How To Use:
ROS - MUSIC Toolchain

April 25, 2016

This tutorial shows how to use the ROS - MUSIC Toolchain by drawing an example of a Braitenberg Vehicle. I describe the robotic setup and display the commands necessary to run ROS, MUSIC and the Toolchain, but will not go into detail about the ROS or MUSIC background. If you are interested in learning more about ROS or MUSIC, I suggest to read the ROS tutorial[1] or the MUSIC manual[2].

For the example of the Braitenberg Vehicle, I use Gazebo for the robotic simulation, ROS and MUSIC for communication and NEST for neural simulations. In this tutorial, I start to describe the nature of a Braitenberg Vehicle, continue with the robotic setup and the MUSIC specifications and will end with a summary.

1 On the Braitenberg Vehicle

A Braitenberg Vehicle is a cybernetic creature described by Braitenberg in his book "Vehicles: Experiments in synthetic psychology"[3]. Here, I implement the Vehicle **3b** called "explorer".

The vehicle has two sensors, attached in the front of the vehicle, and two motors, attached to its rear moving the vehicle according to their speed (see figure 1). The sensors are connected to the motors in a crossed and inhibitory way, meaning, the left sensor is (negatively) connected to the right motor and vice versa. Initially, the motors run with a constant positive speed moving the vehicle straight forward. If the left sensor is activated by sensing a source (for example by sensing light), the right motor is slowed down resulting in a right turn of the robot. Activating the right sensor results in a left turn.

[1] http://wiki.ros.org/ROS/Tutorials
[2] https://www.incf.org/documents/program-documents/Music-UsersManual.pdf
[3] Braitenberg, Valentino. Vehicles: Experiments in synthetic psychology. MIT press, 1986.

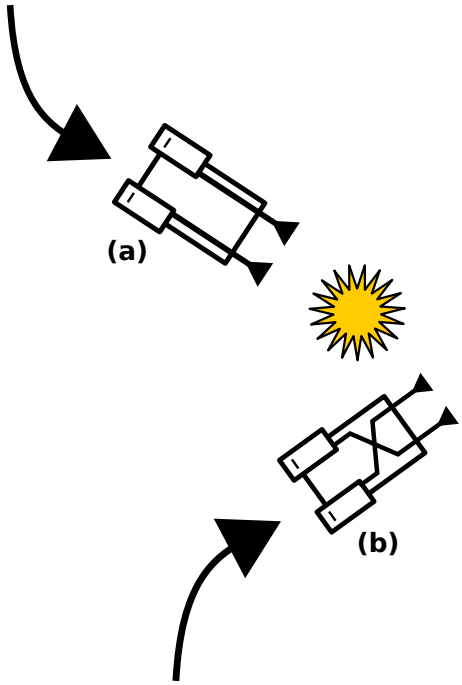

Figure 1: Braitenberg Vehicle 3 "Explorer"

Braitenberg describes the behavior of this vehicle as following:

'You will have no difficulty giving names to this sort of behavior. These vehicles LIKE the source, you will say, but in different ways. Vehicle 3a LOVES it in a permanent way, staying close by in quiet admiration from the time it spots the source to all future time. Vehicle 3b, on the other hand, is an EXPLORER. It likes the nearby source all right, but keeps an eye open for other, perhaps stronger sources, which it will sail to, given a chance, in order to find a more permanent and gratifying appeasement.' (p. 12)

2 On the Robotic Setup

I implemented the vehicle as Pioneer 3AT robot simulated in Gazebo. The Pioneer has four wheels and has an attached laser scanner, publishing on the ROS topic "/Pioneer3AT/laserscan", and having its field of view in front of the robot (see figure 2). The laser scanner implements the two sensors of the Braitenberg Vehicle. The activity of left sensor of the "explorer" is realized as the average distance measured on the left hemisphere of the laser scanner; the right sensor is implemented accordingly.

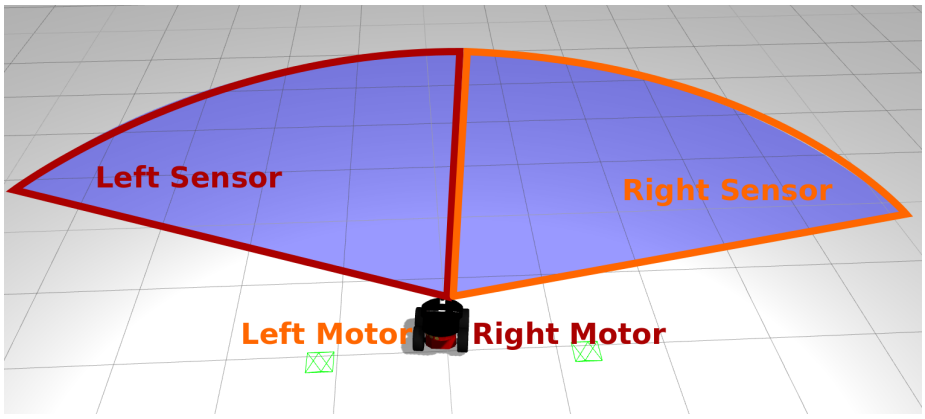

Figure 2: Robotic realization of the Braitenberg Vehicle 3 "Explorer"

In the description of the "explorer", two motors are used, which don't have an exact equivalent in the robotic hardware. Instead of controlling the motors of the Pioneer3AT directly, it is steered in a simpler way by using a ROS::Twist command[4], which consists of a linear and an angular part. The linear part determines the speed of the robot, the angular is turning the robot, both can have positive or negative values (meaning forward or backward speed, or left/right turns). The Pioneer3AT is listening for commands at the ROS topic called "/Pioneer3AT/cmd_vel". The Pioneer3AT setup is specified in an ".sdf" file and can be found in the repository[5].

3 On the MUSIC Configuration File

In this section, we map the functionality of the Braitenberg Vehicle to the Pioneer3AT hardware. Although this could be done in an easier way, I'm using all the parts of the ROS - MUSIC Toolchain to demonstrate its capability to perform closed-loop simulations.

3.1 Common Adapter Properties

All adapters in the toolchain have some parameters in common.

- music_timestep: Specifies the interval between MUSIC ticks in seconds. Default: 0.001
- stoptime: Specifies the duration of the simulation in seconds.
- rtf: The realtime factor of the simulation. Default: 1.

[4] http://wiki.ros.org/geometry_msgs
[5] https://github.com/weidel-p/ros_music_adapter

Also, there are the common MUSIC properties "binary", "args" and "np", which I won't explain here, but can be read up in the MUSIC documentation[6].

The common behavior of all adapters in the toolchain is to specify an input and an output port which reflect the properties of the neural and robotic simulation. For example, the laserscanner of the Pioneer3AT measures the distance to object by projecting 100 beams into its environment. This results in a 100 dimensional data array which have to be communicated by MUSIC.

In the following sections, I will describe the setup of the adapters used in this example.

3.2 The ROS Sensory Adapter

The purpose of this adapter is to listen to a specified ROS topic and convert the arriving ROS message into a C++ array which can be communicated via MUSIC. In the MUSIC configuration file we set the ROS topic to "/Pioneer3AT/laserscan", the message type to "Laserscan" and the sensory update rate to 20 Hz (this is found in the specifications of the laserscanner in the robotic setup). The data array has 100 dimensions, one for each beam of the laserscanner.

Algorithm 1

```
[sensor]
  binary=ros_sensor_adapter
  args=
  np=1
  music_timestep=0.05
  ros_topic=/Pioneer3AT/laserscan
  message_type=Laserscan
  sensor_update_rate=20
```

3.3 The Connect Adapter

The connect adapter performs a linear transformation on the incoming array. For this example, the connect adapter is used to reduce the dimensionality of the laserscanner data d from 100 dimensions to two. The new, two dimensional, data array s represents the left and the right sensor in the vehicle:

$$s_0 = sensor_{left} = \frac{\sum_{i=0}^{49} d_i}{50}$$

$$s_1 = sensor_{right} = \frac{\sum_{i=50}^{99} d_i}{50}$$

This transformation is specified in a file containing connectivity weight between all the dimensions of incoming and outgoing data in JSON format.

[6] https://www.incf.org/documents/program-documents/Music-UsersManual.pdf

Algorithm 2

```
[converge]
  binary=connect_adapter
  args=
  np=1
  music_timestep=0.05
  weights_filename=braitenberg3_converge_weights.dat
```

3.4 The Rate Encoder

The rate encoder is translating the incoming data array containing continuous data to spiking activity. More information about the en- and decoding as well as performance measurements can be found in the publication about this toolchain[7].

Algorithm 3

```
[encoder]
  binary=rate_encoder
  args=
  np=1
  music_timestep=0.05
  rate_min=1
  rate_max=200
```

3.5 The NEST Simulation

In the description of the Braitenberg Vehicle, the connections are crossed between the sensors and the motor. This is achieved in a NEST simulation, by connecting a MUSICEventInProxy to a MUSICEventOutProxy via two parrot neurons. The NEST script can be found in the repository.

In the MUSIC configuration file, it is possible to pass arguments (like simulation time, timestep, etc) to NEST by using the "args" field in the MUSIC configuration file.

Algorithm 4

```
[nest]
  binary=./braitenberg3_pyNEST.py
  args=-s 0.05 -t 1000 -n 2
  np=1
```

3.6 The Linear Decoder

The linear decoder is translating spiking activity to a continuous data array by applying a low-pass filter on the incoming spikes with an exponential kernel

[7] https://arxiv.org/abs/1604.04764

(with an time-constant "tau") and weighting the filtered spike traces with a multiplicative factor which is specified in an extra file using the JSON format. In this example, the linear decoder is used to decode the spiking activity received from NEST to continuous data which can be transformed to ROS messages in the ROS Command Adapter.

Algorithm 5

```
[decoder]
  binary=../../linear_readout_decoder
  args=
  np=1
  music_timestep=0.05
  tau=0.03
  weights_filename=braitenberg3_readout_weights.dat
```

3.7 The ROS Command Adapter

The ROS Command Adapter transforms a C++ array to a ROS Message which is then sent to the (simulated) robot via a ROS Publisher. The mapping from array to message type can be done similar as has been shown in the Sensor Adapter, but can also specified in a dedicated file. Here, a ROS::Twist message is created from the filtered activity of the NEST simulation.

The activity of the two neurons is interpreted as the speed of the two motors described in the "explorer". In order to map the speed of the two motors m_0 and m_1 to a ROS::Twist message, the following transformation is used

$$linear.x = \frac{m_0 + m_1}{2}$$

for the translations (or speed) of the robot, and

$$angular.z = m_0 - m_1$$

for the rotation. This mapping is specified in the config file "braitenberg3_twist_mapping.dat".

Algorithm 6

```
[command]
  binary=../../ros_command_adapter
  args=
np=1
music_timestep=0.05
ros_topic=/Pioneer3AT/cmd_vel
message_mapping_filename=braitenberg3_twist_mapping.dat
command_rate=20
```

3.8 Connecting the binaries

After specifying all the binaries in the toolchain they still have to be connected. The dimensionality of the data are specified here in the brackets after each connection.

Algorithm 7

```
sensor.out->converge.in[100]
converge.out->encoder.in[2]
encoder.out->nest.in[2]
nest.out->decoder.in[2]
decoder.out->command.in[2]
```

3.9 Summary

In this example of a Braitenberg Vehicle "explorer", the laserscanner of a Pioneer3AT robot is projecting 100 beams in the environment, resulting in a 100 dimensional sensory data. This sensory data is transformed to a C++ array by the ROS Sensor Adapter and converged to a two dimensional array (implementing the two sensors specified in the "explorer" description) by the Connect Adapter. This compact sensory data is then encoded into spiking activity by a Regular Rate Encoder and sent to NEST. In the NEST simulation, the connections are crossed and sent to the Linear Decoder which reconstructs a continuous signal from the spiking activity. The ROS Command Adapter maps this continuous signal to a ROS::Twist message and sends this command to the Pioneer3AT robot.

4 On the Execution of the Simulation

After installing ROS, MUSIC and the Pioneer3AT robot (can be found in the repository), this example can be executed by running following commands:

Algorithm 8

```
roslaunch pioneer3at demo.launch
gzclient
mpirun -np 6 music braitenberg3.music
```