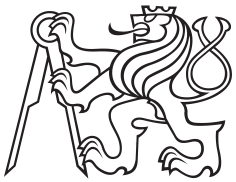
Master Thesis



Czech Technical University in Prague

F3

Faculty of Electrical Engineering
Department of Cybernetics

Haptic terrain exploration with robotic arm

Martin Burian

Supervisor: doc. Tomáš Svoboda, Ph.D.
Field of study: Open informatics
Subfield: Computer vision and digital image
January 2017

Czech Technical University in Prague
Faculty of Electrical Engineering

Department of Cybernetics

DIPLOMA THESIS ASSIGNMENT

Student: Bc. Martin B u r i a n

Study programme: Open Informatics

Specialisation: Computer Vision and Image Processing

Title of Diploma Thesis: Haptic Terrain Exploration with Robotic Arm

Guidelines:

Outdoor terrain robots must be able to traverse various rough terrains. To achieve high traversability the robots often have a complex locomotion morphology like articulated tracks. In order to control the morphology the robot needs to measure its own state (interoceptive sensing) and the surrouding terrain (exteroceptive sensing). However, exteroceptive sensing is failing in many Urban search and rescue (USAR) scenarios, like in the presence of dense smoke. Certain level of robot motion control is possible even in case of missing data but there are situations when additional tactile sensing by using a robot-mounted robotic arm is unavoidable [1]. Recent methods [2,3] bypass time-conuming arm motion planning by a heavy off-line learning from simulations as well as from real rollouts. The goal of this work is to design an efficient exploration policy for the robotic arm. The motion control method [1] produces a tactile sensing request. The arm should explore a free space for its own motion and then get the tactile measurement. The sought policy should minimize the arm movement, it should exploit the arm morphology.

Bibliography/Sources:

[1] M. Pecka, K. Zimmermann, M. Reinsten, and T. Svoboda. Controlling Robot Morphology from Incomplete Measurements. In IEEE Transactions on Industrial Electronics, accepted May 2016.
[2] Levine, S., Wagener, N., and Abbeel, P. (2015). Learning Contact-Rich Manipulation Skills with Guided Policy Search. In IEEE International Conference on Robotics and Automation (ICRA).
[3] Kupcsik, A., Deisenroth, M. P., Peters, J., Poh, L. A., Vadakkepat, P., and Neumann, G. Model-based contextual policy search for data-efficient generalization of robot skills. In Artificial Intelligence. 2014 (on-line, in press).

Diploma Thesis Supervisor: doc. Ing. Tomáš Svoboda, Ph.D.

Valid until: the end of the winter semester of academic year 2017/2018

L.S.

prof. Dr. Ing. Jan Kybic
Head of Department

prof. Ing. Pavel Ripka, CSc.
Dean

Prague, May 27, 2016

Acknowledgements

I would like to thank my supervisor Tomáš Svoboda for his infinite patience, and to my partner Zuzana who helped me survive the creation of this thesis.

Declaration

I declare that the presented work was developed independently and that I have listed all sources of information used within in accordance with the methodical instructions for observing principles in preparation of university theses.

Prague, 9. January 2017

.....

Abstract

In search and rescue missions, the sensors on the Absolem mobile robot used in the TRADR project can fail due to environmental conditions like smoke and dust, making further robot operation difficult and dangerous for the robot. In such cases, the robot-mounted robotic arm can be used to gain information about the environment by obtaining tactile measurements from it. We propose an algorithm to guide the arm during the exploration, along with methods to control the arm movement and to gather the measurements with a 3D force sensor from safe distance. The system is implemented and put to the test in both simulated experiments and real world trials.

Keywords: mapping, robotic arm, coverage path planning, tactile sensing

Supervisor: doc. Tomáš Svoboda, Ph.D.

Abstrakt

Při záchranných misích mohou senzory robotu Absolem používaného v projektu TRADR selhat díky nepříznivým podmínkám prostředí, jako například kouř nebo prach. Bez senzorických dat je další pohyb robotu obtížný a nebezpečný. Robotická ruka, kterou je robot vybaven, může být v takovýchto situacích použita pro získání informací o prostředí snímáním taktilních měření. Navrhli jsme algoritmus, který při exploraci ruku navádí, spolu s metodami pro řízení ruky a snímání měření z bezpečné vzdálenosti. Systém jsme implementovali a otestovali jak v simulovaném, tak v reálném prostředí.

Klíčová slova: mapování, robotická ruka, plánování pokrývajících cest, taktilní vnímání

Překlad názvu: Taktilní průzkum terénu robotickou rukou

Contents

1 Introduction **1**

1.1 Exploration task overview 2

2 Robotic hardware and framework **5**

2.1 ROS 5

2.2 Kinova JACO 6

2.3 MoveIt! 10

2.4 Tactile sensing 16

2.5 Contributions to the hardware and framework 17

3 Maps for Exploration **21**

3.1 Map representation 21

3.2 Exploration algorithm 23

4 Coverage Path Planning **25**

4.1 Applications 25

4.2 Approach families 25

4.3 Compact space heuristic 29

4.4 Generalization to 3D 30

4.5 Method comparison 31

5 Implementation **33**

5.1 System architecture 34

5.2 Optoforce driver 35

5.3 Robot motion 35

5.4 Exploration algorithm 39

5.5 Integration with the Absolem robotic platform 41

6 Experiments **43**

6.1 Benchmark environment 43

6.2 Simulated coverage algorithms . . 43

6.3 Simulated arm experiments 50

6.4 Real arm experiments 54

7 Conclusions **59**

7.1 Results of our work 59

7.2 Further work proposals 60

A Bibliography **61**

B CD contents **65**

Figures

2.1 Kinova JACO 7

2.2 JACO blueprint 8

2.3 MoveIt! architecture 11

2.4 Non-optimal motion plan 13

2.5 Non-optimal motion plan 14

2.6 Optoforce sensor 17

2.7 Sensing tool 19

2.8 Sensing tool 19

2.9 Tool grip handle 20

4.1 Boustrophedon pattern 27

4.2 Trapezoid and boustrophedon decompositions 28

5.1 Overview of the system architecture 34

6.1 2D benchmark environment 44

6.2 Coverage path – random algorithm in 2D 45

6.3 Coverage path – compact space heuristic in 2D 45

6.4 Inner state of heuristic algorithm 46

6.5 Coverage path – neural network in 2D 47

6.6 Original neural network path . . . 47

6.7 Internal state of neural network algorithm 48

6.8 Coverage path – compact space heuristic in 3D 49

6.9 Obstacle avoiding motion plan . . 51

6.10 Position error during simulated test drive 52

6.11 Simulated arm exploring environment 53

6.12 Position error during real arm test drive 55

6.13 The arm touching a table 56

6.14 The real arm exploring empty environment 57

6.15 The real arm exploring a test environment 57

Chapter 1

Introduction

The goal of the TRADR project [1] is to provide rescue workers in urban search and rescue missions in disaster struck areas with intelligent robotic assistance, to increase the efficiency of the operations. "Various kinds of robots collaborate with human team members to explore or search the disaster environment"[1], helping the team as a whole to collect as much data as possible. The focus of the project is on the cooperation and fusion of the human and robot elements, resulting in optimal usage of the unique capabilities of each team member.

One of the robot team members is the BlueBotics Absolem robotic platform, developed originally for TRADR's predecessor project NIFTi [2]. The robot is designed to be remotely driven by an operator to explore the disaster-struck environment and collect information valuable for further decision-making in the whole rescue operation. The robot moves on articulated tracks, which can be used to adapt the morphology of the robot to optimize traversal of rough, uneven terrain, and is equipped with a range of sensors including omnidirectional and thermal camera and a LIDAR. These instruments are used to construct a model of the environment which the operator of the vehicle uses to drive the robot after its goals.

All of the aforementioned sensors are based on light. In difficult visibility conditions, like dense smoke, fog or dust, the data from the sensors cover only a very limited range around the robot, if they are available at all. Then, driving the robot is unsafe, as the dangers of the environment remain hidden from the operator. Sensors based on other physical phenomena than light, like radar and sonar, can be used in such cases; the robot is however equipped with neither of those. It is, on the other hand, equipped with a Kinova JACO robotic arm that can be used to get a tactile measurement of the terrain height in front of the robot [3], giving the operator at least some information.

Even common things as water puddles can, due to the specular reflection of the laser light, mislead the sensors and appear as a hole in the ground. In [4], the problem of partially unknown environment for controlling the morphology

of the robot is solved by learning a policy that can cope with the missing data. The policy can, if it is too unsure if the motion is safe for the robot, request a tactile measurement of terrain elevation in front of the robot. Recently, the individually controllable track segments, *flippers*, on the robot have been equipped with force sensors that detect contact between the track and terrain, giving the robot a means of getting the required data without the hassle of operating the arm.

The methods mentioned can tell the operator if the robot is facing terrain it can traverse. They however cannot inform the operator if any other obstacles are in front of the robot, perhaps including narrow obstacles that would pass between the flippers sensing the ground in front, or overhanging structures that would damage the robot body. Here, the arm mounted on the robot is still irreplaceable.

1.1 Exploration task overview

Our task is to use a robotic arm to explore space in front of a mobile robotic platform to identify obstacles that could damage it. In the big picture, we need to build a piece of software that will drive the robotic arm hardware around the environment intelligently, not crashing the arm into anything in the process, and via this movement explore the environment in front of the robot. All the while, it will be processing data from a 3D force sensing tool we designed to detect the obstacles. To cover the whole complexity of driving a robotic arm in an environment so that all the environment is explored, we will need to acquaint the reader with the workings of many a field, from robotic manipulators to planning efficient exploring paths. We will be working with the robotic arm alone, without the Absolem robotic platform. When the arm is mounted on the robot, the same software can be used to perform the exploration, only the parameters of the space to be explored must be specified.

The whole system (the robotic platform and the arm) run in the Robot Operating System (*ROS*) framework. To maintain system consistency and simplify implementation, we will be implementing our solution in ROS as well. We provide an overview of the framework in Section 2.1, where we briefly present the basic concepts to help the reader grasp what is underlying our implementation decisions.

The arm is an advanced piece of hardware and embedded programming, with built-in controls capable of operating the arm. The functionality is exposed to the system via a manufacturer-provided driver. We describe both the arm hardware and driver in Section 2.2 and mention basic theory of robotic manipulators. We also describe the force sensor and its extension we designed for this application further on in Section 2.4.

Although the driver exposes simple interfaces to control the arm movement, it is necessary to control the arm's motion in context of the whole environment including the robotic platform and any obstacles around the robot. This task is solved by "MoveIt!", a library built around functionalities required to use robotic manipulators. We present the library in Section 2.3. We also include information about the mathematical and algorithmic framework of robotic manipulators and planning and controlling their motion that is crucial for solving our task.

Next, in Chapters 3 and 4, we present the rationale behind our method choice and the overview of state-of-the-art algorithms for our method of choice, coverage path planning.

Finally, we describe the implementation of the algorithm in ROS and experiments with the exploration methods, both simulated and on a real robotic arm.

Chapter 2

Robotic hardware and framework

2.1 ROS

The arm (and the whole TRADR robot) runs under the *Robot Operating System* – ROS [5]. This framework allows us to build robot software in a very modular fashion by providing us with standardized units ("nodes") and M-to-N communication links ("topics"). Each hardware component of the robot (e.g. a LIDAR or a motor driver) has its own driver node, and so does each functional software part (mapping, adaptive traversability etc.). In ROS, we can put these ready-made parts together easily and run them in a distributed fashion, as each node can run on a different computer and the system will automatically relay the information between the hosts. The modularity also guides us to design small and manageable pieces of software that integrate well with the rest of the environment and provide the new functionality we want.

The ROS framework is only a common interface specification, with the actual implementation of a node completely independent on other nodes. Each node can be implemented in a different programming language, as long as they conform with the communication standards.

Nodes run in event loops, referred to as *spinning*. In each loop iteration, an event is taken from the incoming event queue and the appropriate callback is issued.

2.1.1 ROS primitives

Basic unit of information in ROS are *messages*. A message is a tuple with named values of different types, possibly other messages. A Pose message, representing position and orientation of an object in space, contains a Position message (which in turn contains x, y, and z coordinates as floating point

numbers) holding the information about the position of the object, and a Quaternion representing its orientation.

The basic communication links are *topics*. Nodes write messages to topics, and other nodes can subscribe to the topics and consume messages from them. For example a sensor driver node publishes the sensor measurements for the rest of the system to use.

The second communication primitive provided by ROS core are *services*. Services implement request-response behavior, when a client calls the service provided by a node and receives a response for its request. For example, a laser range meter driver could be implemented as a service, accepting requests for measurement and returning the outcome.

2.1.2 Actions

On top of these primitives, another interface type is implemented, the *actions*. Actions are not a part of core ROS as topics and services, but have become a standard part of ROS interfaces.

Actions consist of a set of topics and and are similar to services. A client sends a request to the action server. The action server then sends feedback back to the client as it performs the requested action. When the server is done, the client is notified by a goal message.

A good example for an action server is one for driving the robot to a defined pose. The action server receives a Pose message and sets off. As the robot is under way, the server sends periodic updates on the current pose back to the client. When the final pose is reached, the server notifies the client that the work is done together with optional complimentary data, like the length of the path taken.

2.2 Kinova JACO

We will be working with the Kinova JACO robotic arm. It is a 6 degrees of freedom lightweight (4.4 kg) robotic arm, designed by Kinova Robotics for use in assistive and collaborative applications. Originally designed to be used in assistive robotics, mounted on mobile structures as wheelchairs, JACO's lightness makes it ideal for uses on mobile robots [6].

2.2.1 Serial manipulator

The JACO arm is a serial manipulator, meaning that it is a chain of links interleaved with joints. A movement in a joint affects all the following joints

Figure 2.1: The Kinova JACO 6DOF robotic arm with the 3-finger gripper. Photo taken from [6]

and links in the chain. The base of the chain is often a fixed base, while the tip of the chain is referred to as the *end effector*. The positional behavior of the arm is referred to as arm kinematics.

The compound joint and link transformations (rotations and translations, respectively) link the joint configuration to the end effector pose (as used in ROS terminology, representing position $\mathbf{p}$ and orientation $\mathbf{r}$). Computation of the end effector pose from joint angles $\mathbf{q} = (q_1 \dots q_n)$, a column vector where n is the number of joints, is referred to as *Forward Kinematics – FK* computation. From now on, we consider a 6DOF manipulator like JACO is.

The inverse problem is finding joint values that reach the given end effector pose, and is called *Inverse Kinematics - IK*. As many inverse problems, it is considerably harder to solve than the forward problem [7, sec. 2.12]; it is often solved numerically.

When not positions, but velocity of the end effector and joints is studied, it is in the field of differential kinematics. The relation between the joint angular velocities $\dot{\mathbf{q}}$ and end effector pose velocity (usually called *twist*)

$$\mathbf{v} = \begin{pmatrix} \dot{\mathbf{p}} \\ \omega \end{pmatrix},$$

where $\dot{\mathbf{p}}$ represents linear velocity and ω angular velocity, is described by the Jacobian matrix $\mathbf{J}(\mathbf{q}) \in \mathbb{R}^{6\times 6}$ of the manipulator:

$$\mathbf{v} = \mathbf{J}(\mathbf{q})\dot{\mathbf{q}}$$

The inverse problem is again considerably harder, as it involves inverting the Jacobian (if it is regular). As inversions in possibly near singular matrices

are numerically unstable, pseudoinverse that deals with singularities is often used.

2.2.2 Arm kinematics

The arm has slightly unusual geometry in its wrist. The wrist is not spherical (the axes of its joints do not intersect in one point), which means that positioning the arm in a given pose cannot be separated into position and orientation problems. Usage of some inverse kinematics solvers that count on this is thus impossible.

Instead, the wrist consists of three revolute joints that have angular offsets of 60° (see Figure 2.2). This limits the workspace where the arm has full 6DOF capabilities, as the rotation of the wrist is limited; however, the arm exhibits these limitations only in extreme poses.

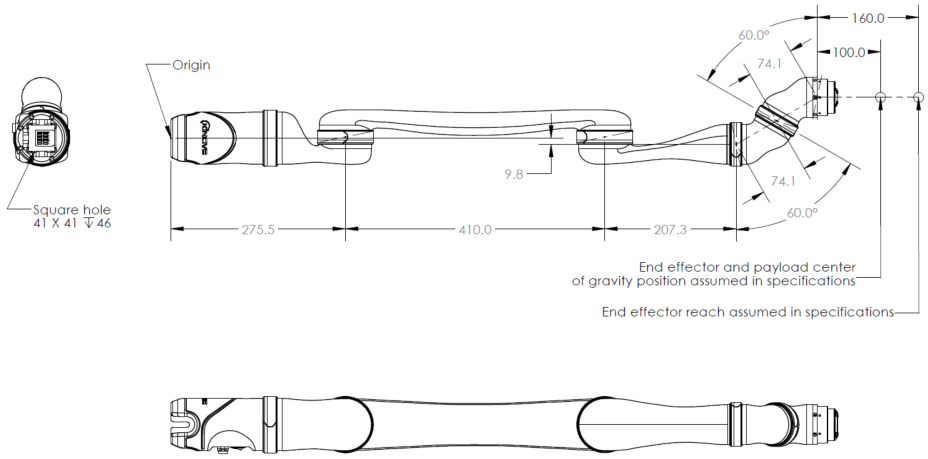


Figure 2.2: JACO arm technical specification. Note the wrist (last three joints) construction that limits the end effector orientation. Taken from [8]

The arm manufacturer states the reach of the arm is 90 cm. The arm (including the gripper) can actually reach beyond 100 cm, but only with limited rotation. In the 90 cm work range, a wide range of hand orientations is still possible. On the other side, the arm cannot reach all orientations if the end effector is close to the base. We observed that to reach closer than 40 cm and point the gripper away from the base, the arm needs to be in extreme poses, which are better avoided as they bring the arm close to self-collision.

2.2.3 Kinova ROS driver

The manufacturer provides an open-source ROS library to operate the arm [9]. The driver runs a node that communicates with the arm via a USB

interface and relays incoming commands from other components. The driver is an evolution of a former JACO ROS driver that supported only the JACO arm. Recently, Kinova replaced the JACO-specific driver with the new one, which supports also other Kinova arms.

The driver provides interfaces for moving the arm to joint space and Cartesian space targets, controlling fingers of the gripper and directly driving Cartesian velocity of the end effector and angular velocities of each joint [10]. It also provides two utility services for homing the arm and for stopping all movement immediately.

The last mentioned service is the only soft emergency stop available for the arm in case of any mishaps. The dependency on the framework to forward the message is impractical. The only other option is the arm's power-off switch, which shuts the arm down completely. The motors then come loose and the arm needs to be held in place by the operator not to come crashing down.

2.2.4 Cartesian and joint space targets

Driving the arm to joint space and Cartesian space targets is possible via the respective action servers. The action servers accept Pose, resp. JointAngles message describing the desired target state. They report the current position back as the movement proceeds, and in the end return the actual pose the arm ended in. For the Cartesian pose target, no IK computation is performed by the driver itself; the target pose is only fed into the arm and the internal control algorithms take care of the rest. The movement also does not take any obstacle avoidance or collision prevention into account, except self-collisions that are checked. This renders the features practically unusable if we know there are obstacles in the environment.

2.2.5 Cartesian and joint space velocity

Controlling the arm movement via directly specifying velocities is implemented quite differently. To set Cartesian or joint angular velocities, the client publishes a message to a specified topic. The driver then forwards the message to the arm, where it is consumed from a queue by the arm's control algorithm. The control algorithm runs at 100 Hz, and sets the velocity for the next iteration from the queue. Thus, once the client stops publishing to the topic, the movement stops immediately. The control is arguably implemented this way to minimize latency of stopping any movement. It also means that to set the velocity for a longer time, the client needs to publish to the velocity control topic periodically at 100 Hz. With lower frequency, the movement will be jagged, as some control loop iterations will set zero velocity. If the

frequency were higher, the input queue of the arm would overflow, causing the arm to stall.

2.2.6 Utility services

Homing the arm is possible by calling a home service in the ROS driver. The home pose can be adjusted in the arm firmware by Kinova's arm controlling software.

The stop service stops all movements of the arm immediately. To resume operation, the arm must be started again by calling a start service. When restarted, all commands that came after stopping are dropped and need to be re-issued if they are to be carried out.

2.3 MoveIt!

The ROS framework was primarily designed to operate whole mobile robots; it is not so well suited to control robotic manipulators. To help in this area, the MoveIt! package [11] was designed to facilitate this task. It encompasses the full stack necessary to operate a robotic arm, from managing the state of the scene, through manipulation, motion planning, collision-checking, IK solving to outputting commands that can be used directly to control a manipulator.

2.3.1 MoveIt! architecture

MoveIt runs a large monolithic node called move group. The move group is responsible for all the operations of the arm. It consumes the state of the arm (joint angles) from the driver, as well as commands from the user issued via client libraries. The architecture is illustrated in Figure 2.3.

The central node represents the move_group, which listens to input from the robot hardware in the lower right, as point cloud sensors and introceptive sensors. It outputs commands for the manipulator to the controller with `JointTrajectoryAction` interface and is controlled by the inputs on the left. MoveIt has a plugin-oriented extensible architecture, where all services like IK solving and motion planning are defined as plugins, interchangeable with other implementations.

Even though MoveIt is very popular, it is very poorly documented. The best source of information on how things work exactly is the automatically generated code documentation, and the source code itself [12]. Some of the features, as motion planning and collision object management, are far

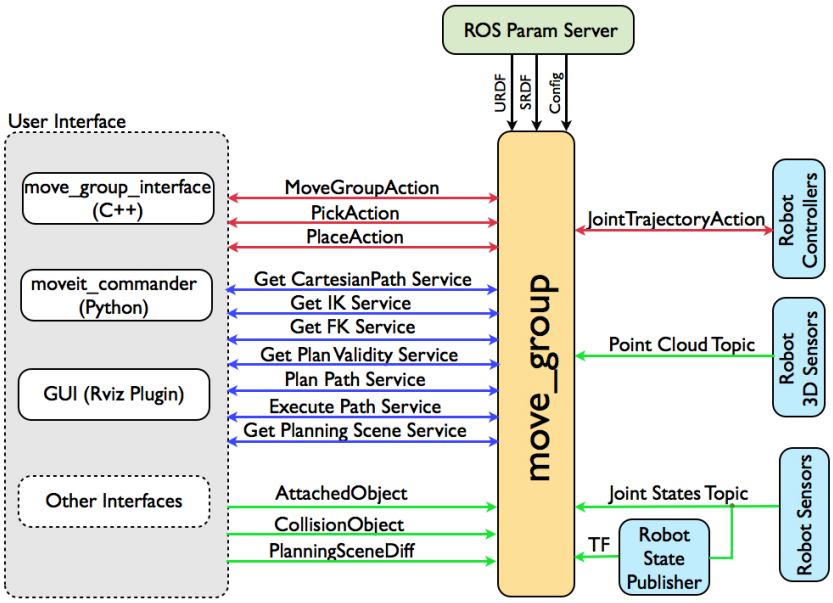


Figure 2.3: Illustration of MoveIt! architecture. Taken from [11]

from stable. Setting things up is rather complicated, and while the basic functionality works well, the more advanced things have many problems.

2.3.2 User interface

The user interface, or client libraries, for MoveIt are prepared in C++ and python. The libraries contain objects and utility functions to use the functionality of the move group and communicate with the move group via its ROS topic and service interface. The client libraries can be used directly to access the motion planning capabilities to reach some higher-level goals.

2.3.3 Robot interface

MoveIt interfaces with the robot hardware by an action server implementing the `JointTrajectoryAction`. The server then drives the arm hardware. The `JointTrajectoryAction` is specified as a trajectory in space of joint angles and velocities, prepared by the motion planner with respect to both the arm kinematics (preventing collisions) and the arm dynamics (taking limits on joint velocities into account). The trajectory waypoints specify both the desired joint angles and joint velocities necessary to reach the next waypoint. Each waypoint also specifies a point in time in which the arm should have reached it.

In case of the Kinova ROS driver, no such server is provided by the manufacturer. A pull request (code contributed by a developer) on GitHub containing this functionality exists, but is not a part of the release yet. We used an implementation of this server by Matěj Balga [13] , which we updated to work with the newest drivers from Kinova. The server is rather simplistic. It follows the joint velocities defined in the waypoints, forwarding them to the arm driver for as long as the waypoint specifies.

At the trajectory end, the current arm joint pose is compared to the desired final pose. If they are different, a simple P regulator is used to move the arm joints to within tolerance of the final positions. A regulator is a component in a feedback loop that computes values of action variables (here joint velocities) based on the current state of a system. The variables are fed to a system to get the system to a desired state (here the final joint angles). A P regulator computes the action variables based on the error between the current system state and the desired state.

This behavior is very apparent after following a more complex trajectory, which includes many direction changes. It could prove dangerous for the arm when maneuvering in proximity of obstacles, as the arm strays off the collision-free trajectory. The final movement is also very fast, and could damage the arm if anything were to happen. We assume this happens because the driver forwards the velocities to the arm directly, without taking accelerations into account. The arm firmware has acceleration limits built in, and when the limit is applied and the set velocity is truncated, the actual arm position diverges from the position expected by the driver.

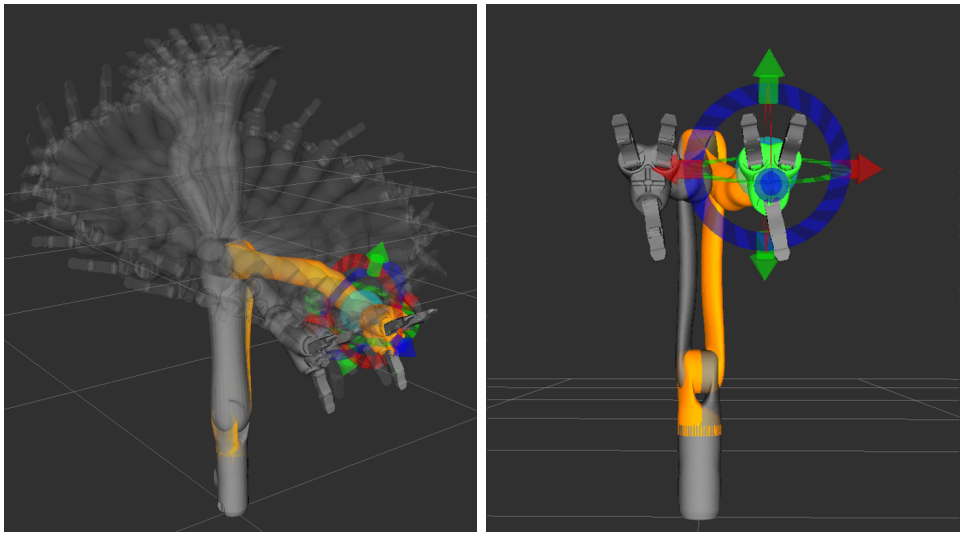
The driver could possibly be improved by interpolating the velocities on a spline rather than linearly, smoothing out the velocity changes and thus not letting the arm firmware interfere with the control.

2.3.4 Inverse Kinematics solver – TRAC-IK

During our initial experiments with the arm, we encountered puzzling behaviors of the motion planners. The arm would sometimes move along a seemingly random, unnecessarily long path. At closer inspection, we concluded that the planning was actually optimal, but what was wrong was the Inverse Kinematic (*IK*) solution for the target pose. Such plan is illustrated in Figure 2.4.

Because of the slightly non-standard arm kinematics (see section 2.2.2), the default IK solver used by MoveIt (the KDL numerical solver) would sometimes produce a solution that indeed reaches the desired end effector pose, but that is very far in joint space from the current joint state (e.g. the first joint is rotated), even when a very small end effector motion is required.

This is caused by the KDL implementation. KDL finds the IK solu-



a: **b:**

Figure 2.4: When planning a small movement (10cm up) with KDL, motion plan (a) was produced; notice the cause in (b), the first joint configuration changed by 180°. The solid gray arm is the start state, the orange arm the goal state, with the intermediate planned states in semi-transparent gray

tion by iteratively descending the pose error by performing its local linear approximation[14] and taking a step that minimizes it, analogic to the Newton method. The descent is performed by the inverse Jacobian method, but Moore-Penrose pseudoinverse is used in KDL because *"it is more efficient to compute"*[14], while it is numerically more stable when computed via SVD decomposition.

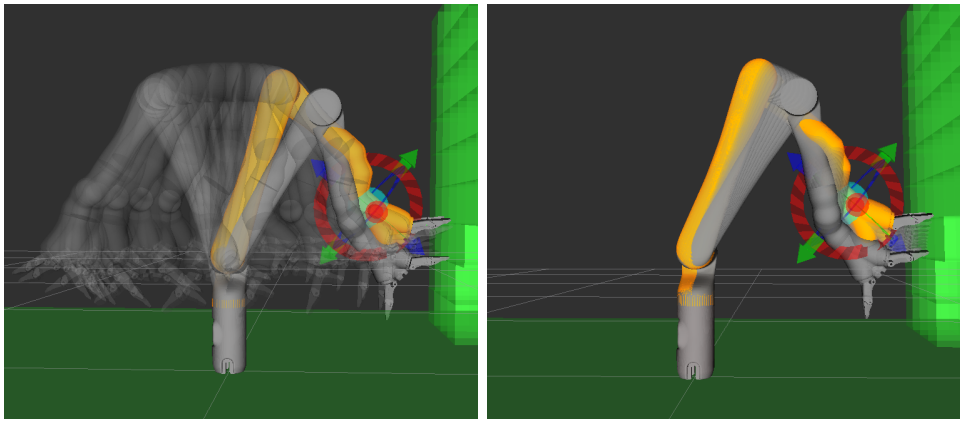
The descent can get trapped in local minima when some of the joints limits are reached. This is solved by restarting the search from a random seed position. Then, the found position can be one very distant from the original position in joint space.

This issue is tackled by a new IK solver released recently, TRAC-IK [14]. The solver implements another approach to solving the IK problem by formulating it as a Sequential Quadratic Programming problem, minimizing Cartesian error $|e|^2$ with joint limits as constraints. The solver also allows the user to specify other constraints, like a wish to minimize joint-space distance between the seed (current arm joint values) and the result.

This setting allows us to eliminate the large joint space changes for minor movements, which was an issue with the JACO arm before.

A similar issue, which we failed to eliminate in the end, was caused by joint limit settings in the arm. While all the joint actuators allow continuous rotation (limited in software to ± 27.7 turns [8]) and only the second and third are limited by the arm design, when the arm is configured in MoveIt,

a limit of ±1 turn is imposed on the unlimited joints. The setup guide for TRADR project mentions limiting the joints is crucial to allow efficient planning. While this is true, it also causes artifacts like the path planned in Figure 2.5a, where as in the previous case a tiny change in configuration needs seemingly unreasonably complex trajectory which is completely justified from the machine point of view, but looks nonsensical to an observer.



a: **b:**

Figure 2.5: Motion plan affected by limited joints. In (a), the joint limit would be crossed if the arm took the straight path (b), and the IK solver and planner had no other chance than to perform full rotation in the first joint to stay within the limits. Plan (b) was generated after executing the first plan, which moved the arm away from the joint limit. The solid gray arm is the start state, the orange arm the goal state, with the intermediate planned states in semi-transparent gray

2.3.5 Motion planning

Planning the motion of the manipulator in an environment with obstacles is a very complex task, suffering heavily from the curse of dimensionality. We are planning in 6-dimensional joint space from a given start pose to the target pose. Search-based planning algorithms fail because of the high dimension of the planning space, augmented by the need of fine space discretisation to keep the path smooth in the continuous real world.

Specialized planners based on sampling (Rapidly-exploring Random trees – *RRT* [15] or Kinodynamic Interior-Exterior Cell Exploration – *KPIECE*) are used to make planning the movement faster. The planners set constraints for the movement, like joint angle and velocity limits, collision checks etc. Then, starting either only from the start state (unidirectional) or from the start and target state at once (bi-directional methods), the planner tries to construct a sequence of configurations starting in the start state, ending in the target state and not including any configurations that are in collision with the environment.

MoveIt has a plug-in interface to motion planning libraries, the default being OMPL [16], which implements a wide variety of algorithms, including the two mentioned here. Based on experiments with the arm and recommendations from [3] and [16], we used the KPIECE family of planners, namely LBKPIECE, Lazy Bi-directional KPIECE.

When a motion plan, a series of configurations from the start to the end, is computed, it is usually unnecessarily complicated, performing redundant movements. To smooth the trajectory out, the plan is simplified by a process in which states that are unnecessary, which means the path from their parent state to their child state satisfies the constraints as well, are removed. It is not unusual for a raw plan of movement of tens of centimeters to contain about two hundred states, which are later all removed, because the straight trajectory from the first to the second state is satisfactory.

2.3.6 Moving the arm without planning

The problem with motion planning is that it is a complex task in itself, and, even though it is readily available as a part of MoveIt, it is slow and sometimes yields bad plans, even after employing the optimizing IK solver TRAC-IK[1]. In case of small movements, where a trivial linear path is often a good enough, if not optimal, solution, computing the whole huge motion plan is plain waste of resources and time.

The previous work [3] used motion planning to carry out a simple downward motion, moving in 1 cm steps. The same unnecessarily long paths have been reported.

It is not necessary to move the arm by following a motion plan. As we mentioned above, the Jacobian maps joint velocities $\dot{\mathbf{q}}$ to Cartesian twist $\mathbf{v}$

$$\mathbf{J}(\mathbf{q})\dot{\mathbf{q}} = \mathbf{v}.$$

Conversely, the Jacobian inverse (if it exists) maps Cartesian twist to joint speeds:

$$\mathbf{J}^{-1}(\mathbf{q})\mathbf{v} = \dot{\mathbf{q}}$$

The inverse does not need to exist, or can be highly unstable as the Jacobian can be badly numerically conditioned. The near singularity of the Jacobian means that any of the null space of joint speeds would cause end effector movement in the desired direction. Or that the end of the end effector cannot be moved in the desired direction without requiring infinite speeds of the joints. In the second case, we need to leave the singularity in another direction.

[1] based on simple experiments not mentioned in this thesis, which consisted of simply moving the arm around with the motion planner

In the first case, the problem is underconstrainted and can be solved for least square norm solution, which means the slowest joint motion. With advantage we use the Moore-Penrose pseudoinverse expressed in SVD decomposition of $\mathbf{J} = \mathbf{USV}^T$:

$$\mathbf{J}^\dagger(\mathbf{q}) = \mathbf{VS}^{-1}\mathbf{U}^T$$

This approach gives us opportunity to check the conditioning of the operation, inspecting the singular values and zeroing out those that are below some threshold and could cause the solution to fail when inverted in constructing the pseudoinverse. The pseudoinverse then gives us the joint velocities directly:

$$\dot{\mathbf{q}} = \mathbf{J}^\dagger(\mathbf{q})\dot{\mathbf{p}}$$

We can then drive the robot along a trajectory with known speeds without expensive motion planning. A very simple example can be simple linear trajectories, where the Cartesian velocity part of the twist is constant, and the angular part is zero.

2.4 Tactile sensing

Previous work in the field of tactile exploration at FEE [3] used effort readings from the arm joints to detect tool contact when exploring terrain. To prevent arm damage, the movement had to be very slow, making it painstaking to explore even very little of the environment.

To improve exploration efficiency, we used a 3D force sensor to detect whether our tool is in contact with the environment. This allows us to move the arm much faster with confidence that we will be able to stop the arm with latency low enough to guarantee the arm nor the force sensor would be harmed.

2.4.1 Optoforce sensor

The force sensor used is an Optoforce OMD-20-SE-40N sensor. Optoforce sensors represent a novel approach to 3D force sensor construction, as their working principle enables them to be made very rugged, resilient.

The Optoforce sensor, seen in cross-section in Figure 2.6b, consists of an aluminium base plate, made in different designs, and a hollow silicone hemispherical dome. The interior of the dome is coated with a reflective layer. A light emitter, IR LED, is positioned on the base plate in the center of the dome. Four light-sensitive diodes are placed around it.

When the sensor is operational, the LED emits light which bounces around the dome and provides known, constant illumination on the photodiodes.

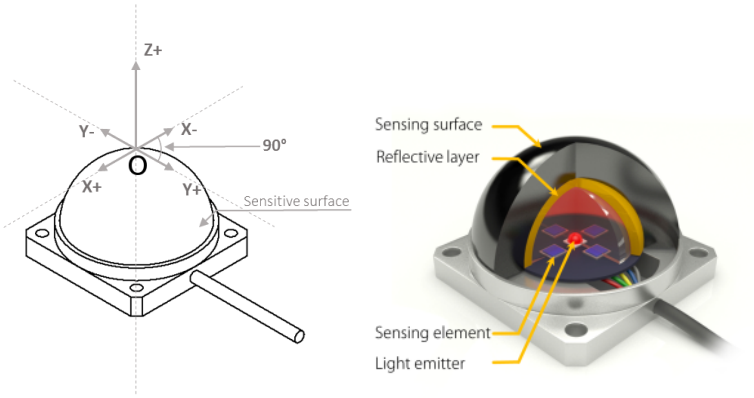


Figure 2.6: The optoforce sensor reference frame (a) and cross-section (b). Taken from [17]

When force is applied to the dome, it deforms, and the deformation causes the illumination of the diodes to shift. From this shift, it is possible to determine the direction in which the force has been applied. According to the manufacturer, deformations in range of hundreds of nanometers can be measured [17]. After some time in operation, the sensor occasionally gets confused and reports rather large false forces (up to 10N on z axis). A strong compression of the dome resets the sensor to correct readings.

The sensor has its own DAQ (*Data AcQuisition*) module, which communicates with a computer via a serial link emulated over USB. The manufacturer provides a C++ library that exposes the sensor functionality in an API. It however depends on the problematic Qt framework. Shadow Robot Company [18] implemented a ROS node that connects to the sensor serial interface, configures sensor parameters like measurement and filter frequency, reads the data output by the sensor and publishes them on a ROS topic to be processed by other components. The readings published are in Newtons, as the driver is provided with a "sensitivity report", a calibration result provided by the manufacturer that relates the internal units used by the sensor to actual physical forces. The driver software was released under GPLv2. We simplified the driver and slightly adjusted it to work better in our setup.

2.5 Contributions to the hardware and framework

2.5.1 Arm simulator

To simplify development, we implemented a mock arm driver mimicking the Kinova ROS driver. The mock driver is meant to be used with MoveIt! as a drop-in replacement of the original driver, and hence it only implements

`FollowJointTrajectory` action. It receives a joint speed trajectory, and replays it, publishing mock arm configuration to the respective topic where the rest of the system expects robot joint readings. Then, the rest of the system cannot distinguish the mock arm from the real robot.

The mock driver can be configured to execute the joint trajectories faster, to save time the user spends watching the simulated arm move. The mock driver does not simulate arm dynamics in any way, it only tries to follow the trajectory in the same way as the real driver does. In the end, instead of being driven by a regulator to the real target angles, we set the joint angles directly.

The simulator also mimicks the driver's ability to drive the arm by specifying joint speeds. It reads a joint velocity input at 100 Hz, behaving exactly as the real driver as described in 2.2.5.

This simulator allows us to test the system without the need of using, driving and taking care of an expensive physical arm.

2.5.2 Sensing tool and measurement processing

The Optoforce sensor we use is relatively small. Mounting it on the JACO arm and using it to sense direct contact with an obstacle would mean the arm would get uncomfortably close to creating. We devised a way to extend the sensor so that force readings are taken not at the sensor dome, but at a stick. A sensing tool was designed and built, consisting of a handle designed for the arm gripper for easy attachment and a frame that transfers forces from a sensing stick to the sensor. This increases the arm range and makes it possible to keep the arm safely away from unknown obstacles.

The sensor is mounted in a frame together with a stick. The stick ends at the sensor with a specially designed cap that transfers forces from the stick to the dome of the sensor itself. The other end of the stick is blunted not to be dangerous to its surroundings.

The stick acts as a lever with the pivot point being the assembly frame. When a lateral force is applied to the stick, the sensor records force in opposite direction. Axial force is applied directly to the sensor. The principle is illustrated in Figure 2.7b.

The sensor dome is in contact with a cap attached to the stick. The cap is pressed against the dome with a preloaded force, to ensure accurate measurement of even very small forces applied to the stick. The deformation of the dome is limited by a pair of nuts that limit the travel of the stick.

The whole assembly is mounted to a lump of plastic ergonomically designed for the JACO 3-finger gripper, so that the sensor can be attached to the

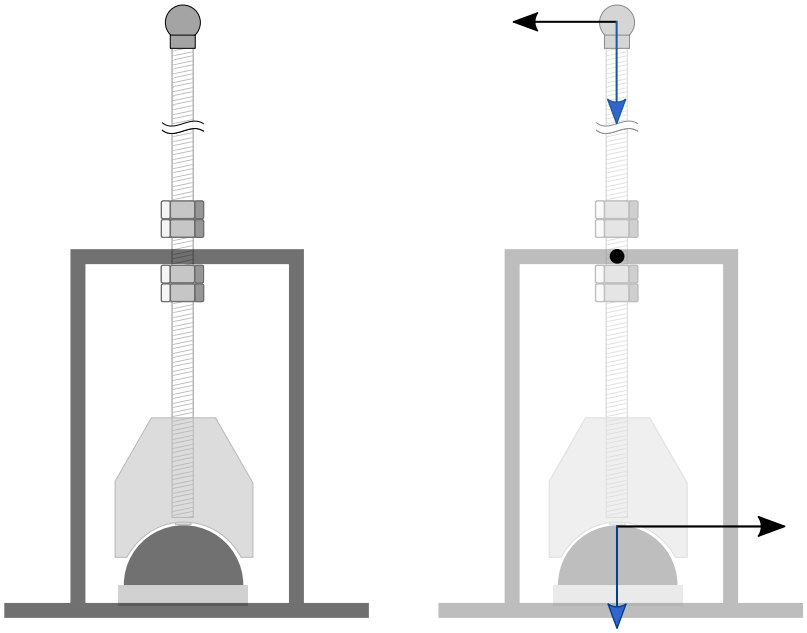


Figure 2.7: The sensing tool assembly in cross-section (a) and forces in the instrument (b)

Figure 2.8: The sensing tool held in the JACO arm's gripper

hand without any hassle and operatively, and could, in theory, be mounted somewhere on the robot and taken by the arm automatically only when necessary.

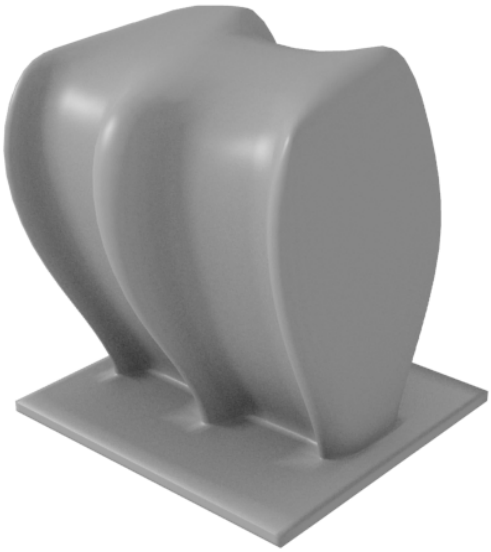


Figure 2.9: The ergonomic grip handle we designed to facilitate mounting hardware onto the arm. The square base can be used to mount any object to the handle in the future. The 3-finger gripper on the arm can hold it firmly enough to withstand considerable force

The sensor is connected to the DAQ and then the computer through external cables. This limits the arm, as for example a full joint rotation would wind the cable around a link. For this reason, we left the cable free from the arm and used it only when the experiment needed it. In the arm, a builtin pass-through cable is available, which would eliminate the external cable and allow us to fully use the arm. We however do not have the means to use the cable to transfer sensor data, although we hope it will be possible in the future.

The measurements from the sensor are processed by other components of the system. By observing the data obtained by the sensor during random movement of the arm (no contact with the stick), the gravitational force of the stick itself, combined with vibrations, can cause lateral forces of up to 1 N. We recommend detecting contact by thresholding the contact force by 2 N for the lateral forces and 4 N for axial forces.

Chapter 3

Maps for Exploration

The robot is to explore the environment in front of it, and to record its findings in a map. The map needs to be represented in such way that it can be used both in further exploration, to prevent the arm from hitting obstacles, and to further operate the robot, e.g. decide if it is possible to move forward. We also need to find a framework in which we will formulate and solve the problem of guiding the robot through the whole environment. The decision needs to be made on how to drive the arm.

3.1 Map representation

We need to represent a metric, three-dimensional map. Such maps can be represented either continuously, maintaining precise positions of points and objects represented as geometric primitives, or discretized, storing information about discrete parts of the environment.

Of continuous maps, the only representation applicable to our task are point clouds. In point clouds, the map is represented by a set of points. Those can then be for example avoided by the robot because they represent obstacles. Our sensing instrument detects contact not in single point, but on the whole tool. It is then impossible to reconstruct the precise point at which the contact occured. Furthermore, continuous coverage of the environment would increase the complexity of the coverage algorithm. We do not consider any continuous map representation as a candidate for our solution.

On the other hand, discrete representations are easy to implement, and can be used without loss of resolution if the parameters are tuned correctly. Planning in discretized environments is easier as well. In discrete maps, the whole volume of the environment is discretized and the discrete elements, representing a portion of the environment, represent some information about it in the map.

The basic representation of a 3D environment in a grid is an elevation map. The map is represented in a 2D grid, with its x and y coordinates corresponding to a horizontal plane in the environment, where each cell (x_i, y_i) contains the height z_i of the top of the obstacle. This representation is very memory-efficient, using only $N \times M$ memory locations, where N and M are the dimensions of hirizontal grid slice. The huge drawback is that no vertical structure can be represented, including overhangs, ceilings etc.

Elevation maps have been improved upon by [19] in Multi-level surface map (MLSM). MLSM is able to represent multiple vertically stacked objects in a single cell. The map cell contains, for each object, a probability distribution of the estimated top of the object (as the main motivation of MLSM is for traversal), plus a "depth" that represents the vertical size of the object, which is the distance from the mean top value to the lowest point in the object. MLSM maintains the low storage complexity by storing sparse information about the vertical structure.

A full discrete representation of a 3D environment is possible in an occupancy grid. The environment is discretized into a 3D grid of voxels, typically cubes, of which the size of their sides corresponds to the resolution of the map. In each voxel, arbitrary data describing the corresponding part of the environment can be stored. Typically, this can be belief in occupancy of the cell when dealing with sensors of probabilistic nature. Dense grids are very memory hungry, needing full N^3 of memory space.

The dense occupancy grid can be stored more efficiently when represented in an octree. Octrees are fractal structures where each cubic cell can be subdivided into eight cubes with sides half as long. In the octree, cells whose subcells are for example all occupied need not be subdivided and can be represented by only one number. An octree can represent the same information as a dense grid, but at lower memory cost. An implementation of octrees for occupancy grids, called Octomap [20], is used in our software framework. Octomap is highly memory efficient, can be serialized and compressed to small data bundles and is designed to aggregate data from LIDARs and other range sensors in a probabilistic framework.

We will be dealing with relatively small environments, as the workspace of the arm is limited. Memory efficiency is not crucial. We do not need any probabilistic framework for map building, as our sensor is highly deterministic and we only need to represent free, full and unexplored environment. We can thus safely use the basic occupancy grid with benefits of simple implementation and without paying the dire memory cost of representing larger environments.

3.2 Exploration algorithm

We represent the environment in a 3D grid. The grid needs to be traversed by the arm with the sensing tool; every cell must be visited to determine if it is occupied or free (except the cells that are found to be unreachable because of obstacles).

We can formalize our grid as a graph where each cell is represented by a vertex. The robot then moves between the cells. To have each move cover exactly one cell, we allow transitions only between neighboring grid cells, and only from a cell to its 6-neighborhood.

To traverse the whole environment optimally, we would like to visit each cell in the grid exactly once. In other words, we would like to follow a Hamiltonian path through the grid. Planning such path through a general graph is NP-complete. In case of the free grid, however, the path is trivial: a zig-zag pattern filling the environment layer by layer[1]. When obstacles are present, the trivial solution is no longer possible.

As an approximation of a Hamiltonian path on the graph, we tried to plan the shortest path visiting all the vertices. We described the problem in an integer linear program that tries to compute the shortest sequence of actions that covers the whole space, avoiding known obstacles. The solution of the ILP proved to be intractable for instances larger than $4 \times 4 \times 4$ cells, as it contains quadratic number of variables compared to the number of cells in the environment.

RRT, random sampling motion planning algorithm mentioned above, has been modified in [21] to directly find a path that acquires sensor data from the whole environment. The new algorithm, RITA, samples the control space just like plain RRT. RITA however optimizes both environment coverage by sensors and path length. RITA shows promising results, but with sensors that cover large portions of space, like laser range finders. In our case, the path would need to be very dense as we have to use only contact sensor.

As finding an optimal path is highly complex, we considered solving the task with Reinforcement Learning (RL). In RL, the agent (the algorithm controlling the arm in our case) learns which actions to take in given state by interactive trial and error. We formalize a set of state S, and actions A. In our case, the state could be the current position of the robot and the map of the environment, and actions "move left" and "move forward", or directly joint velocities for the arm joints. We then define reward function $f : S \mapsto \mathbb{R}$ which specifies a reward the agent receives in given state. The agent learns a function $q : S \mapsto A$ that, for a state $s \in S$, returns action a that maximizes the reward the robot gets in the future. The problem in this case is the representation of the function, as the state space is extremely

[1] We will return to the zig-zag pattern in Section 4.2.3

large ($\{0,1\}^{N\times N\times N}$). The recently very popular deep neural networks have been used to represent a complex, high-dimensional function in RL [22]. This makes using RL in this case tractable, but highly complex. As the agent needs to interact with the arm to learn how to use it to explore space, it would be necessary either to allow the agent to operate the arm for a prolonged period of time while taking measures against damaging the arm in the process, or to build an extensive simulation of the whole system where the agent could try policies safely and gradually learn to operate the arm to fulfill its goals.

The problem of traversing the whole environment is not uncommon and arises in various applications. The problem is referred to as Coverage Path Planning (*CPP*) problem, and is aimed to be solved by clever algorithms that solve the highly complex planning problem in reasonable time by hiding some of the complexity in heuristic solutions. The algorithms can be generalized to a 3D world. The CPP is elaborated upon in Chapter 4.

Considering all the methods we presented above, we decided to solve the problem by Coverage Path Planning. It provides tractable, relatively simple solution to the problem of exploring the environment.

Chapter 4

Coverage Path Planning

Coverage Path Planning (*CPP*) is *"the task of determining a path that passes over all points of an area or volume of interest while avoiding obstacles"*[23]. The task comes naturally with different robotic applications, where the robot is to complete some task in every point in the environment.

The problem has been studied both practically and theoretically. In literature, the problem of planning an optimal covering route is referred to as the "lawnmower problem" or the "milling problem" and has been shown to be NP-hard [24]. The methods below either don't give any optimality guarantees, or hide the complexity elsewhere, as solving an integer LP.

4.1 Applications

The first such task that comes to mind is of course cleaning. This sometimes tedious task has been in focus of the robotics community for quite some time, with articles concerning this topic dating back to 1988 [25]. Other applications studied include painting, de-mining, automated agricultural vehicles and so on [23, sec. 1].

4.2 Approach families

Several different approaches to the problem appeared over time. A typical application takes place in a 2-dimensional, previously known environment. We will introduce several inspiring methods of 2D coverage, and then present a scheme to generalize them to 3D.

The robot moves in a defined workspace, a portion of space where it performs its tasks and which it cannot leave. The environment may be known

in advance. In the workspace, there (possibly) are obstacles which the robot needs to avoid in order to not damage itself.

4.2.1 Random strategies

The simplest coverage strategy is a random strategy. The environment is traversed completely randomly, and, hopefully, most of the environment will have been covered at some point in the future. Random strategies are commonly used in vacuum cleaning robots because they do not require any expensive special hardware and are easy to implement. As the robots are autonomous and work when their owner is not at home, optimal performance is not crucial.

Vacuum cleaners following random strategies have been shown to perform relatively well in terms of converging to fully cleaned room in reasonable time [26].

In our case, time is a scarce resource and we need to cover the whole area in the least time possible. The great advantage of random planning is the fact that it does not need any prior information about the environment, which is the characteristic we are looking for in our method of choice.

4.2.2 Heuristic methods

Next in line with simplicity are heuristic strategies. Conference paper [27] describes several such strategies. All of them are used in grid map representation.

The simplest strategy is closest-first, which randomly chooses an uncovered field adjacent to the current one; if none such exists, a BFS is performed to find the closest uncovered field and the algorithm navigates to it.

To improve the results of the aforementioned heuristic, we augmented it by providing it clues on which field to choose. Our heuristic and the rationale behind it is described below in Section 4.3.

Another heuristic method used is a modification of the WaveFront method [28] to unknown environments, called Iterated WaveFront. In the WaveFront method, the environment is traversed along cells equidistant to a goal state selected beforehand. This should ensure that all the far areas will have been covered when we get close to the goal, thus making the coverage as efficient as possible. Iterated WaveFront selects the goal state randomly, computing path in the currently known map. When an obstacle is encountered, the plan is recomputed over the updated map.

Heuristic strategies do not provide any optimality guarantees, and can

perform very badly in the worst cases. They are however very easy to implement, and can cope with unknown environments well.

4.2.3 Exact cellular decomposition

When the environment is known and can be represented, or at least approximated, by polygons or differentiable functions, exact cellular decomposition approaches can be used [23]. This family of approaches performs a decomposition of the workspace with obstacles into distinct cells. The cells are then traversed and each is covered by a trivial zig-zag pattern, sometimes called the *boustrophedon pattern*, an example of which can be seen in Figure 4.1. The traversal order is where the complexity is hidden in this technique.

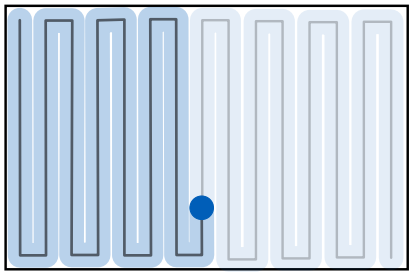


Figure 4.1: Example of the boustrophedon or zig-zag pattern. The environment is sequentially covered by the advancing sweeps

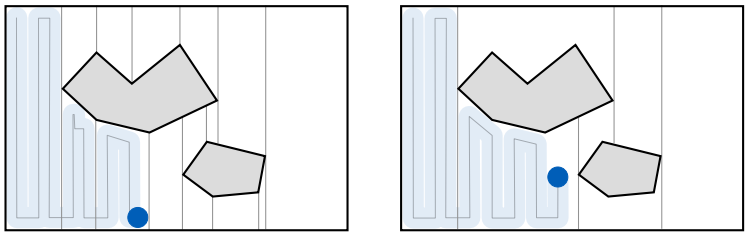
The two commonly used decomposition techniques are trapezoid decomposition and boustrophedon decomposition. Both decompositions work with the assumption that no two critical points[1] share the same x coordinate.

Trapezoid decomposition is usually used to index point location in planar decomposition. Here, only the exterior of obstacle polygons is decomposed. Then an adjacency graph of the cells is created and a path through the graph is computed, visiting every cell (vertex) at least once. As the path is followed, each cell is covered by the zig-zag pattern, and then the robot moves to the next cell.

The trapezoidal decomposition typically generates many adjacent cells that could be merged and still traversed trivially, but more efficiently, because less border areas will need to be handled. The boustrophedon decomposition is similar to the trapezoidal, only certain cells are merged together, eliminating suboptimal coverage of their boundaries. Examples of the two decompositions are presented in Figure 4.2.

An optimal algorithm, working as described above and based on the boustrophedon decomposition, is described in [29]. This class of methods is

[1] see [29] for definition of critical points



a: b:

Figure 4.2: Example environment decomposed by trapezoid (a) and boustrophedon (b) decomposition, with an example of covering path

best for perfectly known environments, which is not our case. We mention it to show how exact solutions in known environments can work.

A method for coverage of unknown rectilinear environments with a contact sensor called CC_r (Continuous Coverage of Rectilinear environments) has been described in [30]. It builds on the fact that while the traditional decompositions explicitly forbid rectilinear environments, as the critical points there do share the same x coordinate, they are easy to decompose themselves as all the cells in the decompositions will be rectangles. CC_r builds the decomposition gradually while exploring the environment. The authors give a proof of completeness in [30], but no statement on the optimality of the algorithm is given. The requirement of rectilinear environment is however very strong. Even though the grid environment representation we use is, strictly speaking, rectilinear from the viewpoint of the continuous space, it is not in the resolution of the grid.

4.2.4 Grid neural network

When the environment is represented in a grid, a neural network-like structure can be used to generate a covering path [31]. Each cell is represented by a neuron, connected to its 8-neighborhood. The dynamics of the network is described by the short term memory shunting equation for activity x_j of neuron j derived from the physical model of neuron [31, sec. III]

$$\frac{\mathrm{d}\,x_j}{\mathrm{d}\,t} = -Ax_i + (B - x_i)\left([I_i]^+ + \sum_{j \in N_i} w_{ij}[x_j]^+\right) - (D + x_i)[I]^- \qquad (4.1)$$

where $[x]^+ = \max\{x, 0\}$, $[x]^- = \max\{-x, 0\}$; A, B and D are model parameters, N_i is the set of neighboring neurons to neuron i and I_i is the external input of neuron i. According to the authors of the algorithm, the network driven by this model is a stable system proven to be bounded in $[-D, B]$ [32].

The external input represents the state of exploration of the environment. In unknown fields, it is set to a large constant E and thus creates high activity in the area. In known fields, it is set to 0, to allow external activities to propagate. In obstacles, it is set to $-E$, to locally repulse the robot. From Equation 4.1 we can see that positive activity propagates through the network, globally attracting the robot to more active areas, while negative activity is restricted to only the neuron with negative external input.

The robot moves through the network, following the highest activity while taking minimization of the number of turns into consideration. As it moves, it marks the visited fields and adjusts the network input. The network is numerically simulated, advancing time at each robot step.

The great advantage of this method is in its ability to represent and cope with dynamic obstacles. When an obstacle appears, the robot can only update the external input of the respective neuron and the map and trajectory will adjust. This allows us to run the algorithm in completely unknown environment. The algorithm does not give any optimality guarantees, and since its authors' experiments included a ranged sensor, while we only work with contact sensing, the algorithm will be making much less informed decisions in our case.

4.3 Compact space heuristic

The closest-first heuristic is the obvious trivial heuristic for the CPP. It only needs to make a reasonable guess on where to go next.

We designed a simple rule the robot follows. We define the heuristic function f on cell x, which has a set of neighboring cells N_x, as

$$f(x) = \begin{cases} |\{i \in N_x | i \textit{ has been explored}\}| + |\{j \in N_x | j \textit{ is a known obstacle}\}| \\ \qquad \text{if } x \text{ is unvisited} \\ 0 \quad \text{otherwise.} \end{cases} \tag{4.2}$$

The robot in cell c goes to the unvisited cell

$$n = \arg\max_{j \in N_c} f(j).$$

In case of a tie between candidates PN_c, the robot tries to decide it by looking at their neighboring cells and choosing candidate

$$n = \arg\max_{j \in PN_c} \sum_{k \in N_j} f(k).$$

If a tie occurs here as well, a candidate is chosen randomly.

If all the neighboring cells have been visited, the robot goes to the nearest unvisited cell.

This heuristic makes sense from multiple points of view. For one, it encodes the common-sense ideas about how to explore an unknown area. We would like to keep the already explored space as compact as possible, to eliminate future need to come back to inspect a few unexplored fields left behind, thus maximizing the number of already visited neighbors of the next cell. The second term encodes the idea that cells next to an obstacle have a higher probability of being obstacles as well, and it is beneficial to explore them sooner to get this knowledge as early as possible and to be able to exploit it.

The heuristic is also closely connected to the well-known heuristic for the knight's tour (or any other Hamiltonian path) problem. We would like to get a Hamiltonian path, as it would minimize revisited cells and thus path length. The original Warnsdorff's heuristic tells us to *"select the move which connects with the fewest number of further moves"*[33]. Pohl's heuristic, based on Warnsdorff's, only adds a tie-breaking rule, applying the idea recursively: *"choose the field whose neighboring fields have the fewest ways to continue"*[33]. Our heuristic follows this rule exactly: maximizing the number of already explored neighbors is equivalent to minimizing the number of possible next moves. The only difference is that the aforementioned heuristics prioritize cell on the edges and corners of the environment, as they have intrinsically less neighbors, while our heuristic takes them as ordinary cells.

4.4 Generalization to 3D

While the grid-based methods presented above can directly be generalized to 3D by applying them to a 3D grid instead of 2D, the exact methods presented rely on cellular decompositions that are relatively easy to compute in 2D but complex in 3D. This makes their direct generalization to 3D impractical. Most effort in 3D coverage has been put into covering 2D surfaces in 3D space, such as car parts (that need to be covered in paint) or more complex terrain on crop fields (that need to be covered by harvesting machines). We however actually need to cover all the cells in a 3D environment.

In our case, the exploration also needs to take into account the kinematic constraints on the arm pose, as we must not allow any part of the arm except the contact sensor to enter unknown areas to prevent possible hardware damage.

Any planar covering algorithm can be extended to 3D by being applied sequentially to consecutive planes. The idea has been described in [34], albeit in the context of covering the ocean floor.

In a continuous case, we divide the workspace along one dimension into a

collection of planes spaced in intervals corresponding to the robot's sensing diameter. This way, when the robot completely covers two consecutive planes, the space in between them will have been covered as well.

When we represent the environment in a grid, we generally choose to use one dimension of the grid, and then cover each layer of the dimension with a planar grid algorithm. In the case of our exploration task, we choose to divide the grid into vertical slices (perpendicular to the arm when it reaches forward), each of which the robot searches in turn, progressing from the closest to the furthest of the planes. This sequence of planes leaves the space between the robot and the currently explored cell known, as it has been searched before. This simplifies reasoning about the environment in which the robot moves and works.

4.5 Method comparison

Of the methods presented above, the ones applicable to our problem are the neural network approach and the heuristic approach. The exact methods need to know the environment completely, to make sure it is traversed optimally.

A possible approach to extend the exact methods to our task is to iterate them, replanning the path every time an obstacle is encountered and hence every time the environment the robot assumes changes. This would be impractically time-consuming, as the replanning would happen very often compared to the time the actual execution of the plan would take.

The heuristic approaches and the NN approach share the common scheme of selecting a path locally, and moving to the closest unknown cell if no such path can be found.

In the heuristic methods, the criterion upon which the local decision is based is strictly local and the global view is taken into account only when the local fails.

In the NN, the global signal propagates through the whole environment and thus the state of the whole map impacts the local decision. No hand-crafted global strategies are necessary. In the end, however, the strategy still boils down to local exploration followed by moving to the globally closest unexplored place when there is nowhere else to go.

The choice of the nearest unexplored place is implicit in the case of neural networks, as the dynamics of the network drive the propagation of activity from unvisited areas such that the signal gets weaker with every step, resulting in the robot going to the closest source of activity via the shortest path when traversing already visited fields.

The heuristic methods have the choice of the place whence to continue the

exploration hand-designed in them. This way, we can design the choice to best suit our needs. In our case, this is an advantage, as, because we are working with a robotic arm in three dimensions, we can get to all the fields in our environment directly, without being constrained by planning in a grid with obstacles.

Chapter 5

Implementation

We divide the environment into a grid $10 \times 10 \times 6$ (width $\times$ height $\times$ depth), from the viewpoint of the robot. A cell of the grid is a cube with its side 10 cm long; the whole grid is positioned in front of the robot and is centered horizontally, the lowest cells are 10 cm above the robot base and the first layer of the grid is 0.5 m far from the base. To allow the robot to perform any moves at all, we assume that all space between the robot and the explored space, including the space behind the robot, is known, which is a reasonable assumption, since the mobile robot supposedly came from there and knows the environment. We further assume a static environment is to be explored.

The robot will move the sensing tool through the grid, following a CPP algorithm. The algorithm aims to minimize the path length in the Cartesian space. Once contact with obstacle is detected by the instrument, the motion will stop, the obstacle is registered in both the mapping algorithm and in the robot driver as a confirmed obstacle. The robot then recovers from the touch and proceeds with the exploration. The robot will always avoid colliding the arm with known obstacles, including the mobile robot it is mounted on. It will also avoid reaching into the unexplored space by other parts than the sensing tool. The sensing tool needs to be included in the collision checking model to avoid obstacles and to only be allowed to enter the space it is to explore.

Making the arm move proved more difficult than expected, with motion planning in tight spaces failing frequently. We managed to offload the motion control to the Jacobian controller, but the planned motion is still unstable and sometimes reports perfectly reachable cells as unreachable.

5.1 System architecture

The architecture of the system is designed to fit into the modular ROS framework. Each functional part is implemented as a standalone ROS node, which can be used independently on the rest of them. The architecture is illustrated in Figure 5.1.

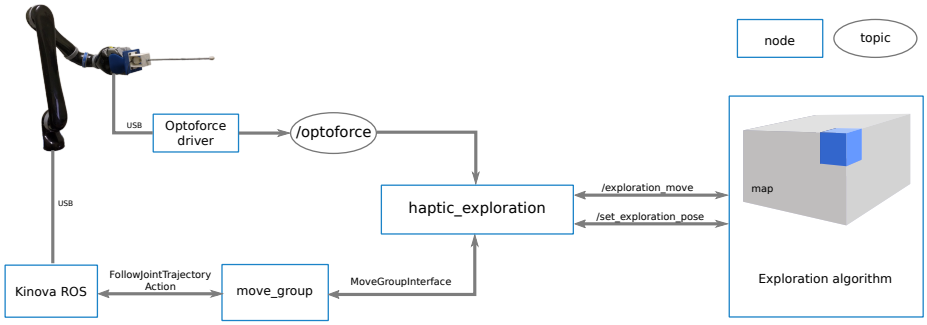


Figure 5.1: Overview of the system architecture

A node runs the optoforce driver and publishes the force readings to a common topic. If any other component wishes to use them, they are available system-wide.

The interaction with the robot and the MoveIt-maintained planning scene is done through `haptic_exploration` node. The node exposes services that control the robot motion and maintains the state of the arm in the exploration grid. The node runs in one process that listens for exploration requests and carries out the desired motions, while simultaneously processing the data from the Optoforce sensor. As the node needs to process the incoming messages, execute services and publish robot instructions simultaneously, the node event loop is carried out by a `AsyncSpinner`, which starts several event-processing threads. The concurrency is automatic, but care needs to be taken when processing the event callbacks in different threads to prevent synchronization issues. The node depends on the `move_group` node running with all its configuration loaded, and on the Kinova ROS driver node.

The exploration algorithm runs in an independent process. It communicates with the main node by calling its services for arm movement, and is informed about the results in the service return messages. In this program, the map of the environment is constructed, and, based on it, the appropriate movements are sent to the main node.

5.2 Optoforce driver

The Optoforce driver node is implemented in Python. Our implementation is based on an open-source driver [18].

The driver connects to the emulated serial port on the system, which requires the user to have sufficient privileges to do so. The sensor is first configured to send the measurements at 100 Hz, together with parameters of the data pre-processing done in the DAQ unit, like filtering.

The driver then starts receiving measurements over the serial line, which are decoded, scaled to Newtons by the calibrated sensitivity scale provided by the manufacturer. Then, the values are published as a `Vector3` message to the `\optoforce` topic.

5.3 Robot motion

We mentioned two different ways of driving the arm in Section 2.3. As both have their pros and cons, we implemented both of them and tested them to decide which one to use. The experiment testing the implementation is described in Section 6.4.1.

The control code is implemented in C++ to ensure optimal performance in the critical control loop, as well as to leverage the more feature-rich API that MoveIt exposes through the C++ module `MoveGroupInterface`. A python interface exists as well, but has limited capabilities.

Collision checking is performed to prevent arm and tool damage. The unknown space is filled with collision cubes, one for each grid cell. This ensures the arm stays safely away from the potential obstacles. The tool is represented by a collision cylinder the size of the stick attached to the end effector. All collision checks in both motion control methods include the tool so that it does not get damaged. Only when a cell is to be explored, its collision cube is removed to allow the stick to enter it and explore it. A better way to implement it would be to modify the allowed collision matrix to allow collisions only between the stick and the cube, instead of anything and the cube as it is now. MoveIt however suffers heavily from synchronization issues, as the structures live in both the `move_group` and the user process, and modifying the collision matrix used in planning appears impossible. If an obstacle is detected in the cell, or if all the planning attempts fail, the cube is added back into the environment, and an unreachable cell is reported.

The robot control services are implemented in the main node. They can however be easily separated into standalone units that can be reused in other projects that require driving the arm.

5.3.1 Jacobian-driven motion

As we wrote, for a Cartesian twist (linear and angular velocity) vector $\mathbf{v}$, we can obtain joint velocity $\dot{\mathbf{q}}$ which will result in end effector movement with the given twist. This can be done by calculating the Moore-Penrose pseudoinverse of the Jacobian matrix, preferably through SVD. Details are given in Section 2.3.6.

The whole process of obtaining joint velocities for a given twist is already implemented in MoveIt, including Jacobian computation. The method is a part of MoveIt C++ library class `RobotState`, which represents the joint configuration of the manipulator. MoveIt expects the twist to be in the end effector reference frame, and thus the Cartesian twist in the world coordinates we use needs to be transformed into that reference frame first. The computation is implemented exactly as stated above, with checks on the singular values during the pseudoinverse computation to guarantee numerical stability of the approach. Near singularities, even the correct results contain very high joint velocities that need to be corrected in order to be executed on the robot, whose joint velocities are limited.

When joint velocity limit of $0.8\,\mathrm{s}^{-1}$ is violated, we attempt to resolve it by slowing the end effector twist down to 50% of its original speed. This check is applied until the joint speeds are within limits, or at most 4 times. If the joint speeds are still too high, the motion stops immediately and a failure is reported for the command issuer to do something else.

The complete algorithm is presented in pseudocode as Algorithm 1.

The motion is controlled in a control loop running at 50 Hz. In each iteration, we first compute the twist we need to apply. It consists of a constant input velocity the robot is requested to follow, and a correction term determined by a P regulator that corrects arm deviations from the desired trajectory. Then, MoveIt computes the joint velocities that will result in the computed twist in current arm configuration. The joint velocity checks stated above are then applied, including the eventual update of the twist and joint velocity re-computation.

When the output joint velocities are known, the robot motion with those velocities until the next iteration is simulated. The state expected to be reached by next control loop iteration is checked for collisions, both self- and with the environment. If a collision were to occur, the motion is stopped immediately.

When the final joint velocities are computed and verified, they are published to the robot driver topic controlling joint velocities. The same velocities are published twice to match the control loop frequencies (see Section 2.2.5).

The motion stops when the desired end point is reached. Currently, the

Algorithm 1 Jacobian motion control

Input:
$\mathbf{v} \leftarrow$ Cartesian twist
$\mathbf{p}_t \leftarrow$ target position

while $\mathbf{p}_t$ is not reached with tolerance ϵ **do**
  $\mathbf{p} \leftarrow$ current arm state ▷ Represents both joint angles and end effector position
  $\mathbf{e}_c \leftarrow$ Cartesian error of $\mathbf{p}$
  if $\mathbf{e}_c > \epsilon_{max}$ **then**
    return failure
  end if
  $\mathbf{v}_t \leftarrow \mathbf{v} + P\mathbf{e}_c$ ▷ P is P-regulator constant
  $\dot{\mathbf{q}} \leftarrow$computeJointVelocity$(\mathbf{p}, \mathbf{v}_t)$

  while checkVelocityLimit$(\dot{\mathbf{q}})$ fails or at maximum 4 times **do**
    $\mathbf{v}_t \leftarrow \frac{1}{2}\mathbf{v}_t$
    $\dot{\mathbf{q}} \leftarrow$computeJointVelocity$(\mathbf{p}, \mathbf{v}_t)$
  end while
  if checkVelocityLimit$(\dot{\mathbf{q}})$ fails **then**
    return failure
  end if

  $\mathbf{p}_n \leftarrow \mathbf{p} + \dot{\mathbf{q}}\Delta t$ ▷ Δt is the duration of one loop, here 20 ms
  if collisionCheck$(\mathbf{p}_n)$ reports collision **then**
    return failure
  end if

  publishJointVelocityTwice$(\dot{\mathbf{q}})$
  run loop at 50 Hz
end while
return success

robot can be driven through cells of the grid defined by the exploration algorithm, and only move along one axis (to the 6-neighborhood of the current cell) by a given number of cells. The algorithm could easily be extended to follow arbitrary lines. The algorithm also does not try to error correct orientation changes.

The motion is initiated by a calling service `/exploration_move`, which is called by message `ExplorationPointRequest` which has three integer parameters x, y and z. They specify the direction and number of cells the robot is to move by. Only one of the numbers can be non-zero, the service call will fail otherwise. The response contains two boolean fields: `reached` that indicates if the target cell was reached, and `obstacle` that defines whether an obstacle was detected by the sensing tool. If `reached` is false, the content

of `obstacle` is undefined.

5.3.2 Planned motion

The planned motion control is implemented using the user-friendly interface of MoveIt, the `MoveGroupInterface`. The motion is started by calling service `/set_exploration_pose`. The request is, as with Jacobian control, an `ExplorePointRequest` with x, y and z fields. Here, the fields specify the point in the grid that is to be explored by planning a path that reaches it. The output is the same as in Jacobian control.

When the request is received, the target pose is specified to MoveIt, collisions of the stick and the target cell's cube are allowed and a motion plan is computed. We use the LBKPIECE motion planner provided in MoveIt. If the planning is unsuccessful, it is retried at most two times (plans may fail to be found in close proximity to obstacles for example). If no solution is found then, an unreachable cell is reported to the exploration algorithm by returning `reached` as false.

Once the plan is computed, movement is initiated through the `MoveGroupInterface` and the node waits for its completion. If the execution was successful, the exploration can proceed and a free cell is reported. If an obstacle was detected during the motion (see Section 5.3.4), the fact is propagated to the caller and the movement stops immediately.

5.3.3 Hybrid solution

The algorithms we use employ both transitions to a neighboring cell and long moves to distant cells. To implement short transitions, we use the Jacobian control in conjunction with the planning as fallback.

We first try to complete the movement by Jacobian control. If the movement fails, either because it encountered a singular configuration of the arm, because the arm would collide with an obstacle or because joint limits, both angular and velocity, would be violated, the algorithm returns back and tries to reach that cell again by planning the motion, possibly several times. Note that on contact detection by the instrument the movement is stopped as well, but it is a success.

When exploring the cells in the lower part of the arm workspace, it is beneficial to point the sensing tool downward. The reasons are twofold: one, the wrist rotation with respect to the third arm link is limited, and pointing the instrument forward could be infeasible, and two, the instrument can then sense the ground below the robot without crashing into it first. The orientation change cannot be handled by the Jacobian controller because

such control was not implemented. We instead plan the motion, also using the plan that reaches an exact location to correct any errors that would have accumulated during driving the arm by the Jacobian controller.

5.3.4 Stopping the movement

The main node is subscribed to the force readings published by the Optoforce sensor. The messages are processed asynchronously to all other operation by the `AsyncSpinner`.

The force readings are checked against the thresholds of 2 N for lateral and 4 N for axial forces. If the thresholds are exceeded, the node enters stop state and all motion is stopped. If there is a plan execution currently in progress, it is stopped via the `MoveGroupInterface`. The Jacobian-controlled movement is stopped in the control loop if the node is in stop state.

If the motion was planned, the arm traces its steps back by following the motion plan backwards until the tool leaves the newly classified obstacle. Then, the exploration algorithm can determine the next cell to be explored and request a planned motion to it (Cartesian control cannot be used, as we are not in a grid cell when we left the obstacle).

When the motion is stopped during Jacobian controlled motion, the robot then reverses the motion and returns back where it came from. As the robot was there previously, the movement is necessarily safe. Nonetheless, all the collisions are still checked.

Once the contact between the tool and the obstacle ceases and the forces drop below the thresholds, the main node returns to normal state. We use the same thresholds here instead of lower values causing hysteresis because we do not base any action on the force drop, we only continue in the retracting movement.

5.4 Exploration algorithm

We implemented both the CPP algorithms we mentioned as feasible in Section 4.5, the compact space heuristic and the neural-network (NN) algorithm. Only the heuristic algorithm is adapted to drive the arm; the neural-network algorithm did not show promising results in the simulated test 6.2. The algorithms are implemented in python and are at the moment run as commands by the user. They can easily be transformed into actions or services that are called by other components of the system.

Both the algorithms share the same basic structure. The map is represented in a 3D integer array. The arm workspace is specified to the algorithms by

marking the unreachable fields as obstacles before the algorithm starts. All other cells are set to the base value of 0. The environment is searched layer by layer; in the y direction in our case.

5.4.1 Heuristic algorithm

In each layer, the heuristic algorithm explores all the cells sequentially. The current cell is marked in the map as empty, as obstacle or as unreachable based on the outcome of the effort to reach the cell. The information about the current cell is propagated to the neighboring cells, adding 1 if the current cell is empty and 2 if it is an obstacle. This way, each so far unexplored map cell holds the value of the heuristic function defined in Equation 4.2. Then, the next cell is found by simply choosing the neighbor with the highest heuristic value and breaking the eventual ties as defined in 4.3.

If no neighboring cell is unexplored, the nearest such cell is found and set as the next exploration target. If no such cell exists, the robot moves to the nearest free cell in the following layer.

5.4.2 Neural-network-inspired algorithm

The NN needs, beside the map, another two arrays that represents the activity of the neurons in the network and their external inputs. Only one map layer needs to be simulated at a time, so 2D arrays are in order. The activity array is initialized to zeros, while the external inputs are initialized to the corresponding map layer. The parameters of the method are set according to the original article [31]: $A = 50, B = 2, D = 1, E = 100$.

We made a few changes to the algorithm. First, we do not take direction changes of the robot into account as does the original algorithm. The article fails to specify the weight with which the turning angle is to be added to the activity anyway. Second, when all the neighboring cells are either already explored or obstacles, we move the robot to the closest unexplored cell as in our heuristic algorithm. The original is meant to be used with whole robots in planar environment, and as such could not move the robots around at will as we can do with the robotic arm.

The neural network is modeled by differential Equation 4.1. As we decide to move the robot based on the activity, we need to model the behavior of the network in time to simulate the activity propagation and compute the appropriate values for neuron activity. We simulate the activity function by the Runge-Kutta 4th order method, which has a great ratio between performance (high precision) and implementation complexity (low).

We simulate the network in 0.05 s steps. In the beginning of each layer,

we run the simulation for 100 steps to let the activities stabilize. Then in each step, the robot chooses the neighboring cell with highest activity. If all neighbours have lower activity than the current cell, and are thus already explored or obstacles, we move the robot to the nearest unexplored cell. If there is no such cell left in this layer, the robot moves to the following layer.

5.5 Integration with the Absolem robotic platform

The software implemented in this thesis is designed to be integrated into the TRADR project, to use the arm when mounted on the mobile robot platform and explore space as needed by the whole robot's goal. The main concern with the integration is obstacle avoidance for the arm, which needs to take the platform itself into account. That should not pose a problem, as complete collision checking is performed even now, and it is only a matter of adding the platform collision model into the environment.

The second matter is the interface via which the whole exploration will be invoked. This will need to be agreed upon by the designers of the components that are to be the initiators of the exploration, including the map format to be returned; if it is to be a point cloud, a representation of the grid or something completely different.

Chapter 6

Experiments

We conducted both simulated experiments and experiments on a real robotic arm to test our solution and its implementation. First, we experimentally tested the new Jacobian control mechanism we implemented, to determine if it was usable for our solution. Then, the coverage path planning algorithms were tested in simulation first. Finally, the compact space heuristic algorithm was put to the test both in simulation and with the real arm.

6.1 Benchmark environment

We created a benchmark environment representing an obstacle in front of the robot. The environment is represented by a grid $10 \times 10 \times 5$ cells. We also tested the algorithms in free space, to see if they can produce optimal (non-overlapping) paths at least in the simplest of environments, despite their heuristic nature.

The environment contains two obstacles, a floor-standing object and an overhang above the object, with enough space for the robot to pass through. A single plane, the second nearest vertical plane to the robot, extracted from the full 3D environment is used as the 2D benchmark for the coverage algorithm. The planar environment is illustrated in Figure 6.1.

In simulating coverage algorithms we ignore the potential arm collisions with previously discovered obstacles. This is accounted for when we simulate the whole arm.

6.2 Simulated coverage algorithms

To test the performance of CPP algorithms we implemented, we performed experiments to determine how well the methods perform in unknown environ-

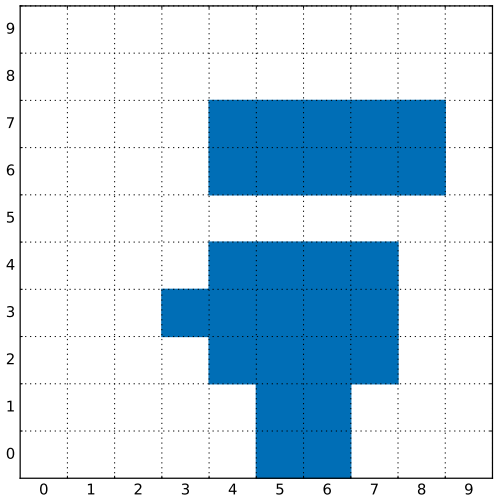


Figure 6.1: The 2D benchmark environment for CPP algorithms. The environment is depicted as a vertical slice as seen by the robot; the "floor" is at the bottom

ments in both 2 and 3 dimensions. Each of the algorithms was run in empty 2D environment to test it in the simplest setting, followed by a run in the benchmark environment described in Section 6.1 Because of the generalization scheme used to adapt the algorithm to explore 3D space, 2D environment is represented in 3D as a grid with only one depth layer.

6.2.1 Two dimensions

Random algorithm

To establish a baseline to test against, we first tested a basic random algorithm. One of the generated paths can be seen in Figure 6.2.

The algorithm randomly chooses an unexplored neighboring field; if no such is available, it goes to the nearest unexplored field. The lengths of the displayed paths are 12 m in the empty and 13 m in the benchmark environment, while the mean lengths of paths generated over five trials were 11.7 m (min. 11.4 m, max. 12.1 m) and 12.8 m (min. 12.5 m, max. 13 m), respectively.

Compact space heuristic

Running the compact space heuristic algorithm in the 2D environment, we obtained paths depicted in Figure 6.3.

The results in empty space are reasonable. The generated path closely

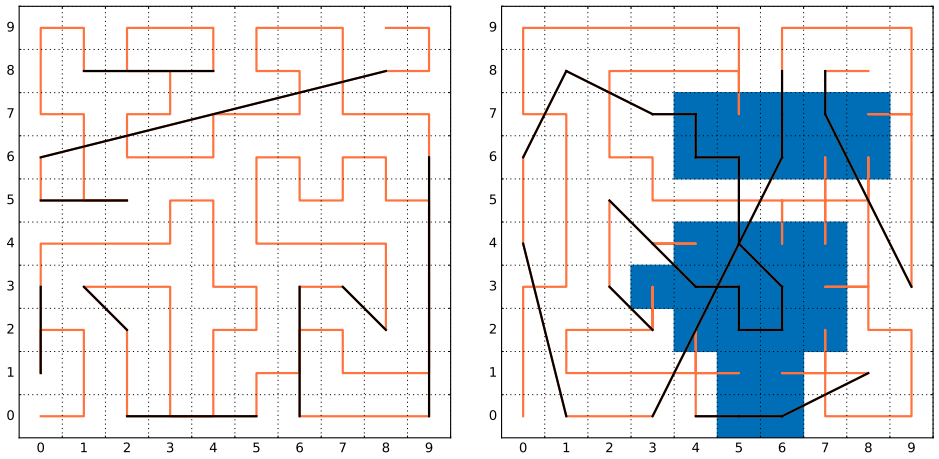


Figure 6.2: Coverage path generated by a random algorithm in an empty (a) and in the benchmark environment (b). The path starts at $[0, 0]$ and is depicted in orange, with transitions to nearest unexplored field depicted in black. The global transitions sometimes cover the previous ordinary ones

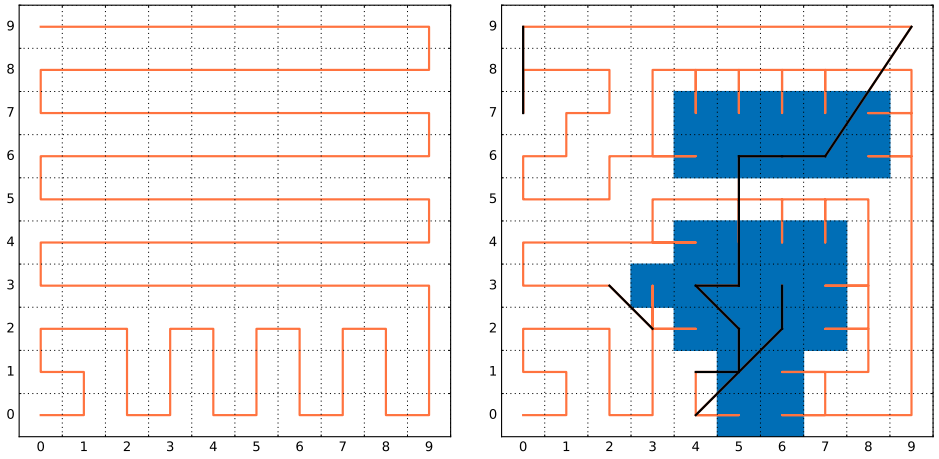


Figure 6.3: Coverage path generated by the compact space heuristic in an empty (a) and in the benchmark environment (b). The path starts at $[0, 0]$ and is depicted in orange, with transitions to nearest unexplored field depicted in black

resembles the boustrophedon pattern used in human-designed algorithms. The already explored space is kept compact, as was the aim of the heuristic. The length of the generated path is 9.9 m, which is the optimal value since no cell was explored twice. This is allowed by the even size of the environment; in case of odd environment width, the lower zig-zag terminates in the lower-right corner and the robot then has to return back to the yet unexplored portion of the environment, traveling through the already known fields again.

In the benchmark environment, we can see that the path starts the same as in the empty environment. When the obstacle is encountered, we can identify the algorithm following the other incentive it has – exploring space near known obstacles. This leads to obstacle-circling, after which the obstacle limits are quickly determined. A snapshot from exploration progress is displayed in Figure 6.4. The length of the path is 12.5 m, better than any run of the random algorithm.

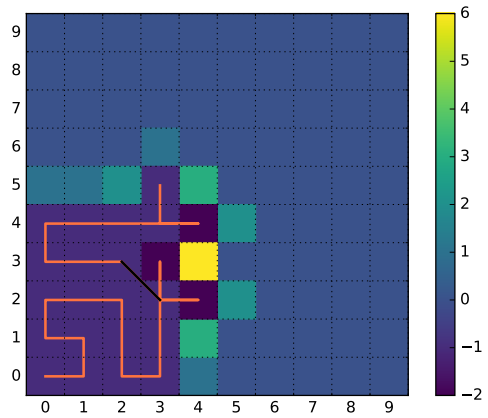


Figure 6.4: Intermediate state of exploration of the benchmark environment by the compact space heuristic. The tool is now at position (3, 5). The fields with value −2 (dark blue) are already discovered obstacles, the fields with −1 were identified as free. The other values represent the value of the heuristic function in the cell. As we can see, the robot will continue right as cell (4, 5) is adjacent to an obstacle, as opposed to (2, 5)

Neural network

The results of the NN algorithm are presented in Figure 6.5.

The NN algorithm tests have been described by its authors, an example of a path generated by the original implementation is presented in Figure 6.6. The trajectory we obtained is completely different, even with the algorithm without the modifications described in 5.4.2. No tuning of parameters achieved such path. The authors do not give any details on their implementation of the network dynamics simulation, and that may be the reason for such failure. We present an insight into the algorithm decision making in Figure 6.7, where

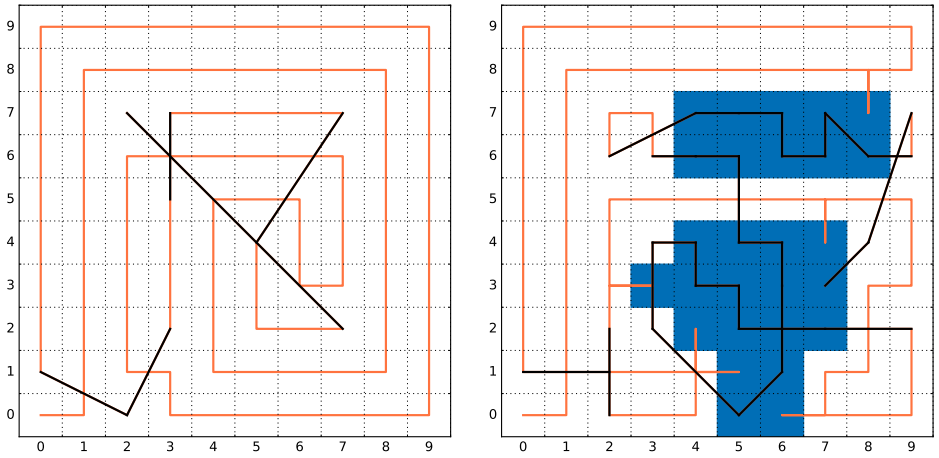


Figure 6.5: Coverage paths generated by the neural network algorithm, in empty environment (a) and the benchmark environment (b)

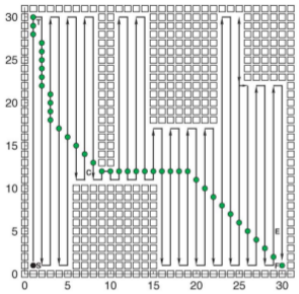


Figure 6.6: The path obtained by the authors of the algorithm (image taken from [31])

the neural activity evolution during exploration is presented. More detail on the activity development can be found in the original article [31].

The path output by our implementation fails to cover even empty environment without overlaps. The path is 11.11 m long and includes 5 transitions to the nearest unexplored cell.

In the benchmark environment, the generated coverage path is 11.75 m long, surprisingly less than in the case of the compact space heuristic. The reason is that the path in the compact heuristic contains many simple moves hitting the obstacles and then returning back to the free space. In the NN case, most of the obstacles are visited by the global transitions, which follow the shortest paths, decreasing the total path length. The global transitions are driven by motion planning, and as such their preparation and execution is more time-consuming than that of the simple motions from one cell to its neighboring cell.

In conclusion, the compact space heuristic provided the most efficient

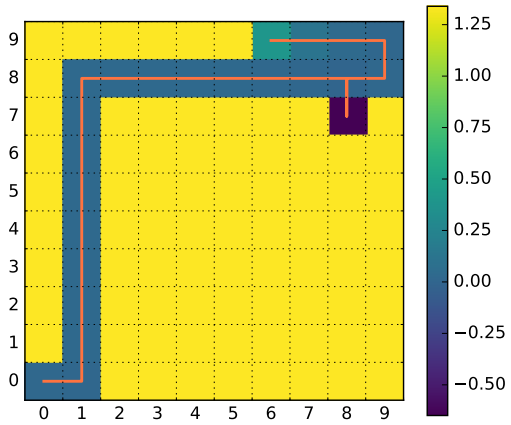


Figure 6.7: The neural activity in the network after a few algorithm steps. The deep blue obstacle and yellow unexplored area are pulled down and up by the external inputs. The robot is at position (6,9); we can observe the neural activities in the robot's trail decaying after setting the external input to zero

solutions to path planning in 2D. The neural network algorithm produced competitive trajectories that however contained many global transfers and were thus more complicated and time-consuming.

6.2.2 Three dimensions

The results of running the coverage algorithms are shown in Figure 6.8. We did not evaluate the random planner in 3D as the performance of the 3D instance should, due to the generalization scheme employed, be directly related to the performance in 2D, where it was outperformed completely.

Compact space heuristic

Expectedly, the algorithm runs in 3D space as well as in two dimensions. In both cases, we can observe the layers in which the environment has been explored. In the empty environment, the algorithm starts and ends on opposite sides, and thus the layers are identical, only mirrored. The found path in Figure 6.8a is again optimal with length of 49.9 m.

When obstacles are present, the coverage path is different in each layer partly because of the varying obstacles, partly because of different entry points, as one layer starts precisely where the other has ended. The path in Figure 6.8b is 56.3 m long.

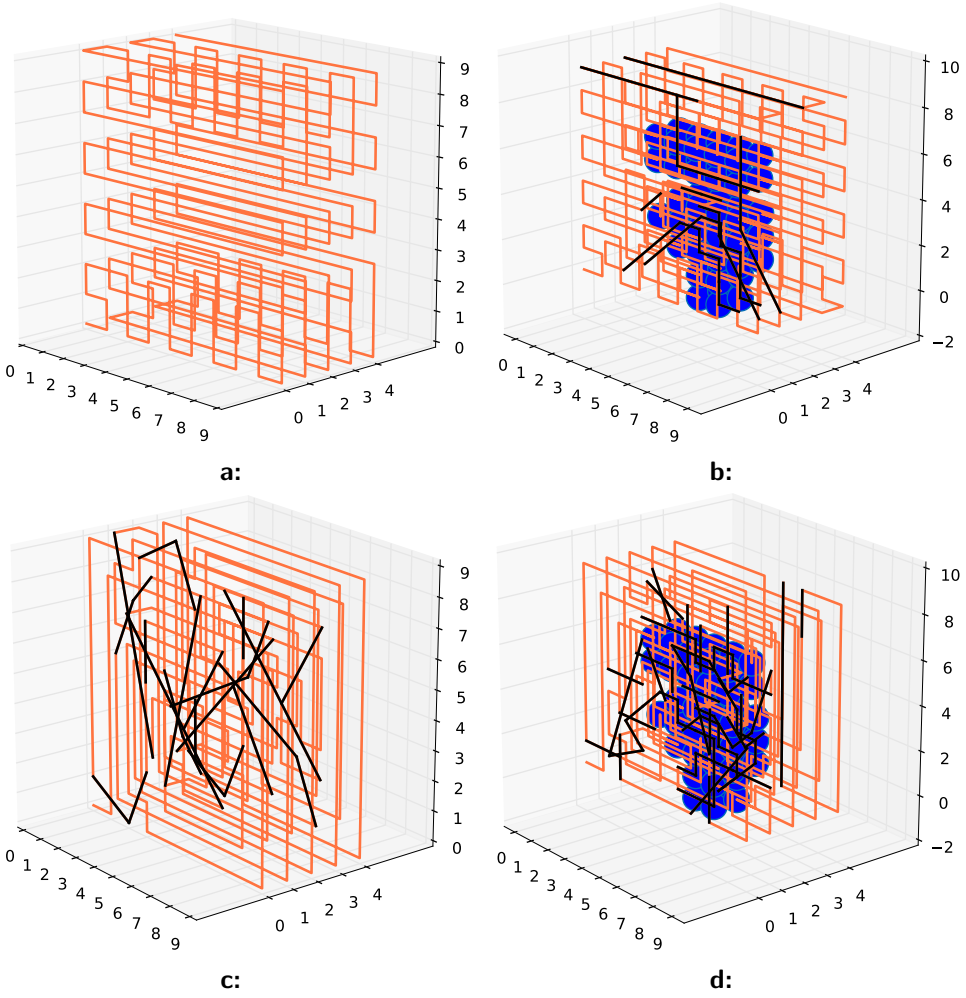


Figure 6.8: The coverage path generated by compact space heuristic in 3D in (a) empty space and (b) benchmark environment. The path by NN algorithm is in subfigures (c) and (d) in empty and in the benchmark environment, respectively. The obstacles are represented by spheres at the given point

Neural network

The results of the NN algorithm are irregular, as the algorithm proceeds in a sort of a spiral, and thus ends in one layer and starts in the other in central locations. The path in Figure 6.8c is 55.2 m long and contains too many global transitions.

In the environment with obstacles, the generated path in Figure 6.8d is 57 m long.

Algorithm	Compact space	Neural network
2D free	**9.9 m**	11.1 m
2D	12.5 m	11.75 m
3D free	**49.9 m**	55.2 m
3D	56.3 m	57 m

Table 6.1: Generated path lengths in the simulated experiments. Numbers in bold represent known optimal solution

6.2.3 Conclusions

The results of our experiments are summarized in Table 6.1.

In all cases, except in the 2D benchmark environment, the compact space heuristic provided more efficient path than the neural network. In the one case, it was at the expense of many global transitions. Hence, we will use the compact space heuristic as the exploration algorithm for the rest of the experiments.

6.3 Simulated arm experiments

6.3.1 Moving the arm

In Sections 2.3.5 and 2.3.6, we mention two ways of driving the robot: motion planning and Jacobian transformation. Both have their advantages and disadvantages. We tested them by running the algorithm with each as the robot driving function. We used only the simulated arm at first to protect the hardware from unfortunate accidents.

We let the simulated arm explore empty space. We evaluated the failures of the movement, such as getting stuck, and the smoothness, precision and optimality of the overall path.

The error measurements both in this section and in Section 6.4.1, which describes the errors measured on the real arm, are based on position reported by the robot. The joint position sensors in this simulation have only rounding errors. The real arm position is reported by rotation encoders in arm actuators, that have angular resolution of 0.068° to 0.047° (three different actuator types are used in the arm), with unknown error [35].

The motion planning method is very reliable in always reaching the desired target pose, and reaching it precisely, when performing a step in the algorithm in the empty environment. In the test run, all the steps were carried out successfully. The planning took hundreds of milliseconds for each step. Even

when using the TRAC-IK solver, sometimes a joint configuration very different from the current one was found as the IK solution by the solver. The planner then had no other option than to plan a more complicated path, perhaps rotating the first joint by 180°.

That being said, the planning success rate dropped drastically when obstacles were present in the planning scene, and in tight restrictions, it often failed completely as the planner was unable to find a way out of the restriction. Mere proximity of obstacles also posed a challenge to the motion planner. If the path was found, which occured as infrequently as once in ten planning trials, the generated paths were messy to say the least. An example of such path can be found in Figure 6.9. In some instances, alarmingly, a plan that caused collision was allowed, because the discretized poses in which collisions were checked skipped the one in collision. The overlap was only very small and short, but could cause a problem in real arm operation.

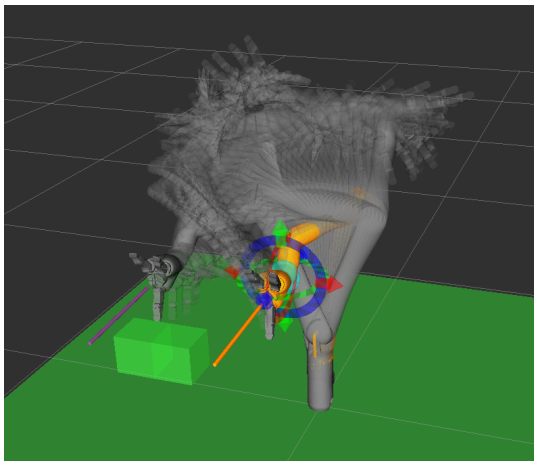


Figure 6.9: An example of a not at all optimal plan, the only one found due to a close obstacle only after five planning attempts. The start state is solid grey, the requested target is orange. The planned path is visualized as a sequence of semi-transparent intermediate arm states

The Jacobian method is extremely fast. The pause at each step is only a few milliseconds, the arm seemingly does not stop at all. During the tests, we noticed slight deviations of the arm from the trajectory it was to follow, and conducted further experiments to determine their nature.

6.3.2 Jacobian control precision

We tested the control algorithm by driving the arm across its workspace along the x axis. The arm followed the given trajectory with tolerance of 2 mm. To further improve the here completely satisfactory result, we turned on the error correcting term described in Section 5.3.1. The precision then got near perfect, with error of at most 0.4 mm. The errors are depicted in Figure 6.10.

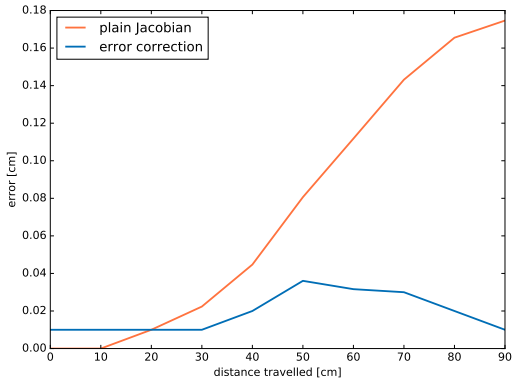


Figure 6.10: Position error during simulated test drive

The error is probably caused by the numerical error attained in integrating the speeds in discrete steps in the control loop.

In some cases, mostly directly in front of the arm base, a near singularity is reached. The tested trajectory passes in front of the arm's base at 45 cm. We can see the effects of the singularity in the plot. While we attempt to solve things by slowing the arm down (see the implementation details in Section 5.3.1), in some configurations, we get so close to the exact singular configurations that no slow-down helps. The driving algorithm identifies such situations and reports failure to higher layers.

The final combined solution described in Section 5.3.3 proved to combine the best properties of both the methods. The fallback to planning is used exactly when necessary. In some cases, however, the Jacobian motion stops due to a collision warning, but it stops too close to the obstacle. Then, even planning fails to move the arm to explore the next cell and it is misclassified as unreachable. This results in false positive obstacles identified in the environment.

6.3.3 Exploring 3D environment

We tested the final coverage algorithm first in the simulator. We ran it first in empty environment, then in the benchmark environment. The benchmark environment was simulated by publishing false force readings when the arm explored an occupied point.

Initially, we ran the algorithm without checking for collisions between the stick and other explored cells, to verify that at least the simplified version can be solved. We ran the algorithm several times, as the motion planning is not deterministic.

In every run in the free environment, a few cells were reported as unreachable due to planning algorithm failure. When the arm was close to a collision cube

of another non-explored cell, the motion planning failed frequently. This was the case very infrequently, though. The arm itself stays safely away from the obstacles, and only the stick is meant to get so close. However, once a cell is marked unreachable and its collision cube left in the environment, the cells that require the arm to get close to the original cell are reported as unreachable as well due to the planning issues. The effect propagates, making a portion of the environment unreachable.

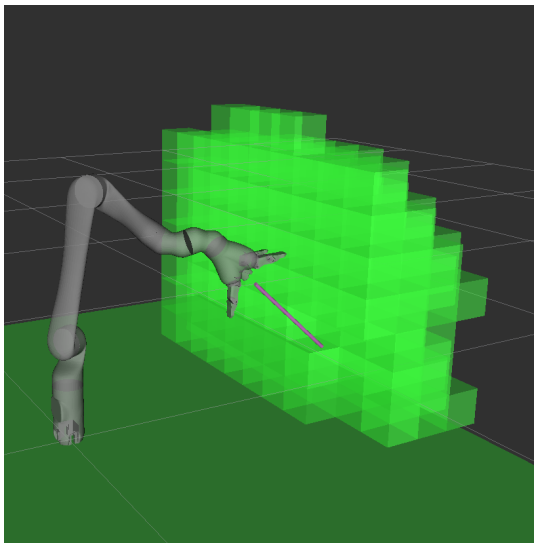


Figure 6.11: The simulated arm with the collision model of the sensing tool attached exploring the environment in simulation

Once the collision stick is attached to the robot model, the effects of obstacle proximity planning issues magnify, as the stick is close to the other collision objects all the time. The Jacobian control works as expected, but virtually all planning requests fail, marking large portions of the environment as unreachable.

The sole cause of the issues is the motion planner. Its sampling of states around the start and goal states fails utterly when it needs to pass through a restriction where there is very low probability of sampling a state that is both valid and then leads to the other state. The high efficiency of the planners in obstacle-free environment is given by the path simplification step, where the trajectories from start to end state, usually containing hundreds of states, are often simplified to direct trajectory from the start to the end state.

We also tested several different planners, to no improvement. If the target pose is in restricted space, the unidirectional planners need to be extremely lucky to sample a state from which the final state is reachable. Bidirectional planners fail to sample any states reachable from the end state directly. Even RRT planners that boast of good configuration state exploration cannot sample any such states.

The issues could in the future be resolved by using another motion planner,

perhaps one specially designed to work in this case that could exploit the structure in the problem, like the knowledge that we are dealing with a grid of cubes.

6.3.4 Conclusions

We tested the arm control in simulated environment. The motion control works as expected in empty space; when the arm gets close to colliding with a collision object, the planning capabilities are drastically reduced. Planning arm motion in the exploration algorithm is usable when stick collision checking is omitted. In that case, the arm managed to explore the environment with misclassifying only a few cells as unreachable.

Further work is necessary to make the exporation more robust, mainly in the field of motion planning. We need to plan motions in tightly constrained spaces, but we have additional knowledge about the space.

6.4 Real arm experiments

Experiments in the simulator are safe and very valuable during development, where we can focus on testing the code rather than checking if the arm is going to be safe. Nonetheless, real world experiments are absolutely necessary, as the system is in the end meant to run on a real rescue robot.

6.4.1 Moving the arm

We aimed to confirm the results of movement simulation on the real hardware. Motion planning exhibited the same behavior as in the simulation. It worked reliably, but in some seemingly random cases, a very complicated plan was generated. These complicated motion plans threaten, among others, the cable linking the sensing tool to the host computer.

When we repeated the Jacobian control test we used with the simulated arm, we discovered that the real arm deviates from the intended trajectory significantly. After following the same trajectory, the arm deviated 6 cm in y and 4 cm in z direction. We assume the error was given by neglecting the arm dynamics, assuming the given joint velocity can be reached immediately by the arm.

This high error gave us the incentive to implement the regulator described in 5.3.1 that would correct the errors. The results were excellent, with maximum error along the path 2 cm and final error 4 mm. The errors are depicted in

Figure 6.12. We then re-ran the simulated experiment with correction to complete the results.

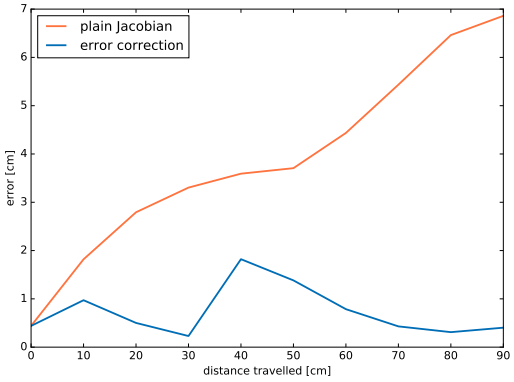


Figure 6.12: Position error during real arm test drive

Another issue we noticed was that the movement was in no case as smooth as in the simulator. The arm came to a complete stop every time a decision was made on where to continue. The movement became nearly as jerky as the planned movement. We account this error to the arm dynamics as well, as getting the arm to move seems to take as much time as planning the next step. This could in the future be resolved by splitting the driving and planning into two independent computations, where the next direction would be known even before the arm arrives at the target position. We did not implement the solution due to its implementation complexity.

Overall, we deem the Jacobian control method with error correction fit to be used to drive the robot during exploration, as long as errors caused by singularities are detected and corrected.

With the real arm, we also tested the manufacturer-provided Cartesian speed control. We do not include any quantitative data on the experiment, because the driver failed to fixate the end effector orientation during movement completely, even when zero angular twist was specified. The arm turned in the direction in which it moved. This may be caused by the arm's origin in assistive robotics, where this is the user-expected behavior, similar to the way a human arm moves. It also limits wrist rotation, which seems to be a common source of complications (singularities) when driving the arm, thus making the control less error-prone. Nonetheless, the need to control angular speed such that the orientation does not change renders the feature unusable for our application.

6.4.2 Stopping the arm

Another of concerns about whether things will work in the real world was about stopping the arm. Will the arm stop in time? We let the arm touch

obstacles that can be moved away if the system failed, and proceeded to tests on real obstacles upon gaining the confidence that the system worked. In Figure 6.13, the arm can be seen trying to touch a table.

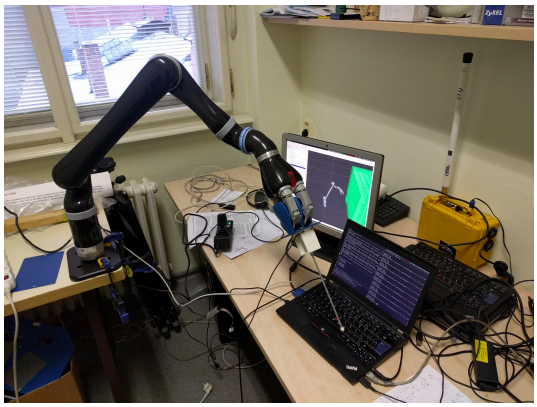


Figure 6.13: The arm touching a table

The system latency proved to be low enough to stop the arm in time. After detecting contact (force reading exceeding the threshold), the arm traveled at most another 0.5 cm with both Jacobian control and planned motion control, corresponding to a delay of 50 ms with the Jacobian controller. The delay is caused both by latencies in the system and the arm dynamics, where stopping the moving arm takes some time.

The measurement tool has some slack that can absorb the extra movement, and, more importantly, the force sensor is based on deformation, magnified by the stick length, so some extra traveled distance does not pose a problem. The only case in which the extra travel does matter is in case of direct axial contact of the stick with a perpendicular surface. Then, the stick does not have any way to bend and could reach the mechanical hard limit on the tool, perhaps even damage it. When using the tool in this mode, we recommend slowing the arm down so that the latency results in movement of only a few millimeters, tolerable for the tool. Better still, we recommend not to use the tool in this mode, which can be avoided by tilting the tool relative to the direction of motion, thus giving it chance to deform before breaking.

6.4.3 Exploring the environment

As in the simulated tests, we let the arm explore free space first. Then, we prepared a simple test environment for the arm to explore. Several test runs had to be terminated because of the cable leading to the sensing tool wrapping around the arm links too tightly. This could in the future be resolved by using the data line built into the arm joints, passing all the way through the arm.

The arm exploring the free space can be seen in Figure 6.14. The exploration

showed the same artifacts as in the simulation: when collision checking for the stick was disabled, only five cells in the workspace were misclassified as unreachable (for reasons stated above). The stick would however have traveled through unexplored space many times, leaving room for accidents.

When the stick was included in the checks, the motion planner failed almost constantly, succeeding only in edge cells of the workspace. This made the algorithm unusable.

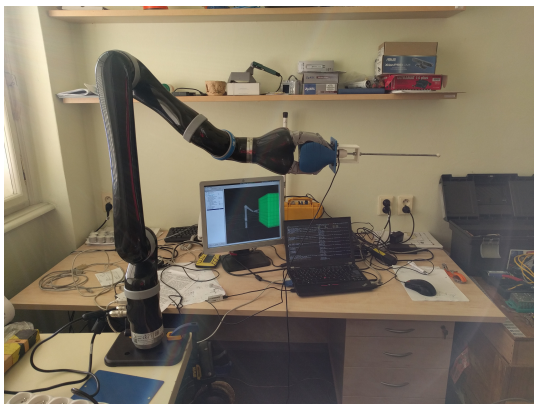


Figure 6.14: The real arm exploring empty environment

The testing environment for the test run with obstacles is presented in Figure 6.15. It was chosen to be simple, without any tight spaces the arm could get trapped in, and mobile, to allow us to stop the experiment if anything bad happened as we were running the experiment with stick collision checks disabled as well.

Figure 6.15: The real arm exploring a test environment

During the test run with stick collisions disabled, the exploration went as expected, including the first collisions with the obstacle from the left side as viewed in the picture. Then, after one contact with the obstacle, the arm successfully returned to non-contact position and the exploration algorithm planned a movement to the following cell. A motion plan was successfully found, but it was one of the plans where the arm moves rather wildly before

reaching the target. The stick attempted to pass through the obstacle again, stopping the movement and ultimately the whole experiment, as the algorithm detected collision outside the target area.

When we enabled the collision checks for the stick, the arm explored the space safely, never crashing the stick into the obstacle. The same falsely unreachable cells were however reported, and while some space has been successfully marked as empty and some parts of the obstacle were detected, overall, most of the map constituted of unreachable cells.

6.4.4 Conclusions

The experiments conducted on the real robot verified our ability to steer the arm in empty space. The dynamics the physical arm brings into the equation did not alter the behavior of the algorithm. Around collision objects, the motion planning was not able to compute a collision-free plan, which resulted in reporting unreachable cells just as in the simulated environment. All the suggestions posed in Section 6.3.4 apply here: implementing a motion planner more suited for this task would improve the robustness and stability of the whole system.

Chapter 7

Conclusions

The tasks of exploring space with a robotic arm proved to be very complex. From the hardware and the low-level concepts of serial manipulator, through robotic software frameworks with not-so-well functioning libraries, to covering unknown spaces, all levels are critical to complete the task.

7.1 Results of our work

We designed and built a tool that extends a force sensor sensing area further away from the arm, while maintaining its sensitivity. The tool is made to be held in the arm's gripper, and can thus be attached to the arm without any tools, even by the robot itself. We also modified the device driver to provide the data we needed. The sensor data is calibrated to the scale in Newtons.

The JACO arm was previously used in the TRADR project only in conjunction with motion planning. We developed a control module that can drive the arm along straight trajectories with reasonable precision of about 2 cm. This mode of control allowed us to speed the whole exploration process up by by-passing the previously necessary, time consuming and computationally expensive motion planning. The controller cannot cope with everything and motion planning is still invaluable for longer motions and motions around arm configurations where the movement cannot be controlled analytically, and over the course of our work, many issues with the planners were identified. Some have been solved, other remain.

We created and tested a Coverage Path Planning algorithm for unknown environments, the compact space heuristic algorithm, that guides the arm exploration. Although the algorithm is heuristic in nature, and as such does not give any optimality guarantees, the exploration trajectories it generates are both short and robot friendly. The compactness of explored space maximizes the robot's operation space, simplifying the planning problems. It is also

well-suited to be used in conjunction with grid maps and the simple linear motion control.

All the previously mentioned components were stitched together to create a prototype of the system that could be put to the test. The only thing needed to make the exploration usable by other parts of the system is to wrap the functionality into an API .

The performance of the complete solution was verified in both simulation and real application. The algorithm showed potential to explore the space completely and efficiently, but to achieve that, it lacks an adequate motion planner. The current planners are unable to find motions in close proximity of obstacles, which is crucial for our application. In Section 7.2 below, we propose an approach to take in designing the planner.

As of now, the system can be used in the TRADR system only in limited way, as we cannot rely on motion planning. Once the planning issue is resolved, the rest of the system is ready to be incorporated into the solution, thoroughly tested and used.

7.2 Further work proposals

The further work needed to build a complete, robust solution is mentioned several times. The main issue concerns motion planning, where a new approach is necessary. We propose to base the motion planning algorithm on the fact that we know that the robot will be reaching only to a finite number of locations in the grid, which limits the configuration space considerably, and that the main obstacles are only the collision cubes representing the unexplored cells. This knowledge could help design a simpler algorithm that would not rely on random sampling and achieve a higher success rate.

On the lower levels of the system, the `FollowJointTrajectory` server needs an update that would improve the path following precision. The finalizing movement of the arm is sudden and fast, and could cause damage to the arm. A different open-source implementation of the server is available as a contribution to Kinova ROS package, but has not been incorporated by the manufacturer as of yet. The question remains if the other solution implements the driver better.

The framework also still has issues with the joint limits on the arm. For MoveIt to support continuous joint rotation, deep changes to the library would almost surely be necessary. The robot would however benefit from it greatly, increasing the quality of the generated motions.

Appendix A

Bibliography

[1] Tradr project. http://www.tradr-project.eu/. Accessed: 2016-01-08.

[2] Nifti project. http://www.nifti.eu/. Accessed: 2016-01-08.

[3] Vojtěch Šalanský. Contact terrain exploration for mobile robot. Master's thesis, FEE CTU, 2015.

[4] Martin Pecka, Karel Zimmermann, Michal Reinstein, and Tomas Svoboda. Controlling robot morphology from incomplete measurements. *IEEE Transactions on Industrial Electronics*, 2016.

[5] Morgan Quigley, Ken Conley, Brian P. Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y. Ng. Ros: an open-source robot operating system. In *ICRA Workshop on Open Source Software*, 2009.

[6] Kinova robotics website. http://www.kinovarobotics.com/service-robotics/products/robot-arms/. Accessed: 2016-12-14.

[7] Lorenzo Sciavicco and Bruno Siciliano. *Modeling and control of robot manipulators*, volume 8. McGraw-Hill New York, 1996.

[8] Kinova jaco technical specification. http://kinovarobotics.com/wp-content/uploads/2015/02/JACO%C2%B2_6DOF_Technical-specifications-v1.0.pdf. Accessed: 2017-01-02.

[9] KINOVA ROS driver. https://github.com/Kinovarobotics/kinova-ros. Accessed: 2016-12-14.

[10] KINOVA ROS api. https://github.com/Kinovarobotics/kinova-ros/blob/master/README.md. Accessed: 2016-12-14.

[11] Moveit! documentation. http://moveit.ros.org/. Accessed: 2016-12-15.

[12] Moveit! source code and its documentation. http://docs.ros.org/indigo/api/moveit_ros_planning_interface/html/namespacemoveit_1_1planning__interface.html. Accessed: 2016-12-15.

[13] Kinova jaco in tradr project. https://redmine.ciirc.cvut.cz/projects/tradr/wiki/Kinova-jaco-arm, internal website. Accessed: 2017-01-07.

[14] Patrick Beeson and Barrett Ames. Trac-ik: An open-source library for improved solving of generic inverse kinematics. In *Humanoid Robots (Humanoids), 2015 IEEE-RAS 15th International Conference on*, pages 928–935. IEEE, 2015.

[15] Steven M LaValle. Rapidly-exploring random trees: A new tool for path planning. 1998.

[16] Ompl, open motion planning library. http://ompl.kavrakilab.org/. Accessed: 2016-01-06.

[17] Optical force sensors – introduction to the technology (whitepaper). http://optoforce.com/wp-content/uploads/2015/12/OptoForce_WhitePaper_Optical_Force_Sensors.pdf. Accessed: 2016-12-17.

[18] Shadow robot company optoforce driver. https://github.com/shadow-robot/optoforce. Accessed: 2016-12-15.

[19] Rudolph Triebel, Patrick Pfaff, and Wolfram Burgard. Multi-level surface maps for outdoor terrain mapping and loop closing. In *2006 IEEE/RSJ international conference on intelligent robots and systems*, pages 2276–2282. IEEE, 2006.

[20] Kai M Wurm, Armin Hornung, Maren Bennewitz, Cyrill Stachniss, and Wolfram Burgard. Octomap: A probabilistic, flexible, and compact 3d map representation for robotic systems. In *Proc. of the ICRA 2010 workshop on best practice in 3D perception and modeling for mobile manipulation*, volume 2, 2010.

[21] Georgios Papadopoulos, Hanna Kurniawati, and Nicholas M Patrikalakis. Asymptotically optimal inspection planning using systems with differential constraints. In *Robotics and Automation (ICRA), 2013 IEEE International Conference on*, pages 4126–4133. IEEE, 2013.

[22] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[23] Enric Galceran and Marc Carreras. A survey on coverage path planning for robotics. *Robotics and Autonomous Systems*, 61(12):1258–1276, 2013.

[24] Esther M Arkin, Sándor P Fekete, and Joseph SB Mitchell. Approximation algorithms for lawn mowing and milling. *Computational Geometry*, 17(1):25–50, 2000.

[25] Fumio Yasutomi, M Yamada, and K Tsukamoto. Cleaning robot control. In *Robotics and Automation, 1988. Proceedings., 1988 IEEE International Conference on*, pages 1839–1841. IEEE, 1988.

[26] Jordi Palacín, Tomás Palleja, Ignasi Valgañón, Ramón Pernia, and Joan Roca. Measuring coverage performances of a floor cleaning mobile robot using a vision system. In *Proceedings of the 2005 IEEE International Conference on Robotics and Automation*, pages 4236–4241. IEEE, 2005.

[27] Vikas Shivashankar, Rajiv Jain, Ugur Kuter, and Dana S Nau. Real-time planning for covering an initially-unknown spatial environment. In *FLAIRS Conference*, 2011.

[28] Alexander Zelinsky, Ray A Jarvis, JC Byrne, and Shin'ichi Yuta. Planning paths of complete coverage of an unstructured environment by a mobile robot. In *Proceedings of international conference on advanced robotics*, volume 13, pages 533–538, 1993.

[29] Raphael Mannadiar and Ioannis Rekleitis. Optimal coverage of a known arbitrary environment. In *Robotics and Automation (ICRA), 2010 IEEE International Conference on*, pages 5525–5530. IEEE, 2010.

[30] Zack J Butler, Alfred A Rizzi, and Ralph L Hollis. Contact sensor-based coverage of rectilinear environments. In *Intelligent Control/Intelligent Systems and Semiotics, 1999. Proceedings of the 1999 IEEE International Symposium on*, pages 266–271. IEEE, 1999.

[31] Chaomin Luo and Simon X Yang. A bioinspired neural network for real-time concurrent map building and complete coverage robot navigation in unknown environments. *IEEE Transactions on Neural Networks*, 19(7):1279–1298, 2008.

[32] Stephen Grossberg. Nonlinear neural networks: Principles, mechanisms, and architectures. *Neural networks*, 1(1):17–61, 1988.

[33] Ira Pohl. A method for finding hamilton paths and knight's tours. *Communications of the ACM*, 10(7):446–449, 1967.

[34] Susan Hert, Sanjay Tiwari, and Vladimir Lumelsky. *A Terrain-Covering Algorithm for an AUV*, pages 17–45. Springer US, Boston, MA, 1996.

[35] Kinova jaco technical specification. `http://kinovarobotics.com/wp-content/uploads/2015/02/K-Series_Technical-specifications-v1.0.21.pdf`. Accessed: 2017-01-07.

Appendix B

CD contents

The CD contains all the source codes we created in directory `src`.

A digital copy of this thesis is in the root of the CD as `buriama8_dp.pdf`