
SPARC Documentation

Release 0

Stefania Ghita, Alexandru Gavril, Miruna Barbu, Andrei Nica, Miha

Aug 21, 2020

Table of contents:

1	Getting Started	1
1.1	Hardware Prerequisites	1
1.2	Software Prerequisites	1
1.3	Project	1
1.3.1	External Platform	2
2	Running and Configuration Pepper with ROS	5
2.1	Installing ROS	5
2.2	Pepper ROS Integration	6
2.2.1	Installing NaoQI	6
2.2.2	Installing the required ROS modules	6
2.3	Testing the Installation	6
3	ROS Modules List	9
3.1	Configuration	9
4	Vision configuration	13
4.1	Software Prerequisites	13
4.2	Testing the vision component	14
4.3	Testing the vision component	14
4.4	Fine-tuning the face recognition module	15
4.5	Generate new QR codes	15
5	Navigation Module	17
5.1	Creating a new map	17
5.2	Running the navigation module on a map	17
6	Speech Component	19
6.1	Speech recognition publisher	19
6.2	Speech recognition subscriber	20
6.2.1	Wit.ai configuration	20
6.2.2	Understanding new phrases	20

Welcome to the SPARC Project documentation.

1.1 Hardware Prerequisites

- Pepper Robot running Naoqi 2.5 or higher
- RaspberryPI 3
- RPLidar A1

1.2 Software Prerequisites

1. Ubuntu 14.04 operating system (or any other operating system which is compatible with ROS Indigo)
2. Install ROS [<http://wiki.ros.org/indigo/Installation/Ubuntu>]
3. Python 2.7
4. Pytorch installed and configured on the machine running CUDA 8.0+

1.3 Project

The project is organized as follows:

- The ***external_platform*** folder contains the python project used for external processing like running the Vision module and interacting with the ROS environment.
- The ***master_catkin_ws*** is the ros workspace used on the external machine as well and contains the Navigation module and all ROS related nodes.
- The ***rpi_catkin_ws*** is the ros workspace used on the Raspberry PI 3 and it is used to perform data acquisition.

1.3.1 External Platform

The python project is organized in a modular way with each module having a specific task.

Navigation

The module contains all the scripts interacting with the SLAM module. It contains the publishing nodes for the location of the detected objects, people and QR codes. It also interacts with the SLAM module to send the current robot position in the system.

Robot Interaction

The module contains all the scripts used to interact with the system from an external module. The module is used by the speech module to send command to the system. In a similar fashion, the graphical user interface makes use of this module to interact with the system.

Task Management

The module contains all the scripts used for the task management and planning. It is responsible with the management of the commands queue and ensures when and if a task is executed. All the commands received by the robot interactions module is forwarded to the task management module.

The planning is done using a Queue of Tasks. A task is defined by a node in a binary tree, having a success child and a fail child, both which are defined also as Tasks.

Vision

The module is responsible with all the components used for detection, recognition and segmentation of person, objects and qr codes.

TOC:

- Running and Configuration Pepper with ROS
 - Installing ROS
 - Pepper ROS Integration
 - * Installing NaoQI
 - * Installing the required ROS modules
 - Testing the Installation
- Vision Module
 - Testing the Vision module
 - Face recognition module
 - * Adding a new face
 - * Removing existing faces
- Navigation Module
 - Creating a new map
 - Running the navigation module on a map

- Speech Module
 - Wit.ai configuration
 - Understanding new phrases
- Integration of new command nodes

Running and Configuration Pepper with ROS

The Sparc system requires having a machine running ROS Indigo which will run the master node and a Raspberry PI 3 running ROS used for data acquisition from the RP Lidar A1 or similar Lidar devices.

Any similar compatible devices which send [Laserscan](#) data to the master node can be used. The configuration for the SLAM module will have to be fine tuned and adapted with the device.

2.1 Installing ROS

1. Setup your sources.list - Setup your computer to accept software from packages.ros.org. ROS Indigo ONLY supports Saucy (13.10) and Trusty (14.04) for debian packages.

```
sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_release -sc) main"␣  
↪> /etc/apt/sources.list.d/ros-latest.list'
```

2. Set up your keys

```
sudo apt-key adv --keyserver hkp://ha.pool.sks-keyservers.net --recv-key␣  
↪0xB01FA116  
sudo apt-get update
```

3. Install the full package of ROS `sudo apt-get install ros-indigo-desktop-full`

If the ros-indigo cannot be found in the sources, make sure you have Ubuntu 14 or other compatible operating system! Check the ROS wrapper compatibility before continuing installing the system!

1. Initialise ROS DEP `sudo rosdep initrosdep update`
2. Setup the environment: `echo "source /opt/ros/indigo/setup.bash" >> ~/.bashrcsource ~/.bashrc`
3. Install python-ros `sudo apt-get install python-rosinstall`

2.2 Pepper ROS Integration

Pepper requires the Naoqi framework installed and running in order to be able to interact with Pepper. Choregraphe is also recommended for debugging. After installing the Naoqi framework the Sparc system requires having Pepper integrated into the ROS framework, therefore although Pepper cannot directly run ROS, a ROS wrapper can be installed in order to perform data acquisition from the robot and send command using ROS `common_msgs`.

2.2.1 Installing Naoqi

1. Download the C++ and Python SDK (2.5+) <https://community.aldebaran.com/>
2. Extract the python SDK (pynaoqi-python2.7-x.x.x-linux64.tar) and the C++ SDK (naoqi-sdk-x.x.x-linux64.tar) in ~/naoqi ~/naoqi/naoqi-sdk-x.x.x-linux64/naoqi

3. Export the SDK path variable:

```
export PYTHONPATH=~/naoqi/pynaoqi-python2.7-x.x.x-linux64:$PYTHONPATH
echo 'export PYTHONPATH=~/naoqi/pynaoqi-python2.7-x.x.x-linux64:$PYTHONPATH' >>
↪ ~/.bashrc
```

4. Run python and check that you can import the library

```
from naoqi import ALProxy
```

2.2.2 Installing the required ROS modules

- The following command will install all the necessary ROS Modules that are used as wrappers over the Nao-QI framework and required to interact with Pepper.

```
sudo apt-get install ros-indigo-driver-base ros-indigo-move-base-msgs ros-indigo-
↪octomap ros-indigo-octomap-msgs ros-indigo-humanoid-msgs ros-indigo-humanoid-
↪nav-msgs ros-indigo-camera-info-manager ros-indigo-camera-info-manager-py
```

- Install the Pepper specific modules

```
sudo apt-get install ros-indigo-pepper-.*
```

2.3 Testing the Installation

1. Run `roscore` on the master node
2. Run `roslaunch pepper_bringup pepper_full.launch nao_ip:=<yourRobotIP> roscore_ip:=<roscore_ip> [network_interface:=<eth0|wlan0|vpn0>]`
 - `roscore_ip` is the ip of the master node
 - `nao_ip` is the IP of Pepper (can be found by pressing the button on Pepper's chest)
3. Run `roslaunch rviz rviz`

The robot should be shown inside the rviz application. If this fails to happen make sure that the pepper meshes are installed and accessible. To install the meshes run `sudo apt-get install ros-indigo-pepper-meshes`.

If you have any problems moving the robot with `move_base_simple` goals in rviz try publishing a static transform between the map frame and any other virtual frame in order to ensure that the map frame is available. This can be done by running: `roslaunch tf static_transform_publisher 0 0 0 0 0 0 1 map my_frame 10`.

If you still get problems, check that your laptop supports OpenGL 3 (it is displayed when opening RVIZ) and try changing the frame from the Global Settings in rviz.

The robot should be displayed correctly and you can subscribe to additional modules by adding sensors using the add button in RVIZ.

CHAPTER 3

ROS Modules List

1. Pepper ROS Integration - `pepper_bring_up`
2. SLAM - `hector_slam`
3. Localization after initial mapping - `amcl`
4. Navigation - `move_base`
5. Data Acquisition on the Raspberry PI 3 - `rplidar`

3.1 Configuration

The two main launch files containing the full system launch can be found in `sparc/master_catkin_ws/src/pepper_sparc/launch/saved_slam.launch` and in `sparc/master_catkin_ws/src/pepper_sparc/launch/full_slam.launch`.

The **saved_slam.launch** starts the Pepper ROS integration with **pepper_bring_up**, loads a map from the maps folder inside the **pepper_sparc** node and runs **amcl** to localize the robot on the given map.

The **full_slam.launch** starts the Pepper ROS integration with **pepper_bring_up**, runs **hector_slam** to build the map.

Both launch files are compatible with the launch file used for running the **move_base** node, which can be found in the same folder under **navigation.launch**.

The system works with both the saved map and with the the SLAM module.

1. Hector_SLAM:

The main launch file can be found at: `sparc/master_catkin_ws/src/slam_launch/hector_mapping.launch`

```
<node pkg="hector_mapping" type="hector_mapping" name="hector_mapping" output="screen"
↪">
  <param name="pub_map_odom_transform" value="true"/>
  <param name="map_frame" value="map" />
  <param name="base_frame" value="base_footprint" />
  <param name="odom_frame" value="odom" />
```

(continues on next page)

(continued from previous page)

```

    <param name="map_resolution" value="0.05"/>
    <param name="map_size" value="1024"/>
    <param name="map_start_x" value="0.5"/>
    <param name="map_start_y" value="0.5" />
    <param name="laser_z_min_value" value="-1.0" />
    <param name="laser_z_max_value" value="1.0" />
    <param name="update_factor_free" value="0.9"/>
    <param name="update_factor_occupied" value="0.9" />
    <param name="map_update_distance_thresh" value="0.4"/>
    <param name="map_update_angle_thresh" value="0.06" />
    <param name="laser_min_dist" value="1.0"/>
</node>
<node pkg="tf" type="static_transform_publisher" name="base_to_laser_broadcaster"
↪ args="0 0 0 0 0 0 /base_footprint /laser 100" />

```

When running SLAM the map is updated every time the robot has moved over the **map_update_distance_thresh** value and every time the robot has rotated over **map_update_angle_thresh**.

The values for **update_factor_free** and **update_factor_occupied** were increased in order to ensure that the mapping can work in dynamic environments since the navigation module will take the dynamics into account.

1. AMCL

The mail configuration file can be found inside the saved_slam_launch file.

```

<node pkg="amcl" type="amcl" name="amcl" output="screen">
  <!-- Publish scans from best pose at a max of 10 Hz -->
  <param name="odom_model_type" value="diff"/>
  <param name="odom_alpha5" value="0.1"/>
  <param name="transform_tolerance" value="0.2" />
  <param name="gui_publish_rate" value="10.0"/>
  <param name="laser_max_beams" value="180"/>
  <param name="min_particles" value="5000"/>
  <param name="max_particles" value="50000"/>
  <param name="kld_err" value="0.05"/>
  <param name="kld_z" value="0.99"/>
  <param name="odom_alpha1" value="0.2"/>
  <param name="odom_alpha2" value="0.2"/>
  <!-- translation std dev, m -->
  <param name="odom_alpha3" value="0.8"/>
  <param name="odom_alpha4" value="0.2"/>
  <param name="laser_z_hit" value="0.5"/>
  <param name="laser_z_short" value="0.05"/>
  <param name="laser_z_max" value="0.05"/>
  <param name="laser_z_rand" value="0.5"/>
  <param name="laser_sigma_hit" value="0.2"/>
  <param name="laser_lambda_short" value="0.1"/>
  <param name="laser_lambda_short" value="0.1"/>
  <param name="laser_model_type" value="likelihood_field"/>
  <!-- <param name="laser_model_type" value="beam"/> -->
  <param name="laser_likelihood_max_dist" value="5.0"/>
  <param name="update_min_d" value="0.2"/>
  <param name="update_min_a" value="0.5"/>
  <param name="odom_frame_id" value="odom"/>
  <param name="base_frame_id" value="base_footprint"/>
  <param name="resample_interval" value="0.5"/>
  <param name="transform_tolerance" value="0.1"/>

```

(continues on next page)

(continued from previous page)

```
<param name="recovery_alpha_slow" value="0.0"/>
<param name="recovery_alpha_fast" value="0.0"/>
<param name="initial_pose_x" value="5.5"/>
<param name="initial_pose_y" value="10.0"/>
<param name="initial_pose_a" value="-2.6"/>
</node>
```

When running **amcl** the position of the robot is updated every time the robot has moved over the **update_min_d** value and every time the robot has rotated over **update_min_a**. The initial pose x,y and z values should be set as the initial guess for the algorithm and closer to the robot's home position as possible.

1. RPLidar

The main launch file can be found in `sparc/rpi_catkin_ws/src/launch/rplidar.launch`.

1. Move_Base

The main configuration files for move_base can be found in: `sparc/master_catkin_ws/src/pepper_sparc/nav_conf/`.

The parameters depend a lot on the environment.

Vision configuration

The vision component is composed of a set of independent modules, each responsible for one particular vision task (person detection, face recognition, QR code recognition, etc.). The implementation is located in the `sparc/external_platform/vision` folder, with the modules distributed in the following folder hierarchy :

- **facenet** - contains `FfaceNnet` network for face recognition and `Haar` cascades for face detection
- **image_provider** - contains the script that gets the stream of images from the robot
- **qr_codes_handler** - contains the script that generates, detects and recognizes QR codes
- **segmentation** - contains `Resnet-18-8s` network for person segmentation
- **tracking** - contains implementation of `SORT` algorithm for object tracking
- **yolo2** - contains `YOLOv2` network for object detection
- **data_processor.py** - auxiliary file for processing data that comes from the vision modules
- **vision_manager.py** - file for combining all the modules' outputs into a single vision result
- **run_vision.py** - main file for running the vision component as a stand-alone application

4.1 Software Prerequisites

To test the vision component both as a stand-alone application or within the big project, the machine must install all the following required packages:

- Cython (version $\geq 0.27.3$)
- H5py (version $\geq 2.7.1$)
- Matplotlib
- Naoqi (version ≥ 2.5)
- NumPy
- OpenCV-Python (version $\geq 3.3.0.10$)

- Python (version 2.7)
- Pillow
- Psutil
- PyCrayon (version >=0.5)
- PyTorch
- PyZBar
- Requests
- TensorFlow (version >=1.2)
- Scikit-image
- Scikit-learn
- SciPy

4.2 Testing the vision component

```
python run_vision.py [-r] [-v]
```

The script gets input images from a video camera or the robot's camera and displays 3 images:

- **RGB image** - input image with overlapped detections
 - **Blue bbox** - detected person. The text above represents the person ID, accuracy of detection and distance to the person
 - **Yellow bbox** - detected object. The text above represents the object name, accuracy of detection and distance to the object
 - **Pink bbox** - detected QR code. The text above represents the QR code ID, accuracy of detection and distance to the QR code
 - **Green/Red bbox** - recognized/unrecognized face. The text above represents the person name and accuracy of recognition
 - **Depth image** - depth image with overlapped bboxes representing detected people
 - **Segmented image** - black and white image, where the white pixels represent the segmented detected person
- tensorflow==1.2 Scipy scikit-learn matplotlib Pillowrequests Psutil

Numpy

Scikit-image

pyzbar

Naoqi 2.5.5 (if robot input required)

4.3 Testing the vision component

```
python run_vision.py [-r] [-v]
```

The script gets input images from a video camera or the robot's camera and displays 3 images:

- List item

Optional arguments: -r, --robot_stream = use the input from the robot's camera. By default the script uses the video camera of the machine. -v, --verbose = display information about execution time.

4.4 Fine-tuning the face recognition module

The input directory that contains the training images to fine-tune the pre-trained model are in `sparc/external_platform/vision/facenet/input_dir`.

The pre-trained model is located in `sparc/external_platform/vision/facenet/pre_model`, while the fine-tuned model is located in `sparc/external_platform/vision/facenet/classifier`.

1. Adding a new face to the database

- Add a new folder containing a set of images of the new face in `sparc/external_platform/vision/facenet/input_dir`. The folder name should match the person name.
- Align the face dataset using `python aligndata_first.py`
- Retrain the last layer of the pre-trained model using `python create_classifier_se.py`

1. Removing existing faces

- Delete the folder associated with the face to be removed from `sparc/external_platform/vision/facenet/input_dir`
- Retrain the last layer of the pre-trained model using `python create_classifier_se.py`

4.5 Generate new QR codes

The images representing the existing QR codes are located in `sparc/external_platform/vision/qrcodes_handler/qr_codes` folder. To generate new QR codes:

1. Delete existing images from `qrcodes_handler/qr_codes`
2. Modify the `generate_QRcodes` function in `qrcodes_handler/qrcodes_handler.py` and add each new QR code as follows:

```
new_qr_code = pyqrcode.create('new_qr_code_name') new_qr_code.png('./qr_codes/new_qr_code_image.png', scale=20)
```
3. Run `python qrcodes_handler.py`

Navigation Module

The navigation module is build upon ROS modules which are integrated into the main software. The navigation module is working in two phases, the SLAM phase in which Pepper is used to map the environment and must be moved manually or with an exploration technique in order to build a map and the navigation phase, when the robot can move on the saved map, in a dynamic environment.

5.1 Creating a new map

A new map can be created by using the **full_slam_launch** file found in the **pepper_sparc** node. Once the SLAM process is began, the robot can be moved in the simulator in order to start the mapping process. The **hector_slam** node publishes the occupancy grid on the */map* topic.

The map_saver node can be used at any point to save the map. The folder containing the maps can be also found inside the **pepper_sparc** node.

In order to save the map simply run:

```
roslaunch map_server map_server mymap.yaml
```

Make sure you save and mark the starting position of the robot, since this will be considered the home position for the navigation phase.

5.2 Running the navigation module on a map

Given a new map build in the previous step, **mymap.yaml** can be used at any point to navigate with Pepper. The robot should be in the home position. The starting home position must be edited in `sparc/master_catkin_ws/src/pepper_sparc/nav_conf/base_local_planner_params.yaml` as described in the ROS Modules chapter.

First the map should be loaded and the robot should localize itself on the map. This can be done by running the **saved_slam_launch** file inside the **pepper_sparc** node. You should be able to see the robot in the right starting

position. A small error in the starting position is not a reason to worry, since the localization will improve as soon as the robot starts to move.

The **navigation.launch** can be found in the same **pepper_sparc** node and launches the **move_base** node in order to make Pepper navigate in a dynamic environment while avoiding obstacles.

Speech Component

The speech component is based on a publisher-subscriber paradigm. The speech detected by the publisher is transmitted to the subscriber through a ROS topic. This component is implemented in **sparc/external_platform/robot_interaction** folder:

- `speech_recognition_publisher.py`
- `speech_recognition_subscriber.py`

The publisher and subscriber communicate through *speech_text* channel. If the topic name is changed, it must be changed in both files:

```
# Speech_recognition_subscriber.py
self.recognition_subscriber = rospy.Subscriber('speech_text', String, self.callback_
↪text)

# Speech_recognition_publisher.py
self.publisher = rospy.Publisher('speech_text', String, queue_size=10)
```

6.1 Speech recognition publisher

The publisher file uses a Google speech recognition engine, for both English and Romanian languages. The script can be configured to work in either language using the *language* parameter in the file. Moreover, the implementation allows the user to input the sentences from keyboard, through the *audio_stream* parameter.

As the publisher uses a ROS topic to send the recognized sentences, a *roscore* instance must be started. To run the script:

```
python speech_recognition_publisher.py
```

The script will prompt a question “Audio stream? [y/n]:” to which the user can respond with “y” (yes) or “n” (no), which represent if the script should use the input from the microphone or not.

6.2 Speech recognition subscriber

The subscriber file contains a class that subscribes to the ROS topic and tries to interpret the recognized speeches using [wit.ai](#). The wit.ai application extracts the action to be done and the parameters, if possible, and passes the result to the rest of the application through the argument method received in the class' constructor.

To run the script (if you run it as a standalone application, please make sure that the `robot_stream` flag is false and):

```
python speech_recognition_subscriber.py
```

6.2.1 Wit.ai configuration

To work for both languages English and Romanian, two applications were created: [sparc](#) and [sparc-ro](#). Each application is trained with a set of queries specific to the associated language.

The wit.ai application extracts relevant entities from the queries. The user-defined entities used in the project are:

- **intent** = specifies which type of action is intended. The *intent* entity is seen as a trait of the sentence and considers the sentence as a whole. The application is trained to recognize the following intents:
 - **hello** = greet someone
 - **stop** = stop the current behavior
 - **say** = say something
 - **go to** = go to a position
 - **find** = find a target in the environment
 - **look** = look around for the target
 - **health** = display health statistics
 - **reminders** = display and read out reminders for a person
 - * **next/previous** = intents required for voice interaction in reminders behavior (the user can vocally switch to next or previous reminder)
- **target** = parameter for *say*, *go to*, *find*, *look* and *reminders* intents. The entity is attached to a specific part of the sentence, which represent the target. Each keyword of the target entity can have multiple synonyms which can be configured.
- **health_entity** = parameter for *health* intent. Similar to *target* entity.

To use a wit.ai application, an access key must be provided. The access keys for this project are defined in `sparc/external_folder/config.py` file, in the `access_keys` parameter.

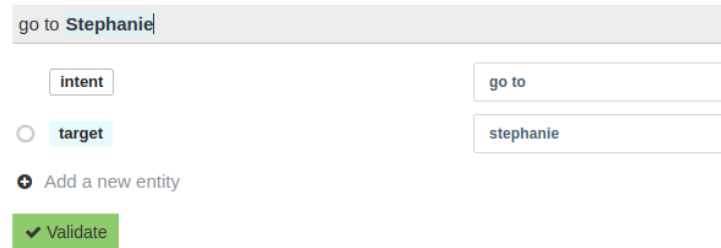
The `mandatory_intent_entities` parameter in `sparc/external_folder/config.py` file is a dictionary which contains all the existing intents with the associated list of mandatory entities. That means that if an entity from the mandatory list is not extracted from the sentence when that specific intent is recognized, the command is invalid. To add a new intent, a new (*key*, *value*) pair needs to be added in the dictionary, where the key represent the intent name and the value the list of mandatory entities.

6.2.2 Understanding new phrases

To test with new sentences the wit.ai application, follow the next steps:

1. Go to the [application page](#)

2. In the “*User says*” box introduce your sentence (e.g.: go to Stephanie)



go to Stephanie

☒ intent ☐ target

go to

stephanie

[Add a new entity](#)

Validate

3. If previously trained, the application will extract some entities
response example

If you want to train the application with the new sentence, select the right entities and click *Validate*. This will start the re-training process (the training status, *Done*, *On-going* or *Scheduled*, can be seen in the top-left corner). To associate a *target* or *health_entity* entity to the input sentence:

1. Select the part of the sentence associated with the entity
2. Click “*Add a new entity*”
3. Choose the appropriate entity
4. Select the associated value (keyword)