University of Bonn

Lab Course Mobile Robots (MA-INF 4310)

INTRUDY: The Intruder Detection and Following System

Programmer's Manual

Authors:
Pablo Aponte
Ilya Manyugin

Supervisor:
Dr. Nils Goerke

Bonn, 2013

Contents

Preface

This document describes INTRUDY, The Intruder Detection and Following System. The emphasis of this document is made on the software part of INTRUDY, with some introduction of the underlying theoretical concepts and the description of the hardware platform.

1 General Description

1.1 Hardware Platform

INTRUDY is run upon the ROOMRIDER, the teaching and research mobile robot platform developed by the Department of Autonomous Intelligent Systems (Informatik VI) at the University of Bonn [6, III.A. Roomrider].
The platform itself is based on the iRobot 530 Roomba Vacuuming Robot. Additionally it features a SICK S300 laser scanner and a notebook, enabling the control of the robot's movements via the serial interface.
The laser scanner supports a scanning azimuth of 270° covering the range from -135° to +135°, where 0° is the frontal point of the platform. The laser scanner data is available with a resolution of 0.5° as an array of 540 values. The range measured by the scanner to the obstacle can vary from 5 cm to 30 m.

1.2 Software Platform

In the software part the Robot Operating System (ROS) was used as the main software framework for the software development controlling the robot[1]. ROS provides many services to ease the development and testing of the software written for different kinds of robots, among these services are hardware abstraction and transport of the messages between processes ("nodes" in ROS terminology). The Robot Operating System is a flexible framework for writing robot software [1]. It provides a set of tools and libraries for programming complex robot behavior, and allows for an operating system. ROS is distributed under a BSD free software licence.

For the purpose of this work, the Long-Term Support release of Ubuntu Linux 10.04.2 Lycid Lynx was used as the main operating system running on the notebook, and the ROS Elecrtic distribution was ran in that environment.

ROS supports packaging as a mean of distribution of its components, and the two main ROS packages used in our work are `roomrider_driver` and `sicks300`. The roomrider driver package is a driver used to control the RoomRider platform and to get the sensory information via the serial interface of the Roomba [2]. The sicks300 package is a

driver that reads the scan data of a Sick S300 Professional Laser Scanner via its serial interface [4]. Both packages were developed within the Autonomous Intelligent Systems Department at the University of Bonn.

ROS provides a choice of three programming language bindings (so-called 'client libraries' in ROS terminology): `roscpp` for C++ development, `rosjava` for Java and `rospy` for Python bindings. In this work we chose to use Python programming language and the `rospy` client API for the development of INTRUDY, because of the flexibility and the ease of prototyping this language provides. Also, Python has a renown and extensive scientific library `scipy`, which provides a large set of high-level mathematical functions and allows for faster computations, as parts of it are written in plain C. This allows us to reduce the gap between the choice of the Python and the C++ client libraries.

2 The Intrudy ROS Package

2.1 Running Intrudy

Since the code was written in Python there is no need to compile anything. To run it, the roslaunch command is used:

```
roslaunch intrudy intrudy.launch
```

It is important to note that `intrudy.py` in the `scripts` folder must have file system permissions to be read and executed.

The node description of the INTRUDY is as following:

```
<node pkg="intrudy" type="intrudy.py" output="screen" name="intrudy" ></node>
```

2.2 Package Structure

As briefly described in section 1.2, INTRUDY is written in Python programming language using the ROS Python client library; this architectural desicion also affects the way the ROS package is organized.

INTRUDY is distributed as a ROS package, a directory tree of a special structure. As the official documentation [3] states, a ROS package can *contain ROS nodes, a ROS-independent library, a dataset, configuration files, a third-party piece of software, or anything else that logically constitutes a useful module.*

The INTRUDY package directory tree is shown in Figure 1.

```
intrudy
├── CMakeLists.txt ..... CMake build file
├── mainpage.dox ..... Doxygen mainpage documentation
├── Makefile
├── manifest.xml ..... Package Manifest
├── launch ..... Roslaunch launch scenarios directory
│   ├── intrudy.launch ..... Main launch file
│   └── intrudy_sim.launch ..... Simulation launch file
└── scripts ..... Executable scripts (Python code) directory
    ├── controller.py ..... Intrudy controller
    ├── config.py ..... Intrudy configuration file
    ├── icp.py ..... The ICP module
    ├── intrudy.py ..... Intrudy main module
    ├── record.py ..... The Logging module
    └── tracking.py ..... The Tracking module
```

Figure 1: INTRUDY package directory tree

2.3 Robot Steering Module

The movements of the robot are driven by the data that is returned by the target tracking module described in the section 2.5. In order to aid more smooth movements of the robot and to impelemnt the target re-acquisition, a Finite State Machine (FSM) was implemented within the program.

This FSM has four internal states, as following:

1. *Surveillance*
   In this state INTRUDY is stationary (i.e., the command given on each iteration is $\{0, 0\}$, that is, both the linear and angular speeds are set to zero), and the movement detection threshold in the target tracking module is set to a lower value, in order to aid early target detection.

2. *Following*
   In this state INTRUDY is moving, following a target, and the effective command is dependent on both the angle of the target relative to INTRUDY's front and the distance between the INTRUDY and the target.

3. *Re-Acquisition*
   The FSM transits to this state from the Following state if the Tracking Module reports a target loss (i.e., the return value of the `track` method is `None`).
   In this state INTRUDY is still moving towards the place where the target was last detected, simultaneously trying to re-acquire the target. If INTRUDY fails to re-acquire target within 5 time steps (a configurable parameter of INTRUDY specified in `config.py`) since the target loss, it transits to the Surveillance state. If the target was successfully re-acquired, it transits to the Following state.

4. *Collision*
   A collision detection algorithm ensures that all the range values retrieved from the SICK S300 laser scanner are greater than 0.15 m. If this assertion doesn't hold, we say that we detected a collision with an obstacle, and INTRUDY transits into the Collision state. In this state INTRUDY runs at each time step the collision detection algorithm in order to check, whether the obstacle has changed its location. Therefore, if INTRUDY collides with a stable object, it cannot resolve the collision by itself, and the only way to make it operable again is for the user to move the robot away from the obstacle.

These four states, as well as the transition paths between them, are shown in Figure 2. In the figure, t is the target, either *None*, or (ϕ, r) meaning the angle and the distance to the target in polar coordinate system, respectively.

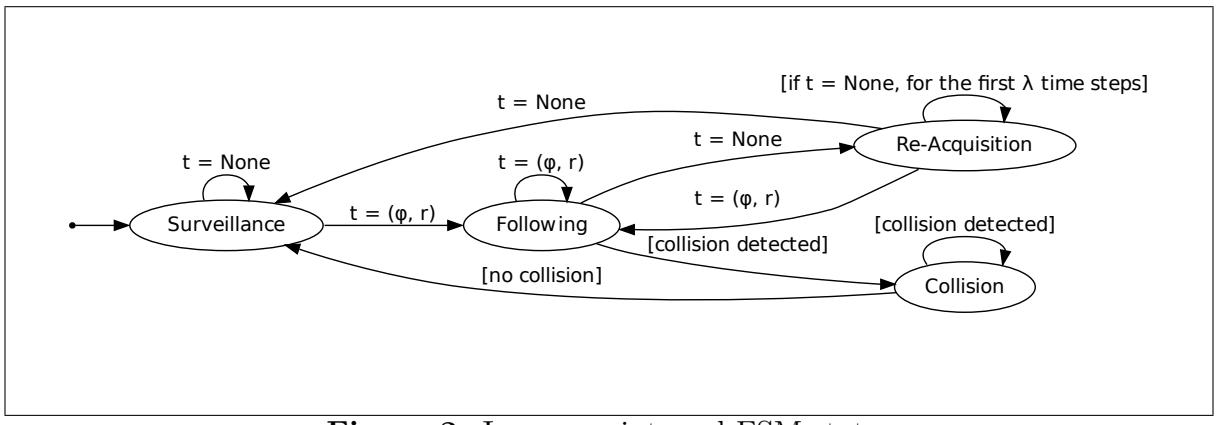

Figure 2: INTRUDY internal FSM states

2.4 Movement Detection and The Iterative Closest Point Algorithm

The Iterative Closest Point (ICP) algorithm is a point set registration algorithm[5]. In this work we employ ICP as a powerful algorithm for calculating the displacement of the objects between two raw scans made by the SICK S300 laser scanner.

The input of the ICP algorithm consists of the two sets of points (point clouds) from the raw scans, X and P, where X is the first scan, also called the *source*, and P is the second scan called the *target*. The key idea of this algorithm is to find a correspondence between these two sets by finding the transformation (rotation R and translation t) needed to transform the first set into the other, such that the sum of the squared error E, as shown in Formula 1, is minimized.

$$E(R,t) = \frac{1}{|P|}\sum_{i=1}^{|P|}(x_i - R \cdot p_i - t)^2 \tag{1}$$

That is, this iterative algorithm keeps the *target* cloud fixed, while it transforms the *source* cloud, calculating the squared error (1) and comparing it to an arbitrary threshold, e.g. 0, until converged.

The main steps of the algorithm are as following:

1. Find the correspondence between points: for each point in X find the closest point in P.
2. Find the correlation matrix between the new set and the corresponding points.
3. Do an Singular Value Decomposition (SVD) to obtain the matrices necessary for the formula of the rotation matrix.
4. Obtain the translation vector using the rotation matrix and the mean of the two sets.
5. Apply both the rotation and translation matrix.
6. Recalculate the mean squared error (Formula 1) and check if it is below the threshold, else go to step 1.

In the INTRUDY project we implement and widely use a technique called the ICP Optimization. The main idea of the ICP Optimization is to use the ICP algorithm and previous knowledge about the target (its position), in order to reduce the search space in the newly obtained scan.

Given two scans s_0 and s_1, we apply the ICP algorithm using both of them, changing s_0 in such a way that the static objects (such as walls, chairs, etc.) remain in the same position in both scans, while the displaced objects (e.g., a person walking) are the ones that change between scans. We get the transformation T as the output of the ICP algorithm, and we apply this transformation to the s_0 scan and the previously obtained target position p_t. Applying the transformation T to the previous target position t_p, we get the estimation of the target in the next scan $p'_t = T(p_t)$. This estimation is saved for later use in a class variable.

2.5 The Tracking Module

The Tracking Module is arguably the most important part of INTRUDY. This module allows the robot to track the target's position and follow it. The overall principle of functioning of the Tracking Module can be descbribed as follows.

1. Given two range scans, we truncate the values to the maximum range R_{max} (configurable parameter of the system), that is, for each distance d_i in each of the scans, if $d_i > R_{max}$, then $d_i \leftarrow R_{max}$. Additionaly the scans are converted from a robot-centric polar coordinate system into Cartesian coordinate system. If INTRUDY is not in *surveillance mode* (see section 2.3 for the list of INTRUDY modes), the ICP optimization of the first scan is done and, at the same time, the previously detected target, if any, is moved accordingly to this ICP optimization (see section **??**). This optimization is done in the first scan so that we can also recalculate the position of the previously obtained target.

2. After the optimization, the corresponding points are found in the two sets (scans). These are the points that are considered to have been a subject to transformation due to the target movement. We consider all the n points of the 'new' scan S_1 and compute the Euclidean distance E_d between each point from S_1 and all the points from the set S_0. If all the Euclidean distances are greater than a configurable parameter alpha, then the point $p_{S_1,i}$ is considered an 'outlier' (see Formula 2).

$$\forall p_{S_1,i} \in S_1, \forall p_{S_0,j} \in S_0, E_d(p_{S_0,j}, p_{S_1,i}) > \alpha \tag{2}$$

   This approach allows us to ignore the noise encountered in the data that is introduced by the resolution of the laser scanner, if the variance introduced by the noise is less than α.

3. After the outliers are found, it is necessary to find the difference between scans S_0 and S_1 in those points; thus we obtain the set of differences D. If each difference $d_i \in D$ is too small, it is considered to be noise and will be discarded. Then, the outliers are grouped together if they are situated in the neighboring range values in the scan (e.g., $\{p_{S_1,50}, p_{S_1,51}, \ldots, p_{S_1,70}\}$), and if they are of the same sign in the difference (either $d_i \geq 0$ or $d_i < 0$).

   Once the groups are formed, the most probable target position will be found. For this, the differences of the outliers should be considered. The predicted movement can be of three different types:

   - all positive differences, meaning the target moved towards the robot, i.e., $\Delta y < 0, \Delta x = 0$;
   - all negative differences, meaning the target moved away from the robot, i.e., $\Delta y > 0, \Delta x = 0$;
   - combination of positive and negative differences, meaning $\Delta x \neq 0$.

   The groups are thresholded by their size, the ones whose size is less than β are filtered out.

4. Once the most probable targets are found, we use a heuristic to determine which target is the best, and, therefore, the actual target the algorithm returns. The heuristic selects that target, which is the closest one to the previous target location, determined in the previous run. If there is no previous target location, the algorithm will select the target consistion of the greatest number of points.

   If the robot is in either the *following* or the *re-aquisition* state, it will not use the full range of its lasers, but instead only a 90° window centered in the previous target location.

   While in this states, the target position will be also updated (transformed) to account for the robots movements.

3 Source Code Documentation

Each class of the INTRUDY system is well documented, the style of the comments is compliant with the Epytext markup language and the full API reference, as well as the introspect of the modules can be generated with the Epydoc documentation generator. It supports many output formats, among them HTML, PDF and LaTeX.

An example Epydoc output for the INTRUDY code is shown below.

Functions

debug_info(*msg*)

Debug message function

enum(***enums*)

A function creating new type alike to the enum in C

Methods

__init__(*self*)

Constructor

Overrides: object.__init__

laser_callback(*self*, *data*)

Laser callback called by the subscriber

main_loop(*self*)

Main execution loop

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

4 Evaluation of Results

4.1 Tests Setup and Description

After several qualitative tests in the simulator and on the real robot, it was concluded that the system should be tested in nine different setups.

The first four tests where done in a controlled enviroment, where several plywood boards worked as the walls. Figure 3 shows the first test setup and the enviroment described before.

Figure 3: Test Setup Environment

For these four tests, the robot with the INTRUDY system was always in the same starting position, but the robot with the target system had its position changed in each test. The "target" robot had the same behaviour in all enviroments: move forward at the speed of 0.25 m/s until an obstacle is detected, then halt. The four initial positions of the "target" were:

1. The "target" was 1 meter away from the INTRUDY robot and both robots had the same orientation.
2. The "target" was 4 meters away from the other robot and the robots were facing each other.
3. The "target" was 1 meter to the left and 2 meter to the front of the INTRUDY robot. With respect to INTRUDY's coordinate system, the target robot was looking at 0°

while INTRUDY was looking at 90°.

4. Similar to the third test but the target robot was at the right side this time looking with an angle of 180° with respect to INTRUDY's coordinate system.

The last five tests had the same robot initial position as the first test, but they were done in the following enviroments:

5. The mobile robotics lab without any plywood simulating the walls, with many obstacles present, such as chairs and table legs.
6. The hallway in front the mobile robotics lab without any obstacles, but with doorways present.
7. The hallway T-shaped intersection near the previously mentioned lab.
8. The octagonal room near the lab, which had chairs, metal objects and glass doors as challenges.
9. The octagonal room, but with plywood covering the chairs and the glass doors.

4.2 Qualitative Results

In all the tests the INTRUDY robot managed to follow the "target", but in most of the cases it has stopped several times during the following process.

Four aspects of the test results were taken in account during the qualitative evaluation of the robot:

1. Did INTRUDY follow the target, i.e., did it constantly move towards the target?
2. Did INTRUDY find the target frequently, i.e., was INTRUDY able to reacquire the target while moving and stopped less times.
3. Did INTRUDY stop after the target stopped?
4. Did INTRUDY stop close to the final target's position?

This aspects were evaluated during the tests with four possible values:

- it does sometimes (✓),
- it does constantly (✓✓),
- it does it one or two times (✗),
- it does not do it at all (✗✗).

The obtained qualitative results are shown in Table 1.

Test	Aspect 1	Aspect 2	Aspect 3	Aspect 4
1	✓✓	✗	✓	✓✓
2	✓✓	✓	✓	✓✓
3	✓✓	✓✓	✗	✗
4	✓	✗	✗✗	✗✗
5	✓✓	✓	✗	✗✗
6	✓✓	✓	✓✓	✓
7	✓✓	✓	✓✓	✓
8	✓✓	✓✓	✓✓	✓✓
9	✓✓	✓	✗	✓

Table 1: Qualitative results of the nine tests

The test number 8, in the octagonal room, prooved to be the best one qualitative speaking even though this enviroment contained more obstacles and objects that creates more noise in the sensors.

4.3 Quantitative Results

Since the quantitive results are not enough to determine how good the system is, the scans and changes in the the system states were logged . This logs were parsed afterwards to obtain some measurements to compare each test and to see if the system actually is doing what it is suppose to do. The measurements done are the following:

- Time needed for initial detection.
- Difference between last time it was found and last time the target moved.
- Number of times the target was found in each of the states.
- Number of times the target was not found in each of the states.

Also, the accuracy should be one of the measurements done to this result, but due to time limitations it was not done. The results can be observed in Table 4.3 and in Figure 4.

Test	Initial Detection	Diff. of last movement and last time found
1	0.94 secs	5.26 secs
2	7.13 secs	4.5 secs
3	0.76 secs	6.59 secs
4	0.96 secs	4.30 secs
5	0.90 secs	-5.80 secs
6	1.05 secs	-0.97 secs
7	1.0 secs	-3.76 secs
8	1.07 secs	0.27 secs
9	0.98 secs	2.93 secs

Table 2: Quantitive results of the nine tests

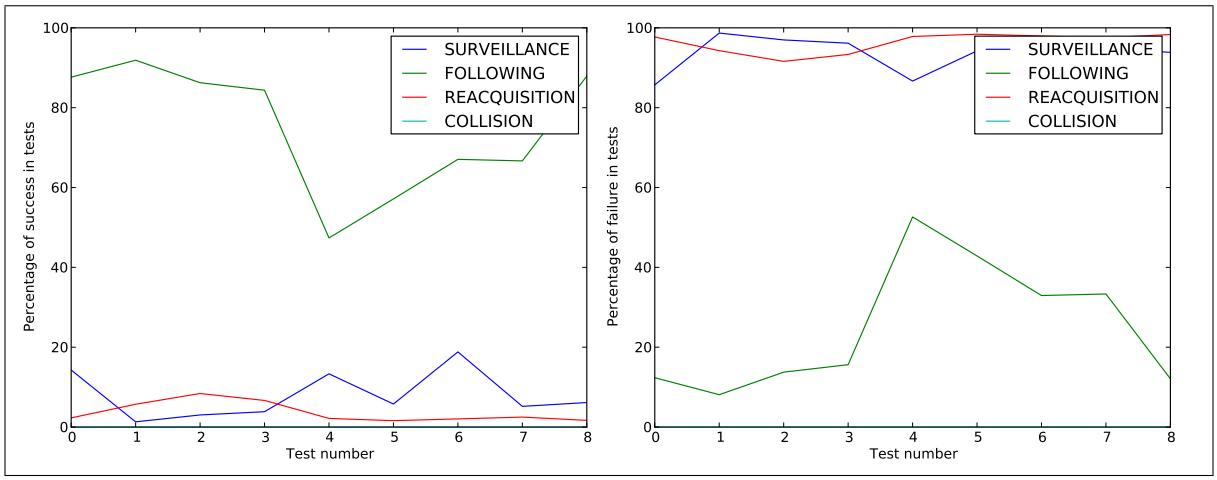

Figure 4: Percentage of the times INTRUDY found the target successfully (left) and unsuccessfully (right) for each state

5 Future development

Even though the current implementation of INTRUDY achieves the goals set for it, it is not completely an error-prone implementation. In this section we will propose a set of steps to be taken in future to improve the performance of INTRUDY with respect to the evaluation described in section 4.

At the current implementation status, the system has difficulties following the target continuously. Several parts of the algorithm could be changed in order to achieve a better performance. This report will mention three of these possible changes.

1. Better ICP implementation: One of the several variations of the ICP algorithm can be used to obtain better results when aligning the two scans. This will yield a more accurate target and will permit the user to increase the sensibility of the system.
2. Reduce the effect of the noise in the target detection: During the analysis of the results it was observed that several of the scans contained faulty measurements: the maximum range value.
3. Increase the search window in the re-acquisition mode for each time step: Increasing this window size will enable INTRUDY to find the target that moves faster than we expected, or that was not found in several time steps and that still continued moving during the re-acquisition phase.

Also, a better obstacle avoidance algorithm can be implemented to aid better overall performace of INTRUDY.

References

[1] General description of the Robot Operating System. `http://www.ros.org/about-ros`. Accessed: 2013-11-21.

[2] RoomRider driver package description in ROSWiki. `http://wiki.ros.org/roomrider_driver`. Accessed: 2013-11-21.

[3] ROS Packages in ROSWiki. `http://wiki.ros.org/Packages`. Accessed: 2013-11-21.

[4] SICKS300 driver package description in ROSWiki. `http://wiki.ros.org/sicks300`. Accessed: 2013-11-21.

[5] Paul J Besl and Neil D McKay. Method for registration of 3-d shapes. In *Robotics-DL tentative*, pages 586–606. International Society for Optics and Photonics, 1992.

[6] Nils Goerke and Sven Braun. Building semantic annotated maps by mobile robots. In *Proceedings of the Conference Towards Autonomous Robotic Systems, Londonderry, UK*, 2009.