CSSR for

Culturally Sensitive Social Robotics for Africa

D5.5.1.L Programming by Demonstration

Due date: **9/12/2024**
Submission Date: **9/12/2024**
Revision Date: **16/12/2024**

Start date of project: **01/07/2023** Duration: **36 months**

Lead organisation for this deliverable: **Carnegie Mellon University Africa**

Responsible Person: **Daniel Barros** Revision: **1.1**

Project funded by the African Engineering and Technology Network (Afretec) Inclusive Digital Transformation Research Grant Programme		
Dissemination Level		
PU	Public	**PU**
PP	Restricted to other programme participants (including Afretec Administration)	
RE	Restricted to a group specified by the consortium (including Afretec Administration)	
CO	Confidential, only for members of the consortium (including Afretec Administration)	

Executive Summary

Deliverable 5.5.1.L documents the development and functionality of a ROS package designed to enable the Pepper robot to execute upper body and hand gestures learned through Programming by Demonstration (PbD). This software module is tailored for deployment on the real Pepper robot and is not intended for use in simulation.

The deliverable includes:

- The documented code for the `programming_from_demonstration` ROS package.
- A comprehensive report detailing the system architecture, component requirements, development procedures, and an explicit definition of the module's functional characteristics.
- A user manual to assist users in configuring and launching the module, with additional instructions for developers who wish to adapt the package for custom applications.

The package's interface design covers input, output, and control data, while also specifying appropriate data structures. All development activities adhere to the software engineering standards outlined in Deliverable D3.2.

Contents

1 Introduction

This deliverable extends D5.5.1 Gesture Execution by introducing programming gestures by demonstration. This technique enables the robot to learn iconic and symbolic gestures directly from human demonstrators. Users interact with the system by performing gestures in front of an RGBD camera. Human joint positions are estimated using a skeletal model and retargeted to Pepper's joint angles. Various filters can be applied to increase the smoothness of imitation of the demonstrated movements.

The package includes a component for recording data from multiple Pepper sensors during the demonstration process. Additionally, a graphical user interface is provided to facilitate the control and management of demonstrations and data collection.

Section 2 specifies the requirements for the software module, covering functionality, performance, and usability aspects. Section 3 outlines the functional characteristics of the module, focusing on the retargeting of angles from the skeletal model to Pepper's joints. Section 4 addresses the interface design, specifying the data inputs, outputs, and control mechanisms. It also describes how these data are accessed or made available, whether through files or ROS mechanisms like subscribers, services, or actions. Section 5 details the module design, including implementation specifics and the connections between components. Section 6 provides instructions for running the PbD system. Section 7 documents the unit testing procedures for the module's components, ensuring reliability and performance. Finally, Section 8 offers guidance for developers on extending the software for custom applications.

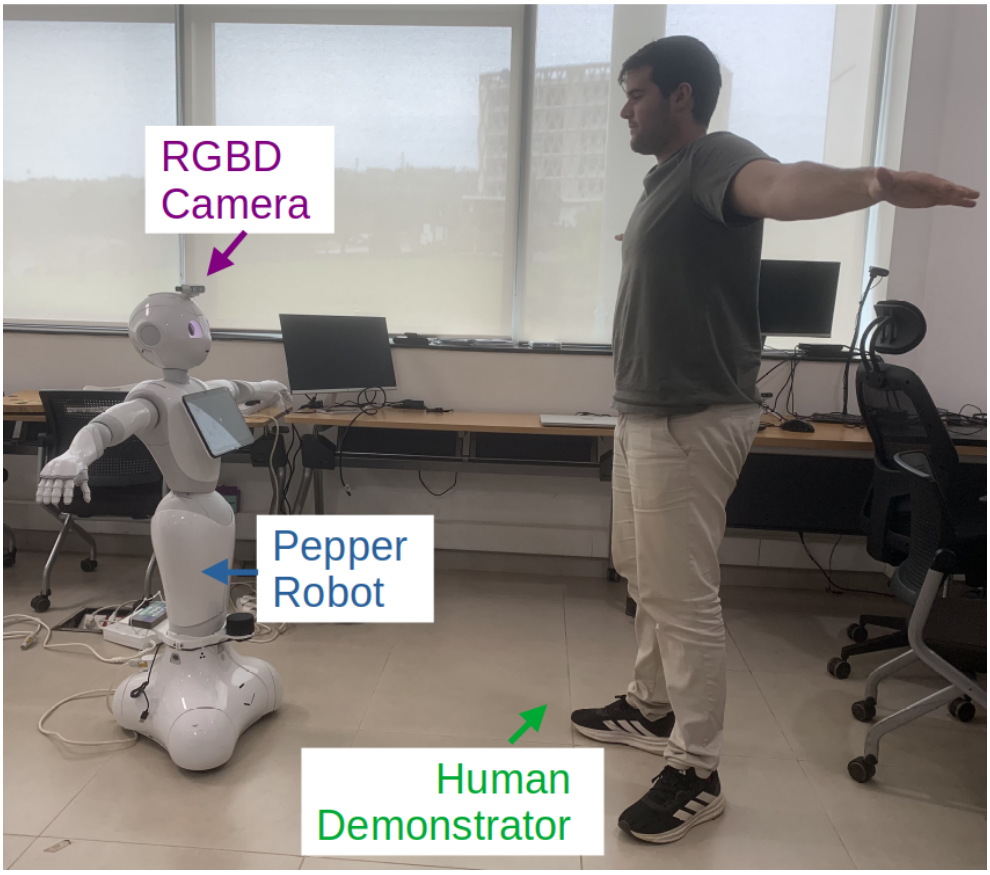

Figure 1: Programming by Demonstration Setup: The human demonstrator gestures in front of an RGBD camera and Pepper imitates the movements.

2 Requirements Definition

- **Core Functionality:**
  - The software must enable the Pepper robot to imitate upper-body gestures performed by humans in real-time (online) and from video (offline).
  - Pepper should be able to imitate all gestures performed in the coronal plane and in front of the body.
  - If joint angles cannot be calculated, the controller must maintain its previously commanded joint positions.
- **Data Processing and Filtering:**
  - A filtering function must smooth noisy angle estimations from the skeletal model, enhancing the naturalness of Pepper's movements.
  - Switching between different filtering options should be easy and intuitive for the user.
- **Demonstration and Recording:**
  - The module must allow users to record demonstrations, storing data in an organized and accessible format.
  - Recorded trajectories must be easy to store and replay.
  - It should be possible to record any type of data coming from Pepper's sensors.
  - The recording component must be expandable to support other user input modalities and robots.
- **User Interface:**
  - An intuitive user interface must be provided to facilitate control of the demonstration and recording process.
  - Users must have real-time access to system logs during the demonstration process to gain operational insights.

3 Module Specification

The Programming by Demonstration module, also referred to as PepperTrace module, consists of three components. The first one, `skeletal_model`, addresses the first and second set of requirements by enabling Pepper to imitate the upper body movements of a human demonstrator. The second one, `demonstration_recorder`, deals with the third set of requirements by implementing data collection from the Pepper robot. Finally, the `demonstration_gui` delivers an intuitive graphical user interface. Figure 2 provides the functional graph as the system overview.

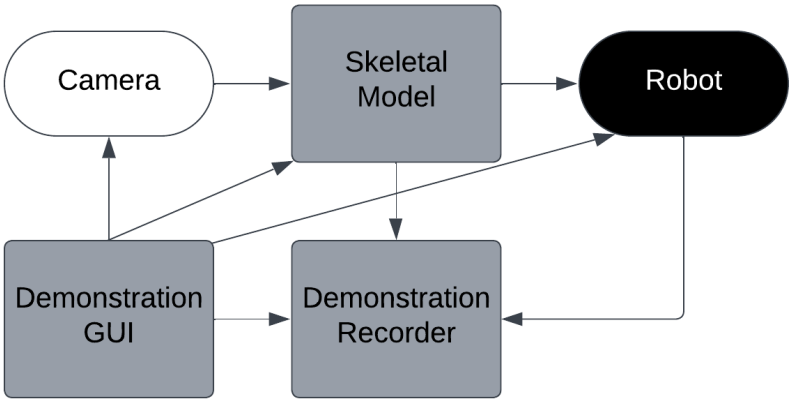

Figure 2: Functional graph of the PbD module: the Skeletal Model processes visual and depth images to generate joint commands for the robot; the Demonstration Recorder logs these commands and data from the robot sensors; the Demonstration GUI provides user control over all the other components. The camera can be replaced by a pre-recorded video feed.

The main computation is carried out in the `skeletal_model` component, as shown in Figure 3. By retargeting human joint positions to humanoid joint angles, the Pepper robot is endowed with the ability to imitate upper-body gestures performed by the human demonstrator. Occlusions of arm joints are not handled, so only movements in the coronal plane and in front of the body are imitated. As soon as one landmark on the human upper body is not detected in the frame, no new command will be sent, and Pepper's arm controllers will maintain the last commanded joint angles.

The input to this component is abstracted as visual and depth images, meaning the source incoming frames can be either a live image feed or a prerecorded video stream. This component also includes filtering of the retargeted joint angles to smooth noisy angle estimations from the skeletal model, enhancing the naturalness of Pepper's movements.

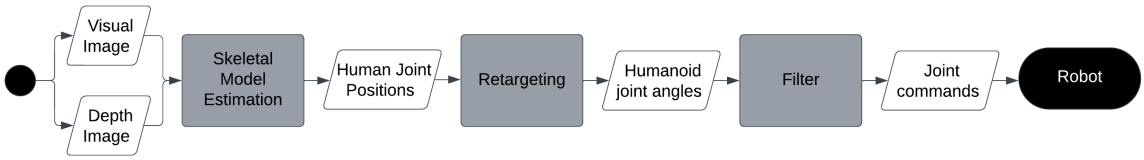

Figure 3: Computational graph of the PbD module: Synchronized visual and depth images from the camera are used to estimate the skeletal model, i.e. the positions of the human joints. These are then retargeted to the joint angles of the Pepper humanoid robot. A filter is applied to the incoming joint angles to increase the naturalness of the movement in real-time and the resulting joint commands are fed to the robot arm controllers.

The `demonstration_recorder` component can be seen as the backend of the application. It provides a robust recording function that saves data locally in a consistent, structured format. This recorded data can be replayed by the robot, as illustrated in Figure 4 and Figure 5. The component integrates with the GUI to handle user inputs and display system information, offering an intuitive interface for users.

Designed with a modular approach, the `demonstration_recorder` component can be customized for various robot platforms and user input devices. While the default user interface is the GUI provided by the `demonstration_gui` component, developers have the flexibility to integrate alternative or additional input methods (see Section 8.1 for details).

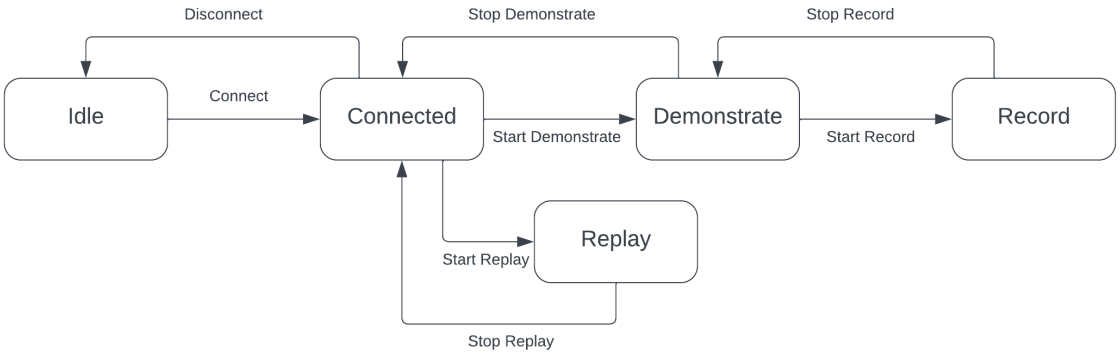

Figure 4: State machine graph of the PbD module: Barring internal errors that are displayed to the user in the terminal and the GUI, binary user input determines the state of the system: every active state has an entry and exit function that needs to be explicitly called. Exception is “Stop Replay”, which can be stopped manually or stops automatically when the recorded motion has been fully performed.

The GUI serves as the frontend for controlling the system's functionality. It allows users to initiate and manage recordings, configure the types of data to be recorded, and select filters to apply to the retargeted joint angles. Additionally, the GUI displays real-time system logs, providing valuable insights for troubleshooting and monitoring during operation. The user can manually clear the logs box at any point.

Figure 5 illustrates an example workflow for recording and replaying demonstrations.

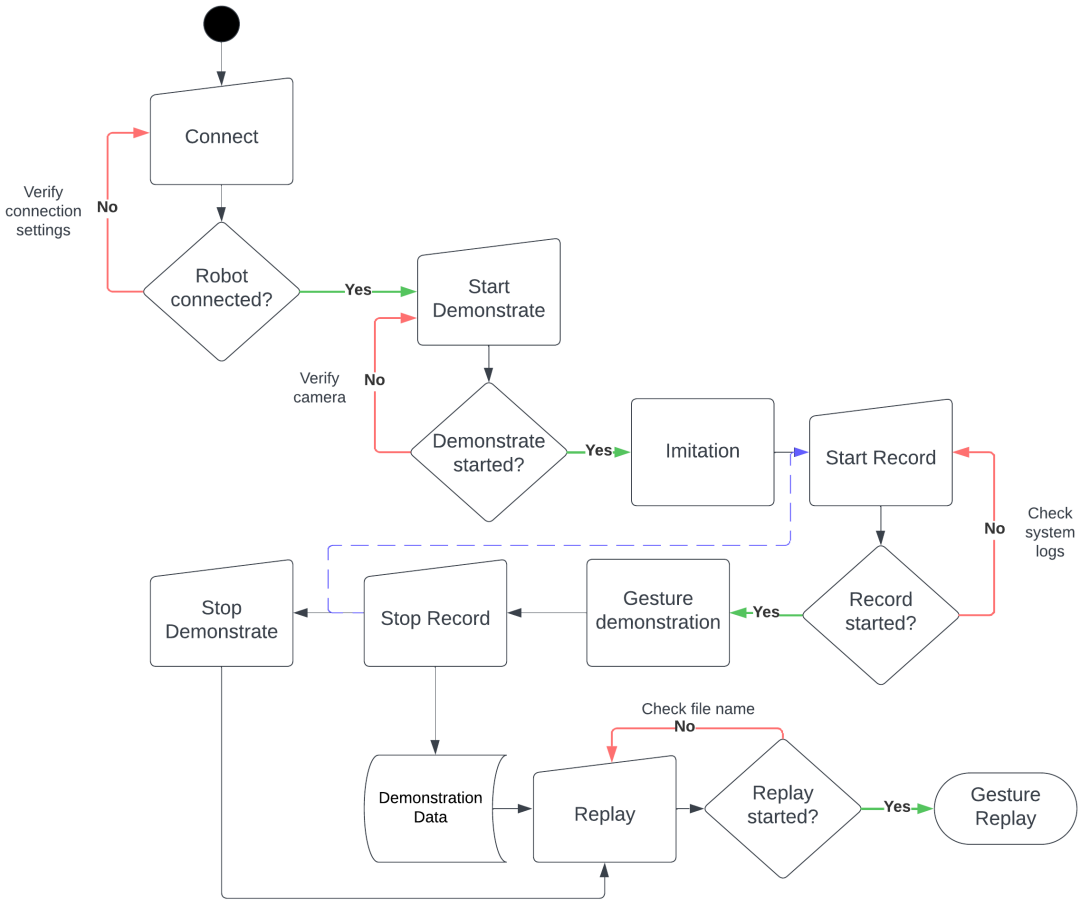

Figure 5: Behavioural graph of the PbD module: The user-centered graph shows an example procedure for demonstrating, recording and replaying. Rectangular boxes represent observable processes, diamond-shaped boxes stand for checks, non-rectangular boxes represent user action. "Gesture Replay" is the termination element. The blue line indicates the available option of recording multiple demonstrations in one session.

4 Module Design

4.1 Design Overview

The Programming by Demonstration module is implemented in the Python programming language and is designed to process visual and depth images for programming robot gestures by demonstration. Component-based robotic engineering [1] is employed using ROS Noetic as the middleware for interprocess communication [2]. The module consists of three components: `skeletal_model`, `demonstration_recorder`, and `demonstration_gui`. Each component consists of a ROS node. The terms *component* and *node* are used interchangeably throughout the following sections. Same applies to the terms *module* and *package*.

4.2 Image Source

The source of the visual and depth images is the ROS driver of an Intel RealSense RGBD camera [3] in the case of a live image feed or a custom driver in the case of video. The depth image is aligned with the color image. Both drivers are single-component external modules, `realsense2_camera` and `video_driver`,included as dependencies. Both are implemented as a ROS node, which publishes visual and aligned depth images on two separate topics at a fixed, configurable frequency. This approach stems from the implementation of the Intel RealSense ROS driver [3].

4.3 Skeletal Model

The core of the system, the `skeletal_model` node, incorporates the following design choices:

- Incoming visual and depth images are synchronized, ensuring consistent processing
- Synchronized inputs are fed to the MediaPipe deep learning model for human joint pose estimation in cartesian space(Figure 6)
- x and y coordinates are retrieved from MediaPipe and concatenated with the z coordinate from the depth map to generate 3D landmarks.
- 3D landmarks are retargeted to Pepper's joints using the GA-LVVJ algorithm [4].
- Calculated joint angles are filtered to improve the naturalness of movements. Butterworth and biological motion filters are already implemented and it is possible, though not necessary, to add more (see Section 8.4).
- Filtered joint angles are fed forward to Pepper's arm joint controllers for execution. No feedback control is implemented at this level.

Figure 6: Skeletal model of the human demonstrator is estimated using Google's MediaPipe pose estimation framework [5].

4.4 Demonstration Recorder

The `demonstration_recorder` component comprises several interconnected subcomponents:

- **Data Logger:** Implements functions to start and stop recording time-series data of the robot's joint angles during demonstrations.
- **Robot Event Handler:** Manages robot control actions such as connecting to the robot, starting, or stopping the demonstration process.
- **User Input Handler:** Processes inputs from the user interface to configure and control the demonstration. The default user interface is the GUI included with this module, but it can be replaced or extended with a user-defined interface for added flexibility (see Section 8.1).
- **Information Display:** Logs and displays system actions and events in real-time, providing operational feedback. By default, messages are displayed in the GUI.
- **Demo Recorder:** Coordinates the overall process by managing user inputs and delegating handling of incoming events to the Robot Event Handler or the Data Logger.

During operation, user commands are processed by the User Input Handler and shared as events with the Demo Recorder. Depending on the type of event, the Demo Recorder forwards the commands to either the Robot Event Handler or the Data Logger. The outcomes or responses from these subcomponents are then published by the Information Display.

This modular design inspired by [6] ensures the system is robot-and input-device-agnostic, allowing it to be adapted for use with any robotic platform and to record any type of data, making it flexible for various applications.

4.5 Demonstration GUI

The GUI for the demonstration process (Figure 7 is implemented using the third-party library PyQt5. Communication with the rest of the system is facilitated by a ROS node, which uses signals and slots [7] to forward messages to the main Qt application and publish GUI commands to the PbD system. The GUI contains the following elements:

- Entry fields to enter the robot IP and port to connect to the robot
- Buttons for capturing user commands (CONNECT, START/STOP DEMONSTRATE, START/STOP RECORD, CLEAR).
- A log box to display real-time system logs.
- Radio buttons for choosing the filter type
- Check boxes to select the data to be recorded, respectively.
- Integration of the MediaPipe feed with the skeletal model, providing a visual representation of the pose estimation process during demonstrations as shown in Figure 6.

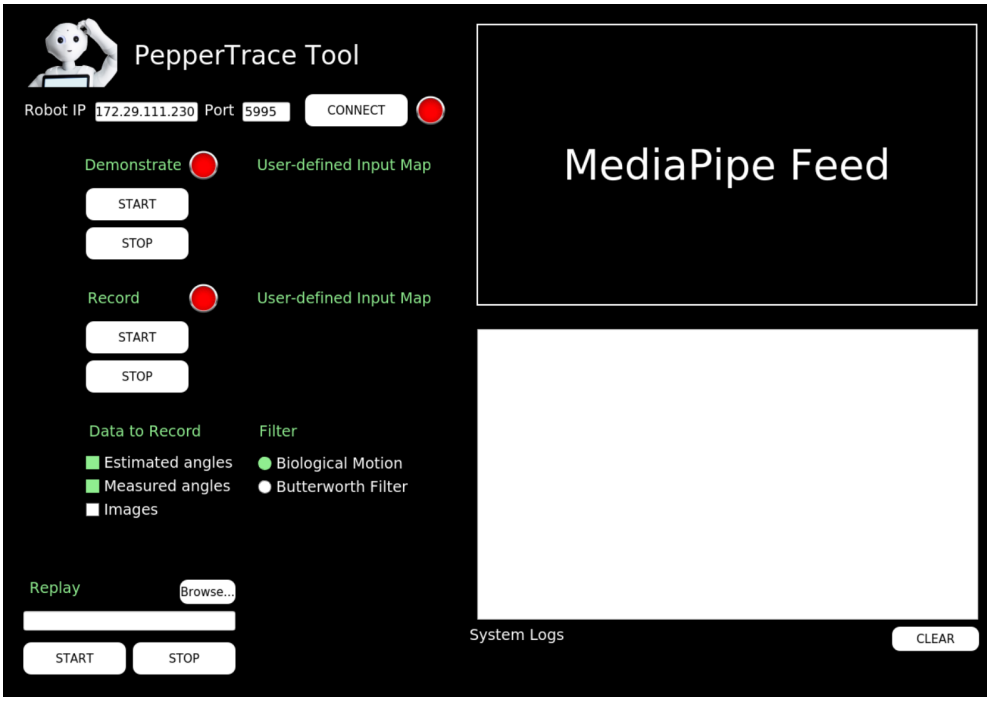

Figure 7: Graphical User Interface for the Programming by Demonstration system

5 Implementation

5.1 File Organization & Purpose

The source code for programming by demonstration is structured into three components: `skeletal_model`, `demonstration_recorder` and `demonstration_gui`. Each of these components operates as a ROS node, with distinct roles: retargeting estimated human joint positions to Pepper's joint angles, recording demonstration data, and providing a GUI as the user interface for the PbD system. Each component contains the following elements:

- **/config folder:** contains configuration files in the JSON format. These files store parameters and settings that can be adjusted to modify the behavior of the component without changing the source code.
- **/launch folder :** includes ROS launch files that automate the startup process of ROS nodes and set parameter values.
- **/src folder:** Source files are divided into one application and multiple implementation files. The rationale behind the nomenclature is that the user of a node does not need to know the implementation details to use it as an application. As a rule, each implementation file contains one Python class, according to the name of the file. The inclusion of an `__init.py__.py` file signals to Python that the directory should be treated as a package with importable scripts [8]
- **CMakeLists.txt file:** provides build instructions for the component using CMake, which is integrated with ROS through the catkin build system [9].
- **README.md file:** offers documentation and usage instructions for the component.

The `demonstration_gui` additionally contains an `images` folder, which can contain any number of image files, typically in the `*.png` format. These images are displayed in the GUI as logos or status identifiers.

Along with the component directories, the root directory of the module contains a package.xml file, which includes meta-information about the ROS package, e.g the package name, version, description, maintainers, license, and dependencies on other ROS packages. It also contains the top level CMakeLists.txt and README.md files for module-level build instructions and documentation, respectively.

Figure 8 shows the file structure of the `programming_by_demonstration` package.

```
programming_by_demonstration
├── demonstration_gui
│   ├── config
│   │   └── demonstration_gui_configuration.json
│   ├── images
│   │   └── *.png
│   ├── launch
│   │   └── demonstration_gui.launch
│   ├── src
│   │   ├── __init__.py
│   │   ├── demonstration_gui_application.py
│   │   ├── demonstration_gui_elements_implementation.py
│   │   ├── demonstration_gui_logic_implementation.py
│   │   └── demonstration_gui_ros_interface_implementation.py
│   ├── CMakeLists.txt
│   └── README.md
├── demonstration_recorder
│   ├── config
│   │   └── demonstration_recorder_configuration.json
│   ├── launch
│   │   └── demonstration_recorder.launch
│   ├── src
│   │   ├── __init__.py
│   │   ├── demonstration_recorder_application.py
│   │   ├── demonstration_recorder_data_logger_implementation.py
│   │   ├── demonstration_recorder_demo_recorder_implementation.py
│   │   ├── demonstration_recorder_information_display_implementation.py
│   │   ├── demonstration_recorder_robot_event_handler_implementation.py
│   │   ├── demonstration_recorder_recorder_ros_utils.py
│   │   ├── demonstration_recorder_rosbag_recorder_implementation.py
│   │   ├── demonstration_recorder_states_and_events_implementation.py
│   │   └── demonstration_recorder_user_input_handler_implementation.py
│   ├── CMakeLists.txt
│   └── README.md
├── skeletal_model
│   ├── config
│   │   └── skeletal_model_configuration.json
│   ├── launch
│   │   └── skeletal_model.launch
│   ├── src
│   │   ├── __init__.py
│   │   ├── skeletal_model_application.py
│   │   ├── skeletal_model_estimation_implementation.py
│   │   ├── skeletal_model_filters_implementation.py
│   │   └── skeletal_model_retargeting_implementation.py
│   ├── CMakeLists.txt
│   └── README.md
├── CMakeLists.txt
├── package.xml
└── README.md
```

Figure 8: Directory structure of the PbD module consisting of three components each with config, launch, src folders and CMakeLists.txt and README.md files. The root directory also includes the package.xml file.

5.2 Skeletal Model Implementation

Python Classes

The `skeletal_model` component is implemented with several classes that achieve the functionality depicted in Figure 3. The class diagram is depicted in Figure 9.

- **SkeletalModelEstimation - main implementation:**
  - uses synchronized ROS `message_filters` subscribers to receive visual and depth images
  - processes images in a callback: gets 3D landmarks in cartesian space in the camera frame from MediaPipe and depth map, runs retargeting and filtering functions and publishes the angles on a ROS topic. Numerical quantities like 3D landmarks and joint angles are processed as Numpy arrays [10].
  - the current image with skeletal model overlay as in Figure 6 is published on a ROS topic in the main node loop. A mutex lock prevents simultaneous access to the current image from image callback and main loop.
- **DataFilter - filter implementation:**
  - Uses a Numpy array as the data type for the filter window.
  - Saves points in the window and applies filter to it when it is full and a new point arrives. The specific filtering function that is called depends on the value of the filter_type attribute of type String. The class contains a setter method for this attribute.
- **HumanToHumanoidRetargeting - retargeting implementation:**
  - main function get_angles returns a Numpy array of Pepper joint angles from 3D landmarks on the human body
  - class contains helper functions for calculation of each joint angle for both arms

Configuration File

The configuration file is named `skeletal_model_configuration.json`. It contains a list of key-value pairs that determine the operation of the node.

Table 1: Configuration file for the skeletal model

Key	Value Type	Description
`camera_intrinsics`	`List of lists of floats`	Contains matrix with camera's intrinsic parameters
`image_width`	`String`	Specifies the width of color and depth used.
`image_height`	`String`	Specifies the width of color and depth images used.

Table 1: Configuration file for the skeletal model

Key	Value Type	Description
color_image_topic	String	Topic where color image feed is published.
depth_image_topic	String	Topic where aligned depth image feed is published.
left_arm_command_topic	String	Topic to publish control signal for left arm.
right_arm_command_topic	String	Topic to publish control signal for right arm.
gui_commands_topic	String	Topic to receive user commands to change system behavior (e.g. filter)
skeletal_model_feed_topic	String	Topic to publish images with skeletal model overlay.

Input File

During offline usage, a video file in the format `.bag` serves as input to the system. It should be placed in the `skeletal_model/data` folder and be called `video_input.bag`. It should contain depth information. The `src` folder contains an utility script to record a video with the Intel RealSense camera.

Output Data File

There is no output data file for the actuator test. The result of the angle calculation from the skeletal model is published on the skeletal model image feed and arm controller topics.

Topics Subscribed

Table 2: Topics subscribed by the skeletal_model node.

Topic	Message Type
/camera/color/image_raw	sensor_msgs/Image
/camera/aligned_depth_to_color/image_raw	sensor_msgs/Image
/gui/commands	std_msgs/String

Topics Published

Table 3: Topics published by the skeletal_model node.

Topic	Message Type
/mediapipe/image_feed	sensor_msgs/Image
/pepper_dcm/LeftArm_controller/command	trajectory_msgs/JointTrajectory
/pepper_dcm/RightArm_controller/command	trajectory_msgs/JointTrajectory

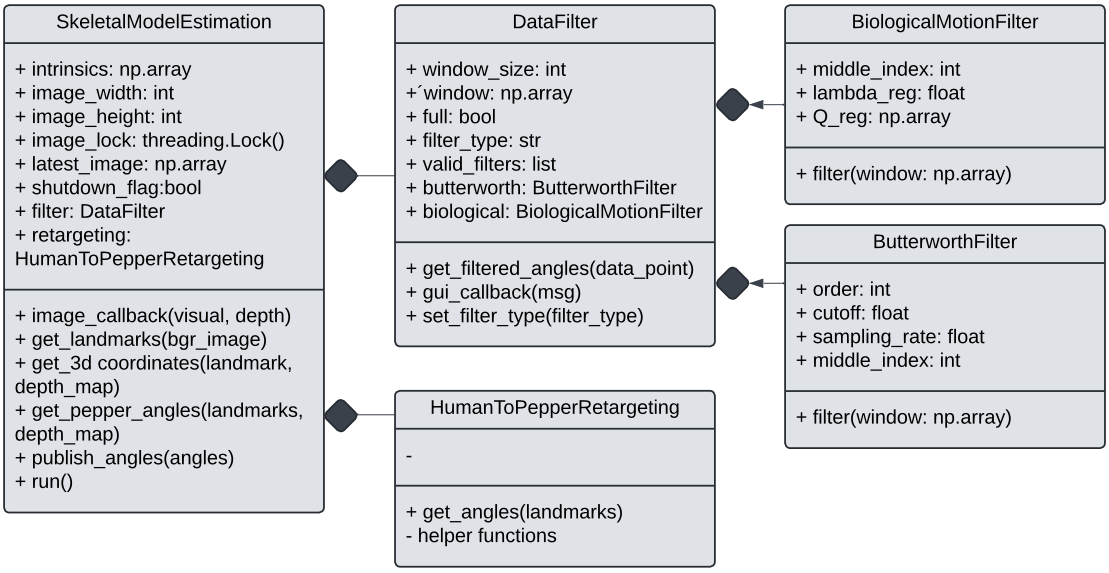

Figure 9: UML class diagram for the skeletal model component. Class visualization contains name, attributes and methods from top to bottom. Class relationships are of composition (filled diamond), as classes where symbol ends instantiate the others as attributes.

Launch File

The `skeletal_model.launch` launch file is used to launch the skeletal_model node. The required parameter is set to true. The same file also launches the ROS driver for the camera. In this implementation it is the ROS driver for the Intel RealSense camera with launch parameters:

- `align_depth`: whether to align depth image to color image (possible values: true, false ; default: true).
- `color_fps`: frequency that images are captured in frames-per-second (possible values: 6,15,30 ; default: 15).
- `color_width`: specifies width of the color image, depth image is aligned to this width (possible values: see Figure 10, default: 1280).
- `color_height`: specifies height of the color image, depth image is aligned to this height (possible values: see Figure 10, default: 720).

Format	Resolution	Frame Rate (FPS)	Comment
Z [16 bits]	1280 x 720	6, 15, 30	Depth
	848 x 480	6, 15, 30, 60, 90	
	640 x 480	6, 15, 30, 60, 90	
	640 x 360	6, 15, 30, 60, 90	
	480 x 270	6, 15, 30, 60, 90	
	424 x 240	6, 15, 30, 60, 90	
YUY2 [16 bits]	1920 x 1080	6, 15, 30	Color Stream from RGB camera (Camera D415 & D435/D435i/D435f/D435if)
	1280 x 720	6, 15, 30	
	960 x 540	6, 15, 30, 60	
	848 x 480	6, 15, 30, 60	
	640 x 480	6, 15, 30, 60	
	640 x 360	6, 15, 30, 60	
	424 x 240	6, 15, 30, 60	
	320 x 240	6, 30, 60	
	320 x 180	6, 30, 60	

Figure 10: Table with available image formats using USB 3.1 Gen 1 and the Intel RealSense D435 [11]

5.3 Demonstration Recorder Implementation

Python Classes

The `demonstration_recorder` component is implemented with several classes that achieve the functionality depicted in Figure 5. The class diagram is depicted in Figure 11.

- **Event, EventType, States, Commands - event implementation (Figure 12):**
  - The Event class defines an event with an event type, a command and optional arguments.
  - The EventType is defined as a Python Enum and specifies DATA_LOGGING, ROBOT_CONTROL and TERMINATE as possible types.
  - DataLoggerCommands, DataLoggerStates, RobotCommands and RobotStates are all Enums and define the available states and commands for the data logger and the robot event handler.
- **GuiInputHandler - user input implementation:**
  - Processes user input from the GUI buttons in a ROS subscriber callback by posting events to an event queue shared with the DemoRecorder.
  - It contains the input map object as an attribute that the user can define if adapting this class to another input device. The keys are "Demonstrate and "Record" and the values represent the corresponding inputs for Start and Stop. In the current implementation, these are the GUI buttons, but in a custom implementation they can refer to e.g. the buttons on a mouse, see Section 8.1.
- **PepperRobotEventHandler - robot interface implementation:**
  - Contains handle_event function that calls other class methods for connecting/disconnecting the robot and starting/stopping demonstrate mode.
  - Connection and demonstrate processes are started in subprocesses that execute ROS launch files.
  - A monitor function checks whether the demonstrate process exits and reports an exit directly to the user via the GUI by publishing a message on a ROS topic.
- **PepperROS1DataLogger - data recording implementation:**
  - Contains handle_event function that calls other class methods for starting/stopping data logging, starting/stopping replay and setting the data, and indirectly the ROS topics, that should be recorded.
  - Record and replay processes are started in subprocesses using `rosbag record` and `rosbag play` respectively.
  - A monitor function checks whether the replay process exits and sets the state of the class to IDLE on stopped replay.
  - The set_topics function receives types of data to record and maps it to the respective topics, setting the topic_list attribute that the record process uses to record the topics.

- **GuiInformationDisplay - information display implementation:**
  - This class uses a ROS publisher to publish information from the other systems to the GUI.
  - It also publishes the input map on startup, prompted by the DemoRecorder.
- **DemoRecorder - orchestrating implementation:**
  - Implements the main loop which keeps the component running and delegates incoming tasks from the user input handler to the data logger or the robot event handler.
  - The information returned by both event handling systems is forwarded to the information display for informing the user through the GUI.
  - At startup, it publishes the input map for display in the GUI and passes the event queue to the user input, which becomes shared.
  - A terminate event triggered by user input will stop the system. This can be included, see Section 8.1, but out of the box works by entering Ctrl-C in the terminal.

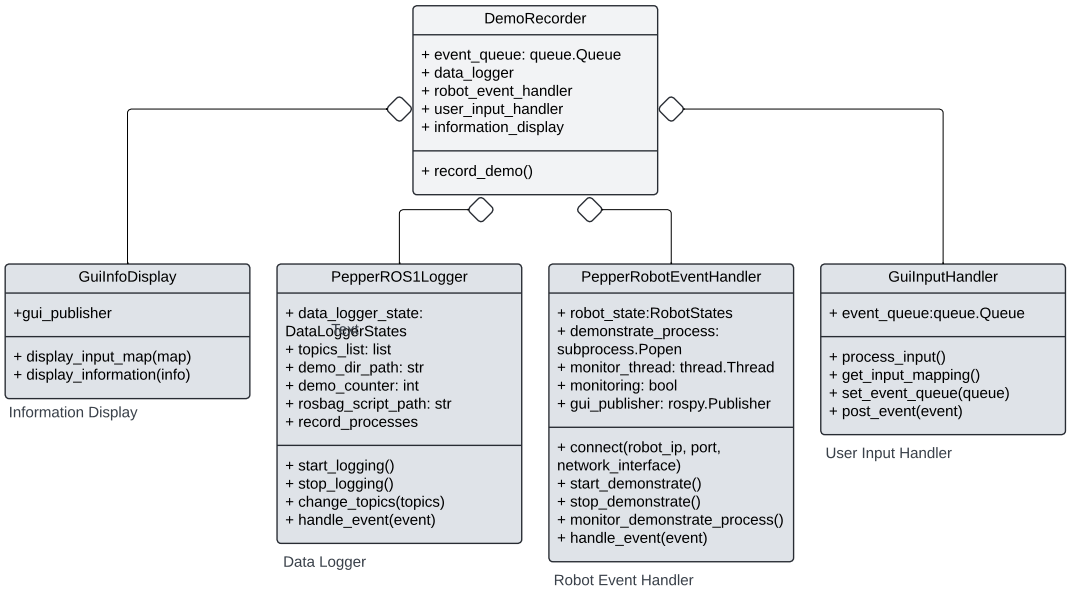

Figure 11: UML class diagram for the demonstration recorder component. Class visualization contains name, attributes and methods from top to bottom. Class instances are aggregated by the DemoRecorder (unfilled diamond)

Configuration File

The configuration file is named `demonstration_recorder_configuration.json`. It contains a list of key-value pairs that determine the operation of the node. The `.` here denote nested parameters in the json structure.

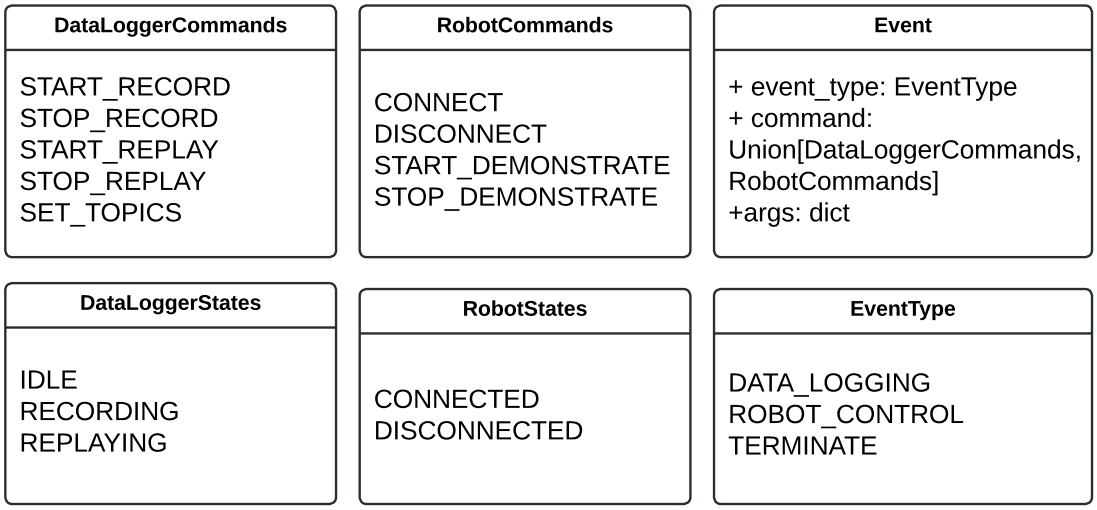

Figure 12: Event types, states and enums as Python Enums, Event as a Python class

Table 4: Configuration file for the demonstration recorder

Key	Value Type	Description
`data_logger.topics_list`	`List of strings`	Specifies the list of topics to be recorded during demonstration.
`data_logger.data_dir`	`String`	Specifies the root directory where the bag files will be stored.
`data_logger.demo_name`	`String`	Specifies the name of the demonstration and will be the directory inside data_dir where the bag files will be stored.
`gui_system_logs_topic`	`String`	Topic where messages to be displayed in the GUI are published.
`gui_commands_topic`	`String`	Topic to receive user commands to change behavior (e.g. start recording).
`skeletal_model_feed_topic`	`String`	Topic to publish images with skeletal model overlay.

Input File

There is no input file for the demonstration_recorder node. The demonstrations are recorded from messages that are sent on relevant topics in the ROS system.

Output Data File

Output .bag files comprise recorded demonstration data saved in `<data_dir>/<demo_name>` on the local machine.

Topics Subscribed

Table 5: Topics subscribed by the demonstration_recorder node.

Topic	Message Type
/gui/commands	std_msgs/String
/joint_states*	sensor_msgs/JointState
/pepper_dcm/LeftArm_controller/command*	trajectory_msgs/JointTrajectory
/pepper_dcm/RightArm_controller/command*	trajectory_msgs/JointTrajectory

*These topics are examples of topics that can be recorded by the data_logger with rosbag.

Topics Published

Table 6: Topics subscribed by the demonstration_recorder node.

Topic	Message Type
/gui/system_logs	std_msgs/String

Launch File

The demonstration_recorder.launch launch file is used to launch the demonstration recorder node. No launch parameters are required.

5.4 Demonstration GUI Implementation

Python Classes

The demonstration_gui component is implemented with several classes that achieve the design and functionality of the GUI (Figure 7). The class diagram is depicted in Figure 13.

- **Ui_MainWindow - visual elements implementation:**
  - setupUi function instantiates all elements that can be seen in the GUI such as labels, text boxes and buttons.
  - retranslateUi function sets the text for all the elements and translates them to their specified position in the main window.
  - This Python class is generated by PyQt5 from the .ui file generated with the Qt Designer application, where the GUI was put together.
- **RosThread and RosNode - ROS interface implementation:**
  - The Qt application needs to run in the main thread, so RosThread inherits from QThread to spin a ROS node in a secondary thread
  - The RosNode class contains this node which has two subscribers, one for system logs and another for the skeletal model image feed, and one publisher for system commands. It forwards incoming ROS messages to the MainWindow via PyQt signals and publishes commands from the MainWindow to a ROS topic

- **MainWindow - GUI logic implementation:**
  - This class instantiates both the others. Having access to the GUI elements from Ui_MainWindow, it connects the buttons to the class methods that enable the desired functionality, e.g. sending commands to the RosThread instance for publishing.
  - Incoming messages from the RosThread are processed by PyQt slots, which display the logs in the system logs box or the skeletal model image feed above it.
  - It also changes the color of status symbols on the GUI, i.e. the light icons, between green and red depending on log messages from the other components of the Programming by Demonstration system.

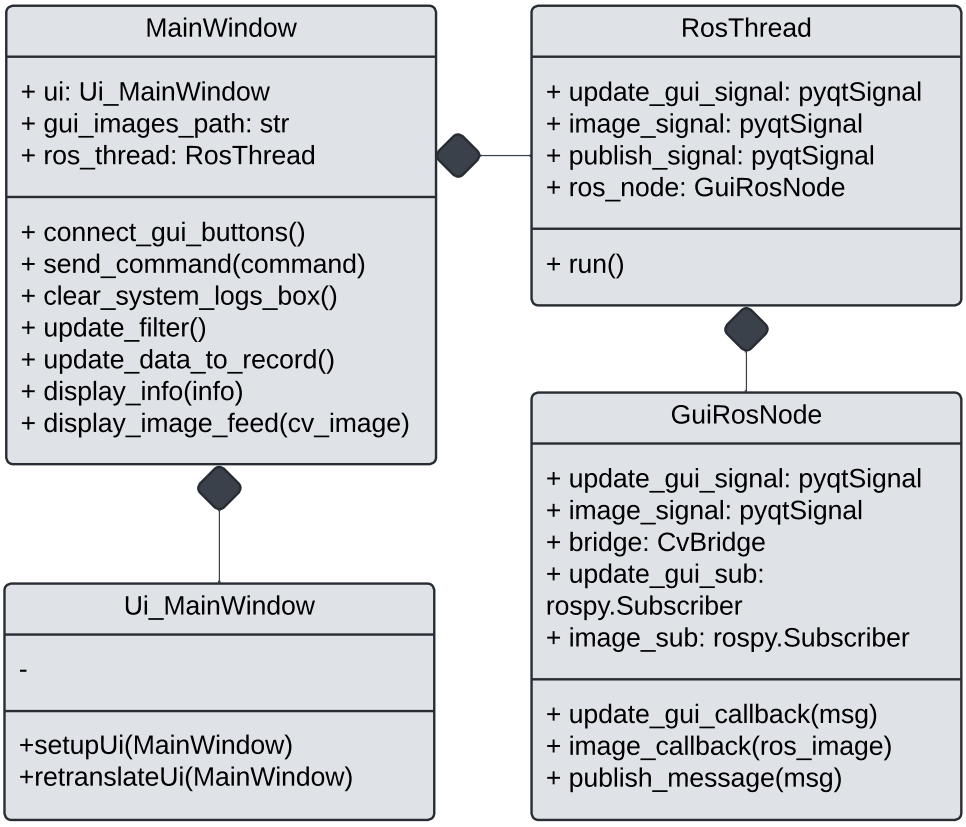

Figure 13: UML class diagram for the demonstration gui component. Class visualization contains name, attributes and methods from top to bottom. Class relationships are of composition (filled diamond), as classes where symbol ends instantiate the others as attributes.

Configuration File

The configuration file is named `demonstration_gui_configuration.json`. It contains a list of key-value pairs that determine the operation of the node. The `.` here denote nested parameters in the json structure.

Table 7: Configuration file for the demonstration GUI

Key	Value Type	Description
gui_system_logs_topic	String	Topic where messages to be displayed in the GUI are published.
gui_commands_topic	String	Topic to publish user commands to change behavior in other subsystems (e.g. filter in skeletal_model).
skeletal_model_feed_topic	String	Topic to publish images with skeletal model overlay.

Input File

There is no input file for the demonstration_gui node. It responds to user input which it passes on to the rest of the PbD system.

Output Data File

There is no output data file demonstration_gui node. System logs are flushed to the terminal and the GUI itself.

Topics Subscribed

Table 8: Topics subscribed by the demonstration_gui node.

Topic	Message Type
/gui/system_logs	std_msgs/String
/mediapipe/image_feed	sensor_msgs/Image

Topics Published

Table 9: Topics published by the demonstration_gui node.

Topic	Message Type
/gui/commands	std_msgs/String

Launch File

The demonstration_gui.launch launch file is used to launch the demonstration_gui node. No launch parameters are required.

6 Running the Programming by Demonstration System

Each component of the PbD system builds functionality on top of the other. This section will show how to run the components one by one and as a whole.

6.1 Run as a whole

To run all the components described in 6.2 at once, run:

```
# Launch entire programming_from_demonstration system
roslaunch programming_by_demonstration programming_by_demonstration.
   launch \
robot_ip:=<robot_ip> network_interface:=<network_interface> \
align_depth=<align_depth> color_fps:=<color_fps> \
color_width:=<color_width> color_height:=<color_height>
```

6.2 Run components one by one

The first step is to wake the robot with:

```
# Wake up the Pepper robot
roslaunch pepper_interface_tests actuatorTestLaunchRobot.launch \
robot_ip:=<robot_ip> network_interface:=<network_interface>
```

After waking the robot, the user can run this command to launch the skeletal_model node and the Intel RealSense ROS driver:

```
# Launch skeletal_model node and Intel RealSense ROS driver
roslaunch programming_by_demonstration skeletal_model.launch \
align_depth=<align_depth> color_fps:=<color_fps> \
color_width:=<color_width> color_height:=<color_height>
```

The user can now stand in front of the camera and the Pepper robot will imitate the demonstrated gestures.

To record demonstrations, the user needs to launch the demonstration_recorder node with:

```
# Launch demonstration_recorder node
roslaunch programming_from_demonstration demonstration_recorder.launch
```

By using the implemented input device or sending ROS messages to the `/gui/commands` topic, the user can start and stop recording demonstrations, which are saved as bag files.

Finally, do enable smoother operation with the use of a GUI, the user can activate it with:

```
# Launch demonstration_gui node
roslaunch programming_from_demonstration demonstration_gui.launch
```

This allows the user to connect to the robot, start/stop demonstrating, start/stop recording data and configuring the demonstration process all from the GUI.

7 Unit Tests

Unit tests with stub nodes are provided for each of the three components.

Skeletal Model Unit Test

To run the unit test:

```
# Launch unit test for skeletal_model
roslaunch programming_from_demonstration skeletal_model.launch unit_test
    :=true
```

- Launches the skeletal_model component and a stub ROS camera driver node with publishers on the color and depth image topics
- Driver publishes one visual and depth image pair from a provided image pair (publishes multiple times to check if filter works after window of 3 is filled)
- skeletal_model component calculates retargeted Pepper angles and logs them in the terminal. The output should look like this (check for similar values):

```
 # Launch unit test for skeletal_model
 [INFO] [1733825149.548106]: Publishing images...
 [INFO] [1733825149.553157]: Images published successfully.
 [INFO] [1733825149.765129]: Received image at timestamp:
1733825149547950267
 [INFO] [1733825149.771075]: Received depth at timestamp:
1733825149547950267
 {'LShoulderPitch': 1.9798229543652721, 'LShoulderRoll':
1.2625669566199917, 'LElbowRoll': -0.2965326466083589, 'LElbowYaw
': -2.7877042812891686, 'LWristYaw': 0, 'RShoulderPitch':
1.9550295042596533, 'RShoulderRoll': -1.0647517275688356, '
RElbowRoll': 0.3358612598878903, 'RElbowYaw': -2.537359649353141,
 'RWristYaw': 0}
 [WARN] [1733825150.317308]: Filter not yet ready: insufficient
data in window
 [INFO] [1733825150.555286]: Publishing images...
 [INFO] [1733825150.558529]: Images published successfully.
 [INFO] [1733825150.562605]: Received image at timestamp:
1733825150555179834
 [INFO] [1733825150.563171]: Received depth at timestamp:
1733825150555179834
 {'LShoulderPitch': 1.9745303521213002, 'LShoulderRoll':
1.2551816026093596, 'LElbowRoll': -0.30925236016320845, '
LElbowYaw': -2.809200505750682, 'LWristYaw': 0, 'RShoulderPitch':
 1.9574034886210552, 'RShoulderRoll': -1.0656933454976425, '
RElbowRoll': 0.3358612598878903, 'RElbowYaw': -2.537359649353141,
 'RWristYaw': 0}
 [WARN] [1733825150.583938]: Filter not yet ready: insufficient
data in window
 [INFO] [1733825151.560466]: Publishing images...
 [INFO] [1733825151.562860]: Images published successfully.
```

```
 [INFO] [1733825151.565321]: Received image at timestamp:
1733825151560360193
 [INFO] [1733825151.565907]: Received depth at timestamp:
1733825151560360193
 {'LShoulderPitch': 1.941745668296518, 'LShoulderRoll':
1.2477372386287904, 'LElbowRoll': -0.30491699983987486, '
LElbowYaw': -2.803759898660347, 'LWristYaw': 0, 'RShoulderPitch':
 1.9526899036771832, 'RShoulderRoll': -1.079199719003931, '
RElbowRoll': 0.33249978758958854, 'RElbowYaw':
-2.528508713797143, 'RWristYaw': 0}
 [INFO] [1733825152.564869]: Published single pair of images 3
times. Shutting down.
```

Demonstration Recorder Unit Test

To run the unit test:

```
# Launch unit test for demonstration_recorder
roslaunch programming_from_demonstration demonstration_recorder.launch
    unit_test:=true
```

- Launches the demonstration_recorder component and a stub ROS node for system control with publishers on the `/gui_commands` and `/unit_test` topics
- Stub node publishes "RECORD[/demonstration_recorder/unit_test]" and then "START_RECORD" to the `/gui_commands` topic .
- Stub node publishes three messages to the `/unit_test` topic
- Stub node publishes "STOP_RECORD" and then "START_REPLAY,/root/workspace/demo_data/unit_test/demo1_demonstration_recorder_unit_test.bag" on the `/gui_commands` topic
- The demonstration_recorder component services these requests, essentially running through the behavioral graph in Figure 5
- The expected output is:

```
 # Launch unit test for skeletal_model
 [INFO] [1733820757.971987]: System ready to start recording
 [INFO] [1733820759.912621]: Published: RECORD['/
demonstration_recorder/unit_test']
 [INFO] [1733820759.913496]: Received GUI Command: RECORD['
unit_test']
 [INFO] [1733820759.916777]: Event posted. Type: EventType.
DATA_LOGGING, Command: DataLoggerCommands.SET_TOPICS
 [INFO] [1733820759.917311]: Received event: DataLoggerCommands.
SET_TOPICS
 [INFO] [1733820759.920464]: Set topics to record: ['/
demonstration_recorder/unit_test']
 [INFO] [1733820759.922253]: Displaying: "[Data Logger] Set data
to record: unit_test "
```

```
 [INFO] [1733820759.923024]: [Data Logger] Set data to record:
unit_test
 [INFO] [1733820760.917509]: Published: START_RECORD
 [INFO] [1733820760.918510]: Received GUI Command: START_RECORD
 [INFO] [1733820760.921210]: Event posted. Type: EventType.
DATA_LOGGING, Command: DataLoggerCommands.START_RECORD
 [INFO] [1733820760.921842]: Received event: DataLoggerCommands.
START_RECORD
 [INFO] [1733820760.928813]: Started recording /
demonstration_recorder/unit_test in /root/workspace/demo_data/
unit_test/demo_1_demonstration_recorder_unit_test.bag
 [INFO] [1733820760.930624]: Started recording all specified
topics
 [INFO] [1733820760.931870]: Displaying: "[Data Logger] Data
logger is recording now."
 [INFO] [1733820760.932215]: [Data Logger] Data logger is
recording now.
 [INFO] [1733820761.219923]: Subscribed to /demonstration_recorder
/unit_test with queue size 1000
 [INFO] [1733820761.921558]: Published: Unit Test Message 1
 [INFO] [1733820761.922750]: Recording message on topic: /
demonstration_recorder/unit_test
 [INFO] [1733820762.931992]: Published: Unit Test Message 2
 [INFO] [1733820762.933116]: Recording message on topic: /
demonstration_recorder/unit_test
 [INFO] [1733820763.936018]: Published: Unit Test Message 3
 [INFO] [1733820763.937031]: Recording message on topic: /
demonstration_recorder/unit_test
 [INFO] [1733820764.940185]: Published: STOP_RECORD
 [INFO] [1733820764.941267]: Received GUI Command: STOP_RECORD
 [INFO] [1733820764.943920]: Event posted. Type: EventType.
DATA_LOGGING, Command: DataLoggerCommands.STOP_RECORD
 [INFO] [1733820764.944848]: Received event: DataLoggerCommands.
STOP_RECORD
 [INFO] [1733820764.948580]: Stopping the recording process...
 [INFO] [1733820765.569578]: Stopped recording successfully
 [INFO] [1733820765.570798]: Displaying: "[Data Logger] Data
logger is stopped"
 [INFO] [1733820765.571157]: [Data Logger] Data logger is stopped
 [INFO] [1733820766.945135]: Published: START_REPLAY,/root/
workspace/demo_data/unit_test/
demo_1_demonstration_recorder_unit_test.bag
 [INFO] [1733820766.946170]: Received GUI Command: START_REPLAY,/
root/workspace/demo_data/unit_test/
demo_1_demonstration_recorder_unit_test.bag
 [INFO] [1733820766.956333]: Event posted. Type: EventType.
DATA_LOGGING, Command: DataLoggerCommands.START_REPLAY
 [INFO] [1733820766.956714]: Received event: DataLoggerCommands.
START_REPLAY
 [INFO] [1733820766.959354]: Starting replay of /root/workspace/
demo_data/unit_test/demo_1_demonstration_recorder_unit_test.bag
 [INFO] [1733820766.965006]: Displaying: "[Data Logger] Started
replaying recording: /root/workspace/demo_data/unit_test/
```

```
demo_1_demonstration_recorder_unit_test.bag"
 [INFO] [1733820766.965712]: [Data Logger] Started replaying
recording: /root/workspace/demo_data/unit_test/
demo_1_demonstration_recorder_unit_test.bag
 [ INFO] [1733820767.163282353]: Opening /root/workspace/demo_data
/unit_test/demo_1_demonstration_recorder_unit_test.bag

 Waiting 0.2 seconds after advertising topics... done.

 Hit space to toggle paused, or 's' to step.
  [RUNNING]  Bag Time: 1733820763.837444   Duration: 1.906034 /
2.008535
 Done.
 [INFO] [1733820769.966203]: Replay process completed.
 [INFO] [1733820770.959234]: Unit test sequence completed.
 [INFO] [1733820770.962189]: Terminating /
demonstration_recorder_node...
 killing /demonstration_recorder_node
 shutdown request: user request
 killed
 [INFO] [1733820771.336907]: demonstration_recorder_stub finished.
```

Demonstration GUI Test

To run the unit test:

```
# Launch unit test for demonstration_gui
roslaunch programming_from_demonstration demonstration_gui.launch
   unit_test:=true
```

- Launches the demonstration_gui component
- Button presses are simulated in sequence
- Stub script simulates Button presses
- Logs from button presses are captured in the terminal with expected output:

```
 # Launch unit test for skeletal_model
 [INFO] [1733821693.462117]: Sub to: /gui/system_logs
 [INFO] [1733821693.475778]: Performing Unit Test: True
 [INFO] [1733821694.478073]: Running unit tests...
 [INFO] [1733821694.481028]: Testing CONNECT button...
 [INFO] [1733821695.483899]: Testing DISCONNECT button...
 [INFO] [1733821696.487381]: Testing START_DEMONSTRATE button...
 [INFO] [1733821696.491195]: Publishing on /gui/commands: CONNECT
,172.29.111.230,5995
 [INFO] [1733821697.493879]: Testing STOP_DEMONSTRATE button...
 [INFO] [1733821697.497546]: Publishing on /gui/commands:
START_DEMONSTRATE
 [INFO] [1733821698.500063]: Testing START_RECORD button...
```

```
 [INFO] [1733821698.503252]: Publishing on /gui/commands:
STOP_DEMONSTRATE
 [INFO] [1733821699.505606]: Testing STOP_RECORD button...
 [INFO] [1733821700.510478]: Testing START_REPLAY button...
 [INFO] [1733821700.516749]: Publishing on /gui/commands:
STOP_RECORD
 [INFO] [1733821701.519234]: Testing STOP_REPLAY button...
 [INFO] [1733821702.523401]: Testing BROWSE button...
 [INFO] [1733821702.530924]: Publishing on /gui/commands:
STOP_REPLAY
 [INFO] [1733821703.532863]: Testing CLEAR button...
 [INFO] [1733821719.140138]: All tests completed.
```

- Additionally, in the GUI, a FileDialog window should open and the system logs box should be cleared before the end of the test

8 For Developers

There are several ways to adapt and extend the system to a custom application, robot or input device. This section provides directions to developers who want to make these changes to the tool.

8.1 Use another input device

The GUI's buttons can be used to control the demonstration process. Other inputs such as a dedicated mouse or keyboard keys can be more intuitive and practical during demonstrations. Here are the necessary steps to add a new input device. As the communication is done via ROS, there is no limit to the amount of input devices that can be used, although simplicity is advised.

- Implement a custom User Input Handler class for your input device with the same methods as the implementation in this module
- Poll the input device in its own thread
- Replace the implementation of the User Input Handler in `demonstration_recorder_application.py` with your own or add it to the list of inputs

8.2 Demonstrate for another robot

The current implementation is focused on Pepper. However the modular approach allows developers to retarget human joint positions to the joints of other humanoid robot by following these steps:

- Implement a custom Robot Event Handler class for your robot with the same methods as the implementation in this module
- Implement a custom Keypoints to Angles class for retargeting human joint positions to the kinematics of your robot
- Replace the implementation of the Robot Event Handler in `demonstration_recorder_application.py` and the implementation of Keypoints to Angles in `skeletal_model_implementation.py` with your own
- If necessary, change the topic names in the configuration files to suit the topic names from your robot

8.3 Record more data

Data collection is done with `rosbag`. Each recorded topic is saved to a different `.bag` file. To change the topics being recorded during demonstrations you can:

- If not existent, add a ROS node publishing information you are interested in to a ROS topic
- Change the `topics_list` to include all topics you want data from

8.4 Change filter

Filters are implemented as separate classes for modularity and included in the DataFilter class for integration.

- Go to the `skeletal_model_filters_implementation.py` file
- Implement your filter class similar to the the existing filter implementations
- include your filter in the DataFilter class alongside the others, making sure it is in the `valid_filters` list, instantiated as an attribute and its filter method called in the `get_filtered_angles` function

References

[1] D. Brugali and A. Shakhimardanov. Component-based robotic engineering (Part II). *IEEE Robotics & Automation Magazine*, 17(1):100–112, 2010.

[2] M. Quigley, K. Conley, B.P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A.Y. Ng. ROS: an open-source robot operating system. In *ICRA Workshop on Open Source Software*, 2009.

[3] Intel Corporation. ROS wrapper for intel® realsense™ devices. https://github.com/IntelRealSense/realsense-ros/tree/ros1-legacy, 2024. Accessed: 2024-12-02.

[4] Z. Zhang, Y. Niu, Z. Yan, and S. Lin. Real-time whole-body imitation by humanoid robots and task-oriented teleoperation using an analytical mapping method and quantitative evaluation. *Applied Sciences*, 8(10):2005, 2018.

[5] C. Lugaresi, J. Tang, H. Nash, C. McClanahan, E. Uboweja, M. Hays, F. Zhang, C. Chang, M. Yong, J. Lee, et al. Mediapipe: A framework for building perception pipelines. *arXiv preprint arXiv:1906.08172*, 2019.

[6] D. Barros, C. Willibald, A. Padalkar, and J. Silvério. Enhancing probabilistic imitation learning with robotic perception for self-organising robotic workstation. In *CoRL 2024 Workshop on Mastering Robot Manipulation in a World of Abundant Data*, 2024.

[7] The Qt Company. Signals and slots in qt for python. https://doc.qt.io/qtforpython-6/overviews/signalsandslots.html, 2024. Accessed: 2024-12-02.

[8] Better Stack Community. What is __init__.py for? https://betterstack.com/community/questions/what-is-init-py-for/, 2024. Accessed: 2024-12-05.

[9] Open Source Robotics Foundation (OSRF). Catkin Documentation. https://wiki.ros.org/catkin, 2024. Accessed: 2024-12-05.

[10] C. Harris, J. Millman, S. Van Der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. Smith, et al. Array programming with numpy. *Nature*, 585(7825):357–362, 2020.

[11] Intel Corporation. Intel® realsense™ depth camera d435 datasheet. https://www.intelrealsense.com/download/21345/?tmstv=1697035582, 2018. Accessed: 2024-12-02.

Principal Contributors

The main authors of this deliverable are as follows (in alphabetical order).

Daniel Barros, Carnegie Mellon University Africa.
David Vernon, Carnegie Mellon University Africa.

Document History

Version 1.0
First draft.
Daniel Barros.
14 December 2024.

Version 1.1
Formatting fixes.
Daniel Barros.
16 December 2024.