workspace transform $T^{world}_{gripper}$. On success, GraspValidator returns the set of gripper transformations that are collision free. When planning for just the arm, these transforms are later checked for the existence of inverse kinematics solutions.

In order to maintain clarity, we assume that every function presented saves the **state** of all objects it interacts with and restores it upon returning. The **state** in this case is each link's position, orientation, and collision-enabled values.

The simplest form of grasp planning is when the configuration space is $\{q_{arm}, q_{gripper}\}$, and inverse kinematics can be directly applied to find the goal configurations. The first goal sampling algorithms using grasp sets is presented in GoalSampler_Grasps_Arm (Algorithm 3.5). It first samples a grasp from $\mathcal{G}_{stable}$ and validates it with the current environment. The set of transforms returned by GraspValidator are all checked for inverse kinematics solutions and that they entire robot configuration is in the free space. If there is a grasp transform that does not have a solution, then the robot will not be able to complete the task and the grasp is skipped. Otherwise, the transform $\delta + \epsilon$ distance away from the original grasp is taken as the goal to feed to the RRT planner. This distance is *extremely* important in making the planning robust to errors when executing on a real robot.

Given a target end-effector transformation, we define the set of robot configurations q_{arm} that achieve it with

$$IK(T) = \{\, q_{arm} \,|\, T = FK(q_{arm}) \,\} \tag{3.3}$$

where $FK(q_{arm})$ is the forward kinematics transforming robot configuration. For each inverse kinematics solution produced by $IK(T)$, we check it for the free space and return it the arm and gripper configuration to the planner. The next time GoalSampler_Grasps_Arm is called, it should resume from the point it left off instead of starting from the same grasp again. In practice, this programming construct can be easily implemented with **coroutines** [Knuth (1973)] or Python **generators** without changing the function.

Using the environment in Figure 3.4, the size of the Barrett Hand/mug grasp set is 341. The time it takes to sample one valid goal configuration is $\mathbf{0.04 \pm 0.01s}$. It is actually very

[2]Usually set between 1cm and 2cm

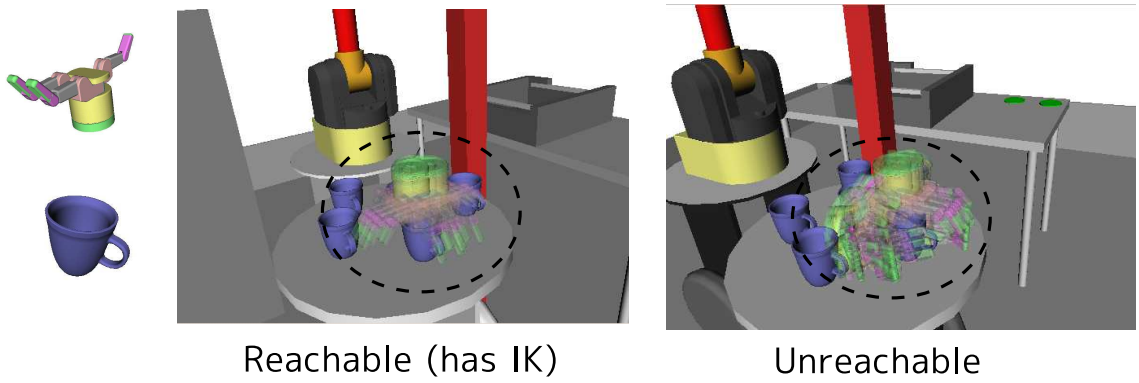


Figure 3.6: Divides the grasps that pass GRASPVALIDATOR into the set of reachable (has inverse kinematics solutions) and unreachable grasps for one target.

Algorithm 3.5: GOALSAMPLER_GRASPS_ARM()

```
1  SETCONFIGURATION(gripper,q)
2  for (target, T_gripper^target, q_gripper, v^target, δ) ∈ G_stable do
3      GripperTransforms ← GRASPVALIDATOR(T_target^world T_gripper^target, q_gripper, T_target^world v^target, δ)
       /* Validate all gripper transforms.                                  */
4      if ∀_{T∈GripperTransforms}, ∃ q_arm s.t. q_arm ∈ IK(T) ⋀(q, q_gripper) ∈ C_free then
5          T ← LASTELEMENT(GripperTransforms)
6          for q_arm ∈ IK(T) do
7              if (q_arm, q_gripper) ∈ C_free then
8                  return (q_arm, q_gripper)
9          end
10 end
```

common for the GRASPVALIDATOR to return grasps that are completely unreachable by the robot. Figure 3.6 shows an object that is slightly far away from the robot, of the 120 grasps that are accepted by the GRASPVALIDATOR, only **11%** have inverse kinematics solutions. The entire planning phase to the initial pickup is $\mathbf{0.7 \pm 0.1s}$.

As was mentioned previously, it is also necessary to consider the possible destinations of the target during the grasp selection process. Every possible target destination has to be checked for possible collisions with the target object, gripper, and arm. GRASPVALIDATOR_WITHDESTINATIONS (Algorithm 3.6) shows the entire grasp validation process. The destination check requires the target to be **moved temporarily** to that location in order to accurately simulate the expected collisions. Collisions with the robot are ignored since

Algorithm 3.6: GRASPVALIDATOR_WITHDESTINATIONS(**target**,$T^{world}_{gripper}$,q,v^{world},δ)

```
GripperTransforms ← GRASPVALIDATOR(T^world_gripper,q,v^world,δ)
if GripperTransforms ≠ ∅ then
    for T_target,new ∈ TargetDestinations do
        SETTRANSFORM(target,T_target,new)
        if ALLCOLLISIONS(target) ⊆{robot} then
            T^world_gripper,new ← T_target,new T^-1_target T^world_gripper
            if ∃ q_arm s.t. q_arm ∈ IK(T^world_gripper,new) ⋀(q_arm, q) ∈ C_free then
                /* IK exists and target is collision-free.        */
                return GripperTransforms
    end
return ∅
```

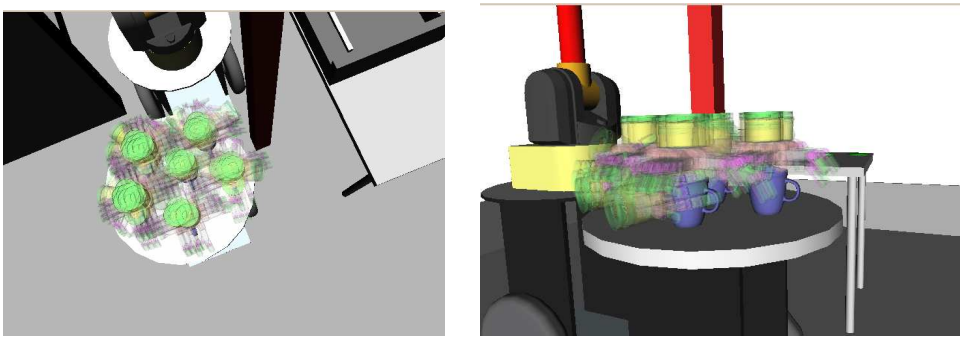


Figure 3.7: Shows all the valid, reachable grasps that can be sampled from simultaneously. The grasp set size is 341 and there's 6 target objects for a total of 136 valid grasps.

the correct robot configuration is not set at that point. It should be noted that the inverse kinematics check is only valid if just planning for the arm. When planning for the base, it becomes very difficult to consider robot reachability, so these types of checks are handled elsewhere as discussed in Section 3.6. Using the environment in Figure 3.4 where there is 102 potential target destinations, the average time to sample a configuration is $\mathbf{0.2 \pm 0.1s}$.

The goal sampler itself can easily consider multiple targets simultaneously by creating a super-grasp set as shown in Figure 3.7.

Even with such a straightforward goal sampler, we can immediately start seeing potential bottlenecks that could slow down planning. For example, the set of grasps might be too big (more than 1000+ grasps), so need to prioritize or sample them well so valid grasps are tested first. We have to be careful that the prioritization function doesn't require a lot of

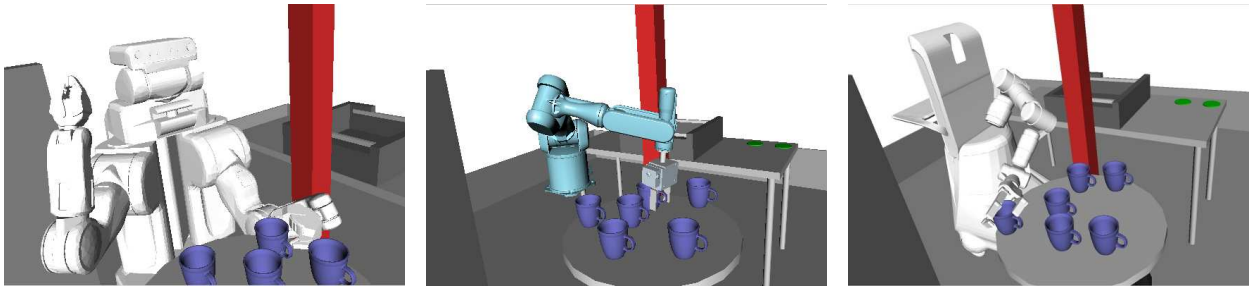


Figure 3.8: Several robot arms performing grasp planning.

computation. Another common problem is that the GraspValidator does not consider inverse kinematics, but some grasps are obviously not reachable as was shown in Figure 3.6. In [Berenson et al (2007)], we used an ad hoc method to determine if a grasp is reachable, however a much quicker and more formal method is using the reachability space of the robot to prune grasps (Section 4.3).

Figure 3.8 shows other robots performing the same grasp planning task with all parameters auto-generated from the robot knowledge-base.

3.4 Planning with Nonlinear Grasping Constraints

For a large class of constrained tasks that deal with objects part of the environment like door opening, the robot end-effector does not have to be rigidly attached to the object throughout the entire motion. In fact, as long as the object is *caged* [Rimon and Blake (1986); Rimon (1999)] by the end-effector, moving the end-effector can produce a corresponding motion of the object. In other words, there exists a *grasp space* the end-effector can reside in such that the target object desired motion can be achieved, while greatly increasing $\mathcal{C}_{\text{free}}$ providing the robot motions that were not possible before. We present an algorithm that can effectively change grasps while attempting to move a constrained object as shown in Figure 3.9.

In [Diankov et al (2008b)], we presented a novel motion planning formulation for performing constrained tasks such as opening doors and drawers. Previous work on constrained manipulation transfers rigid constraints imposed by the target object motion directly into the robot configuration space. This often unnecessarily restricts the allowable robot motion, which can prevent the robot from performing even simple tasks, particularly if the robot has limited reachability or low number of joints. Our method computes *caging grasps* specific to the object and uses efficient search algorithms to produce motion plans that satisfy the task constraints. This method can significantly increase the range of possible motions of

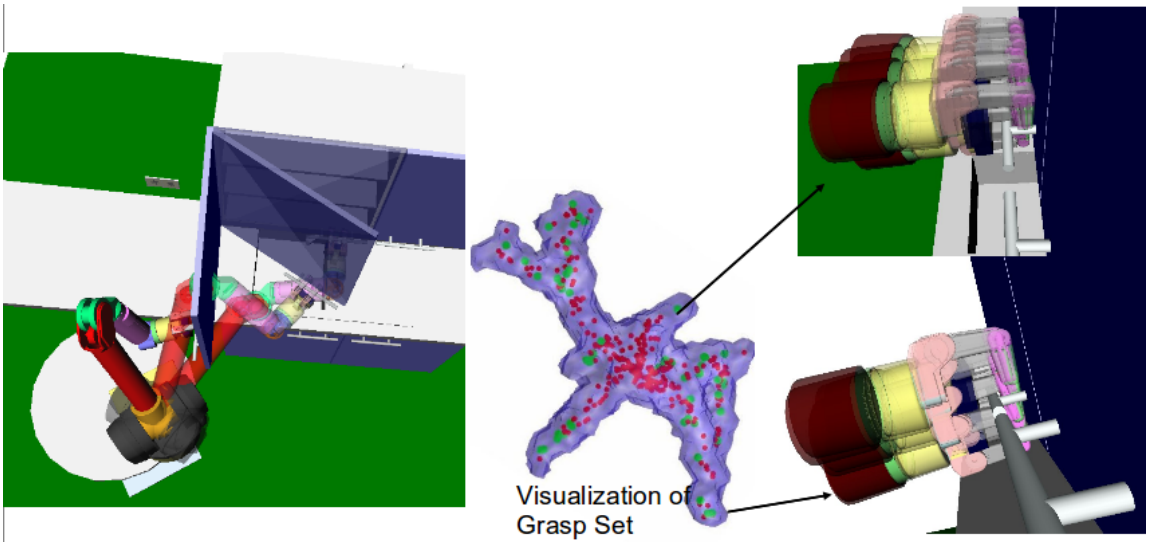


Figure 3.9: A robot hand opening a cupboard by caging the handle. Space of all possible caging grasps (blue) is sampled (red) along with the contact grasps (green).

the robot by not having to enforce rigid constraints between the end-effector and the target object. We illustrate our approach with experimental results and examples running on two robot platforms.

The expanded range of motion comes at the cost of algorithmic complexity. In the absence of a rigid grasp, care must be taken to ensure that the object does not slip out of the robot end-effector. Of greater concern is the fact that there no longer exists a one-to-one mapping from robot motion to object motion: since the object is loosely caged, there can exist end-effector motions that produce no object motion, and object motion without explicit end-effector motion.

3.4.1 Relaxed Formulation

We formulate the problem using the configuration space of the robot arm $q_{arm} \in \mathcal{Q}_{arm}$, the configuration space of the end-effector $g \in \mathcal{G}$, and the configuration of the constrained target object $q_{target} \in \mathcal{R}$. Each of these spaces is endowed with its corresponding distance metric $d : \mathcal{X} \times \mathcal{X} \rightarrow \mathbb{R}$. As in the previous sections, g contains the 6D pose of the end-effector in $SE(3)$. Although our presented method is general enough to include the gripper preshape in the grasp configuration, for this analysis we assume that the preshape $q_{gripper}$ is fixed.

In the relaxed task constraint formulation, each target object is endowed with a task frame which is rigidly attached to it, and a set of grasps G represented in that task frame. The set G is carefully chosen to ensure that any grasp is guaranteed to *cage* the object. If

we define $\mathcal{R}_g$ to be the set of target configurations reachable under a grasp g, then the target at configuration q_{target} is caged by the robot if $q_{target} \in \mathcal{R}_g \subset \mathcal{R}$ and every point on the the boundary of $\mathcal{R}_g$ is in collision with the end-effector at pose g. Although this is a conservative definition of a cage, it is necessary because end-effector is the only physical body known with certainty and caging should be environment independent.

In congruence with the traditional task constraint formulation, we describe the pose of grasps in G with respect to a coordinate frame that is rigidly attached to the object, termed the *task frame*. A transform $T_{q_{target}}$ relates the task frame at an object configuration q_{target} to the world reference frame. The utility of this representation arises from the observation that, under a rigid grasp, the pose of the end-effector is invariant in the task frame. This allows us to compute and cache $\mathcal{G}_{stable}$ offline, thereby improving the efficiency of the online search. Similar to Equation 3.2,we denote the new grasp transformation of a given target configuration q_{target} as:

$$T_{q_{target}} G = \{\, T_{q_{target}} g \mid g \in G \,\}. \tag{3.4}$$

One of the relaxed planning assumptions is that the end-effector of any configuration of the robot always lies within the grasp set G with respect to the task frame. Because this couples the motion of both the object and the robot during manipulation, their configurations need to be considered simultaneously. Therefore, we define the relaxed configuration space $\mathcal{C}$ as

$$\mathcal{C} = \{(q_{arm}, q_{target}) \mid q_{target} \in \mathcal{R}, q_{arm} \in \mathcal{Q}_{arm}, FK(q_{arm}) \in T_{q_{target}} G\} \tag{3.5}$$

In this case, $\mathcal{C}_{\text{free}}$ includes $(q_{target}, q_{arm}, q_{gripper})$. Given these definitions, the relaxed task constraint problem becomes:

> *Given start and goal configurations q_{target}^{start} and q_{target}^{goal} of the object, compute a continuous path $\{q_{target}(s), q_{arm}(s)\}$, $s \in [0,1]$ such that*

$$q_{target}(0) = q_{target}^{start} \tag{3.6}$$

$$q_{target}(1) = q_{target}^{goal} \tag{3.7}$$

$$\{q_{target}(s), q_{arm}(s)\} \in \mathcal{C}_{\text{free}} \tag{3.8}$$

$$FK(q(1)) \in T_{q_{target}(1)} G_{contact} \tag{3.9}$$

Equation 3.6 and Equation 3.7 ensure that the target object's path starts and ends at the desired configurations. Equation 3.8 forms the crux of the relaxed task constraint planning problem. Because the caging criteria dictates that each grasp be in the grasp set

G, Equation 3.8 ensures that any robot configuration $q(s)$ produces a grasp $FK(q(s))$ that lies in the world transformed grasp set $T_{q_{target}(s)}G$. Equation 3.9 constrains the final grasp to be within a contact grasp set $G_{contact} \subseteq G$. This set is formally defined in Section 4.2. Informally, any grasp in this set is in contact with the object and guarantees that the object will not move away from the goal.

While the above equations describe the geometry of the problem, we make several assumptions about the physics of the problem. These assumptions constrain the automatically generated grasps we use for planning as well as the motion of the robot and object during manipulation. Our analysis is purely quasi-static. The robot moves slow enough that its dynamics are negligible. Furthermore, we assume that the object's motion is quasi-static as well. This can be achieved in practice by adding a dash pot to the hinges, damping their motion, or by a sufficient amount of friction in the case of an object being dragged across a surface. We also assume that we have access to a compliant controller on the robot. Under this assumption, we are guaranteed that the robot will not jam or exert very large forces on the object being manipulated. We discuss the generation of the caging grasp set in Section 4.2.2.

In the following sections we discuss the planning algorithms and robot results assuming a grasp set has been generated. We describe two planning algorithms to solve the relaxed constraint problem: a discretized version and a randomized version. The randomized algorithm is more flexible and makes less assumptions about the problem statement, however the discretized algorithm is simple to implement and useful for explaining the concepts behind relaxed planning (as well as the motivation for a randomized algorithm).

3.4.2 Discretized Algorithm Formulation

The underlying assumption of the discrete formulation is that a desired path of the target object is specified. Specifying the path in the object's configuration space as an input to the planner is trivial for highly constrained objects like doors, handles, cabinets, and levers. The configuration space of these objects is one dimensional, so specifying a path from a to b is easily done by discretizing that path into n points. In the more general case where an object's configuration space can be more complex, we denote its desired path as $\{q_{target,i}\}|_1^n$ where each of the configurations $q_{target,i}$ have to be visited by the object in that order.

The discrete relaxed constrained problem is then stated as: given a discretized object configuration space path $\{q_{target,i}\}|_1^n$, find a corresponding robot configuration space path

Algorithm 3.7: $Q \leftarrow$ DiscreteSearch()

```
for i = 1 to n − 1 do
    G_i ← T_{q_target,i} G
    for g ∈ G_i do
        if (IK_{i,g} ← IK(g)) ≠ ∅ then
            break
        G_i.remove(g)
    end
    if G_i = ∅ then
        return ∅
end
for g ∈ T_{q_target,n} G_contact do
    for q_arm ∈ IK(g) do
        Q_next ← DiscreteDepthFirst(q_arm, n − 1)
        if Q_next ≠ ∅ then
            return { Q_next, q_arm }
    end
end
return ∅
```

$\{q_i\}|_1^n$ such that

$$\forall_{1 \leq i \leq n} \; (q_{target,i}, q_i) \in \mathcal{C}_{\text{free}} \tag{3.10}$$

$$FK(q_n) \in T_{q_{target,n}} G_{contact} \tag{3.11}$$

$$\forall_{1 < i \leq n} \; d(FK(q_{i-1}), FK(q_i)) < \epsilon_1 \tag{3.12}$$

$$\forall_{1 < i \leq n} \; d(q_{i-1}, q_i) < \epsilon_2 \tag{3.13}$$

where Equation 3.10 and Equation 3.11 constrain the end-effector to lie in the current grasp set defined for the object and Equation 3.11 guarantees the final grasp is in contact. To satisfy the continuity constraint on the robot configuration space path, Equation 3.12 and Equation 3.13 ensure that adjacent robot and grasp configurations are close to each other.

A straightforward discrete planning approach to solve this problem is provided in DiscreteSearch (Algorithm 3.7). We begin by first running a feasibility test through the entire object trajectory. This step is also used to initialize the grasp and kinematics structures used for caching. We assume an inverse kinematics solver is present for every arm. Furthermore, if the arm is redundant the solver will return all solutions within a discretiza-

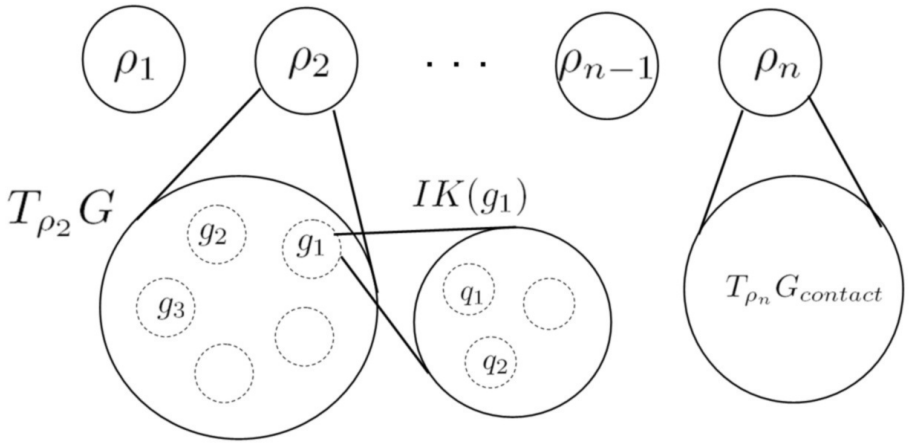


Figure 3.10: The basic framework used for planning discrete paths $\{q_i\}|_1^n$ in robot configuration space to satisfy paths $\{q_{target,i}\}|_1^n$ in object configuration space.

tion level. We compute the set of contact grasps that will keep the object in form-closure at its desired final destination $q_{target,n}$ (line 3.7). For each grasp in this set we compute IK solutions for the complete configuration of the robot, and for each IK solution we attempt to plan a path through configuration space that tracks the object path $\{q_{target,i}\}$ using depth first search (line 3.7)[3].

3.10 provides a diagram of the discrete search framework. Given an object path $\{q_{target,i}\}|_1^n$ we search for a robot path $\{q_i\}|_1^n$ that consists of a sequence of robot configurations $q_i, 1 \leq i < n$ such that $FK(q_i) \in T_{q_{target,i}} G$ and $FK(q_n) \in T_{q_{target,n}} G_{contact}$. Each of these configurations q_i is generated as an IK solution from one of the grasps in the grasp set $T_{q_{target,i}} G$. The depth first search process takes a robot configuration at a time step j and calculates all the robot configurations that correspond to valid grasps at time $j-1$ (i.e. are members of set $T_{q_{target,j-1}} G$), then recursively processes each of these configurations until a solution is found.

3.4.3 Randomized Algorithm Formulation

There are several disadvantages to the discretized algorithm. First, it is highly dependent on the discretization level of the grasp set and IK solver. For robots with six degrees of freedom or less, enumerating all IK solutions isn't a problem. However, as soon as the joints

[3]This depth first search expands states in the same order as A* would using a heuristic function based on (an underestimate of) the target object distance to goal.

Algorithm 3.8: $\{q_{object,new},\ q_{new}\} \leftarrow \text{SAMPLENN}(q_{target},\ q_{arm})$

```
1  G ← ∅, q_new ← ∅
2  while q_new = ∅ do
3      q_object,new ← RANDOMCLOSECONFIG(q_target)
4      if not EXIST(G_{q_object,new}) then
5          G_{q_object,new} ← T_{q_object,new} G'
6      g_new ← SAMPLEWITHOUTREPLACEMENT(G_{q_object,new})
7      if g_new ≠ ∅ then
8          q_new ← SAMPLEIK(g_new, q)
9      else if CHECKTERMINATION() then
10         return {∅, ∅}
11 end
12 return {q_object,new, q_new}
```

increase or a mobile base is considered, the discretization required for $IK(g)$ to reasonably fill the null space grows exponentially. Second, the desired object trajectory is fixed, which eliminates the possibility of moving the door in one direction and then another to accomplish the task (see [Stilman et al (2007a)] for an example where this is required).

To overcome these limitations, we also applied a randomized planner to the problem. We chose the Randomized A* algorithm [Diankov and Kuffner (2007)], which operates in a similar fashion to A* except that it generates a random set of actions from each state visited instead of using a fixed set. Randomized A* is well suited to our current problem because it can use the target object distance to goal as a heuristic to focus its search, it can guarantee each state is visited at most once, it does not need to generate all the IK solutions for a given grasp, and it can return failure when no solution is possible. The key difference between Randomized A* and regular A* is the sampling function used to generate neighbors during the search. For our relaxed constraints problem the task of this sampling function is to select a random configuration $(q_{target,new}, q_{new})$ and a random grasp $g_{new} \in T_{q_{target,new}} G$ such that $q_{new} \in IK(g_{new})$. Ideally, this should be done efficiently without wasting time considering samples previously rejected for the same configuration. The A* criteria will ensure that the same configuration isn't re-visited and that there is progress made towards the goal, so the sampling function needs only return a random configuration in $\mathcal{C}_{\text{free}}$ around the current configuration (q_{target}, q_{arm}) as fast as possible.

SAMPLENN (Algorithm 3.8) provides the implementation of the sample function. It first samples a target object configuration $q_{object,new}$ close to the current configuration q_{target} (line

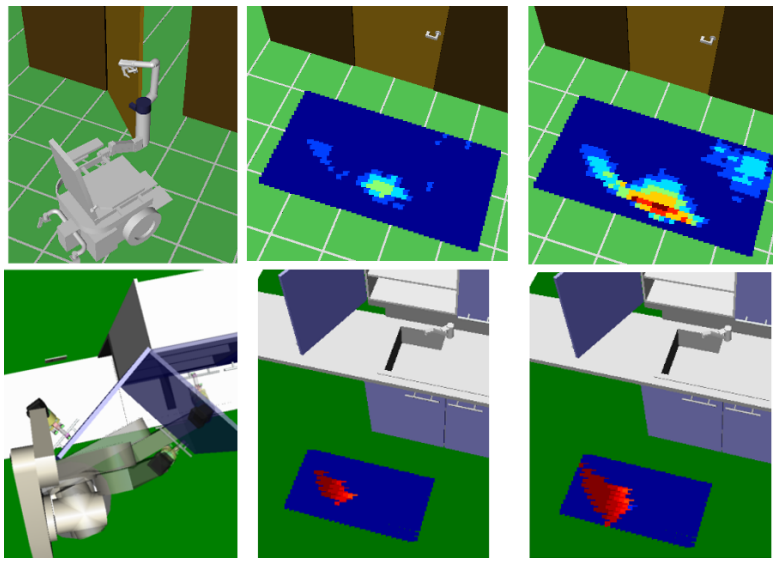


Figure 3.11: Comparison of fixed feasibility regions (left) and relaxed feasibility regions (right) for the Manus and Puma robot arms.

3), then searches for feasible grasps from the new grasp set $T_{q_{object,new}}G'$ (line 6), and then samples a collision-free IK solution close to q (line 8). In order to guarantee we sample the entire space, RandomCloseConfig should discretize the sampling space of the target configuration so that the number of distinct $q_{object,new}$ that are produced is small. This is necessary to ensure that sampling without replacement is efficient. Each time a sample is chosen (line 6), it is removed from $G_{q_{object,new}}$ so it is never considered again, an operation that takes constant time. If the target is close to its goal then G' is the contact grasp set $G_{contact}$, otherwise G' is the regular grasp set G. Once a grasp is found, SampleIK samples the null space of the IK solver around q until a collision-free solution is found. If not, the entire process repeats again. If all grasps are exhausted for a particular target configuration, the sampler checks for termination conditions and returns false (line 9).

3.4.4 Experimental Validation

Relaxing task constraints through caging grasps has enabled real-world implementations of constrained task execution using low DOF robots. We show experimental results on the 6DOF Manus Assistive Robot Service Manipulator [Exact-Dynamics-BV (1991-present)] and the 7DOF Barrett WAM [Barrett-Technologies (1990-present)], involving the tasks of autonomously pulling doors and cabinets open at arbitrary placements of the robot base. We compare our caged grasp planning approach to a traditional planner that enforces rigid

	Set Size	Discrete (Successes)	Discrete (Failures)	Randomized (Successes)	Randomized (Failures)
6DOF Manus Arm	550	0.235 s	0.234 s	0.143 s	0.23 s
6DOF Puma Arm	300	1.49 s	0.043 s	1.83 s	0.028 s

Table 3.1: Statistics for the scenes tested showing average planning times (in seconds) and size of the grasp sets used.

task constraints. The results shown in Figure 3.11 indicate that relaxing task constraints through caging grasps provide a much a greater motion envelope for the robot as well as versatility in base placement. This expanded range of allowable motions of the robot directly results in: 1) improvements in the efficiency and the success rate of planning for a variety of constrained tasks; 2) greater success in executing the desired motion and achieving the final object goal state.

	Discrete	Randomized
6DOF Manus Arm	441%	503%
6DOF Puma Arm	130%	126%

Table 3.2: Increase in Feasibility Space when using relaxed planning compared to fixed-grasp planning.

In each scene, the robot is randomly positioned and oriented on the floor, and then the planners are executed. Thousands of random positions are tested in each scene to calculate average running times (Table 3.1). The parameters for the randomized algorithm stayed the same across all robots. To show that relaxed grasp sets really do increase the regions the arm can achieve its task from, Figure 3.11 shows the feasibility regions produced with the relaxed grasp set method and the fixed grasp method. The fixed grasp method uses a single task-frame grasp throughout the entire search process. To make things fair, we try every grasp in G before declaring that the fixed grasp method fails. Table 3.2 shows how many times the feasibility region increased for the relaxed methods compared to the fixed method. As expected, the lower dimensional manipulators benefit greatly from relaxed task constraints. Furthermore, the door can be opened much further using the relaxed approach than with the fixed grasp method.

Figure 3.12 shows experiments with several robots in simulation to show the generality of the algorithm. Each grasp set was trained in a matter of minutes. Along with the analytic

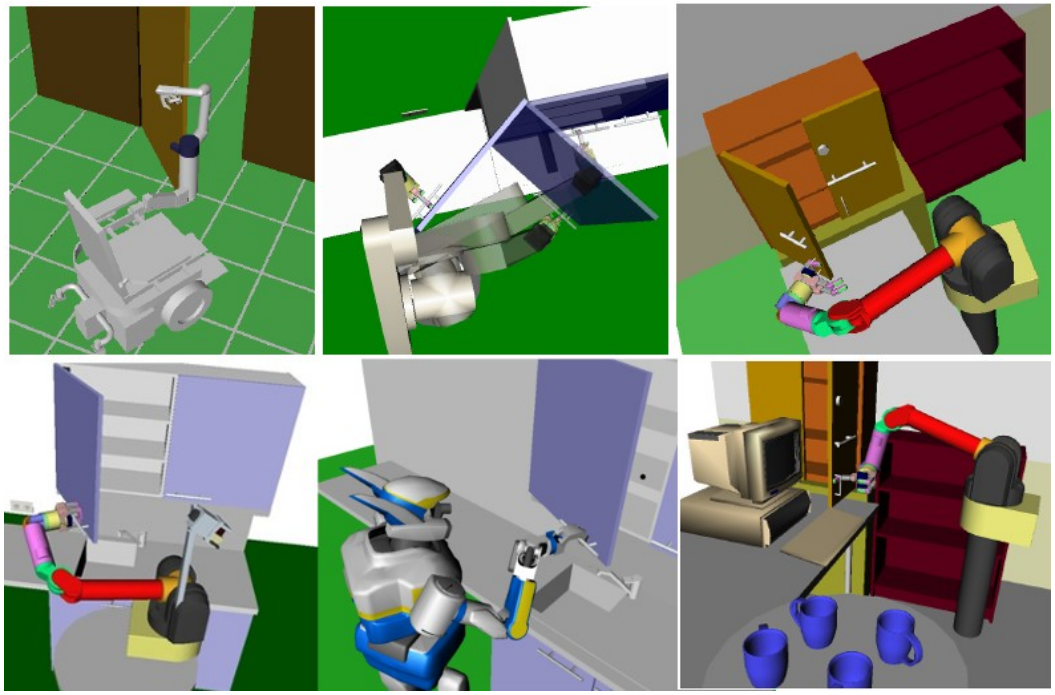


Figure 3.12: Example simulations using the relaxed grasp-set planning formulation.

inverse kinematics solvers presented in Section 4.1, we can immediately get each of these robots to open doors.

Figure 3.14 shows the grasp planning and caging grasp formulations combined to implement a more complex task. The task is to put a cup from the table on the right into the cupboard on the upper left corner. The Barrett WAM first opens the cabinet to a specified angle and then plans to grasp the cup while taking into account the geometric constraints of the cup's destination. Once the cup is placed inside the cupboard, the robot releases it while making sure its fingers do not collide with the obstacles. Then it plans to a safe position to move to its target preshape, and then closes the door.

Finally, it is possible to consider multiple preshapes and disjoint grasp sets to allow the robot to change grasping strategies. For example, consider the top example in Figure 3.13 where a mobile robot is attempting to open a refrigerator as wide as possible. Because the door handle makes a wide arc, it is impossible for the robot to open it all the way with just a hook grasp. Therefore, we add push and pull grasps to do caging grasp definition: instead of checking for caging in both directions, pushing and pulling requires that the object is constrained in only one direction. The algorithm using disjoint sets is simple. First, we randomly choose a disjoint grasp set and attempt to pull the door as wide as possible. When

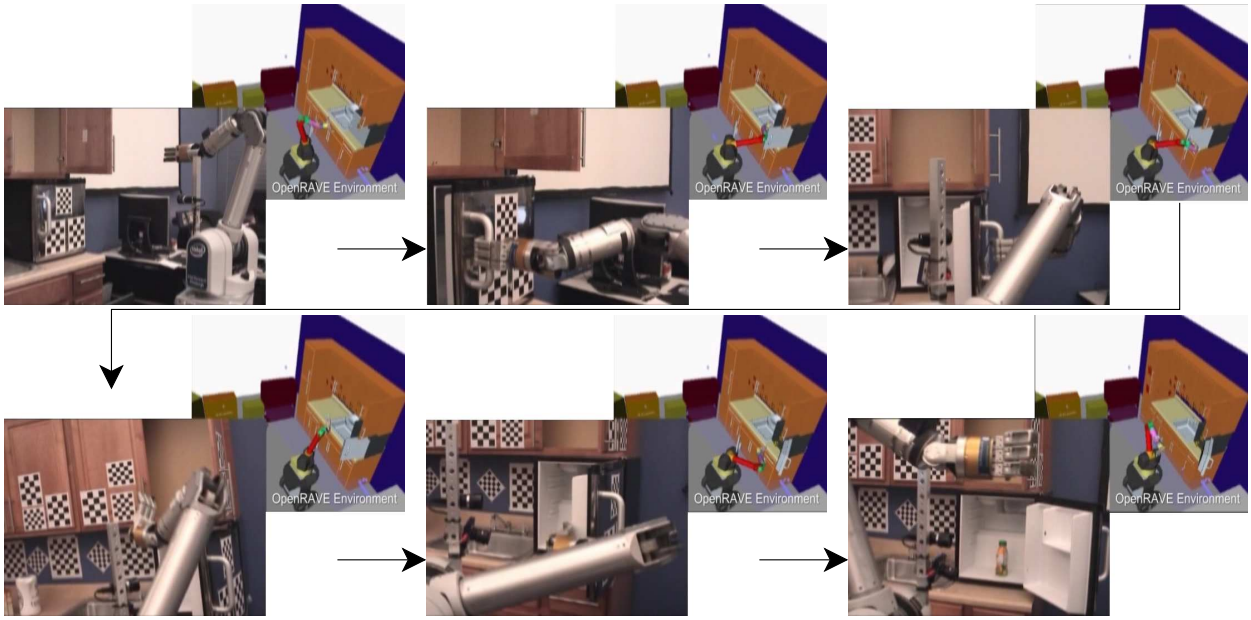


Figure 3.13: WAM arm mounted on a mobile base autonomously opening a cupboard and a fridge.

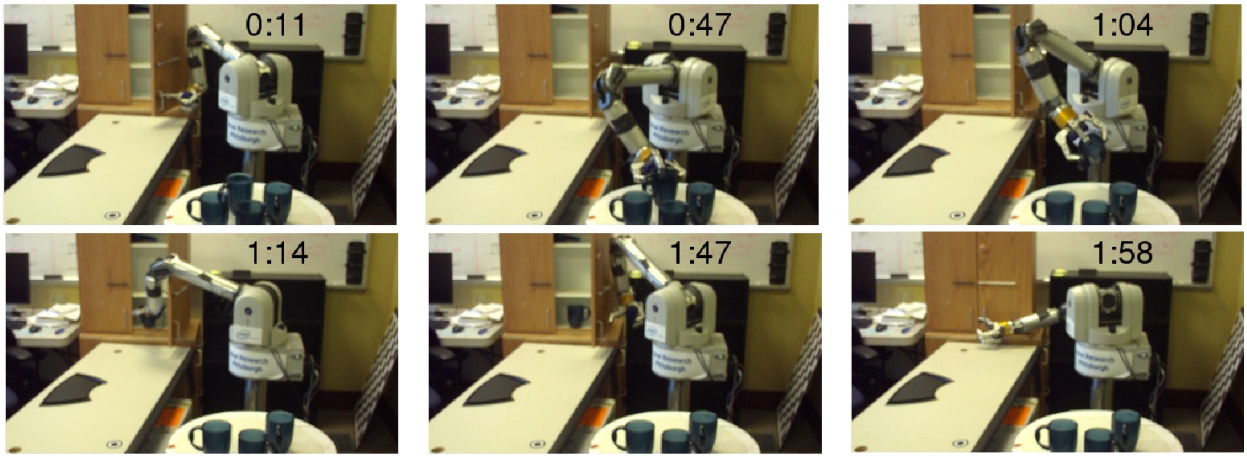


Figure 3.14: WAM arm autonomously opening a cupboard, putting in a cup, and closing it. Wall clock times from start of planning are shown in each frame.

the door cannot be pulled any further and it still isn't at its desired configuration, then we choose another disjoint set and continue.

Figure 3.13 shows three different tasks that have become possible with pushing/pulling and disjoint sets. Mobile bases usually have a ± 5 cm accuracy in getting to their goal destination; therefore, it becomes necessary to use the feasibility maps in Figure 3.11 to aim the base to a dense probability region to avoid planning failures. Being able to explicitly consider the feasibility regions can greatly help in designing the robot work space for industrial situations. Furthermore, allowing grasps to change while planning for tasks that do not require great precision can greatly increase the free space of the task. This increased space can be used to allow robots with lower number of joints to complete the same tasks as more complex, and expensive robots.

3.5 Planning with Free-Joints

Planning for a certain set of joints while treating the others as *free* is a sub-task that pops up very frequently in manipulation planning. When grasp planning, it is really not necessary to increase the planning configuration space to $\{q_{arm}, q_{gripper}\}$ for the final goal path. If there's only a few variations of the grasp preshapes $q_{gripper}$, then the robot can first plan its arm to a position where it can freely move its gripper to one of the preshapes as shown in Figure 3.15. After the robot achieves the preshape, the planner can just consider moving the arm to the goal where the planning configuration space is just $\{q_{arm}\}$. Combined with the tremendous reduction in the free space, there are also several benefits in moving the

hand separately from the arm when considering the real robot hardware. It is very common for the gripper and arm hardware systems in a robot to be controlled by separate real-time loops and possibly separate hardware. This makes it a little difficult to tightly synchronize trajectories that simultaneously move both the arm and the gripper, especially the path approaches obstacles. By guaranteeing the gripper will not move during an arm trajectory, we could relax the synchronization requirements on the hardware.

Algorithm 3.9: RRTExplore(q_{init})

```
1 Init(T, q_init)
2 for iteration = 1 to N do
3     q_free ← Extend(T_a, Sample(C_free))
4     if q_free ≠ ∅ then
5         if CheckGoalCondition(q_free) then
6             return Path(T, q_free)
7 end
8 return ∅
```

The *free-joints* problem is unique in that the robot has a small number of joints q_{fixed} that need to safely move in a desired path $\tau_{fixed}(t)$, but collisions can prevent them from moving there. In order to solve it, we treat the rest of the joints as *free* and the problem reduces into finding a robot configuration with the *free* joints where the *fixed* joints can safely move along $\tau_{fixed}(t)$. This problem is also applicable to robot joints not directly in q_{arm} and $q_{gripper}$ like moving the torso joints in humanoid robots (Figure 3.16). The search itself given in RRTExplore (Algorithm 3.9) just expands an RRT tree and checks if $\tau_{fixed}(t)$ is achievable after every extension operation.

The more interesting part is in the termination condition function given by CheckGoalCondition (Algorithm 3.10). The fixed joints are moved through all points in the

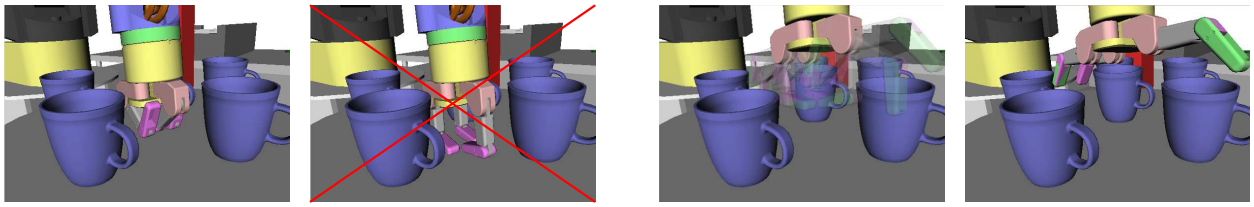


Figure 3.15: Shows a gripper whose fingers need to open, but cannot due to the table. Planning by setting the free joints to the arm can move the gripper to a safe location to open its fingers.

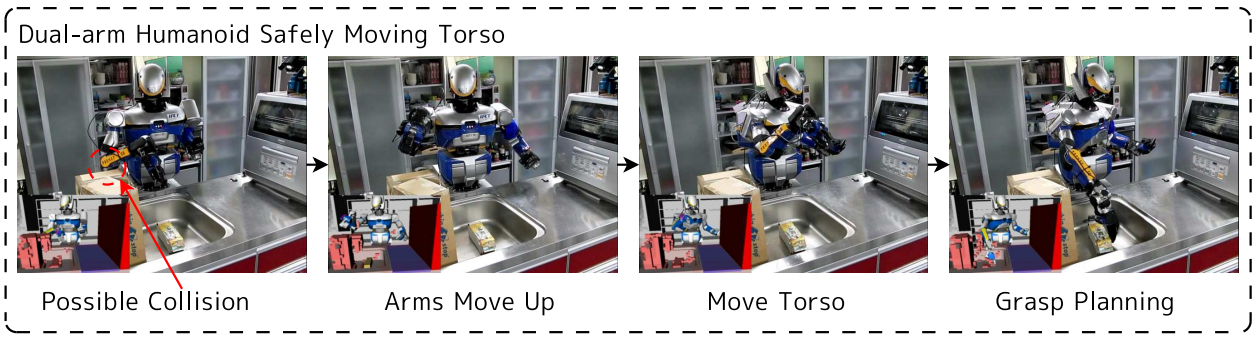


Figure 3.16: The planner is also applied to moving the torso joints of the robot such that the arms do not hit anything.

desired trajectory and the combined configuration with q_{free} is used to check for collisions and other constraints. We would also like to guarantee that the links attached to the fixed joints are not close to any obstacles at their goal configuration given by $\tau_{fixed}(1)$. The links attached to the fixed joints are randomly moved by a small perturbation Δ that defines a ball around the workspace in which the fixed links should be collision-free. The attached links for the fixed joints in Figure 3.15 all the gripper links. However, the attached links for the fixed joints in Figure 3.16 are all the upper-body links since the torso joints affect all of them. Therefore, GetDependentLinks returns all the *rigidly attached* and *dependent* links of the fixed joints.

Algorithm 3.10: CheckGoalCondition(q_{free})

```
for t ∈ [0, 1] do
    if {q_free, τ_fixed(t)} ∉ C_free then
        return false
end
SetConfiguration(fixed,τ_fixed(1))
Transforms ←GetTransforms(GetDependentLinks(fixed))
for iteration = 1 to M do
    NewTransforms ← ApplyTransform(Transforms,RandomTransform(Δ))
    SetLinkTransforms(GetDependentLinks(fixed), NewTransforms)
    if IsLinkCollision(GetDependentLinks(fixed)) then
        return false
end
return true
```

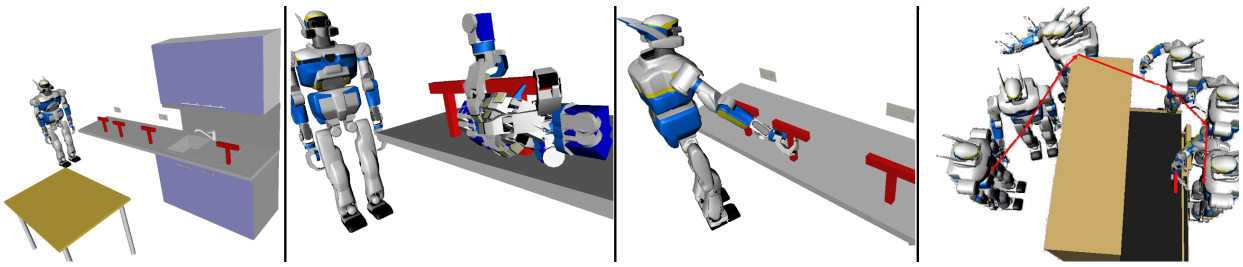


Figure 3.17: When a robot is far away from its goal, it must also plan for moving its body along with its arm. The robot should first find the possible grasps from which it can sample robot goal locations. The planning algorithms will then join the two configurations.

As will be shown in later sections, we frequently extract dependent sub-parts of the robot to make computations faster. Because the fixed joints are usually very few, we set $\tau_{fixed}(t)$ as the linear segment that joints the initial configuration and the destination configuration; in other words, no planners are involved in determining the fixed joint path. Using this approximation, the total algorithm runtime itself is on the order of **0.1 s** for example in Figure 3.15, and **1.0 s** for example on 3.16. The time greatly depends on the correct discretization levels and scene complexity, which is why the humanoid case is slower.

3.6 Planning with Base Placement

The most general manipulation planning problem is when the robot has an initial estimate of its target object and needs to also plan for a base placement that allows it to grasp the object (Figure 3.17). Because the target object can be potentially far away from the robot as shown in Figure 3.17, the robot cannot rely solely on its inverse kinematics functions to determine feasibility of the grasps; it has to first sample a base placement and then test for the existence of solutions. In this section we discuss two approaches to planning with base placement. The first approach relies on an informative base placement sampler and assumes that the robot can navigate to the sampled positions. This approach separates the planning into a two stage navigate-and-move-arm process, so it has the advantage that it does not require tight synchronization with the base and arm controllers. However, more time will be wasted sampling good robot positions because it cannot guarantee the base position is reachable by the robot. The second approach, which we call *BiSpace* planning, adds simultaneous validation of the robot base path along with finding the correct grasp for the target object. By uniquely combining work and configuration space searching, it has the advantage of finding paths for much more complex situations than robots moving on a 2D floor. We first begin with sampling base placements before moving onto the planning

algorithms.

3.6.1 Base Placement Sampling

In the context of grasp planning, we would like to sample all possible base placements q_{base} such that a grasp planning path with the arm and gripper is guaranteed to exist. Using the relational graph in Figure 2.4, the path from *Target Object* to *Robot Base* passes through grasp sets and inverting the reachability map of the arm. In Section 4.4 we present a grasp reachability distribution that allows us to query the base placements given a set of possible grasps corresponding to all target objects. The sampler

$$\{q_{base}, q_{gripper}, \text{GripperTransforms}\} \leftarrow \textsc{Sample\_GraspReachability}(arm, \mathcal{G})$$

takes in the desired arm to move and the transformed world grasp set, and returns a base placement sample along with all information about the grasp. The returned grasp is validated using GraspValidator (Algorithm 3.4) and therefore the gripper transforms are also returned. In this section we cover its usage in the context of sampling base placements.for planners.

Algorithm 3.11: GoalSampler_BasePlacement()

```
for iteration = 1 to N do
    arm ← Sample(arms)
    G ← ⋃_target T^world(q_target) ∘ G_stable(arm, target)
    {q_base, q_gripper, GripperTransforms} ← Sample_GraspReachability(arm,G)
    SetConfiguration({base,gripper},{q_base, q_gripper})
    if not IsLinkCollision(GetIndependentLinks(base)) then
        T ← LastElement(GripperTransforms)
        for q_arm ∈ IK(T) do
            if (q_base, q_arm, q_gripper) ∈ C_free then
                return (q_base, q_arm, q_gripper)
        end
end
```

GoalSampler_BasePlacement (Algorithm 3.11) shows the final goal sampler including base placement. The information used for sampling is: the poses of all the target objects, the set of arms for grasping, a set of grasps for each target/arm pair, the inverse reachability of each arm, and the obstacles in the environment. An arm is first sampled and all the grasp

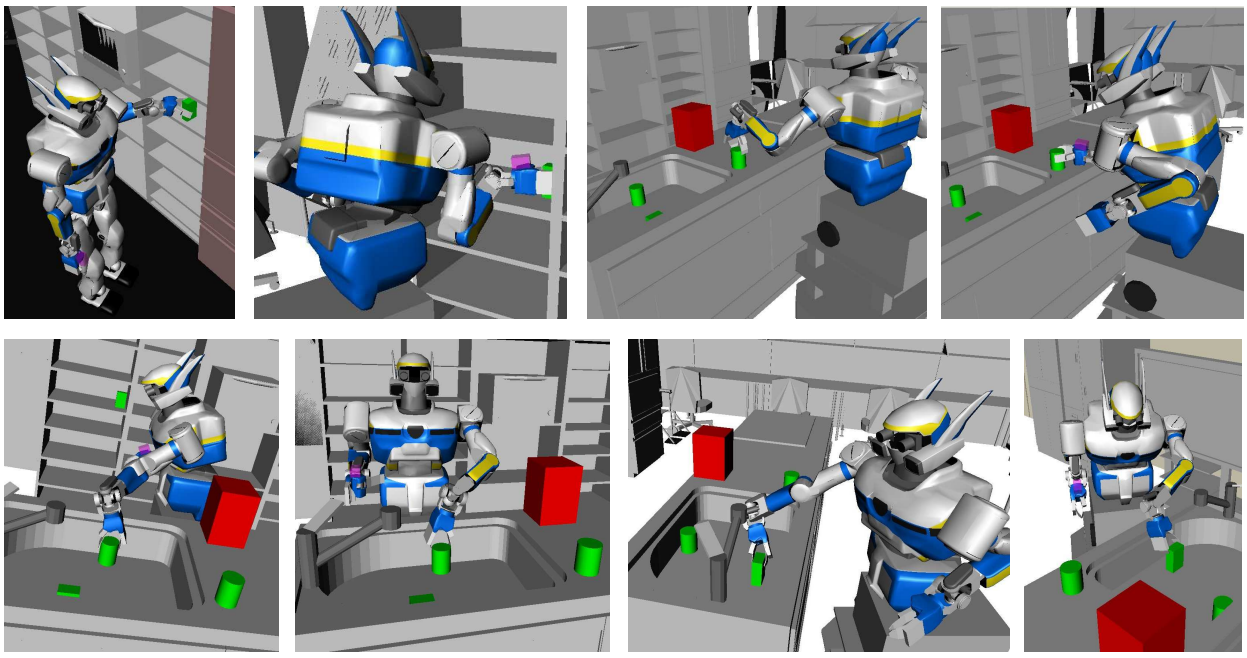


Figure 3.18: Several configurations returned from GoalSampler_BasePlacement considering both robot arms, the humanoid's torso, and multiple objects.

sets belonging to the gripper of that arm are transformed into the world and stored into G. The reachability sampler lazily calls GraspValidator before committing to a certain grasp; therefore the return grasp information is valid in the current environment up to the gripper links. These grasps are them passed into the grasp reachability sampler, which returns a potential base placement along with a chosen grasp. However, grasp reachability does not encode obstacles in the environment, so any base placement it returns has to be validated for collisions and inverse kinematics solutions. As an early pruning step, all links that are dependent on the base, but not the arms are extracted with GetIndependentLinks(**base**) and tested for collisions with the current environment. If links are not in collision, we start iterating across all inverse kinematics solutions until a full base-arm-gripper configuration inside $\mathcal{C}_{\text{free}}$ is found.

Figure 3.18 shows the results of the goal sampler on a kitchen environment with a humanoid robot and four target objects. The average time to produce one full configuration is on the order of **0.5 s** where six different arms are being sampled from. In order to prove that grasp reachability can outperform randomized base placement sampling, we performed several hundred experiments with the scenes in Figure 4.22. With grasp reachability, it takes on the order of $\mathbf{0.1 - 0.4s}$ to return the first base placement that has a collision-free inverse

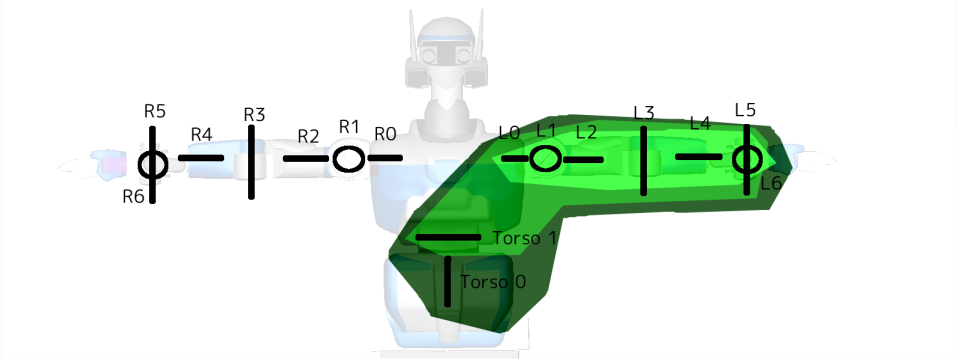


Figure 3.19: Humanoid upper body structure used to extract *arm* chains.

kinematics solution to grasps the object. With randomized sampling, the time was about 1.2 times slower for the WAM scene, and on average 4-5 times slower for the HRP2 because its reachability is more constrained.

Each *arm* consists of a chain of consecutive joints where different arms can share common joints. The humanoid in this example has 2 torso joints before its kinematic structure branches into the left and right 7DOF arms (Figure 3.19). Therefore, we can define six different, semantically meaningful arms: three left arms and three right arms each of 7, 8, and 9 joints. Unfortunately, analytical inverse kinematics solutions become very difficult for higher number of joint, and sometimes the torso joints are used for different tasks, so we cannot just consider the 9 DOF versions of the left and right arms. Choosing the arm in GoalSampler_BasePlacement() takes into account the complexity of the inverse kinematics solvers for that chain.

Because we're dealing with a distribution on a 2D plane, it is possible to speed up the sampling process by projecting the robot base and all collision obstacles on the floor and compute the Minkowski sum. The Minkowski sum could be multiplied with the grasp reachability distribution to form a new distribution that further encodes collision obstacles. Such a method would be beneficial for static industrial environments, but would not speed up computation for quickly changing environments unless the Minkowski sum is progressively computed as configurations get rejected.

3.6.2 Two-Stage Planning with Navigation

The simplest type of planning system using base-placement sampling first uses a navigation planner to assure a path exists to q_{base}. Although it has been shown that RRTs can also be used to plan simultaneously with base-placement configuration, it is much simpler to treat the two processes separately, which is similar to the hand-arm argument presented in Section

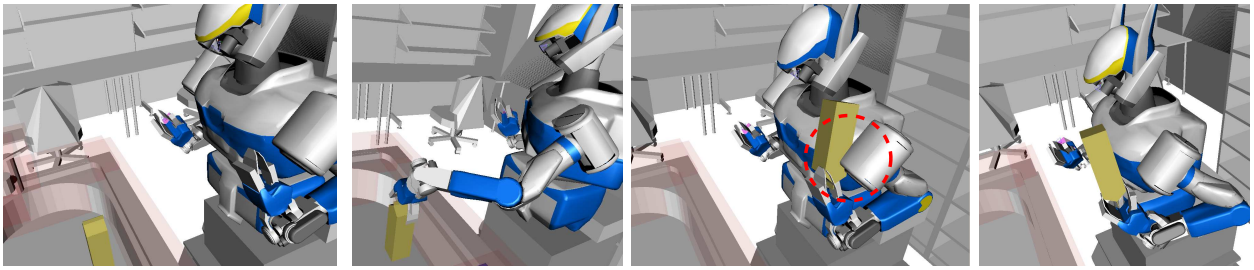


Figure 3.20: Having just a static navigation configuration $q_{navigation}$ is not enough to safely go into navigation mode, sometimes the target object might collide with the robot.

3.5. Once q_{base} has been validated and the robot moves to it using the navigation module, the target object pose is re-confirmed and the grasp planner is executed as shown in Section 3.3.

Algorithm 3.12: GoalSampler_NavigationMode()

$q_{rand} \leftarrow$ Sample($\mathcal{C}$)
SetConfiguration(**base**, q_{rand})
if not IsLinkCollision(GetIndependentLinks(**base**)) **then**
    $\Delta T \leftarrow \begin{bmatrix} I_3 & \text{Sample}(\mathbb{R}^3) \\ 0 & 1 \end{bmatrix}$
    $Q \leftarrow \emptyset$
    for arm $\in$ DisjointArms() **do**
        $T^{world}_{gripper} \leftarrow$ FK(**arm**, $q_{navigation}$)
        Append(Q, $IK(\Delta T\, T^{world}_{gripper})$)
    end
    for $\{q_{arm,0}, q_{arm,1}, ...\} \in Q_0 \times Q_1 \times ...$ **do**
        $q \leftarrow$ MergeConfigurations(q_{rand},$\{q_{arm,0}, q_{arm,1}, ...\}$)
        if $q \in \mathcal{C}_{\text{free}}$ **then**
            return q
    end

Because navigation planners work on the 2D plane by assuming the robot has a 2D footprint, it is necessary to move all the robot's limbs in the convex prism defined by the 2D footprint. If the robot is not grasping anything, this is this step is trivial since a $q_{navigation}$ configuration for the arms can be preset. However, if the robot has just grasped an object and needs to folds its arms to navigate, there is a chance that the object might get into self

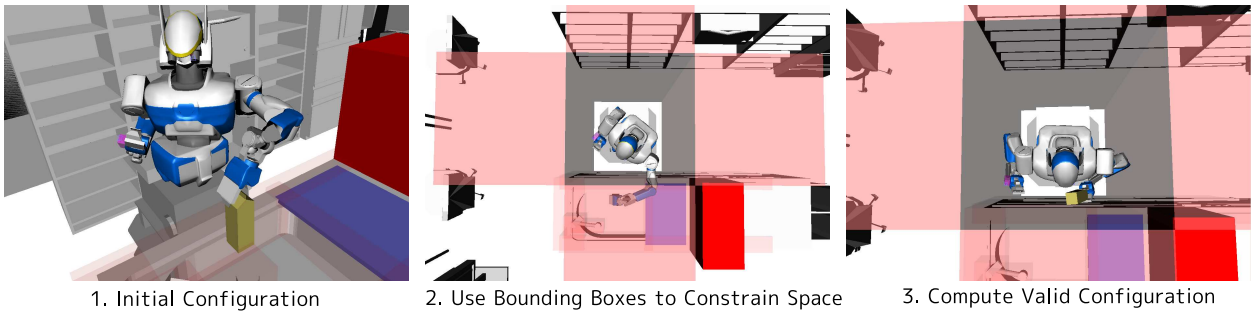


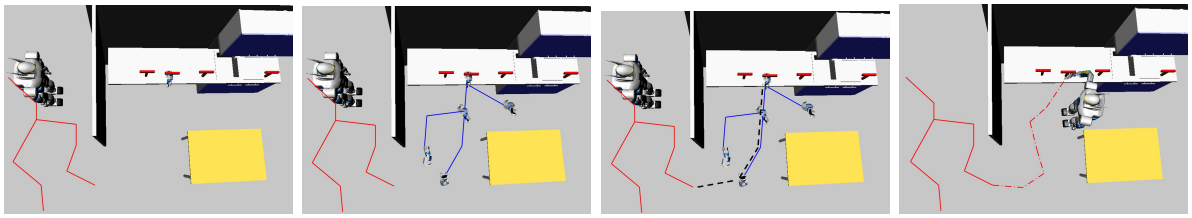
Figure 3.21: Robot needs to plan to a configuration such that all its limbs are within the base navigation footprint.

collision with one of the robot links as shown in Figure 3.20.

In order to sample a navigation mode configuration, we temporarily set up a set of bounding boxes across the footprint of the robot as shown in Figure 3.21. GoalSampler_NavigationMode (Algorithm 3.12 searches for a collision free configuration using this constraint. In order to guarantee more natural goal poses, we sample symmetric end-effector positions for each of the arms with the orientations copied from $q_{navigation}$. A random configuration of the robot is first sampled along with a random translation ΔT to move all the end-effectors in. The original end-effector position of each of the robot arms is extracted from $q_{navigation}$, ΔT is added to its translation, and the inverse kinematics solutions are stored. Because arms can share joints, we designate a set of arms that share no common joints with DisjointArms. Once all the arm inverse kinematics solutions have been set, we take their cross product and check if any configuration lies in $\mathcal{C}_{\text{free}}$. If all arms have a collision-free configuration at these end-effector positions, then the sample is used in the planner. Given the scene in Figure 3.21, it takes approximately **0.1s** to sample a valid configuration and less than a second to get a path.

3.6.3 BiSpace Planning

Although the two-stage mobile manipulation algorithm is really simple to implement and useful when the base placement sampler returns valid solutions that are also reachable most of the time, there are no guarantees with how feasible a path is until the navigation planner has validated it. The two-stage process also requires the entire robot be under its navigation footprint at all times while moving, which can constrain the space for more complex tasks base movement than on the floor. We solve these problems by presenting a planning algorithm called *BiSpace* [Diankov et al (2008a)] that produces plans to complex high-dimensional problems by simultaneously exploring both the robot base and arm space $\{q_{base}, q_{arm}\}$. In



(1) (2) (3) (4)

Figure 3.22: BiSpace Planning: A full configuration space tree is grown out from the robot's initial configuration (1). Simultaneously, a goal back tree is randomly grown out from a set of goal space nodes (2). When a new node is created, the configuration tree can choose to *follow* a goal space path leading to the goal (3). Following can directly lead to the goal (4); if it does not, then repeat starting at (1).

the following process we assume the robot base is holonomic; however BiSpace can also be applied to non-holonomic robots like humanoids; by setting a big base, the resultant goals produced by BiSpace have a high probability to be reached by the base since a path is produced. We specifically focus on using BiSpace's special characteristics to explore the work and configuration spaces of the environment and robot. Furthermore, we use the reachability space of the robot for constructing informed heuristics to informatively search through the high-dimensional spaces involved with arms.

The core idea of the BiSpace algorithm is to grow two search trees in different configuration spaces at the same time. It combines elements of both bidirectional RRTs and the RRT-JT algorithm [Weghe et al (2007)]. One tree explores the full robot configuration space starting from the initial configuration and guarantees feasible, executable, and collision-free trajectories, while the other tree explores a goal space starting from the set of goal configurations and acts as an adaptive, well informed heuristic. The BiSpace algorithm proceeds by extending RRTs in both spaces. Once certain conditions are met, the forward tree attempts to *follow* the goal space tree path to the goal. Figure 3.22 shows how the robot path eventually follows the workspace trees of the gripper.

For clarity, we denote a configuration with q and a goal space configuration with b, usually a goal space configuration is the transformation of the gripper $T_{gripper}$. We assume that there exists a mapping $F(\cdot)$ from the configuration space to the goal space such that $F(q)$ maps to exactly one goal space configuration. Using this notation, given a goal space distance metric $\delta_b(F(q), b)$, the goal of planning is to find a path to a configuration q such that $\delta_b(F(q), b_{goals}) < \epsilon_{\text{goal}}$.

The flow is summarized by BiSpace (Algorithm 3.13). The **forward** variable is used to keep track of which tree to grow. If **forward** is **true**, then the configuration space tree

Algorithm 3.13: BiSpace(q_{init}, b_{goals})

```
/* ρ ∈ [0,1] - uniform random variable                              */
1  forward ← false
2  Init(T_f, q_init); Init(T_b, b_goals)
3  for iter = 1 to maxIter do
4      if forward then
5          for fiter = 1 to J do
6              q ← Extend(T_f)
7              if ρ < FollowProbability(q) then
8                  b_follow ← NearestNeighbor(T_f, q)
9                  {success, q'} ← FollowPath(q, b_follow)
10                 if success then
11                     return success
12         end
13     else
14         for biter = 1 to K do  Extend(T_b)
15     forward ← not forward
16 end
17 return failure
```

is extended **J** times, using the standard RRT extension algorithm Extend [LaValle and Kuffner (2001)]. Alternatively, if **forward** is **false**, then the goal space tree is extended **K** times. After each iteration, the value of **forward** is flipped so that the opposite tree is extended during the subsequent iteration. After a new node q is added to the configuration space tree, a *follow* step is performed from q with probability FollowProbability(q). If a *follow* step is performed, then q is extended toward b_{follow} and its parents.

The differences between BiSpace and BiRRTs become clear in the *follow* step. In the BiRRT case, *following* consists of connecting the two trees along the straight line joining q and b_{follow}; this is possible since b_{follow} is in the same configuration space as q. Because each branch of the both the forward and backward trees in the BiRRT algorithm represent a valid collision free path in the configuration space, connecting the two trees immediately implies a path can be found from the start configuration to the goal configuration. However, that is not true with the BiSpace algorithm. Since the goal space is different from the configuration space, the path suggested by the goal space tree must be validated in the configuration space.

Each unique path from a node in the goal space tree to a goal can be used by the forward

tree as a heuristic to informatively bias extension toward the goal. Starting from b_{follow}, such a path can be extracted by recursively following its parents. The forward tree can use the goal space path generated by b_{follow} as a bias to greedily *follow* it. If the forward tree succeeds in reaching the goal, a solution is returned (Figure 3.22). Otherwise, the search continues as before.

Path following is an integral part of the BiSpace algorithm. It generates a very powerful bias as to where the configuration tree should grow by using the nodes in the goal space tree. Each goal space node has already validated a subset of the conditions necessary for the configuration tree to follow it.

We present a simple, but effective, implementation of path following using a stochastic gradient approach as shown in FollowPath (Algorithm 3.14). The forward tree slowly makes progress by randomly sampling configurations that get close to the target goal space node b. Whenever the forward tree stops making progress, it checks if b has any parents. If it does, b is set to its parent and the loop repeats. If there are no more parents, the goal space distance from q to the final parent $b.root$ is checked: if this distance is within the goal threshold, the function returns success; otherwise it returns false.

FollowPath can require a lot of samples if SampleNeighborhood uniformly samples the neighborhood of q. This is especially a problem for the high-dimensional configuration spaces used in manipulation planning. Instead, we sample each of the dimensions one at a time while leaving the rest fixed. This type of coordinate descent method has been shown to perform better than regular uniform sampling in optimization and machine learning algorithms [Luo and Tseng (1992)]. Because it is not always beneficial to be greedy due to many local minima, we introduce $\gamma_{inflation}$ to relax the distance metric we are minimizing. The inflation has a similar effect to inflating the goal heuristic in A*; $\gamma_{inflation} = 1.4$ is used for all results.

We use inverse kinematics solution validation as an optional addition to FollowPath. After the forward tree terminates at a configuration q, an IK solution can be checked for a subset of the DOFs of the configuration space. If there exists a solution, we can run a bidirectional RRT using the subset of DOFs used for IK to find a path from q to the new goal configuration. For example, if a 7 DOF arm is mounted on a mobile platform, its full configuration space becomes 10 dimensional, however, the arm's IK equations will still remain 7 dimensional. Having such a check greatly reduces planning times and is not prohibitively expensive if the IK equations are in closed form. While some algorithms ignore IK solutions, BiSpace can naturally use inverse kinematics to its advantage. Empirical results suggest that BiSpace can experience a 40% decrease in planning time when exploiting available IK solutions.

Algorithm 3.14: FollowPath(q, b)

```
   /* ρ ∈ [0,1] - uniform random variable                              */
 1 success ← false
 2 for iter = 1 to maxFollowIter do
 3     best ← null
 4     bestdist ← γ_inflation * δ_b(F(q), b)
 5     for i = 1 to N do
 6         q' ← SampleNeighborhood(q)
 7         if δ_b(F(q'), b) < bestdist then
 8             bestdist ← δ_b(F(q'), b)
 9             best ← q'
10     end
11     if best is null then
12         if b.parent is null then
13             break
14         b ← b.parent
15     else
16         q ← T_f.add(q, best)
17 end
18 success ← δ_b(F(q), b.root) < ε_goal
   /* Optional IK test                                                 */
19 if not success and (q' ← IKSolution(q, T_b.goals)) then
20     {success, q} ← BiRRT(T_f, q, q')
21 return {success, q}
```

Since BiSpace is a randomized algorithm, it cannot detect in a finite amount of time that a given collision-free grasp is impossible to reach. Therefore, seeding BiSpace with only one grasp at a time is dangerous as the planner might never find a solution. Instead, it is favorable to seed the BiSpace planner from the beginning with as many feasible grasps as possible using the precomputed grasp tables, increasing the likelihood that at least one of the grasps can be reached. Since the Extend operation is not affected by the number of trees being grown, incorporating multiple goals in the goal space does not affect efficiency [Okada et al (2004)].

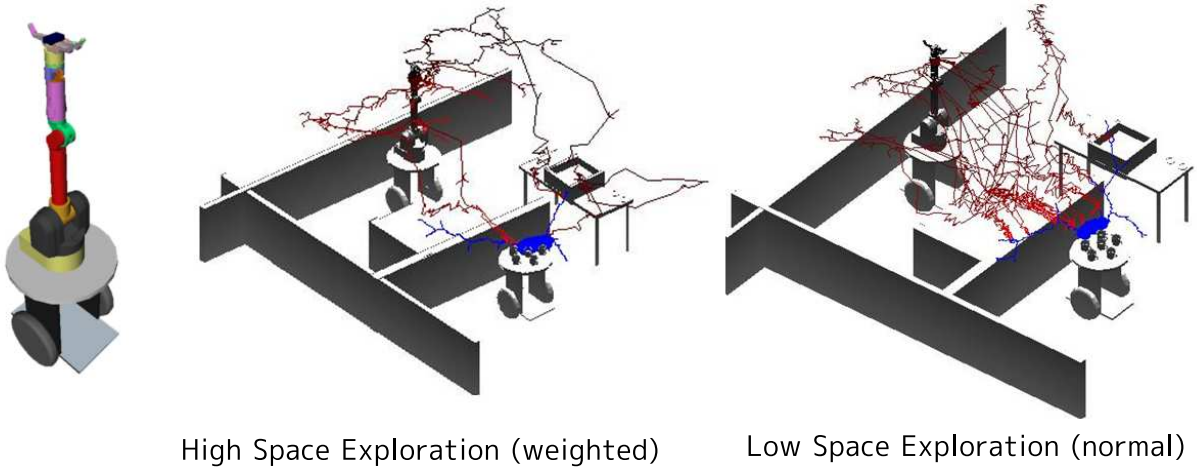


Figure 3.23: Comparison of how the distance metric can affect the exploration of the arm. The top image shows the search trees (red/black) generated when the distance metric and follow probability is weighted according to Equation 3.14. The bottom image shows the trees when the distance metric stays uniform across the space; note how it repeatedly explores areas. The goal space trees are colored in blue.

Reachability Heuristic

It is clear that successfully moving to a specific grasp requires the robot move its base so that its reachability volume coincides with the particular grasp. Given a target grasp g, we would like to compute a probability distribution $P_g(p)$ of completing the grasp as a function of the base placement p. In Section 4.4 we introduce a reachability map that can retrieve a the set of all possible base placements for a given end-effector transformation. This set can be treated as a Mixture of Gaussians and probably can be computed directly.

Workspace Exploration Heuristic

As humans, we employ different navigation strategies based on our distance to a goal object. When a person is far away from an object of interest, they care primarily about moving their body in a direction that will get them close to the object. When they are close, they usually plant their feet and use their arms to make contact with the object. We can achieve the same behavior in BiSpace by modifying the configuration space distance metric such that

$$\delta(q) = |\omega(q) q_{arm}| + |q_{base}| \tag{3.14}$$

where ω weights the important of base exploration vs arm exploration. When the robot

base is far away from the goal, the weight ω should be small so that the robot takes bigger steps on average. This suggests a simple monotonic function for ω:

$$\omega(q) \propto \exp\left\{-\frac{\min_i |goal_i - BasePosition(q)|^2}{2\sigma^2}\right\} \tag{3.15}$$

where σ is proportional to the length of the arm. Figure 3.23 demonstrates the behavior of BiSpace when using the modified distance metric, and empirical results show that planning times reduce by 20% when this metric is used.

Follow Probability

The farther the robot is away from the goal, the less chance it will have of reaching it through FollowPath. The reason is because FollowPath itself is not exploration-centric like RRTs; it is meant for greedily approaching the goal when the body and hand of the robot are relatively unobstructed by complex environment obstacles. We present two metrics to compute the follow probability: the kinematic reachability explained in Section 4.3 and the distance falloff $\omega(q)$ given by Equation 3.15. Both metrics monotonically decrease as the robot gets farther from the goal. The kinematic reachability is more informed since it is a 6D table reflecting the real arm kinematics while $\omega(q)$ is much easier to compute and often very effective as shown in Figure 3.23. The correct follow probability can have a dramatic effect on planning times, sometimes reducing it by 60-70%.

BiSpace Experiments

We compare BiSpace with RRT-JT [Weghe et al (2007)] and the two-stage navigation approach described in Section 3.6.2. A mobile base adds three degrees of freedom to the configuration. Randomized algorithms are known to have a long convergence tail, we terminate the search after 10-20 seconds and restart. This termination strategy produces much faster average times for all algorithms. Note however that every termination counts against the final planning time for that particular algorithm. Termination times were uniquely set for each algorithm in order to give it the fastest possible average time. The average planning time is recorded in Table 3.3 where each algorithm is run on each scene 16-30 times. Other parameters like RRT step size and goal thresholds were kept the same for all algorithms. To demonstrate the generality of these algorithms, we evaluated scenes using both the HRP2 humanoid and a WAM arm loaded on a segway (Figure 3.24).

Since BiRRTs operate only in the full configuration space it would be unfair if they were seeded with the final solutions without any penalties. In order to make comparison fair, we

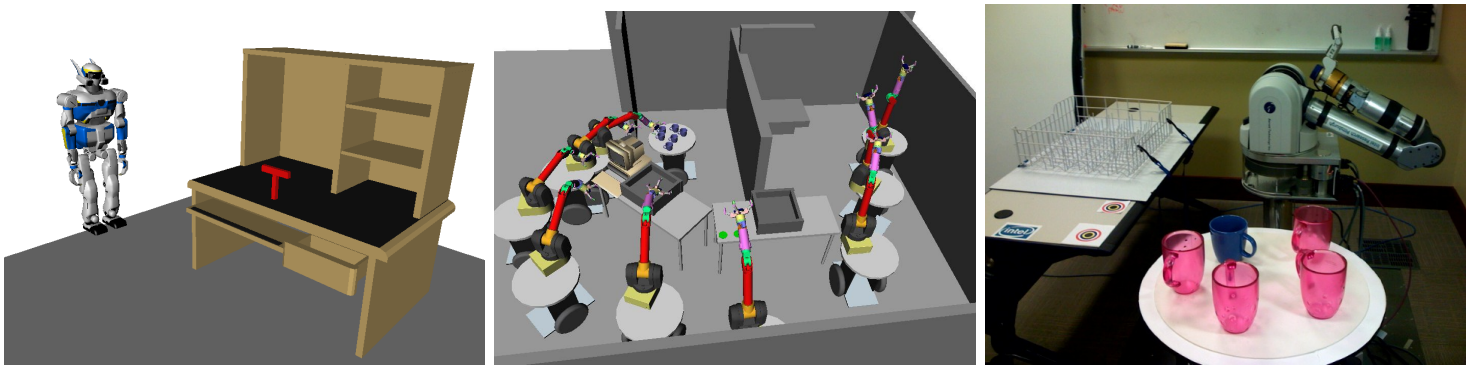


Figure 3.24: Scenes used to compare BiSpace, RRT-JT, and BiRRTs.

	BiSpace	RRT-JT	BiRRTs
HRP2 table (11 DOF, easy)	**33**	53	68
HRP2 table (11 DOF, harder)	**45**	528	78
HRP2 random (11 DOF)	**37**	170	40
WAM/segway (10 DOF)	**17.25**	25.2	22.93
WAM (7 DOF)	0.44	11	**0.37**

Table 3.3: Average planning time in seconds for each scene for the scenes in Figure 3.24.

randomly sample full configuration solutions for a given target grasp until a collision-free, feasible configuration is generated. The recorded time is added to the final planning time. The sampling takes somewhere from 2-9 seconds for HRP2 and less than 1 second for the WAM on segway.

When planning for the HRP2 robot, we make the assumption that its base can freely travel on the floor and the legs do not need to move. Once BiSpace has planned a global trajectory, later footstep planners can add the necessary leg movements and dynamics to make the HRP2 move. In order to allow for leg space, an invisible cylinder is super-imposed over the lower body. Thus the planning space for HRP2 is reduced to 11 degrees-of-freedom: 3 for the base, 1 for the waist, and 7 for the arm. As can be seen from Figure 4.19, most of the hand reachability lies shoulder height to the side of the robot. This makes it hard for the robot to manipulate objects in front of it at waist height, which is why all the planners require significant planning time.

One of the hardest scenes for BiSpace is when the target object is on a table and HRP2 has to circle the table to get to it (Figure 3.25). Here, the goal space tree produces many false paths directly over the table, which the HRP2 cannot follow to the end. This process

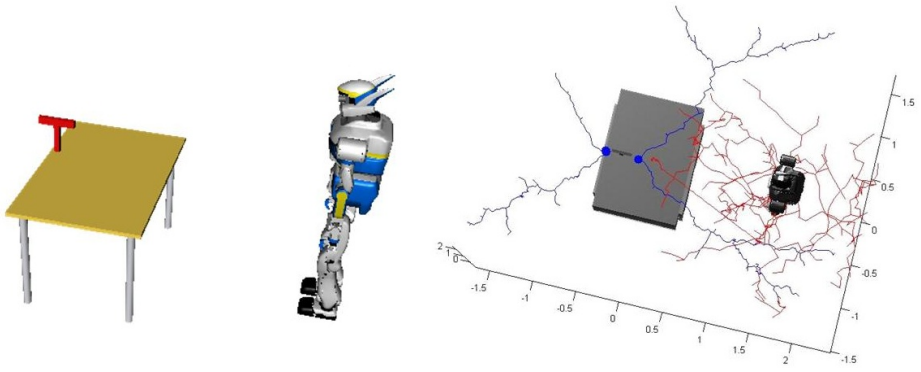


Figure 3.25: Hard scene for BiSpace. The forward space tree (red) does not explore the space since it is falsely led over the table by the goal space tree (blue).

goes on until the rest of the configuration space tree finally explores the space on the other side of the table. This limitation is characteristic of bi-directional RRTs also and provides a good example of why exploration is always a crucial ingredient in sampling-based planners.

We tested two main scenes for the WAM: a living room scenario where the WAM is mobile, and a scenario where the WAM has to put cups in a dishwasher. The WAM arm has 7 degrees of freedom and very high reachability making planning very fast. BiSpace compares relatively well with BiRRTs, however it is a little slower due to the extra overhead in the FollowPath function.

The BiSpace algorithm can efficiently produce solutions to complex path planning problems involving a goal space that is different from the configuration space. We presented several heuristics that exploit the kinematic structure of the robots to speed up planning.

3.7 Discussion

We have worked to find a reasonable and sufficient set of planning algorithms that can cover the entire spectrum of manipulation planning problems proposed in Chapter 2. We have shown their application in a number of different tasks with a number of different robot platforms. Most importantly, all information that the planning algorithms use can be computed by a geometric analysis of the environment. In fact, we showed how the planning knowledge-base plays a key role in instantiating samplers and functions used throughout planning. It is very difficult to measure the time-complexities of sampling-based planners because of the high dependency on the environment and the sampling distribution. Furthermore, the RRT family of planners are known to have long tails for planning time distributions, making it

difficult to prove theoretical time bounds for them. Instead, we have presented timing experiments of thousands of simulations for environments that are representative of home and industrial settings.

All the planning theory culminated in a generalized configuration sampler that can efficiently reason what grasps to use in an environment and can sample informative base placements. Along with covering the basic grasp planner, we also covered other common planning situations like planning to open a gripper or planning to a *navigation mode* when using in conjunction with navigation planners. Planning with a mobile base requires not only efficient searching for the paths as shown in the BiSpace algorithm, but also careful selection of the goals. For example, choosing a grasp requires simulating the grasp in the environment to assure only the targeted object contacts with the gripper. Quickly choosing a base placement requires computation of the arm reachability along with simultaneously all grasp sets for all target objects. All these planning algorithms have been deployed on real robots systems and the computation times written at the end of each respective sub-section have been verified over many trials.

Chapter 4

Manipulation Planning Knowledge-base

The information a task relies on can be divided into information defining the task and independent of the current state of the environment and information obtained at run-time like obstacles and target positions that the robot has no way of pre-computing beforehand. In this chapter we analyze the relational structure of information pertaining to the robot and task specifications and develop several offline algorithms that can pre-compute this information into a form that makes online retrieval as quick as possible. We organize all this information into a *planning knowledge-base* where all knowledge-base models are specifically optimized to make the manipulation processes discussed in Chapters 3 and 5 as quick and accurate as possible. We achieve this by identifying basic-building blocks and explicitly encoding their inter-dependencies. The goal is to quickly compute answers to questions about the inter- and intra-relationships of these blocks by approximating the acquired models and caching the information for quick retrieval.

Ideally, the planning knowledge-base should give a sense of how the robot operates within its workspace and how the task definition can affect the choices the robot has to make. Most of the building blocks are based on the geometry of planning problems where precisely defined computations do not have a trivial solution to their inverse computation, or require analyzing the behavior of the computation across the entire input space. Using the main components from Figure 2.4, we can precisely define their relationships to produce the planning knowledge-base graph of Figure 4.1. This shows a computational dependency graph of the components commonly used to solve manipulation tasks. At the left we start with the robot and task specifications. Along with the kinematics and geometry, each robot has a labeled set of chains that serve as the arm and gripper groups. Each gripper is attached to

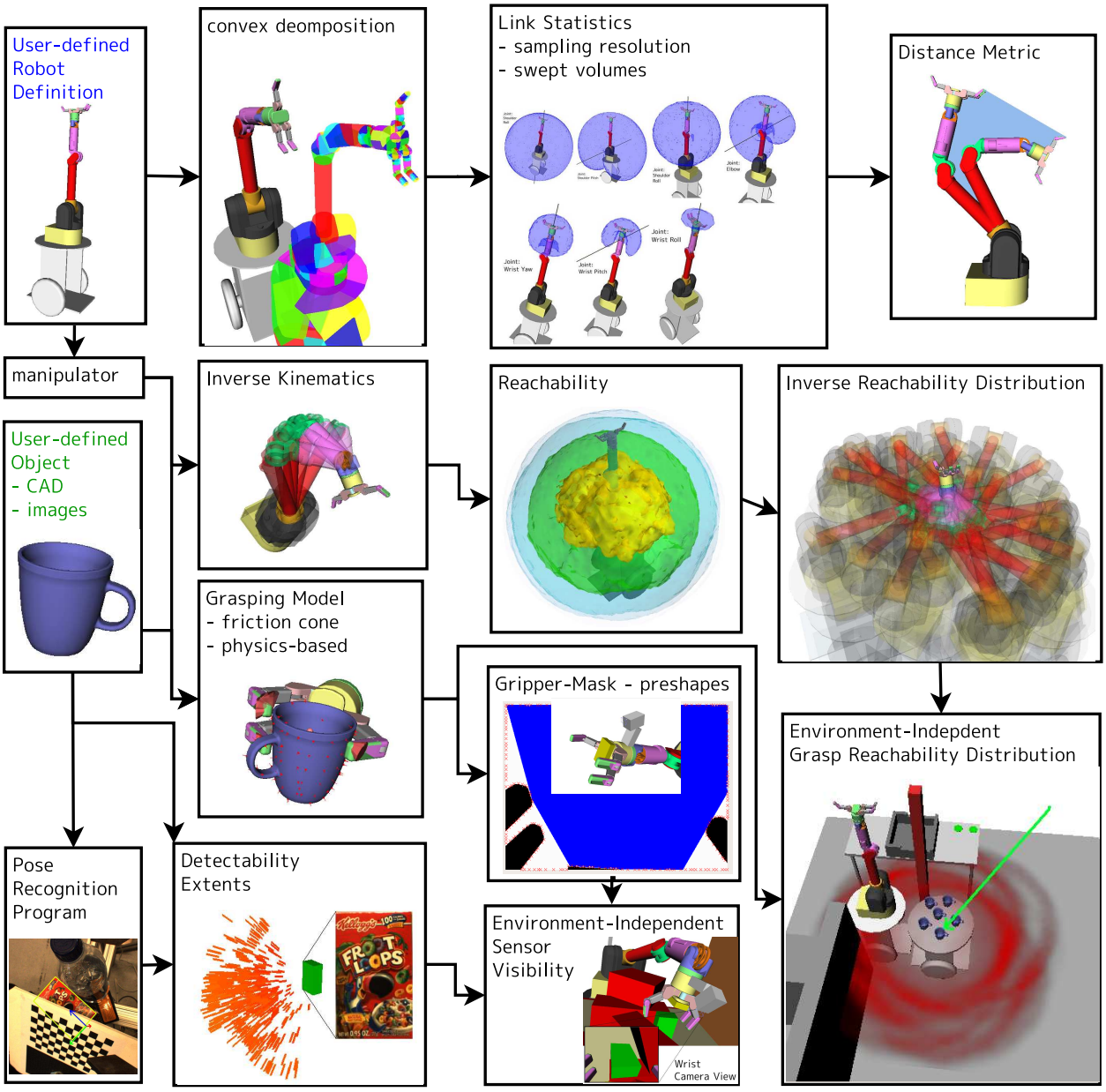


Figure 4.1: Computational dependency graph of all the components extracted from the relational graph in Figure 2.4.

an arm and has a specific 3D direction of approach. On the other hand, the target object just contains the CAD model and a set of training images that allow vision algorithms to create an object-specific pose recognition program for the target object.

We begin with a motivation for all the components in Figure 4.1. Analytic inverse kinematics equations for each arm are required for fast planning and many of the geometric analyses following. In Section 4.1, we introduced an algorithmic approach named **ikfast** for finding the most robust closed-form solutions. Inverse kinematics allows exact computation of the arm reachability space that is used for many planning heuristics (Section 4.3). Inverting the reachability and projecting it down to 2D planar movements allows us to start computing base distributions (Section 4.4). From the target object perspective, we generate all the grasps that handle the object according to the task specification. In Section 4.2 we cover two algorithms for computing grasps: force closure grasps and caging grasps. We show the usefulness of these grasps for a wide range of tasks. For each pose recognition program trained from the object specification, we gather a map of all the camera locations that can robustly detect the object (Section 4.7). This combined with the preshapes of the grasps allows us to compute a sensor visibility map for the robot. By combining grasp sets and inverse reachability, we can start reasoning about how the robot base gets distributed depending on the location of the target object; we call this map *grasp reachability* and cover its generation and usage in Section 4.5. We also discuss the advantages of using convex decompositions for the collision geometry of a robot (Section 4.6). Convex decompositions give us much simpler geometries to work with allowing us to pad the robot and to approximate the volume of each of its links. Once we have an estimate of the volume a link takes, we can compute swept volumes for each of the joints and from there compute much better configuration space distance metrics (Section 4.6.3).

The knowledge-base should be optimized for accurate modeling of the underlying geometry and fast usage times during online planning. Because every model has several parameters, we provide experimental data to help provide an intuition for the discretization factors and thresholds involved with each of the models. However, it should be noted that we are not concerned with the computation time of the generation process for each component. We assume that such databases can be computed offline in a cluster of computers and then stored on a server for quick download and usage. In fact when we cover the OpenRAVE robotics environment responsible for the experiments in this thesis, we cover the computation of unique hashes for the robot and target bodies so that the data can be indexed more consistently (Section A.1.5).

4.1 Inverse Kinematics

The relationships between the workspace movement of the robot sensors and grippers with the configuration of the robot is expressed through kinematics. For planning algorithms and other workspace analysis, the inverse problem of going from desired frames of reference on the robot to joint angles is equally important. In this thesis we identify several problems in inverse kinematics and provide an automated analysis and solutions for them. The most popular inverse kinematics problem is to compute all the joint angles q_{arm} that move a link to a specified translation and orientation $T = \begin{bmatrix} R & t \end{bmatrix}$. Pure mathematical attempts to analytically solve the general problem could not handle singularities and resulting programs are not always optimal [Low and Dubey (1987)]. Therefore, many flavors of numerical methods have been developed involving numerical gradient descent using $\frac{\delta T}{\delta q}$. In fact, most of the literature revolving around generalized inverse kinematics solutions revolves around numerical methods disregarding the analytical problem as too difficult to solve. Although general analytical solutions to the problem have been proposed [Manocha and Zhu (1994)], the methods can become numerically unstable due to the heavy mathematical machinery like eigendecomposition.

In this section, we look at the automatic generation of *a minimal, numerically stable analytical inverse kinematics solver* using an algorithmic search-based approach. We note that it is difficult to prove existence of a solution with algorithmic approaches; however, we will show that by sacrificing a little generality, the presented analysis can produce much more stable closed-form solutions to most of the common robot arms available today. Closed-form solutions are vital for motion planning for two reasons:

1. Numerical inverse kinematics solvers will always be much slower than closed form solutions. But because planners require being able to process thousands of configurations per second, it is critical to have fast IK solvers. The closed-form code generated by our proposed method can produce solutions on the order of **6 microseconds**. As a comparison, most numerical solutions are on the order of 10 milliseconds and have to worry about convergence.

2. The null space of the solution set can be explored because all solutions are computed. This gives planning algorithms more freedom to move the robot around.

In this thesis, we develop the **ikfast** algorithm to perform this generation process. As a program, **ikfast** generates a C++ file that directly compiles into a solver. Although inverse kinematics can become arbitrarily complex with closed-chains, we only consider kinematic chains employing hinge and prismatic one degree of freedom joints.

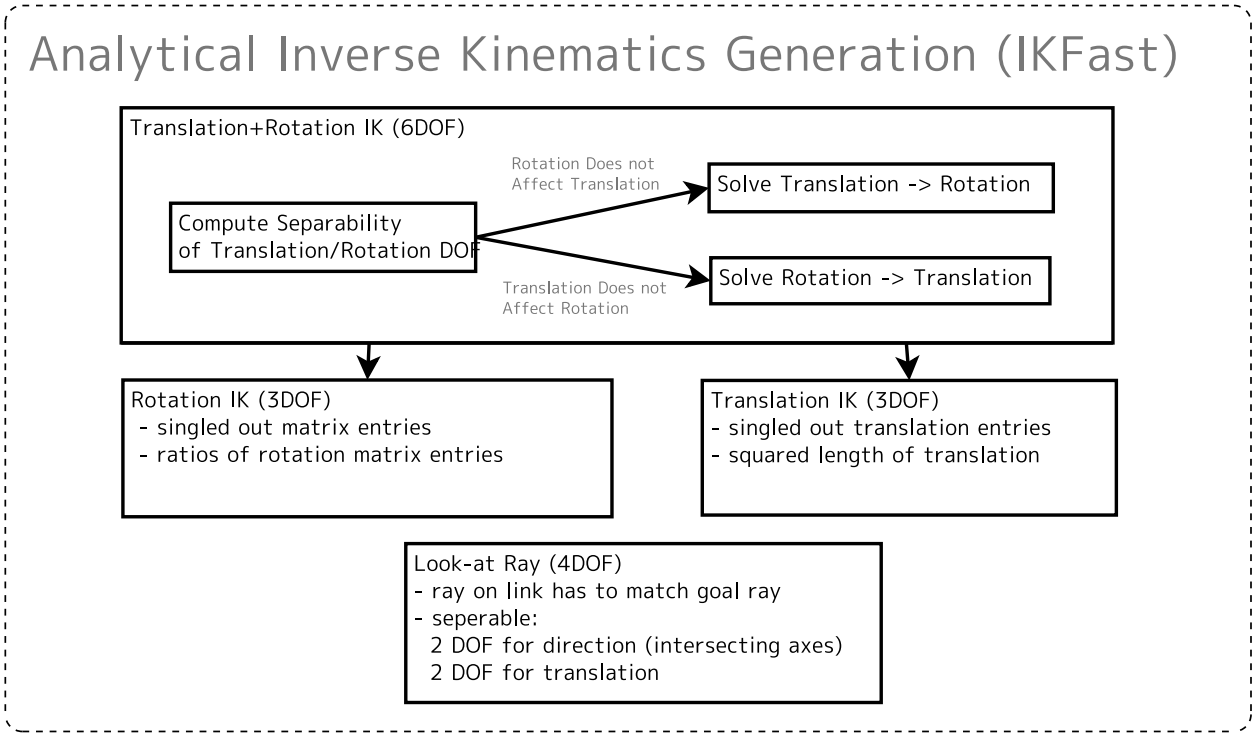


Figure 4.2: An outline of the parameterizations **ikfast** can solve along with how the equations are decomposed.

We identify several inverse kinematics parameterizations important for robotics:

- **Transformation IK (6DOF Translation+Rotation)** - The most common form of inverse kinematics used to get a desired workspace movement to a set of configurations. This IK is vital for quick grasp planning where each grasp specifies an end effector transformation. Furthermore, this IK can be used to orient sensors to a desired location for securing visibility.
- **Translation IK (3DOF)** - Used for achieving a 3D position without worrying about orientation. Many problems in robotics like pushing buttons and other objects involve only a point contact.
- **Rotation IK (3DOF)** - Used for achieving a particular orientation without concern of the translation component. There are visibility problems in computer vision where a part must be observed from all orientations in order to build a good appearance/perception model of it. Such tasks do not pose strict constraints on sensor position.

- **Look-at Ray (4DOF)** - This is another parameterization used for camera visibility for computing the pose of objects in the world. A camera can be thought of as a frustum which has a specific position and direction. When we need to move the camera sensor to observe an object, we can parameterize the problem as placing the camera anywhere along a ray pointing to the object. Although the closer objects provide better measurements, there are no strict constraints on distance to object or roll around the ray axis.

Figure 4.2 shows an outline of the four parameterizations and general approach to solving each. In the following sections we discuss the **ikfast** implementation in detail and evaluate its performance at the end.

4.1.1 Basic Formulation of Inverse Kinematics

The fundamental chain of transformations that compute the end effector link frame is defined by:

$$T_{ee} = T_0 J_0 T_1 J_1 T_2 J_2 ... T_n = \begin{bmatrix} r_{00} & r_{01} & r_{02} & p_x \\ r_{10} & r_{11} & r_{12} & p_y \\ r_{20} & r_{21} & r_{22} & p_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.1}$$

where T_i is a constant affine transformation, and J_i is a transformation depending on joint value j_i, which can be a rotation around an arbitrary axis v_i:

$$J_i(j_i) = I_{4x4} + \sin j_i \begin{bmatrix} 0 & -v_{i,z} & v_{i,y} & 0 \\ v_{i,z} & 0 & -v_{i,x} & 0 \\ -v_{i,y} & v_{i,x} & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} + (1 - \cos j_i) \begin{bmatrix} 0 & -v_{i,z} & v_{i,y} & 0 \\ v_{i,z} & 0 & -v_{i,x} & 0 \\ -v_{i,y} & v_{i,x} & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}^2 \tag{4.2}$$

or a translation along an axis:

$$J_i(j_i) = \begin{bmatrix} I_{3x3} & j_i v_i \\ 0 & 1 \end{bmatrix} \tag{4.3}$$

We derive all constraints used for solving the IK by changing the frame of reference of Equation 4.1:

$$
\begin{aligned}
T_0^{-1} T_{ee} T_n^{-1} &= J_0 T_1 J_1 T_2 J_2 ... J_{n-1} && (i = 0) \\
J_0^{-1} T_0^{-1} T_{ee} T_n^{-1} &= T_1 J_1 T_2 J_2 T_3 ... J_{n-1} && (i = 1) \\
T_1^{-1} J_0^{-1} T_0^{-1} T_{ee} T_n^{-1} &= J_1 T_2 J_2 T_3 J_3 ... J_{n-1} && (i = 2) \\
&... \\
T_{n-1}^{-1} J_{n-2}^{-1} ... T_0^{-1} T_{ee} T_n^{-1} &= J_{n-1} && (i = 2n-2)
\end{aligned} \tag{4.4}
$$

For each of the equations above, we define $F_{ee,i} = \begin{bmatrix} R_{ee,i} & t_{ee,i} \\ 0 & 1 \end{bmatrix}$ and $F_i = \begin{bmatrix} R_i & t_i \\ 0 & 1 \end{bmatrix}$ such that $F_{ee,i} = F_i$. In order to simplify the equations that will be covered below for several real robots, we let $c_d = \cos j_d$, and $s_d = \sin j_d$.

In the past, there have been four different approaches to solving the IK problem. The first is an algebraic approach where the problem resolves into a high-degree univariate polynomial. Such an approach can easily become ill-conditioned when searching for roots. A second approach is based on analyzing the structure of solutions sets of polynomial systems [Wampler and Morgan (1991)], but suffers from numerical ill-conditioning. The third approach is based on linear algebra where the problem can be formulated as eigendecomposition [Raghavan and B.Roth (1990); Manocha and Zhu (1994)]. Although the linear algebra approach is the fastest from all the proposed methods, it still suffers from numerical problems due to eigendecomposition of 48x48 matrices. Our method differs in the past approaches in that we emphasize numerical stability over solution generality.

The constraints in Equation 4.5 correspond to equations of the form:

$$
\sum_i a_i \prod_j c_j^{p_{i,j}} s_j^{q_{i,j}} j_i^{r_{i,j}} = 0 \tag{4.5}
$$

where $p_{i,j} + q_{i,j} + r_{i,j} <= 1$. Combining with the $c_j^2 + s_j^1 - 1 = 0$ trigonometric constraints for all revolute joints, we have a system of up to $2n$ unknown variables. The challenge to building a stable IK solver is to exploit as much structure of the problem as possible by first starting with all variables that can be solved with low-degree polynomials. In this paper, we cover the kinematic complexity up to solving intersections of conics. The problem can become more complex with pairs of joint variables being coupled, in which case intersections of quadrics is necessary [Dupont et al (2007)]. Although extremely rare in today's commercial robots, if more joint coupling is present, we fall back to a general solver like [Manocha and Zhu (1994)].

4.1.2 Evaluating Equation Complexity

The first challenge is using the forward kinematics equations to set up all possible constraints on the variables. The solver searchers for solutions prioritizing the simplest and most unambiguous solutions first. A variable can be solved in multiple different ways, each way having its own set of degenerate cases. We consider two levels of complexity:

- **Solution complexity** deals with the number of discrete solutions a joint value can take. Introducing square roots and inverse sin/cos methods introduces two possible solutions. Sometimes this is unavoidable because the underlying kinematics dictate several possible solutions, but most of the time one of the solutions can be inconsistent with the rest of the kinematics.
- **Numerical complexity** deals with the number of operations required to compute a specific equation. Equations with the same solution complexity might not have different numerical complexities. Numerical complexity also penalizes divides since a potential degenerate case might occur.

When deciding which variable to solve for, the **solution complexity** is evaluated for each variable and a set of solutions having the same minimum complexity are chosen. Inside each of these equations, the one with the minimum **numerical complexity** is chosen.

Given the current set of known variables that have been solved, we search for equations with the following priority:

- $\cos j_d$ and $\sin j_d$ can be solved from two linearly independent equations polynomial in $\cos j_d$ and $\sin j_d$. Polynomials of degrees more than 1 are penalized.
- If no linearly independent equations exist, search for equations of the form $a \cos j_d + b \sin j_d = c$. This will allow us to solve for the correct angle within π radians.
- Finally resolve to equations of the form $(\cos j_d)^2 = a$, which can have up to four answers:

$$j_d = \{\cos^{-1}\sqrt{a}, -\cos^{-1}\sqrt{a}, \cos^{-1} -\sqrt{a}, -\cos^{-1} -\sqrt{a}\} \tag{4.6}$$

- Any solution that can be solved in closed-form.

If there are degenerate cases like divides by zero, an explicit branch occurs in the generated code that sets all the instances of the condition to zero and re-analyzes the equations. There are two categories of divides by zero. One where the condition can be directly evaluated to a value of a joint variable like $\cos j_d = 0$; the other cannot be solved so easy or is

not related to joint solutions like $p_x^2 + p_y^2$. The former allows us to explicitly create branches for $j_d = \frac{\pi}{2}$ and $j_d = -\frac{\pi}{2}$, which we will show below is necessary to get correct solutions. The latter divide by zero category makes it more difficult to propagate changes in compile time, so is penalized more.

For example, when presented with the following set of equations:

$$\cos j_0 = a_0 \tag{4.7}$$

$$a_1 * \sin j_1 + a_2 * \cos j_1 = a_3 \tag{4.8}$$

$$a_4 * \sin j_1 + a_5 * \sin j_1 = a_6 \tag{4.9}$$

solving for j_0 yields two solutions:

$$\{\cos^{-1} j_0, -\cos^{-1} j_0\}. \tag{4.10}$$

If the equations for j_1 are linearly independent, solving for j_1 yields:

$$sj_1 = \frac{a_3 a_5 - a_2 a_4}{a_1 a_5 - a_2 a_4} \tag{4.11}$$

$$cj_1 = \frac{a_1 a_4 - a_3 a_4}{a_1 a_5 - a_2 a_4} \tag{4.12}$$

$$j_1 = \textbf{atan2}(sj_1, cj_1) \tag{4.13}$$

There are several advantages to the solution of j_1 when compared with j_0:

- **atan2** function is extremely robust to zeros and infinities and will return a solution in the full circle $[0, 2\pi]$. It is available on all math libraries.
- The domain of $\cos^{-1}$ and $\sin^{-1}$ is $[-1, 1]$ making the functions unstable due to numerical imprecision. There are also two solutions, one of which might not be consistent with the robot kinematics.

Therefore solutions like Equation 4.11 should be prioritized over Equation 4.10; in fact, equations in the form of $\sin^{-1} x$, $\cos^{-1} x$, and $\sqrt{x}$ should be only used as a last resort when there is nothing else available.

In order to apply a solution in a real working IK solver, there needs to be guarantees that the solution will always produce the correct answer. Developing an IK solver that works 100% of the time is difficult because many degenerate cases arise. For example, it is a common mistake for researchers to apply Equation 4.11 as is without considering the degenerate case $a_1 a_5 - a_2 a_4 = 0$ when solving for sj_1 and cj_1. This **divide by zero** is a serious problem and

completely changes the priority equations have to be considered. As we will see in examples below, this can actually affect what variable is solved first. Unfortunately, we cannot just cancel out the denominators in the hope of eliminating the zero because

$$\textbf{atan2}(\frac{b_0}{c}, \frac{b_1}{c}) \neq \textbf{atan2}(b_0, b_1). \tag{4.14}$$

The sign of c can affect the quadrant the angle is returned in, thus potentially returning an angle that is π radians from the correct result. Therefore, the correct approach is to first evaluate the potential **divide by zero** conditions and branch to a different set of solutions where the **divide by zero** is delayed and other variables are prioritized; eventually a different a solution will be found for the original variable.

In the following sections we will be using the Kuka KR5 R850 industrial manipulator (Figure 4.4 and the PR2 personal service robot 4.3 for explaining inverse kinematics generation. The PR2 service robot has seven joints, which adds a redundancy to the kinematics. This issue will be discussed in further detail below, but for now we assume that the value of the first joint j_0 is set by the user.

4.1.3 Solving 3D Translation IK

There are two types of equations setup for 3D translation using Equation 4.4:

$$t_{ee,i} - t_i = 0 \tag{4.15}$$

$$\|t_{ee,i}\|^2 - \|t_i\|^2 = 0 \tag{4.16}$$

We show how to solve a subset of the joints of the PR2 for translation. We treat the wrist position where the last three joints intersect as the translation target. The PR2 kinematics equations are

$$t_0 = \begin{pmatrix} 0.1c_0+0.4c_0c_1+0.321c_0c_1c_3-0.321s_0s_2s_3-0.321c_0c_2s_1s_3 \\ 0.1s_0+0.4c_1s_0+0.321c_0s_2s_3+0.321c_1c_3s_0-0.321c_2s_0s_1s_3 \\ -0.4s_1-0.321c_3s_1-0.321c_1c_2s_3 \end{pmatrix}, t_{ee,0} = \begin{pmatrix} p_x \\ p_y \\ p_z \end{pmatrix}$$

$$t_1 = \begin{pmatrix} 0.1+0.4c_1+0.321c_1c_3-0.321c_2s_1s_3 \\ 0.321s_2s_3 \\ -0.4s_1-0.321c_3s_1-0.321c_1c_2s_3 \end{pmatrix}, t_{ee,1} = \begin{pmatrix} p_xc_0+p_ys_0 \\ p_yc_0-p_xs_0 \\ p_z \end{pmatrix}$$

$$t_4 = \begin{pmatrix} 0.4+0.321c_3 \\ 0.321s_2s_3 \\ -0.321c_2s_3 \end{pmatrix}, t_{ee,4} = \begin{pmatrix} -0.1c_1-p_zs_1+p_xc_0c_1+p_yc_1s_0 \\ p_yc_0-p_xs_0 \\ -0.1s_1+p_zc_1+p_xc_0s_1+p_ys_0s_1 \end{pmatrix}$$

$$t_6 = \begin{pmatrix} 0.321c_3 \\ 0 \\ -0.321s_3 \end{pmatrix}, t_{ee,6} = \begin{pmatrix} -0.4-0.1c_1-p_zs_1+p_xc_0c_1+p_yc_1s_0 \\ -0.1s_1s_2+p_yc_0c_2+p_zc_1s_2-p_xc_2s_0+p_xc_0s_1s_2+p_ys_0s_1s_2 \\ -0.1c_2s_1+p_xs_0s_2+p_zc_1c_2-p_yc_0s_2+p_xc_0c_2s_1+p_yc_2s_0s_1 \end{pmatrix}$$

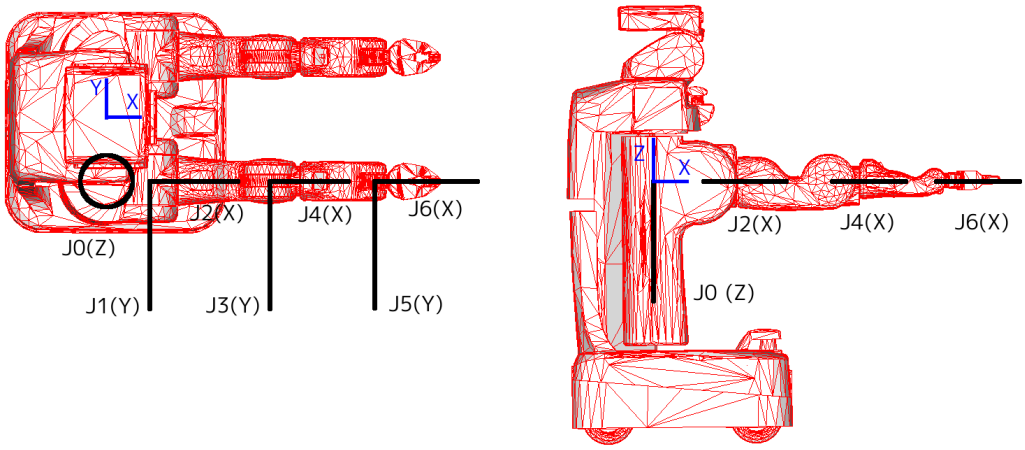


Figure 4.3: The labeled joints (black) of the PR2 robot's right arm.

We treat j_0 as the free parameter, so c_0 and s_0 are known. The first solution found is for j_3 where we get the constraint:

$$\|t_{ee,4}\|^2 - \|t_4\|^2 = -0.066959 - 0.08c_1 + 0.2p_x c_0 + 0.2p_y s_0 - 0.8p_z s_1 + 0.8p_x c_0 c_1 + 0.8p_y c_1 s_0 - {p_x}^2 - {p_y}^2 - {p_z}^2 = 0$$

$$\Rightarrow c_3 = -0.9853621495 - 0.7788161994 c_0 p_x - 0.7788161994 p_y s_0 + 3.894080997({p_x}^2 + {p_y}^2 + {p_z}^2)$$

This equation has two solutions, and no other constraint on j_3 can be computed, so we add both as possible solutions. Geometrically thinking, j_3 is the elbow of the robot, and usually there are two solution when computing the length of the base to the arm tip.

Treating j_3 as a known value, the next solvable variable is j_2:

$$t_{ee,1} - t_1 = p_x s_0 - p_y c_0 + 0.321 s_2 s_3 = 0$$

$$\Rightarrow s_2 = -3.115264797 \frac{p_x s_0 - c_0 p_y}{s_3}$$

There are no other constraints that can be formed, so we compute two solutions for j_2 from $\sin j_2$. Geometrically this makes sense because two angles can achieve the same direction in two different ways; once one is set, the other angle is uniquely determined. However, you'll note that there is a potential **divide by zero** problem with $\sin j_3$. Because this is the only possible solution, we have to compute a new set of constraints given $j_3 \in 0, \pi$. For example, we get the following new equations when setting $j_3 = 0$:

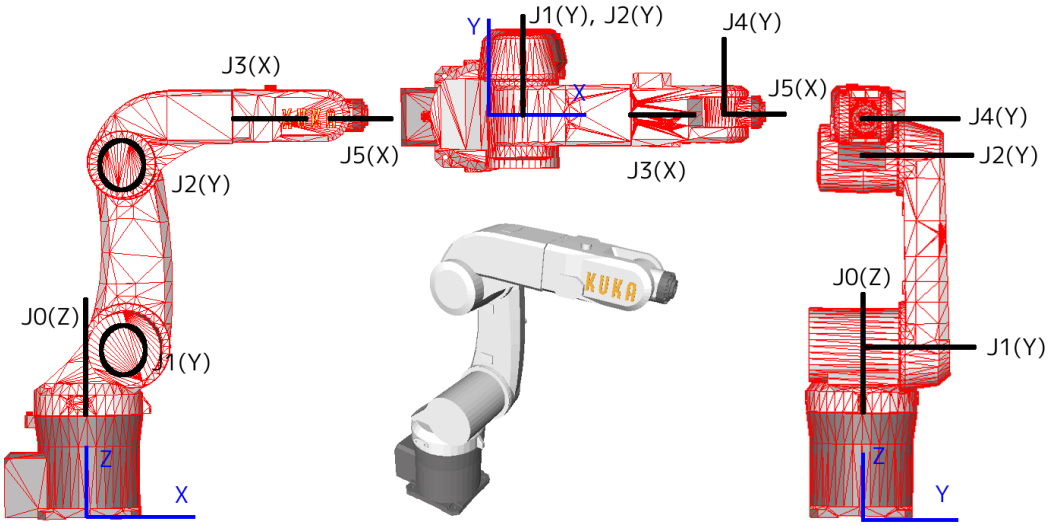


Figure 4.4: The labeled joints (black) of the Kuka KR5 R850 industrial robot.

$$t'_0 = \begin{pmatrix} 0.1c_0+0.721c_0c_1 \\ 0.1s_0+0.721c_1s_0 \\ -0.721s_1 \end{pmatrix}, t'_{ee,0} = \begin{pmatrix} p_x \\ p_y \\ p_z \end{pmatrix}$$
$$t'_1 = \begin{pmatrix} 0.1+0.721c_1 \\ 0 \\ -0.721s_1 \end{pmatrix}, t'_{ee,1} = \begin{pmatrix} p_xc_0+p_ys_0 \\ p_yc_0-1p_xs_0 \\ p_z \end{pmatrix}$$
$$t'_4 = \begin{pmatrix} 0.721 \\ 0 \\ 0 \end{pmatrix}, t'_{ee,4} = \begin{pmatrix} -0.1c_1-1p_zs_1+p_xc_0c_1+p_yc_1s_0 \\ p_yc_0-1p_xs_0 \\ -0.1s_1+p_zc_1+p_xc_0s_1+p_ys_0s_1 \end{pmatrix}$$

The solutions to the remaining variables become clear with

$$j'_1 = \mathbf{atan2}(-1.386962552p_z, -0.1386962552+1.386962552c_0p_x+1.386962552p_ys_0) \tag{4.17}$$

Going back to the original non-zero branch, with j_2 and j_3 computed, the final solution to j_1 becomes

$$j_1 = \mathbf{atan2}\left(\frac{p_z(0.1-c_0p_x-p_ys_0)-0.321c_2s_3(0.4+0.321c_3)}{-(0.4+0.321c_3)(0.1-c_0p_x-p_ys_0)+0.321c_2p_zs_3}, \frac{(0.4+0.321c_3)^2-{p_z}^2}{-(0.4+0.321c_3)(0.1-c_0p_x-p_ys_0)+0.321c_2p_zs_3}\right) \tag{4.18}$$

which is unique. Therefore the translation component of the PR2 can have up to 4 unique solutions.

We also provide the equations to the Kuka manipulator in order to show the similarities of the analyses. The first thing worth nothing is that the joint axes for j_1 and j_2 are aligned, which hints that j_0 can be decoupled from the computations. The kinematics equations are:

$$t_0 = \begin{pmatrix} 0.075c_0+0.365c_0s_1+0.405c_0c_1c_2+0.09c_0c_1s_2+0.09c_0c_2s_1-0.405c_0s_1s_2 \\ -0.075s_0-0.365s_0s_1+0.405s_0s_1s_2-0.405c_1c_2s_0-0.09c_1s_0s_2-0.09c_2s_0s_1 \\ 0.335+0.365c_1+0.09c_1c_2-0.405c_1s_2-0.405c_2s_1-0.09s_1s_2 \end{pmatrix}, t_{ee,0} = \begin{pmatrix} p_x \\ p_y \\ p_z \end{pmatrix}$$

$$t_1 = \begin{pmatrix} 0.075+0.365s_1+0.405c_1c_2+0.09c_1s_2+0.09c_2s_1-0.405s_1s_2 \\ 0 \\ 0.335+0.365c_1+0.09c_1c_2-0.405c_1s_2-0.405c_2s_1-0.09s_1s_2 \end{pmatrix}, t_{ee,1} = \begin{pmatrix} p_xc_0-1p_ys_0 \\ p_xs_0+p_yc_0 \\ p_z \end{pmatrix}$$

$$t_4 = \begin{pmatrix} 0.405c_2+0.09s_2 \\ 0 \\ 0.09c_2-0.405s_2 \end{pmatrix}, t_{4,ee} = \begin{pmatrix} 0.335s_1-0.075c_1-1p_zs_1+p_xc_0c_1-1p_yc_1s_0 \\ p_xs_0+p_yc_0 \\ -0.365-0.335c_1-0.075s_1+p_zc_1+p_xc_0s_1-1p_ys_0s_1 \end{pmatrix}$$

We get two solutions with j_0:

$$j_0 = \{-\mathbf{atan2}(-p_y, -p_x)\ ,\ \pi - \mathbf{atan2}(-p_y, -p_x)\} \tag{4.19}$$

This allows us to solve for j_2:

$$a{=}0.6190937723{+}2.212228413p_z{+}0.4952750178c_0p_x{-}0.4952750178p_ys_0{-}3.301833452({p_x}^2{+}{p_y}^2{+}{p_z}^2)$$

$$\Rightarrow j_2{=}\{-2.92292370771547{-}\sin^{-1} a\ ,\ 0.218668945874327{+}\sin^{-1} a\}$$

Finally, we solve for j_1:

$$b{=}0.075{+}p_ys_0{-}c_0p_x$$

$$j_1{=}\mathbf{atan2}\Big(\frac{b(0.365{+}0.09c_2{-}0.405s_2){-}(0.335{-}p_z)(0.405c_2{+}0.09s_2)}{-(0.335{-}p_z)^2{-}b^2}, \frac{(0.335{-}p_z)(0.365{+}0.09c_2{-}0.405s_2){+}(0.405c_2{+}0.09s_2)b}{-(0.335{-}p_z)^2{-}b^2}\Big)$$

Conic Sections

The constraints in Equation 4.15 might not be so forgiving as to always guarantee there is one joint variable singled out. It is very common for manipulators to have two joint variables coupled which give rise to the following equations:

$$A_{j,0} + A_{j,1}c_0 + A_{j,2}s_0 + A_{j,3}c_1 + A_{j,4}s_1 + A_{j,5}c_0c_1 + A_{j,6}c_0s_1 + A_{j,7}s_0c_1 + A_{j,8}s_0s_1 = 0 \tag{4.20}$$

where $A_{j,k}$ is a set of constants of the j^{th} constraint computed from all frames of reference of Equation 4.15. Treating c_0, s_0, c_1, and s_1 as independent unknown variables, it is very difficult to find a closed form solution of this family of equations. However, by taking advantage of the property that $c_k^2 + s_k^2 = 1$, we can solve Equation 4.20. First we treat the set of equations as linear with respect to the pairwise variables c_0c_1, c_0s_1, s_0c_1, and s_0s_1, and solve for them using basic linear algebra techniques. This allows us to single out the pairwise variables like this:

$$c_0c_1 = B_{0,0}c_0 + B_{0,1}s_0 + B_{0,2}c_1 + B_{0,3}s_1 + B_{0,4} \tag{4.21}$$

$$c_0s_1 = B_{1,0}c_0 + B_{1,1}s_0 + B_{1,2}c_1 + B_{1,3}s_1 + B_{1,4} \tag{4.22}$$

$$s_0c_1 = B_{2,0}c_0 + B_{2,1}s_0 + B_{2,2}c_1 + B_{2,3}s_1 + B_{2,4} \tag{4.23}$$

$$s_0s_1 = B_{3,0}c_0 + B_{3,1}s_0 + B_{3,2}c_1 + B_{3,3}s_1 + B_{3,4} \tag{4.24}$$

$$0 = B_{4,0}c_0 + B_{4,1}s_0 + B_{4,2}c_1 + B_{4,3}s_1 + B_{4,4} \tag{4.25}$$

Although there are four equations here, there is two underlying degrees of freedom, meaning that two equations can be formulated as a combination of the other two. In order to remove the coupled variables from the left side, we pick two equations, square both sides, and add them together. For example, (Equation 4.21)^2 + (Equation 4.22)^2 yields c_0^2 for the left side. Applying the transformation for all valid combinations gives:

$$
\begin{aligned}
c_0^2 &= D_{0,0}c_0^2 + D_{0,1}c_0s_0 + D_{0,2}c_0 + D_{0,3}s_0 + D_{0,4} \\
s_0^2 &= D_{1,0}c_0^2 + D_{1,1}c_0s_0 + D_{1,2}c_0 + D_{1,3}s_0 + D_{1,4} \\
c_1^2 &= D_{2,0}c_1^2 + D_{2,1}c_1s_1 + D_{2,2}c_1 + D_{2,3}s_1 + D_{2,4} \\
s_1^2 &= D_{3,0}c_1^2 + D_{3,1}c_1s_1 + D_{3,2}c_1 + D_{3,3}s_1 + D_{3,4}
\end{aligned} \tag{4.26}
$$

where $D_{j,k}$ is constant for $k < 4$ and $D_{j,4}$ is the remainder. If one of $D_{j,4}$ is a constant value, we can solve for cos and sin of a single variable by formulating the problem as the intersection of two conic sections setting $x = \cos j_k$ and $y = \sin j_k$:

$$
E_0x^2 + E_1xy + E_2y^2 + E_3x + E_4y + E_5 = 0 \Rightarrow \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}^T \begin{bmatrix} E_0 & \frac{E_1}{2} & \frac{E_3}{2} \\ \frac{E_1}{2} & E_2 & \frac{E_4}{2} \\ \frac{E_3}{2} & \frac{E_4}{2} & E_5 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \Rightarrow \mathbf{x}^T C_0 \mathbf{x} = 0
$$

$$
x^2 + y^2 - 1 = 0 \Rightarrow \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}^T \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \Rightarrow \mathbf{x}^T C_1 \mathbf{x} = 0 \tag{4.27}
$$

We first note that any solution $\mathbf{x}$ that satisfies both C_0 and C_1, will also satisfy a linear combination of them:

$$
\mathbf{x}^T(C_0 + \lambda C_1)\mathbf{x} = 0 \tag{4.28}
$$

The goal is to produce a third conic $C_2 = C_0 + \lambda C_1$ such that C_2 is degenerate. A degenerate conic has solutions that are: a point, a line, or two intersecting lines. Such geometric primitives are much easier to work with and we can immediately find the intersection of them with the unit circle represented by C_1. C_2 s degenerate when its determinant is 0. This amounts to solving a cubic equation in λ:

$$
\begin{aligned}
|C_0 + \lambda C_1| &= 0 \\
\Rightarrow \lambda^3 + F_2\lambda^2 + F_1\lambda + F_0 &= 0 \\
F_2 &= E_0 + E_2 - E_5 \\
F_1 &= E_0E_2 - E_0E_5 - E_2E_5 + \frac{E_3^2 + E_4^2 - E_1^2}{4} \\
F_0 &= -E_0E_2E_5 + \frac{E_0{E_4}^2 - E_1E_3E_4 + E_2{E_3}^2 + E_5{E_1}^2}{4}.
\end{aligned}
$$

which can be analytically solved using methods in [Weisstein (1999-present)].

For every real solution to λ, we compute the degenerate conic C_2 and find its null space $NS(C_2)$, which we can use to compute the line:

$$\forall v \in NS(C_2), \quad \begin{bmatrix} x & y & 1 \end{bmatrix} \cdot v = G_0 x + G_1 y + G_2 = 0. \tag{4.29}$$

Intersecting with the unit circle gives us the following quadratic equation:

$$(G_1 y + G_2)^2 + G_0^2 y^2 = G_0^2 \tag{4.30}$$

As an example of an application to these equations, we take the PR2 kinematics, but this time choose j_2 to be the free parameter. This gives us many coupled equations, with the following being the simplest:

$$\begin{aligned} 0.321 s_2 s_3 &= c_0 p_y - p_x s_0 \\ 0.2568 c_3 &= -0.253041 - 0.2 c_0 p_x - 0.2 p_y s_0 + {p_x}^2 + {p_y}^2 + {p_z}^2 \end{aligned}$$

Squaring all sides and adding the two equations gives a conic equation in c_0 and s_0:

$$E_0 c_0^2 + E_1 c_0 s_0 + E_3 c_0 + E_4 s_0 + E_5 = 0$$

$$E_0 = 9.704874759(\frac{{p_x}^2}{{s_2}^2} - \frac{{p_y}^2}{{s_2}^2}) + 0.6065546724({p_y}^2 - {p_x}^2)$$

$$E_1 = -1.213109345 p_x p_y + 19.40974952 \frac{p_x p_y}{{s_2}^2}$$

$$E_2 = 0$$

$$E_3 = -1.534832009 p_x + 6.065546724(p_x {p_y}^2 + p_x {p_z}^2 + {p_x}^3)$$

$$E_4 = -1.534832009 p_y + 6.065546724(p_y {p_x}^2 + p_y {p_z}^2 + {p_y}^3)$$

$$E_5 = 0.02906143424 - 9.704874759 \frac{{p_x}^2}{{s_2}^2} + 7.067605371 {p_y}^2 + 7.674160043({p_x}^2 + {p_z}^2) - 15.16386681({p_x}^4 + {p_y}^4 + {p_z}^4 + 2{p_x}^2{p_y}^2 + 2{p_x}^2{p_z}^2 + 2{p_y}^2{p_z}^2)$$

Once we solve for the first joint variable, we can continue searching for the others starting again at Equation 4.15. The divide by $\sin j_2$ is clear in these equations, so we specifically test the IK when $j_2 \in \{0, \pi\}$. For example, setting $j_2 = 0$ and re-solving allows us to solve for j_0 immediately:

$$j_0 \in \{-\textbf{atan2}(-p_y, p_x), \pi - \textbf{atan2}(-p_y, p_x)\} \tag{4.31}$$

4.1.4 Solving 3D Rotation IK

There are two types of equations setup for 3D rotation:

$$R_{ee,i} = R_i \tag{4.32}$$

$$\frac{R_{ee,i}(j_1)}{R_{ee,i}(j_2)} = \frac{R_i(j_1)}{R_i(j_2)} \tag{4.33}$$

where j_1 and j_2 index one of the 9 elements in the rotation matrix. Unlike translation, the rotation solution can be solved without changing the frame of reference. However, there much more frequent degenerate cases and divides by zero, which makes the final solutions complex. A geometric analysis of the problem will show that three angles can provide for at least 2 unique solutions for every rotation matrix. If there is ever more than 2 unique solutions, then it is because two joint axes have aligned, and an infinite amount of unique solutions can be computed.

Here we use the last three axes of the PR2 and Kuka robots to show how 3D rotations are solved. The PR2 rotation kinematics are:

$$R_8 = \begin{bmatrix} c_5 & s_5 s_6 & c_6 s_5 \\ s_4 s_5 & c_4 c_6 - c_5 s_4 s_6 & -c_4 s_6 - c_5 c_6 s_4 \\ -c_4 s_5 & c_6 s_4 + c_4 c_5 s_6 & -s_4 s_6 + c_4 c_5 c_6 \end{bmatrix}, R_{ee,8} = \begin{bmatrix} r_{00} & r_{01} & r_{02} \\ r_{10} & r_{11} & r_{12} \\ r_{20} & r_{21} & r_{22} \end{bmatrix} \tag{4.34}$$

The most apparent equation here is $c_5 = r_{00}$ because it *almost* allows computation of j_5. In fact, if $r_{00} \in \{\pm 1\}$, we can deduce that $s_5 = 0$, so it is possible to compute j_5 directly. Because one of our constraints from Equation 4.32 involves dividing by elements of R, one very dangerous possibility is that any one of c_4, s_4, c_5, s_5, c_6,or s_6 can appear in the denominator; and if zero, will make the equation degenerate. Therefore, a the rule of thumb when generating the equations it take advantage of *all* computed constraints on the joint variables. From the above constraint, if $r_{00} \in \{\pm 1\}$, then we know for a fact that $s_5 = 0$ and can solve for j_5, therefore we create an extra branch in the generated code setting j_5 to 0 and π. This gives the following new matrix evaluated at $j_5 = 0$:

$$R'_8 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & c_4 c_6 - s_4 s_6 & -c_4 s_6 - c_6 s_4 \\ 0 & c_4 s_6 + c_6 s_4 & c_4 c_6 - s_4 s_6 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(j_4 + j_6) & -\sin(j_4 + j_6) \\ 0 & \sin(j_4 + j_6) & \cos(j_4 + j_6) \end{bmatrix} \tag{4.35}$$

which shows that both j_4 and j_6 are aligned and can be solved by:

$$j_4 + j_6 = \mathbf{atan2}(r_{21}, r_{11}) \tag{4.36}$$

This type of analysis shows that **ikfast** can also return an infinite amount of solutions by explicitly storing the relationship between joint values. Testing $j_5 = \pi$ would yield an opposite relationship $j_4 - j_6 = \mathbf{atan2}(r_{21}, r_{11})$. It is worth nothing here that these relationships would have been completely ignored if we did not pursue the $c_5 = r_{00}$ constraint.

Back to the original problem, we get these possible solutions:

$$j_4 = \mathbf{atan2}(r_{10}s_5, -r_{20}s_5)$$
$$j_6 = \mathbf{atan2}(r_{01}s_5, r_{02}s_5)$$

Both involve multiplication of s_5, which would be meaningless if $s_5 = 0$. However from the above discussion, we are guaranteed that $s_5 \neq 0$, so the equations are valid. Because **atan2** is a divide, we can ignore the absolute value of s_5 from the computation and only concern ourselves with its sign. However, that cannot be computed at all. Further analysis of R_8 would yield that the solutions are consistent with s_5 being positive or negative. Therefore, we compute two solutions for j_4:

$$j_4 = \{\mathbf{atan2}(r_{10}, -r_{20}) \ , \ \pi + \mathbf{atan2}(r_{10}, -r_{20})\} \tag{4.37}$$

Treating j_4 as a known variable allows us to compute the final solutions:

$$j_5 = \mathbf{atan2}\left(\frac{r_{10}}{s_4}, r_{00}\right)$$
$$j_6 = \mathbf{atan2}\left(\frac{r_{01}}{s_5}, \frac{r_{02}}{s_5}\right)$$

A divide by zero condition is detected $\sin j_4 = 0$, which by going through similar logic described above allows us to compute a different set of solutions:

$$j_5 = \mathbf{atan2}\left(\frac{-r_{20}}{c_4}, r_{00}\right)$$
$$j_6 = \mathbf{atan2}\left(\frac{-r_{12}}{c_4}, \frac{r_{11}}{c_4}\right)$$

In this branch, we know that $\cos j_4 \neq 0$ since we are assuming $\sin j_4 = 0$.

4.1.5 Solving 6D Transformation IK

There are two classes of kinematic chains that can be solved using the above analyses as sub-modules by assuming the translation and rotation components are separable. Separability allows much simpler solutions where the most complex polynomials have to be solved are quadratic. The first type is when the last 3 joint axes intersect at a common point (Figure 4.5a), this implies that

$$\begin{bmatrix} R_{ee,6}(j_0, j_1, j_2) & t_{ee,6}(j_0, j_1, j_2) \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} R_6(j_3, j_4, j_5) & t_6 \\ 0 & 1 \end{bmatrix} \tag{4.38}$$

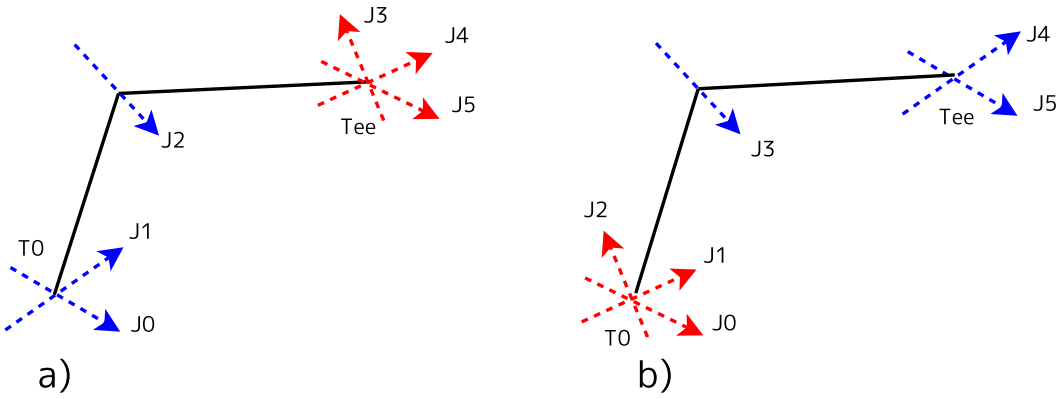


Figure 4.5: Two types of separable kinematic chains, which allow rotation joints (red) to be solved separately from translation joints (blue). T_0 is the base link frame, T_{ee} is the end effector link frame.

where t_6 is constant. This allows us to build up the constraints for just the j_0, j_1, j_2 variables by using Equation 4.15 for all $i <= 6$. After the first 3 joint values are solved, we can treat $R_{ee,6}(j_0, j_1, j_2)$ as a known constant matrix and solve directly for $R_6(j_3, j_4, j_5)$ by using constraints built by Equation 4.32.

The second type is when the first 3 joints intersect at a common point (Figure 4.5b), which produces the following separation:

$$\begin{bmatrix} R_{ee,6}(j_0, j_1, j_2) & t_{ee,6} \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} R_6(j_3, j_4, j_5) & t_6(j_3, j_4, j_5) \\ 0 & 1 \end{bmatrix}. \tag{4.39}$$

We first solve for the translation component by building up all constraints in Equation 4.15 for all $i >= 6$. Then we treat $R_6(j_3, j_4, j_5)$ as a known constant matrix and solve for $R_{ee,6}(j_0, j_1, j_2)$.

4.1.6 Solving 4D Ray IK

A ray is a 3D line with a preference to direction. It is defined by

$$r(t) = p + s\, d \qquad s \in \mathbb{R} \tag{4.40}$$

where p is any position along the ray and d is a unit direction for a total of four degrees of freedom. Given the target ray $\mathbf{r}_n(s) = p_n + s\, d_n$ in the coordinate system of the last link, the inverse kinematics of a ray is the problem is matching $\mathbf{r}_n$ to a globally specified ray $\mathbf{r}_{ee}(s) = p_{ee} + s\, d_{ee}$ (Figure 4.6):

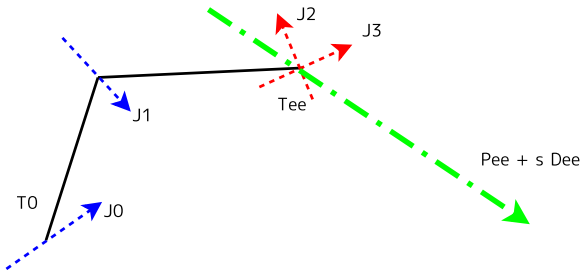


Figure 4.6: The parameterization of the ray (green) with the last two axes interesting.

$$\exists_{s\in\mathbb{R}}\ p_{ee} + s\ d_{ee} = T_0 J_0 T_1 J_1 T_2 J_2 ... T_n \begin{bmatrix} p_n \\ 1 \end{bmatrix} \tag{4.41}$$

$$d_{ee} = T_0 J_0 T_1 J_1 T_2 J_2 ... T_n \begin{bmatrix} d_n \\ 0 \end{bmatrix} \tag{4.42}$$

Here we present a solution to the inverse ray kinematics problem for the case that:

- The last 2 joints intersect at a common point.
- The problem is transformed so that the position of the ray in the last link's coordinate system is at the origin: $T_n \begin{bmatrix} p_n \\ 1 \end{bmatrix} = 0$. If this is not the case, then T_n could be modified without loss of generality to satisfy this condition.

These constraints allow us to separate ray position from ray direction:

$$\exists_{s\in\mathbb{R}}\ R_{ee,4}(j_0, j_1)\ (p_{ee} + s\ d_{ee}) + t_{ee,4}(j_0, j_1) = t_4 \tag{4.43}$$

$$R_{ee,4}(j_0, j_1)\ d_{ee} = R_4(j_2, j_3)\ d_n. \tag{4.44}$$

The fundamental difference between between ray IK and 3D translation/3D rotation IK is that the ray equations for the translation component are not represented by an equality sign. Equation 4.43 is 3 dimensions, but there are only two real constraints; ignoring any equation can lead to completely wrong solution. The equations hold for only one value of s, which is necessary to normalize the input ray position p_{ee} with that of the ray forward kinematics. However, dealing explicitly with s introduces new dependencies: the solutions to j_0 and j_1 depend on s, and the solutions to j_2 and j_3 depend on j_0 and j_1.

We eliminate the free parameter s in Equation 4.43 by taking the cross product with the direction at each frame of reference:

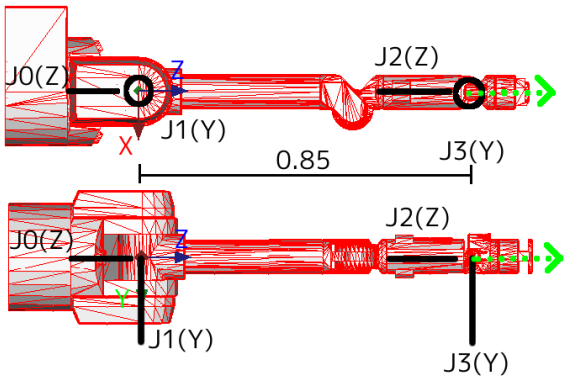


Figure 4.7: The parameterization of the ray (green) using a sub-chain of the Barrett WAM arm.

$$\begin{aligned}(R_{ee,i}\, d_{ee}) \times R_{ee,i}\,(p_{ee} + s\, d_{ee}) &= (R_{ee,i} d_n) \times t_i \\ (R_{ee,i}\, d_{ee}) \times (R_{ee,i}\, p_{ee}) &= (R_{ee,i} d_n) \times t_i \\ R_{ee,i}\,(d_{ee} \times p_{ee}) &= (R_{ee,i} d_n) \times t_i \end{aligned} \tag{4.45}$$

We find a system of equations for j_0 and j_1. Because there is coupling due to the cross product, the system of constraints is a conic section solvable using the techniques in Section 4.1.3. For direction, the equations become

$$d_{const} = R_4(j_2, j_3)\, d_n, \tag{4.46}$$

which is a sub-problem of solving the full 3D rotation.

We show an example of solving ray IK using a sub-chain of the Barrett WAM arm to represent the ray (Figure 4.7. From Equation 4.26 we compute an quadratic equation in only c_1, solving it produces:

$$\cos j_1 = \tfrac{1}{0.85}\left(p_z {d_x}^2 + p_z {d_y}^2 - p_x d_x d_z - p_y d_y d_z \pm \sqrt{0.85^2 - (p_x d_y - p_y d_x)^2 - (p_z d_x - p_x d_z)^2 - (p_z d_y - p_y d_z)^2)}\right)$$

We then compute an equation for j_0:

$$j_0 = \textbf{atan2}\left(-\frac{p_z d_y - p_y d_z - 0.85 c_1 d_y}{d_z s_1}, -\frac{p_x d_z - p_z d_x + 0.85 c_1 d_x}{d_z s_1}\right) \tag{4.47}$$

Here we detect another divide by zero situation and handle it by setting j_1 to $0, \pi$. Geometrically this aligns axes j_0 and j_2 allowing for an infinite number of solutions. produces the following solution:

$$a = \mathbf{atan2}\left(-0.85d_y + p_z d_y - p_y d_z, -0.85d_x + p_z d_x - p_x d_z\right)$$
$$j_0 = \{-a, \pi - a\}$$

The direction equations are formulated as:

$$c_2 s_3 = d'_x$$
$$s_2 s_3 = d'_y$$
$$c_3 = d'_z$$

where $\mathbf{d}'$ is computed from j_0 and j_1:

$$d'_0 = d_z s_1 + c_0 c_1 d_x - c_1 d_y s_0$$
$$d'_1 = c_0 d_y + d_x s_0$$
$$d'_2 = c_1 d_z + d_y s_0 s_1 - c_0 d_x s_1$$

First d_z is checked for being on the boundary $\{\pm 1\}$, otherwise it is evaluated as:

$$j_2 = \mathbf{atan2}\left(d'_y, -d'_x\right) \tag{4.48}$$

Then j_3 is evaluated as:

$$j_3 = \mathbf{atan2}\left(\frac{d'_y}{s_2}, d'_z\right) \tag{4.49}$$

assuming $\sin j_2 \neq 0$, or otherwise it is

$$j_3 = \mathbf{atan2}\left(-\frac{d'_x}{c_2}, d'_z\right) \tag{4.50}$$

4.1.7 Handling Redundancies

When solving IK, the robot might have more degrees of freedom than the IK parameterization. For example, the Barrett WAM has 7 degrees of freedom, but IK only requires 6. In this case, we pick a joint that has *the least importance* and assume that it is set by the user before running the IK. We call these joints **free joints**, the joints the IK is solving for are called **active joints**. During planning, the range of the free joints is discretized and all values are tested until a solution satisfying the joint limits, collision, and planning constraints is found. The discretization of this **free space** depends on how important the joint is to

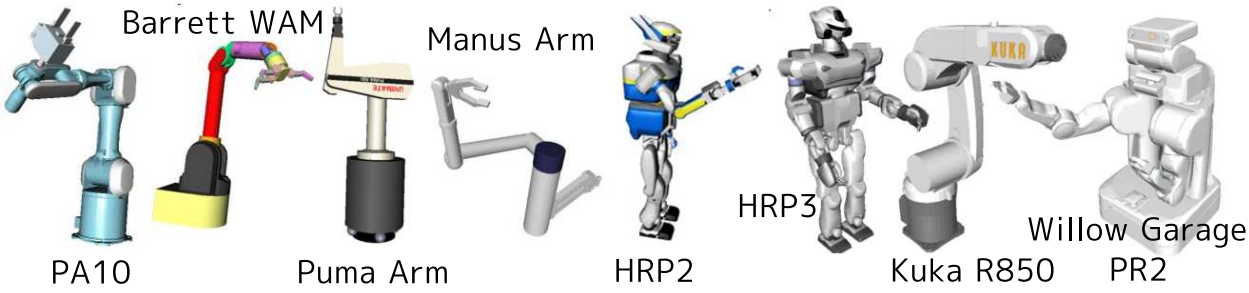


Figure 4.8: Several robot platforms whose inverse kinematics can be solved by **ikfast**.

the IK solution. In order to make the searching as quick as possible, the discretization value should be as high as possible without sacrificing solution feasibility. A good rule of thumb is that joints lying closer to the base are more important.

However, we cannot just get away with choosing the farthest joint from the base; some joints cannot be picked as free because they can destroy the independence between the translational and rotational components. For example, a 6D IK solution presented here requires the first 3 or last 3 active joints to intersect at a common point. The general heuristic we use for automatically choosing free joints is to start from the joints farthest from the base and continue to choose closer until a solution is found.

Even though free joints are set by the user, degenerate cases due to their values can still occur. For example, the Barrett WAM has 4 joints for solving the translation component, and choosing the free joint to be j_2, we can see that joint axes can align and create degenerate cases when $j_2 \in 0, \frac{\pi}{2}, pi, \frac{3\pi}{2}$. We create a special case for each degenerate value and re-solve the IK equation from the start. Finding these degenerate values can be very challenging at times; however most, if not all, robots have joint axes orthogonal to each other, so it is safe to assume that configurations become degenerate on on all four $\frac{\pi}{2}$ boundaries. In fact such checks, even if not necessary, will only make the solution more robust with the cost of computation time.

4.1.8 IKFast Results

Figure 4.8 shows several types of robot arms where ikfast can successfully solve the solutions. For each robot, we test just running the ikfast solver and its success rate (Table 4.1). Success is measured by first sampling 1000-10000 times a random configuration and inputting the destination of the robot manipulator to the IK function. If the robot has free joint values, then these values are discretized and sampled. Failures are divided into returning a wrong solution and failing to find a solution. The former is really dangerous; from experiments,

	Computation	Success	Discretization
Mitsubishi PA10	$7\mu s$	100%	0.1 radians
Barrett WAM	$6\mu s$	100%	0.1 radians
Puma Arm	$6\mu s$	100%	
Manus Arm	$6\mu s$	100%	
HRP2	$6\mu s$	99.5%	0.1 radians
Kuka R850	$5\mu s$	100%	
Willow Garage PR2	$6\mu s$	98.2%	0.1 radians
Willow Garage PR2	$6\mu s$	99.2%	0.02 radians
WAM Ray (Figure 4.7)	$4\mu s$	99.99%	

Table 4.1: Average time for calling the ikfast solver with the success rate of returning correct joint values for transformation queries.

Parameter Name	Parameter Description
Free Joints	joints that are freely set, used for handling redundancies
Discretization	the discretization of the free joint ranges when searching
Precision	number of significant digits to compute kinematics solutions in
Rounding	the smallest number before being counted as zero

Table 4.2: Analytic Inverse Kinematics Parameters

ikfast **never** returns a wrong solution, so is not an issue here. Looking at the table, the manipulators with 6 joints succeed all the time. Manipulators with 7 joints had the free joint discretized at 0.1 radians so had failures because the discretization was not enough; setting a smaller discretization fixes the problems, but is not practical. The free joint of the WAM, PA10, and HRP2 manipulators was the 3rd joint. Both WAM and PA10 have very big joint ranges, so there is always a solution found. The HRP2 joint ranges are small, so there are times when a solution has not been found. The free joint for the PR2 is the first joint, which greatly affects the solution space. Consequently, we see that PR2 succeeds a less than its counterparts when the discretization value is the same.

Although each ikfast solution can compute the exact result, the number of operations involved in the computation can greatly influence the numerical imprecision that gets introduced due to floating-point operations. We have attempted to prioritize the searching of solutions so the most stable and most confident solutions are selected. Even if the optimal solutions are not always found, we have showed that the solutions are very simple, do not

require advanced math computations, and even minimize the number of divides that have to be done. Therefore, they are much more robust than the previous general linear algebra methods proposed for analytic inverse kinematics.

Table 4.2 shows the parameters of the ikfast algorithm. We've discussed how to automatically set free joints and discretization values. As defaults, we compute all values to 10 significant digits and have a default zero rounding at 10^{-7}.

ikfast fundamentally changes the way researchers approach manipulation planning problems. Instead of relying on gradient-based methods plagued with numerical errors and slow computation times, all researchers can safely integrate analytical IK into their planners and begin searching with all solutions.

4.2 Grasping

Grasping is one of the fundamental differences between motion planning and manipulation planning. Grasping places explicit workspace constraints on the motions of the robot gripper with respect to the target objects. Many grasping models have been proposed that analyze contact modes and parameterize the geometry and space of grasps. However, there are many advantages of using *grasp sets* rather than parameterized models like Support Vector Machines [Pelossof et al (2004)] or explicit goal regions [Berenson et al (2009b); Prats et al (2007b)]. Although it might be more compact to explicitly parameterize the space of good grasps, such methods can be susceptible to modeling error. A grasp can easily become invalid by moving the gripper just a few millimeters in the wrong direction, if the parameterizations try to fill in data holes, they could easily allow invalid grasps. Instead, we stress the need for representing all valid grasp spaces as sets. Planning and sampling from the sets is just as convenient as parameterized models. In our analyses we attempt to maintain generality by relying solely on sets of grasps, which can approximate a complex distribution really well, and sampling the set is a simple matter of weighted selection. Furthermore, grasp sets can be ordered based on priority and space exploration similar to how deterministic samplers like the van der Corput sequence progressively cover a space [LaValle (2006)].

Each grasp is parameterized by the preshape of the gripper q_{grasp}, its start transformation with respect to the target object T_{grasp}^{object}, and the direction of approach to the object v^{object}. This is the minimal information necessary for the grasp planners covered in Section 3.3 to move the gripper to the correct location. The analyses that happen after this point can be divided into three different components:

- **Picking a grasping strategy.** Once the gripper is at the desired start position with respect to the object, the grasping strategy determines how the gripper and its joints

move in order to grasp the object. This could involve analyses with sensors in the loop.

- **Evaluating Grasp Performance** - Once contact with the object has been determined, the performance of that contact needs to be evaluated to determine if a grasp is stable.
- **Searching the Space of all Grasps** - Even though the grasp computation is an offline process, the space of all grasps satisfying the evaluation criteria must be efficiently explored and the manifold must be represented by a method that can be easily used in a planner like in [Goldfeder et al (2007)].

In light of these components, we present an analysis of three different grasping strategies:

- **Stable Grasps** - We show to find a set of physically stable grasps for the object and gripper pair by analyzing contact forces.
- **Caging Grasps** - We show how to open doors and drawers without imposing force closure constraints, which greatly increases the robot's feasibility space to complete the task.
- **Insertion Grasps** - We show examples of how work with grasps that insert a part of the gripper into objects.

4.2.1 Force Closure Squeezing Strategy

One of the simplest grasping strategies shown in Figure 4.9 is to approach the object from a given direction and close the fingers until they cannot move anymore [Miller (2001)]. Touch sensors and feedback loops are not considered, and as long as the fingers can constantly provide torque, a simple geometric analysis of point contacts should suffice for grasping rigid objects with a rigid hand.

Once the contact points have been extracted, most researchers use a very conservative measure called force closure [Kragic et al (2001); Zheng and Qian (2006)] that checks to see if any force acting on the object can be compensated by the friction cones generated from the contact points. Any force applied from the friction cone at that point contact results can be described as a 6D point in the wrench space. By combining all the possible wrenches from all the contact points and computing their convex hull, it is possible to determine if all possible forces can be compensated by testing if the convex hull includes the origin.

From the definitions above, the space of all grasps is $5 + 2 + 1 + |q_{gripper}|$ DOF. $|q_{gripper}|$ are for the preshape of the gripper, 2DOF are for the direction of approach, and the 5DOF are for the starting pose of the gripper $T^{object}_{gripper}$ with respect to the object where 1DOF is for the distance along the direction and starts at infinity. The last DOF represents the *stand-off*

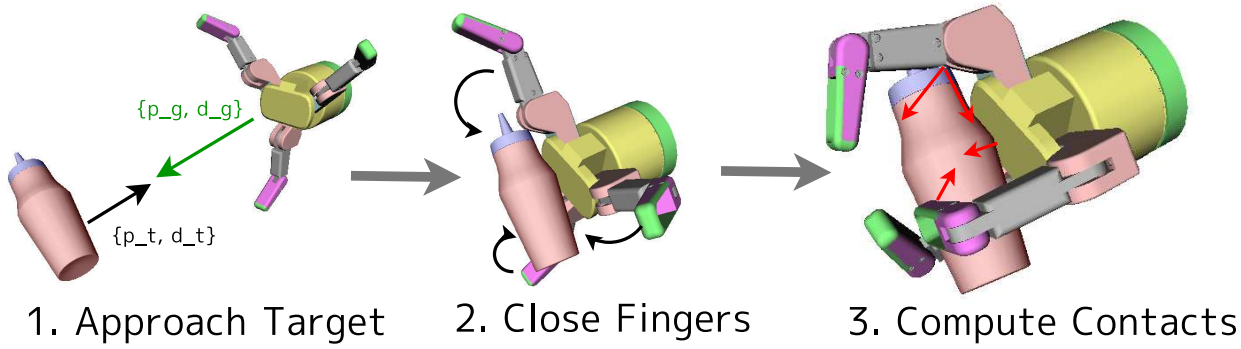


Figure 4.9: The gripper first approaches the target object based on a preferred approach direction of the gripper. Once close, the fingers close around the object until everything collides, then contact points are extracted.

distance that the gripper moves back when hitting the target and before closing its fingers [Berenson et al (2007)]. Even if the preshape is held constant, efficiently exploring an 8DOF space is very difficult [Ciocarlie and Allen (2009)].

Our method of parameterizing the space first defines a ray $\{p_g, d_g\}$ on the gripper's coordinate system, and a ray $\{p_t, d_t\}$ on the target. Because each ray is 4 DOF and we have two rays in two coordinate systems, we can represent the 6 DOF of the grasp space, the other 2 DOF are manually set as the roll around the approach axis and the final standoff. We compute the gripper pose $T^{object}_{gripper}$ by aligning the two rays together and making the directions point towards each other (Figure 4.9). Figure 4.10 shows how the rays $\{p_t, d_t\}$ on the target are sampled. We first start with sampling the surface of the object by casting rays from a bounding box (or sphere) to the target object. Then for each hit point, we compute the surface normal of the object. Using the surface normal, we can sample all directions that

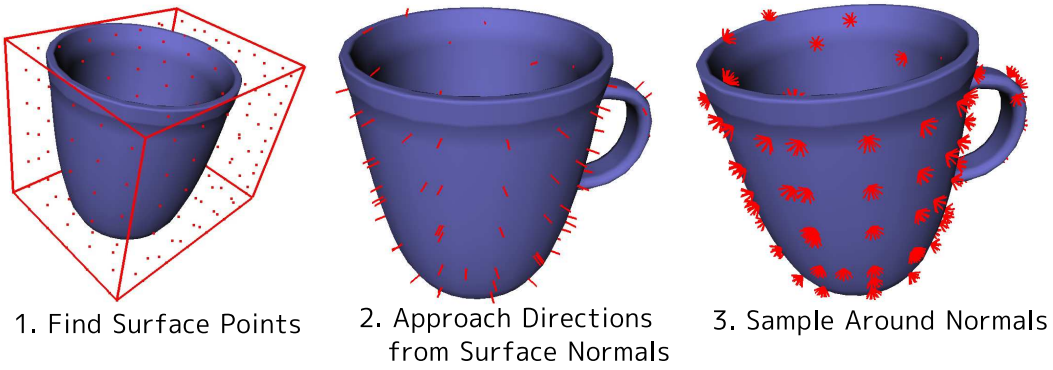


Figure 4.10: A simple way of parameterizing the grasp search space.

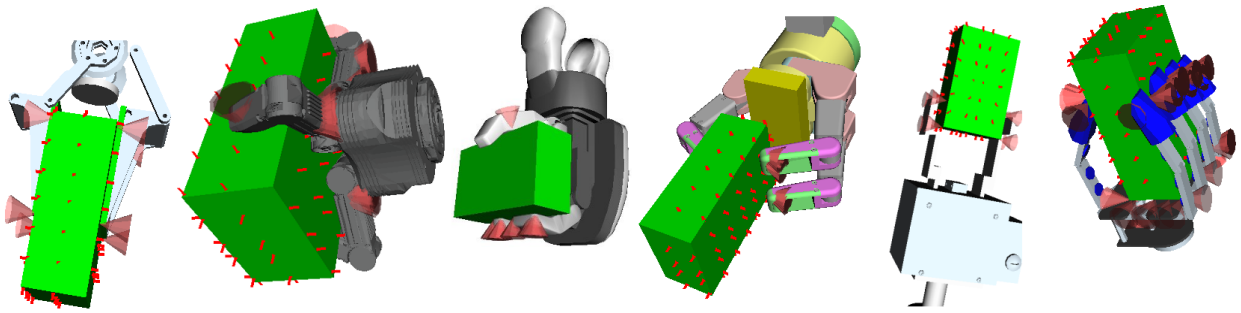


Figure 4.11: Examples of the types of robot hands that can be handled by the grasp strategy.

are within a fixed angle between each normal. Depending on how dense the sampling is, it can yield anywhere from 1,000 to 100,000 different locations to test grasps. This analysis is very effective and can be applied to many different grippers as shown in Figure 4.11.

For the 4DOF Barrett Hand, we usually find that 30% of the tested grasps yield force closure, so it is not necessary to explore the space much. In fact, we could get away with setting just one approach ray $\{p_g, d_g\}$ for the gripper. For the 1DOF HRP2 hand, the results are much more grim with less than 3% of the explored grasps yielding force closure. We attribute this with difficulties of 1DOF grippers with the grasp strategies. This requires several approach rays $\{p_g, d_g\}$ to be set for 1DOF grippers. Changing $\{p_g, d_g\}$ and the stand-off can have an immense effect on the type of grasps produced. Figure 4.12 shows how changing directions can prioritize between pinch grasps and power grasps. Avoiding certain parts of the gripper is another factor that should be considered when building grasp sets. Sometimes sensors can be attached to the gripper, and they need to be inserted in the geometry of the gripper for avoiding spurious contact with anything.

One of the biggest criticisms against computing force closure on point contacts is that *fragile grasps* can easily be classified as force closure stable grasps. Figure 4.13 shows several fragile grasps with high force closure scores with the HRP2 hand. In order to prune such grasps, many researchers have proposed adding noise to the grasp generation process and computing force closure afterward. If the randomly displaced grasps fail with a certain percentage, then they are rejected. Although such a method has been shown to work with the Barrett Hand where 30% of the grasps are in force closure [Berenson et al (2007)], it rejects **all** grasps for HRP2. The reason is because edge contacts that occur when the gripper's surface is aligned with the target surface are reduced to point contacts when even a small amount of noise is added.

We present a new robustness measurement method that computes a grasp's repeatability rather than its stability when the target object experiences translational and rotational

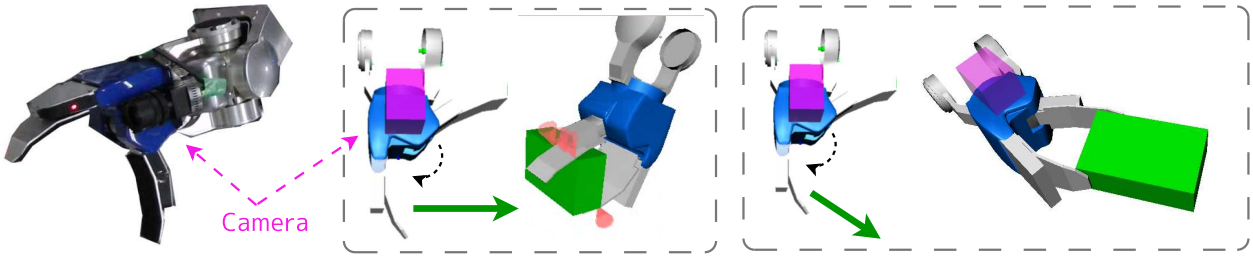


Figure 4.12: Grasps for the HRP2 gripper with a wrist camera mounted. Grasps contacting the wrist camera are rejected automatically. As the gripper approach varies the grasps change from power grasps to pinch grasps.

disturbances. A Δ_T offset in the target will produce a Δ_P offset for the final gripper position along with a Δ_Q joint offset. Using knowledge of the geometry of the object and kinematics of the gripper, we can compute the standard deviations of the movement of each point on the surface of the target σ_T and the gripper σ_G. The main intuition behind computing grasp repeatability is that for fragile grasps, the gripper will slip and therefore move more than the object has been disturbed:

$$\sigma_T \ll \sigma_G.$$

For stable grasps, the object and gripper deviations should be the same:

$$\sigma_T \approx \sigma_G.$$

ComputeGraspRepeatability (Algorithm 4.1) shows how to compute the repeatability statistics. It first takes the expected standard deviation of the target object as a parameter. Then for every iteration, it samples a random rotation and translation such that any point on the objects surface moves no more than σ_T. In order to compute rotations with the correct error, we rotate around a random axis with the rotation angle scaled by the inverse of the maximum distance of a point on the target surface. The grasp strategy is executed with the new target transformation, and we record the the gripper transformation

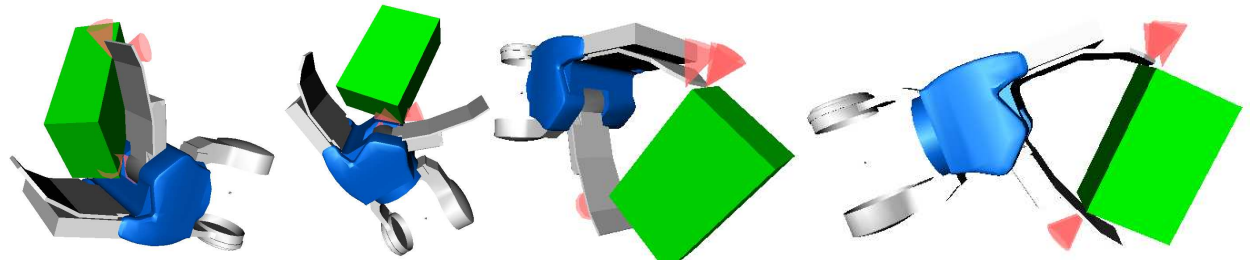


Figure 4.13: Fragile grasps that have force closure in simulation (contacts shown).

Algorithm 4.1: ComputeGraspRepeatability(σ_T)

/* $\rho \in [0,1]$ - uniform random variable */

$\mathcal{T}_G \leftarrow \emptyset\,, \quad \mathcal{Q} \leftarrow \emptyset$

for *iteration = 1 to N* **do**

  RotationContribution $\leftarrow \rho$

  $\theta \leftarrow$ RotationContribution * $\frac{\sigma_T}{\text{MaxDistance}(\textbf{target})}$

  $$T_{object} \leftarrow \begin{bmatrix} Rodrigues(\begin{bmatrix} \rho-0.5 \\ \rho-0.5 \\ \rho-0.5 \end{bmatrix}, \theta) & \text{RotationContribution} * \begin{bmatrix} 2\rho-1 \\ 2\rho-1 \\ 2\rho-1 \end{bmatrix} \\ 0 & 1 \end{bmatrix}$$

  SetTransform(**target**,T_{object})

  $\{T^{object}_{gripper}, q_{gripper}\} \leftarrow$ GraspStrategy()

  Append($\mathcal{T}_G$,ExtractTranslation($T^{object}_{gripper}$))

  Append($\mathcal{Q}$,$q_{gripper}$)

end

$\sigma_{joints} \leftarrow$ StandardDeviation($\mathcal{Q}$)

$\sigma_Q \leftarrow \max_{chain} \sum_{j \in chain} \sigma_{joints}(j)$ * MaxJointContribution(j)

$\sigma_G \leftarrow \sigma_Q +$ StandardDeviation($\mathcal{T}_G$)

if $\sigma_G > \gamma \sigma_T$ **then**

  return fragile

return stable

with its final joints into $\mathcal{T}_G$ and $\mathcal{Q}$. We compute the contribution from the gripper joints by multiplying the standard deviation of the angle with the maximum distance of any child link from the joint's axis. These errors are then accumulated across the kinematic hierarchy to yield σ_Q. We compute the final σ_G by adding σ_Q to the translational standard deviation of the gripper positions. This value is compared with σ_T to yield the final decision for a repeatable grasp. γ is set to 0.7 for all experiments and N is set to 40 when σ_T is set to 1cm.

When building grasp sets for the HRP2 hand, out of 18,000 tests 228 succeed the force closure test. When applying the ComputeGraspRepeatability filter, about 171 of grasps are left with **all** fragile grasps pruned out. 75% of the original grasps remain showing that the method does not reject that many good grasps in comparison to previous robustness measurements. The computation times increase proportional to N.

Table 4.3 shows the set of parameters necessary to for generating force-closure grasp sets.

Parameter Name	Parameter Description
Target Surface Sampling	how to sample the target surface for approach rays
Robot Surface Sampling	how to sample the robot surface for approach rays
Gripper Preshapes	a set of initial joint values for the gripper
Rolls about Direction	set of rolls of the robot around the approach axes
Standoffs	set of standoffs from the target object
Friction	friction between contact surfaces

Table 4.3: Force-closure Grasping Parameters

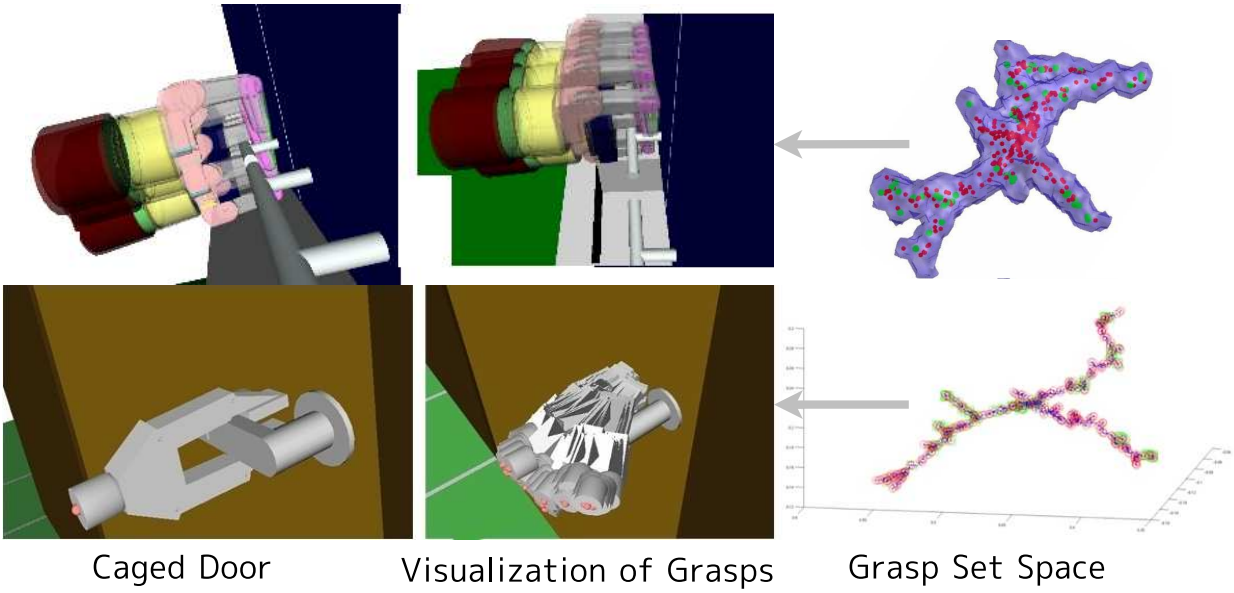


Figure 4.14: Example caging grasp set generated for the Manus Hand.

4.2.2 Caging Strategy

For objects rigidly constrained by the environment, we can relax the grasp definition by considering caging grasps. Section 3.4 showed that planning with caging grasps can greatly increase the feasible range of manipulator motions. We use RRTExplore (Algorithm 3.9) to generate a caging grasp set by exploring the space around an initial seed caging grasp g. In our experiments, we parameterize the grasp space by freezing the joints of the end-effector and searching with the end-effector pose in $SE(3)$ with three dimensions for translation and four dimensions for rotations represented as quaternions. Before RRTExplore is run, we set the target to a configuration where the handle is exposed well. Even though we only consider collisions between the object and the gripper, a door is composed of several moving

links, which can interfere with the caging sets. In practice, we found that the RRT explored the constrained space quickly and efficiently due to its neighborhood extension step; uniform randomized sampling would be very inefficient in this case. Because RRTs are resolution complete, they can return grasps that are very close to each other, therefore, we run a nearest neighbor filter using the union of all the valid caging grasps. This allows us to reduce the caging grasp set by 10 times while still maintain a good span of the caging space. Figure 4.14 shows two different caging grasp sets produced from this process.

Algorithm 4.2: TestCagingLinearPath($\mathcal{T}_{gripper}$,$\Delta\theta$)

```
1  success ← 0  for iteration = 1 to N do
2      SetTransform(gripper,T_gripper ΔT)
3      if not CheckCollision(gripper,target) then
4          for Δq ∈ SampleConfigurationDirections() do
5              caged ← false
6              for θ ∈ [−Δθ, Δθ] do
7                  SetConfiguration(target,q_target + θΔq)
8                  if CheckCollision(gripper,target) then
9                      caged ← true
10             end
11             if not caged then
12                 break
13         end
14         if not caged then
15             return false
16         success ← success + 1
17         if success > M then
18             return true
19     return false
20 end
```

TestCagingLinearPath (Algorithm 4.2) is a method to check if a grasp is caging the target configuration by searching for an escape route in a random direction Δq. First, we randomly jitter the grasp transformation with ΔT to allow us to reject fragile grasps as shown in 4.15. Before we can test a new jittered gripper, we have to check that the gripper and target are not initially colliding. If not in collision initially, we iterate on a cached set of configuration directions on the target. For 1 DOF doors, two directions are tested $\{-1, 1\}$,

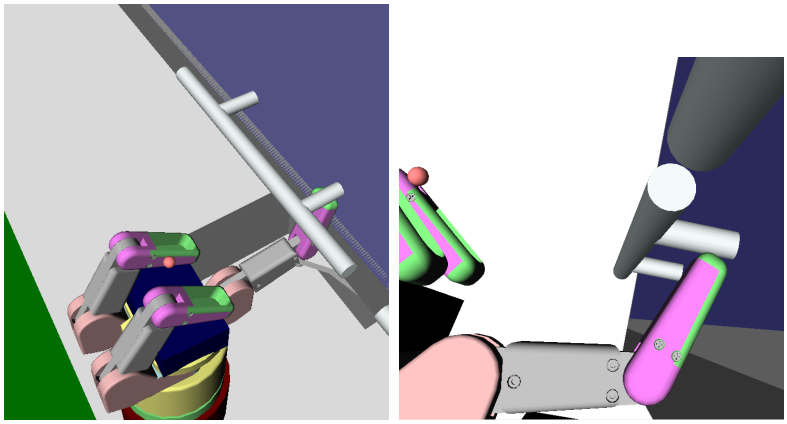


Figure 4.15: A fragile grasp that was rejected by the random perturbation in the grasp exploration stage, even though it mathematically cages the handle.

for higher configuration spaces, the directions have to be sampled carefully. If the door can move at least $\Delta\theta$ along a specific direction without getting into collision with the gripper, then there is no cage and the grasp is rejected. If M jittered configurations succeed the caging test, then we return success. We set M on the order of 20 and the number of trials N for collision-free jittered configurations on the order of 100.

The initial seed grasp g caging the target object can be obtained by randomly exploring the surface of the object until it passes the caging criteria. Efficiently searching the surface of an object requires the samples are uniformly distributed and can explore the space rapidly [LaValle (2006)].

By using caging grasps rather than grasps that fix the target object's configuration through contact force, we are able to provide the manipulator with significantly more flexibility in accomplishing the task while still guaranteeing the object cannot escape from the end effector's control. One additional detail discussed in Section 3.4 is the contact grasp set $G_{contact}$. At the end of the plan, we need to localize the target object being caged by the gripper. By always ending with a grasp that achieves the *form closure* condition, then we can guarantee that the object maintained at its desired final configuration,

4.2.3 Insertion Strategy

In industrial settings, it is very common for manipulators to use magnets or insert a pin into holes in the part rather than to use power grasps. In fact power grasping of target objects is mostly done in the research field where grippers are more human-like and rarely done in industrial manipulation. We show how the theory of the previous sections can be

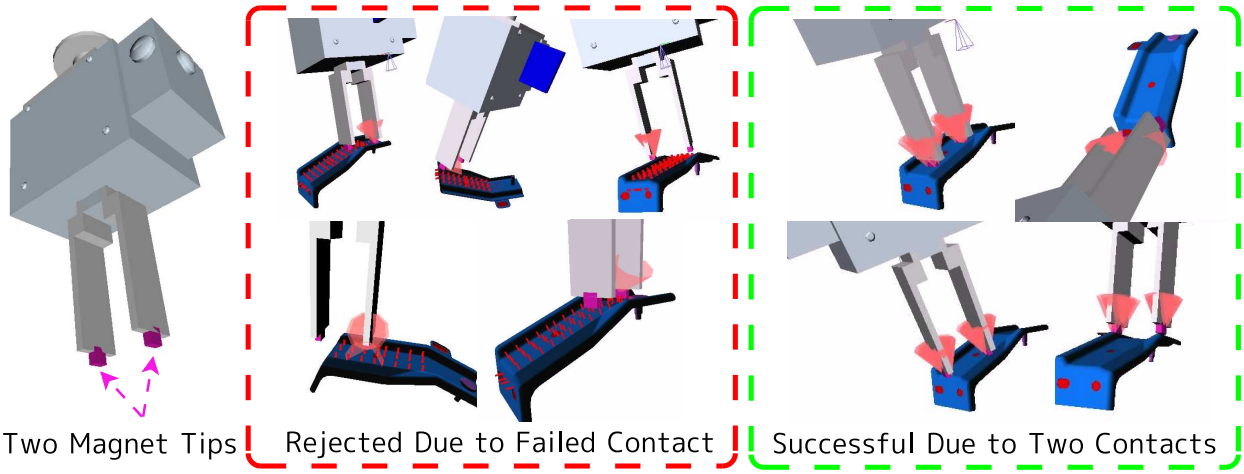


Figure 4.16: Goal is for both manipulator tips to contact the surface of the target object. The left box shows failure cases that are pruned, the right box shows the final grasp set.

applied to a very simple example of generating and using grasp sets for a gripper with two magnetic tips. Figure 4.16 shows a manipulator with two magnets attached at its fingers and the bracket that needs to be picked up. Since the magnets act as point contacts, grasps are much more stable when both tips contact the part surface in a flat region such that noise cannot upset the part's relative transformation.

In order to generate the grasp tables, we first sample the target bracket surface for possible approach directions similar to Figure 4.10. For every grasp tested, we gather the contacts and test for these two things instead of force closure:

- the contact points footprint is as big as the distance between the magnet tips,
- and all contact normals roughly point in the same direction.

We use ComputeGraspRepeatability to make sure that the magnet tips are contacting

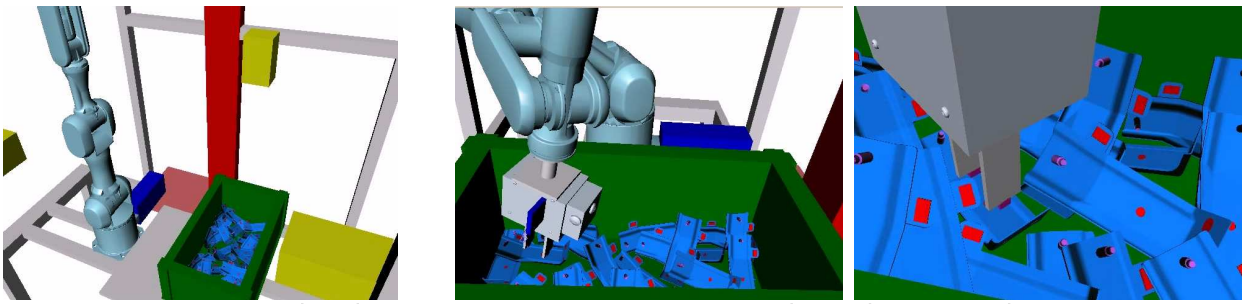


Figure 4.17: Idustrial robot using the magnet tip grasps to pick parts out of a bin.

a wide area and not a small obtrusion that is hard to hit when executing the grasp in the real world. Figure 4.16 shows the results of the grasp generation process. The red box to the left shows some of the bad grasps that are present in the tested sets. The green box in the right shows the final set of grasps each of the magnets is contacting a wide flat region. Using this grasp set, we can immediately apply the grasp planners to simulate a robot picking up parts from a bin and placing them at a destination (Figure 4.17).

4.3 Kinematic Reachability

Every manipulator has a reachability region that specifies where the gripper can move to. The reachability space can be used to quickly analyze the workspace for task feasibility without performing any planning as shown in [Zacharias et al (2007, 2009)]. Knowing the reachability region can help robots quickly prune bad grasps, or can help mobile robots move to the correct place with the highest reachability. Reachability also plays an important role for two-handed robots since have to figure out where to place the object so that both manipulators can handle it [Zacharias et al (2010)]. This section tackles the problem of intelligently biasing configuration space samples when the only goals given are the final grasps.

In this thesis the reachability space is used for:

- quickly pruning grasps when planning without base placement (Section 3.3),
- introducing biasing in configuration samplers to help explore regions where the arm is more maneuverable (Section 3.6.3),
- computing the inverse reachability for base placement sampling (Section 4.4),
- and quickly pruning the visible sensor locations (Section 5.1).

We represent the kinematic reachability as $KR : SE(3) \rightarrow \mathbb{R}$ that maps a 6D end-effector pose with respect to the arm's base into the density of valid solutions. There are two ways to compute such a map: uniformly sample 6D end-effector positions and record the number of inverse kinematics solutions, or randomly sample arm configurations and accumulate the end-effector poses. The latter method requires normalizing with respect to the density of the sampled arm configurations and keeping track of unique solutions. The normalization is non-trivial and can lead to biasing errors, therefore we prefer the former explicit 6D end-effector sampling method. There are two advantages with it: the entire workspace is guaranteed to be explored, and the analytical inverse kinematics solvers of Section 4.1 allow exact computation of the size of the solution space.

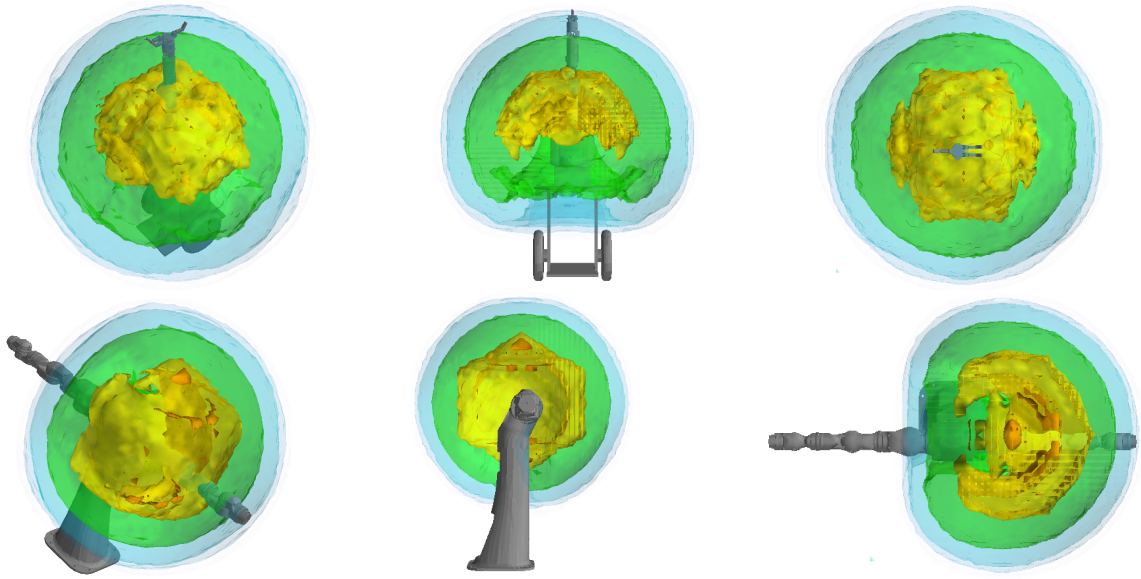


Figure 4.18: Barrett WAM and Yaskawa SDA-10 kinematic reachability spaces projected into 3D by marginalization of the rotations at each point.

We first need to sample a set of positions in sphere in $\mathbb{R}^3$ around the arm. Because computation is proportional to the volume of the sphere, we use the following analysis to find the smallest sphere encompassing the end-effector's possible reach. Every joint has an anchor a_i and a joint axis d_i. We compute the closest point p_i on $\{a_i, d_i\}$ to the next consecutive joint on the chain $\{a_{i+1}, d_{i+1}\}$ using:

$$t_i = \frac{((a_i - a_{i+1}) \times d_{i+1}) \cdot (d_i \times d_{i+1})}{|d_i \times d_{i+1}|^2}$$

$$p_i = a_i + t_i d_i. \tag{4.51}$$

We set the last point p_n as the tip of the end effector used for inverse kinematics. Finally, we set the sphere center to p_0 and the radius to $\sum_i |p_{i+1} - p_i|$. For prismatic joints, we add the greatest joint limit into the radius.

GENERATE_KINEMATICREACHABILITY() (Algorithm 4.3) takes the desired resolution of the translation and rotations samples and produces a list of poses in $SE(3)$ with their solution density. Uniformly sampling $SE(3)$ with low discrepancy and dispersion is very difficult; furthermore, it is not desirable to introduce any random components with database building since random sampling on average has much worse dispersion than informed deterministic sampling [LaValle (2006)]. Fortunately, there exist good deterministic samplers for $\mathbb{R}^3$ and $SO(3)$, which are covered in Section 4.3.1. The cross product of the independent translation

Algorithm 4.3: GENERATE_KINEMATICREACHABILITY(**arm**,Δp_R, $\Delta\theta_R$)

$\{p_i\} \leftarrow$ COMPUTEJOINTINTERSECTIONS()
Radius $\leftarrow$ PrismaticLimits $+ |\Delta p_R| + \sum_i |p_{i+1} - p_i|$
$\{t_i\} \leftarrow \{t \mid t \in$ SAMPLES$(\mathbb{R}^3, \Delta p_R), |t - p_0| \leq$ Radius$\}$
$\{r_i\} \leftarrow$ SAMPLES$(SO(3),\Delta\theta_R)$
Reachability $\leftarrow \emptyset$
for $\{r, t\} \leftarrow \{r_i\} \times \{t_i\}$ **do**
    $valid \leftarrow 0$
    for $q_{arm} \in IK(\{r, t\})$ **do**
        SetConfiguration(**arm**,q_{arm})
        if not ISLINKCOLLISION(GETATTACHEDLINKS(**arm**)) **then**
            $valid \leftarrow valid + 1$
    end
    APPEND(Reachability,$\{r, t, valid\}$)
end

and rotation samples yield a final sampling of $SE(3)$. For every end-effector pose $\{r, t\}$, we count the number of inverse kinematics solutions that are not colliding with the attached links of the arm and create the final list. Figure 4.18 shows the projected reachability volume generated for two different robots evaluated with $\Delta\theta_R = 0.25$ and $\Delta p_R = \begin{bmatrix}0.040.040.04\end{bmatrix}$. The number of $SO(3)$ samples is 576 and the number of $\mathbb{R}^3$ samples is $\frac{4\pi Radius^3}{3\Delta p_{R,x}\Delta p_{R,y}\Delta p_{R,z}}$.

Each of the operations using the reachability space requires efficient nearest neighbor retrieval given an end effector pose. We store each rotation $r \in SO(3)$ as a quaternion. If two quaternions are close to each other and on the same hemisphere, then we can define a simple distance metric between two poses:

$$\delta(\{r_1, t_1\}, \{r_2, t_2\}) = \sqrt{const \min(|r_1 - r_2|^2, |r_1 + r_2|^2) + |t_1 - t_2|^2} \tag{4.52}$$

where the euclidean distance for close quaternions is close to the Haar measure on $SO(3)$. Because most kd-tree implementations work in Euclidean space, we add both $\{r, t\}$ and $\{-r, t\}$ to simultaneously represent the two hemispheres. The nearest neighbors implementation require a little book-keeping in order not to count a rotation twice. The final implementation of KR (Algorithm 4.4) uses kd-trees and is reminiscent to kernel density evaluation. Intersection between a set of poses and the reachability is performed by evaluating KR for each of the poses and thresholding the density of the valid solutions. Because the nearest neighbor query is a tree search, intersection with a set of M poses is on the order of $O(M \log |Reachability|)$ time.

Algorithm 4.4: KR(r,t)

$valid \leftarrow 0$
$\omega \leftarrow 0$
$\Delta_R \leftarrow \sqrt{c\Delta\theta_R^2 + |\Delta p_R|^2}$
for $\{r_i, t_i, valid_i\} \in$ NEARESTNEIGHBORSRADIUS*(Reachability,* $\{r,t\}$*,* $2\,\Delta_R$*)* **do**
    $\omega_i \leftarrow exp(-\frac{\delta(\{r_i,t_i\},\{r,t\})^2}{0.5\Delta_R^2})$
    $valid \leftarrow valid + \omega_i\, valid_i$
    $\omega \leftarrow \omega + \omega_i$
end
return $\frac{valid}{\omega}$

Parameter Name	Parameter Description
$\Delta\theta_R$	discretization for rotations in terms of the Haar measure
Δp_R	discretization for translations

Table 4.4: Kinematic Reachability Parameters

Figure 4.19 shows the comparison of the reachability spaces of the left arm for a humanoid robot. If the arm includes one of the chest joints in its definition, then the reachability spaces will be much spatially bigger and about 1.5 times more dense. The wrist has a differential drive mechanism that produces a really unique joint limits range of the two motors. By considering the full range of the joint limits, the reachability density in $\mathbb{R}^3$ increases by two times. Given the small reachability volume of HRP2 compared to the robots in Figure 4.18, it is vital to use the entire robot free space.

Table 4.4 describes all the parameters necessary for generating the reachability.

4.3.1 Uniform Discrete Sampling

Because computation of KR directly depends on the number of samples of $SO(3)$ and $\mathbb{R}^3$, ideally we want the minimal number of samples that still span the entire pose space of the arm and have the lowest dispersion and discrepancy.

To sample $SO(3)$, we use recent results from [Yershova et al (2009)] where they use the

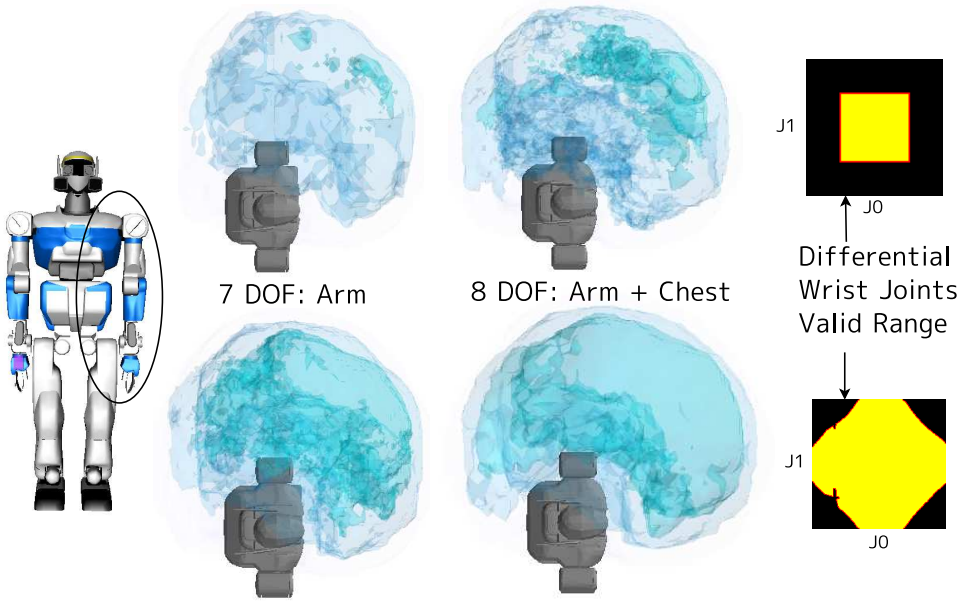


Figure 4.19: The HRP2 reachability space changes when the chest joint is added (left vs right). Also, the HRP2 wrist has a unique joint limits range, which can increase the reachability density by 2x if handled correctly.

fact that $SO(3) \cong S^1 \tilde{\otimes} S^2$ and represent each rotation with Hopf coordinates $\{\theta, \phi, \psi\}$ where

$$\{\theta, \phi\} \in S^2 \,, \qquad \psi \in S^1$$

$$quaternion = \begin{bmatrix} \cos\frac{\theta}{2}\cos\frac{\psi}{2} \\ \cos\frac{\theta}{2}\sin\frac{\psi}{2} \\ \sin\frac{\theta}{2}\cos\phi + \frac{\psi}{2} \\ \sin\frac{\theta}{2}\sin\phi + \frac{\psi}{2} \end{bmatrix} .$$

The problem reduces to uniformly sampling S^1 and S^2 and then taking their cross product to form full set of Hopf rotations. The HEALPix algorithm [Gorski et al (2005)] is used for sampling a multi-resolution grid of the 2-sphere S^2. In order to maintain low discrepancy, the method in [Yershova et al (2009)] works by subdividing each dimension by 2 from the previous level. This yields a total of $m_1 m_2 8^L$ samples where m_1 is the initial subdivisions of S^1 set to 6, and m_2 is the initial subdivisions of S^2 set to 12.

To sample $\mathbb{R}^3$, we use a lattice where the three axes are prime with respect to the field of fractions on the integers $\mathbb{Z}$ [Matousek (1999)]. Assuming we need to generate N samples

inside a cube whose sides are in $[0, 1]$, the i^{th} sample is generated by:

$$\left(\frac{i}{N}, \{0.5 + i\frac{\sqrt{5}}{2}\}, \{0.5 + i\sqrt{13}\}\right) \tag{4.53}$$

where $\{x\}$ denotes the fractional part of x. Using this sampler, the number of samples required for the average distance between neighbors to be 0.04 is N=4733.

4.4 Inverse Reachability

Computing placements of the robot base depending on the task and robot kinematics is important for analyzing the scene [Stulp et al (2009)]. We define inverse reachability $IR_{arm} : SE(3) \rightarrow (SE(2) \rightarrow \mathbb{R})$ as a function that takes in an end effector pose and returns a distribution on the 2D plane of where the base can be in order to achieve that particular end-effector pose. Because we are keeping the end-effector as a parameter, it is necessary to work with the inverse transformations of all the poses stored in the kinematic reachability map. In order to convert this map to 2D base movements, we *project* the inverted 6D map onto the 2D plane and start counting solution density. Because inverse reachability stores a purely geometric relationship, the return values of IR are the density of solutions rather than probability of existence of a solution. Some types of robots like humanoids can have joints connecting the arm base and the robot base like the torso joints. The maps generated in the following analysis are only valid for the particular torso joints it was trained with; therefore application of the map requires the robot be moved to those joints ahead of time, or multiple maps be trained for every unique value of the torso joints.

In order to efficiently construct and query the inverse reachability map, we first compute a set of equivalence classes invariant to 2D translations and in-plane rotations. Without loss of generality we assume that the 2D plane that the base moves on is on XY going through the origin. We let $\mathcal{X}$ be a discrete set of all kinematically reachable end-effector poses with respect to the robot base. The equivalence classes are built using the poses in $\mathcal{X}$. From a mathematics point of view we are interested in the inverse map, but from a computational point of view we first cluster the equivalence classes in the original map before inverting. We define a rotation on the XY plane as $R_z(\theta)$ and the group of all rotations as

$$\mathcal{R}_z = \left\{R_z(\psi) \mid \psi \in \mathbb{S}^1\right\} \tag{4.54}$$

We can now define an equivalence class of all rotations similar to $r \in SO(3)$ that differ only by a rotation about the z axis as:

$$r\,\mathcal{R}_z = \{r_z * r \mid r_z \in \mathcal{R}_z\}. \tag{4.55}$$

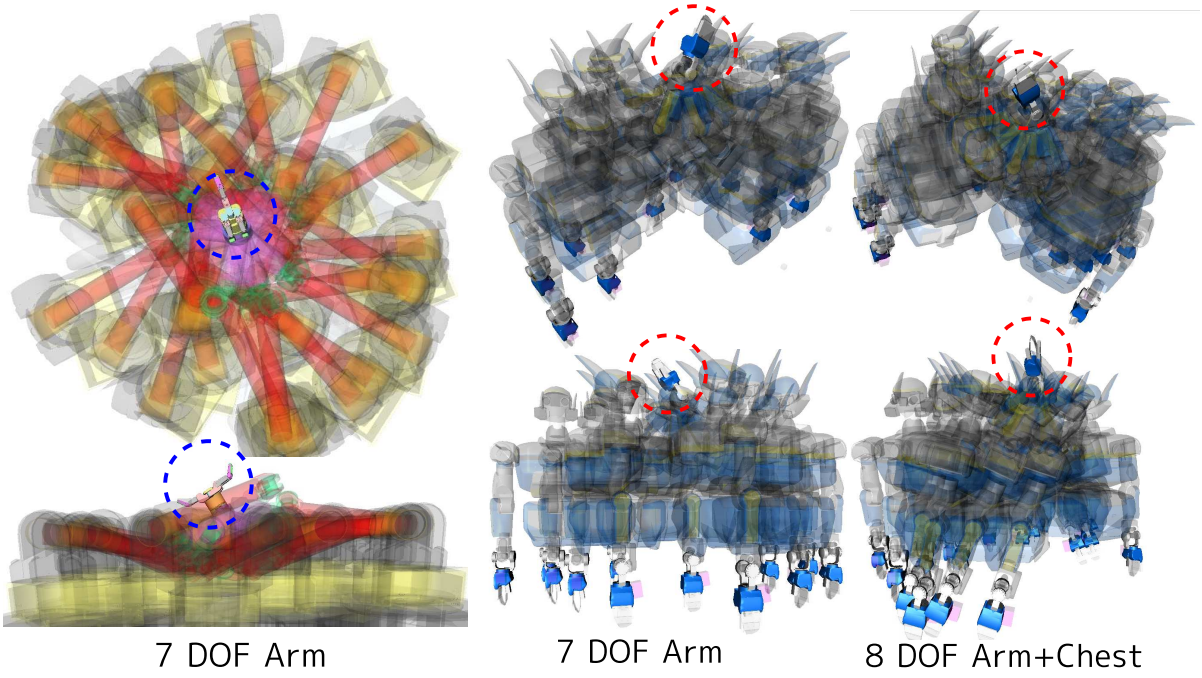


Figure 4.20: Shows one of the extracted equivalence sets for the Barrett WAM (869 sets), HRP2 7 DOF arm (547 sets), and HRP2 8 DOF arm+chest (576 sets). Each was generated with $\Delta\theta_I = 0.15$, $\Delta z_I = 0.05$.

where $*$ denotes the action of applying one rotation after another. Similarly the set of plane transformations can be defined by

$$\mathcal{P}_z = \left\{ \{R_z(\psi), (x, y, 0)\} \mid \psi \in \mathbb{S}^1, x \in \mathbb{R}, y \in \mathbb{R} \right\} \tag{4.56}$$

For clarity, we denote $p(r, t) \in SE(3)$ is the pose with rotation $r \in SO(3)$ and translation $t \in \mathbb{R}^3$. It is homeomorphic to the transformation notation T used so far, so it is used interchangeably. Given an arbitrary pose $p \in SE(3)$, its equivalence class with respect to 2D plane movements becomes:

$$p\,\mathcal{P}_z = \{p_z * p \mid p_z \in \mathcal{P}_z\}\,. \tag{4.57}$$

We compute the equivalence classes for the entire set $\mathcal{X}$ by sampling a pose $p_i \in \mathcal{X}$, and extracting all similar poses to form a discrete set $\mathcal{X}_i$:

$$\mathcal{X}_i = \{p \mid p \in \mathcal{X}, p \in p_i\,\mathcal{P}_z\}\,. \tag{4.58}$$

The entire process is repeated with the remaining poses $\mathcal{X} - \mathcal{X}_i$ until we have a set of equivalence sets $\{\mathcal{X}_i\}$. Each extracted equivalence set $\mathcal{X}_i$ is essentially indexed by an out-of-plane rotation r_μ modulo rotations on z and a distance from the 2D plane z_μ and, which is 3 degrees of freedom in total. For each $\mathcal{X}_i$, we extract $\{r_\mu, z_\mu\}$ and convert each of the poses in $SE(3)$ into poses in $SE(2)$. The 2D poses are then inverted and stored into the inverse reachability map indexed by $\{r_\mu, z_\mu\}$.

Every equivalence class $p\,\mathcal{P}_z$ has a representative element that has no translation along XY and rotation along Z. ProjectPose(p) (Algorithm 4.5 shows how a pose gets projected into the representative element of its class.

Algorithm 4.5: ProjectPose(p)

$\{r, t\} \leftarrow$ ExtractQuaternionTranslation(p)

$\psi \leftarrow$ **atan2**$(-r_z, r_w)$

$$r' \leftarrow \begin{bmatrix} q_w \cos\psi - q_z \sin\psi \\ q_x \cos\psi - q_y \sin\psi \\ q_y \cos\psi + q_x \sin\psi \\ q_z \cos\psi + q_w \sin\psi \end{bmatrix}$$

$t' \leftarrow \begin{bmatrix} 0 & 0 & t_z \end{bmatrix}$

return $\{r', t', \psi, t_x, t_y\}$

Generate_InverseReachability (Algorithm 4.6) shows how to generate the list of equivalence classes taking into account practical issues like thresholds for set membership and computation complexity. The algorithm first starts by re-ordering $\mathcal{X}$ so that the poses with the most neighbors are considered first, which allows extraction of more meaningful equivalence sets. Neighbor density is computed modulo 2D plane movements where the rotation component is weighted by $\frac{1}{\Delta\theta_I}$ and the translation component by $\frac{1}{\Delta z_I}$. The reason for weighting is because nearest neighbors treats the pose as a 7 DOF quaternion and translation value. Usually mixing up translation and rotation is very difficult, but we roughly estimate their importance using the thresholds set by the user. The densest pose $p_i \in \mathcal{X}$ is sampled and its nearest neighbors modulo 2D plane movements are computed such that the out-of-plane rotation and height of the neighbor are within the $\Delta\theta_I$ and Δz_I from p_i. The information on the neighbors are stored into three sets:

- $\mathcal{X}_i$ - the original poses most similar to p_i,
- $\mathcal{X}_{projected,i}$ - the poses with 2D translations and in-plane rotation removed,
- $\mathcal{X}_{2D}$ - the extracted 2D translations and in-plane rotations.

Since $\mathcal{X}_{projected,i}$ are not necessarily all the same, we compute the mean rotation and height that minimizes the sum of squared distances; for rotations, we use nonlinear optimization with the Haar measure. The mean projected poses of each equivalence set are used for nearest neighbor computation in the sampling phase of the inverse reachability map.

Algorithm 4.6: GENERATE_INVERSEREACHABILITY(**arm**,$\Delta\theta_I$,Δz_I)

$\mathcal{X} \leftarrow \left\{T_{arm}^{base} p(r,t) \mid \{r,t,valid\} \in Reachability, valid > 0\right\}$
/* Order the elements in $\mathcal{X}$ based on density */
$\mathcal{X}_{projected} \leftarrow \{\text{PROJECTPOSE}(p) \mid p \in \mathcal{X}\}$
$searchthresh \leftarrow \sqrt{(1-\cos\Delta\theta_I)^2 + (\sin\Delta\theta_I)^2}$
Density
$\leftarrow \{\|\text{NEARESTNEIGHBORSRADIUS}(\mathcal{X}_{projected}, p, searchthresh)\| \mid p \in \mathcal{X}_{projected}\}$
$\mathcal{X} \leftarrow \text{ORDER}(\mathcal{X}$,Density)
/* Build equivalence sets */
$\mathcal{X}_{all} \leftarrow \emptyset$
for $p_i \leftarrow \text{HIGHESTDENSITYELEMENT}(\mathcal{X})$ **do**
    $\{r'_i, p'_i\} \leftarrow \text{PROJECTPOSE}(p_i)$
    $\mathcal{X}_i \leftarrow \emptyset,\ \mathcal{X}_{projected,i} \leftarrow \emptyset,\ \mathcal{X}_{2D} \leftarrow \emptyset$
    for $p \in \mathcal{X}$ **do**
        $\{r', t', \psi, t_x, t_y\} \leftarrow \text{PROJECTPOSE}(p)$
        if $\delta(r'_i, r') \leq \Delta\theta_I$ **and** $|t'_i - t'| \leq \Delta z_I$ **then**
            APPEND($\mathcal{X}_i$,p)
            APPEND($\mathcal{X}_{projected,i}$,$\{r', t'\}$)
            /* Store the inverted 2D pose */
            APPEND($\mathcal{X}_{2D}$,$\{-\psi, -t_x\cos\psi - t_y\sin\psi, t_x\sin\psi - t_y\cos\psi\}$)
    end
    $\{r_\mu, z_\mu\} \leftarrow \text{MEAN}(\mathcal{X}_{projected,i})$
    $\{r_\sigma, z_\sigma\} \leftarrow \text{STANDARDDEVIATION}(\mathcal{X}_{projected,i})$
    $\mathcal{X} \leftarrow \mathcal{X} - \mathcal{X}_i$
    APPEND($\mathcal{X}_{all}$,$\{r_\mu, z_\mu, r_\sigma, z_\sigma, \mathcal{X}_{2D}\}$)
end

Figure 4.20 shows some of the 2D poses in single equivalence sets. The HRP2 8 DOF reachability space has 200,000 poses and 576 equivalence sets were extracted from it using $\Delta\theta_I = 0.15$ and $\Delta z_I = 0.05$. As a comparison, the 7 DOF arm has 547 sets, which shows that regardless of how complex the arm chain gets, the complexity of the inverse reachability

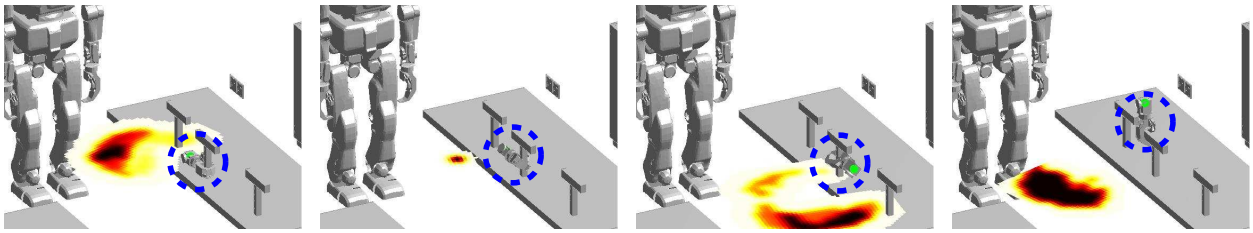


Figure 4.21: Base placements distributions for achieving specific gripper locations; darker colors indicate more in-plane rotations.

map is mostly governed by the thresholds for set membership. Figure 4.21 shows the base placement distributions for several equivalence classes where the gripper of the robot is grasping the target object.

The inverse reachability maps can be efficiently sampled by first indexing into the equivalence sets and then sampling the underlying probability distribution represented by a Mixture of Gaussians. INDEX_INVERSEREACHABILITY (Algorithm 4.7) uses the standard deviation of the elements in each equivalence set to return the set with the highest probability of including the pose. Because PROJECTPOSE removes all 2D base contributions, it is possible to compute the distance to the equivalence sets with a weighted Euclidean norm. The computation is further sped up by organizing $\mathcal{X}_{all}$ into a kd-tree.

Algorithm 4.7: INDEX_INVERSEREACHABILITY(p)

1 $\{r, t, \psi, t_x, t_y\} \leftarrow$ PROJECTPOSE(p)
2 $D_{best} \leftarrow \infty\,, \quad \mathcal{X}_{2D,best} \leftarrow \emptyset$
3 **for** $\{r_\mu, z_\mu, r_\sigma, z_\sigma, \mathcal{X}_{2D}\} \in \mathcal{X}_{all}$ **do**
4 $\quad D \leftarrow \left(\frac{\delta(r,r_\mu)}{r_\sigma \Delta\theta_I}\right)^2 + \left(\frac{\delta(t,z_\mu)}{z_\sigma \Delta z_I}\right)^2$
5 $\quad$ **if** $D < D_{best}$ **then**
6 $\quad\quad D_{best} \leftarrow D\,, \quad \mathcal{X}_{2D,best} \leftarrow \mathcal{X}_{2D}$
7 **end**
8 **return** $\{D_{best},\ X_{2D,best}, \psi, t_x, t_y\}$

SAMPLE_INVERSEREACHABILITY (Algorithm 4.8) extracts the closest equivalence set and then uses kernel density estimation on the set of 2D poses $\mathcal{X}_{2D}$ to sample from a continuous distribution. The world pose of the gripper is first converted into the robot base's coordinate system, which takes into account the torso joints that joint the robot base to the arm base. Once the closest equivalence class is returned, a discretization threshold D_τ is used to determine if an equivalence set is too far away for inverse kinematics to come up

Algorithm 4.8: SAMPLE_INVERSEREACHABILITY($T^{world}_{gripper}$)

$\left\{D, \mathcal{X}_{2D}, \psi^{gripper}, t_x^{gripper}, t_y^{gripper}\right\} \leftarrow$ INDEX_INVERSEREACHABILITY($T^{base}_{world} T^{world}_{gripper}$)

if $D > D_\tau$ **then**

    /* No base placements for this gripper pose. */

    return $\emptyset$

$\{\psi', t'_x, t'_y\} \leftarrow$ SAMPLEMIXTUREGAUSSIANS($\mathcal{X}_{2D}$, $\sigma = \left(\Delta\theta_R \frac{\Delta z_I}{\Delta\theta_I},\ \Delta p_{R,x},\ \Delta p_{R,y}\right)$)

$\{\psi^{world}, t_x^{world}, t_y^{world}\} \leftarrow$ PROJECTPOSE(T^{world}_{base})

$q_{base} \leftarrow \{\psi^{world}, t_x^{world}, t_y^{world}\} * \{\psi^{gripper}, t_x^{gripper}, t_y^{gripper}\} * \{\psi', t'_x, t'_y\}$

return q_{base}

Parameter Name	Parameter Description
$\Delta\theta_I$	discretization for rotations in terms of the Haar measure
Δz_I	discretization of the distance normal to plane of movement.
Bandwidth σ	the bandwidth of each gaussian kernel, depends on discretization
Joint Values	preset values for joints connecting the robot base to the manipulator base

Table 4.5: Inverse Reachability Parameters

with a valid solution. We use gaussian kernels with a bandwidths that take into account the discretization of the reachability generation process: in-plane rotation bandwidth is $\Delta\theta_R \frac{\Delta z_I}{\Delta\theta_I}$ and XY translation bandwidth is $(\Delta p_{R,x},\ \Delta p_{R,y})$. Intuitively the factors should be proportional to the underlying reachability discretization; furthermore, the rotation and translation have to be normalized with respect to each other with the $\frac{\Delta z_I}{\Delta\theta_I}$ factor. The distribution of poses in $\mathcal{X}_{2D}$ assumes that the gripper is at the origin of the coordinate system. In order to transform this into the world, we transform every sampled pose in $\mathcal{X}_{2D}$ by the 2D translation and in-plane rotation extracted by the gripper. Because we were working in the robot base's coordinate system, we have to transform back to the world coordinate system to yield the final q_{base} configuration. $*$ is the transformation operator on 2D poses. Note that we are not transforming by the full pose T^{world}_{base} since we've already indexed an equivalence set that takes its height and out-of-plane rotations into account.

In the analyses above, each pose in the reachability set $\mathcal{X}$ is treated with equal weight, which makes the explanations easier. But in practice, the inverse kinematics solver and self-collision checking allows certain end-effector poses to have many more configurations than others. When building the inverse reachability maps, we keep track of the count of inverse kinematics solutions of each original pose in $\mathcal{X}$. This information propagates down into the

2D pose sets for each equivalence set $\mathcal{X}_{2D}$ and in the end we end up with a weighted mixture of gaussian distribution. When considering the full computational cost of sampling inverse reachability, empirical results show that we can generate feasible base placement results 2.5 times faster than uniform random sampling.

Table 4.5 shows the parameters necessary to generate the inverse reachability maps.

4.5 Grasp Reachability

By combining grasp sets and inverse reachability, it becomes possible to compute the relation between the target object and the base placement of the robot at its current configuration. We call this relationship *grasp reachability* and use it in the form of a distribution on q_{base} to sample base placements as covered in Section 3.6.1. Figure 4.22 shows how the distribution behaves when grasps validated against the current environment are used to seed the inverse reachability maps. The BarrettHand/WAM combo creates really large grasp reachability regions because of two major contributing factors: the Barrett Hand can grasp the cup from almost any direction, and the WAM reachability space is very big. The middle column shows the grasp reachability space when the 7DOF arm is computation, and the right column shows the a scene with four objects while using including the chest as part of the arm definition.

Algorithm 4.9: SAMPLE_GRASPREACHABILITY(**arm**, $\mathcal{G}$)

```
1  weights ← ∅
2  G ← ∅
3  for grasp ∈ 𝒢 do
4      GripperTransforms ← GRASPVALIDATOR(grasp)
5      if GripperTransforms ≠ ∅ then
6          T^world_gripper ← LASTELEMENT(GripperTransforms)
7          𝒳_2D ← INDEX_INVERSEREACHABILITY(T^base_world T^world_gripper)
8          APPEND(weights,|𝒳_2D|)
9          APPEND(G,{grasp, GripperTransforms})
10 end
11 {grasp, GripperTransforms} ← WEIGHTEDSAMPLING(𝒯,weights)
12 q_base ← SAMPLE_INVERSEREACHABILITY(LASTELEMENT(GripperTransforms))
13 return {q_base, grasp.q_gripper, GripperTransforms}
```

The grasp reachability sampler validates the set of possible grasps and uses them to aggregate the inverse reachability maps into a mixture of gaussians. SAMPLE_GRASPREACHABILITY

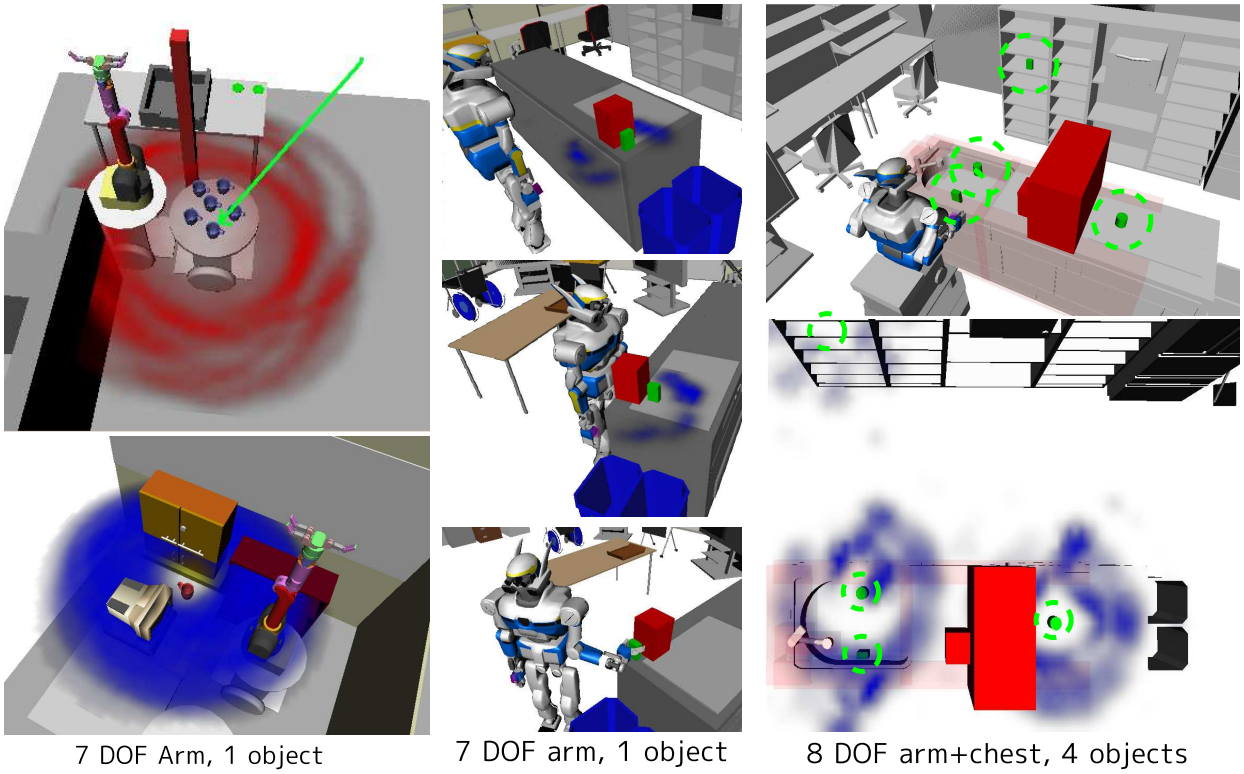


Figure 4.22: The grasp reachability map using validated grasps to compute the base distribution (overlayed as blue and red densities). The transparency is proportional to the number of rotations.

(Algorithm 4.9) shows a naive implementation of the processes necessary to sample a configuration. In the beginning, all the valid grasps are computed and the weight is computed from the number of poses contained inside its associated 2D pose map. Because the inverse reachability maps are mixture of gaussians, the weights are additive, so we can first gather all the equivalence sets of all the different grasps and evaluate a weight for each grasp depending on the number of basis vectors in its distribution. We can sample from the entire distribution by first choosing a grasp based on its weight. Once a grasp is sampled, we just call into the inverse reachability map sampler using Algorithm 4.8.

In practice, we progressively build up the distribution and cache previous computations as the sampler is called again, which allows a new sample to be computed on the order of milliseconds while taking less than **0.2s** to initialize the distribution. Besides sampling, there are many other practical uses for grasp reachability like performing workspace analysis. Whenever a task has to be performed by a robot in a factory, engineers have to carefully consider the robot kinematics, robot placement, part trajectories, and surrounding obstacles and safety measures. Engineers have to pick the cheapest and most robust solution. By

Parameter Name	Parameter Description
Validity Threshold	threshold of the distribution to accept grasps

Table 4.6: Grasp Reachability Parameters

explicitly modeling the workspace of the robot in the context of manipulating parts, it becomes possible to choose the cheapest robot that can achieve the task and how to minimize the workspace. Once a workspace heuristic based on the current task has been developed, it is possible to turn the workspace design problem into a robot kinematics design problem [III and Shimada (2009)].

Table 4.6 shows the parameters necessary to sample from grasp reachability.

4.6 Convex Decompositions

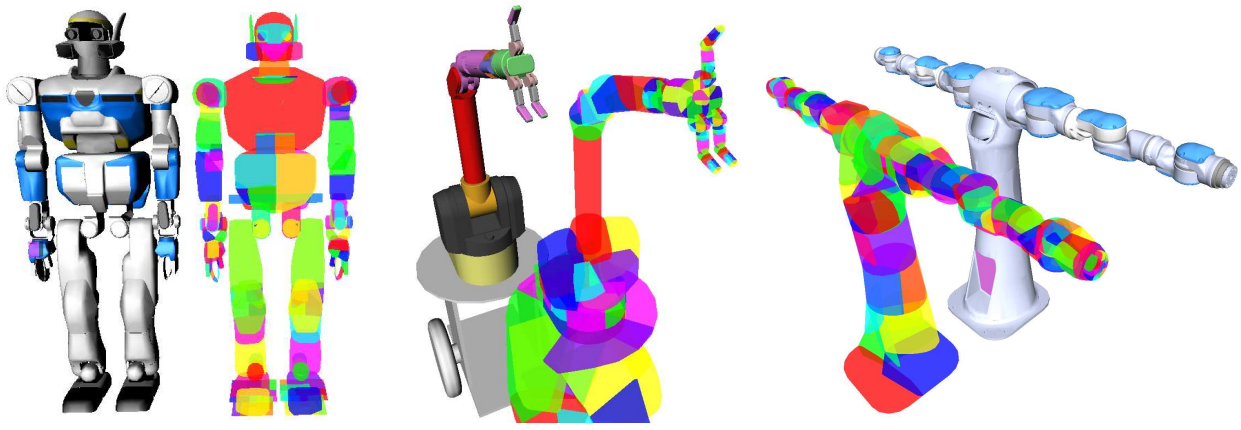


Figure 4.23: Examples of convex decompositions for the geometry of the robots.

We explain the usefulness of convex decompositions in manipulation planning algorithms and present three different applications to manipulation planning that become possible using the decomposition. It is common for researchers to approximating a mesh using a convex hull for its mathematical simplicity, but convex hulls can lead to very gross approximations to the geometry rendering them useful only for pre-testing a conditions before doing a re-test with the real geometry. It is also common to approximate a mesh as a decomposition of primitive shapes like boxes and spheres. However, the most general primitive that lends itself to many mathematical theories is the convex hull, and decomposing a mesh into a set

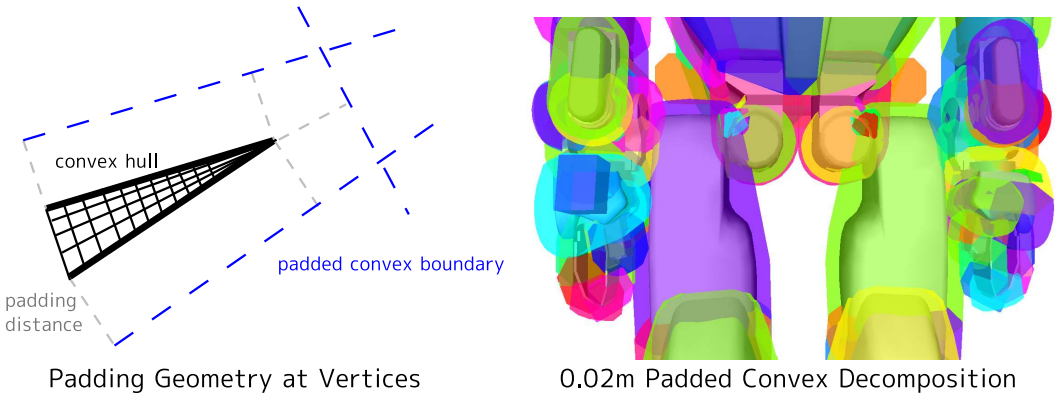


Figure 4.24: Examples of convex decompositions for the geometry of the robots.

of convex hulls can lead to many reductions in the complexity of the math. Figure 4.23 shows convex decompositions for three robots using the library from [Ratcliff (2006)].

4.6.1 Padding and Collisions

Collision checking with convex hulls is no more complex than collision checking with triangle meshes; both use the efficient Separating Axis Theorem [Eberly (2001)]. Although triangles can be considered a very simple type of convex hull, the convex decomposition can place restrictions on the eccentricity of the convex shape, making collision tests more stable by disallowing sharp edges. Furthermore, each of the robots shown in Figure 4.23 has on the order of one hundred hulls making collision detection very fast. In its simplest form, a convex hull is defined by a set of planes Π where A 3D point x is inside the convex hull if

$$\forall_{P \in \Pi} \; P \begin{bmatrix} x \\ 1 \end{bmatrix} \geq 0. \tag{4.59}$$

One operation commonly performed for planning is to pad the collision meshes with 5mm-10mm such the robot can at least keep a safety margin around environment obstacles. Adding padding to guarantee a distance δ introduces cylinders for every edge and spheres for every vertex in the collision mesh, which can be slow to handle. In order to compute a padded convex hull, we add a plane for every edge and a plane for every vertex. The normals of the new planes are the average of the normals adjacent planes with each plane going through their respective edge and vertex. Then all the planes are pushed δ back along their normals:

$$\forall_{P \in \Pi \bigcup \Pi_{edge} \bigcup \Pi_{vertex}} \; P \begin{bmatrix} x \\ 1 \end{bmatrix} \geq -\delta. \tag{4.60}$$

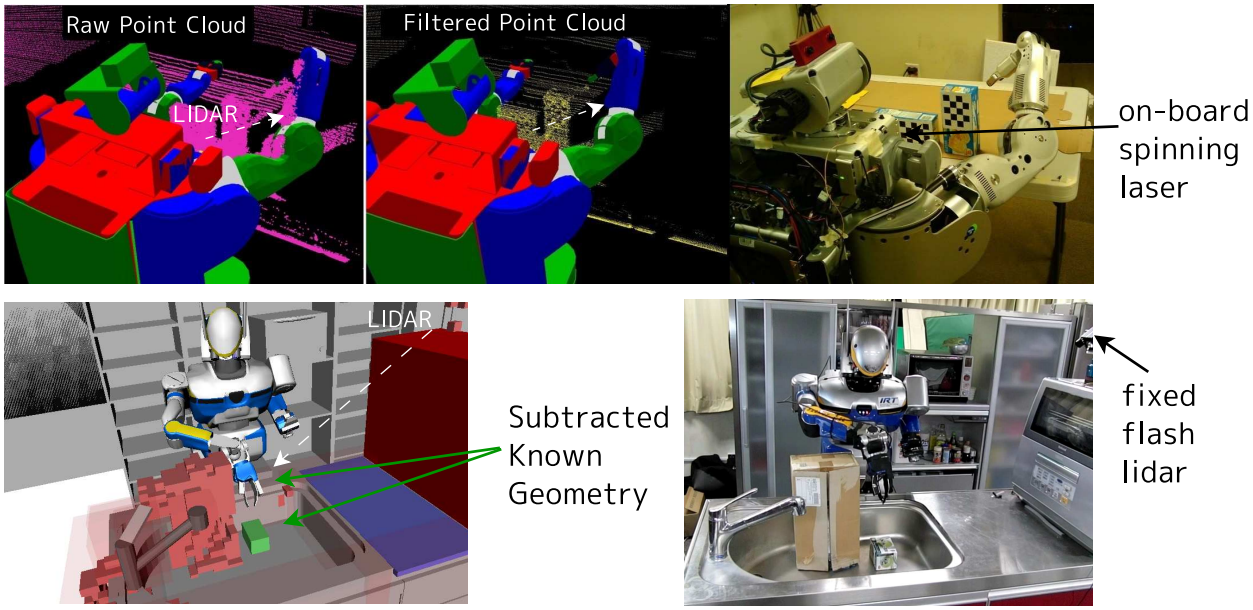


Figure 4.25: Convex decomposition makes it possible to accurately prune laser points from the known geometry.

Figure 4.24 shows the final padded convex hull. It should be noted that convex decomposition meshes **cannot** be used for self-collision tests. Self-collisions are very special should be checked with the original meshes, otherwise the robot will initially start in self-collision and will never be able to move. Although hard self-collision checking with convex hulls is not recommended, it is possible to generate a continuous gradient proximity distance [Escande et al (2007)]. Such proximity distances are commonly used to apply safety torques to the robot joints inside the controller real-time loop. The proximity distance can also be used to prioritize configurations when sampling the configuration space.

4.6.2 Advantages of Volume Representations

When using stereo vision or laser range data to compute unknown dynamic obstacles in the environment, it becomes necessary to remove points that lie on already known geometry in the environment. For example, Figure 4.25 shows two scenes where the laser range finder also captures part of the robot arm. These points can be accurately pruned by using padded convex hull, an operation which is very difficult to perform with arbitrary closed triangle meshes approximating the surface of the robot. Along with subtracting the robot's points, the bottom scene also shows subtraction of an already detected target object. Every point that passes these tests is added as a small box with the size depending on the error model

of the sensors. From experiments with planning in environments with real-time range data, it is not feasible to check all the range data with all the obstacles in the environment; even with parallelization, it will kill the update rate. Therefore, we only prune range points that intersect the robots and target objects we plan with.

4.6.3 Configuration Distance Metrics

All randomized planners first define a configuration space in which a path is searched in. The distance metric in this space greatly affects planner performance because it dictates the step size of the planner. As was discussed in Chapter 3, a very important operation in path planning is checking if a line segment defined by points q_0 and q_1 is completely inside the free space:

$$\forall_{t\in[0,1]}\ tq_0 + (1-t)q_1 \in \mathcal{C}_{\text{free}} \tag{4.61}$$

where the most difficult question is how to discretize the $[0,1]$ interval so that the last number of checks are made. Many space decomposition methods have been proposed for efficient line collision checking, but the simplest and fastest one is to discretize the interval with a step size Δs using a distance metric $\delta(q,q)$, LineCollisionDetection (Algorithm 4.10) shows a naive implementation. First a Δq is computed using subtraction that respects the 0 and 2π identification for continuous joints. For every step, we find a new configuration along Δq such that the distance from the current configuration is exactly Δs. Since we are assuming a triangle inequality, we can test if we have surpassed the line-segment and terminate the collision.

The problem is finding a distance metric that can reduce samples on line collisions while helping RRTs sample the space of end-effector positions more uniformly. Even without considering a particular family of distance metrics, we can scale each coordinate of the configuration space before feeding it into the distance metric δ.

The most natural scaling should be proportional to the *importance* of each degree of freedom in moving the geometry of the robot. For the same physical distances, more important DOFs should have a larger distance than less important DOFs. One measure of the *importance* of a joint is by using the swept volume of all the robot links that are dependent on the DOF as shown in Figure 4.26. Exactly computing the swept volume is a very difficult problem [Kim et al (2003)], so we use a discretization technique by first discretely sampling a set of points on the convex decomposition, rotating them around the spans of each of the joints, using kernel density estimation to get the point density of the volume it spans, and finally thresholding the density to get a volume. Figure 4.26 shows the volumes generated using this simple, but powerful technique.

Algorithm 4.10: LINECOLLISIONDETECTION(q_0,q_1)

$q \leftarrow q_0$
$\Delta q \leftarrow q_1 \ominus q_0$
if $q \notin \mathcal{C}_{\text{free}}$ **then**
    return true
while true do
    $q \leftarrow q + t\Delta q \quad$ **s.t.** $t > 0 \bigwedge \delta(q, q + t\Delta q) = \Delta s$
    if $\delta(q0, q1) \geq \delta(q0, q)$ **then**
        /* Passed the end, so check the final point */
        return $q_1 \notin \mathcal{C}_{\text{free}}$
    if $q \notin \mathcal{C}_{\text{free}}$ **then**
        return true
end
return false

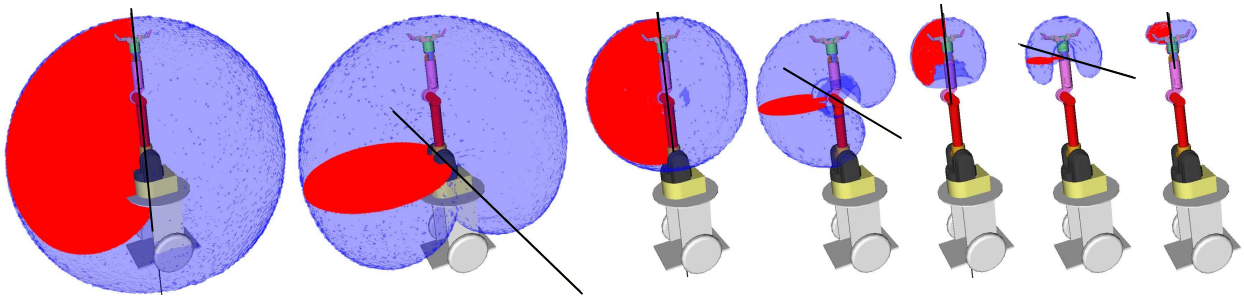


Figure 4.26: The swept volumes of each joint, where the parent joint sweeps the volume of its children's joints and its link.

Intuitively, the base joints should have more weight because of their larger impact on the end effector than joints farther in the chain. We compute the distance metric weights for the i^{th} joint using:

$$\omega_i = \left(\text{CROSSSECTION}(SV_i) * \sum_{j \in DependentLinks_i} LV_j \right)^{\frac{1}{3}} \tag{4.62}$$

where SV_i is the swept volume of the joint axis, and LV_j are the individual link volumes. Using just the area of the cross section is misleading since it measures all possible locations of the link positions, and does not consider where the links currently are. This could greatly skew the contribution of each joint, therefore we also multiply by the sum of volumes of all

	j_0	j_1	j_2	j_3	j_4	j_5	j_6
WAM Volume (m^3)	4.346	2.536	0.781	0.395	0.143	0.110	0.043
WAM Cross Section (m^2)	0.2668	0.0306	0.0297	0.0035	0.0023	0.0004	0.00039
WAM $\sum LV_j$ (m^3)	0.0066	0.0052	0.0050	0.0039	0.00364	0.00361	0.00357
WAM Weights ω_i	0.121	0.054	0.053	0.024	0.020	0.012	0.011
Puma Weights ω_i	0.133	0.090	0.015	0.006	0.002	0.001	-

Table 4.7: Statistics and final distance metric weights for the Barrett WAM joints.

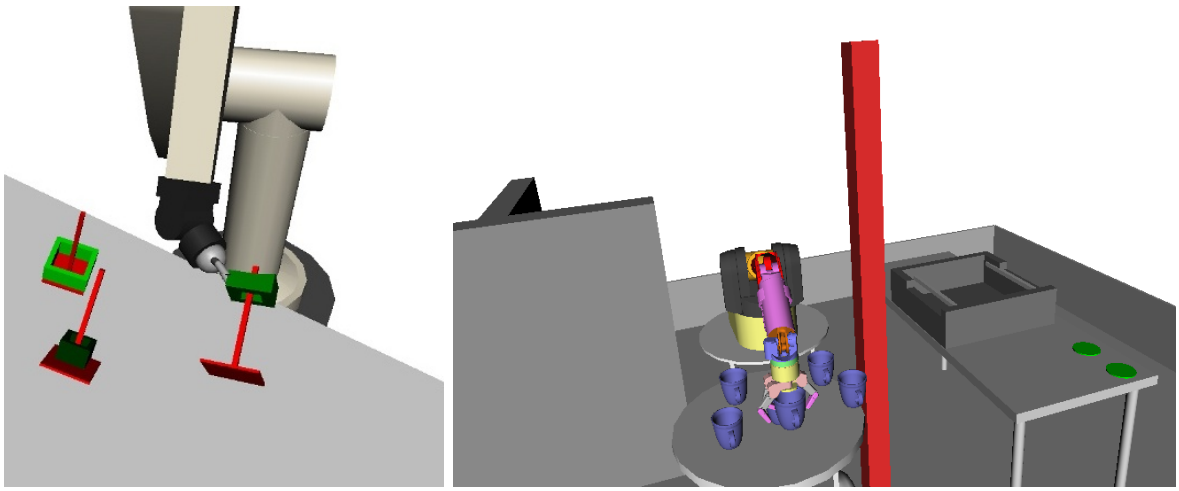


Figure 4.27: Example scenes used to test the effectiveness of the configuration metric.

links that are dependent on the i^{th} joint. Finally we take the cubed root since we're dealing with a weight on a single axis, but using values computed from sums on three dimensions. Table 4.7 shows the volumes, cross sections, and final weights for each of the joints on the 7DOF WAM and 6DOF Puma robots.

In order to test the effectiveness of the weights, we compute the average time to compute

	Uniform Weights	Swept Volume Weights (ω_i)
WAM scene	2.89s	1.24
Puma Scene	0.71s	0.57s

Table 4.8: Statistics and final distance metric weights for the Barrett WAM joints.

Parameter Name	Parameter Description
Collision δ	padding for the surfaces of each convex hull
Range Data δ	padding for removing colliding range data
Discretization	used for approximating the volume of a convex hull for swept volumes

Table 4.9: Convex Decomposition Parameters

a path on two scenes (Figure 4.27) when using both uniform weights and ω_i. The distance metric is weighted euclidean distance with 0 and 2π identified for continuous rotation joints. For purposes of fair comparison, we keep the planning step size the same for all tests: $\Delta s = 0.01$. Table 4.8 shows that the planning times are 0.5-0.8 times the original planning times. It should be noted that the ω_i weights allows much higher step sizes and therefore faster times since the max end effector movement given a $\Delta s = 0.01$ is smaller and more consistent than with uniform weights.

Table 4.9 shows the parameters related to using convex decompositions.

4.7 Object Detectability Extents

One of the major contributions of this thesis is to explicitly consider the *detectability extents* of an object during the planning process. The vision algorithm used to detect an object and camera sensor roughly define all the poses the object could appear in front of the camera such that its 6D pose can be computed accurately. Assuming that lighting is controlled or the vision algorithm's image features are invariant to lighting changes, we can compute the detection extents of the object by waving the camera or object around the other and building a map of all extracted object poses. The data gathered is intrinsically 6DOF, so it would take a very long time to explore the full 6D space of the relative poses between the camera and object. However, some degrees of freedom are notably less important than others in determining detectability. Assuming perspective distortions do not play a major role, then the rotation around the camera axis and translation on the image plane can be freely changed without affecting the pose detection results. Therefore, we define the projected map down to 3DOF as the *detectability extents* of the object. Figure 4.28 shows the raw extents gathered using a simple textured plane object detector. We parameterize the extents with respect to the distance to the object λ, and the direction of the camera view axis with v.

Using knowledge of the target object CAD model, robot kinematics, and camera calibration, it is possible to completely automate the data-gathering process. Similar to the

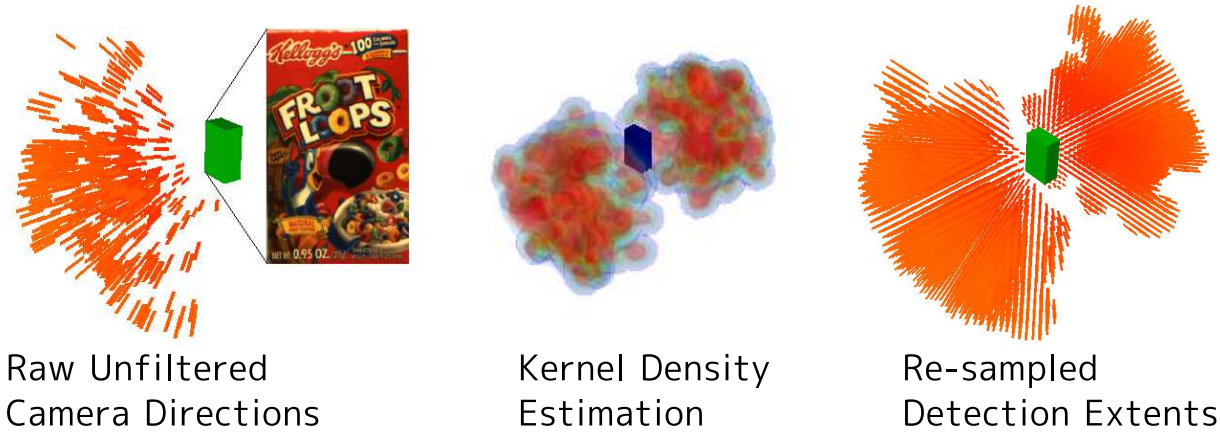


Figure 4.28: Camera locations that can successfully extract the object pose are saved.

automated camera calibration technique discussed in Chapter 6, we can attach a camera to the robot and automatically explore the space around the target object. The poses stored are independent of any errors from the robot encoders or extrinsic camera calibration, so no precision is lost if we just care about gathering the extents. However, relative positions of robot encoders are usually accurate to within 0.0001 radians, so one important piece of information that can be extracted is the error between the expected change in pose and the actual change in pose. The change in pose comes from using the initial detection results $T_{object}^{camera}(0)$ and the initial robot link position attaching the camera $T_{link}^{robot}(0)$ as ground truth. For any new observation i, the following must hold:

$$\delta\left(T_{link}^{robot}(i)^{-1}T_{link}^{robot}(0)\ ,\ T_{object}^{camera}(i)^{-1}T_{object}^{camera}(0)\right) = 0 \tag{4.63}$$

where δ is a distance metric on poses. The error can be used to scale the confidence values of the detection algorithm. Because no vision algorithm is perfect, there is always a chance it makes a mistake in the detection process or computes an inaccurate pose. We combine three different measures for quantifying the amount of *trustability* of the detected pose.

- The first is the vision algorithm confidence γ_{vision}, every algorithm should be able to return a value on how well its internal template fits the image.
- The second is the neighbor density around the extents map $\gamma_{stability}$. Computing this is a little tricky because it is dependent on the underlying pose sampling distribution. If using a robot, we can control this density, otherwise it is very difficult. Therefore, for every point we gatherer all the points around a ball of radius r, and compute how well

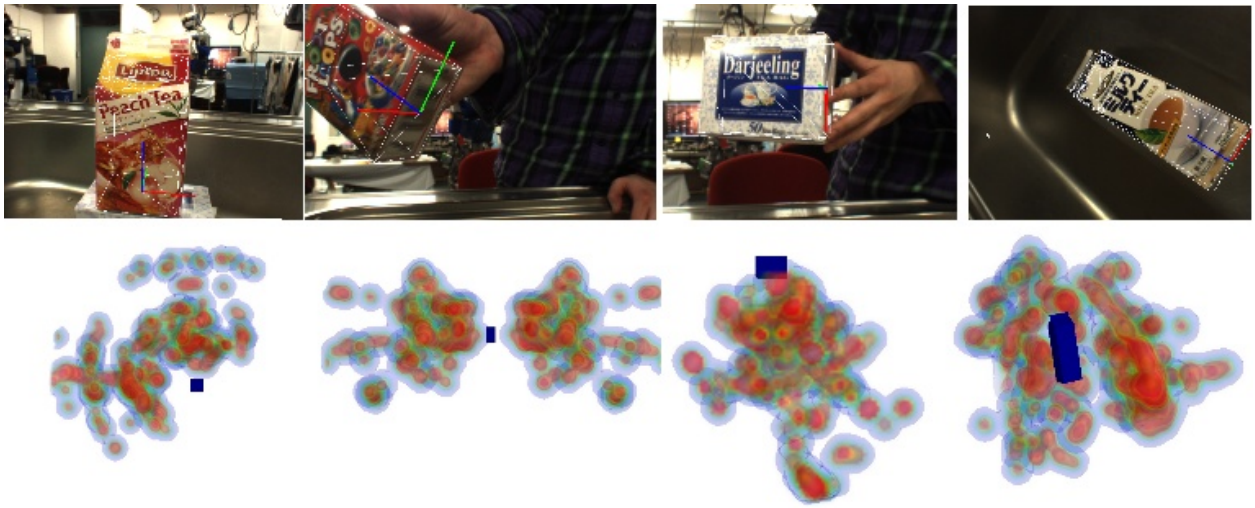


Figure 4.29: The probability density of the detection extents of several textured objects using a SIFT-based 2D/3D point correspondence pose detection algorithm.

Parameter Name	Parameter Description
Bandwidth	the bandwidth of each kernel for generating a probability distribution
Density Threshold	the neighbor density threshold for pruning outliers
Discretization	Used for sub-sampling the distribution for storing a discrete set of extents
Sampling	if using a robot, controls the sampling density of the sphere

Table 4.10: Detectability Extents Parameters

the entire ball is filled. Such an operation is easily achieved using nearest neighbors pruning and counting (Figure 4.28 middle). A *stable region* means that the camera or object can move slightly and still be guaranteed to be detected.

- The third is based on the expected error from Equation 4.63. It is converted into a confidence by modeling the probability as a gaussian.

The final confidence of a pose is a 3D map computed by:

$$\gamma = \gamma_{vision}\ \gamma_{stability}\ e^{-c\delta^2} \tag{4.64}$$

We use γ to prioritize sampling of the camera extents covered in Section 5.1. Figure 4.29 shows the detectability extents of several objects using a SIFT image features-based plane detector. Just like with inverse reachability, we can use the detection extents to help with the design of industrial bin-picking workspaces. The maximum distance an object can be

detected can be used to optimize the robot location with respect to the bin where target objects are in.

Table 4.10 shows the parameters related to generating detectability extents.

4.8 Discussion

The planning knowledge-base is an important concept for helping achieve automatic construction of manipulation programs. In this chapter, we divided the knowledge required beyond the robot and task specifications into seven different categories. Our formulations in each of these categories rests upon a vast amount of fundamental research proposed by previous researchers; however, one of the most important contributions here is tying each of these categories together and really focusing on automated generation of each component. We presented details and algorithmic descriptions beyond the fundamental theories to help understand these components in the context of applying them to real robot scenarios.

Computing fast and numerically stable analytical solutions to the inverse kinematics problem has been a classic problem with many mathematical solutions proposed. However, treating it as a search problem allows us**ikfast** to explore the entire solution space to find the quickest and most robust method. Although there exist robot kinematics that **ikfast** cannot solve yet, the kinematics of most robots used in the world today can be solved for. Assuming the existence of analytic inverse kinematics solutions can also help shift the focus of planning algorithms to use them rather than to assume no knowledge of the problem. We believe that **ikfast** will become a standard tool for future robotics research.

Grasping is one of the fundamental distinctions between manipulation planning and motion planning. In order to reason efficiently about grasps, we motivated a discrete grasp set approach to modeling the space. We presented three different methods that fall into this definition, and in each of them showed how to parameterize and discretize the space. Furthermore, we presented several algorithms to remove grasps that can be prone to slipping.

Considering arm reachability is one effective way of relating the configuration space of the robot to the workspace. Although the concept of reachability has been around for some time in the motion planning community, the parameterizations and usages different quite radically. In this chapter we stuck with the most flexible representation of the reachability by using 6D maps. We presented several algorithms on efficiently generating, sampling, and inverting the reachability space. Inverse reachability opens up a lot of possibilities in the context of base placement planning. From this analysis emerged a new concept called *grasp reachability* that allows us to explicitly relate a probability distribution on the robot base with a target object to be grasped. We believe such analyses will become very important in

automating the design of industrial scenarios.

We argue that convex decompositions are the most efficient way to represent body geometry. They can accurately model any geometry of the robot, and using them for collision and proximity detection is very straightforward due to their SAT-nature. We showed how convex decompositions are used for padding and processing range data. Furthermore, they allow efficient swept volume computation of the robot links. We presented a distance metric that computes weights for the joints based on the amount of geometry each joint can affect. We discussed several real-world issues with using convex decompositions like using them only for environment collisions and not self-collisions. Our hope is that all these issues convince every planning library developer to natively support and use convex decompositions.

Explicitly modeling the detectability extents of target objects allows us to methodically analyze the *information space* offered by a vision algorithm. We can determine where a camera should be move in the workspace without relying on a human to second-guess and teach the robot these heuristics. Furthermore, we presented a confidence measure map that helps prioritize the sampling of the camera locations. Usage of detectability extents is one of the pillars and major contributions of this thesis and planning with them will be discussed in Chapter 5.

Rather than individually treating these components in the planning knowledge-base, we argue the necessity of the information dependency graph in Figure 4.1. By explicitly keeping track of the information being used for generation of the model, we can split up real domain knowledge specified by a person from knowledge that can be auto-generated. It is true that a some of the generation processes are controlled by thresholds and discretization values. We offer some insight in how to automatically set these values, but this type of second-order automation still requires a lot of research. Some database components take a really long time to generate, so it is inefficient to have to rebuild the entire planning-knowledge base whenever a small modification is made to the robot or target object. Fortunately, the dependencies allow keeping track of changes and updating only those components that are affected. In Section A.1.5, we cover one way of indexing changes. In the future, the dependency graph can form the basis for a global database of robots and tasks.

We explicitly state the parameters each component uses for its generation in Tables 4.2, 4.3, 4.4, 4.5, 4.6, 4.9, and 4.10. Although we have discussed ways of automatically setting each of the parameters, their optimal values can depend on the precision requirements of the task, the repeatability of the robot, and the units the entire environment is defined in. Because the specifications mentioned in Section 2.1 do not encode such information, the setting of the values cannot be truly automated. For example, in this thesis, all robots are defined in meters and are human-sized. Such knowledge would allow us to set the reachability

discretization to 0.04 meters without sacrificing too much imprecision since we can argue that a robot's behavior should not change much if the target object moved by 0.04 meters. Future improvements to the robot and task specifications could define a few paramters that would allow an automatic analysis of error propagation vs speed of retrieval. Such analyses would allow a user to tweak the accuracy or responsivity of the system without worrying about the details.

Chapter 5

Planning with Sensor Visibility

We present a system that introduces an efficient object information gathering stage into the planning process. Specifically, we concentrate on how cameras attached to the robot can be used to compute an accurate target object pose before attempting to plan a grasp for the object. It is common for robot systems to treat grasp planning independent from the sensing capabilities of the robot, which results in a separation of the perceptive and planning capabilities that can lead to gross failures during robot execution. For example, a robot that sees an object from far away should not attempt to plan a complete path with grasps involved at that point; an early commitment of a global plan when environment information is inaccurate can lead to failures in the later execution stages. Instead, the robot should plan to get a better view of the object. Thus, it is important to setup the planning process to prioritize gathering information for the target object before prematurely wasting computation committing to a grasp plan.

There are many methods of handling sensor uncertainty in the planning phase depending on the level of expected noise in the system. For example, really noisy sensors that frequently return false positives require the planner keep track of object measurements using EKF filters [McMillen et al (2005)]. Furthermore, planning in dynamic environments requires the robot to keep track of what regions are stale due to no sensor presence, what regions are being actively updated, and where the robot is with respect to the map [Rybski and Veloso (2009)]. The less assumptions made on the quality of sensing data and the predictability of the environment means that the planning and sensor feedback loops have to be very tight, which makes the manipulation problem very difficult. By focusing on industrial settings and mostly static environments as outlined in Chapter 2, we can separate the problems due to dynamic, unpredictable changes and those of sensor noise and calibration errors. In Section 4.7, we built a *detectability extents* model that captures the entire sensing process and its

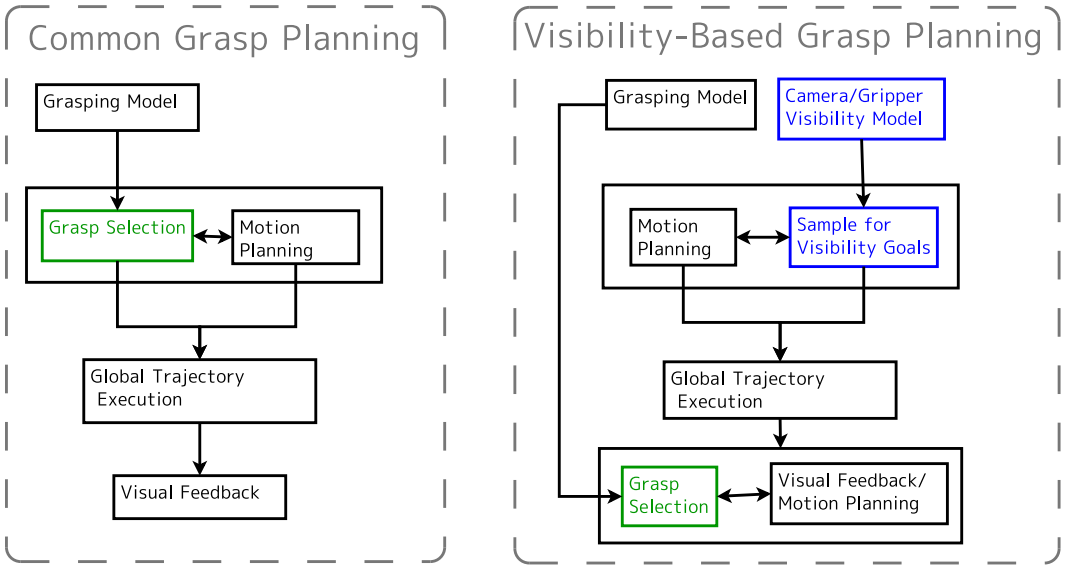


Figure 5.1: Comparison of a commonly used grasping framework with the visibility-based framework. Because the grasp selection phase is moved to the visual feedback step, our framework can take into account a wider variety of errors during execution. The robot platforms used to test this framework (bottom).

noise models. It encodes a probability directly proportional to the confidence of the detection process, the noise introduced in it, and the density of neighboring regions. By using just the detectability extents, we hope to find better and more confident measures of the locations of objects before attempting to grasp them. Because detectability extents encodes a probability distribution of where the camera should observe an object from, the planning process can mostly set aside the direct output of the vision algorithm and only concentrate on sampling the extents.

We focus on a framework that can effectively use cameras attached to the gripper. Unlike cameras mounted to the base or the head of the robot, gripper cameras can be maneuvered into really tight spaces. This allows the robot to view objects in high cupboards, microwaves, or lower shelves as easily as grasping them. Furthermore, the detection results from the gripper camera are transferred directly into the gripper coordinate system, which removes any robot localization or encoder errors from the final pose of the target object.

In Section 5.1, we show how to quickly sample robot configurations using the detectability extents models such that the object becomes detectable in the camera image. We use these samplers as a basis for a two-stage visibility planning process shown in Figure 5.1. In Section 5.2, we present experimental results that show performance of this method is as good as regular grasp planning, even though a second stage was added.

There are two steps to incorporating visibility capabilities:

1. A planning phase that moves the robot manipulator as close as safely possible to the target object such that the target is easily detectable by the onboard sensors.

2. An execution phase responsible for continuously choosing and validating a grasp for the target while updating the environment with more accurate information.

In Section 5.3, we present a visual feedback method that integrates grasp selection. This allows us to naturally fill the gap between the motion planning stage that picks grasps and the execution stage that attempts to reach those grasps. In most research, the entire environment with obstacle avoidance is rarely considered because of the computational complexities with gradient descent approaches. It is also hard for gradient descent to consider nonlinear constraints like properly maintaining sensor visibility and choosing grasps. To solve these problems, we present a stochastic-gradient descent method of planning that does not have the singularity problems frequently associated with gradient-descent visual servoing methods. Although many proposed robot systems include planning and vision components, these components are treated independently, which prevents the system from reaching its full potential.

The visual feedback framework is different from past research [Prats et al (2007a); Kim et al (2009)]. in that it analyzes how the environment and the plan as a whole are affected during the visual-feedback execution. By combining grasp planning and visual feedback algorithms, and constantly considering sensor visibility, we can recover from sensor calibration errors and unexpected changes in the environment as shown in [Diankov et al (2009)]. The framework incorporates information in data-driven way from both planning and vision modalities during task execution, allowing the models it uses to be automatically computed from real sensor data.

Case studies of a humanoid robot picking up kitchen objects and an industrial robot picking up scattered parts in a bin are presented in the context of this theory. We also show how to sample base placements guaranteeing a visible configuration for mobile manipulation.

5.1 Sampling Visibility Configurations

A *visibility robot configuration* guarantees that the camera attached to the robot is moved to a location where the target object is unobstructed and can be detected by the vision algorithm. The following have to be considered:

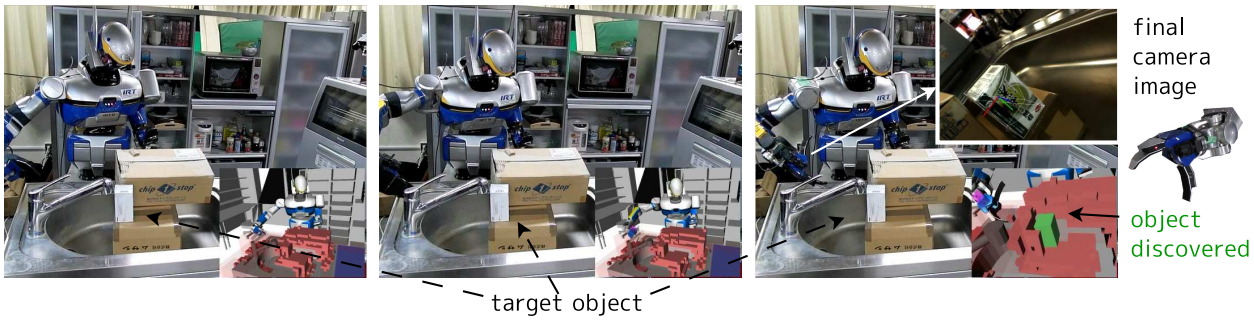


Figure 5.2: Object initially hidden from the robot, so its pose is not known with much precision. It takes the robot three tries before finding it.

- **Detectability.** The detectability extents map of the object used from the planning knowledge-base (Section 4.7). The stored map includes the camera direction v to the object, and the distance from the object λ.
- **Reachability.** Once a valid camera configuration can be sampled, the robot kinematics must allow the camera to be placed there.
- **Occlusions.** The target needs to be inside the camera view and no other obstacles should be in front of its path.

Although the final goal is to grasp the object, sampling visible configurations should not be considering grasps.

Before beginning to sample configurations, we assume a rough location of the target object. In the four-step manipulation process covered in Section 2.4, the first step is a search phase that detects a very rough locations of all objects of interest. This visibility phase can be considered as a reconfirmation process of the initial detection. Furthermore, the initial detection most likely occurs when the robot is far away from the object, therefore the error on the pose can be huge.

Due to noise on the target object, a sampled camera location does not necessarily guarantee the object will be visible once the robot moves there. Figure 5.2 shows a case where the object pose is initialized with a $\pm 4cm$ error, so the robot has to sample three different configurations before finding the object. Fortunately, the confidence values of the target detectability extents encode the camera neighborhood density, so camera samples can be organized keeping in mind that the object pose could be inaccurate.

5.1.1 Sampling Valid Camera Poses

The first step is to sample 6D camera poses with respect to the target coordinate system. The detection extents are first sampled using the confidence map to get the camera direction v and distance λ. The remaining parameters left for a full pose is the 2D image offset p_{2D} and the roll θ around the camera axis. If the camera image was infinite in size, then sampling any offset and in-plane rotation would suffice. The camera rotation $R(v,\theta)$ and translation $t(v,\theta)$ in the object coordinate system is defined as:

$$R^{object}(v,\theta) = \textsc{Rodrigues}\left(\begin{bmatrix}0\\0\\1\end{bmatrix} \times v, \cos^{-1} v_z\right)\begin{bmatrix}\cos\theta & -\sin\theta & 0\\ \sin\theta & \cos\theta & 0\\ 0 & 0 & 1\end{bmatrix}$$

$$t^{object}(v,\theta) = -\lambda\, R(v,\theta)\, K^{-1}\, p_{2D} \tag{5.1}$$

where K is the 3x3 intrinsic camera matrix assuming a pin-hole camera:

$$K = \begin{bmatrix}f_x & 0 & c_x\\ 0 & f_y & c_y\\ 0 & 0 & 1\end{bmatrix} \tag{5.2}$$

The usage of RODRIGUES [Belongie and Serge (1999-present)] allows the camera z-axis to be oriented towards the desired direction v by the minimum angle. In order to convert the 2D image offset into a 3D translation, p_{2D} is first converted into camera coordinates by K^{-1}. A multiplication with $R(v,\theta)$ converts the point into object coordinates. Finally the point rests on the $z = 1$ camera plane, so it is multiplied by the desired distance λ to get the final offset.

Unfortunately, the image has boundaries, and it is necessary to check that the entire object lies completely inside the image. Instead of transforming the object into image space, it is much easier to transform the image boundaries in object space. We start with defining the camera space boundaries on the $z = 1$ camera plane:

$$\Pi_x = \begin{bmatrix}-f_x\\0\\width - c_x\end{bmatrix} \quad \Pi_{-x} = \begin{bmatrix}f_x\\0\\c_x\end{bmatrix} \quad \Pi_y = \begin{bmatrix}0\\-f_y\\height - c_y\end{bmatrix} \quad \Pi_{-y} = \begin{bmatrix}0\\f_y\\c_y\end{bmatrix} \tag{5.3}$$

where $(width, height)$ are the dimensions of the image. A 3D point p is inside if:

$$\forall_i \Pi_i p >= 0 \tag{5.4}$$

Since the planes pass through the origin, the w component of the vectors is 0. Given a pose of the camera $\begin{bmatrix} R & t \end{bmatrix}$, it is possible to transform a plane Π_{2D} into a 3D plane in the object coordinate system with:

$$\Pi' = \begin{bmatrix} R\Pi_{2D} \\ -t \cdot \Pi_{2D} - \Delta_{safety} \end{bmatrix} \tag{5.5}$$

where Δ_{safety} pushes the planes forward a small safety margin since the object pose can be uncertain. The set of 3D planes $\{\Pi'\}$ define the camera visibility volume in the object coordinate frame $\mathcal{V}_{object}$ as a pyramid originating at t. To keep track of coordinate frames, for notational convenience, we use

$$\mathcal{V}_{object} = T^{object}_{camera} * \mathcal{V}_{camera} \tag{5.6}$$

Containment testing is just computing dot products. Setting aside environment occlusions, as long as the object is inside $\mathcal{V}_{object}$ and uses the detectability extents, then all of the sampled camera poses should yield good views. In order to reduce the search p_{2D} is set to the center of the image convex hull $(\frac{width}{2}, \frac{height}{2})$.

Gripper Masks

Because the camera is attached to a link on the robot, it could always be looking at a constant static silhouette of the robot geometry. Figure 5.3 shows two examples of cameras attached to the gripper links of the robots, with the robot fingers blocking the camera from observing that part of the environment. It might not always be possible to design sensors that have a complete unobstructed view; therefore, we go over the computations necessary to handle these *gripper masks* as efficiently as possible. Clearly, any visible configuration should be outside of the gripper mask and inside $\mathcal{V}_{camera}$.

Although Section 5.1.2 shows how to handle arbitrary environment occlusions using ray casting, having a constant mask in the camera image leads itself to a unique reduction in the equations without any extra computation cost. The idea is to compute the convex polygon of the largest free space in the camera image that does not contain the gripper mask (blue regions in Figure 5.3). A valid camera sample has to fully contain the projection of the object in this convex polygon, which is effectively the same computation as performed with the camera visibility volume $\mathcal{V}_{camera}$. In fact, approximating the region with convex hulls allows us to define a new visibility volume $\mathcal{V}_{staticmask}$ and replace $\mathcal{V}_{camera}$. We set the new p_{2D} offset as the center of this new convex hull.

Computing the largest collision-free convex hull is a hard problem [Chew and Kedem (1993); Borgefors and Strand (2005)], so we use a randomized algorithm that samples supporting points on the boundary of the gripper mask and checks if the resulting convex

Figure 5.3: The real robots with the a close-up simulation of the wrist cameras are shown in the top two rows. Given the real camera image, the silhouette of the gripper (bottom) is extracted (black) and sampled (red), then the biggest convex polygon (blue) not intersecting the gripper mask is computed.

polygon is all inside free space. Depending on the number of supporting points, running this procedure for an hour should yield a good approximation of the largest volume.

Representing the free space with one convex hull might leave out big chunks of free space where it is possible for object to be visible in. If this becomes a problem, then representing it with a convex decomposition should still maintain the computational advantages of the visibility volumes, except there will be more than one to handle.

The computational efficiency of the mask has one disadvantage which is that if the mask changes, the convex hull will have to be recomputed. The masks in Figure 5.3 depend on the preshape of the gripper, and it becomes necessary to always use the same preshape when performing the visibility testing. If this becomes a problem, caching multiple masks can be a viable option since the convex hull with N planes requires just $3N$ numbers. If the DOF of the gripper or reflected joints are too much, then it is only recommend to cache only the static unchanging regions of into the mask and leave the rest for the environment occlusion checking.

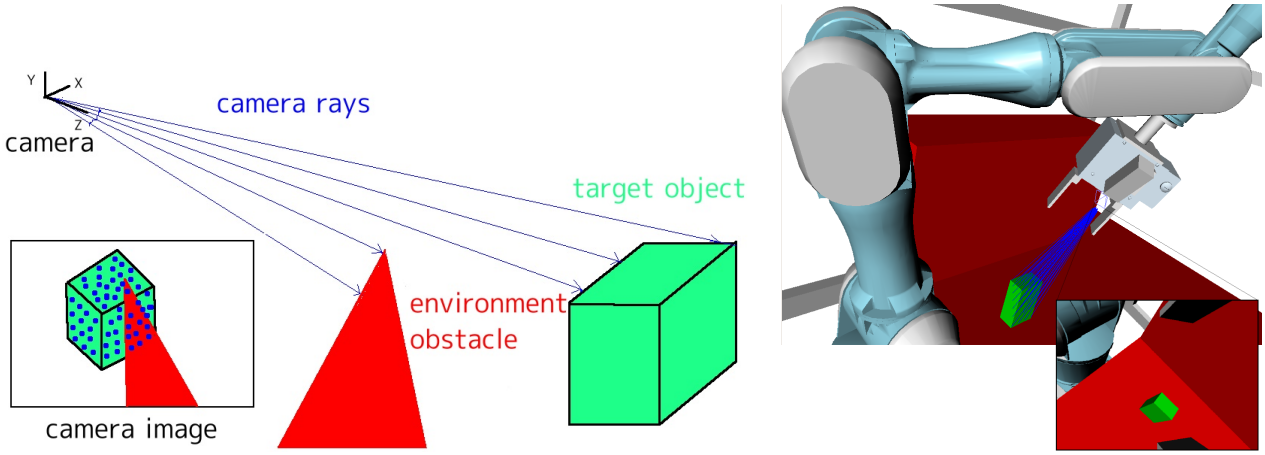


Figure 5.4: For every camera location, the object is projected onto the camera image and rays are uniformly sampled from its convex polygon. If a ray hits an obstacle, the camera sample is rejected. The current estimate of environment obstacles is used for the occlusion detect. As the robot gets closer and new obstacles come into view and a better location of the target is estimated, the visibility process will have to be repeated.

5.1.2 Detecting Occlusions

To check if the environment obstructs the target object when viewed from a specific camera location, we shoot all rays from the camera toward the object and check if they hit an obstacle before they hit the object. In the past, this was done by rendering the image and letting the graphics hardware perform the depth computations [Michel (2008)]. But for speed and potential parallel processing requirements, we present a method that samples the projected object surface and checks ray collisions using a standard collision checker (Figure 5.4). Sampling the projected surface allows us to remove any complex geometry and just consider the region of the object inside the camera; it also allows control of the density of rays.

Defining the point set of the surface of the objects as $\mathcal{O}$, and a camera pose as $T = \begin{bmatrix} R & t \end{bmatrix}$, the set of all projected points in the camera image is

$$proj(T, \mathcal{O}) = \{proj(K\ (R\,p + t)) \mid p \in \mathcal{O}\}. \tag{5.7}$$

Then the directions of all the rays originating at the camera origin that should be tested for collisions are defined by

$$\mathcal{R}(T) = K^{-1}\ \textsc{SampleArea}(proj(T, \mathcal{O})) \tag{5.8}$$

where SAMPLEAREA uniformly samples the projected area of the object. The density of sampling is controlled so as not to miss small details, it is set to $\frac{20}{f_x}$ for all experiments. The object is fully visible in the camera if all rays hit the object and it is fully inside the visibility volume $\mathcal{O} \subseteq \mathcal{V}_{object}$ (Figure 5.4). An issue that comes up with frequently in real robot experiments is that the initial target object can be slightly inside an environment obstacle like a table. This means that we should threshold the number of obstacle ray hits as some percentage of the object visible area.

Algorithm 5.1: $q \leftarrow$ SAMPLETRIANGLEAREA(v_0,v_1,v_2,ρ,$N_{allowed}$)

$v_1 \leftarrow v_1 - v_0$; $\quad v_2 \leftarrow v_2 - v_0$
$L_2 \leftarrow |v_2|$; $\quad n_2 \leftarrow \frac{1}{|v_2|} v_2$
$proj_1 \leftarrow |n_2^{\perp} \cdot v_1|$
$num_1 \leftarrow \lfloor \frac{proj_1}{\rho} \rfloor$
$\Delta_1 \leftarrow \frac{v_1}{num_1}$; $\quad \Delta_2 \leftarrow \frac{v_1 - v_2}{num_1}$; $\quad \Delta_L \leftarrow |n_2 \cdot v_1| + |n_2 \cdot (v_1 - v_2)|$
$r_1 \leftarrow v_0$; $\quad r_2 \leftarrow v_0 + v_2$
for $j \in [0, num_1]$ **do**
    $num_2 \leftarrow \lfloor \frac{L_2}{\rho} \rfloor$; $\quad r \leftarrow r_1$
    $\Delta_r \leftarrow \frac{r_2 - r_1}{num_2}$
    for $k \in [0, num_2]$ **do**
        if not TESTRAY(r) **then**
            $N_{allowed} \leftarrow N_{allowed} - 1$
            if $N_{allowed} < 0$ **then**
                return false
        $r \leftarrow r + \Delta_r$
    end
    $r_1 \leftarrow r_1 + \Delta_1$; $\quad r_2 \leftarrow r_2 + \Delta_2$; $\quad L_2 \leftarrow L_2 + \Delta_L$
end
return true

Unfortunately, uniformly sampling the projected region is dependent on the complexity of $\mathcal{O}$. In practice, we temporarily substitute the target object with a simpler convex shape like a box; orientated bounding boxes are especially easy to handle since they only show at least three of their six faces. In the end, each primitive reduces to a set of triangles which we sample with a density of ρ. SAMPLETRIANGLEAREA (Algorithm 5.1) shows how a triangle sampled given the projected triangle points in camera space. $N_{allowed}$ is the number of ray hits allowed before an occlusion is declared. We call the final function to test for

occlusions OnObject($r, T^{object}_{camera}, \mathcal{O}$). Its implementation just projects the primitive and calls SampleTriangleArea repeatedly.

5.1.3 Sampling the Robot Configuration

In this analysis, we assume the camera is attached to the end of an arm that has a 6D inverse kinematics solver. We discuss ways of relaxing this assumption towards the end. SampleVisibility (Algorithm 5.2) shows how a camera location is sampled, robot configuration is chosen, and all the constraints are checked before accepting it. First we sample the transformation of the camera outlined above using the vision algorithm's detectability extents (Section 5.1.1). Then the object is checked if it lies completely inside the camera visibility volume $\mathcal{V}$. If it does, we sample the robot inverse kinematics solutions of the manipulator until we find a collision free robot configuration such that no robot link intersects with the camera visibility volume. For every solution, we set the robot configuration q and check if any part of the environment or robot is occluding the object using ray casting (Section 5.1.2). The inverse kinematics need to be solved before the environment check since the robot can block the camera.

Algorithm 5.2: $q \leftarrow$ SampleVisibility($\mathcal{O}$)

```
1 while {v, λ} ← SampleDetectionExtents() do
2     T^object_camera ← T_O [ R(v,θ)   −λ R(v,θ) K^−1 p_2D ]
                            [   0                  1      ]
3     if O ⊆ T^object_camera * V_camera then
4         for q ← IK(T^world_object T^object_camera T^camera_arm) do
5             if ∀_{r∈R(T^camera_object)} OnObject(r, T^object_camera, O) then
6                 return q
7         end
8 end
```

To achieve faster sampling times, we perform the following optimizations:

- SampleDetectionExtents samples without replacement.
- Once we have the camera transform and before checking for an IK solution, we check collision with just the gripper. Because the gripper preshape is set, all the child link transforms of the gripper can be determined regardless of the arm joints.

	PA-10	WAM
Sample First Solution	**0.026s**	**0.009s**
Sample First Solution (many obstacles)	**0.538s**	**0.097s**
Planning Time	**0.188s**	**1.215s**
Planning Time (many obstacles)	**0.905s**	**1.289s**

Table 5.1: Average processing times for the first visibility stage for thousands of simulation trials.

Sampling with Ray Parameterizations

Currently the sampling of camera locations relies on a 6DOF inverse kinematics solver, which constrains the camera to only lie on the end effectors of arms. However, by treating the camera viewing direction as a 4DOF ray, we can use the ikfast ray inverse kinematics solver that only requires 4 joints to freely move around the environment (Section 4.1.6). Using ray inverse kinematics can relax the camera to be attached to the elbow, where it is more natural. We would be able to remove the unnecessary 2DOF redundancy set by p_{2D}, thus decreasing the size of the models and decreasing planning times. It should be noted that although 2DOF are redundant, they can greatly increase the free space of the camera, so any decisions to set a camera lower in the arm should be confirmed with simulations.

5.2 Planning with Visibility Goals

In order to plan to the sampled camera regions, we use RRTConnect* with SampleVisibility as the goal sampler. This allows all possible goals to be considered while not demanding their full computation during planner initialization. To compute average running times, we create 20 scenes with randomly placed obstacles and record the times for each individual component. Example scenes are shown in Figure 5.7. To test the effectiveness of the framework, we have three different robots perform reach-and-grasp tasks in complex environments shown in Figure 5.7:

- The first scenario is a PA-10 arm with a one degree gripper grasping a box.
- The Barrett WAM with the Barrett Hand inside a kitchen environment.
- The HRP3 humanoid robot with a 6 DOF hand inside a kitchen environment.

For the first planning stage, we record the sampling and planning times. The average time it takes to sample the first good valid inverse kinematics solution that meets all the visibility constraints is shown in (Table 5.1). Even with the extra visibility constraints, it

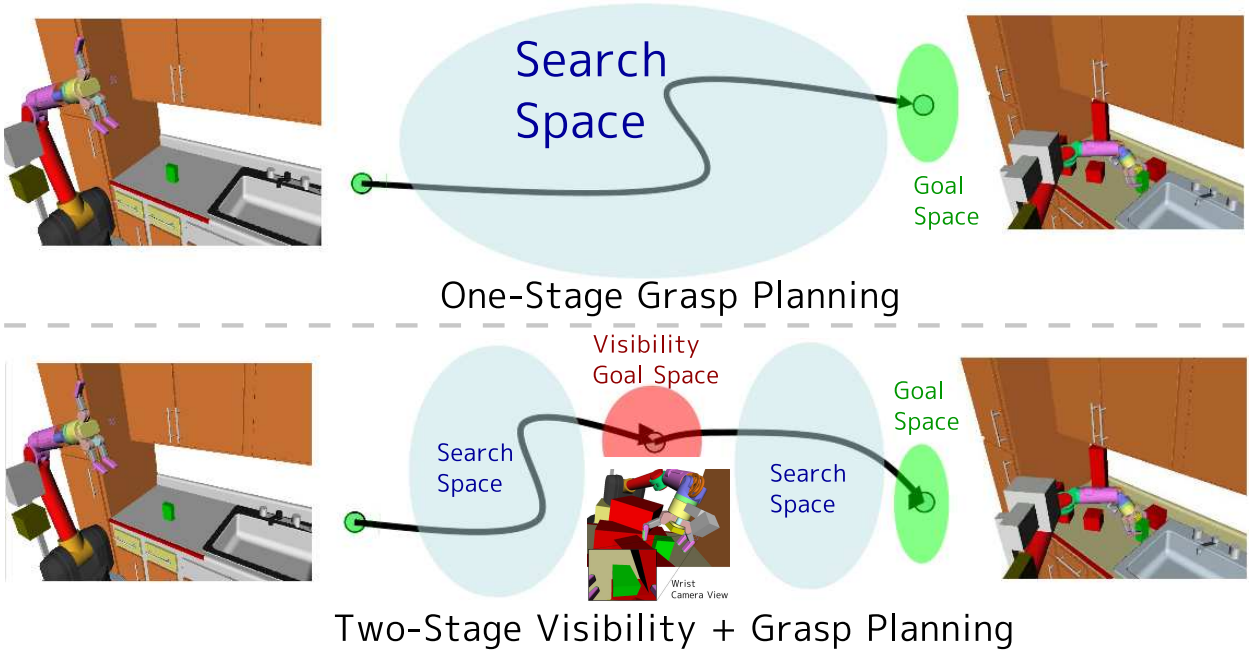


Figure 5.5: The two stage visibility planning method is as efficient as the one-stage grasp planning method because it divides and conquers the free space.

surprisingly takes a short amount of time to sample a goal. Furthermore, we combine the sampling with a planner and compute the average time it takes to move the robot from an initial position to a sampled goal. Note that planning times include the time taken for sampling the goals.

In fact when performing grasp planning by moving first to to a visibility goal before attempting to grasp the object, total planning times in general are very similar to grasp planning from the beginning. This phenomena occurs because the visibility goals can act as a *keyhole configuration* by first getting the camera, and consequently gripper, very close to the target object. Figure 5.5 shows how the search spaces compare when using a one-shot method vs the divide-and-conquer method with sampling the visibility space. Because the required precision of robot execution is greatly reduced for the first stage to a visibility goal, planning becomes ridiculously easy and fast. Only obstacles need to be avoided and the final gripper position is usually no where near an obstacle. Once at a visibility configuration, the gripper is already so close to the object, that it becomes possible to start visual servoing to the final grasp without considering a planner. The next section shows results when the second stage is replaced by a stochastic gradient descent method rather than a full search-based planning algorithm.

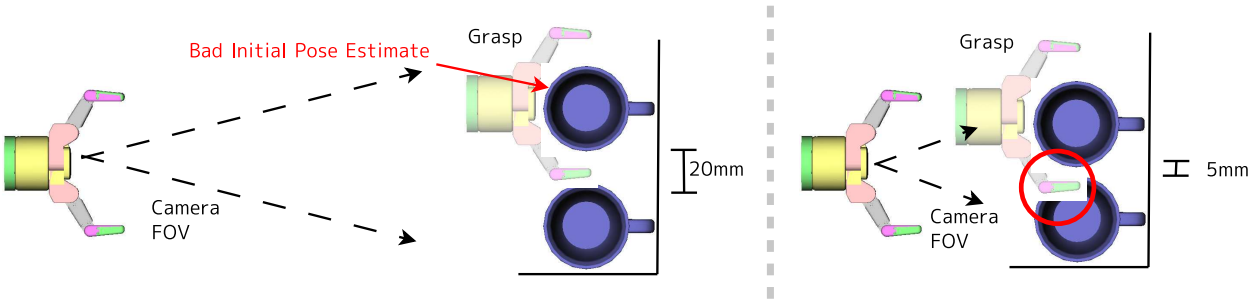


Figure 5.6: When initially detecting objects, the robot could have a wrong measurements. If it fixes the grasp at the time of the wrong measurement, then when it gets close to the object, the grasp could be infeasible. This shows the necessity to perform grasp selection in the visual feedback phase.

5.3 Integrating Grasp Selection and Visual Feedback

One necessary component for robust behavior in quickly changing scenes is a real-time visual-feedback phase that can compensate for environment uncertainties. The purpose of visual feedback is to update the virtual world quickly and to decide if the current plan is still valid or needs to be re-planned. Much work has been done in combining position-based feedback, impedance control, and other linear constraints. In the standard visual servoing formulation, the robot goal is defined with respect to a *task frame*; as new information comes in, the task frame is updated and the robot attempts to get closer to the goal defined in this frame using gradient descent techniques [Prats et al (2007a); Kim et al (2009); Kragic et al (2001)]. Having one goal fixed to the task frame and adding potential-fields for obstacles allows gradient-based methods to work really well in simple scenes.

In order to motivate the necessity for grasp selection in the visual feedback phase, consider the example in Figure 5.6 of the Barrett Hand attempting to grasp one of two objects in a scene with a camera attached to its gripper. When the robot first starts the manipulation planning phase, it is far away and computes that the objects are 20mm apart and decides that it can safely put one of its fingers between the objects. As the robot switches to its visual-feedback loop and starts getting better measurements of the objects, it updates their poses and sees they are actually 5mm apart. This sudden change in distance now prevents the robot to put its fingers safely between the two objects. This requires the robot to choose a new grasp quickly to compensate for the change, or otherwise it will declare failure.

Recently, a framework for combining randomized planners and visual servoing techniques has been proposed by [Kazemi et al (2009)]; it maintains constraints using a planner and locally modifies this trajectory in real-time using visual servoing. This allows relaxing of the

Algorithm 5.3: path $\leftarrow$ VisualFeedbackWithGrasps(q_{start}, G_{raw})

```
1  G ← Sort(using δ_gripper(FK_gripper(q_start), G_raw))
2  γ ← 2.5
3  while G ≠ ∅ do
4      for g ∈ G do
5          q_goal = arg min_{q∈IKSolutions(g)} δ(q, q_start)
6          if δ(q_goal, q_start) < γ then
7              path ← RandomDescent(q_start,q_goal)
8              if path ≠ ∅ then
9                  return path
10             else
11                 G ← G − {g}
12             end
13     end
14     γ ← 1.5 γ
15 end
16 return ∅
```

constraints on the environment obstacles, but the reality is that the goal is not just one fixed pose with respect to the task frame, it is all possible grasps that can manipulate the object. If the feedback stage does not consider the grasp selection process during visual feedback, it cannot quickly compensate for changes in environment and will have to restart the entire planning process from the start. In fact, we stress the importance of performing grasp selection for the target during visual-feedback execution because more precise information about the target's location and its surroundings is available (Figure 5.1).

5.3.1 Stochastic-Gradient Descent

The grasp selection process in the gradient descent algorithm should take into account both the collision obstacles and the current robot position. Although many grasps could be collision-free and reachable from the robot's perspective, the selection process is more successful when they are prioritized depending on the current environment. We use two metrics to prioritize grasps. The first is the difference between rotations of the gripper used to represent the grasp space:

$$\delta_{gripper}(T_0, T_1) = \cos^{-1} |\textsc{ExtractQuaternion}(T_0) \cdot \textsc{ExtractQuaternion}(T_1)| \tag{5.9}$$

where the $\cos^{-1}$ is the Haar measure on $SO(3)$. Each grasp is checked for the existence of a collision-free inverse kinematics solution. The second metric for prioritizing grasps is the distance between the solution and the current robot configuration.

VisualFeedbackWithGrasps (Algorithm 5.3) shows how the grasps are ordered using these two metrics before calling the stochastic gradient descent algorithm. G_{raw} is the set of all possible, validated grasps using GraspValidator (Algorithm 3.4), γ forces the closest configuration solutions to be considered first before the farthest ones, $\delta(q, q)$ is the distance metric on the configuration space of the robot used for choosing closer inverse kinematics solutions.

It is also possible to maintain the visibility of the target while computing a path. We define $\mathcal{C}_{visible}$ as the space of all collision-free robot configurations in $\mathcal{C}_{free}$ that maintain the visibility constraints with the object:

$$\begin{aligned} \mathcal{C}_{visible} = \{\ q \mid q \in \mathcal{C}_{free}, \\ \mathcal{O} \in T^{object}_{camera} * \mathcal{V}_{camera}, \\ \forall_{r \in \mathcal{R}(FK^{camera}_{object}(q))} \textsc{OnObject}(r, q, \mathcal{O})\ \} \end{aligned} \tag{5.10}$$

RandomDescent (Algorithm 5.4) shows the visual feedback algorithm. Given a grasp transform, we first find the closest inverse kinematics solution in configuration space and set that as the goal. Then we greedily move closer to the goal and validate with $\mathcal{C}_{visible}$. SampleNeighorhood returns a sample in a ball around q. After the grasp frame T_g gets within a certain distance τ from the goal grasp, we start validating with $\mathcal{C}_{free}$ instead since it could be impossible for the object to be fully observable at close distances. Because the gripper is already very close to the object and the object is not blocked by any obstacles due to the visibility constraints, such a simple greedy method is sufficient for our scenario. Another advantage of greedily descending is that an incomplete plan can be immediately returned for robot execution when planning takes longer than expected.

It should be noted that it is not always possible to maintain the visibility constraints. In fact when the current configuration is close to q_{goal}, it is not important to sense the target anymore. Furthermore, if the gripper pose is very far away from the grasp current chosen, then we can also relax the visibility constraint. It is very common for the robot to get stuck moving in one direction due to joint limits, so it has to completely reverse directions to get into the correct reachability region. Relaxing the constraints on $\mathcal{C}_{visible}$ allows this to happen.

Algorithm 5.4: path $\leftarrow$ RANDOMDESCENT(q_{start}, q_{goal})

```
path ← {q_start}
q ← q_start
for i = 1 to N do
    q_best ← ∞
    for i = 1 to M do
        q' ← SAMPLENEIGHBORHOOD(q)
        if q' ∈ C_free and δ(q', q_goal) < δ(q_best, q_goal) then
            if δ_gripper(FK_gripper(q_goal), FK_gripper(q')) > τ or q' ∈ C_visible then
                q_best ← q'
    end
    if q_best ≠ ∞ then
        path.add(q_best)
        q ← q_best
        if δ(q_best, q_goal) < ε then
            return path
end
return ∅
```

Using the same scenes as in Figure 5.7, we record average time to complete a visual feedback plan while maintaining visibility constraints (Table 5.2). In order to compute the how much visibility constraints affect the times, we also record statistics for the feedback stage ignoring the camera. Although the times vary greatly depending on the situation, results show that the feedback algorithm can execute at 2-10Hz. Looking at the planning times, scenes with more obstacles usually finish faster than scenes with fewer obstacles. This phenomena is most likely because obstacles constraining the feasible configuration space of the robot and guide it toward the goal faster.

Just like all feedback algorithms, VISUALFEEDBACKWITHGRASPS is constantly running and re-validating the scene. When it returns, start executing the path and run it again assuming that the robot will switch to the new trajectory in Δt s in the future. The path to the object is always validated and can sometimes stop the robot [Ferguson and Stentz (2006)].

As discussed in Section, 5.2, the combined planning times for the first planning stage and the second visual feedback stage are comparable to the planning times of previous manipulation systems that do not even consider visibility [Berenson et al (2007); Diankov

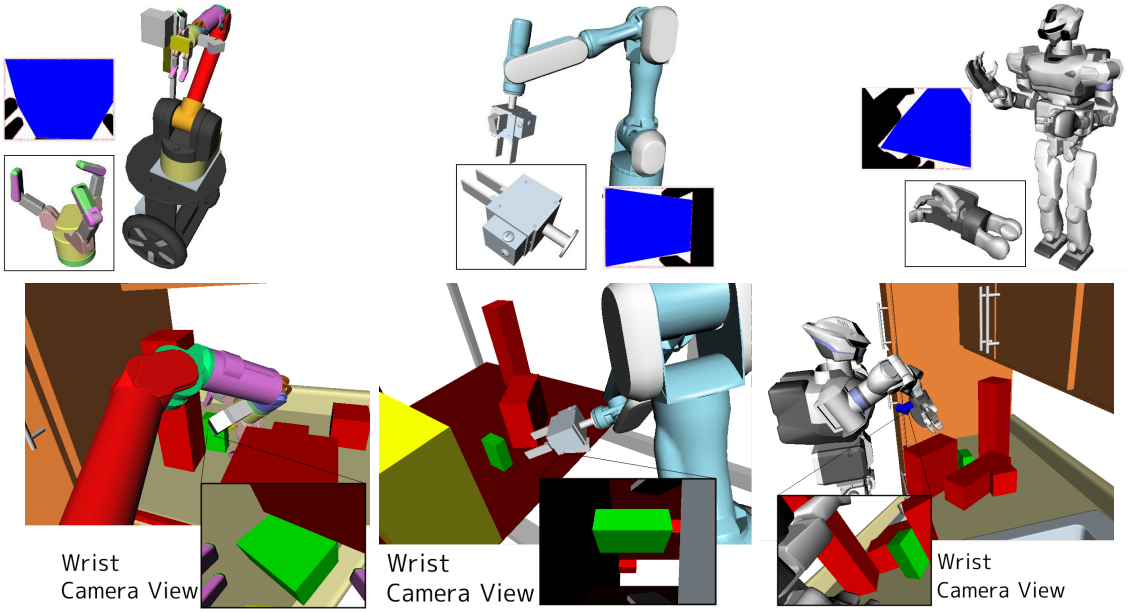


Figure 5.7: The scenes tested with the visibility framework. The images show the robots at a configuration such that the visibility constraints are satisfied.

	PA-10	WAM
Few obstacles/Visibility Constraints	**0.626s (93%)**	**1.586s (96%)**
Many obstacles/Visibility Constraints	**0.512s (83%)**	**0.773s (67%)**
Few obstacles/No Visibility	**0.117s (94%)**	**0.406s (97%)**
Many obstacles/No Visibility	**0.098s (86%)**	**0.201s (71%)**

Table 5.2: Average planning times (with success rates) of the visual feedback stage.

et al (2008a); Srinivasa et al (2008)]. The second visual feedback stage does not have to consider complex planning scenarios since the target is right in front of the camera, this allows it to finish quickly.

5.4 Humanoid Experiments

Using the concepts from grasp planning, visibility, and reachability, we show how to perform simple pick-and-place tasks with a dual-arm humanoid robot. We attached a camera to the to right gripper (Figure 4.12) and trained the detectability extents of five different objects. Figure 5.11 shows the environment where a flash LIDAR is placed on the top right corner

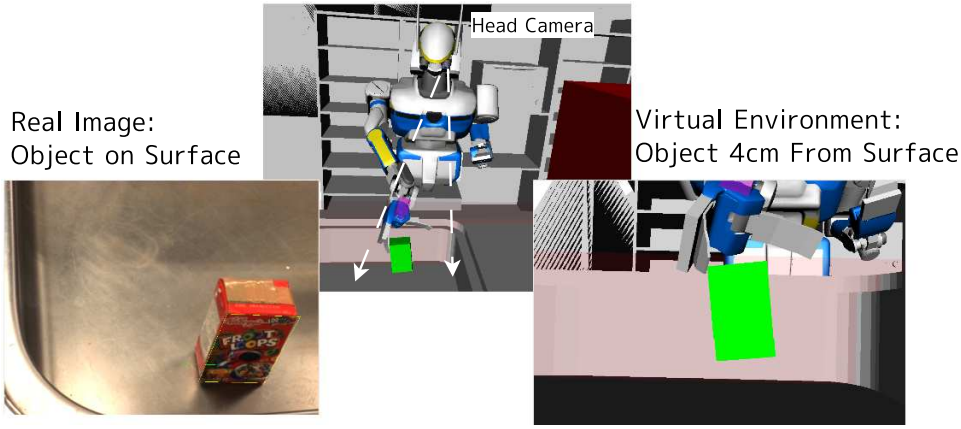


Figure 5.8: The calibration error from the head camera can be very inaccurate when detecting over large distances.

in order to provide real-time measurements of obstacles in the environment. As discussed in Section 4.6.2, we remove the robot and detected target object from the range data before attempting to avoid it. The two intrinsic parameters on both cameras are accurately calibrated; however, the extrinsic calibration were not set accurately for both in order to test the robustness of the system against calibration errors. Because the head cameras have a big focal length, far away objects are easy to see, which also implies bigger pose detection errors. Figure 5.8 shows that we can get up to 4cm error when detecting the pose of an object with the head cameras. The sink was chosen because it is very difficult to see all places of the sink with the head cameras, while the gripper camera has complete freedom. The sink spatial location also really stresses the reachability space of the robot. Without informative base placement sampling (Section 3.6.1), it becomes very difficult to be able to grasp the object. We set four different arm definitions: 7DOF left arm, 7DOF left arm + 1 DOF torso, 7DOF right arm, 7DOF right arm + 1 DOF torso. The robot actually has 2 torso joints, but it is inefficient to have a 9DOF inverse kinematics solver. Therefore, inverse reachability models are precomputed for the remaining over torso joint using these values: $\{-\frac{\pi}{4}, \pi, \frac{\pi}{4}\}$. The planner considers all 4 arms along with all 12 inverse reachability models.

The goal for the robot is to pick up objects from the sink and place them on the counter to the right of the image. First, the robot searches for objects using the head camera by moving around the environment. Assuming the base is fixed for now, the robot first samples a visible configuration. For really small reachability spaces and a really robust algorithm, the volume of the possible camera configurations can be immense, but the reachable camera configurations is only a percentage of that, so SampleVisibility will spend a lot of time rejecting samples. In order to restrict sampling to reachable regions, we intersect the reach-

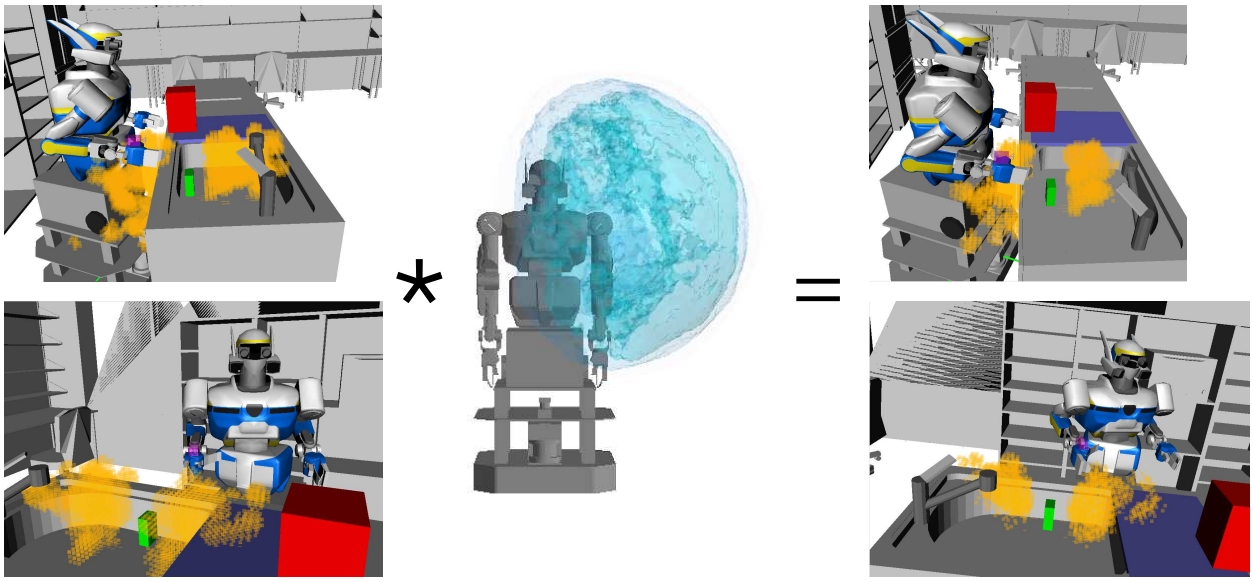


Figure 5.9: Can efficiently prune out all the visibility candidates before sampling by using reachability.

ability spaces and camera visibility poses as shown in Figure 5.9. If all the data is cached in kd-trees, the intersection operation is on the order of a few milliseconds, therefore a lot of time is saved. Once the robot moves to a visible configuration, it recomputes the target pose and starts grasp planning. Figure 5.10 shows the sequence of moves made. The grasp plan always moves the gripper to a retreated grasp along the grasp direction, this allows for padding object boundaries so robot does not hit it on its way there (Section A.3.1). The robot then moves along the grasp direction, grasps the target, and moves it to its destination. Figure 5.11 shows several real-world experiments on similar scenes, but different objects. After all the components were put together and padding was used for avoiding edge collisions, the robot could succeed in grasping the objects 9 out of 10 times. All failures were due to slipping of the gripper fingers while grabbing the object. There are two explanations for this:

- There could be imperfections in modeling of the gripper geometry and thus getting a bad grasp in the real world, while it is stable in simulation. The gripper fingers were custom made, and no CAD model was readily available for them, therefore they were measured by hand. All fragile grasps were pruned using the repeatability metric in Section 4.2.1 with a noise of 0.01m. Manually checking every grasp in the grasp set confirms that no fragile grasps were present.
- The pose of the object was a little off due to the vision algorithm. As we previously

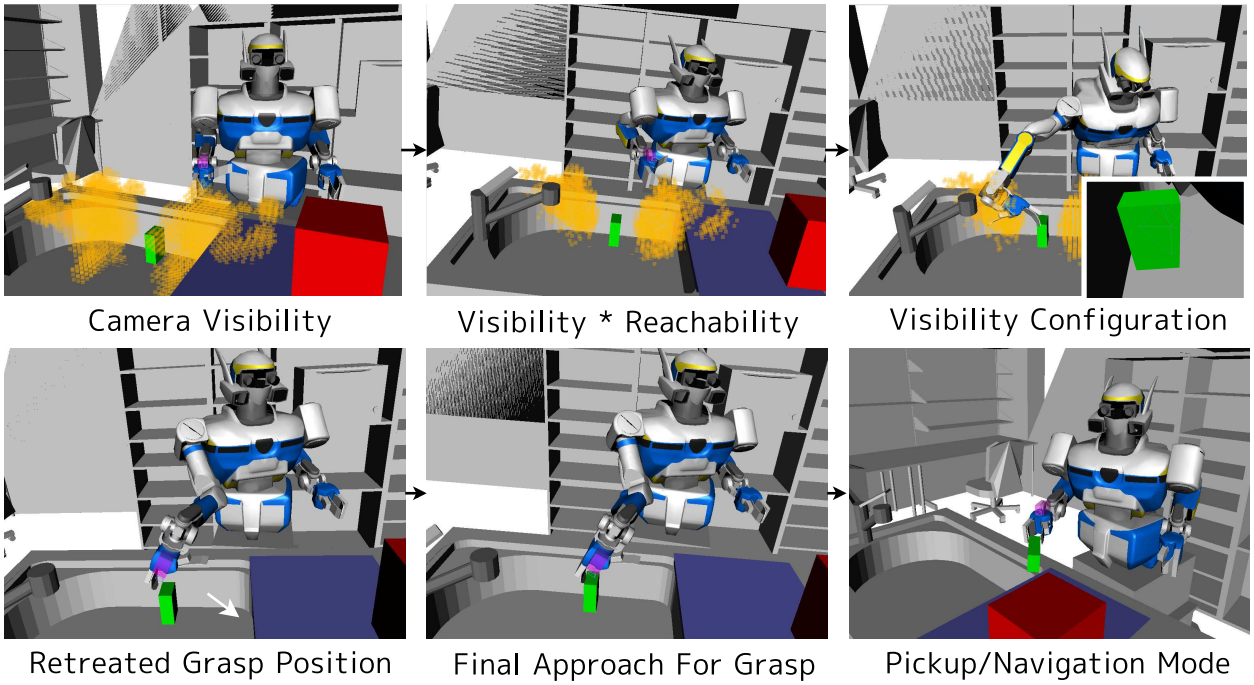


Figure 5.10: The full process of mobile manipulation using visibility, reachability, and base placement models.

mentioned, none of the camera positions were accurately calibrated because we wanted to test the power of the gripper cameras. The fact that most of the grasps succeeded is because the gripper camera usually detected objects from 0.1m-0.2m, which reduces the calibration error to a few millimeters.

We also performed mobile manipulation experiments as shown in Figure 5.12. Once the object is visible, the robot starts sampling base placements such that it can achieve a visibility configuration. The robot then moves to this configuration and continues sampling visibility configurations until it finds the object. Just considering visibility can lead to the situation in Figure 5.13 where the robot cannot grasp the object from that location, which would require the robot base to move again to grasp it. For wheeled bases on a carpet, using a laser range finder with a SLAM-built map, the navigation process accuracy is on the order of 2cm-4cm, which means that the object always has to be recomputed by the camera when the base stops moving. Going back to the original problem of sampling a base placements that allow grasps and visibility configurations, it is important to keep in mind that the object's pose is not known accurately. As a result, full collision detection during grasp validation cannot be done. However, the reachability of the grasp ignoring collisions

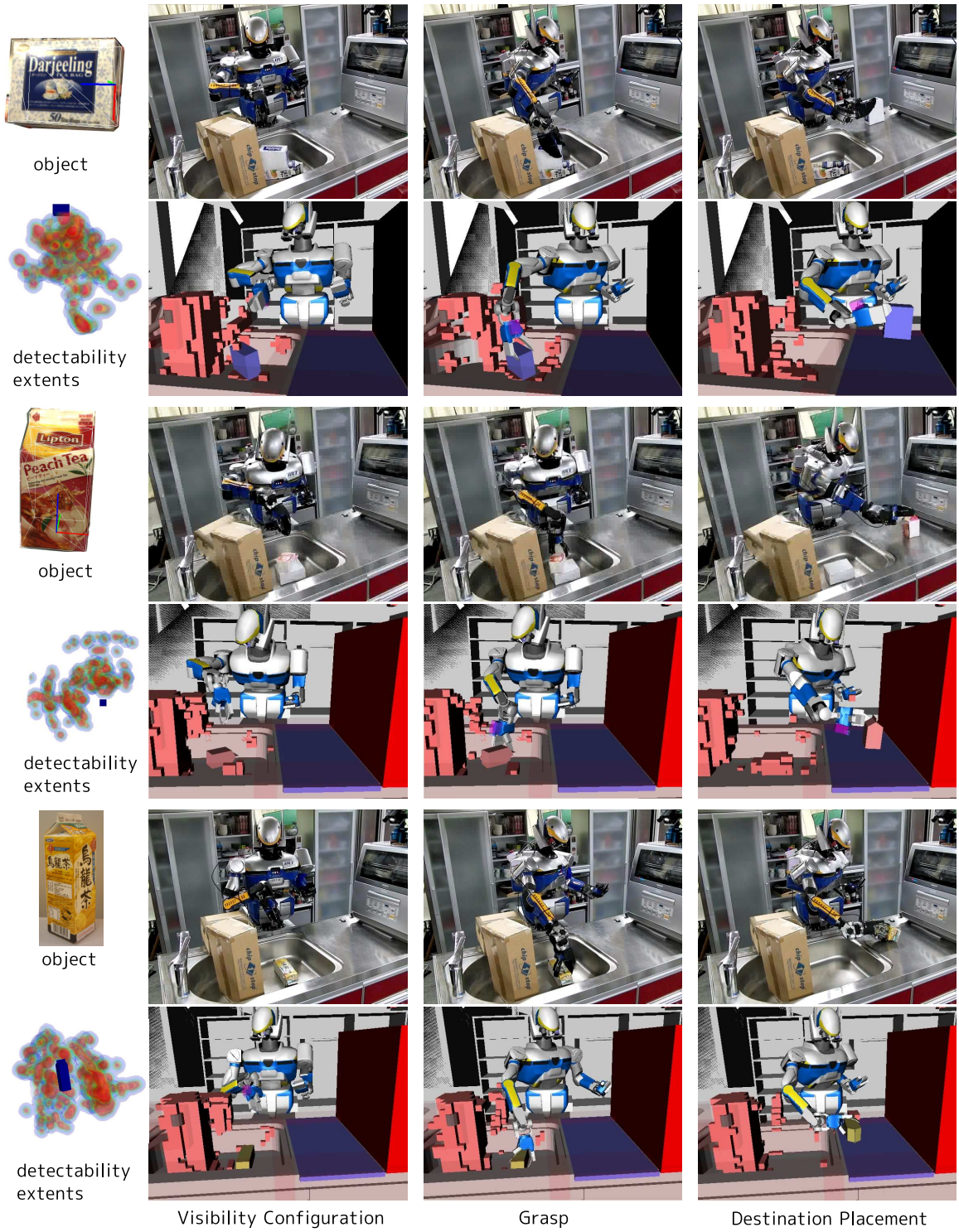


Figure 5.11: Manipulation with dynamic obstacles from range data (red boxes) using visibility configurations.

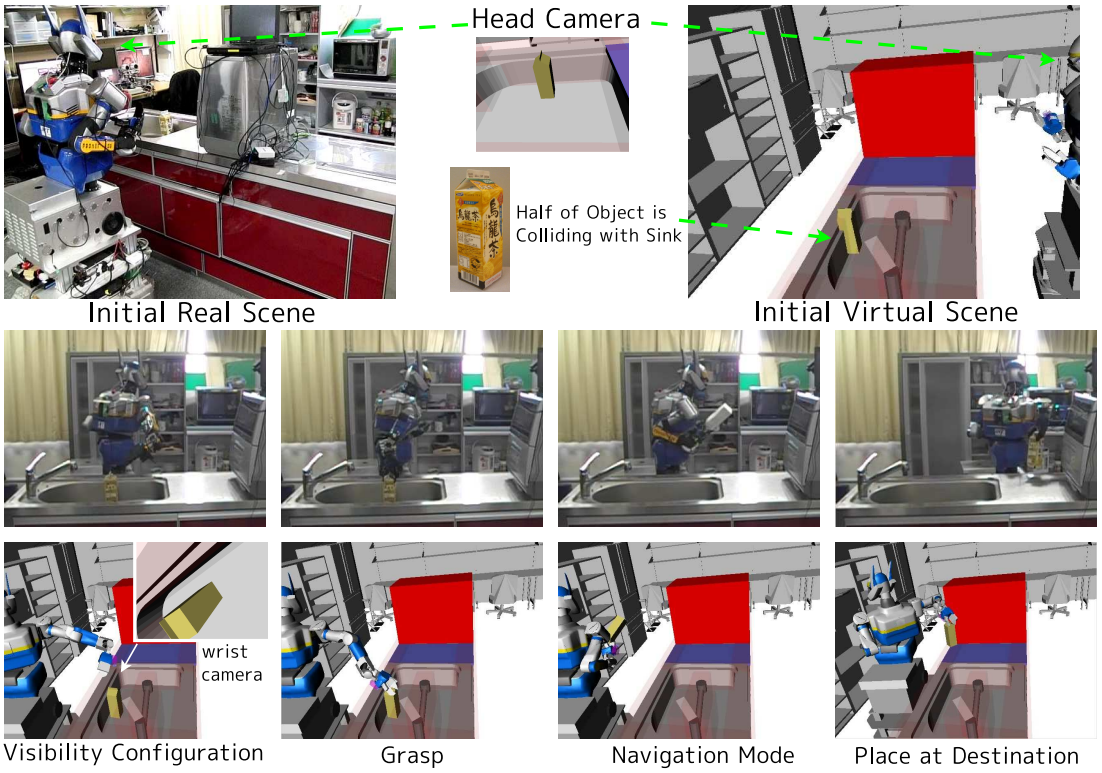


Figure 5.12: The full process of mobile manipulation using visibility, reachability, and base placement models. In order for the robot to grasp the object, it has to move to the other side. But at the other side, the only way to see object is with its wrist camera.

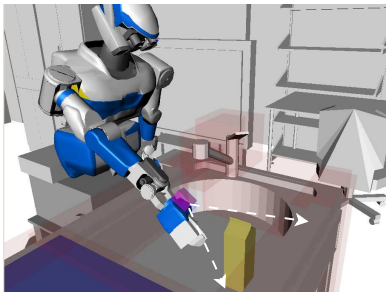


Figure 5.13: When sampling base placements with visibility, the reachability of grasp sets also have to be considered, otherwise the robot will need to move again before it can grasp the object.

can still be considered. Using the reachability multiplication trick in Figure 5.9, we can start sampling base placements the prioritize visibility, but also consider the grasp set. Of the few experiments performed with mobile manipulation, all them succeeded in grasping the object on the first time.

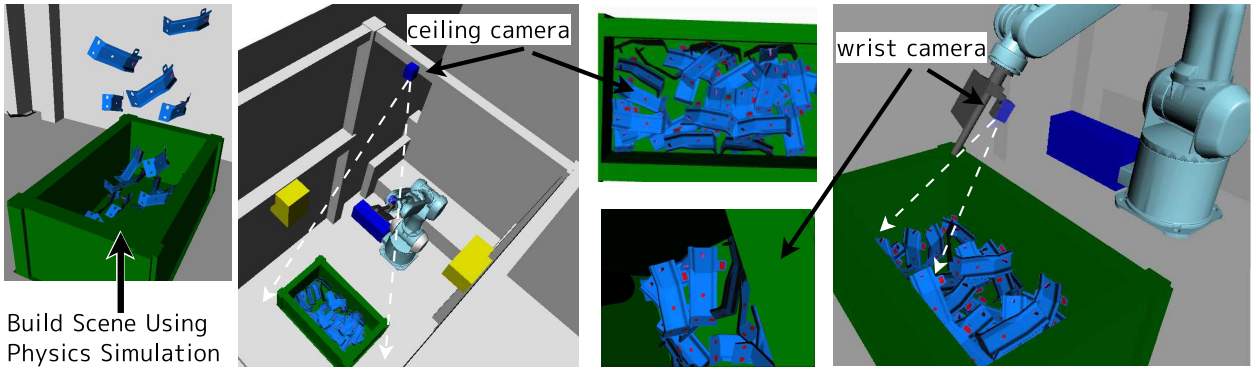


Figure 5.14: Industrial bin-picking scene has a ceiling camera looking at a bin of parts and a gripper camera.

5.5 Industrial Bin-Picking Experiments

One of the most applicable areas of this work is bin-picking of industrial parts scattered in a bin. Pick-up accuracy has to be high because a robot is commonly needs to fit the part into a designated region for assembly into bigger devices. We test this framework in simulation for the two-camera system shown in Figure 5.14. There is a ceiling camera that looks directly into the bin and a gripper camera attached with a pole used inserting into the holes of the parts. One of the unique characteristics of the bin-picking problem is that the robot does not know the entire state of the bin, it only knows the target objects the ceiling camera managed to detect. Given that the ceiling camera can be 2 meters from the parts, the depth of the detected parts could be very uncertain. The uncertainty is reflected in the detectability extents of the part by confirming that the part can still be detected across a

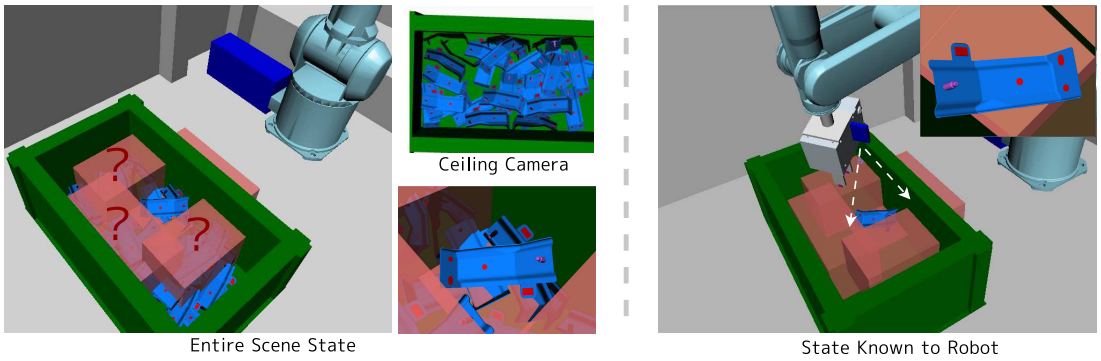


Figure 5.15: Collision obstacles around target part need to be created for modeling unknown regions because the robot does not have the full state of the world.

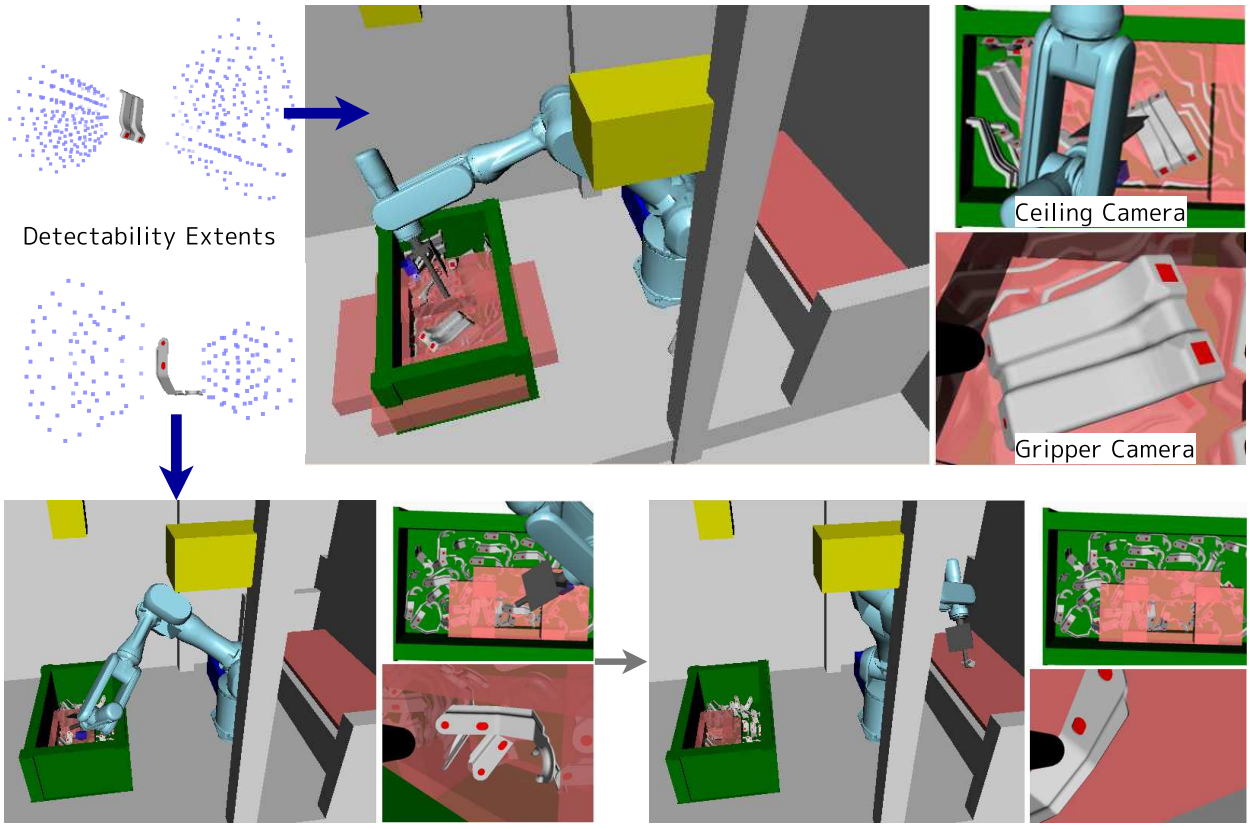


Figure 5.16: Grasp Planning is tested by simulating the unknown regions.

cylinder pointing towards the ceiling camera.

In order to plan in a scene where the immediate surroundings of the target object are unknown, we have to gather a model of the likely distribution of obstacles around a part. Fortunately, we have the CAD model of the part, so we can use physics to simulate parts scattered in a bin (Figure 5.14). By collecting statistics on how the parts are scattered, we can compute statistics on the obstacles around the parts. We parameterize the obstacles by the expected empty space around the part. Figure 5.15 shows an example when the empty region is parameterized by a box: which is the height and spread offsets from the part location. Once a part is detected, the robot creates the obstacles and starts sampling visibility configurations and planning grasps (Figure 5.15 right). We performed experiments in simulation using a variety of parts as shown in Figure 5.16. Each part had a particular region where the manipulator needed to insert itself. The detectability extents are tuned for prioritizing flat surfaces with many features. Using these grasps and detectability extents, The robot is able to pick up all parts from a bin and place them in their respective destinations. Once a

part on the top is chosen, it takes on the order of seconds for the robot to find a visibility configuration and a feasible grasp.

By using simulation, we can guarantee that the geometric complexities of a task are solvable with the particular environment setup. We can use statistics to build up predictive models of the arrangements of other parts so that we can make more informed decisions about the obstacles the robot cannot model. Such information is invaluable for automatic design of industrial workspaces.

5.6 Discussion

In this chapter we introduced a planning framework based on the visibility of the target objects. We showed how to efficiently apply the object's detection extents to sample visibility configurations that allow the robot to accurately recompute a target's pose. By explicitly considering the visible regions of objects, we can compute an upper-bound on the object pose error, and therefore greatly reduce the uncertainty the grasping module has to deal with. Furthermore, planning results suggest that inserting visibility configurations into the planning loop does not increase computation times by much. This discovery suggestions that the visibility configuration space acts as a keyhole configuration by divide and conquering the free space of the robot. The visibility sampler we presented relies on the data-driven *detectability extents* of the target object and vision algorithm pair. Using it allows us to sample camera locations that are guaranteed to observe the object, even under pose uncertainties as large as 4cm. Instead of relying on parallel computation and exotic hardware to speed up the process, the sampler carefully arranges the criteria that must be tested and uses ray testing as much as possible.

Once a visible configuration has been reached, we resume grasping either using the grasp planning pipeline or a visual feedback approach. In order to make visual feedback robust to large changes in the object pose as error is reduced, we presented an extension to a stochastic gradient descent algorithm that can reason about grasps while approaching the object. The gradient descent method is fast enough to be used as a visual feedback algorithm in case the scene is prone to frequent changes.

Throughout the entire chapter, we have discussed the advantages and usage of cameras attached to the gripper of the robot. An interesting analogy can be made to human manipulation to prove the necessity of sensors attached to a gripper. In human manipulation, eyesight serves to provide context and a rough perception of the scene, objects, and their locations. However, the final steps in grabbing objects by humans mostly rely on tactile sensors from the hands rather than eyesight. A human can manipulate as dexterously in the

dark as in light; but as soon as the sense of touch goes away, the manipulation strategies are significantly slowed down. Unfortunately, tactile sensing for robots is still far from becoming a mainstream commodity used in research labs. That tactile sensors need contact with the environment to make measurements, which makes then prone to wear-and-tear damage and not feasible for industrial scenarios. Until tactile sensors become reliable and cheap, we argue that gripper cameras will remain for a long time the best and most affordable alternative to completing a fast loop with the target object.

A gripper camera configuration can provide enormously accurate information because the camera/gripper can fit into tight spaces that base- or head-mounted cameras cannot observe. Furthermore, if the camera is attached to the same frame of reference that all grasps are defined in, we can greatly eliminate any kinematic errors due to absolute joint encoder errors. We showed results of a humanoid robot using a gripper camera to manipulation objects inside a sink, which has low-visibility. Even though the humanoid cameras were not accurately calibrated, it allowed the robot to consistently pick up objects without much failures. This suggests that visibility algorithms can reduce the required calibration on robots. This idea is especially applicable in industrial scenarios where robots perform their tasks without stopping for long periods of time. By relaxing the number of times calibration needs to be done, we can assure that robots can be operational longer without maintenance.

Chapter 6

Automated Camera Calibration

One important problem in autonomous manipulation is synchronizing the robot's internal representation of the world with the real world by *calibrating* its sensors. Calibration is the problem of computing the correct sensor models and so information can be aggregated into an accurate world representation. In this chapter we present an extension to the manipulation framework presented in this thesis that allow us to automate the intrinsic and extrinsic camera calibration processes. Complete automation implies that the system has to measure the quality of the calibration results, which requires finding the correct error measurements. We present initial results at the end of this chapter focusing on the validation and measurement of the calibration results.

The calibration quality encodes the quality of the sensor, the quality of the estimated acquisition model, and the quality of the sensor's localization within the environment. The former is dependent on the manufacturing process and design of the sensor. The latter two qualities representing the intrinsic and extrinsic sensor models estimate geometrical parameters about the sensor. In this section, we examine the problem of calibrating the intrinsic and extrinsic parameters of a camera in context of having a robot in the environment. A plethora of algorithms have been presented for intrinsic camera calibration [Hartley and Zisserman (2000); Datta et al (2009)] both involving known patterns whose measuring properties are stable, and algorithms that directly use natural features of the image without requiring users to engineer their environment. In particular, recent advances in [Datta et al (2009)] have showed how to maintain numerical stability even if the number of training images have increased to a point where image noise starts to become a huge factor.

Although many software packages like OpenCV [Bradski and Kaehler (2008)] and the Camera Calibration Toolbox [Bouguet (2002)] have succeeded in automating intrinsic camera calibration, truly automatic extrinsic camera calibration with respect to a robot frame of

reference tends to be ignored; there are no known reliable calibration packages that solve the automation complexities necessary for it. Although many formulations for extrinsic calibration exist, its data gathering phase usually requires a user to hold a pattern in several locations in front of the camera, along with providing a set of measurements revealing the relationship between the camera frame and pattern frame of reference. Because a user is responsible for providing the data in previously proposed systems, the system just has to worry about finding the most likely extrinsic model that fits the data, it does not concern itself with scoring the data itself. It is the data gathering phase that makes camera calibration painful for users; therefore, such systems are not truly automated.

6.1 Problem Statement

An ideal automated calibration system should be able to use the robot and task specifications discussed in Section 2.1.2 to calibrate itself without any human intervention. The goal is a **one-click calibration** system that works for any environment the robot is in. Industrial robots have to be re-calibrated all the time, and placing special restrictions on the environment for calibration can make it a time-consuming processes that impacts the cost of the produced item. Therefore, the ideal calibration process should not require the environment to be restructured before starting the process.

Building up on previous results in camera calibration, we primarily focus on extrinsic camera calibration and automating the data gathering process. Because a robot is present in the scene, it can provide a way to move the pattern with respect to the camera frame of reference, or vice versa. The system should quickly gather an informed set of measurements of the pattern that allow an accurate computation of the camera intrinsic parameters and its location on the robot. Because training data is gathered automatically, we need a confidence measure on how well it can constrain all the parameters of the process. We show how the previous approaches to evaluating the confidence of calibration results really do not hold any information about evaluating the data.

We state the assumptions under which the presented system works in:

- The CAD model and kinematics of the robot are known.
- The robot kinematics and joint encoders are calibrated.
- For added safety, the robot should know the obstacles in its workspace.
- A known calibration pattern is used, and its detectability extents have been measured (Section 4.7).
- An initial estimate of the camera location and the robot link it is attached to. The estimate can be directly estimated from the CAD model.

- The initial condition is that the calibration pattern be visible inside the camera image. The pattern itself could be placed anywhere.

The unknowns that have to be solved through the calibration process are:

- The intrinsic camera parameters are unknown.
- Calibration pattern position is unknown.

We make no restrictions on the frame of reference the camera is placed in as long as an inverse kinematics solver exists to go from workspace to configuration space. In order to show the generalizability of the methods, we analyze two scenarios:

- **Calibrating a gripper camera.** The goal is to compute the relative camera transformation with respect to the link the camera is attached to. The pattern is stationary in the environment. An inverse kinematics solver of the link to the robot base is required.
- **Calibrating an environment camera.** The goal is to compute the relative camera transformation with respect to the robot base. The pattern is firmly attached to or grasped by a robot link, which needs an inverse kinematics solver.

Compared to other systems, one new feature of this system is usage of the visibility samplers covered in Section 5.1 to determine the best views of the pattern with respect to the camera coordinate system. It allows us to compute safe configurations for the robot to move to get the desired measurements. It should be noted that this does not necessarily guarantee the pattern will be visible in all sampled views due to the unknowns in the system and the progressive nature they will be computed in. The system should continue gathering data until the calibration estimates are robust enough.

6.2 Problem Formulation

We begin by defining all the frames used in the calibration process as shown in Figure 6.1:

- T_{camera}^{link} - the relative transformation between the camera sensor and the robot link. We denote the initial estimate from the CAD model by $\tilde{T}_{camera}^{link}$.
- $T_{pattern}^{world}$ - the transformation of the pattern in the world coordinate system. This is unknown.
- $T_{link}^{world}(q)$ - the transformation of the link the camera sensor is attached to with respect to the world coordinate system. q is the configuration of the robot.

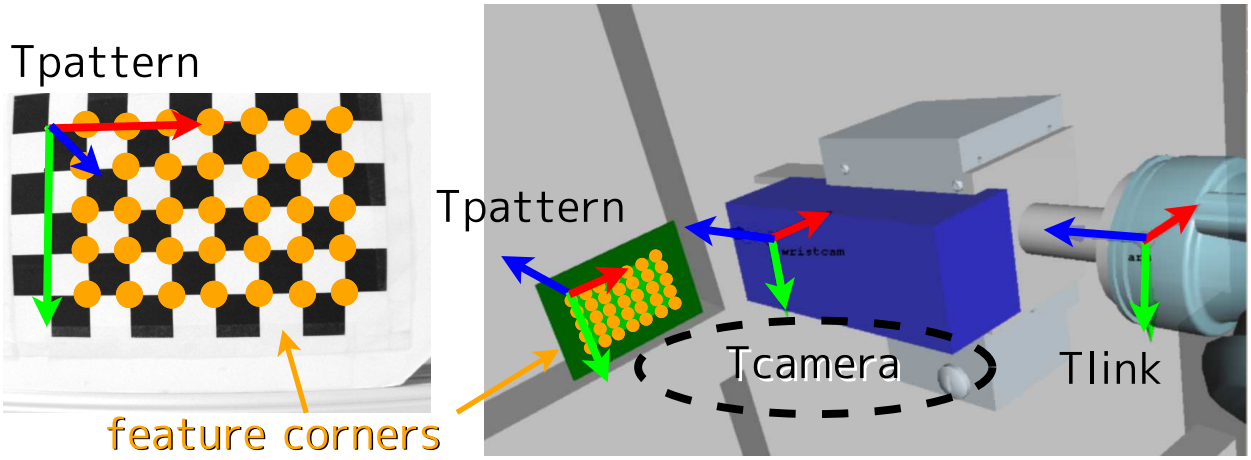


Figure 6.1: Frames used in hand-eye camera calibration.

- $T^{camera}_{pattern}$ - the transformation of the pattern in the camera coordinate system.

The following relationship holds between these frames:

$$T^{world}_{link} T^{link}_{camera} T^{camera}_{pattern} = T^{world}_{pattern} \tag{6.1}$$

For each measurement i, we compute a new robot transformation $T^{world}_{link}(q_i)$ and pattern transformation $T^{camera}_{pattern}(i)$. A pattern consists of a set of N 3D points $\{X_j\}$ in its coordinate system. An image processing algorithm measures the same feature points in the image and corresponds them with the pattern's feature points, which produces a 2D array of image point correspondences $\{\bar{x}^i_j\}$ for the i^{th} measurement. We define a function $x^i_j = f_K(\bar{x}^i_j)$ that transforms an image point to the $z = 1$ plane in the camera space. In practice, we use a nonlinear optimization over the reprojection error as defined in [Bradski and Kaehler (2008); Hartley and Zisserman (2000)].

Combining all information produces the following relation:

$$\forall_i \quad e_i(\omega) = proj((T^{link}_{camera})^{-1}(T^{world}_{link}(q_i))^{-1} T^{world}_{pattern} X_j) - f_K(\bar{x}^i_j) = 0 \tag{6.2}$$

where ω is a parameterization of the calibration parameters. The goal is to produce the best set of robot positions $T^{world}_{link}(i)$ such that the equations built up from these constraints are not degenerate and can compute the correct solution with *high confidence*. The calibration is performed in two steps: first compute the intrinsic parameters using conventional techniques for function f_K, then use the Levenberg-Marquardt algorithm to optimize for the reprojection error $\sum_i |e_i(\omega)|^2$ for these 12 degrees of freedom: $(T^{link}_{camera})^{-1}$ and $T^{world}_{pattern}$. In order to compute

more meaningful gradients, it is common practice to represent rotations with the angle-axis parameterization.

We stress the difference between the minimization error function due to calibration parameters and image noise, and the errors that are due to degeneracies of selecting a bad set of images to calibrate from. Not concerning ourselves with time, we can assume a global minimum solver of the error function. Furthermore, image noise is a quantity intrinsic to the sensor and cannot be directly eliminated. Given these considerations, the only variable we have control over with the robot is the data views. Surprisingly, view selection has not been studied as much in the literature, most probably because a person is usually responsible for picking the best set of views. Because using a robot to automatically search for views does not have the same guarantees, we go over several parameterizations of the calibration equations in Section 6.4 that can be used for computing a confidence for the data.

6.3 Process Outline

We start with describing gripper camera calibration process. Figure 6.2 shows the general outline of the automated data gathering. The user is required to place the calibration pattern in front of the camera. At this step there are no estimated intrinsic parameters of the camera, therefore we cannot compute any geometrically meaningful relationships between the pattern and camera. However, using the fact that the pattern is clearly visible in the first measurement, we parameterize the camera locations. Assuming the pattern is visible in the camera, a small set of neighboring robot configurations that allow the pattern to be viewed from several different orientations are computed using a cone stretching in the negative z-direction of the current camera pose (Figure 6.2 Step 1):

$$
\begin{aligned}
p \in \{ \begin{bmatrix} p_x \\ p_y \\ p_z \end{bmatrix} \in \mathbb{R}^3 \mid \cos \frac{-p_z}{\sqrt{p_x^2 + p_y^2 + p_z^2}} \leq \theta_{cone}, \quad p_z > L_{maxdist} \} \\
d \in \{ \begin{bmatrix} d_x \\ d_y \\ d_z \end{bmatrix} \in S^2 \mid \cos d_z \leq \theta_{dir}, \quad d_x^2 + d_y^2 + d_z^2 = 1 \}
\end{aligned}
\tag{6.3}
$$

where θ_{cone} defines the half-angle of the cone, $L_{maxdist}$ is the maximum length of the cone, θ_{dir} is the maximum angle the camera direction can deviate. The positions and directions are uniformly sampled on $\mathbb{R}^3$ and S^2 (Section 4.3.1). For every position and direction pair,

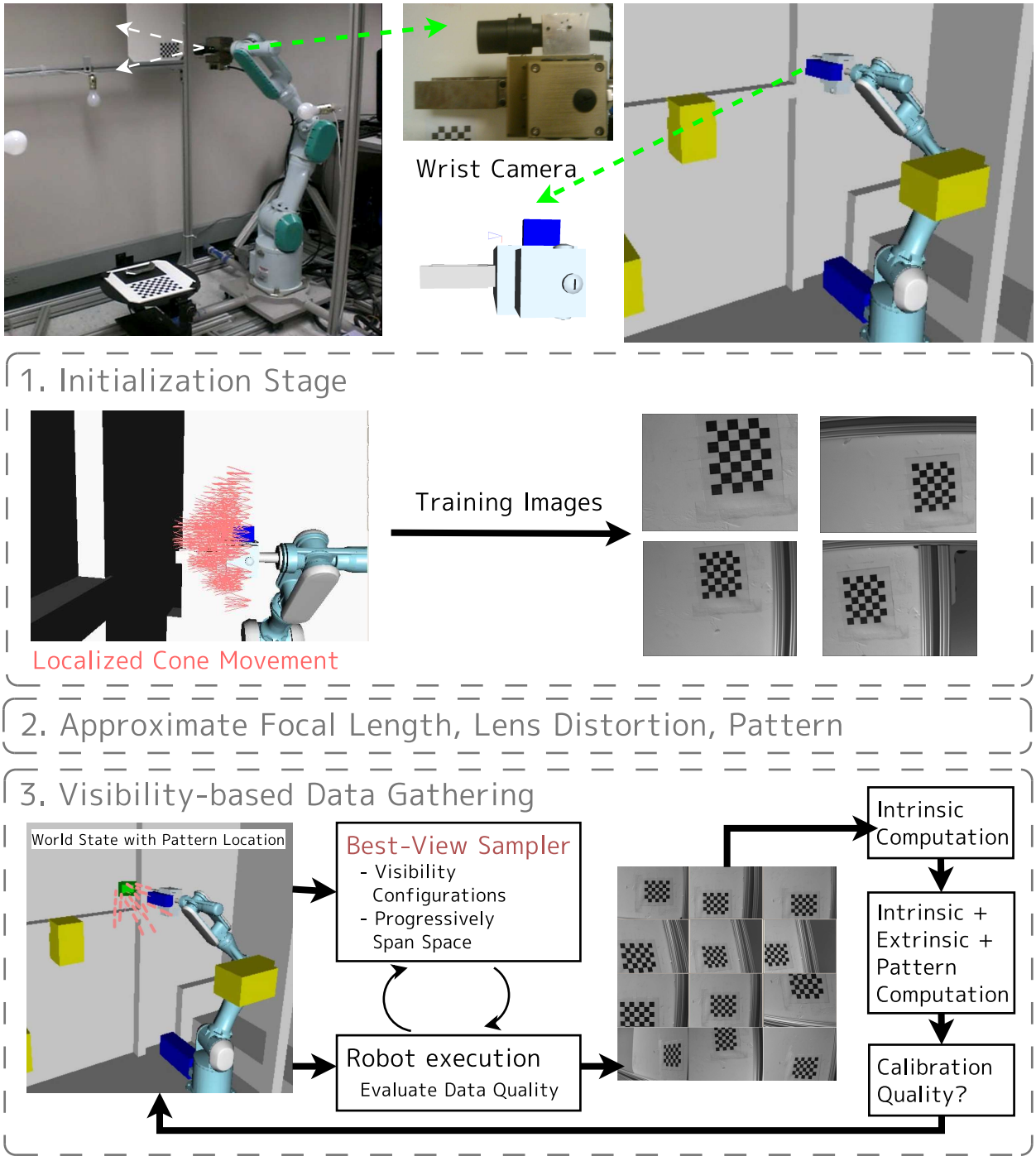


Figure 6.2: Describes the automated process for calibration. Assuming the pattern is initially visible in the camera image, the robot first gathers a small initial set of images inside a cone around the camera (green) to estimate the pattern's location. Then the robot uses visibility to gather a bigger set training images used in the final optimization process.

checkerboard grid	f_x	f_x fixed principal	standard deviation ratio
9x8	2076 ± 197	1992 ± 152	1.296
8x7	2056 ± 23	2080 ± 17	1.353
7x6	2093 ± 19	2085 ± 16	1.188
6x5	2094 ± 51	2116 ± 23	2.217
5x4	2073 ± 33	2082 ± 22	1.500
4x3	2064 ± 118	2053 ± 31	3.806

Table 6.1: Initial intrinsic calibration statics showing the mean and standard deviation of calibration results using 5 images. Note how the standard deviation is much smaller when the principal is fixed to the center of the image.

we find the desired pattern transformation with respect to the original camera coordinate system:

$$T_{camera}(p,d) = \begin{bmatrix} Rodrigues\left(\begin{bmatrix} d_y \\ -d_x \\ 0 \end{bmatrix}, \tan^{-1}\frac{\sqrt{d_x^2+d_y^2}}{d_z}\right) & p \\ 0 & 1 \end{bmatrix}. \tag{6.4}$$

Using Equation 6.4, we can build up the camera samples $\mathcal{T}_{patterns}$ and solve for the initial intrinsic parameters of the camera. Once $5-8$ images are gathered, the focal length and radial distortion of the camera are estimated while setting the principal point to the center of the image. The reason for this reduction in parameters is because the initial set of images are not enough to create strong constraints for all parameters. In practice, we estimate with the principal point, the focal length can be an order of magnitude off from the real length, this makes the later processes impossible to converge. Table 6.1 shows a comparison with fixing the principal point when calibrating the gripper camera as shown in Figure 6.2. As can be clearly seen, the standard deviation on the error is much lower when the principal point is fixed, so the results can be trusted more. Using the initial intrinsic calibration and the rough initial guess of the camera location, we can estimate the world pattern, this allows us to start the visibility-based data gathering that can assure we capture a good set of images that constraint all parameters well.

Ideally, we want to work with an ordered set of poses $\mathcal{T}_{patterns}$ that represent a progressive sampling of views for the camera such that the first N views of the set represent the *best* N views. We define best views in terms of how well the data can constrain the calibration

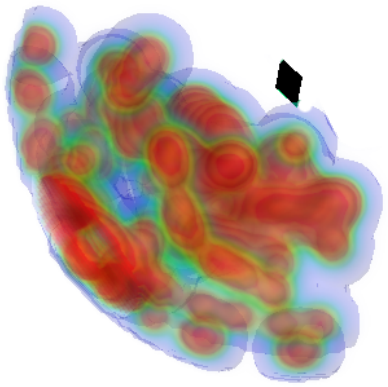


Figure 6.3: The detectability extents of the checkerboard pattern (show in black).

parameters in the optimization process. It has been shown views differing by a roll around the camera axis or distance from the object do not contribute many new constraints. The most interesting constraints come from out-of-plane rotations in S^2, with the next most interesting being large offsets in $\mathbb{R}^2$ on the camera plane that help in measuring radial distortion. Therefore, we can progressively sample $S^2 \otimes \mathbb{R}^2$ to get the ordered set. Furthermore, we want some guarantees that the camera can actually detect the calibration pattern at the sampled locations. Therefore, we build up detectability extents of the pattern (Figure 6.3) and intersect it with the samples on $S^2 \otimes \mathbb{R}^2$. The extents provide the maximum angle of incidence to the calibration pattern plane and minimum and maximum distance from pattern. Because the robot is going through a calibration phase and does not have an accurate position of the pattern or the camera, views on the edge of the detectability extents have a very high probability of being missed; therefore, positions in the most densest regions should be prioritized first. It is possible to force such an ordering on S^2 according to [Gorski et al (2005)]. Even with taking into account position uncertainty errors and detectability, a particular view might be blocked by the environment, so the actual sampling process requires an algorithm to keep track of what is detected and another algorithm to determine when the data gathering process should terminate.

The basic outline of data gathering is presented in GatherCalibrationData (Algorithm 6.1) where the input is a set of possible desired pattern views $\mathcal{T}_{patterns}$ in the camera space. For every pattern, $T_p \in \mathcal{T}_{patterns}$, the world camera position is $T_{link}^{world}(q_{cur})\ \tilde{T}_{camera}^{link}\ T_p$ where q_{cur} is the current configuration of the robot. Because the inverse kinematics equations work with the desired manipulator link, we right-multiply the camera transformations by $(\tilde{T}_{camera}^{link})^{-1}$ to convert to the link space. After all the desired robot link transformations are computed in $\mathcal{T}_{links}$, we start sampling from the set and taking measurements. In ev-

Algorithm 6.1: $CalibrationSamples \leftarrow$ GatherCalibrationData($\mathcal{T}_{patterns}$)

$CalibrationSamples \leftarrow \emptyset$
$\mathcal{T}_{links} \leftarrow \left\{ T_{link}^{world}(q_{cur})\ \tilde{T}_{camera}^{link}\ T_p\ (\tilde{T}_{camera}^{link})^{-1} \quad | \quad T_p \in \mathcal{T}_{patterns} \right\}$
while $\mathcal{T}_{links} \neq \emptyset$ **do**
    $\bar{T}_{link}^{world} \leftarrow$ NextBestView($\mathcal{T}_{links}$)
    $q_{target} \leftarrow \text{IK}(\bar{T}_{link}^{world})$
    if $q_{target} \neq \emptyset$ **then**
        if PlanToConfiguration(q_{target}) **then**
            if IsPatternDetected() **then**
                $CalibrationSamples \leftarrow CalibrationSamples + \left\{ T_{link}^{world}(q_{cur}), \{\bar{x}_j^i\} \right\}$
                /* Prune similar transformations */
                $\mathcal{T}_{links} \leftarrow \{ T \in \mathcal{T}_{links} \quad | \quad \delta(T, T_{link}^{world}(q_{cur})) > \tau \}$
                if CheckCalibrationValidity($CalibrationSamples$) **then**
                    return $CalibrationSamples$
                else
                    continue
    $\mathcal{T}_{links} \leftarrow \mathcal{T}_{links} - \{\bar{T}_{link}^{world}\}$
end
return $\emptyset$

ery iteration, the next best view $\bar{T}_{link}^{world}$ is chosen and inverse kinematics are applied with collision-checking to yield a target configuration q_{target}. The robot attempts to plan a path and move to q_{target}. If successful and the pattern is detected at that configuration, then all similar views from $\mathcal{T}_{links}$ are removed and the robot position $T_{link}^{world}(q_{cur})$ and the pattern feature points $\{\bar{x}_j^i\}$ are stored. Note that the pattern transformation $T_{pattern}^{camera}$ is not needed to compute the reprojection error as defined in Equation 6.2, so is not stored. If any step fails, then only the currently tested link $\bar{T}_{link}^{world}$ is removed from $\mathcal{T}_{links}$. This process is continued until the validation function CheckCalibrationValidity decides the data is enough. If all samples are tested before valid data is found then the function fails.

Using the initial intrinsic parameters, the initial estimate of the world transformation of the pattern is given by

$$\tilde{T}_{pattern}^{world} = (T_{link}^{world} \tilde{T}_{camera}^{link})^{-1} T_{pattern}^{camera} \tag{6.5}$$

where $T_{pattern}^{camera}$ can be easily solved using classic 2D/3D point correspondence techniques [Hartley and Zisserman (2000); David et al (2004)]. Once we have an initial estimate of the

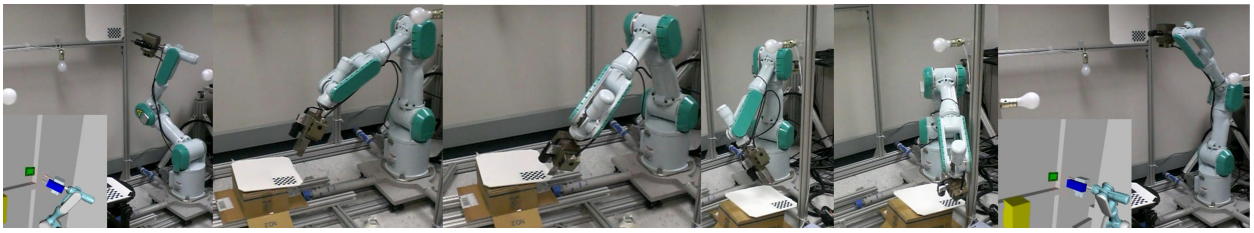


Figure 6.4: Several configurations of the robot moving to a visible pattern location.

pattern in the world (Figure 6.2 right), we use the visibility theory developed in Section 5.1 to sample a feasible and informed set of camera transformations around the pattern using Equation 5.1. Figure 6.4 shows several examples of the robot using the visibility regions of a checkerboard pattern to calibrate itself. In order to evaluate results, we intersect all the rays from the image points together to find the 3D world point agreeing with all images:

$$\begin{aligned} \forall_i x_j^i &= proj(R^i \tilde{X}_j + t^i) \\ \Rightarrow \forall_i (P_2^i \cdot \tilde{X}_j + t_2^i) x_{j,0}^i &= P_0^i \cdot \tilde{X}_j + t_0^i, \quad (P_2^i \cdot \tilde{X}_j + t_2^i) x_{j,1}^i = P_1^i \cdot \tilde{X}_j + t_1^i \\ \Rightarrow (P_2^i x_{j,0}^i - P_0^i) \cdot \tilde{X}_j + t_2^i x_{j,0}^i - t_0^i = 0, &\quad (P_2^i x_{j,1}^i - P_1^i) \cdot \tilde{X}_j + t_2^i x_{j,1}^i - t_1^i = 0 \end{aligned}$$

where the projection matrix $\begin{bmatrix} R^i & t^i \\ 0 & 1 \end{bmatrix} = (T_{link}^{world} T_{camera}^{link})^{-1}$. Because $T_{pattern}^{world}$ is also computed from the optimization process, we compute the 3D error by

$$\frac{1}{N} \sum_j |T_{pattern}^{world} X_j - \tilde{X}_j| \tag{6.6}$$

Table 6.2 shows the reprojection and 3D errors associated with calibrating the gripper camera using several checkerboard patterns. The reprojection errors are usually an order of magnitude larger than the intrinsic errors because we are constraining to only one world pattern whose transformation is determined through the robot's kinematics. The added error between the robot and the reduction of the degrees of freedom contribute to large reprojection errors, which is also a sign that intrinsic error is over-fitting to its data. Another interesting result is that the smaller checkerboards yield much better performance in terms of intrinsic and 3D errors. 3D error is not used in the optimization process because the gradients on the error are not continuous, and take longer to compute.

checkerboard	square length (m)	intrinsic error	reprojection error	3D error (m)
9x8	0.0062	0.448 ± 0.106	13.44 ± 6.48	0.00052 ± 0.00031
8x7	0.0069	0.236 ± 0.113	06.84 ± 3.02	0.00030 ± 0.00009
7x6	0.0077	0.249 ± 0.092	16.47 ± 8.66	0.00026 ± 0.00009
6x5	0.0088	0.313 ± 0.159	11.27 ± 8.34	0.00024 ± 0.00010
5x4	0.0103	0.217 ± 0.080	15.43 ± 8.76	0.00025 ± 0.00007
4x3	0.0124	0.213 ± 0.089	08.81 ± 4.08	0.00018 ± 0.00009

Table 6.2: Reprojection and 3D errors associated with calibrating the extrinsic parameters of the gripper camera. Both intrinsic error and reprojection errors are in pixel distances, with reprojection error using only one pose for the checkerboard by computing it from the robot's position.

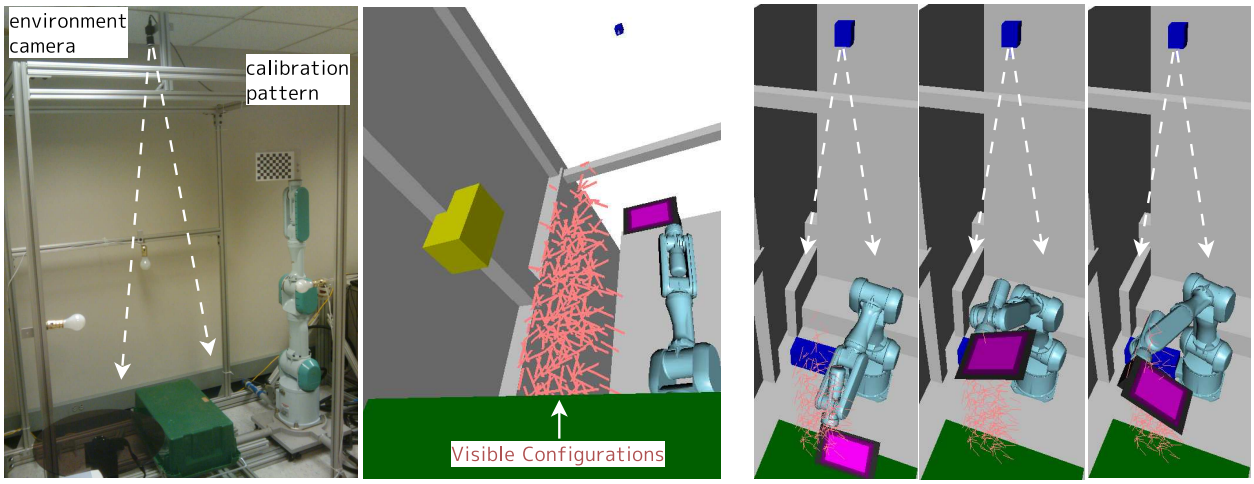


Figure 6.5: Example environment camera calibration environment. The pattern is attached to the robot gripper and robot uses moves it to gather data. Sampled configurations are shown on the right.

6.3.1 Application to an Environment Camera

In the previous section, we generated configurations that ensure that the gripper camera is moved to a visibility region of the pattern. We can always apply the same theory to environment cameras, or cameras that are not attached to the robot links. In order generate data, we assume the pattern is attached to a robot link and an inverse kinematics solver exists. The new coordinate systems relationship is

$$T_{link}^{world} T_{pattern}^{link} T_{camera}^{pattern} = T_{camera}^{world} \tag{6.7}$$

where the transformations to be calibrated are $T^{link}_{pattern}$ and T^{world}_{camera}. The *only* difference with this parameterization is that the camera/pattern relative transformations as computed by Equations 6.4 and 5.1 are inverted when computing $\mathcal{T}_{links}$:

$$\mathcal{T}_{links} \leftarrow \left\{ T^{world}_{link}(q_{cur})\ \tilde{T}^{link}_{pattern}\ T_p^{-1}\ (\tilde{T}^{link}_{pattern})^{-1} \quad | \quad T_p \in \mathcal{T}_{patterns} \right\}. \tag{6.8}$$

where a rough initial value of $\tilde{T}^{link}_{pattern}$ has to be known. Every other step is equivalent to the gripper camera case. Figure 6.5 shows several sampled configurations when the robot is in front of the camera.

6.4 Calibration Quality and Validation

Even with informative best-view sampling, it is not clear when to stop sampling new camera configurations. Even if all samples are exhausted, it is not clear if we can trust the calibration results from the captured data because of the uncertainties in the pattern and camera locations. The total calibration process has 21 parameters: 4 for K, 5 for radial distortion, 6 for world pattern position, 6 for camera position. Therefore, we need evaluation metric for how well the data constraints each of the parameters.

Being able to formulate and solve a set of equations to find the calibration parameters is only the first step towards reliability; it is also necessary to develop methods to test the errors of the calibration data and evaluate how well the data itself constrains all the variables. Even if the global minimum of the reprojection error (Equation 6.2) is computed, it does not give any guarantees on solution quality, especially when considering image spatial discretization and noise due to light quantification. [Bouguet (2002)] uses the standard deviation of the reprojection error $|e(\omega)|$ and their derivative magnitudes with respect to the parameters $|\frac{de(\omega)}{d\omega}|$ to compute a confidence for the final values. Intuitively, this would work for weakly-constrained variables because they do not affect the final error as much and $|\frac{de(\omega)}{d\omega}|$ would be naturally lower, so their uncertainty is bigger. However, because image noise gets in the way, this assumption does not hold as shown in Figure 6.6.

Using the calibration method in [Zhang (2000)], it is possible to solve for the intrinsic calibration matrix ignoring distortion by formulating a set of linear equations. A linear formulation makes the method powerful because a condition number for the equations can be computed. The intuition of the method is to first compute the homography between the image points x^i_j and the local feature points X_j

$$x^i_j = H_i X_j = \begin{bmatrix} h_{i,1} & h_{i,2} & h_{i,3} \end{bmatrix} X_j \tag{6.9}$$

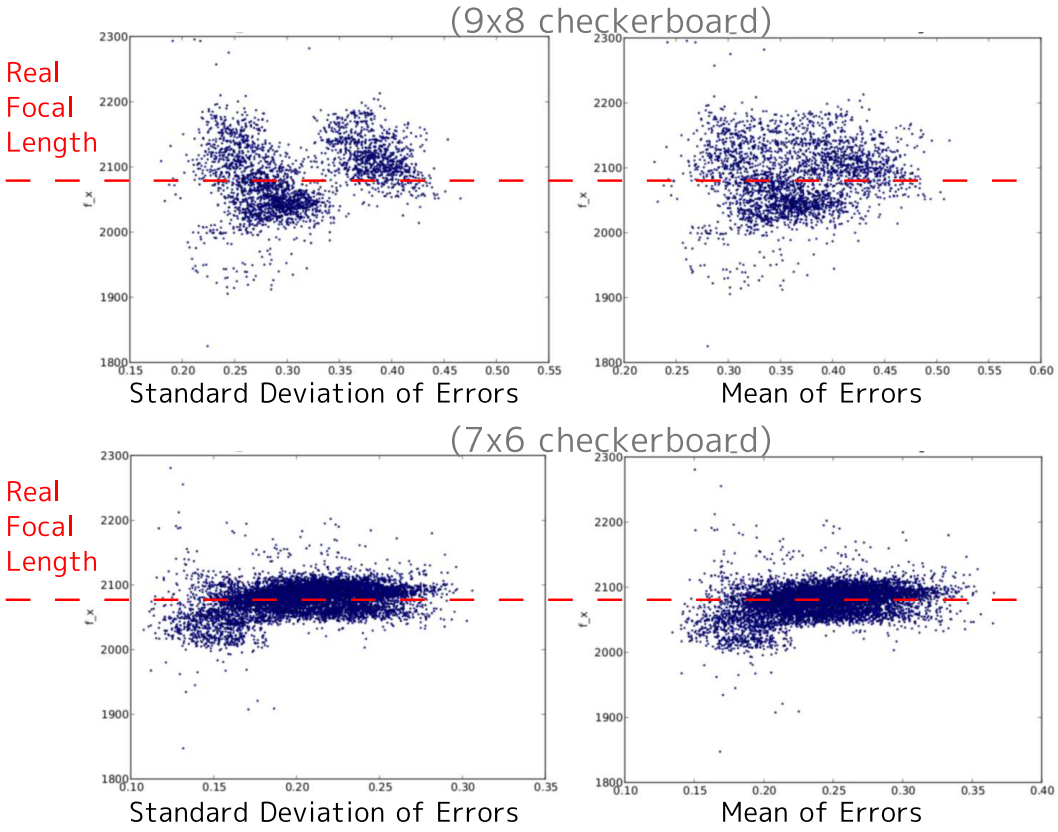


Figure 6.6: A plot of the intrinsic and absolute error (y-axis) vs the gradients of the reprojection error (x-axis). The top plots were generated by taking all combinations of 5 images from a bigger database, the bottom plots were with combinations of 10 images. This shows that gradients do not hold any information about the confidence of result. Left side is projection error used to optimize the intrinsic parameters while right side is the absolute error of f_x.

By assuming that all feature points lie on the $z = 0$ plane, then the homograph can also be represented as:

$$\begin{bmatrix} h_{i,1} & h_{i,2} & h_{i,3} \end{bmatrix} = sK \begin{bmatrix} r_1 & r_2 & t \end{bmatrix} \tag{6.10}$$

where r_1 and r_2 are orthogonal vectors of the column vectors of $T^{camera}_{pattern}$. Using the orthogonality constraint, the following equations can be built

$$h_1^T B h_2 = 0$$
$$h_1^T B h_1 - h_2^T B h_2 = 0$$

where $B = K^{-T}K^{-1}$ is the image of the absolute conic. B has 6 values and every image provides 2 constraints, which can be formulated as a classic $Ab = 0$ problem. Let $\{\lambda_i\}$

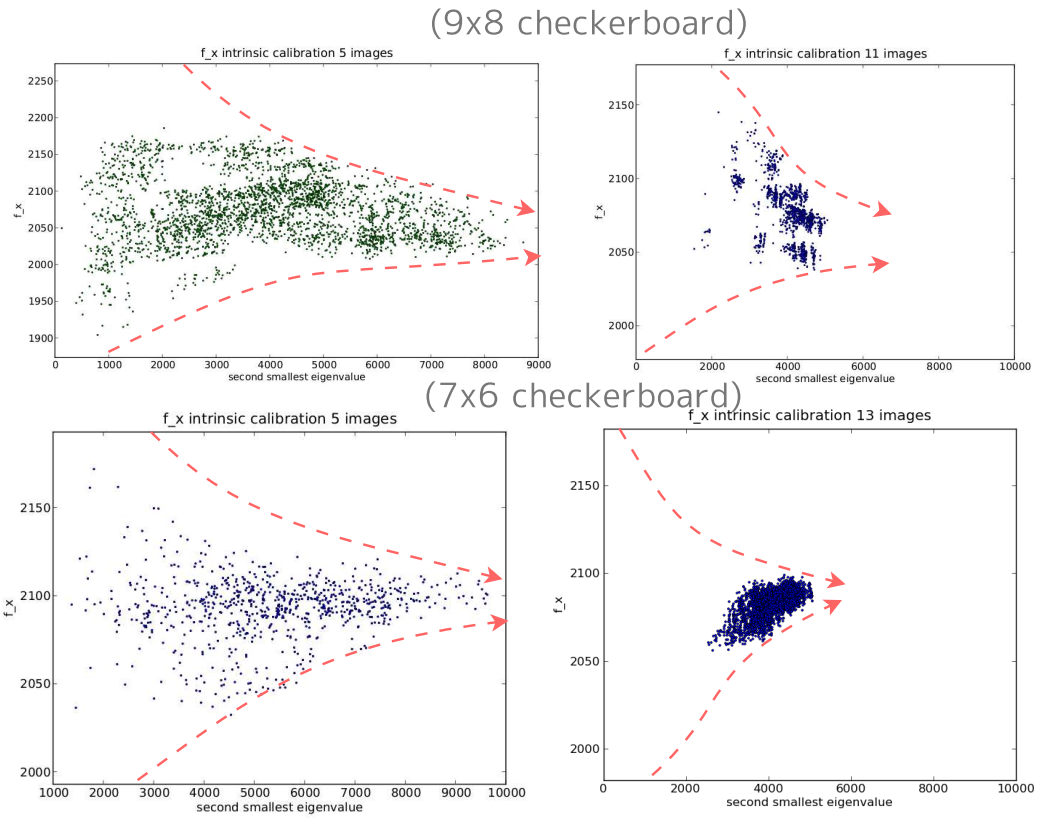


Figure 6.7: Graphs the second smallest eigenvalue of the matrix used to estimate the intrinsic camera parameters. As the eigenvalue increases, the deviation f_x becomes smaller and more stable. The left graphs are done for combinations of 5 images while the right graphs are with more than 11 images. The pattern is more apparent for smaller number of observations. More observations decrease the standard deviation of the solution, but the stability is still dependent on the second eigenvalue.

and $\{b_i\}$ respectively be the eigenvalues and eigen vectors of A such that $\lambda_i \leq \lambda_{i+1}$ and b_i represents the upper triangle of B. Then b_1 is the best null-vector of A and therefore the best solution for B. However ill-conditioned data will lead A to have a null space greater than one dimension, and there so should be more than one eigenvalue λ_i close to zero. Intuitively, this surmounts to checking λ_2 for ill-conditioning effects. Traditionally the ratio $\frac{\lambda_2}{\lambda_1}$ has been used to check for stable null spaces, however image noise and the fact that we are ignoring distortion makes λ_1 fluctuate a lot, thus causing the ratio the yield no information. Figure 6.7 shows how λ_2 varies with respect to the estimated focal length f_x for several checkerboard patterns using different numbers of images. The eigenvalue itself is normalized with respect to the number of equations used to estimate it. An interesting occurring pattern is that

the deviation of f_x from its true value grows smaller as λ_2 increases. The intuition is that the bigger it gets, the more stable the null space of A becomes. An interesting analogy can probably be drawn between the power behind the second smallest eigenvalue λ_2, and the second smallest value used to solve the normalized cuts problem for image segmentation [Shi and Malik (1997)].

This pattern suggests that the second eigenvalue can be used to validate the incoming data. Once B is computed, the signs of all eigenvalues in B have to be the same; otherwise, imaginary solutions would be computed for K, which is not correct. CheckCalibrationValidity (Algorithm 6.2) describes the entire validity process for the calibration data.

Algorithm 6.2: $valid \leftarrow$ CheckCalibrationValidity($CalibrationSamples$)

$A \leftarrow \mathbf{0}_{2|CalibrationSamples| \times 6}$
for $i = 1$ *to* $|CalibrationSamples|$ **do**
    $\begin{bmatrix} h_1 & h_2 & h_3 \end{bmatrix} \leftarrow$ Homography($\{X_j\}$,$\{\bar{x}_j^i\}$)
    A_{2i}=$\begin{bmatrix} h_{11}h_{21} & h_{11}h_{22}+h_{12}h_{21} & h_{12}h_{22} & h_{13}h_{21}+h_{11}h_{23} & h_{13}h_{22}+h_{12}h_{23}+h_{12}h_{23} & h_{13}h_{23} \end{bmatrix}$
    A_{2i+1}=$\begin{bmatrix} h_{11}^2-h_{22}^2 & 2h_{11}h_{12}-2h_{21}h_{22} & h_{12}^2-h_{22}^2 & 2h_{13}h_{31}-2h_{23}h_{21} & 2h_{13}h_{12}-2h_{23}h_{22} & h_{13}^2-h_{23}^2 \end{bmatrix}$
end
$\{b_i\}, \{\lambda_i\} \leftarrow$ OrderedEigenDecomposition($A^T A$)
if $\lambda_2 > \tau_1$ **or** $\lambda_1 > \tau_2$ **then**
    return False
$B \leftarrow \begin{bmatrix} b_{1,1} & b_{1,2} & b_{1,4} \\ b_{1,2} & b_{1,3} & b_{1,5} \\ b_{1,4} & b_{1,5} & b_{1,6} \end{bmatrix}$
if not same Sign(Eigenvalues(B)) **then**
    return False
return True

Another necessary property for the intrinsic calibration algorithm is that it does not lose accuracy as more images are added. Surprisingly, [Datta et al (2009)] performed a set of experiments showing the OpenCV [Bradski and Kaehler (2008)] and Camera Calibration Toolbox [Bouguet (2002)] do not have this property. Instead, [Datta et al (2009)] motivates the usage of ring-based patterns along with an iterative reprojection process.

6.5 Discussion

One prerequisite for true robot autonomy is well-calibrated sensors. Although this basic prerequisite of robotics is known to any researcher in the field, rarely do researchers consider automating the entire calibration phase, including the data gathering phase. Consequently, the most painful part of working with a real robot is maintaining their calibration and managing printing many calibration patterns. In response to this reality, we introduced a completely automated method for calibrating the intrinsic and extrinsic parameters by using the planning and visibility theories developed in Chapters 4 and 5. By focusing on the data-gathering phase and providing an analysis for how to determine the quality of the gathered data. Once the planning and vision knowledge-bases are created, it provides researchers a one-click method to calibrate a robot just by setting a calibration pattern in front of the camera.

The quality of data problem directly relates to how well the calibration pattern can be detected in an image along with the constraints it provides. By taking advantage of the detectability extents of the pattern, we can create a progressive best-view sampler set that the describes where the camera should be placed. Furthermore, by using the visibility samplers, we can simultaneously search for possible locations the robot can move to such that the pattern is unobstructed from view. After data gathering finishes, the pattern position and camera calibration are computed simultaneously. Using the data and calibration quality metrics, we can determine whether to stop, or take re-run the algorithm again with the improved estimates. Furthermore, we showed the flexibility of the method in calibrating an environment camera.

We discovered several surprising results when correlating error metrics to the stability of the calibration solutions they point to. The first discovery was that the standard deviation and reprojection errors really do not hold much valuable information if the data is already degenerate. The algorithms can overfit to the data and the resulting errors will be very small, even though the calibration result is off. The second discovery was the power of the second smallest eigenvalue of a matrix that contained the solution to intrinsic calibration matrix. We showed a very clear correlation between the magnitude of the eigenvalue and the deviation of the focal length from its true value. We can use this test to determine when to stop the calibration sequence, or to continue. Thus, we can provide guarantees on how well the returned solutions model the data.

Chapter 7

Object-Specific Pose Recognition

Recognizing specific rigid objects and extracting their pose from camera images is a fundamental problem for manipulation. Although many pose recognition algorithms exist, researchers usually concentrate on performance of the run-time process and not on the complexity of the work required to automate the building processing of the program for a new object. It is important to differentiate object-specific pose recognition from the general *object recognition* problem. In vision research literature, object recognition also deals with detecting and extracting objects from images that belong to particular classes like faces [Gu and Kanade (2008)], cars [Li et al (2009)], and other common object categories [Griffin et al (2007)]. The challenge is not only to segment out the particular objects that algorithms were trained with, but to find the key features that differentiate one object class from another so that the algorithm can generalize to objects that were never seen. Because training data is sparse compared to the space of all possible objects in a category, object recognition research starts crossing the borders of machine learning, language semantics, unsupervised pattern recognition, and general artificial intelligence. On the other hand, object-specific recognition deals with the analysis and identification of one particular object, which is a well-formed problem that can exercise geometric and photometric analyses that are not so consistent when entire semantic classes of objects are considered.

There are many advantages to concentrating on object-specific pose recognition. In such a formulation, the only variations independent of the object are: lighting and environment changes, object pose changes, and occlusions. Furthermore, it becomes possible to give a training set that covers viewing the object from all directions, which reduces the machine learning complexity of the problem. Because an object-specific program can be specifically tuned to the particular object features, it is possible to speed up the pose extraction process, and increase the detection rate by *over-fitting* the models to the problem. Finally, considering

the low cost of today's memory and computation power, thousands of object-specific models can be easily indexed by a small robot system.

This chapter is divided into two main contributions. The first is creating an object-specific database that holds meaningful statistical analyses of the object's features (Section 7.2). not. Every pose extraction algorithm relies on a common set of statistics, which we organize into a database. In this database, we classify stable and discriminable features while pruning out features due to noise. Stable features are clustered into a handful of representative classes. Using these classes, it is possible to determine whether an arbitrary image feature is part of the object. The second main contribution discussed in Section 7.3 is a new pose extraction algorithm that builds up its pose hypotheses by hallucinating poses given a set of image features, it does not rely on one-to-one feature correspondences making it really powerful for non-textured, metallic object. We term this algorithm *induced-set* pose extraction. The new algorithm uses feature clustering. Along with formulating a new search process, we show how to train an evaluation metric for pose hypothesis. The evaluation metric uses a combination of Adaboost and Markov Chain Monte Carlo methods to train a set of weak learners to only recognize the specific object (Section 7.3.4).

We begin with the basic theories of pose recognition systems and go over previous work. We then introduce the object database and its organizational structure. Finally, we discuss the induced-set pose extraction algorithm and show results in the context of recognizing industrial parts.

7.1 Pose Recognition Algorithms

Most commonly used pose recognition algorithms subscribe to very similar approaches in their offline training and online runtime phases. In the offline model training phase, a set of 3D features that describe the local region of the object surface are learned either through unsupervised learning [Schaffalitzky and Zisserman (2002); Gordon and Lowe (2006); Chia et al (2002)] or through labeled training data. A 3D feature typically contains a view-independent descriptor of a region on the object surface [Lepetit et al (2003); Gordon and Lowe (2006); Morel and G.Yu (2009)], or the less common case of a view-dependent descriptor that also encodes some orientation information. In the runtime phase, correspondences are computed between the extracted 2D image features and the 3D object features. Two features are matched if the distance between them in the feature descriptor space is small. Because the matching process only considers local information about the image, it is prone to making mistakes; hence, algorithms like SoftPOSIT [David et al (2004)], RANSAC [Fischler and Bolles (1981); Chum (2005)], or Least Median of Squares [Zhang et al (1995)] are used to

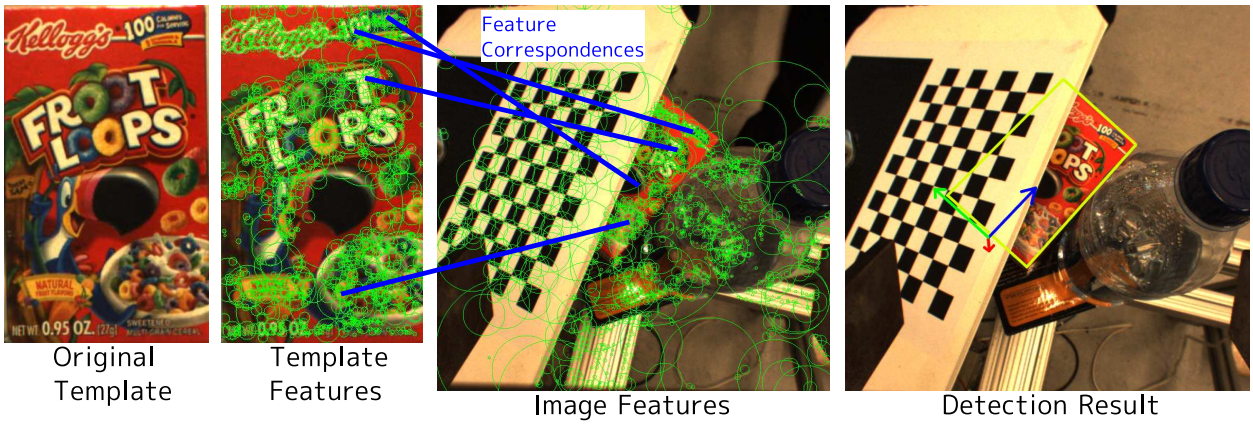


Figure 7.1: A well established pose extraction method using 2D/3D point correspondences to find the best pose. Works really well when the lighting and affine invariant features can be extracted from the template image.

quickly search for a matching that produces the most stable pose (Figure 7.1).

Some pose extraction systems store a set of views labeled with the object pose [Lepetit et al (2003)] instead of explicitly extracting the geometry of the target object; during runtime, epipolar geometry are used between the current image and database images to compute a consistent pose with all constraints. Furthermore, [David and DeMenthon (2005)] has shown how to perform pose recognition when the image features are lines. Because matching lines is more computationally intensive and poses generated by RANSAC are frequently wrong, several heuristics have to be employed to ensure good performance.

A lot of object pose estimation research [Gordon and Lowe (2006); Lepetit et al (2003); Chia et al (2002); David and DeMenthon (2005)] formulates the extraction problem as an iterative algorithm that relies on a previous measure of the pose to seed the search for the current image. At startup time, these algorithms typically rely on simple methods to seed the search, but each method has its own shortcomings that does not have any guarantees on picking the correct object.

Perhaps the set of algorithms closest to our formulation of *induced-set* pose extraction are ones that analyze the object geometry and attempt to come up with Gestalt rules on how to extract a pose estimate from the composition of low level features [Goad (1983); Bolles et al (1983); Grimson and Lozano- Perez (1985)]. Specifically, [Wheeler and Ikeuchi (1992, 1995); Gremban and Ikeuchi (1993, 1994)] have proposed ways to automatically extract features on the object surface and match features to them using discrete search based on aspect graphs. Our method differs from these methods in that we do not formulate the problem with respect to 3D object features and aspect graphs, and we do not search for correspondences between

image features and object surface regions.

7.1.1 Classifying Image Features

We fist explain our notation of the image features and then review popular low-level features that could be used in this framework. There are two categories of feature detectors we use in our framework: point features $\mathcal{F}_0$ describing local regions in the image centered on a point, and curve features $\mathcal{F}_1$ describing a set of connected image points. Each feature has geometric parameters M_{f_d} describing where it is located in the image and visual parameters f_d describing the image at the feature's local neighborhood.

Point Features

For a feature $f_0 \in \mathcal{F}_0$, f_0 is the local descriptor for the image at that region and M_{f_0} is the local coordinate system including position, orientation, and scale. Point feature algorithms are separated into estimating the set of interest points from which the coordinate system of each feature can be computed, and describing the local region around the feature's location. A great summary of interest point detectors can be found in [Mikolajczyk et al (2005)] where the following region detectors proposed in [Mikolajczyk and Schmid (2002, 2004); Tuytelaars and Van Gool (2004); Kadir et al (2004)] are compared with each other.

An image descriptor is typically represented in a vector space with a kernel-based distance metric. This formulation allows a feature to be easily used in machine learning and clustering algorithms. Some of the most popular low-level feature descriptors like Gabor filters are inspired from the sparse-coded nature of the visual cortex [Serre et al (2005)]. These features can encode multiple frequencies and respond to spatial intensity changes, which makes them robust to color and other lighting variations. In order to encode higher level information like texture, [Lowe (2004); Bay et al (2008)] have proposed a scale-invariant set of features. As the name implies the feature's coordinate system encodes the orientation and scale of the region, making it slightly robust to affine transformations. Several new formulations of SIFT [Lowe (2004)] make affine invariance a priority for robust pose extraction [Morel and G.Yu (2009); Wu et al (2008)]. SIFT features represent the local region as an oriented histogram of spatial derivative responses. [Moreels and Perona (2005)] show a survey of the performance of feature descriptors for accurately recovery viewpoint changes.

Line Features

A line feature space $\mathcal{F}_1$ contains all 1D curves from the image with M_{f_1} defining the shape of the curve. There are many algorithms that produce a black and white map detecting

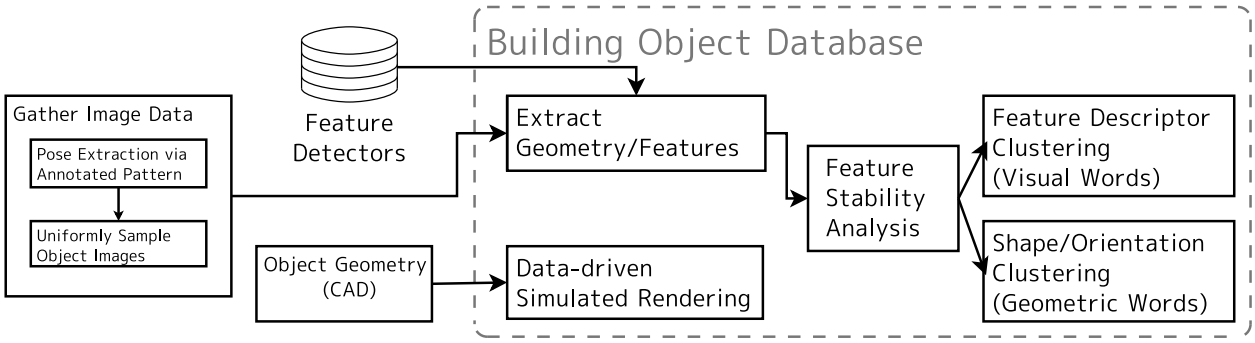


Figure 7.2: Database is built from a set of real-world images, a CAD model, and a set of image feature detectors. It analyzes the features for stability and discriminability.

the edges of the image. By combining all edges into connected components and segmenting them whenever an edge branches, we can compute a set of 1D curves. Although the shapes of the curves encoded in M_{f_1} alone can be used for matching image curves, there also exist edge descriptors [Mikolajczyk et al (2003)] robust enough to detect bicycles in images.

7.2 Building the Object Database

Given a geometric CAD model of the object along with real-world training images labeled with the object pose, the compilation process automatically analyzes the feature statistics and records the relationships between features and poses allowing pose extraction without forcing traditional 2D/3D feature correspondence matches. During the training phase, we compute the stability density of feature detectors on the object surface and use it to extract the most prominent **geometric** and **visual** word clusters. The true power of the presented database comes from the unsupervised learning of stable features. Furthermore there is no restrictions on the quality of the feature detectors to be used; the system itself will determine if a feature detector is suitable for detecting the pose or not.

We first analyze the feature statistics and extract the most *stable* features. We then build a database that stores the relationships between extracted features and the object pose they were found in. Because relationships are view-dependent and stored as pose sets, information pertaining to the visibility of features is preserved and taken advantage of in the search process. For example, if a particular feature only appears on the front side of the object, only poses with the front side showing will be hypothesized when that feature is detected in the image. The run-time search process randomly selects a set of features, generates a set of hypotheses consistent with all of them, and then validates each hypothesis

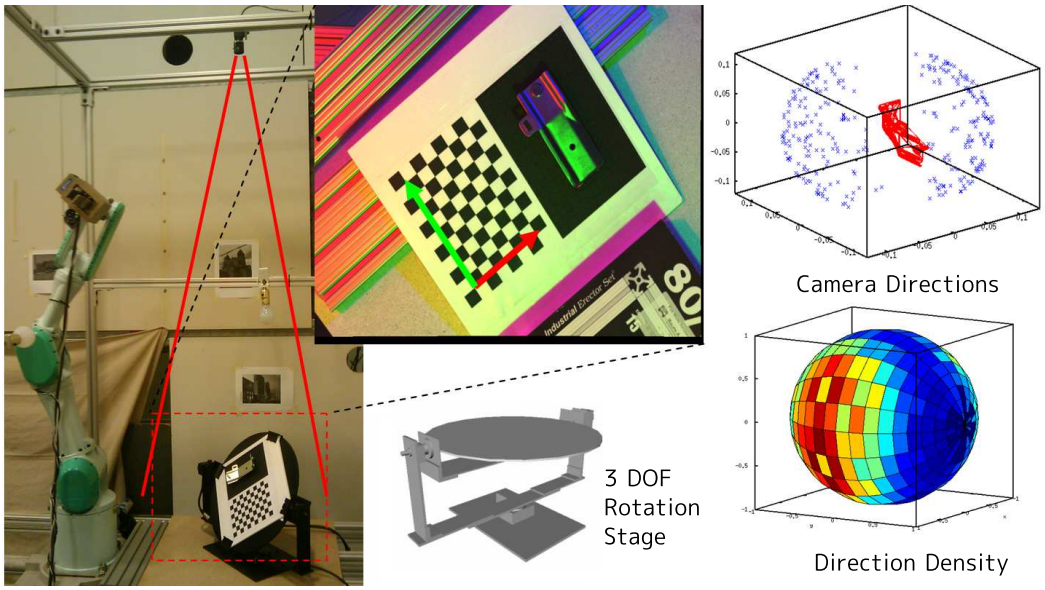


Figure 7.3: A 3DOF motorized stage and a checkerboard pattern for automatically gathering data.

with all the neighboring features using a learned classifier. Since the framework uses the CAD model of the object, it can compute the depth map of the object, measure inter-occlusions due to complex objects, and perfectly segment out the region the object occupies in the image. It it the accessibility of the geometric information that allows us the extract noise-free information and compute the stable geometric and visual words.

The generation process itself is divided into three main components as shown in Figure 7.2. The first four stages analyze all the feature detectors independently of each other.

7.2.1 Gathering Training Data

The CAD model is used to define the coordinate system of the object and to extract the 2D projected region along with expected depth values. The **6D pose** we extract from images is the **affine transformation** of the CAD model with respect to the camera. Figure 7.3 shows the setup we use for gathering the training data. We first gather images annotated with the pose of the object by placing the object next to a stable calibration pattern whose 6D pose can be easily recovered. Then we measure the relative transformation between the pattern's and the real object's coordinate systems. We mount this setup onto a 3DOF platform, solve its inverse kinematics equations via the **ikfast** tool developed in Section 4.1, and take images uniformly distributed around the object's surface. The object's pose can be recovered by multiplying the calibration pattern pose and the measured delta transformation.

Figure 7.4: Example rendering a blue mug from novel views using only one image in the database.

Image-based Rendering

The searching processes for induced-poses require being able to gathering a dense set of poses that produce a specific feature. The images we can realistic gather on a robot stage are on the order of 300-1000, any more images would significantly slow down the analysis processes. Therefore, being able to render the object from novel views can greatly reduce the database. Because we have a labeled set of training images, we can simulate rendering of the part from novel views given the lighting conditions the part was captured with. Assuming the lighting stays constant throughout the entire time the database is capture, we can extract the bi-directional reflectance distribution function of every point on the surface of the object, which will allow accurate simulations of the color. The most naive way to compute image-based rendering is by extracting textures from every image and linearly interpolating the textures based on the nearest neighbors of the novel view. When using such a naive method, stitching two different could be difficult due to lighting variations, therefore it is better to stick with the closest image. Figure 7.4 shows an example of rendering one image from novel use by extracting each texture.

7.2.2 Processing Feature Geometry

The goal of this phase is to transform all image features into the object's 3D local coordinate system. Once all features are in the same coordinate system, we can start analyzing their distributions and occurrence density on the object surface. Because the object geometry is known and we have a labeled training set, the depth map $Depth(T, K) : \mathcal{R}^2 \rightarrow \mathcal{R}$ can be extracted from the image where K is the intrinsic matrix of the camera. We assume any

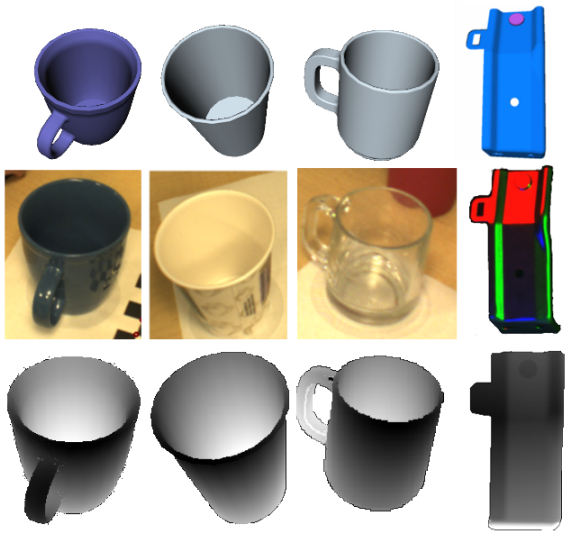


Figure 7.5: Examples CAD models (first row), the training images (second row), and the extracted depth maps (third row) of several objects.

nonlinear distortions due to camera lens are already removed from the incoming images, so we deal with ideal pin-hole cameras. Figure 7.5 shows several examples of objects and their depth maps. Using depth information, we can compute the 3D location of the feature on the object surface. Let $\bar{x} \in M_f$ be an image point on the feature, then the corresponding 3D point is

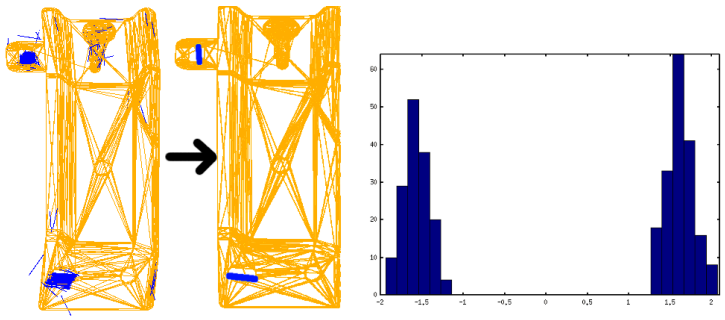
$$X = T^{-1} K^{-1} \begin{bmatrix} \bar{x} \\ 1 \end{bmatrix} \; Depth(T, K)(\bar{x}) \tag{7.1}$$

where $T = \begin{bmatrix} R & t \end{bmatrix}$ is the object pose.

If a point feature has an image orientation θ, we to compute its orientation in the object's coordinate system. By using the object's surface normal $n(X)$ at the 3D location of the feature, we can extract a unique direction D that is orthogonal to $n(X)$ and its projection matches θ:

$$D \propto \left(R^{-1} - n(X)\, n(X)^T R^T \right) K^{-1} \begin{bmatrix} \cos(\theta) \\ \sin(\theta) \\ 0 \end{bmatrix}. \tag{7.2}$$

Figure 7.6 shows the set of raw directions extracted from the entire training database and the resulting mean directions extracted after filtering and clustering. During the pose



Raw Clusters In-plane Angle Distribution on Object Surface (radians)

Figure 7.6: Orientations from image features are projected onto the object surface and are clustered. By re-projecting the raw directions on the surface plane and converting to angles (right graph), we can compute all identified angles (in this case 0 ° and 180 °) and the standard deviation of the particular mean direction ±15.5 °.

extraction process, if a feature coming from the detector used in Figure 7.6 has a direction, we can use the clustered directions to narrow down the originating position to two possible surface positions within a $[-31\,^\circ, 31\,^\circ] \bigcup [149\,^\circ, 211\,^\circ]$ in-plane orientation offset.

7.2.3 Feature Stability Analysis

Once all the features have been transformed to a common coordinate system, we can start examining the occurrence statistics and check if there are any meaningful locations on the object surface where a feature detector fires frequently and is stable. We first compute the density of feature positions on the object's surface using kernel density estimation [Ihler and Mandel (2003)]. The normalized density estimate with respect to the training data density (lower right graph in Figure 7.3) becomes

$$p_{\mathcal{F}_0}(f \in \mathcal{F}_0 \mid X) \tag{7.3}$$

By pruning all positions that are less than a preset threshold, we can single out the stable locations (Figure 7.7). Setting the correct threshold is very tricky since different parts have important features at different distances from each other, also the threshold should be independent of part size. For 0D features, it is enough for the kernel to be just a function of 3D position. However, for 1D features, the direction of the extracted edges is very important information and has to be included in the KDE:

$$p_{\mathcal{F}_1}(f \in \mathcal{F}_1 \mid X, \overrightarrow{X}). \tag{7.4}$$

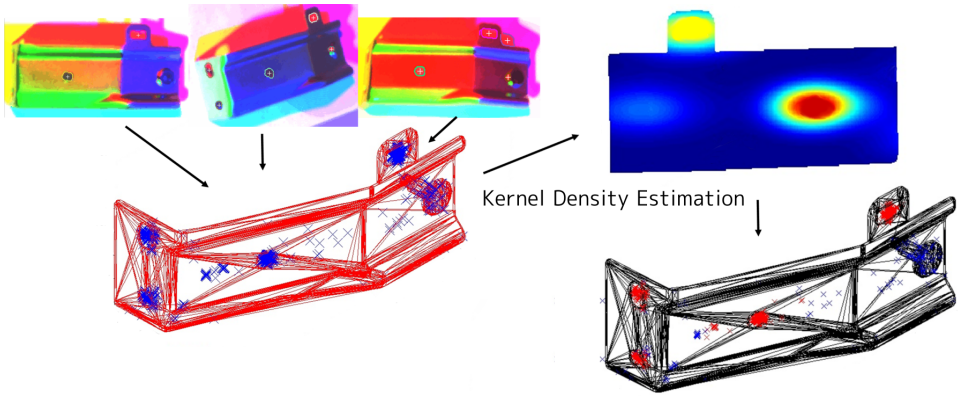


Figure 7.7: Shows the stability computation for a hole detector and the stability detected locations (red).

Because there is no ordering on the directions of the lines (the negative direction is just as likely), it is recommended to train the KDE with both directions specified per sampled point. Figure 7.8 shows the stability results of a line extractor for one part and how the line features in one image get pruned.

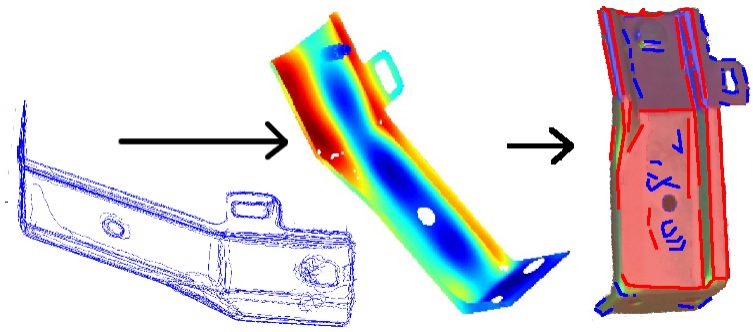


Figure 7.8: Stability computation for a line detector (upper right is marginalized density for surface). The lower images show the lines from the training images that are stably detected (red) and those that are pruned (blue).

After many trials, we have found that first evaluating all the stability measures for all features and setting the kernel bandwidth to be difference of the lower and upper quartiles of the data, with the threshold for stability being the median of the data works the best. Such parameters are independent of object scale, which is a prerequisite for robust thresholds. Figure 7.9 shows the stability distribution and results of a hole feature detector. Surprisingly, some of the holes marked in red on the CAD are determined to be unstable, and therefore bad for detection. When asking a person to set the thresholds manually, it is likely to get a

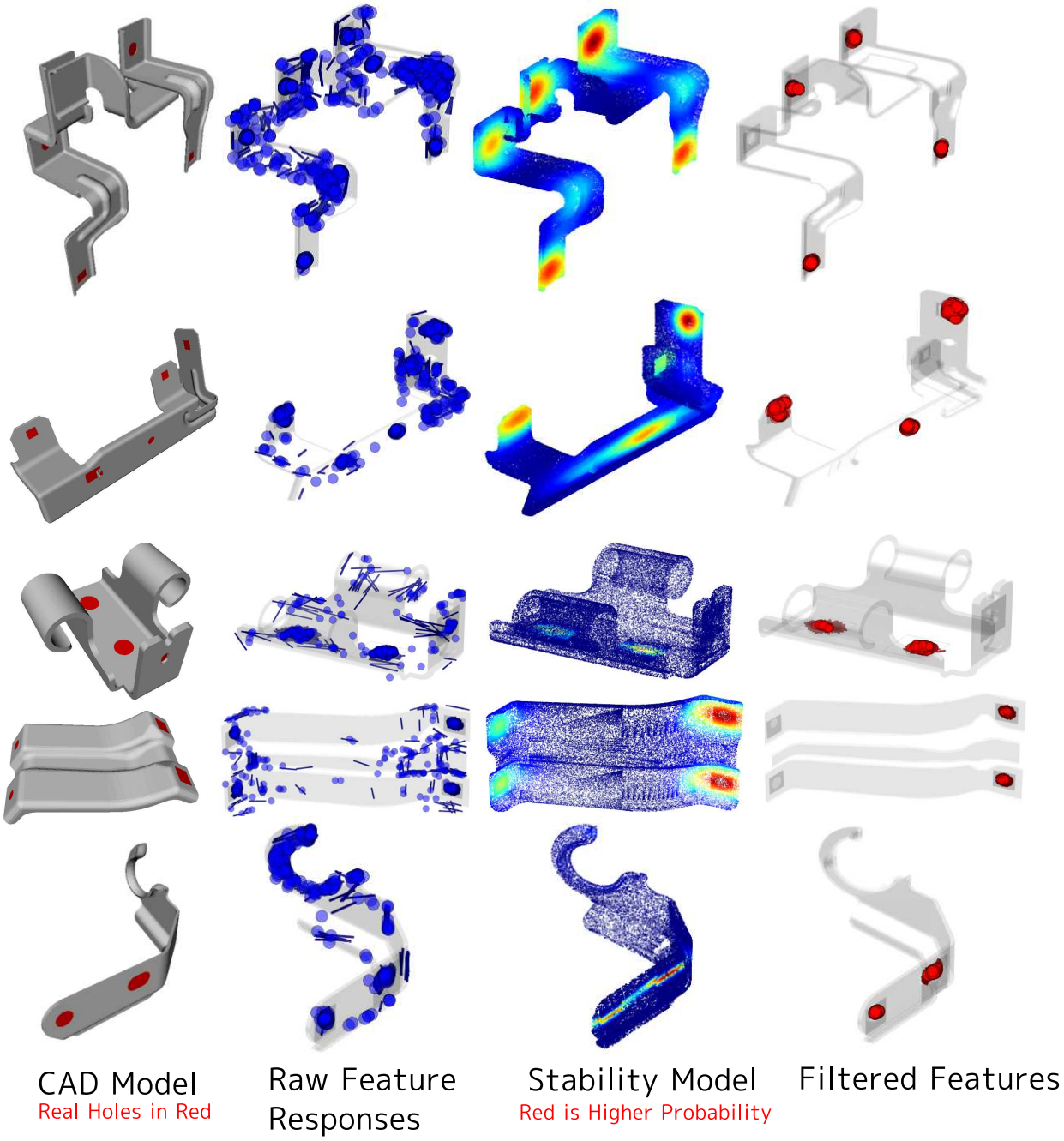


Figure 7.9: Shows the statistics of a hole detector on industrial metallic parts. Each part was trained with 300 images around the sphere. The right side shows the final filtered features, which are used for extracting geometric words.

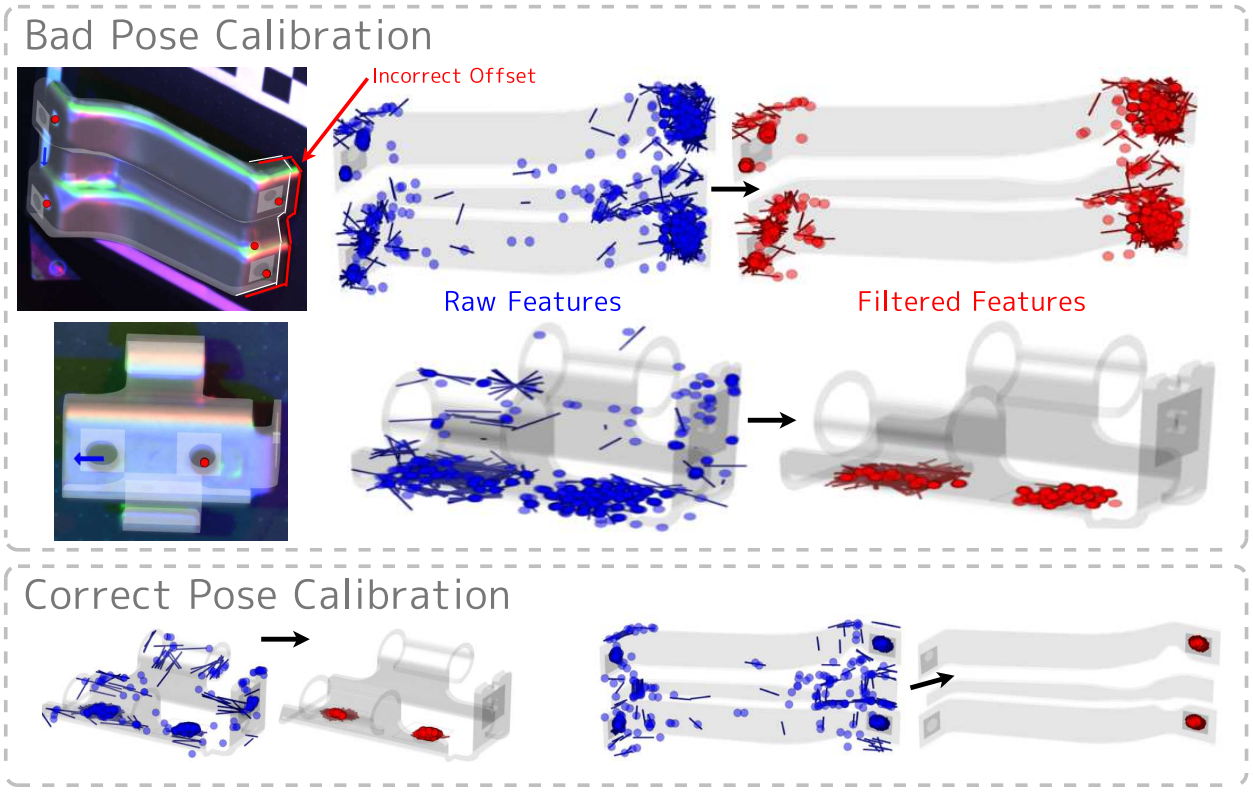


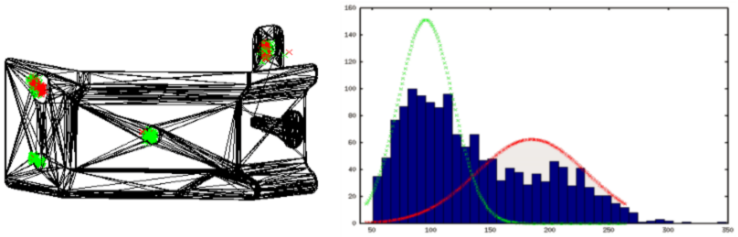
Figure 7.10: Shows the effect of poorly labeled data sets on the raw features (blue) and the final filtered features (red). The filtered clusters for the correctly labeled set become much more clear.

threshold where all holes pass, which is not always correct.

Finally, we should stress the importance of an accurately labeled training set when computing these statistics. Figure 7.10 shows a comparison of the features extracted from a poorly measured labeled set, to ones with a more accurate relative pose calibration pattern. Even when the offset is a couple of millimeters in the real world, it can have dire consequences on the distribution of the affected points, therefore the training image gathering phase should be carefully performed so such errors are not introduced into the system.

7.2.4 Geometric and Visual Words

Using the stable positions for each detector, we can cluster the descriptor vectors for 0D features and the curve shapes for 1D features. We call the mean of all descriptor clusters visual words and the mean of the shape clusters geometric words. Given image features $f \in \mathcal{F}$, we can quickly check if they match to any of computed words $\mathcal{W}_{\mathcal{F}}$ and reject them from being considered as valid features if there is no match. This is a powerful way to reduce



Labeled Features Histogram and Clusters of Sizes

Figure 7.11: The histogram and labels of a feature's descriptor distribution on the object (in this case, it is the size of the holes).

the number of features considered during pose hypothesis generation.

To cluster the 0D feature descriptor space, we employ an Expectation-Maximization algorithm with an automatic initialization of expected number of clusters. Figure 7.11 shows a simple clustering of a hole descriptor for area.

Analogously, we can develop a geometric set of words for 1D features by clustering the stable edge outputs and extracting the mean shape. We first develop a curve alignment algorithm based on Iterative Closest Point methods in conjunction with simulated annealing and *softassign* [Gold et al (1998)]. Once the algorithm fits the curves as best as possible, we can sum all closest-distances between the two curve point clouds to get a rough measure of the shape difference. Using this distance metric, we sample a subset of lines from the training images and compute the distances between all pairs. This distance graph has the property that curves with similar shapes form cliques when the edge connectivity is thresholded based on distance. Using a fast approximate clique finder [Niskanen and stergrd (2003)], we can quickly compute all the curve clusters as shown in Figure 7.12. For each curve cluster, we choose the mean as the curve whose sum of distances to all other curves is the least.

For each word $w \in \mathcal{W}$, we separately compute the probability density p_w of its occurrence along the object's surface. This density is used during the pose evaluation phase.

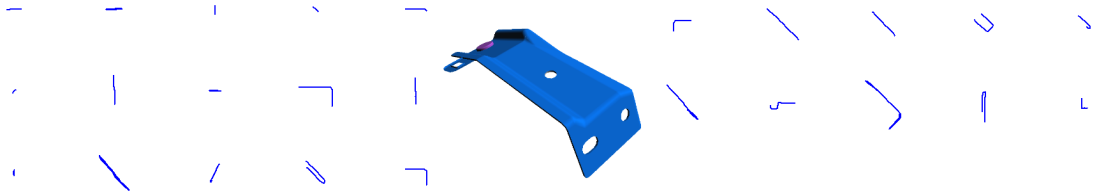


Figure 7.12: The major shape clusters extracted. The clusters on the left (simpler shapes) are more frequent than the clusters on the right (more complex shapes).

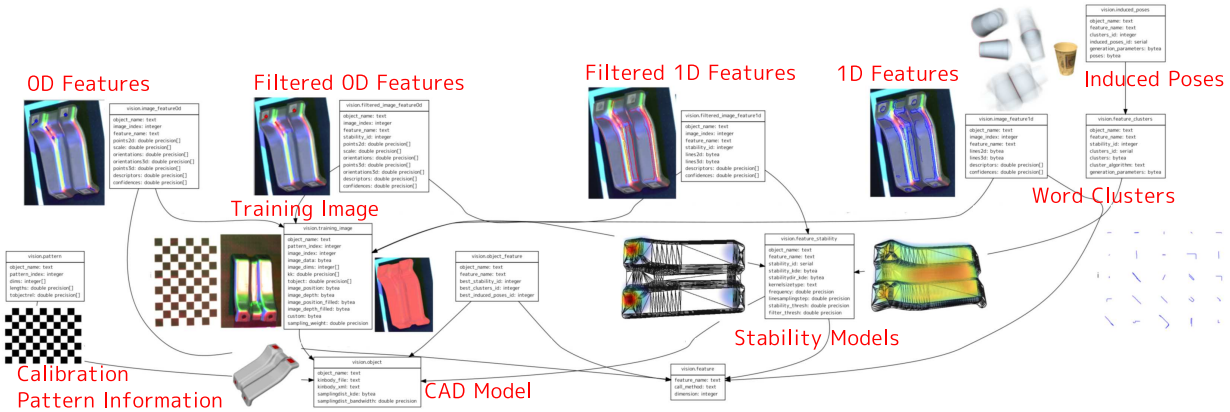


Figure 7.13: PostgreSQL database that stores the relationships and dependencies of all offline processes used for pose extraction and object analysis.

7.2.5 Relational Database

We use the SQL relational database query language for organizing the statistical data and results for a specific object. Each set of algorithms and results are divided into their own table and have unique ids for instances of these parameters. It is very common for processes to rely on the results of other processes. For example, the stability measure relies on the feature extraction and 3D geometric reprojection results in order to build a distribution of features. Each process also has its own generation parameters like thresholds that make it unique. SQL databases allows such relations to be encoded, and thus it can build an information dependency graph of object data shown in Figure 7.13. The functionality of the object database is very similar to the planning knowledge-base discussed in Chapter 4 (Figure 4.1). The database can store the computation of every process with multiple sets of generation parameters, which allows us to easily test new parameters and keep track of stale models because their dependencies were updated.

The biggest advantage of SQL is that it is a query language, which separates the database storage from the actual code and OS used to generate the data. Creating unique ids for every object allows us to create a global database of the appearance and statistics of objects that users can easily query over the network. Although it would be difficult to apply this to home scenarios where the environments are unstructured and the object types are on the order of thousands, factories already have databases of every component that goes into their final products. Augmenting them with the object appearance and statistics could go very far in automating the vision processes of assembly lines handling the parts.

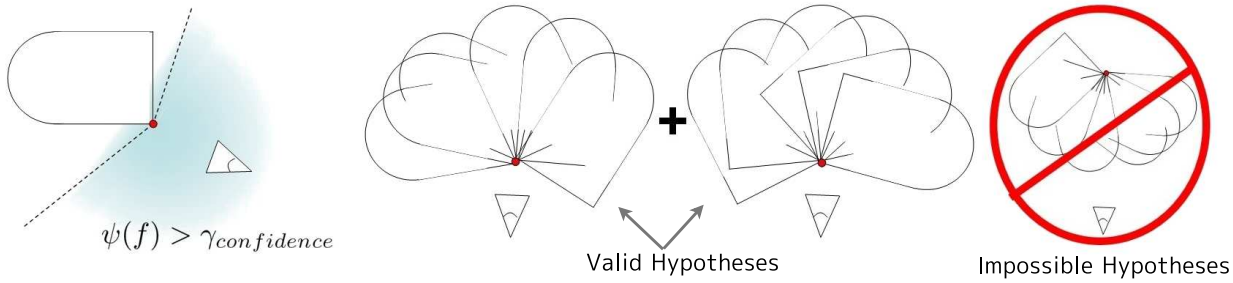


Figure 7.14: An example of the induced poses for a corner detector. The corner confidence $\psi(f)$ (top) is used to prune out observations with low confidence. The final induced poses $\mathcal{T}(f)$ for a corner detector are stored as a set (bottom).

7.3 Pose Extraction using Induced Pose Sets

A fundamental flaw many pose extraction algorithms suffer from is their usage of 3D object features: a traditional extraction process first hypothesizes a set of matching correspondences between image features and the 3D object features and then fits the best pose using linear algebra. The fundamental problem with this traditional view is that the *view-independent* 3D object features are forced to represent the *view-dependent* output of a 2D feature detector. This puts strict scale and orientation invariance constraints on the actual usable feature detectors. Because of the inherent discretized nature of an image, no feature detector exists that can produce a view-independent descriptor on an object with an arbitrarily shaped surface.

To solve these problems, we present a pose search algorithm that *does not* require the use of 3D object features, which allows it to remove any view-independence requirements on feature detectors and completely eliminates the correspondence step between 2D image features and 3D features. Simply stated, our solution is to take a brute-force approach and store all possible views of the object that could generate the particular features of interest. This concept is called **induced pose sets**, which is the set of all pose hypotheses consistent with the image feature they are encoding. At run-time, a set of possible pose hypotheses is queried by matching the image features with the database features. During the run-time pose search process, pose hypotheses are generated from intersecting the induced pose sets using neighboring features in the image. Not relying on 3D features allows us to easily automate the training process because a human or algorithm does not have to struggle matching the output of a feature detector to specific locations on the 3D object geometry.

In order to validate each pose hypothesis with the entire image, we present a classifier that measures the contributions of all the features that lie on the projected region of the

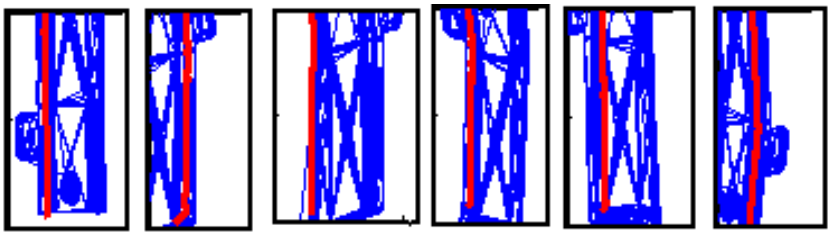


Figure 7.15: Several induced poses for a particular curve (red).

part.

7.3.1 Generating Induced Pose Sets

The concept of induced poses allows us to store most nonlinear, view-dependent information about the relationship between features and the pose of the object. In order to reduce computation and filter unstable features, we only generate induced poses for the visual and geometric words computed in Section 7.2.4. Each feature f can induce a set of object poses $\{T\}$ such that the object seen at one of these poses has a feature similar to f. For example, assume that $\mathcal{F}_0$ is a corner detector, and the object is a square joined with a circle (Figure 7.14). The fact that a corner is visible in an image greatly constrains the possible poses of the object. These constraints are dependent on the geometry of the object, the feature detector itself, and the feature detector's confidence $\psi(f)$. Therefore, the most reliable way of capturing these constraints without imposing any restrictions is by a discrete set of **induced pose hypotheses** $\mathcal{T}(f)$ that cover every word f (Figure 7.14 right):

$$\begin{aligned} \mathcal{T}(f) = \{T \mid & \exists g \in F(Render(T)), \\ & \psi(g) > \gamma_{confidence} \\ & d(f,g) < \epsilon_f, \\ & d(M_f, M_g) < \epsilon_M\} \end{aligned} \tag{7.5}$$

d denotes a distance metric corresponding to the respective inputs and $Render(T)$ is the image of the object at pose T. Equation 7.5 represents the space of poses whose rendered image $Render(T)$ contains contains a feature g that matches with the query feature f with high confidence. In order to minimize space requirements, the poses themselves are normalized with respect to the feature's image position and orientation, which removes three degrees of freedom. Figure 7.15 shows a set of poses for a specific curve. Figure 7.16 shows the sets using mean images.

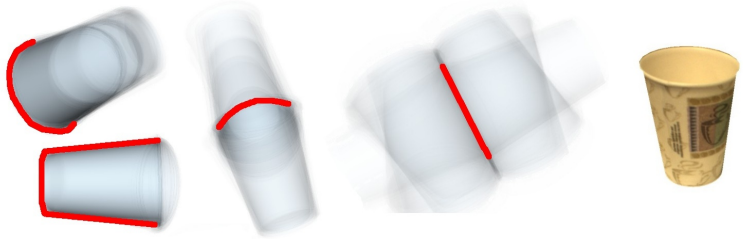


Figure 7.16: Mean images of the induced poses for an edge detector of a paper cup (in simulation).

The construction of induced poses first starts by seeding each of the visual and geometric words with the pose information from the training data. In order to search other poses, we use image-based rendering of novel views (Section 7.2.1). To quickly explore the pose space around the seeds, we use the Rapidly-Exploring Random Trees algorithm [LaValle and Kuffner (2000)], which can maintain all the constraints in Equation 7.5. It is necessary to use RRTs for exploration of the space because the constraints really restrict the solution space, and hitting a point in the solution space by just randomly sampling will take a very long time.

Pairwise Poses

Unfortunately the induced pose sets for point features are very big. If a word does not have an associated orientation, its induced set is 4 DOF. With reasonable discretization values, the size of the set can reach 600000 sets, which leads to memory and speed problems in the pose search phase. In order to reduce the computation, we create a **pairwise pose set** by storing the consistent poses of two point words. If none of the words have orientations, the degree of freedom of the sets is two and the set is parameterized by two image positions. If the first feature has an orientation, then the degree of freedom is one, and the set is parameterized by two image positions and the relative orientation of the pairwise direction with respect to the first image feature's orientation. If there are a total of N visual words for all detectors used, then there are at most $\binom{N}{2}$ pairwise sets. Even though the number of induced sets increases, adding pairwise poses greatly speeds up the search task and reduces memory constraints making it a necessary step for this framework.

7.3.2 Pose Evaluation and Classification

The last stage builds a classifier using all detector responses to decide whether a pose hypothesis is consistent with all the image features inside its boundary.

Up to now we have analyzed the output of individual detectors and covered the generation

Algorithm 7.1: $score \leftarrow$ EVALUATEDETECTOR(T,$\mathcal{F}$)

$score \leftarrow 0$
for $f \in F(I)$ **do**
  $X_{surface} = \text{RAYCAST}(K^{-1}M_f, R^{-1}\begin{bmatrix}0\,0\,1\end{bmatrix}^T)$
  $W \leftarrow \{w \mid w \in \mathcal{W}_{\mathcal{F}} \bigwedge d(f,w)\}$
  if $\mathcal{F}$ *is 0D* **then**
    if not $X_{surface}$ **then**
      continue
    $f^+_{score} \leftarrow \begin{cases} \log \sum_{w \in W} p_w(f \mid X_{surface}), & \text{if } W \neq \emptyset \\ \log p_{\mathcal{F}}(f \mid X_{surface}), & \text{otherwise} \end{cases}$
    $f^-_{score} \leftarrow f^+_{score}$
  else if $\mathcal{F}$ *is 1D* **then**
    if $\frac{|X_{surface}|}{|M_f|} < \alpha^F_5$ **then**
      continue
    $f^+_{score} \leftarrow \sum \frac{\log p_{\mathcal{F}}(f \mid X_{surface}, \vec{X}_{surface})}{density}$
    $f^-_{score} \leftarrow -\alpha^F_4 f^+_{score} + \sum \frac{\log p_{\mathcal{F}}(f \mid X_{surface})}{density}$
  if $\exists\, w \in W,\ d(T, \mathcal{T}(w)) < \alpha^F_1$ **then**
    $score \leftarrow score + \max(0, \frac{f^+_{score} + \alpha^F_2}{\mathcal{F}_{frequency}})$
  else
    $score \leftarrow score - \max(0, \frac{\alpha^F_3 (f^-_{score} + \alpha^F_2)}{\mathcal{F}_{frequency}})$
end
return *score*

of induced pose sets. During run-time, we use this information to generate pose hypotheses by picking a small random set of features and intersecting their pose sets. Due to this process, a hypothesis is only guaranteed to be consistent with a small set of features; therefore, we have to employ a separate pose evaluation metric that takes into account all the features that lie in the projected object region. We first present an evaluation metric for the contribution of individual 0D and 1D feature detectors, then we treat all the detector values as a vector and use Adaboost to find a separating hyper-plane between correct and wrong poses.

Given a pose hypothesis $T = \begin{bmatrix}R & t\end{bmatrix}$, we first find all features inside the object and then compute a score using EVALUATEDETECTOR (Algorithm 7.1). Since we have the geometry of the object, we can inverse project each feature onto the object surface using

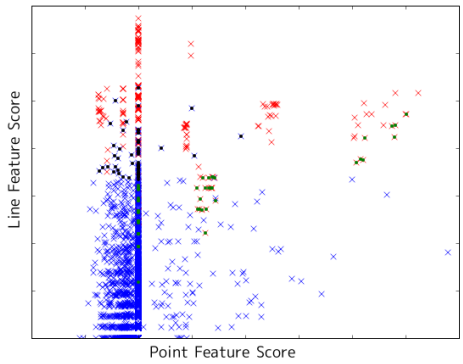


Figure 7.17: The individual detector scores for good and bad poses for the object in Figure 7.3. Using Adaboost, the misclassified good poses and misclassified bad poses are marked on the data points.

RayCast(pos,dir) and then use the stability measures computed in Sections 7.2.3 and 7.2.4 to compute how well the feature location matches the statistical models. If a feature $f \in \mathcal{F}$ is on the object surface and it matches to a word in $\mathcal{W}_{\mathcal{F}}$ and the distance to the induced pose set $d(T, \mathcal{T}(f))$ is small, then it is *positively supporting* and its stability score is added to the total; otherwise, it is *negatively supporting* and its stability score is subtracted. For 1D features, a certain percentage of the curve has to lie inside the object region for it to be considered in the score. Once a feature is verified to lie on the surface, EvaluateDetector first computes the matching words in $\mathcal{W}_{\mathcal{F}}$; if there is a match, the stability density p_w of the particular word is used, otherwise the density of the detector $p_{\mathcal{F}}$ is used.

Classification

The final classifier uses EvaluateDetector to convert all the image features into a D-dimensional detector space and then uses boosting [Pham and Cham (2007)] to separate the space for correct and wrong poses statistics. The detector space greatly depends on the $\alpha^{\mathbf{F}}$ parameters in EvaluateDetector (Algorithm 7.1), which are dependent on the feature detector. Because the final goal is to build a D-dimensional space that easily separates the correct poses from the wrong, we learn the evaluation parameters while simultaneously learning the separation criteria via boosting. The evaluation parameter space α is highly nonlinear, so we use Markov Chain Monte Carlo (MCMC) optimization to search the solution space. For every new state α queried by MCMC, we compute the best AdaBoost classifier and set the probability of the Markov state as $e^{-\epsilon'}$, where the ϵ' is the classification error of the new state. The MCMC state transitions to the newly sampled state with a probability of $e^{\gamma\,(\epsilon-\epsilon')}$.

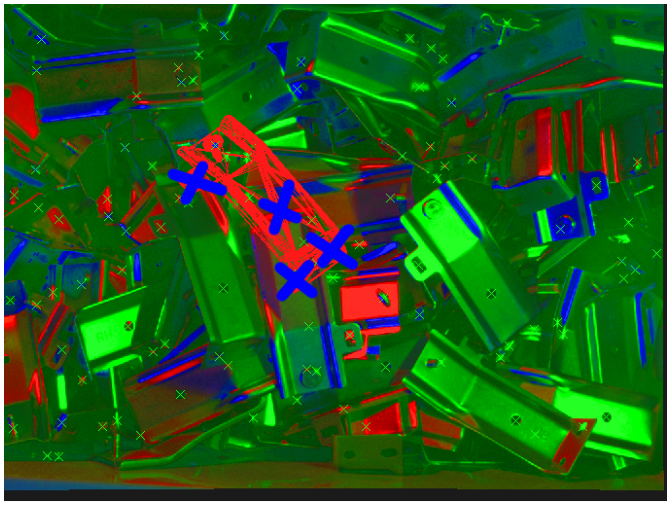


Figure 7.18: For an object that has four holes, there are many valid poses of the object in the real image that can hit all four holes. In fact, using hole features is not enough to extract a confident pose, 1D curve features are absolutely necessary.

By learning a separating classifier, the importance of a feature detector to the evaluation process can be quantitatively computation, a task which was done previously by people. From looking at the classification results of points vs lines for a metallic part shown in Figure 7.17, we observe that point feature scores do not play a key role in determining how good a particular pose is. This result only becomes clearly apparent when trying to extract meaningful poses just from the point detector as shown in Figure 7.18. Pose extraction will frequently return poses where all holes are matched to an image feature, but the pose actually straddles many parts.

7.3.3 Pose Extraction Process

The final pose extraction algorithm is very reminiscent to the way RANSAC searches for solutions. The basic algorithmic process is shown in Figure 7.19. After all the image features are computed, we find all matches with the visual and geometric words database $\mathcal{W}$ computed in Section 7.2.4. For each matched feature f, we gather the canonical induced pose sets from $\mathcal{W}$ and transform them into the current image using the feature's location M_f. Then we pick a random *support feature* and start searching its neighborhood for valid poses using SearchWithSupport (Algorithm 7.2). Given the neighborhood of the supporting feature, we sample K features successfully matched to words in the database (the combinations are sampled without replacement). Because of the compilation process, we are guaranteed that the poses lying in the intersection of the selected features' induced poses are all consistent

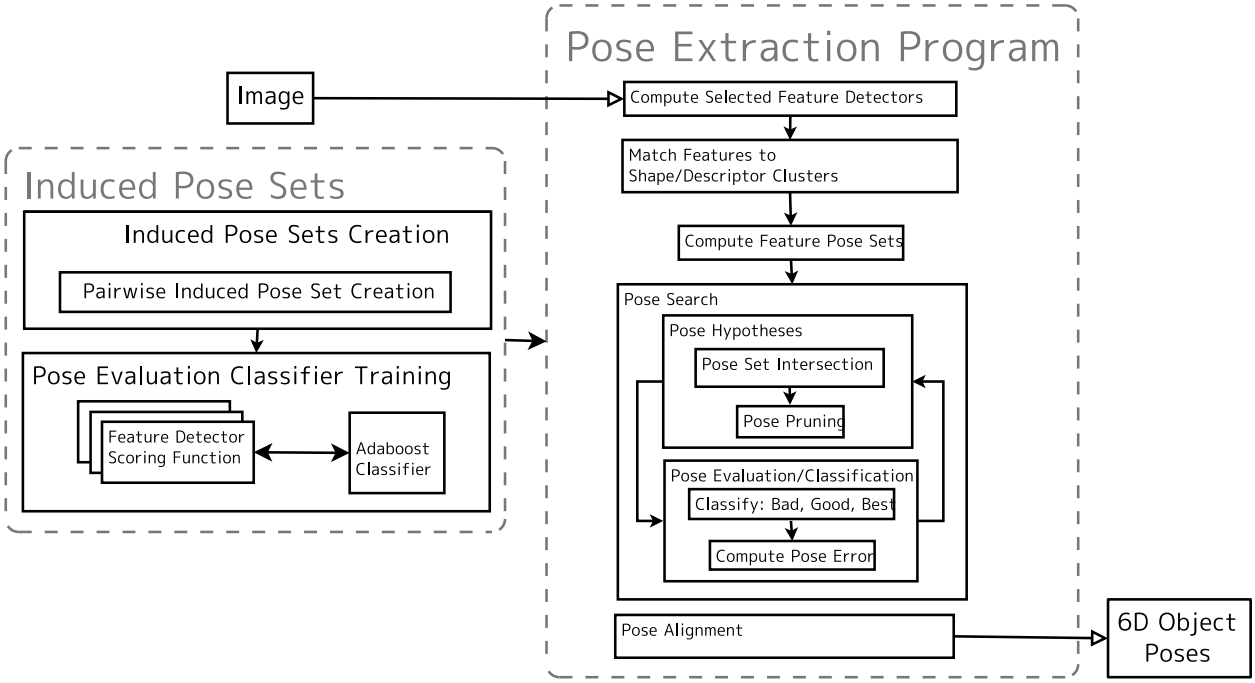


Figure 7.19: The search process first matches image features to a database, then randomly selects a set of features and tests for consistency by intersecting their induced pose sets. Each pose candidate is validated based on positively supporting and negatively supporting features lying inside the projected object region.

with the features. If the intersection is not empty, then we evaluate every valid pose T by first getting the D-dimensional vector in the detection space, and then running the learned Adaboost classifier h. It is important to note that the search process simultaneously considers **all** the consistent pose hypotheses within a given discretization limit.

Figure 7.20 shows an example of transforming the induced pose sets into pose hypotheses for particular image features. The first row shows the pose hypotheses for two point features and the resulting set of poses when the hypotheses are intersected. Even though the number of consistent poses is greatly reduced, it is still not enough to completely constrain to one pose. The second shows the same process except with curves; the pose hypotheses for one curve are much smaller, therefore they constrain the pose more. Finally by choosing two curves and two points that lie on the part, we can completely constrain the pose to the correct one. Although the method is very effective when choosing a set of features that all lie on the same part, RANSAC will more frequently choose image features that do not lie on the same. Because the number of chosen feature is three or four, the intersected pose sets

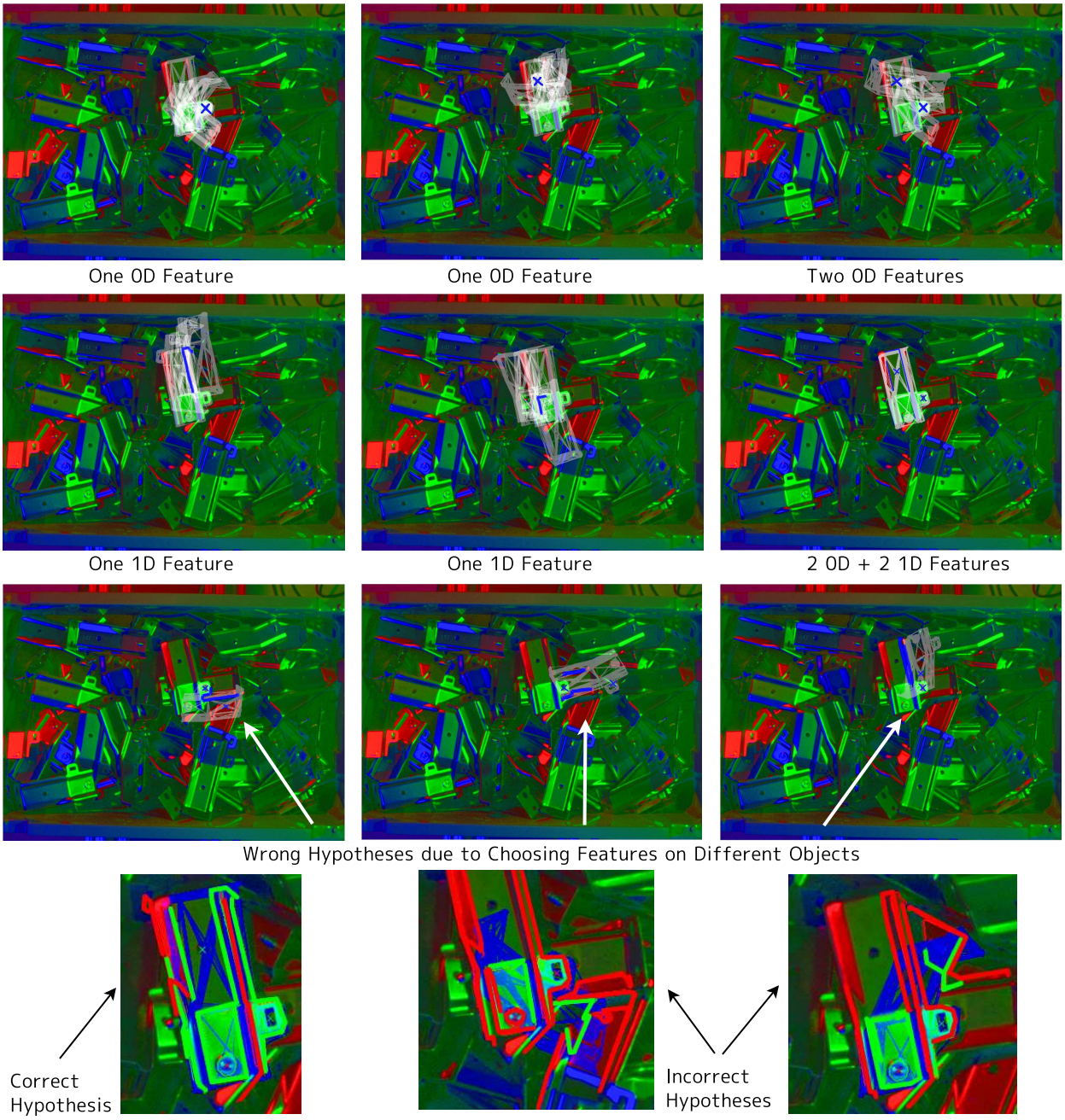


Figure 7.20: Several examples of chosen image features (blue) and their pose hypotheses (in white). The bottom row shows the positively (green) and negatively (red) supporting features of the pose hypothesis (blue).

Algorithm 7.2: $T \leftarrow$ SEARCHWITHSUPPORT(support)

$Neighs \leftarrow$ FEATURENEIGHBORHOOD(support)
for $F =$ SAMPLE$(\mathcal{C}(Neighs, K))$ **do**
  for $T \in \bigcap_{f \in F} \mathcal{T}(f)$ **do**

$$V \leftarrow \begin{bmatrix} \text{EVALUATEDETECTOR}(T, \mathcal{F}^1) \\ \text{EVALUATEDETECTOR}(T, \mathcal{F}^2) \\ \dots \\ \text{EVALUATEDETECTOR}(T, \mathcal{F}^D) \end{bmatrix}$$

    if $h(V) >= 0$ **then**
      return T
  end
end
return $\emptyset$

are very likely to be non-empty. In fact, most of the processing time of the search algorithm is rejecting invalid pose hypotheses because the chosen features do not lie on the same part. In order to speed up search, the evaluation function is written so as to quickly reject a hypotheses based on the order of features it looks at without having to evaluate all the features. The final row shows the positively and negatively supporting features determined from the pose evaluation phase. The correct pose has mostly positive supporting features, therefore it passes the test. The incorrect poses pass through a lot of wild features that do not support it, therefore most of the features it touches subtract the final evaluation score.

7.3.4 Experiments

We have successfully tested the *induced-set* pose extraction method for both metallic industrial parts (Figure 7.21). The training process is time consuming and currently takes on the order of a day for 300 training images. Using a hole and line feature detector, extracting the initial 3D geometry of the features takes about one hour. Compute the stability measure takes 10-20 minutes per feature. Clustering the filtered hole features into the visual and geometric words took minutes. Clustering the filtered curves takes 10-20 hours when using ICP *softassign*; afterward, we reduced this down to less than 30 minutes. Building the induced pose sets takes on the order of 5 hours for each feature where the average time used to search each words is 5 minutes.

The goals were to increase detection rates and quality of the result rather than increase speed, so we set small discretization factors for most of the generation stages in the algorithm.

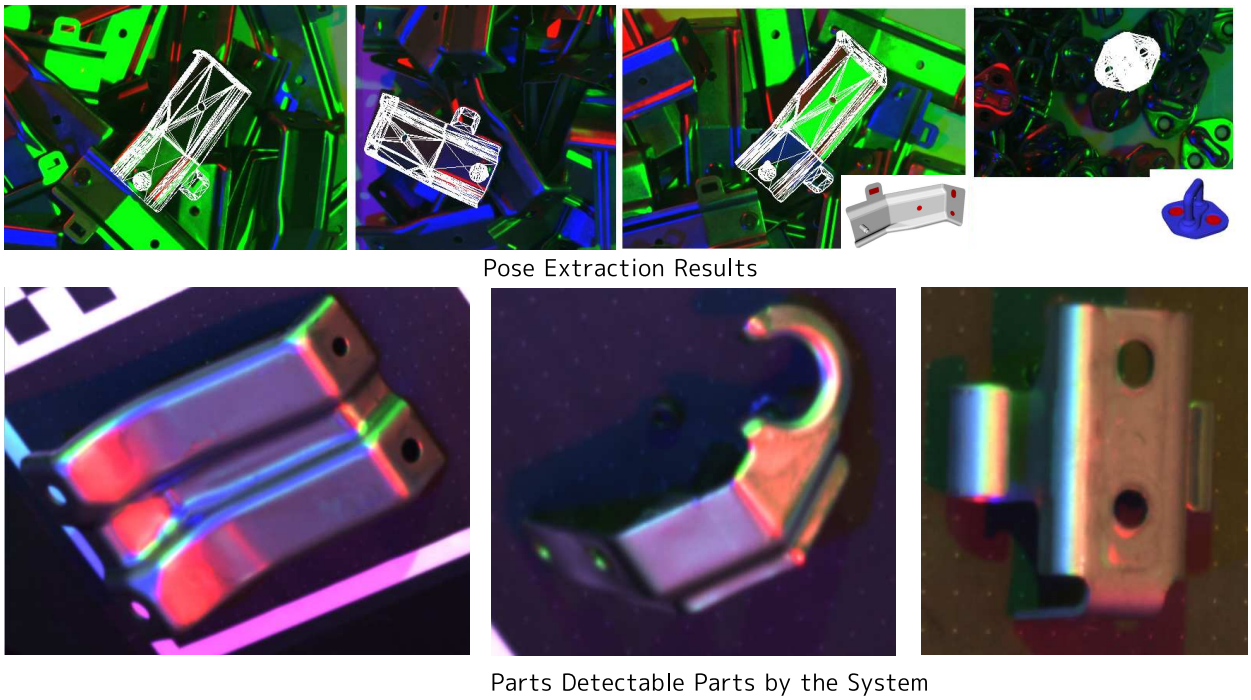


Figure 7.21: Results of pose extraction using induced pose sets.

The detection process for the part shown in Figure 7.20 takes 2 minutes on a multi-core computer system running four threads. The bulk of the time is spent matching image features to the database of words. After all the feature are matched and pose hypotheses are built, it takes 10-20s to detect the first pose in the image when RANSAC samples four features at a time; the combination is not tested again. If the four sampled features are on the same part, then our method will immediately find the correct pose; however, most combinations of features involve more than one part. Therefore, the evaluation function (Algorithm 7.1) was optimized for early rejection of hypotheses, being able to evaluate a pose in less than 0.05 seconds. For a full bin, the success rate is greater than 98%.

Because we are interested in picking up all parts from the bin, we put 100 parts in the bin and took out a part after it was detected by the system. Because the average height and number of candidates for detection slowly decreased as we removed parts, the detection rate started dropping. Of the 100 parts, there were 5 times that the program could not detect a part.

We measured the ground truth error in each of the six degrees of freedom of the pose by mounting the part to high-precision linear and rotary stages. The camera is 2 meters above the part, so we would expect very large errors in Z. The part first starts with its

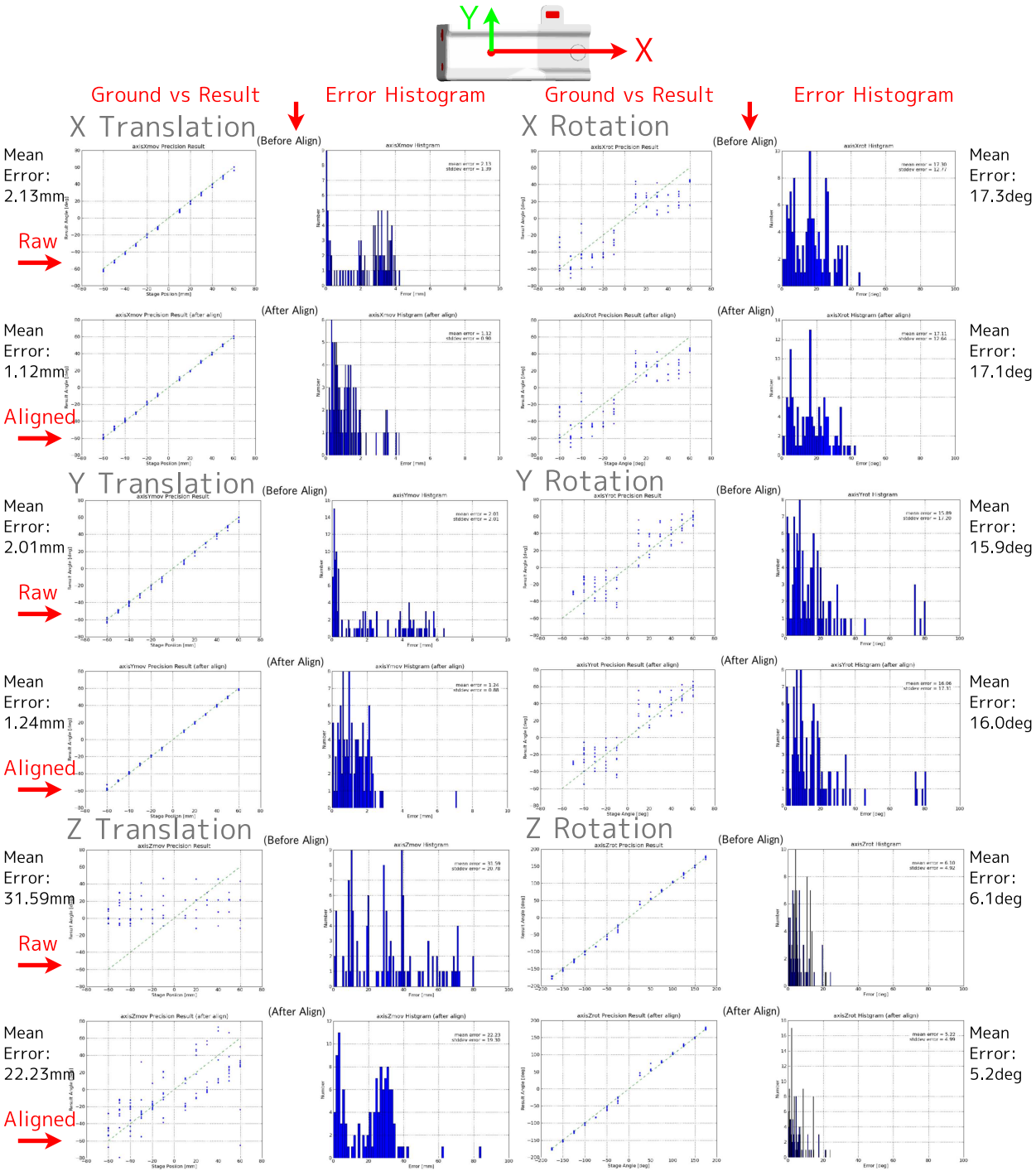


Figure 7.22: Ground-truth accuracy results gathered for each DOF of the pose.

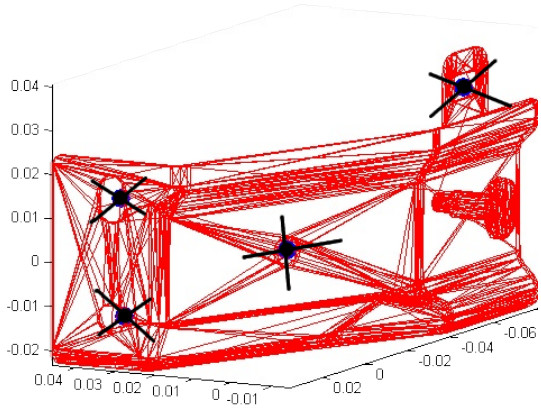


Figure 7.23: Some geometric words cluster very densely around specific points on the object surface. These stable points can be used for alignment.

biggest face pointing towards the camera and the system records the first pose (Figure 7.22 top). For every delta movement the stage makes, the new detected part should have the same delta movement. The deviations from all delta movements of all degrees of freedom are recorded and shown in Figure 7.22. The in-plane movements of the camera were the most accurate, with the average raw translation error being 2.1mm, and the average raw rotation error being 6.1°. The out-of-plane rotation error is on average 17°, we believe this is due to random choosing of supporting features for building up the pose hypothesis. In order to speed up the evaluation function, we train the evaluation function in the learning phase to pass semi-good results also. Because the evaluation function directly looks at the projected footprint of the part, the larger the change in footprint, the more chances it has to intersect with a bad feature. Out-of-plane rotations and changes is depth do not change the footprint of the part that much, in fact they just make it smaller or bigger without offsetting it that much. And our ground-truth measurements are consistent with this observations. It is interesting to note that the rotations around the Y-axis have a little smaller error than those around the X-axis. The reason can be explained again with the footprints since the part is twice longer on its X axis, than its Y axis; therefore rotations around Y move the X-axis part more.

Pose Alignment

Because of the inherent discrete sampling nature of the supporting features and the fact that the evaluation function is trained with slight pose disturbances, it implies that the raw pose measurements returned from the algorithm are not going to be matching the projection of the part perfectly. Therefore, we introduce a post-processing step that *aligns* the part to

the 0D features around it. Basically, some geometric words of the 0D point features cluster densely around the point surface, so the mean of this cluster can be extracted with very high confidence as shown in Figure 7.23. If the detected part uses any of these geometric words in its matching, then it has a 2D position of their measurements, and it has their 3D coordinates. Therefore, we can use a 2D/3D point correspondence algorithms to find a new pose that perfectly aligns the 3D geometric word points to the 2D image points. The degrees of freedom we can fix depends on the number of point correspondences:

- **1 Correspondence.** Move the part only in the XY plane to match.
- **2 Correspondences.** Solve for the XYZ translation and best in plane rotation while maintaining the out-of-plane rotations.
- **3 Correspondences.** Solve for closest transformation using a perspective three points algorithm [Haralick et al (1994)].
- **4+ Correspondences** Solve using standard least squares techniques.

The even rows in Figure 7.22 show the improvement using the alignment. We always have at least one geometric point to align, therefore the XY errors decrease to 1.1mm average error, which reduced by 48%. The next parameters to see a benefit are the Z depth with an average error of 22.2mm (30% reduction), and the in-plane angle with an average error of 5.2° (15% reduction). The out-of-plane rotations were rarely affected, mostly because we rarely had more than 2 point correspondences to align. The alignment phase is very quick to implement and practical to increasing accuracy.

7.4 Discussion

Every pose extraction algorithms needs a set of features on the object to track and base its estimates on. The more discriminable the feature is compared to other features in the scene, the more easily the object can be detected. Furthermore, the stability of the feature on the object's surface directly relates to how well it can be localized within the object coordinate system, this implies that the pose estimates will be more accurate. In the first half of this chapter, we presented an object surface analysis method that can find stable and discriminable features. The training data for the method requires attaching the object of interest to an already known calibration pattern, which makes the method accessible to anyone. The power of having accurately labeled training data allows for a plethora of feature-surface statistics to be computed. In order to have the stability statistics be coordinate system and scale independent, we motivated the usage of the median and difference of lower and upper quartiles of the stability measures for filtering out bad features. The filtered features are

then clustered to yield a set of geometric and visual words specific to the appearance of the object. The analysis is general enough to be applied to any point or curve feature, even if it has a descriptor vector associated with it. Any rigid object of any material can be easily inserted and analyzed by the system. Furthermore, the generality of the method allows it to be used as a pre-processing step in any existing pose extraction algorithm.

We also performed a case study of organizing the computational process of the stability using a modern relational database. The database has many practical advantages of tracking data dependencies and can offer access to the data to other parties over the network. Using this database, we could easily train new objects into the system without worrying about tracking generation parameters and files. We believe that object-specific databases encoding vision information will become mainstream in industrial settings where every part has to be categories and carefully tracked around a factory.

In the second half of the chapter we presented a novel pose extraction method that takes advantage of the visual and geometric words computed from the stability database. Because each word stores the poses of the object that were used to generate it, it gave rise to the formulation of induced pose that inverted the feature generation process. Instead of the object's pose generating the feature, a feature could generate the possible poses it came from, a process very reminiscent to the inverse kinematics problem for motion planning. Induced-set pose extraction has the advantage of not relying on 2D/3D feature correspondences because it can implicitly encode the 3D feature information into a pose set estimate, this allows it to hypothesize all possible consistent poses given a set of image features. By being able to explicitly encode all consistent poses, the induced-set algorithm is not forced to commit early to a wrong decision anywhere in the process.

As long as the image features belong to the same object, induced pose sets can find a correct pose that supports all the features. However when multiple parts are scattered in a bin, the search process has a lot of difficulty picking objects that belong to the same part. Therefore, we also presented a learning-based pose evaluation method using positively and negatively supporting features. Because of the explicit handling of negative supporting features, small occlusions from other objects can really decrease the evaluation score, which guarantees that the detected parts are always completely visible and on the top of the pile. This behavior is desirable for industrial bin-picking scenarios because a robot eventually has to approach from the top and be able to pick up the part unhindered by other obstacles. Furthermore, we can use this behavior in a second way: if the system is asked to evaluate a pose of an already known part, the system can determine if the part is occluded and exactly where the occlusions occur.

Chapter 8

Conclusion

Even today in 2010, robots are not freely roaming the streets and autonomously moving in people-present environments because of frequent execution failures and safety fears. Especially in industrial robotics, execution failures come at the highest cost where an assembly line has to be stopped in order to recover from errors. Although it is possible to design such systems, it takes a lot of manpower and research to meet all the standards and handle all the possible conditions. With today's understanding of the problems, the mathematics and low-level robot control issues of manipulation are well-understood and automated; however, the less automated parts of manipulation continue to be the generation of more analysis-heavy components like object recognition and motion planning. By clearly identifying and showing how to automate several of the analysis-heavy components for pick-and-place manipulation tasks, we can enable many mass-production systems to be configured and put into use quicker, which has far-reaching implications for today's economy.

The presented manipulation architecture should be treated as a minimal set of components that achieves its task while fully exploiting the provided specifications. The contributions of the thesis center on the interplay between the kinematic, geometric, sensor visibility, and vision recognition elements. The algorithm design decisions made in each chapter prioritize the solvability of the problem, the efficiency of the solution, and the decrease in algorithm parameters that have to be tweaked. Furthermore, we concentrate on successful execution and task completion reliability over optimizing other quality metrics like generating smooth, time-optimal, energy-optimal, safety-optimal, or natural-looking robot motions.

8.1 Contributions

As a small step towards these goals, the thesis presented a framework for automated construction of the planning and vision processes required for reliable manipulation tasks. The focus was specifically on the geometric and statistical analyses of sub-problems common in manipulation. We sub-divided the problem into a planning knowledge-base and vision-based database and offered many generic algorithms that help in the entire spectrum of manipulation planning. Chapter 2 starts with the outline of the execution architecture for completely a simple manipulation task and maps out all the components necessary for reliable execution. Components that play a major role in the discussed architecture are: the goal configuration generators that analyze a scene and send informative configuration goals to the motion planners, the knowledge-bases that encode frequently queried information of the robot and task, and planning with sensor visibility. Using the layout of the architecture, Chapter 3 delved into the complexities of the planning algorithms necessary to connect the initial conditions and the goals. It presented the structure of generalized goal configuration samplers that form the basis for representing the goal space of a plan. Several types of planning algorithms were presented: planners having explicit goal spaces that can be sampled from, planners that whose goal condition is to validate a know path, and planners that can change grasps during a plan. Chapter 4 presented the structure of the planning knowledge-base and offered algorithms for the generation and usage of more than seven types of components that are critical for manipulation planning. By modeling the computational dependencies of each components, it has allowed us to be methodical on how information gets tracked throughout the system, and what domain knowledge is necessary for the generalized planners. Chapter 5 presented algorithms that consider the visibility of the sensors and offered two methods of combining the visibility planning with the goal-oriented grasp planning algorithms. It showed the importance and efficiency of having a camera sensor verify any object position before the system can begin planning to grasp it. Two experiments were presented: a mobile dual-arm humanoid that picks up cups from a sink and places them on a counter, and an industrial robot that picks up parts scattered in a bin. Chapter 6 presented a completely automated extrinsic camera calibration system using the planning and visibilities theories presented thus far. An advantage of the calibration system is that it can work in any environment and does not have to be initialized with any accurate measurements of the parameters. In order to verify the quality of the calibration results, we presented a measure based on how well each of the parameters are constrained by the data. Chapter 7 discusses two topics of interest in object-specific pose recognition: extracting stable and discriminable features, and *induced-set* pose extraction using a novel voting-based method that maps image features to pose hypotheses. It presented an automated data gathering method that allows a statistical

analysis of object feature distribution on the surface. The power of the induced-set method is that pose hypotheses can be easily generated from any image. Furthermore, the pose verification function be learned from training data, providing a way to automatically set thresholds and weights. Results were shown on a very difficult industrial manipulation scene and we proved millimeter accuracy of the pose. Chapter A discussed the OpenRAVE architecture and the design decisions that allowed it to become a stable and reliable platform. OpenRAVE consists of a core that provides a safe environment for users, and a interface API that allows users to expand on the functionality without having to recompile the base code. OpenRAVE provides many key technologies that allow successful manipulation execution with real robotics, some of the most critical ones like *padding* and *jittering* were discussed in detail.

The specific contributions of the thesis are:

- A small set of manipulation planning algorithms based on the efficient search properties of the Rapidly-exploring Random Trees formulation. These algorithms allow grasp planning, mobile manipulation planning, and planning with a gripper camera. Chapter 3.
- A planning knowledge-base that identifies frequently used information in the planners and caches its results in a database. The database allows components to track their computational dependencies, which makes it easy to keep track of what needs to be recomputed when the robot and task specifications change. We showed how to use all this information to quickly plan for paths. Chapter 4.
- An algorithm named **ikfast** that solves the analytic inverse kinematics equations of common kinematics. Unlike other proposed methods based on advanced mathematics, **ikfast** searches for the most computationally efficient and numerically stable solution. It can easily handle all degenerate cases. Section 4.1.
- Several grasping analyses for force-closure and caging grasps that can be used directly in manipulation. For force-closure grasps, we presented a grasp space parameterization method and a new *repeatability* measure evaluation for pruning fragile grasps. For caging grasp sets, we showed how to expand the possible ways to grasp doors to allow manipulators of low DOF to easily achieve their tasks. Section 4.2.
- A 6D kinematics reachability formulation that allows for informative samplers and an inverse reachability map for computing distribution of base placements. This formulation was the basis for a new map which we call *grasp reachability* that combines grasp sets and base placement distribution for a simple way of determining where a robot should be placed to grasp an object. Sections 4.3, 4.4, 4.5.
- Motivated the usage of convex decompositions for padding the robot for safely moving

across the environment, pruning range data, and computing a new distance metric based on the swept volumes of the robot links. Section 4.6.

- A general goal configuration generator that uses the planning knowledge-base to quickly analyze the scene and seed planners with possible goal that help achieve the task. We showed that goal generators have the most impact in a planning algorithm, and have turned the bulk of the manipulation planning research into quickly computing the goal space. Section 3.1.
- Algorithms that can quickly sample robot configurations so attached sensors clearly see the target objects of the task. We presented a data-driven object detectability extents model that allows a robot to know what regions of the object are detectable. We also presented a two-stage method of first viewing the object with a gripper camera before attempting to grasp it, we argued that this method is as fast as regular grasp planning. Sections 5.1, 5.2.
- A simple visual feedback method that uses the visibility of the object while simultaneously choosing the best grasp for that object. This allows the robot to compensate for big rotations of the object. Section 5.3.
- A completely automated extrinsic camera calibration method using the planning and visibility sampling theories developed in the manipulation framework. We also presented a new method of computing the confidence on the gathered data so a robot can determine when to stop gathering data. Chapter 6.
- For vision-based object analysis, we presented an feature-surface statistical analysis method to extract stable and discriminable features and be able to generate a set of *visual and geometric words* for the object based on the feature detectors used. These features can be used in any pose-extraction algorithm when determining what to track. Section 7.2.
- For vision-based pose recognition, we presented a novel pose extraction algorithm that directly maps image features to a set of pose hypotheses of the object. These hypotheses are used in a RANSAC algorithm to compute poses from a bin of scattered metallic objects. Section 7.3.
- The OpenRAVE environment that integrates all the components discussed in this thesis. OpenRAVE allows really fast development of planning algorithms and provides many organizational structures for the database and robot information. Appendix A.
- A set of general guidelines for defining the lowest level of manipulation autonomy. Section 2.5.

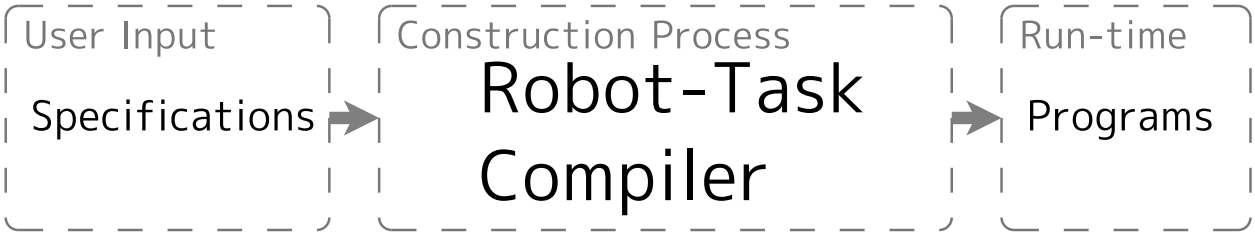


Figure 8.1: The construction process can be treated as an offline compiler that converts the specifications into run-time programs while optimizing the speed of its execution.

8.2 Future of Robotics: Robot-Task Compilers

One intriguing concept is to treat the manipulation construction process as a compiler that takes in a domain language encoding the robot and task specifications, and outputs a system of executable programs that will complete the task on the real robot (Figure 8.1). Such a domain language will stray from the traditional Turning machine thinking of encoding one execution pipeline, and instead become a high-level script that injects robot and task specific information into a preset execution framework. Because compilation is an offline process, any amount of computational power and simulation accuracy can be supplied to produce the most reliable and fastest programs.

Just like traditional compilers leverage the structure of CPU architectures for optimizations, a robot-task compiler needs to leverage predictions about the physics world in order to cache commonly occurring situations. Fortunately, engineers and scientists have created very accurate models of how materials, friction, and dynamics work, so they can be used to predict all possibilities of the environment and robot state in simulation without having to execute a real robot. Just like traditional compilers can optimize the order of instructions to hit cache lines, a robot-task compiler should make leverage the physics models to construct the most efficient structures possible to make runtime robot execution fast and reliable. A robot-task compiler can start to build up internal models of all possible scenarios a robot can get into, which allows it to analyze the program as a whole and cache predictions about states the robot can get into.

In this thesis, we have introduced and formulated a robot-task compiler as a construction phase that builds knowledge-bases and converts a set of generic algorithms into algorithms for the specific robot and task. Although the current structure is still in its infant stages, we have thoroughly discussed many of the analyses required for reliable autonomous manipulation. Future extensions of the robot-task compiler concept should first focus on a more formal specification language to completely encode a pick-and-place task. Aggressive pursuit in robot-task compilers has a potential to yield a new class of tools for programming complex

robotic manipulation programs.

Appendix A

OpenRAVE - The Open Robotics Automation Virtual Environment

We present the OpenRAVE architecture used to supplement the theoretical contributions of this thesis. OpenRAVE is an open-source cross-platform software architecture that is targeted for real-world autonomous robot applications and includes a seamless integration of simulation, visualization, planning algorithms, scripting environments, and control algorithms. The narrative in this appendix focuses on the users of the manipulation framework and guides them through understanding key advances in robotics architectures that allow for easier implementation of manipulation programs. We discuss the organization of OpenRAVE and why these choices were made in the context of users developing complex robotics systems.

One of the challenges in developing real-world autonomous robots is the need for integrating and rigorously testing high-level motion planning, perception, and control algorithms. Up to now, the thesis has covered the theory of algorithms and presented many results for manipulation scenarios while conveniently skipping the architecture and implementation details. The quality of the research results directly reflects the quality of the implementation of a system. It is necessary to develop a clean and consistent architecture that can easily handle all manipulation processes described in Chapter 2 before any meaningful statements can be made about automation.

We developed OpenRAVE with these general design goals:

- Have a plugin-based architecture that allows users to expand its functionality without having to recompile the base code. Most functionality should be offered as plugins, thus keeping the core as simple as possible.

- Offer many motion planning algorithm implementations that can be easily extended to new tasks.
- Make it easy to debug components during run-time without having to recompile or restart the entire system in order to prevent flushing of the in-memory environment state.
- Allow the OpenRAVE core to be used as a simulation environment, as a high-level scripting environment, as a kinematics and dynamics backend module for robot controllers, or as a manipulation planning black box in a distributed robotics environment.
- Allow simple offline planning database generation, storage, and retrieval.
- Support a multi-threaded environment and allow easy parallelization of planners and other functions with minimal synchronization required on the user side.

One of OpenRAVE's strongest points when compared with other planning packages is the idea of being able to apply algorithms to any scenario with very little modification. Users of OpenRAVE can concentrate on the development of planning and scripting aspects of a problem without having to explicitly manage the details of robot kinematics, dynamics, collision detection, world updates, sensor modeling, and robot control.

We first start with the OpenRAVE core design and discuss the programming models that become possible. Within the context of programming paradigms, we present a new layer of functionality that all robotics architectures should implement that goes beyond the basic kinematics, collision detection, and graphics interface requirements of classic robotics libraries. OpenRAVE provides a set of interfaces that let users modify existing functions and expand OpenRAVE-enabled modules without having to recompile OpenRAVE or deal with messy monolithic code-bases. We go through each interface's design and its usage within the entire system. Furthermore, we discuss how OpenRAVE is used with real robotics systems and motivate several key functions that make it possible for planning-enabled robots to work consistently in a continuously changing and unpredictable environment. We conclude with future work and other lessons learned in robotics architectures.

A.1 Architecture

Figure A.1 shows the interaction of the four major layers composing the architecture:

- **Core Layer.** The core is composed of a set of interface classes defining how plugins share information, and it provides an environment interface that maintains a world state, which serves as the gateway to all functions offered through OpenRAVE. The

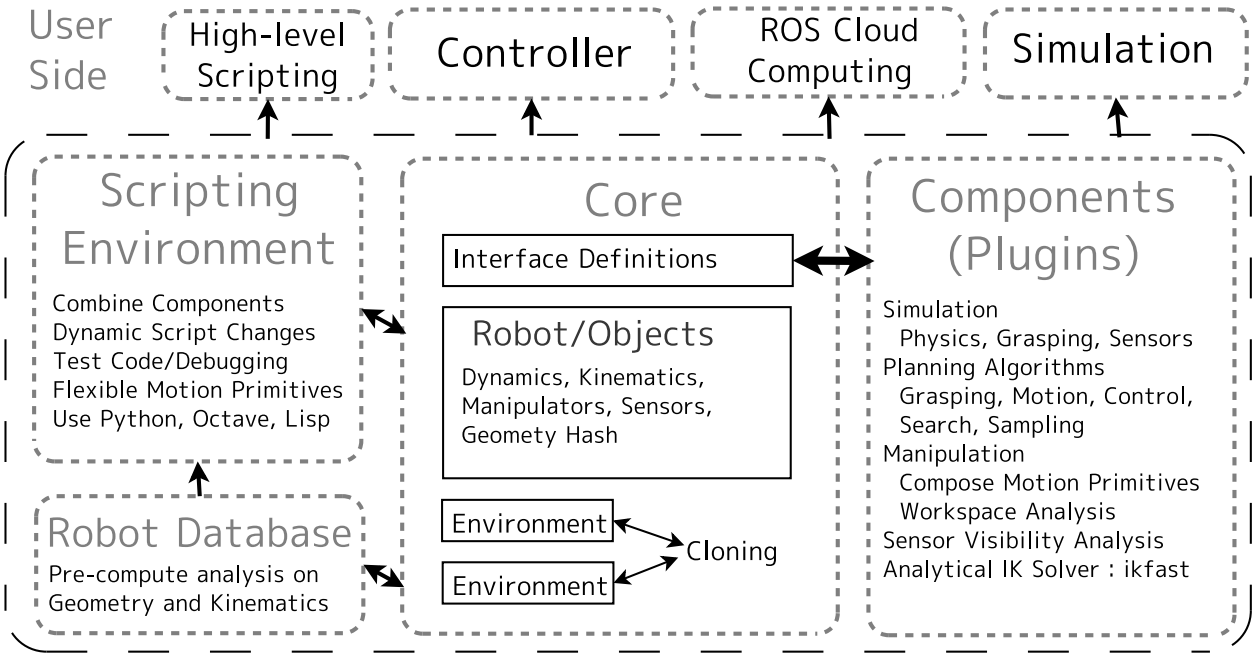


Figure A.1: The OpenRAVE Architecture is composed of four major layers and is designed to be used in four different ways.

global state manages the loaded plugins, multiple independent environments, and logging. On the other hand, the environment combines collision checkers, viewers, physics engines, the kinematic world, and all its interfaces into a coherent robotics world state.

- **Plugins Layer.** OpenRAVE is designed as a plugin-based architecture which allows to create new components to continuously improve its original specifications. Each plugin is an implementation of a standard interface that can be loaded dynamically without the need of recompiling the core. Following this design, different kind of plugins can be created such as sensors, planners, controllers or physics engines. The core layer communicates with the hardware through the plugins using more appropriate robotics packages such as Player [Gerkey et al (2001)] and Robot Operating System (ROS) [Quigley et al (2009)]. Using plugins, any planning algorithm, robot control, or sensing-based subsystem can be distributed and dynamically loaded at run-time; this distributed nature frees developers from struggling with monolithic code-bases.
- **Scripting Layer.** OpenRAVE provides scripting environments for Python and Octave/Matlab. Python communicates with the core layer directly with in-memory calls, making it extremely fast. On the other hand, the Octave/Matlab scripting protocol send commands through TCP/IP, with a plugin offering a text server on the core side.

Scripting allows real-time modifications to any aspect of the environment without requiring shutdown, making it ideal to testing new algorithms. The Python scripting is so powerful, that most of the examples and demo code are offered through it. In fact, users should treat the scripting language as an integral part of the entire system, not as a replacement to the C++ API.

- **Robot Database Layer.** Implements the planning knowledge-base covered in Chapter 4 and provides simple interfaces for its access and generation parameters. The database itself mostly consists of kinematic, quasi-static, dynamic, and geometric analyses of the robot and the task. If the robot is defined properly, then all these functions should work out of the box.

The main API is coded in C++ using the Boost C++ libraries [Dawes et al (1998-present)] as a really solid basis of low-level management and storage structures. The Boost flavors of shared pointers allow object pointers to be safely reference counted in a heavily multi-threaded environment. Shared pointers also allow handles and interfaces to be passed to the user without having to every worry about the user calling upon invalid objects or unloaded shared objects. Furthermore, OpenRAVE uses *functors* and other abstracted objects commonly seen in higher level languages to specify function pointers for sampling distributions, event callbacks, setting robot configuration state, etc. The Boost-enabled design makes the the C++ API really safe and reliable to use along with saving the users a lot of trouble doing bookkeeping on their end. Furthermore, it allows the *Resource Acquisition Is Initialization* (RAII) design pattern [Stroustrup (2001)] to be fully exploited allowing users to ignore the complexities of multi-threaded resource management.

A.1.1 Environment

The OpenRAVE state is divided into a global state and an environment-dependent state. The global state holds information common to all threads, all environments, and all interfaces. The environment state holds information and settings just for the objects and interfaces created in the environment. Environment states are independent from each other, whereas global states affect everything. At initialization, every interface is branded to one specific environment, thus making the environment the only way for it to create other interfaces and query objects. The environment manages:

- creating, loading, and destroying kinematic bodies, robots, and interfaces,
- creating plugins and the interfaces offered through them,
- a single collision checker synchronizing with the current objects,

- a single physics engine synchronizing with the current objects,
- drawing/plotting simple shapes on the 3D environment, and
- triangulation of the user-specified parts of the scene.

Locking

Because OpenRAVE is a highly multi-threaded environment, the environment state like bodies and loaded interfaces could be simultaneously accessed. In order to safely write or read the state, a user has to *lock* the environment, which prevents any other process from modifying the environment while the user is working. By using *recursive locks*, it allows a lock to be locked as many times as needed within the same thread, greatly reducing the lock management when a state changing function calls another state changing function. This safety measure helps users by always guaranteeing the environment is locked when calling global level environment functions like creating new bodies or loading scenes, regardless if the user has locked it. However, directly accessing the bodies and robots is dangerous without having the environment lock acquired.

Simulation Thread

Every environment has an internal time and a *simulation thread* directly attached to a physics engine. The thread is always running in the background and periodically steps the simulation time by a small delta for the physics engine and on all the simulation-enabled interfaces. By default, the thread is always running and can always potentially modify the environment state; therefore, users always need to explicitly lock the environment whenever playing with the internal state like modifying bodies by setting joint values or link transformations. If not careful, the controller or physics engine will overwrite them. By default, the simulation thread just sets the object positions depending on their controller inputs, but a physics engine can be attached to integrate velocities, accelerations, forces, and torques.

The simulation thread can feel like a nuisance at first, but its division of robot control into control input computation and execution greatly helps users only concentrate on feeding commands to the robot without worrying about the simulation loop. It also allows a world update to happen in one one discrete time step.

Cloning

One of the strengths of OpenRAVE is in allowing multiple environments to work simultaneously in the same process. Environment cloning allows OpenRAVE to become truly parallel by managing multiple environments and running simultaneous planners on top of

them. Because there is no shared state across the clone and the original environment, it is not possible to use an interface created from one environment in another For example, if a planner is created in one environment, it should be used only by objects in that environment. It is not possible to set a planner to plan for objects belonging to a different environment. This is because a planner will lock the environment and expect the objects it controls to be exclusively under its control.

Because the environment state is very complex, the cloning process can control how much of it gets transferred to the new clone. For example, all existing bodies and robots can be cloned, their attached controllers can be cloned, the attached viewer can be cloned, the collision checker state can be cloned, and the simulation state can be cloned. Basically the clone should be able to perform any operations that can be done with the original environment without any modification in the input parameters.

When cloning real robots, one extremely important feature that OpenRAVE cloning offers is the ability to maintain a real-time view of the world that sensors continuously update with new information. When a planner is instantiated, it should make a copy of the environment that it can exclusively control without interfering with the updating operations. Furthermore, the real-world environment possibly has robot controllers connected to real robots, having a clone gives the ability to set simulation controllers guarantees robot safety while planning; commands from a cloned environment would not accidentally send commands to the real robot.

A.1.2 Validating Plugins

Every plugin needs to export several functions to notify the core what interfaces it has and to instantiate the interfaces. When a plugin is first loaded, it is validated by the environment and its interface information is queried so the core can register the names.

There are many mechanisms in the validation process to prevent old plugins to be loaded by the core. OpenRAVE is updated frequently and all user plugins are not necessarily re-compiled when the OpenRAVE API changes. Therefore, we will encounter many cases when a plugin exports the correct functions, but does not implement the correct API. Using interfaces from plugins compiled with a mismatching The API can lead to unexpected crashes that are very difficult to debug, so it is absolutely necessary to detect this condition. One possible solution is to add version numbers to the API to enforce checking before an interface is returned from the plugin to the environment, but this method is brittle. It forces to keep track of a version number for every interface along with a global version number. Furthermore, every developer has to remember to increment the version when something even small changes, which can be easily forgotten and lead to serious errors later on.

We solve interface validation by computing a unique hash of the interface functions and members by running each interface through a C++ lexer, gathering the tokens that affect the C++ code structure, and then creating a 128bit unique MD5 hash. We create a hash for each interface definition and the environment. The hashes are hard coded into the C++ header files and can be queried by two methods: a static function returning the hash of the program calling the function, and a virtual function returning the hash the interface was compiled with. An interface is only valid if its virtual hash is equivalent to the static hash of the core environment. For a plugin to be loaded correctly, first the environment hashes have to match. If they do, then the individual interfaces checked and only matching interfaces are returned to the core, and from there dispatched to other plugins. Such consistency checks ensure that stale plugins will never be loaded.

A.1.3 Parallel Execution

Being able to execute a planner in multiple threads is important for applications that require speed and solution quality Because there is always a trade-off between solution quality and time of computation, some applications like industrial robots require the quickest and smoothest past to their destinations. Fortunately, environment cloning allows planners to create an independent environment for every thread they create, which enables them to call kinematics and collision functions in each respective thread without worrying about data corruption. Growing an RRT tree in a multi-threaded environment just requires one copy of the kd-tree structure to be maintained. The query operations mostly work with Euclidean distance on the configuration space, so are really fast. Furthermore, adding a new point takes $O(log)$ time, so it shouldn't be a bottleneck in the search process compared to collision checking. Finally, environment locking allows threads to gain exclusive access to the environment. The rule of thumb is that any interface belonging or added to the environment requires an environment lock before any of its methods can be called.

A.1.4 Exception and Fault Handling

By using the C++ Standard and Boost libraries, OpenRAVE can recover from almost all errors that a user can experience without causing the program to shutdown on the spot. Invalid pointer and out-of-range accesses are extremely dangerous because they can modify unrelated memory, which causes the program to crash at a place completely unrelated to the root cause of the problem. Avoiding such problems has been one of the the highest priorities for the design. The core always surrounds any user code coming from plugins and callbacks with try/catch blocks, this allows the core to properly handle the error and notify the user

of a problem without tearing down the environment. Because exception handling is slow, there is a fine balance of when a function should return an error code and what it should throw an exception. In OpenRAVE, exceptions should never occur in normal operation of the program, they should only be for unexpected events of the program. For example, planners failing is an expected event dependent on the current environment, so planners should return an error code with the cause of the failure rather than throw an exception. In other words, exceptions convey the structural errors of the program that point to places in the code that should be fixed by the user. The following operations should throw exceptions in OpenRAVE:

- invalid plugin or interface hashes,
- invalid commands being sent to interfaces,
- invalid arguments passed to functions,
- invalid pointers or out-of-range parts of lists are accessed,
- environment is not locked when it should be
- a resource is present when it should be,
- a math operation is not consistent with the rest of the environment,
- environment naming constraints are not maintained,
- unrecognized enumerated types are given, and
- instantiation order is not maintained.

A.1.5 Hashes for Body Structure

A new concept that came out of OpenRAVE is the idea of creating unique hashes of a body's structure. Every body has an online state that includes:

- names of the body, its links, its joints,
- link transformations, velocities, and accelerations in the world,
- and *attached* bodies.

All other information is independent of the environment and can be categorized into the kinematics, geometry, and dynamics of the body. Furthermore, robots have categories for attached sensors and manipulators. The planning knowledge-base stores all cached information about a body and a robot, so it needs an consistent way of indexing this information. Indexing by robot names is not reliable because it is very difficult to remind a user to change the name every time the body structure is changed. Therefore, OpenRAVE provides functionality to serialize the different categories of a body and create a 128-bit MD5 hash. Each

of the models in the planning knowledge-base relies on different categories of the robot. For example:

- inverse kinematics generation only uses the kinematics of a sub-chain of the robot defined by the manipulator and the grasp coordinate system,
- kinematic reachability cares about the robot geometry of the manipulator because it implicitly stores self-collision results,
- inverse reachability further uses the links connecting the base robot link to the base manipulator link,
- grasping cares about the geometry of the target body and the kinematics and geometry of the gripper,
- convex decompositions only care about the link geometry, and
- inverse dynamics cares only about the dynamics properties of each link and the kinematics.

There are several challenges to developing a consistent index across all operating systems and compilers since floating point errors could creep in when normalizing floating-point values. However, the idea of such an index could greatly help in developing a worldwide robot database that anyone can use.

A.2 Interfaces

An interface is the base element from which every possible way to expand OpenRAVE functionality is derived from. An interface always comes from a plugin and is owned by an environment. Each of the interface types attempt to package a closed set of functions that are commonly used in robotics architectures. However, it is impossible to predict what all users need and how technology will evolve, so each interface provides a flexible way to receive commends from users in the form of streams. Each stream first starts with a command name and then specifies arguments, much like a function call or remote procedure call, except the stream does not enforce any format requirements. Furthermore, each interface can contain extra annotations for holding parameters or other configuration-specific information. This allows users to set parameters in an XML file for initializing the interface. The interfaces themselves manage all the low-level library and plugin information, so users are guaranteed the interface will be always valid as long as its reference is held.

A.2.1 Kinematics Body Interface

Each kinematics body can be thought of as the base geometric object in OpenRAVE. It is composed of a collection of rigid-body links connected with joints. The kinematics are a graph of joints and links, there is no enforced hierarchy. The basic body provides:

- setting and getting joint values,
- setting and getting the transformations of all links,
- getting the velocities of each joint or link,
- self collision detection functions,
- kinematic hierarchy querying - The underlying structure of kinematics body is a list of links, not a tree. However, after some careful analysis, the parent and child links of a particular link can be extracted.
- jacobian computation - both translational and rotational,
- attaching bodies online - a necessary function for grasp planning; for example, an object is rigidly grasped by a hand requires the collision bodies to be *attached*, and
- exploring its kinematics structure.

A.2.2 Robot Interface

A robot is a special type of kinematics body that needs higher level functionality for its control and movement in the environment and its interaction with other objects. The extra functions are:

- **Manipulator** - Every robot supports a list of manipulators that describe the links the robot should use when manipulating parts of the environment. Usually manipulators are serial chains with a Base link and an End Effector link. Each manipulator is also decomposed into two parts: the arm and the hand. The hand usually makes contact with the objects while the arm transfers the hand to its destination. The manipulator class also has an optional pointer to an inverse kinematics solver providing inverse kinematics functionality.
- **Active Degrees of Freedom** - When controlling and planning for a robot, it is possible to set the degrees of freedom that should be used. For example, consider planning with a humanoid robot. There should be in easy way to specify to the planners that only the planning configuration space consists of only the right arm while keeping the rest of the joints the same. Or consider the case where we care about navigation of the humanoid robot. Here we would want to control the translation of the robot on the plane and its orientation. Perhaps we want to do footstep planning

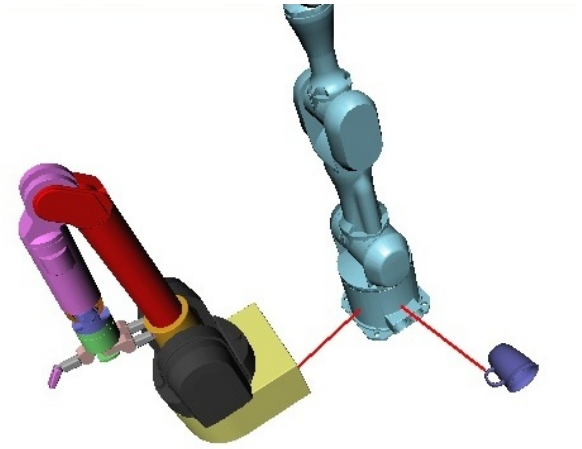


Figure A.2: Distance queries.

and also care about controlling the two legs. All this is possible with the *active degrees of freedom* feature provided by OpenRAVE.

- **Grabbing bodies** - It is possible for a robot to attach a kinematics body onto one of its links so that when the link moves, the body also moves. Because collision detection will stop being checked between the robot and the body, you could say that the body becomes a part of the robot temporarily. This functionality is necessary for manipulation planning. Whenever the robot is carrying a body, all collisions between the robot and that item should be ignored once the body has been grasped.
- **Attached Sensor** - Can attach any number of sensors to the robot's links. The sensor transformations will be completely owned by the robot. Whenever the robot link is updated, the sensor will be automatically moved to the new location.

A.2.3 Collision Checker Interface

A collision checker interface can be set for every environment, which has the checker progressively synchronize its internal world with the OpenRAVE world. The interface provides a wide range of collision and proximity testing functions along with testing primitives like rays. All collisions functions take in a collision report, which can fill in extra information about contact points, number of collisions, and collided links. OpenRAVE offers a set of global options to set the state of the collision checker. For example, it is possible to request more precise computationally intensive information like distance to obstacles (Figure A.2), but such an option is not turned on as default. New collision checkers can be easily set onto an existing environment by a single function call, making collision checker performance

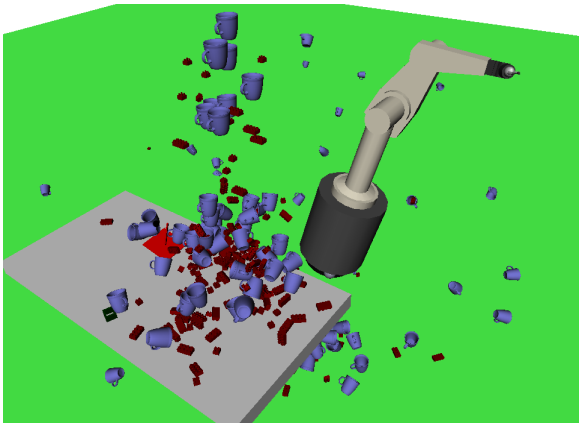


Figure A.3: Physics simulations.

comparison very simple. In fact, this allows bugs to be caught between collision checkers because a user can quickly set another checker to confirm the behavior.

A.2.4 Physics Engine Interface

Similar to collision checkers, a physics engine can be set for the environment to correctly move the objects in the simulation thread (Figure A.3). Physics engines allow objects to maintain a velocity and acceleration and similarly maintain a separate internal world to the openrave world. The physics engine only modifies the state of the world inside the simulation time step.

A.2.5 Controller Interface

In order to control certain robot hardware within the OpenRAVE environment, a controller interface communicating with the hardware-specific libraries is created and attached onto an existing robot. All commands given to move the robot should be sent through the controller. The default format for a command is a timed trajectory of joint values, velocities, and expected torques that the robot is expected to hit. Just like physics engines, a controller based on simulation results should update the robot during the simulation step. If a controller is based on a real hardware, it should always lock the environment and update the robot with the current encoder values. The controller interface allows the user to query the current state of the robot and its completion of the commands sent.

A.2.6 Inverse Kinematics Interface

Each inverse kinematics solver is defined on a subset of joints of a robot specified by the manipulator. Given the position in the workspace that an end effector should go to, an inverse kinematics solver should find the joint configuration to take that end-effector there. Because it is common for a solution to have a null space, the solver exposes functions that can query the null space for a particular solution or all solutions. Several parameterizations of the workspace goal are supported:

- **6D pose** - translation and rotation defining the full coordinate system of the end-effector.
- **3D translation** - end-effector should hit a particular point
- **3D rotation** - end-effector rotation,
- **3D look at direction** - a defined direction on the end-effector coordinate system should look at a point.
- **4D ray** - a defined ray on the end-effector should align with a destination ray.

Each of these parameterizations is important in manipulation and other workspace analyses. Furthermore, the actual inverse kinematics querying can support several checks before a solution is really considered valid:

- **Environment Collisions** - Will check environment collisions with the robot,
- **Self Collisions** - Will check the self-collision of the robot,
- **Joint Limits** - Will check the joint limits of the robot,
- **Custom Filter** - Will use a custom function provided by the user to validate or *modify* the solution.

A.2.7 Planner Interface

In OpenRAVE, the basic purpose of a planner is to find a path starting at some initial configuration that reaches a goal condition while satisfying problem-specific constraints. All planners are assumed to be geometric in nature, so that they can be adapted to different situations. In order to quantify all the possible parameterizations and configurations of a planner, OpenRAVE introduces a **planner parameters** structure that holds everything specific to the problem. The planning algorithm itself should hold everything specific to the search process of the configuration space defined by the **planner parameters**. As default behavior, planners plan for robots and they use the robot's active degrees of freedom for the configuration space; however, the planner parameters structure can easily define a

new configuration space of the robot that is independent of the robot. This gives the user flexibility in choosing the configuration space. The usage of a planner is simple:

1. Acquire a planner pointer from environment.
2. Fill a **planner parameters** structure defining the instance of the problem. The structure has many fields for describing planning entities like start position, goal condition, and the distance metric. Try to use these fields as much as possible. Later on, this will allow users to easily swap planners without having to change the parameters structure much.
3. Initialize the planner using the robot and planner parameters. This also resets any previous information the planner had stored.
4. Plan for a path passing in a trajectory pointer for the output. If the function returns true, then the trajectory will be filled with the geometric solution in the active DOF configuration space of the robot. It is possible to preserve the previous search space for the planner if changing the goal conditions and reusing the previous computations.

Planning Parameters

All the information defining a planning problem should be specified in a **planner parameters** structure, which attempts to cover most of the common data mentioned in Chapter 3. The fields to set are:

- **Cost Function** - takes in a configuration and outputs a single value showing the cost of being in this region.
- **Goal Function** - takes in a configuration and outputs a value showing proximity to the goal.
- **Distance Metric** - distance between two configurations.
- **Constraint Function** - takes in a previous and new configuration and projects the new configuration to satisfy problem specific constraints. Returns false if new configuration should be rejected.
- **Sample Function** - Samples a random configuration for exploring the configuration space.
- **Sample Neighbor Function** - Samples the neighborhood of an input configuration, controls how far samples can be using the distance metric.
- **Sample Goal Function** - Samples a random goal configuration.
- **Set/Get State Functions** - Sets and gets the configuration state.

- **State Difference Function** - Used to find the difference between two states. This function properly takes into account identifications for joints without any limits.
- **Goal Configurations** - Explicit seeding of the goal configurations when initializing the planner.
- **Lower/Upper Limits** - Very rough limits of each DOF of the configuration space used to normalize sampling and verify configurations.
- **Resolution** - The resolution to check line collisions and used for other discretization factors dependent on the space.
- **Step Length** - The step length for discretizing the configuration space when searching.
- **Max Iterations/Time** - Controls the maximum number of iterations or time that the planners should compute until it gives up.
- **Path Optimization Parameters** - Special parameters that will be used in post-processing the output paths to smooth them of any irregularities.
- **Check Self Collisions** - If the planner should check self-collisions of the robot for every configuration. Some configuration spaces, like navigation, might not modify the internal joints of the robot, so this step could speed up planning.

However there are many different types of inputs to a planner, so it is impossible to cover everything with one class. Therefore, **planner parameters** has a very flexible and safe way to extend its parameters without destroying compatibility with a particular planner or user of the planner. This is enabled by the serialization to XML. Using XML as a medium, it is easy to exchange data across different derivations of **planner parameters** without much effort or incompatibilities. Because of these serialization capabilities, it becomes possible to pass in the planner problem across the network or a different thread. Furthermore, a base planner can read in any XML structure and ignore the fields that it does not recognize, meaning that users can re-use old planners without having to tune their parameters structures.

Path Optimization

Path smoothing/optimization is regarded as a post-processing step to planners. Path optimization algorithms take in an existing trajectory and filter it using the existing constraints of the planner. In fact, functionality there is no difference between a *path optimization* planner and a regular planner besides the fact that a trajectory is used as input. Because a planner already has a trajectory as an argument to its plan path method, supporting path optimization does not cause any major API changes to the infrastructure.

The **planner parameters** structure reflects what optimization algorithm to use for post processing the trajectory. By default, this is the linear shortcut method; however, many

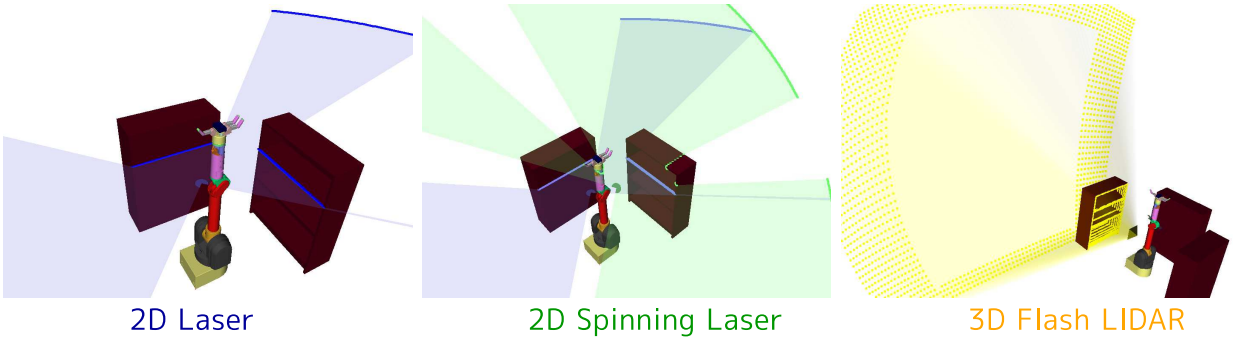


Figure A.4: Several simulated sensors.

different optimization algorithms exist, each with their own sets of parameters.

This type of planner post-processing actually allows users to chain planners in the same way that filters are chained for sensing data. Of course, users can continue to smooth in planners without relying on this framework. However, explicit control of path smoothing allows custom parameters to be easily specified.

A.2.8 Trajectory Interface

A trajectory is a path between a set of configuration space points that can be annotated with velocities, accelerations, expected torques, and the affine transformation of the robot. The OpenRAVE trajectory class supports several interpolation methods and provides users with an easy interface to query the values along a trajectory to feed into a controller.

A.2.9 Sensor Interface

A sensor measures physical properties from the environment and converts them to data. Each sensor is associated with a particular position in space, has a geometry with properties defining the type of sensor, and can be queried for sensor data. Similar to Player [Gerkey et al (2001)], OpenRAVE attempts to hard code the interfaces to the sensors so that users have a simpler time inter-operating with each other. Figure A.4 shows some of the simulated laser types provided.

A.2.10 Sensor System Interface

When connecting OpenRAVE to a perception system, several outside modules will start populating the environment with their own sensor measurements. Working with a real perception system is very difficult because as soon as some obstacle blocks the sensor, the

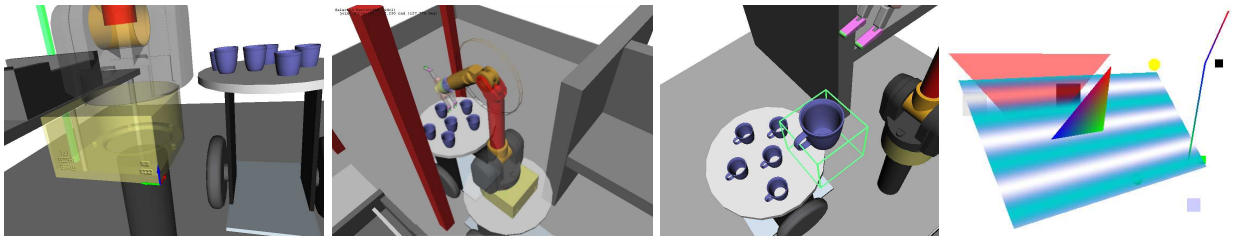


Figure A.5: Simple functions a viewer should support.

system would not be able to perceive the target object anymore and might decide to remove it from the environment. The sensor system interface allows control of what objects should be *released* from the control of the perception system, what objects should be *locked* so the system cannot destroy them, and what objects should be *destroyed*. Every kinematics body has a structure that specifically manages the state for the object and allows planners and other systems to mark objects as being used, so they shouldn't be deleted. Furthermore, as a robot grasps an object, the perception system should completely release it or hand it over to a new perception system because the object's state is now controlled by the gripper.

A.2.11 Viewer Interface

Viewer is responsible for showing the current state of its attached environment. Because OpenRAVE is a scripting-oriented environment, not much functionality is expected from viewers beyond being able to render the objects, plot simple primitives, and move objects around the environment (Figure A.5). Each viewer should support a set of geometric primitive plotting functions to allow users to annotate their scene with meaningful information about their task. In fact by adding too much functions to a viewer, users might get the wrong impression that a viewer is the recommended way to interact with the OpenRAVE world, which is the wrong message. It is much more critical to keep exposed functionality to a minimum in order to speed up rendering and not have to maintain too much unnecessary code.

A.2.12 Modular Problem Interface

The modular problem interface represents a chunk of code that a user would want to execute in the context of the OpenRAVE environment to control a demo or expand functionality. Each modular problem provides a **main** function that is executed when the problem is loaded into the environment. Furthermore, the problem has access to the internal simulation thread

so it can define its own dynamics behaviors. A lot of problems wrap up planners and other neat little functions and provide their functionality as string commands supported through the base interface class. For example a popular modular problem interface is the **BaseManipulation** problem. At startup, users instantiate it with the name of the robot that the base manipulation should control. One of the functions it offers is to move the end effector of a manipulator on the robot to a specified location. The **BaseManipulation** problem offers a command that accepts a set of 6D positions of the end effector, calls their inverse kinematics, creates the necessary goal samplers, and calls a default randomized planners to find a path.

As OpenRAVE grows into a more mature environment, users will always find the need for new interfaces and new functionality. By supporting a generic modular problem interface, it allows users to quickly code up their ideas and show a proof of concept without being bound by the specific interfaces.

A.3 Working with Real Robots

Running planners on real robots doesn't always go as smoothly as in simulation. The problems occur because the real robot joint values are used to set the environment joint values, which means that the robot can initially start in environment or self collision. Furthermore, there are always errors in the robot localization, environment modeling, and perception system, so executing non-collision path could potentially graze obstacles or the target object along the way. In practice, such errors are fatal to the application and execution of any planning algorithm. Fortunately, experience has provided with two very effective ad hoc methods to resolve almost all of these problems: padding and jittering. Although we spent little time on them in the manipulation planning sections, they can easily turn in a 5% success rate of task completion into 100%. Any OpenRAVE planners use these two the full extent.

A.3.1 Padding

Obstacles and the robot need to be padded with at least 5mm-10mm around the surface before computing collision queries. Section 4.6.1 motivated the use of convex decompositions for uniformly padding the surfaces. However, padding cannot be applied all the time in all situations. Because joint encoders are very accurate, checking self-collisions with a padded mesh is meaningless most of the time; in fact, it could cause the robot to get into random collisions that can affect the feasibility of the plan.

In this thesis we spent a great deal of effort showing why target objects should be treated differently from obstacles the robot is trying to avoid; the robot actually needs to get close and make contact with the targets. If making grasp sets with a padding object, it could cause the robot to hallucinate contact points that don't exist in reality, which could yield very bad grasps. The objects should be their original size when creating grasp tables and approaching the final grasp to the target object. Therefore, padded objects should be used only when avoiding them is clearly the goal of the plan. The target object should be padded only in the first stage of grasp planning (Section 3.3) where the gripper gets close, but does not approach the target. Sampling visibility should also work with the original objects. Any other situation requires obstacle be padded.

A.3.2 Jittering

The most effective way of moving the robot out of grazing environment or self collisions is to randomly move the robot joints a small distance and see if the robot got out of collision. Although many papers have been written on computing proximal distance and finding the shortest path to get the robot out of collision, they tend to be slow. Furthermore, the straight line in configuration space between the initial colliding configuration and a close non-colliding configuration is usually a very good approximation of the shortest path to take. Jittering first starts searching within a small ball of the current configuration and slowly increases that ball until a reasonable maximum limit on the configuration distance is reached.

The distance metric presented in Section 4.6.3 becomes very important in determining what joints affect more volume and what joints do not. By not having a well calibrated distance metric, small steps in configuration space could move the base joint enough distance to account for a 5mm-10mm jumps in the end effector. Such huge jumps will most likely cross obstacles and really get the robot in trouble if it tries to move across the obstacles. Therefore, we have to rely on a distance metric that takes into account the average swept volume of each joint.

A.4 Discussion

OpenRAVE provides an environment for testing, developing, and deploying manipulation planning algorithms in real-world autonomous manipulation applications. The biggest challenge is developing an integrated architecture that allows for rapid development, powerful scripting, and the combination of many modules that inter-operate with many libraries. Attempting to support everything could spread an architecture thin and make it lose its value;

therefore, we make it easy to connect OpenRAVE to other systems through plugins. The plugin architecture allows OpenRAVE to solely focus on geometric and kinematic the analyses. This focus allows is to be easily integrated into existing robotics systems that concentrate on other tasks like low-level control, message protocols, perception, and higher-level intelligence systems. We covered the final OpenRAVE architecture and a lot of the decisions that went into its design. OpenRAVE already supports a plethora of functions like environment cloning, geometry hashes, planner parameters, advanced exception handling, and grabbing bodies that other robotics architectures are just beginning to realize the importance of.

One of the earliest and best decisions when starting to develop OpenRAVE was to make it open-source. Its open nature has allowed a community to flourish within it. Furthermore, a lot of the community has helped uncover many issues and bottlenecks with previous OpenRAVE versions. This has allowed the architecture to naturally evolve as user demand increased. In fact, not having such continuous feedback from a community of over a hundred researchers would not have allowed OpenRAVE to grow so far and become as popular as it is today. Because we believe in supporting commercial ventures, we release OpenRAVE code in the GNU Lesser General Public License and the Apache License, Version 2.0. The core is protected by the LGPL so we can keep track of changes where the hope is to maintain a single OpenRAVE distribution that can satisfy everyone. The LGPL requires any changes to the core to be made public if used in products. All scripts and examples and released under the much less restrictive Apache License, so users have the freedom to modify it in any way they want without any worry.

Many people have argued that releasing research prematurely as open-source could help other parties take advantage of the work, which is beneficial for the original author. Such statements completely miss the point of research and sharing information, and our hope is that the success of OpenRAVE can show to the robotics community that sharing code can greatly benefit the original author's goals and spur more growth in the community. Making code open source allows research to be used in many more places and gives it a lot of exposure.

OpenRAVE will continue to be a community effort to organize manipulation planning research. Eventually we hope to setup official planning benchmarks to easily test planning, collision, physics, and other algorithms in the context of manipulation tasks.

References

Albu-Schaffer A, Haddadin S, Ott C, Stemmer A, Wimbock T, Hirzinger G (2007) The dlr lightweight robot: design and control concepts for robots in human environments. Industrial Robot: An International Journal 34(5):376–385

An CH, Atkeson CG, Hollerbach JM (1988) Model-Based Control of a Robot Manipulator. MIT Press, Cambridge, Massachusetts

Ausubel J (2009) The population delusion: Ingenuity wins every time. New Scientist 203(2727):38–39

Baker C, Ferguson D, Dolan J (2008) Robust mission execution for autonomous urban driving. In: 10th International Conference on Intelligent Autonomous Systems

Barrett-Technologies (1990-present) Barrett whole arm manipulator. URL http://www.barrett.com

Bay H, Ess A, Tuytelaars T, Gool LV (2008) Surf: Speeded up robust features. Computer Vision and Image Understanding (CVIU) 110(3):346–359

Belongie, Serge (1999-present) Rodrigues' rotation formula. URL http://mathworld.wolfram.com/RodriguesRotationFormula.html

Berenson D, Diankov R, Nishiwaki K, Kagami S, Kuffner J (2007) Grasp planning in complex scenes. In: Proceedings of IEEE-RAS International Conference on Humanoid Robots (Humanoids)

Berenson D, Srinivasa S, Ferguson D, , Kuffner J (2009a) Manipulator path planning on constraint manifolds. In: Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)

Berenson D, Srinivasa S, Ferguson D, Collet A, Kuffner J (2009b) Manipulator path planning with workspace goal regions. In: Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)

Bertram D, Kuffner J, Dillmann R, Asfour T (2006) An integrated approach to inverse kinematics and path planning for redundant manipulators. In: Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)

Bolles RC, Horaud P, Hannah MJ (1983) 3dpo: A three-dimensional part orientation system. In: IJCAI, pp 1116–1120

Borgefors G, Strand R (2005) An approximation of the maximal inscribed convex set of a digital object. In: Image Analysis and Processing ICIAP

Bouguet JY (2002) http://www.vision.caltech.edu/bouguetj/calib$_doc$

Bradski G, Kaehler A (2008) Learning OpenCV. O'Reilly Media Inc.

Brock O, Kuffner J, Xiao J (2008) Motion for manipulation tasks. Handbook of Robotics (O Khatib and B Siciliano, Eds)

Cameron S (1985) A study of the clash detection problem in robotics. In: In Int. Conf. Robotics Automation, pp 488–493

Chestnutt J (2007) Navigation planning for legged robots. PhD thesis, Robotics Institute, Carnegie Mellon University

Chestnutt J, Michel P, Nishiwaki K, Kuffner J, Kagami S (2006) An intelligent joystick for biped control. In: Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)

Chew LP, Kedem K (1993) A convex polygon among polygonal obstacles: Placement and high-clearance motion. Computational Geometry: Theory and Applications 3(2):59–89

Chia KW, Cheok AD, Prince S (2002) Online 6 dof augmented reality registration from natural features. In: Proc. of the Intl. Symp. on Mixed and Augmented Reality (ISMAR)

Chum O (2005) Two-view geometry estimation by random sample and consensus. PhD thesis, Czech Technical University

Ciocarlie M, Goldfeder C, Allen P (2007) Dimensionality reduction for hand-independent dexterous robotic grasping. In: Proceedings of the IEEE International Conference on Intelligent Robots and Systems (IROS)

Ciocarlie MT, Allen PK (2009) Hand posture subspaces for dexterous robotic grasping. Int J Rob Res 28(7):851–867

Curless B, Curless B, Curless B, Curless B, Levoy M, Levoy M, Levoy M, Levoy M (1995) Better optical triangulation through spacetime analysis. In: In ICCV, pp 987–994

Datta A, Kim J, Kanade T (2009) Accurate camera calibration using iterative refinement of control points. In: Workshop on Visual Surveillance (VS), 2009 (held in conjunction with ICCV).

David P, DeMenthon D (2005) Object recognition in high clutter images using line features. In: International Conference on Computer Vision (ICCV)

David P, Dementhon D, Duraiswami R, Samet H (2004) SoftPOSIT: Simultaneous pose and correspondence determination. International Journal of Computer Vision 59(3):259–284

Dawes B, Abraham D, Rivera R (1998-present) Boost c++ libraries. URL http://www.boost.org/

Diankov R, Kuffner J (2007) Randomized statistical path planning. In: Proceedings of the IEEE International Conference on Intelligent Robots and Systems (IROS)

Diankov R, Kuffner J (2008) Openrave: A planning architecture for autonomous robotics. Tech. Rep. CMU-RI-TR-08-34, Robotics Institute, URL http://openrave.programmingvision.com

Diankov R, Ratliff N, Ferguson D, Srinivasa S, Kuffner J (2008a) Bispace planning: Concurrent multi-space exploration. In: Proceedings of Robotics: Science and Systems (RSS)

Diankov R, Srinivasa S, Ferguson D, Kuffner J (2008b) Manipulation planning with caging grasps. In: Proceedings of IEEE-RAS International Conference on Humanoid Robots (Humanoids)

Diankov R, Kanade T, Kuffner J (2009) Integrating grasp planning and visual feedback for reliable manipulation. In: Proceedings of IEEE-RAS International Conference on Humanoid Robots (Humanoids)

Divvala SK, Hoiem D, Hays JH, Efros AA, Hebert M (2009) An empirical study of context in object detection. In: IEEE Computer Society Conference on Computer Vision and Pattern Recognition

Drake SH (1989) Using compliance in lieu of sensory feedback for automatic assembly. PhD thesis, Department of Mechanical Engineering, Massachusetts Institute of Technology

Dupont L, Hemmer M, Petitjean S, Schömer E (2007) Complete, exact and efficient implementation for computing the adjacency graph of an arrangement of quadrics. In: ESA'07: Proceedings of the 15th annual European conference on Algorithms, pp 633–644

Eberly D (2001) Intersection of convex objects: The method of separating axes. Tech. rep., Geometric Tools, LLC, URL http://www.geometrictools.com/

Escande A, Miossec S, Kheddar A (2007) Continuous gradient proximity distance for humanoids free-collision optimized-postures. In: Proceedings of IEEE-RAS International Conference on Humanoid Robots (Humanoids)

Exact-Dynamics-BV (1991-present) Manus arm. URL http://www.exactdynamics.nl

Ferguson D (2006) Single Agent and Multi-agent Path Planning in Unknown and Dynamic Environments. PhD thesis, Carnegie Mellon University

Ferguson D, Stentz A (2004) Delayed D*: The Proofs. Tech. Rep. CMU-RI-TR-04-51, Carnegie Mellon Robotics Institute

Ferguson D, Stentz A (2005) The Field D* Algorithm for Improved Path Planning and Replanning in Uniform and Non-uniform Cost Environments. Tech. Rep. CMU-RI-TR-05-19, Carnegie Mellon School of Computer Science

Ferguson D, Stentz A (2006) Anytime RRTs. In: Proceedings of the IEEE International Conference on Intelligent Robots and Systems (IROS)

Fikes R, Nilsson N (1971) Strips: a new approach to the application of theorem proving to problem solving. Artificial Intelligence 2:189–208

Fischler MA, Bolles RC (1981) Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography. Communications of the ACM 24(6):381–395

Gerkey B, Vaughan RT, Stoy K, Howard A, Sukhatme GS, Mataric MJ (2001) Most Valuable Player: A Robot Device Server for Distributed Control. In: Proceedings of the IEEE International Conference on Intelligent Robots and Systems (IROS)

Goad C (1983) Special purpose, automatic programming for 3d model-based vision. Proc DARPA Image Understanding Workshop

Goerick C, Bolder B, Janssen H, Gienger M, Sugiura H, Dunn M, Mikhailova I, Rodemann T, Wersing H, Kirstein S (2007) Towards incremental hierarchical behavior generation for humanoids export. In: Proceedings of IEEE-RAS International Conference on Humanoid Robots (Humanoids)

Gold S, Rangarajan A, ping Lu C, Pappu S, Mjolsness E (1998) New algorithms for 2d and 3d point matching: Pose estimation and correspondence. Pattern Recognition 31(8):1019–1031

Goldfeder C, Allen P, Lackner C, Pelossof R (2007) Grasp planning via decomposition trees. In: Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)

Gordon I, Lowe D (2006) What and where: 3d object recognition with accurate pose. In: Toward Category-Level Object Recognition, pp 67–82

Gorski KM, Hivon E, Banday AJ, Wandelt BD, Hansen FK, Reinecke M, Bartelmann M (2005) HEALPix: a framework for high-resolution discretization and fast analysis of data distributed on the sphere. The Astrophysical Journal 622:759–771

Gravot F, Haneda A, Okada K, Inaba M (2006) Cooking for humanoid robot, a task that needs symbolic and geometric reasonings. In: Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)

Gremban KD, Ikeuchi K (1993) Appearance-based vision and the automatic generation of object recognition programs. In: Three-Dimensional Object Recognition Systems, Elsevier Science Publishers, B.V.

Gremban KD, Ikeuchi K (1994) Planning multiple observations for object recognition. International Journal of Computer Vision 12(1):137–172

Griffin G, Holub A, Perona P (2007) Caltech-256 object category dataset. Tech. Rep. 7694, California Institute of Technology, URL http://authors.library.caltech.edu/7694

Grimson W, Lozano- Perez T (1985) Recognition and localization of overlapping parts from sparse data in two and three dimensions. In: IEEE Conf. on Robotics and Automation, pp 61–66

Gu L, Kanade T (2008) A generative shape regularization model for robust face alignment. In: Proceedings of The 10th European Conference on Computer Vision

Harada K, Morisawa M, Miura K, Nakaoka S, Fujiwara K, Kaneko K, Kajita S (2008) Kinodynamic gait planning for full-body humanoid robots. In: Proceedings of the IEEE International Conference on Intelligent Robots and Systems (IROS)

Haralick BM, Lee CN, Ottenberg K, Nolle M (1994) Review and analysis of solutions of the three point perspective pose estimation problem. International Journal of Computer Vision 13(3):331–356

Hartley R, Zisserman A (2000) Multiple View Geometry in Computer Vision. Cambridge University Press

Hauser K, Ng-Thow-Hing V (2010) Fast smoothing of manipulator trajectories using optimal bounded-acceleration shortcuts. In: Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)

Hauser K, Bretl T, Latombe J (2008) Motion planning for legged robots on varied terrain. International Journal of Robotics Research 27(11-12):1325–1349

Hollinger G, Ferguson D, Srinivasa S, Singh S (2009) Combining search and action for mobile robots. In: Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)

Ihler A, Mandel M (2003) http://www.ics.uci.edu/ ihler/code

III FLH, Shimada K (2009) Morphological optimization of kinematically redundant manipulators using weighted isotropy measures. In: Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)

Jain A, Kemp CC (2008) Behaviors for robust door opening and doorway traversal with a force-sensing mobile manipulator. In: Proceedings of the Manipulation Workshop in Robotics Science And Systems

Kadir T, Zisserman A, Brady M (2004) An affine invariant salient region detector. In: In Proc of the 8th European Conference on Computer Vision, pp 345–457

Kaneko K, Kanehiro F, Kajita S, Hirukawa H, Kawasaki T, Hirata M, Akachi K, Isozumi T (2004) Humanoid robot hrp-2. In: Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)

Kazemi M, Gupta K, Mehrandezh M (2009) Global path planning for robust visual servoing in complex environments. In: Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)

Kim DJ, Lovelett R, Behal A (2009) Eye-in-hand stereo visual servoing of an assistive robot arm in unstructured environments. In: Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)

Kim YJ, Varadhan G, Lin MC, Manocha D (2003) Fast swept volume approximation of complex polyhedral models. In: ACM Symposium on Solid Modeling and Applications

Knuth DE (1973) Fundamental Algorithms, The Art of Computer Programming, vol 1, 2nd edn. Addison-Wesley

Kojima M, Okada K, Inaba M (2008) Manipulation and recognition of objects incorporating joints by a humanoid robot for daily assistive tasks. In: Proceedings of the IEEE International Conference on Intelligent Robots and Systems (IROS)

Kragic D, Miller AT, Allen PK (2001) Real-time tracking meets online grasp planning. In: Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)

Kuffner J, LaValle S (2000) RRT-Connect: An Efficient Approach to Single-Query Path Planning. In: Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)

Kuffner J, Nishiwaki K, Kagami S, Inaba M, Inoue H (2003) Motion planning for humanoid robots. In: Proceedings of the International Symposium on Robotics Research (ISRR)

LaValle S (2006) Planning Algorithms. Cambridge University Press (also available at http://msl.cs.uiuc.edu/planning/)

LaValle S, Kuffner J (2000) Rapidly-exploring random trees: Progress and prospects. In: Robotics: The Algorithmic Perspective. 4th Int'l Workshop on the Algorithmic Foundations of Robotics (WAFR)

LaValle S, Kuffner J (2001) Randomized kinodynamic planning. International Journal of Robotics Research 20(5):378–400

Lepetit V, Vacchetti L, Thalmann D, Fua P (2003) Fully automated and stable registration for augmented reality applications. In: Proc. of the Intl. Symp. on Mixed and Augmented Reality (ISMAR)

Li Y, Gu L, Kanade T (2009) A robust shape model for multi-view car alignment. In: IEEE Conference on Computer Vision and Pattern Recognition

Li Z, Canny J (1993) Nonholonomic Motion Planning. Kluwer, Boston, MA

Likhachev M, Ferguson D (2009) Planning long dynamically feasible maneuvers for autonomous vehicles. International Journal of Robotics Research 28(8):933–945

Low KH, Dubey RN (1987) A comparative study of generalized coordinates for solving the inverse-kinematics problem of a 6r robot manipulator. International Journal of Robotics Research 5(4):69–88

Lowe D (2004) Distinctive image features from scale-invariant keypoints. International Journal of Computer Vision 60(2):91–110

Luo Z, Tseng P (1992) On the convergence of coordinate descent method for convex differentiable minimization. Journal of Optimization Theory and Applications 72(1):7–35

Malisiewicz T, Efros AA (2008) Recognition by association via learning per-exemplar distances. In: CVPR

Manocha D, Zhu Y (1994) A fast algorithm and system for the inverse kinematics of general serial manipulators. In: Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)

Marthi B, Russell SJ, Wolfe J (2008) Angelic Hierarchical Planning: Optimal and Online Algorithms. In: ICAPS

Mason MT (2001) Mechanics of Robotic Manipulation. MIT Press, Cambridge, MA

Matousek J (1999) Geometric Discrepancy: an Illustrated Guide. Springer, Berlin

McMillen C, Rybski P, Veloso MM (2005) Levels of multi-robot coordination for dynamic environments. In: Lynne E Parker ACS Frank E Schneider (ed) Multi-Robot Systems. From Swarms to Intelligent Automata, vol 3, Springer, pp 53–64

Michel P (2008) Integrating perception and planning for humanoid autonomy. PhD thesis, Robotics Institute, Carnegie Mellon University

Mikolajczyk K, Schmid C (2002) An affine invariant interest point detector. In: ECCV '02: Proceedings of the 7th European Conference on Computer Vision-Part I, pp 128–142

Mikolajczyk K, Schmid C (2004) Scale & affine invariant interest point detectors. Int J Comput Vision 60(1):63–86

Mikolajczyk K, Zisserman A, Schmid C (2003) Shape recognition with edge-based features. In: Proceedings of the British Machine Vision Conference, vol 2, pp 779–788

Mikolajczyk K, Tuytelaars T, Schmid C, Zisserman A, Matas J, Schaffalitzky F, Kadir T, Gool LV (2005) A comparison of affine region detectors. International Journal of Computer Vision 65(1/2):43–72

Miller AT (2001) Graspit!: A versatile simulator for robotic grasping. PhD thesis, Department of Computer Science, Columbia University

Moreels P, Perona P (2005) Evaluation of features detectors and descriptors based on 3d objects. In: International Conference on Computer Vision (ICCV)

Morel J, GYu (2009) Asift: A new framework for fully affine invariant image comparison. SIAM Journal on Imaging Sciences 2

Morris AC (2007) Robotic introspection for exploration and mapping of subterranean environments. PhD thesis, Robotics Institute, Carnegie Mellon University

Neuronics (2001-present) Katana arm. URL http://www.neuronics.ch/cms_en

Niskanen S, stergrd PRJ (2003) Cliquer user's guide, version 1.0. Tech. Rep. T48, Communications Laboratory, Helsinki University of Technology

Okada K, Ogura T, Haneda A, Fujimoto J, Gravot F, Inaba M (2004) Humanoid motion generation system on hrp2-jsk for daily life environment. In: International Conference on Machantronics and Automation (ICMA)

Okada K, Kojima M, Sagawa Y, Ichino T, Sato K, Inaba M (2006) Vision based behavior verification system of humanoid robot for daily environment tasks. In: Proceedings of IEEE-RAS International Conference on Humanoid Robots (Humanoids)

Okada K, Tokutsu S, Ogura T, Kojima M, Mori Y, Maki T, Inaba M (2008) Scenario controller for daily assistive humanoid using visual verification. In: Proceedings of the International Conference on Intelligent Autonomous Systems (IAS)

Oriolo G, Vendittelli M, Freda L, Troso G (2004) The SRT Method: Randomized strategies for exploration. In: Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)

Pantofaru C, Hebert M (2007) A framework for learning to recognize and segment object classes using weakly supervised training data. In: British Machine Vision Conference

Pelossof R, Miller A, Allen P, Jebara T (2004) An svm learning approach to robotic grasping. In: Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)

Peters J, Schaal S (2008) Learning to control in operational space. International Journal of Robotics Research 27(2):197–212

Pham MT, Cham TJ (2007) Online learning asymmetric boosted classifiers for object detection. In: In Proc. IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'07)

Prats M, Martinet P, del Pobil AP, Lee S (2007a) Vision/force control in task-oriented grasping and manipulation. In: Proceedings of the IEEE International Conference on Intelligent Robots and Systems (IROS)

Prats M, Sanz P, del Pobil A (2007b) Task-oriented grasping using hand preshapes and task frames. In: Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)

Prats M, Martinet P, del Pobil A, Lee S (2008a) Robotic execution of everyday tasks by means of external vision/force control. Journal of Intelligent Service Robotics 1(3):253–266

Prats M, Sanz PJ, del Pobil AP (2008b) A sensor-based approach for physical interaction based on hand, grasp and task frames. In: Proceedings of the Manipulation Workshop in Robotics Science And Systems

Quigley M, Gerkey B, Conley K, Faust J, Foote T, Leibs J, Berger E, Wheeler R, Ng A (2009) (ros): an open-source robot operating system. In: ICRA Workshop on Open Source Software in Robotics

Raghavan M, BRoth (1990) A general solution for the inverse kinematics of all series chains. In: Proc. of the 8th CISM-IFTOMM Symposium on Robots and Manipulators

Ratcliff JW (2006) http://code.google.com/p/convexdecomposition/

Rimon E (1999) Caging planar bodies by one-parameter two-fingered gripping systems. The International Journal of Robotics Research 18:299–318

Rimon E, Blake A (1986) Caging 2d by one-parameter two-fingered gripping systems. In: Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)

Russell S, Norvig P (1995) Artificial Intelligence: A Modern Approach. Prentice Hall, Inc.

Rusu RB, Marton ZC, Blodow N, Dolha M, Beetz M (2008) Towards 3d point cloud based object maps for household environments. Robotics and Autonomous Systems Journal (Special Issue on Semantic Knowledge)

Rybski P, Veloso MM (2009) Prioritized multihypothesis tracking by a robot with limited sensing. EURASIP Journal on Advances in Signal Processing 2009:138–154

Rybski PE, Roumeliotis S, Gini M, Papanikopoulos N (2008) Appearance-based mapping using minimalistic sensor models. Autonomous Robots 24(3):159–167

Saidi F, Stasse O, Yokoi K, Kanehiro F (2007) Online object search with a humanoid robot. In: Proceedings of IEEE-RAS International Conference on Humanoid Robots (Humanoids)

Schaffalitzky F, Zisserman A (2002) Multi-view matching for unordered image sets, or how do i organize my holiday snaps. In: Proc. of European Conference on Computer Vision (ECCV)

Schneiderman H, Kanade T (2004) Object detection using the statistics of parts. International Journal of Computer Vision 56(3):151–157

Schwarzer F, Saha M, claude Latombe J (2003) Exact collision checking of robot paths. Algorithmic Foundations of Robotics V 7:25–41

Sentis L (2007) Synthesis and Control of Whole-Body Behaviors in Humanoid Systems. PhD thesis, Stanford University, Stanford, California

Serre T, Wolf L, Poggio T (2005) Object recognition with features inspired by visual cortex. In: Proc of Conf. on Computer Vision and Pattern Recognition

Shi J, Malik J (1997) Normalized cuts and image segmentation. In: Proc of the 1997 Conference on Computer Vision and Pattern Recognition (CVPR '97), IEEE Computer Society

Srinivasa S, Ferguson D, Weghe MV, Diankov R, Berenson D, Helfrich C, Strasdat H (2008) The robotic busboy: Steps towards developing a mobile robotic home assistant. In: Proceedings of the International Conference on Intelligent Autonomous Systems (IAS)

Srinivasa S, Ferguson D, Helfrich C, Berenson D, Collet A, Diankov R, Gallagher G, Hollinger G, Kuffner J, Weghe MV (2009) Herb: A home exploring robotic butler. Journal of Autonomous Robots

Stilman M (2007) Task constrained motion planning in robot joint space. In: Proceedings of the IEEE International Conference on Intelligent Robots and Systems (IROS)

Stilman M, Nishiwaki K, Kagami S (2007a) Learning object models for humanoid manipulation. In: IEEE International Conference on Humanoid Robotics

Stilman M, Schamburek J, Kuffner J, Asfour T (2007b) Manipulation planning among movable obstacles. In: Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)

Stolarz J, Rybski P (2007) An architecture for the rapid development of robot behaviors. Master's thesis, Robotics Institute, Carnegie Mellon University

Stoytchev A, Arkin R (2004) Incorporating motivation in a hybrid robot architecture. Journal of Advanced Computational Intelligence and Intelligent Informatics 8(3):269–274

Stroustrup B (2001) Exception safety: Concepts and techniques. In: Advances in Exception Handling Techniques, Lecture Notes in Computer Science, vol 2022, Springer Berlin / Heidelberg, pp 60–76, URL http://dx.doi.org/10.1007/3-540-45407-1_4

Stulp F, Fedrizzi A, Beetz M (2009) Learning and performing place-based mobile manipulation. In: In Proceedings of the 8th International Conference on Development and Learning (ICDL)

Sudsang A, Ponce J, Srinivasa N (1997) Algorithms for constructing immobilizing fixtures and grasps of three-dimensional objects. In: In J.-P. Laumont and M. Overmars, editors, Algorithmic Foundations of Robotics II

Torralba A, Murphy KP, Freeman WT (2007) Sharing visual features for multiclass and multiview object detection. IEEE Transactions on Pattern Analysis and Machine Intelligence 29(5):854–869

Tuytelaars T, Van Gool L (2004) Matching widely separated views based on affine invariant regions. Int J Comput Vision 59(1):61–85

Urmson C, Anhalt J, Bagnell D, Baker CR, Bittner R, Clark MN, Dolan JM, Duggins D, Galatali T, Geyer C, Gittleman M, Harbaugh S, Hebert M, Howard TM, Kolski S, Kelly A, Likhachev M, McNaughton M, Miller N, Peterson K, Pilnick B, Rajkumar R, Rybski PE, Salesky B, Seo YW, Singh S, Snider J, Stentz A, Whittaker W, Wolkowicki Z, Ziglar J, Bae H, Brown T, Demitrish D, Litkouhi B, Nickolaou J, Sadekar V, Zhang W, Struble J, Taylor M, Darms M, Ferguson D (2008) Autonomous driving in urban environments: Boss and the urban challenge. J Field Robotics 25(8):425–466

Wampler C, Morgan A (1991) Solving the 6r inverse position problem using a generic-case solution methodology. Mechanisms and Machine Theory 26(1):91–106

Weghe MV, Ferguson D, Srinivasa S (2007) Randomized path planning for redundant manipulators without inverse kinematics. In: Proceedings of IEEE-RAS International Conference on Humanoid Robots (Humanoids)

Weisstein EW (1999-present) Cubic formula. URL http://mathworld.wolfram.com/CubicFormula.html

Wheeler M, Ikeuchi K (1992) Towards a vision algorithm compiler for recognition of partially occluded 3-d objects. Tech. Rep. CMU-CS-TR-92-185, Robotics Institute

Wheeler M, Ikeuchi K (1995) Sensor modeling, probabilistic hypothesis generation, and robust localization for object recognition. In: IEEE Transactions on Pattern Analysis and Machine Intelligence, vol 17

Wu C, Chipp B, Li X, Frahm JM, Pollefeys M (2008) 3d model matching with viewpoint-invariant patches (VIP). In: Proc of Conf. on Computer Vision and Pattern Recognition

Wyrobek K, Berger E, der Loos HV, Salisbury K (2008) Towards a personal robotics development platform: Rationale and design of an intrinsically safe personal robot. In: Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)

Yershova A, Jain S, LaValle S, Mitchell J (2009) Generating uniform incremental grids on so(3) using the hopf fibration. International Journal of Robotics Research

Zacharias F, Borst C, Hirzinger G (2007) Capturing robot workspace structure: Representing robot capabilities. In: Proceedings of the IEEE International Conference on Intelligent Robots and Systems (IROS)

Zacharias F, Sepp W, ChBorst, Hirzinger G (2009) Using a model of the reachable workspace to position mobile manipulators for 3-d trajectories. In: Proceedings of IEEE-RAS International Conference on Humanoid Robots (Humanoids)

Zacharias F, Leidner D, Schmidt F, ChBorst, Hirzinger G (2010) Exploiting structure in two-armed manipulation tasks for humanoid robots. In: Proceedings of the IEEE International Conference on Intelligent Robots and Systems (IROS)

Zhang L, Curless B, Seitz SM (2002) Rapid shape acquisition using color structured light and multi-pass dynamic programming. In: In The 1st IEEE International Symposium on 3D Data Processing, Visualization, and Transmission, pp 24–36

Zhang L, Curless B, Seitz SM (2003) Spacetime stereo: Shape recovery for dynamic scenes

Zhang Z (2000) A flexible new technique for camera calibration. In: IEEE Transactions on Pattern Analysis and Machine Intelligence

Zhang Z, Deriche R, Faugeras O, Luong Q (1995) A robust technique for matching two uncalibrated images through the recovery of the unknown epipolar geometry. Artificial Intelligence 78(1-2):87–119

Zheng Y, Qian WH (2006) An enhanced ray-shooting approach to force-closure problems. Journal of Manufacturing Science and Engineering 128(4):960–968