Soccer Behaviours Exercise Sheet

Philipp Allgeuer

Autonomous Intelligent Systems, Computer Science, Univ. of Bonn, Germany
pallgeuer@ais.uni-bonn.de
http://ais.uni-bonn.de

1 Introduction

The goal of this exercise sheet is to introduce you to the programming of soccer behaviours using C++ and the Robot Operating System (ROS) software framework. We will be using the open source State Controller Library to create a finite state machine that can control a NimbRo-OP robot.

The target outcome of this exercise, alongside an enriched understanding of the State Controller Library and the NimbRo-OP software framework, is the implementation of a simple ball approach and kick behaviour that can avoid obstacles and kick the ball towards the goal. A kinematic simulation of a NimbRo-OP robot in a TeenSize soccer environment will be used to evaluate your developed behaviour algorithm.

2 The State Controller Library

The State Controller Library is a generic platform independent C++ framework that allows for the realisation of finite state machines and multi-action planning generalisations thereof. The library comes as part of the NimbRo-OP source distribution, but is also freely available as an independent C++ library at http://sourceforge.net/projects/statecontroller/. It is suggested at this point for you to look at the Doxygen documentation of the State Controller Library, in particular the module description page, until you can answer the following questions. This will make the remainder of the exercise easier to complete. The documentation can be found in the source directory of the NimbRo-OP code under src/nimbro/doc/out/html/group__StateControllerLibrary.html, or alternatively, if the documentation has not been compiled on your system yet, you can access the documentation by downloading the library from the aforementioned SourceForge page and opening the doc/State Controller Library.html file.

Questions you should be able to answer:

- What is a state controller and what is it used for?
- What is the required base class for every state controller object?
- What is the recommended base class for every state controller state? What arguments and template parameters do you need to supply to the constructor of this class?

- What is the state queue, and how do you transition to another state?
- What are the two possible actions that can be returned from the `execute()` callback of a state?
- What are the state and the state controller callbacks, and when do each of them get called?
- How do you share data between states? How do you access state controller data from a state?
- What are the differences between state variables, state parameters and shared variables?

Note however that as this application is fairly simple, it may not be necessary to use more complex features of the library, such as for example enqueueing multiple states into the state queue. The template source files for this exercise (see later) may also help you with the process of familiarising yourself with the library.

3 The ROS Framework

Almost all of the calls to ROS-specific functions have been abstracted away from the required exercise code, and can be found in `behaviour_exercise.cpp`. The collection of functions and functionality that performs this abstraction however must still be learnt. Before continuing you should familiarise yourself with the following features.

Useful additional features to understand:

- Send gait command: One of the first questions that naturally arises is how to make the robot walk. This can be done using the `sendGaitCommand()` function. It is recommended that you look into this function's source code to understand what it does. Each of the gait target velocity vector components are specified using dimensionless parameters on the unit interval (i.e. from 0 to 1). A boolean flag is used to turn the gait on and off as a whole.
- ROS console messages: ROS provides support for displaying debug, informative, warning and error messages in the console. Throttling and stream support is also included for each variant of the console functions. For example, the following warning will be printed at most every 0.5s, and uses the `C++` streaming syntax.

  ```
  ROS_WARN_STREAM_THROTTLE(0.5, "Encountered bad x = " << x);
  ```

  Refer to `http://www.ros.org/wiki/rosconsole` for more information.
- Plotter: When ROS console messages are insufficient for debugging, the plotter may be used. Refer to the `plotScalar()` and `plotVector()` functions for more information on this. A time history of the data passed to these functions appears in graphical form in the visualisation GUI (see *Getting Started*).

– Configuration server: When writing a behaviour it is often useful to be able to change certain parameters and try again without recompiling or restarting the behaviour node. This allows for faster algorithm tuning and design iterations. This can be done by defining configuration parameters, which can subsequently be modified during runtime using the parameter tuner widget in the visualisation GUI (see *Getting Started*). Refer to the definition and use of the `m_be_run()` parameter in the template exercise code for an example of how to define a boolean configuration parameter. Configuration parameters of floating point type can also be defined in an analogous manner, except that the parameter class constructor needs to be called with additional parameters specifying allowable ranges and so on.

4 Getting Started

For this section you will need an open console (e.g. Konsole) and an open code editor (e.g. KDevelop). Note that at the end of your session you will have to **save all files that you have modified onto a USB** to prevent losing them when someone else uses the computer after you.

4.1 Resetting the Workspace

If someone has already modified the behaviours source code on your computer, or if you just want to be sure, run the following:

```
$ nimbro source
$ git reset --hard HEAD
```

4.2 Running the Exercise Code

1. Navigate to the `behaviour_exercise` package folder in the editor and open `behaviour_exercise_template.h` and `behaviour_exercise_template.cpp`. Examine those source files and see what you think they will do.

   ```
   $ roscd behaviour_exercise
   ```

2. Run the following command:

   ```
   $ nimbro make
   ```

   This command should successfully build the NimbRo-OP code, including the behaviours exercise template code.

3. In four **separate** consoles (or console tabs), run the following commands in the order given. Always wait for each process (or 'node') to start up before proceeding to the next. Note that tab completion may come in handy here.

   ```
   $ roscore
   $ roslaunch behaviour_exercise behaviour_exercise_robot.launch
   $ roslaunch behaviour_exercise behaviour_exercise_behaviour.launch
   $ roslaunch behaviour_exercise behaviour_exercise_visualisation.launch
   ```

You should see that the robot process loads several motions and then finishes initialising, the behaviour process starts running the idle state, and the visualisation process opens up a graphical user interface (GUI) with a 3D visualisation. Play around with the parameter tuner, plotter and RViz widgets, in particular looking for configuration parameters and plotted variables with the keywords 'Behaviour Exercise'.

4. Press the Fade In button on the left hand side of the screen (in the diagnostics widget) to make the robot stand up. It is only necessary to do this once for every time that you restart the robot node. Once the robot is in the standing state you may signal to the behaviour node that it should start controlling the robot by checking the run configuration parameter. As of yet the robot should not react though, as you have not written any control code yet! By modifying some of the other configuration parameters however, you may observe how the field is randomly initialised when run is checked.

4.3 Customising the Template Exercise Code

You may have noticed that the behaviours node declares whose solution it is, using a call to ROS_WARN. Find the corresponding line in the source code and add your name and date in place of the default text.

5 Writing the Ball Approach and Kick Behaviour

1. Following the hints given in the template code, create a new state called `SearchForBallState`. Make sure you display a ROS info message at the start of the `execute()` callback for debugging purposes. Also, make the robot turn on the spot when the state controller is in this state. Now uncomment the four commented lines in the `execute()` callback of the idle state in order to allow the state controller to enter the search for ball state when appropriate. Make and launch the code as described in the *Getting Started* section, and verify that the robot turns on the spot when run is checked. You may choose to implement a more advanced search for ball behaviour when you have completed a few more of the following exercise questions, and have a functioning behaviour node.
2. Create another state called `ApproachBallState`, once again adding code to display a (throttled) ROS info message to display the current state. For now just make the robot walk straight ahead with constant velocity when it is in the ball approach state. Add code to `SearchForBallState` to transition to the ball approach state when the ball is seen, and add code to `ApproachBallState` to transition back to the search for ball state when the ball has not been seen for 1.0s. Refer to the `RobotSC::haveBall()` function for this. Make and launch the code as described in the *Getting Started* section, and verify that the robot now turns on the spot until it sees the ball, then starts walking straight ahead until it cannot see the ball again, at which point it starts turning on the spot again, and so on.

3. Write a more sophisticated ball approach behaviour that aims to position the right (or left) foot of the robot behind the ball, with the robot facing the nominated goal. The `ballVector()`, `goalVector()`, `haveBall()` and `haveGoal()` functions will be of great use here. Make and launch the code, and verify that your algorithm works with randomised robot and ball positions, as well as when the `random_goal` configuration parameter set.
4. Add code to the ball approach state that transitions to a new state called `KickBallState`, whenever the robot is positioned well enough behind the ball to be able to kick it into the goal. In the `execute()` callback, use the `robotIsStanding()` and `playRightKick()` functions to execute the required kick. It may prove useful to override the `activate()` callback of this state, for example so that you can send a gait command to stop walking only exactly when the kick ball state is entered. Add code to return to the search for ball state when the kick is over (i.e. the robot is standing again).

6 Taking it Further

If you have not done so already, now enable the `use_imperfect_measurements` configuration parameter. This simulates noise and uncertainty in the object detections. You can now also enable the simulation of an obstacle using the `use_obstacle` parameter. You will need to improve your ball approach behaviour to be able to avoid the obstacle, avoid kicking the ball into the obstacle, and deal with the fact that it may be obstructing vision of the ball. This would be an appropriate time also to improve your search for ball behaviour. Further improvements could also include dynamically choosing which foot to kick with, rather than always using a fixed foot.

7 If You Get Stuck

If you get stuck, then you can always ask the supervisor of the exercise for help. If you run out of time to finish the exercise or need an example to look at, then you may refer to the `behaviour_exercise_sample.h` and `behaviour_exercise_sample.cpp` source files, which contain a complete sample solution to the exercise. You can compile the sample in place of your exercise code by modifying `CMakeLists.txt` on the marked line, and changing the header included in the `behaviour_exercise.cpp` source file. Revert these changes to compile your exercise again. No cheating if you are still working on the exercise though!

Enjoy!

20/07/13