CENTER FOR
MACHINE PERCEPTION

CZECH TECHNICAL
UNIVERSITY IN PRAGUE

TECHNICAL REPORT

Configuration space of the NIFTi Robot

Tomáš Nouza

nouzato1@fel.cvut.cz

September 30, 2013

Supervisor: Ing. Michal Reinštein, Ph.D.

Center for Machine Perception, Department of Cybernetics
Faculty of Electrical Engineering, Czech Technical University
Technická 2, 166 27 Prague 6, Czech Republic
fax +420 2 2435 7385, phone +420 2 2435 7637, www: http://cmp.felk.cvut.cz

Configuration space of the NIFTi Robot

Tomáš Nouza

September 30, 2013

Abstract

This document is a report from the summer internship at CMP at CTU FEL from 24 June to 19 July 2013 (4 weeks). It describes the usage of the ROS (Robot Opearating System) [1] package arm_navigation [2] for the detection of the free configuration space of the NIFTi [3] robot.

Contents

1 Introduction **3**

2 Arm navigation stack **4**

2.1 Configuration 4

2.2 Environmental server 4

2.3 Moving with the virtual robot 6

3 Flipper collision detection **7**

3.1 Algorithm used 7

4 Evaluation **8**

1 Introduction

The NIFTi robot (on the Fig. 1) moves using two main belts and four smaller belts which are used for easier obstacle traverse. These smaller belts (called flippers) are mounted on the each end of the main belt and can be turned up and down but they do not have enough power to raise the robot to stay only on them. Currently the angle of the each flipper is set manually by the robot operator. Automation of setting the flippers angle according to the terrain is desirable because it reduces the robot operator work load during driving of the robot and he can concentrate more on the scene around the robot.

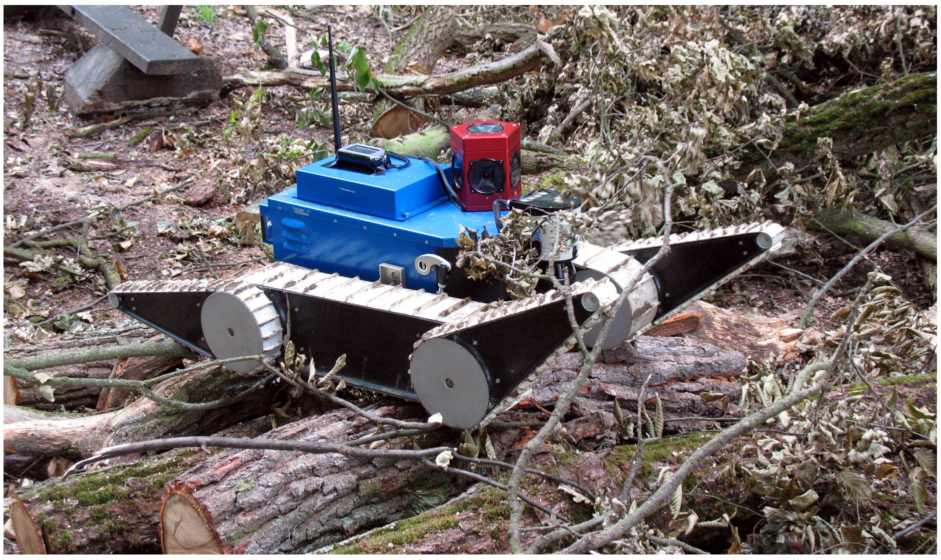

Figure 1: The NIFTi robot

Before the robot is allowed to autonomously set any flipper to the desired position, there must be confidence that this movement will not harm the robot. There are two cases which must be prevented:

1. overload of the flipper motor which happens e.g. when the main track is not in the contact with the floor
2. loss of the stability which leads to the robot turn over

This work aims to help with the implementation of the so called Adaptive traversability in the meaning of predicting the impact of the robot body movements. Firstly there was a research of the currently available working ROS implementations which can be used, secondly there was implemented simple ROS node for detecting the free manipulation space of the robot flippers.

2 Arm navigation stack

The ROS stack `arm_navigation` [2] contains packages that permit collision-free arm navigation can and since flipper is 1DOF arm, it can be used. Unfortunately it is deprecated and in the ROS Groovy version is replaced with `MoveIt!` which is newly a core part of the ROS.

2.1 Configuration

To use the arm_navigation stack, there is some configuration to be done. To make the configuration process easier, there is a wizard which needs only an URDF file describing the robot. These wizard can be launched using:

```
roslaunch planning_environment planning_description_configura
tion_wizard.launch urdf_package:=<Your urdf package name> urdf_
path:=<relative path to the urdf file in your urdf package>
```

As an output from that wizard there will be created a new package containing all the configuration files needed. More informations about the wizard can be found in [4]. The important generated file later used in this work is `<robot_name>_planning_description.yaml`. More informations about this file and all others generated files can be found in [5].

2.2 Environmental server

All the movement planning is done using the environmental server. It is a part of the `planning_environment` and can be launched using:

```
roslaunch planning_environment environment_server.launch
```

In this work, environment server is used to detect collision of the robot with the collision map. The collision map is build from the poincloud2 messages using package `octomap_server` [6]. Usage of the `octomap_server` is more documented on the https://cw.felk.cvut.cz/wiki/misc/projects/nifti/sw/adaptive_traversability#occupancy_map

All the communication with the environment server is done using the service `/environment_server/set_planning_scene_diff` which can be called like in the next example:

```
#define SERVICE = "/environment_server/set_planning_scene_diff";
arm_navigation_msgs::GetPlanningScene::Request req;
arm_navigation_msgs::GetPlanningScene::Response res;
```

```
if(!ros::service::call(SERVICE, req, res)) {
   ROS_WARN("Can't get planning scene");
   return -1;
}
```

Once the service is called, the environment server can be used to detect the current state and collisions. For example this code will get the current state of the robot saving the rotation of each joint as a `std::map`

```
static planning_models::KinematicState* state;
static planning_environment::CollisionModels* collision_models;
state = collision_models->setPlanningScene(res.planning_scene);

std::map<std::string, double> state_vals;
state->getKinematicStateValues(state_vals);
```

Once the state is known, it can be easily modified. For example this code will virtually turn the front right flipper 90° up:

```
state_vals["front_right_flipper_j"] -= 0.785;
state->setKinematicState(state_vals);
```

To detect the collisions there is a function:

```
std::vector<arm_navigation_msgs::ContactInformation> contacts;
collision_models->getAllCollisionsForState(*state, contacts, 1);
```

The message for the `arm_navigation_msgs::ContactInformation` is defined as follows:

```
uint32 ROBOT_LINK=0
uint32 OBJECT=1
uint32 ATTACHED_BODY=2
std_msgsHeader header
uint32 seq
time stamp
string frame_id
geometry_msgsPoint position
float64 x
float64 y
float64 z
```

```
geometry_msgsVector3 normal
float64 x
float64 y
float64 z
float64 depth
string contact_body_1
string attached_body_1
uint32 body_type_1
string contact_body_2
string attached_body_2
uint32 body_type_2
```

Beside the information about the collision position there is also information which part of the robot is in the collision and if it is a contact with the environment or with another part of the robot (self-collision).

At the end, the cleaning can be done using the function:

```
collision_models->revertPlanningScene(state);
```

2.3 Moving with the virtual robot

For moving with the mobile robot there is a virtual joint which connect the robot body with the global coordination frame[1]. Turning the wheel of the robot will not move it but only turn the wheel. Virtual joints names are in the table 1. The rotation is defined by the quaternion notation.

Table 1: Names of the virtual joints

translation	rotation
`floating_trans_x`	`floating_rot_x`
`floating_trans_y`	`floating_rot_y`
`floating_trans_z`	`floating_rot_z`
	`floating_rot_w`

The virtual NIFTi robot can also move with the following joints even not all of them has motor and can be controlled in the real situation:

[1]http://answers.ros.org/question/11534/arm-navigation-with-a-virtual-robot/ shows another approach but during testing it in this work the collision detection was not properly working

```
laser_j
left_track_j
front_left_flipper_j
rear_left_flipper_j
right_track_j
front_right_flipper_j
rear_right_flipper_j
```

3 Flipper collision detection

Autonomous adaptive traversability is a hard task with lots of subtasks. One on them is a detection of the free configuration space. This task was implemented as a ROS service called `flipper_configuration_space`. This service receives 8 angles in radians as a float32 number which represents an angle requirements for all four flippers for angle up and down from the current flipper position. Output is also 8 angles which represents how much can be each flipper turned before it hits some obstacle (e.g. floor). Maximum returned angle is equal to the input one which means that the flipper can be turned along the collision free trajectory. If the 0 is returned, it means that the flipper is currently in or very close to a collision and can not be turned in this direction. In real situation it can raise the robot over the obstacle or simply chop down the grass but this decision is left for the higher cognition functions of the robot.

3.1 Algorithm used

The detection process is simple using the environment server (see the section 2.2). The collision map is build from the pointcloud2 messages using the octomap server as on the Fig. 2. The current state of the robot is obtained from the environment server and then all the flipper are tested separately for the collisions and the angles are iteratively increased/decreased about 0.2 rad[2] until they reach the required angle. If the collision is reported, process for the current flipper and direction is terminated and the last collision free angle is reported.

[2]the angle between leading edges of the flipper is 0.4 rad

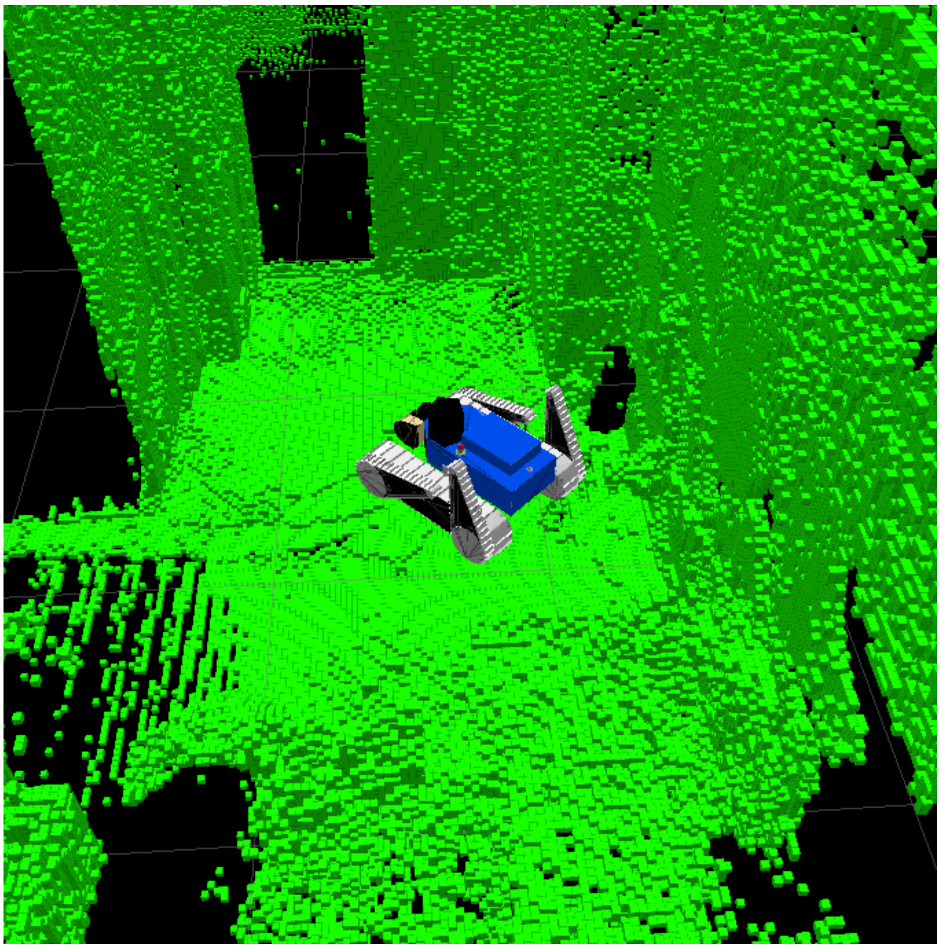

Figure 2: Collision map build by octomap server

4 Evaluation

Despite all the efforts the service was not successfully tested because the laser scanner detected the robot flippers as an obstacle. Therefore the front flippers were in collision with itself and the service allowed their movement about 0 rad. The solution of this problem is in the more accurate laser data filtration to eliminate the robot body from the pointcloud2 message which is used for building the occupancy grid. The visualization of the flipper in the collision with itself is on the Fig. 3.

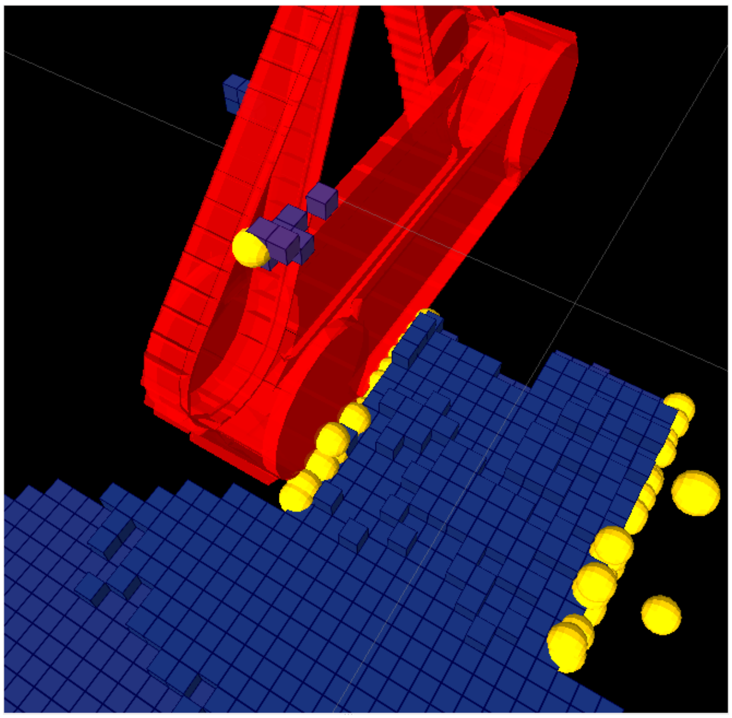

Figure 3: Right track (red) with detected collision (yellow dots) and the fake collision on the front right flipper. Other yellow dots are collisions with left track which is not visualized.

References

[1] "Robot Operating System." http://www.ros.org/wiki/ROS. Accessed: 26/04/2013.

[2] "arm_navigation stack." http://wiki.ros.org/arm_navigation. Accessed: 24/09/2013.

[3] "Natural human-robot cooperation in dynamic environments." http://www.nifti.eu. Accessed: 26/04/2013.

[4] "Planning description configuration wizard." http://wiki.ros.org/arm_navigation/Tutorials/tools/Planning%20Description%20Configuration%20Wizard. Accessed: 24/09/2013.

[5] "Understanding and adjusting the auto-generated arm navigation application." http://wiki.ros.org/arm_navigation/Tutorials/tools/Understanding%20and%20adjusting%20the%20autogenerated%20arm_navigation%20application. Accessed: 24/09/2013.

[6] "octomap_server." http://wiki.ros.org/octomap_server. Accessed: 24/09/2013.