MANCHESTER
1824
The University of Manchester

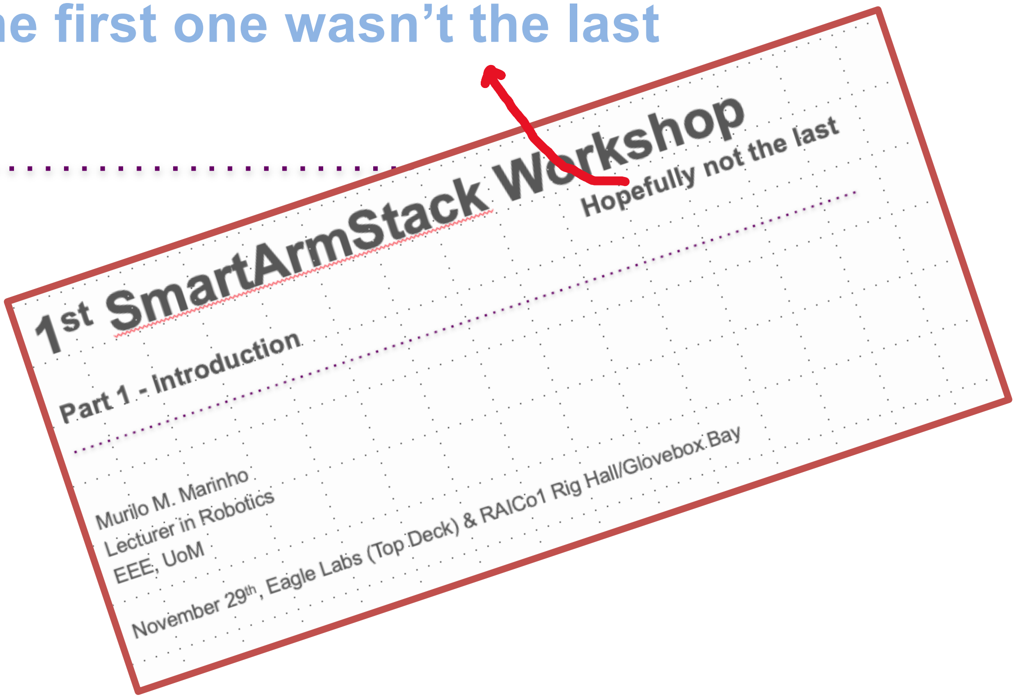
2nd SmartArmStack Workshop

Phew, the first one wasn’t the last

Part 2 – Making new sas::RobotDrivers

Murilo M. Marinho
Lecturer in Robotics
EEE, UoM

April 2nd, Eagle Labs (Top Deck) & RAICo1



1st SmartArmStack Workshop

Hopefully not the last

Part 1 - Introduction

Murilo M. Marinho
Lecturer in Robotics
EEE, UoM

November 29th, Eagle Labs (Top Deck) & RAICo1 Rig Hall/Glovebox Bay

Summary

Why was I ~~forced~~invited to be here AGAIN?

- Project Sustainability: Person leaves, projects die off
  - Research stalled
  - Follow-up projects must reinvent the wheel.
- Project Collaboration: no mutual benefit
  - Person A creates code with their preferred settings
  - This code is indecipherable to any person $B \neq A$.
- Time efficiency: On-topic procrastination
  - Person C spends 6 months making yet another robot driver unusable for any $D \neq C$.
  - This cannot be used directly for a research paper, just as support for experiments.
  - This cannot (should not) be part of their PhD thesis.

Keep legacy alive

- DQ Robotics
  - The de facto common language among many groups.
  - Contribute here always when suitable.
- SmartArmStack
  - Complementary to DQ Robotics.
  - Contribute here always when suitable.
- Whatever you propose next
  - Complementary to DQ Robotics and SmartArmStack.

Use SmartArmStack!

What is SAS?

SmartArmStack in a glance

- Arguably not rubbish
- Abstract away driver code from ROS code
- Abstract away kinematic controller code from ROS code
- Many utility functions
  - Datalogger (export data to .mat)
  - Clock (a proper clock with data-keeping and statistics)
  - Conversions between ROS interfaces, DQ Robotics, Eigen…
- C++ and Python (pybind11) support

Packages in SAS

LGPL packages

General purpose	**sas_core**	LGPL	The part of SAS that does not depend on ROS2. Planned to be moved away completely.
	sas_common	LGPL	Common ROS2 code used throughout SAS.
	sas_datalogger	LGPL	Log information at execution time and output as .mat file.
	sas_msgs	LGPL	ROS messages that were made redundant in ROS2.
	sas_conversions	LGPL	Conversions among ROS2 messages, DQ Robotics, Eigen, std::vector etc
	sas_robot_driver	LGPL	Client—Server library for configuration-space monitoring and control. Generic ROS node for any sas_robot_driver subclasses.
	sas_robot_kinematics	LGPL	Client—Server library for task-space monitoring and control.
Robot specific	**sas_robot_driver_ur**	LGPL	A sas_robot_driver implementation for UR's bCap controlled robots
	sas_robot_driver_kuka	LGPL	A sas_robot_driver implementation for KUKA’s FRI controlled robots
	sas_robot_driver_denso	LGPL	A sas_robot_driver implementation for DensoWave's bCap controlled robots

MANCHESTER 1824
The University of Manchester

Which ones are closed source?

- Source is closed but you can use the binary for Non-Commercial Purposes

sas_operator_side_receiver	NonCommercial	Receive any number of master device data from external PC and expose it to ROS.
sas_patient_side_manager	NonCommercial	Manage master device behaviour, including clutch switching and expose to the SAS Client—Server paradigm.
sas_robot_kinematics_constrained_multiarm	NonCommercial	Centralized control of any number of arms with configurable constraints obtained from CoppeliaSim.
sas_robot_driver_escon	NonCommercial	Possibly commercializable in projects such as ImpACT, Moonshot, and JAXA
sas_robot_driver_aia	NonCommercial	Possibly commercialise as part of the Moonshot multi-arm platform
sas_robot_driver_festo	NonCommercial	

For existing robots

- Use and rejoice in C++ and Python!
  - You can benefit from kinematic controllers
  - Combine them with other robots
  - Benefit from teleoperation packages
  - And so on

What about new robots???

Tutorial

For any new robot

- Two-step magic

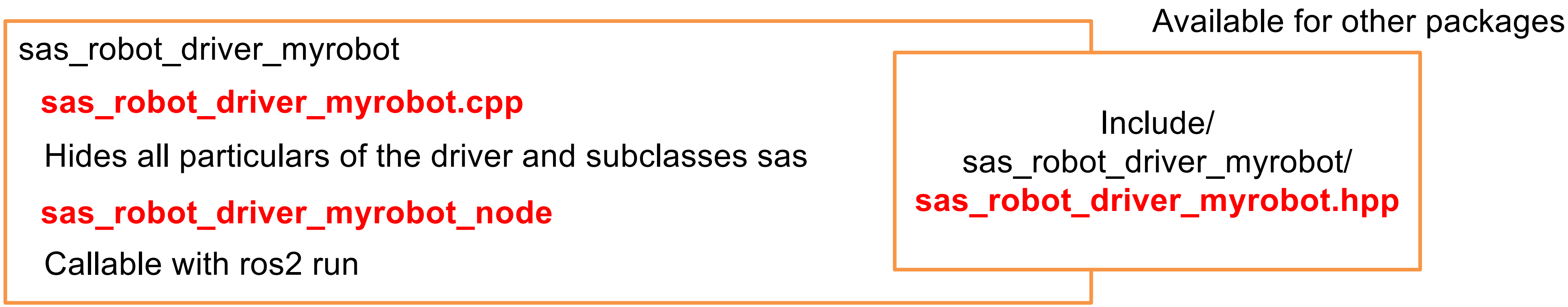
1. Create a subclass of **sas::RobotDriver**
2. Create a ROS2 node that configures the sas_robot_driver and create a ROS2 loop with **sas::RobotDriverROS**
3. Done!

- New robot can benefit from kinematic controllers
- New robot can be combined with other robots
- New robot can benefit from teleoperation packages
- And so on

Overview

sas_robot_driver_myrobot

- The package will be called “sas_robot_driver_myrobot”
  - Create a subclass of “sas_robot_driver”
    - This will encapsulate all particulars of the driver, and these will not be seen externally.
  - Expose a “pure” header that does not depend on internals.
  - Create a node to configure the driver and integrate with ROS2.



sas_robot_driver_myrobot

sas_robot_driver_myrobot.cpp

Hides all particulars of the driver and subclasses sas

sas_robot_driver_myrobot_node

Callable with ros2 run

Available for other packages

Include/
sas_robot_driver_myrobot/
sas_robot_driver_myrobot.hpp

Step 0

Create the package

- package.xml is trivial
- CMakeLists.txt is based on a boilerplate, but we also might depend on the particulars of the driver in this stage.

Step 1

Subclass of sas::RobotDriver

- This class is in sas_core. This is because the class does not depend on ROS2 and eventually these will be an external (not ROS2) package.
- Notice the PIMPL implementation which is highly recommended for this type of implementation.

Step 2

Creating the ROS2 Node

- The boilerplate code will allow you to easily create this code. We leave all ROS2 management to the class sas::RobotDriverROS which
  - Receives as argument a subclass of sas::RobotDriver
  - Has a method called control_loop() that will manage the loop for us.

Step 3

Create a suitable ROS2 launch file

- The launch file is simply to set the parameters of the ROS2 node for us.
- The only notable aspect is that the ”name” parameter is very useful for sas. It will define the topic prefix and allow other sas packages to know which drivers to talk to.

Step 4

Python access

- Using the boilerplate code, you can access any sas_robot_driver Node through Python. Notice that we use the Node name to find the particular Node we want to access.

Issues?