CSSR for

Culturally Sensitive Social Robotics for Africa

D5.5.1.1 Gesture Execution

Due date: **30/09/2024**
Submission Date: **10/10/2024**
Revision Date: **04/02/2025**

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

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

Responsible Person: **Adedayo Akinade** Revision: **1.2**

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 D5.5.1.1 is focused on developing a comprehensive software module for Gesture Execution, enabling the Pepper robot to perform a range of body and hand gestures. This module encompasses five distinct gesture types: deictic, symbolic, and iconic hand gestures, as well as bowing and nodding body movements. Deictic gestures refer to pointing to specific things in the environment and is important for establishing joint attention [1]. The development process involves approaches to gesture specification, utilising joint space representations for most gestures and Cartesian space for deictic movements. A key feature of this module is its ability to learn gestures through manual teleoperation or human demonstration, employing RGB-D camera technology to map human skeletal movements onto the robot's joint system. This latter functionality is documented in Deliverable D5.5.1.2 Programming by Demonstration.

The deliverable outlines a software development methodology, including requirements definition, module specification, interface design, module design, coding, and unit testing. Each phase of this process is documented, as outlined in the deliverable. The module integrates with the robot localization system developed in Task 4.2.4, ensuring gesture execution within the robot's environment. This integration is crucial for deictic gestures, where precise pointing in the world frame of reference is essential. Additionally, this module coordinates with the attention subsystem developed in Task 5.3 for controlling the head during diectic gestures to a location. The interface design covers input parameters, output gestures, and control data, specifying appropriate data structures for each gesture type. All coding activities adhere to established software engineering standards as set out in Deliverable D3.2 Software Engineering Standards Manual, ensuring high-quality, maintainable code.

Contents

1 Introduction

This document describes the development and implementation of a ROS node for the execution of gestures on the Pepper robot. The gestures include deictic, iconic, symbolic, bow, and nod gestures. The actuators embedded in Pepper's head enable turn and nod motions, thereby fostering engaging interactions through nuanced head movements. The arm and hand actuators give Pepper the capability to mimic human gestures, enhancing its nonverbal communicative and interactive potential. The actuators in the hands further permit the opening and closing motions. Such functionality isneeded for projects emphasizing non-verbal communication through various gestures and movements.

Moreover, the inclusion of actuators in the torso and hips extends Pepper's mobility, allowing it to perform bends and twists. This flexibility is crucial for adapting Pepper's movements to reflect various cultural norms of body language, thereby enabling a culturally sensitive interaction.

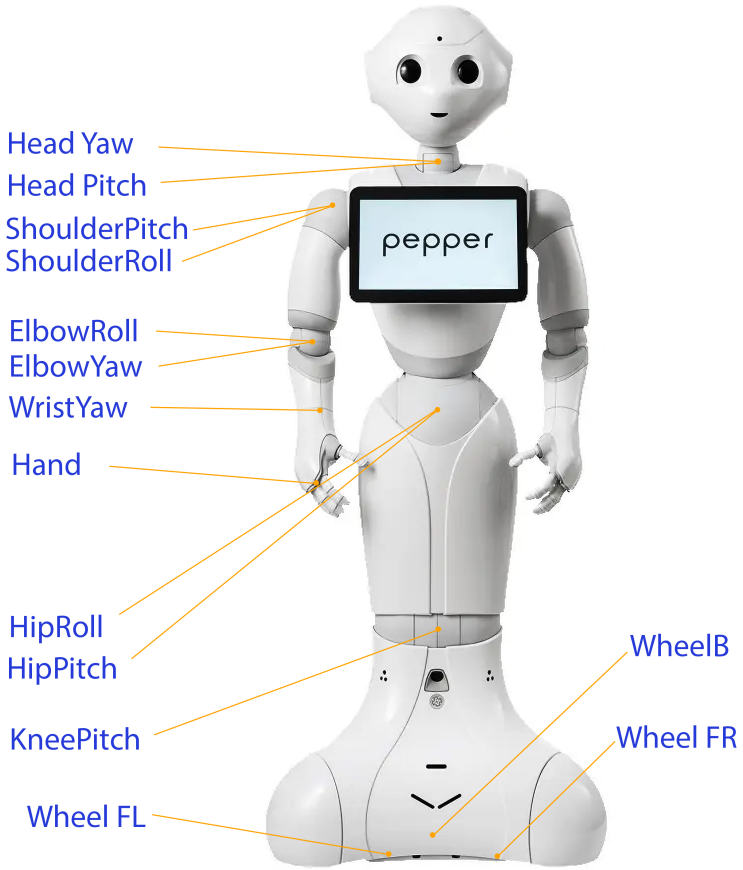

Figure 1: Pepper robot actuators

This deliverable presents a report that details each phase of the software development lifecycle for our gesture execution system. Section 2 describes the requirements definition process, where functional necessities are aligned with the project's overarching goals. This foundational section establishes the framework upon which subsequent development efforts are built.

Section 3 documents the module specifications, providing details on the execution of gestures. Interface design is then addressed, with particular attention given to data exchange mechanisms utilizing ROS middleware and file input/output operations.

The operational framework of the module is described in Section 4, with a focus on the role of the `gestureExecutionConfiguration.ini` configuration file in governing system behaviour. This section also outlines the structured approach implemented for handling and processing message data, ensuring efficient and reliable communication within the system.

Section 5 presents the implemented program code, adhering to the coding standards specified in Deliverable D3.2. This section also provides insights into key algorithms and data structures employed.

2 Requirements Definition

The gesture execution provides the robot with the ability to execute five forms of gesture: deictic, symbolic, and iconic non-verbal hand gestures, and bowing and nodding body gestures. This deliverable is important in identifying the specific user expectations, ensuring that the node is capable of executing the different gestures in various scenarios, including different operations and environments (physical robot and simulator).

As seen from the diagram in Figure 1, the Pepper robot has 20 joint actuators and 3 wheel actuators. The module must be capable of actuating the arm joints to point at a location in the world (for deictic and iconic gestures), actuating the leg joints (for bow gestures), and actuating the head joints (for nod gestures). If the arm cannot achieve the required pose for a deictic gesture, the robot rotates to make the pose achievable, returning to the original orientation once the gesture is complete. Thus, the module must be capable of actuating the wheels. Furthermore, the arm returns to a neutral position by the robot's side when the gesture is complete.

The module must be capable of reading a set of effector waypoints from a file for an iconic gesture and actuating through the points. If an iconic or symbolic gesture involves two arms, they are treated as a composite of two individual gestures, one for each arm.

The pointing location with respect to the robot body, specified by the shoulder pitch and shoulder roll angles, must be computed from the pointing location in the world frame of reference (provided through a service request by a client) and the pose of the robot in the world frame of reference (acquired by subscribing to a topic published by the `robotLocalization` node). The module must be capable of actuating the joints to achieve the target joint angles, interpolating linearly, or adjusting the joint angles, joint angular velocities, and joint accelerations to mimic biological movement by using a minimum jerk model of biological motion.

The module must be able to run in normal mode or verbose mode. In verbose mode, data that is published to topics is also printed to the terminal.

3 Module Specification

The specifications for these gestures are in joint space, except for deictic gestures which are in Cartesian space. Some gestures, e.g., iconic and symbolic hand gestures, are specified by learning the required motions either by manual teleoperation, recording the joint angles, or by demonstration, using an RGB-D depth camera to determine the joint angles of human gestures in a skeletal model and mapping these to the robot joints. Deliverable D5.5.1.2 Programming by Demonstration documents the process of learning this gestures by demonstration. Other gestures, i.e., deictic hand gestures and body gestures, are specified by gesture parameters, such as the pointing location for deictic gestures and the degree of inclination for bowing and nodding, and the joint angles are computed using the kinematic model of the robot head, torso, and arms. For deictic gestures, which require the robot to point at objects in its environment, the pose of the robot in the world frame of reference is used to compute the joint angles while the attention subsystem (documented in D5.3 Attention Subsystem) is utilized to control the robot's head to direct its gaze to the location.

Iconic and symbolic gestures are defined by descriptors that specify the final gesture joint configuration and how that configuration is achieved. Descriptors comprise four elements. Each element is a key-value pair, where the value can be an identifier, a number, a vector of numbers, or a vector of a vector of numbers.

The first key-value pair specifies the gesture type (e.g., `type iconic`, `type symbolic`).

The second key-value pair identifies the ID number (e.g., `ID 01`).

The third element defines the number of waypoints in the trajectory, including the start gesture joint configuration and the final gesture joint configuration.

The fourth element is a vector of joint angles vectors. The number of joint angle vectors is equal to the number of waypoints, including the start joint configuration and the final gesture configuration. Body gestures have three joints: knee pitch, hip pitch, hip roll. Iconic and symbolic gestures have five joints: shoulder pitch, shoulder roll, elbow yaw, elbow roll, and wrist yaw. Before beginning the gesture, the arm is moved from its current joint configuration to the start joint configuration, i.e., the joint angles specified in the first vector in the vector of vector of joint angles.

The number of elements in the vector of joint angles is determined by the gesture type.

Descriptors for each gesture are stored in an external descriptor file.

The joint angles for bow and nod body gestures, as well as hand deictic gestures, are computed at run time using the kinematic model of the robot and the bow angle, nod angle, or the location in the environment to which the robot should point. The bow angle, nod angle, and pointing location are provided as input to the module, along with the time in milliseconds that should elapse between the start of the gesture and the end of the gesture.

The pointing location with respect to the robot body, specified by the shoulder pitch and shoulder roll angles, is computed from the pointing location in the world frame of reference (and supplied as an input to the module) and the pose of the robot in the world frame of reference (provided by the `robotLocalization` node. The direction of the robot's gaze is achieved by a ROS service call to the `overtAttention` node. No waypoints are required for deictic gestures; the joints are actuated to achieve the target joint angles, interpolating linearly, or adjusting the joint angles, joint angular velocities, and joint accelerations to mimic biological movement by using a minimum jerk model of biological motion.

The knee pitch angle is fixed during a bow body gesture and the bow angle corresponds to the change in the hip pitch angle with respect to the default hip pitch angle. Similarly, the nod angle is the change in the head pitch angle with respect to the default head pitch angle. Finally, the arm and fingers are straight in a deictic gesture, with fixed values of elbow yaw, elbow roll, wrist yaw, and

hand angles, so that the palm of the hand is directed upwards, the angles being derived from the pose of the robot with respect to the location to which the robot is gesturing.

The input to the module, as a service request from a client, is a record comprising the gesture type (e.g., `iconic`, `symbolic`, `deictic`, `bow`, `nod`), the gesture ID for symbolic or iconic gestures (e.g., `01`), the duration of the gesture in milliseconds, and either a bow angle in degrees (for a bow body gesture), or a nod angle in degrees (for a nod body gesture), or the three-dimensional coordinates of a pointing location (for a deictic gesture). For deictic gestures, the module also inputs the current robot pose from the `robotLocalization` node.

The output of the module is a sequence of joint angles, joint angular velocities, and, optionally, joint angular accelerations. This output information is compiled into trajectory information, which is sent to the action server created for the appropriate topics, as contained in the topics file for the robot `robotTopics.dat` and the topics file for the simulator `simulatorTopics.dat`. The names of the topics to be used for each actuator is read from this data file comprising a sequence of key-value pairs. The key is the name of the actuator. The value is the topic name. There are two data files, one for the physical robot and another for the simulator. Additionally, the output from the module is a ROS service call to the `overtAttention` node in the case of a diectic gesture.

The module can run in normal mode or verbose mode. In verbose mode, data that is published to topics are also printed to the terminal.

4 Module Design

4.1 Deictic Gestures

Upon receipt of a service request for a location to point to, the module packages a service request to the `overtAttention/set_mode` service to direct the gaze to the location, after which the module employs inverse kinematics to calculate the requisite joint angles for Pepper's arm, enabling precise pointing towards the specified coordinates. This calculation is followed by a validation process to ensure the computed angles fall within the robot's operational limits. The function `void get_arm_angles(int arm, double elbow_x, double elbow_y, double elbow_z, double wrist_x, double wrist_y, double wrist_z, double* shoulder_pitch, double* shoulder_roll, double* elbow_yaw, double* elbow_roll)` takes in the 3D coordinates of the elbow of the robot, which is interpolated based on the lengths of the robot arm, and updates the joint angles for the shoulder and elbow.

A notable feature of this implementation is the optional incorporation of a biological motion model described by [2]. When activated, this model generates trajectories that emulate human-like movements, enhancing the naturalness of the robot's gestures. This feature contributes to the effectiveness of non-verbal communication between the robot and human observers. The function `void compute_trajectory(std::vector<double> start_position, std::vector<double> end_position, int number_of_joints, double trajectory_duration, std::vector<std::vector<double>>& positions, std::vector<std::vector<double>>& velocities, std::vector<std::vector<double>>& accelerations, std::vector<double>& durations)` compute the trajectory parameters required to move from the default position to that configuration. These trajectory parameters include:

- `positions`: The different positions (joint angles) in the trajectory
- `velocities`: The joint velocities at each waypoint in the trajectory
- `accelerations`: The joint accelerations at each waypoint in the trajectory
- `duations`: the duration of movement between each joint angle in the trajectory

The flow of the gesture execution of a deictic gesture is shown in Figure 2 below. Given a location x_p, y_p, z_p in three-dimensional space required to point to, the algorithm for the system is listed in Algorithm 1 below.

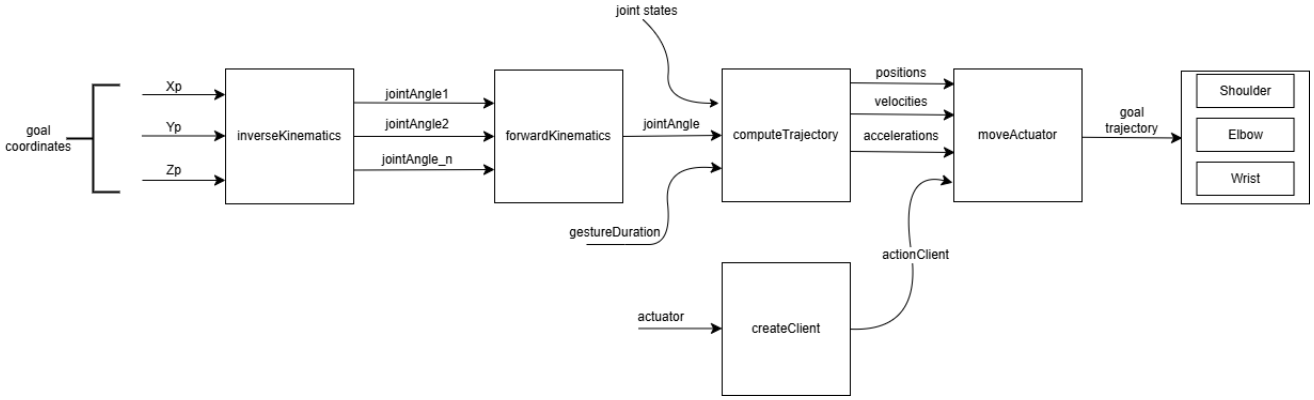

Figure 2: Architecture of the Gesture Control System for Deictic Gestures

4.2 Iconic and Symbolic Gestures

The specifications for these gestures are in joint space, which are specified in different files for the different gestures. The specifications for the wave gesture descriptors are in the file named `waveGestureDescriptors.dat`. Welcome gestures are a composite of two arms, thus, the specification for this gesture is in two files `lArmWelcomeGestureDecriptors` and `rArmWelcomeGestureDescriptors`. The specifications in the files contain the information about each gesture. This information includes the ID of the gesture, the number of waypoints (including the start and end joint angle), and the joint angles at each waypoint (delimited by a semicolon). The gestures have been allocated IDs, which are stored in a file named `gestureDescriptors.dat` and specified in table 1 below:

Table 1: Iconic and Symbolic Gestures and their Allocated IDs

ID	Gesture	Gesture Arm	Descriptor Filename
01	Welcome Gesture	Right Arm	rArmWelcomeGestureDescriptors.dat
		Left Arm	lArmWelcomeGestureDescriptors.dat
02	Welcome Gesture	Right Arm	rArmWelcomeGestureDescriptors.dat
		Left Arm	lArmWelcomeGestureDescriptors.dat
03	Wave Gesture	Right Arm	waveGestureDescriptors.dat
04	Shake Gesture	Right Arm	rArmShakeGestureDescriptors.dat
		Left Arm	lArmShakeGestureDescriptors.dat
05	Shake Gesture	Right Arm	rArmShakeGestureDescriptors.dat
		Left Arm	lArmWelcomeShakeDescriptors.dat

Upon receipt of the service request, the module reads the descriptor file for the ID requested. The joint angles at each waypoint are read from the file and parsed. If activated, the biological motion model computes the trajectory for the motion through the waypoints. The execution phase utilises an action server to translate the computed trajectory into physical movement. This server controls Pepper's arms, ensuring the gesture is performed within the specified duration and with the required precision. Throughout the execution, the system continuously monitors the gesture's progress. The flow of the gesture execution of an iconic gesture is shown in Figure 3 below. Given an ID $gesture\_id$, the algorithm for the system is listed in Algorithm 2 below.

Algorithm 1 Deictic Gesture Execution Algorithm

Require: $biologicalMotionFlag, actuatorJoint, gestureDuration, x_p, y_p, z_p$
Ensure: $gestureDuration > 0$
  $x \leftarrow x_p$
  $y \leftarrow y_p$
  $z \leftarrow z_p$
  $jointAngles \leftarrow$ inverseKinematics(x, y, z) ▷ Compute the joint angles
  for $jointAngle$ in $jointAngles$ **do**
    x_f, y_f, z_f = forwardKinematics$(jointAngle)$ ▷ Obtain the position
    if $x_f, y_f, z_f = x, y, z$ **then** ▷ JointAngle is valid
      break
    else
      $status \leftarrow 0$
      return $status$
    end if
  end for
  $jointClient \leftarrow$ createClient$(actuatorJoint)$ ▷ Create ROS actionClient
  if $biologicalMotionFlag$ is True **then**
    $Positions, Velocities, Accelerations \leftarrow$ computeTrajectory$(jointAngle)$
    $status \leftarrow$ moveActuator$(jointClient, Positions, Velocities, Accelerations)$
  else
    $status \leftarrow$ moveActuator$(jointClient, jointAngle)$ ▷ Move the joint
  end if
  return $status$

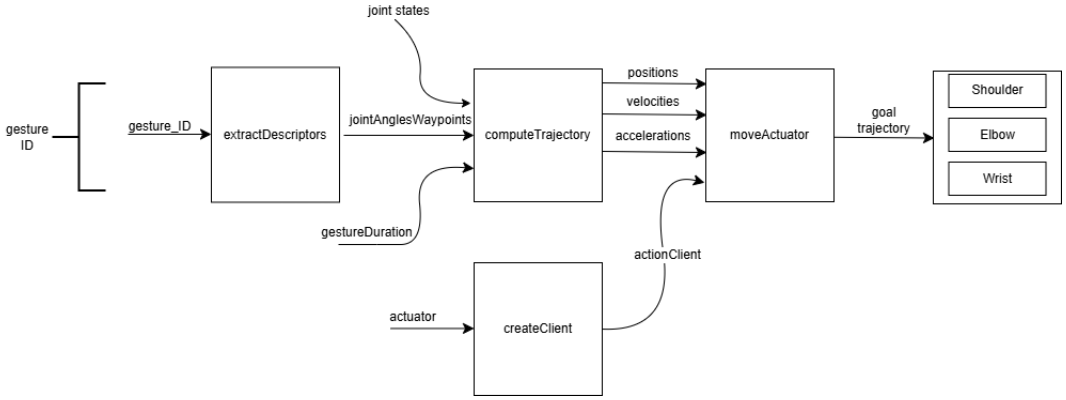

Figure 3: Architecture of the Gesture Control System for Iconic Gestures

Algorithm 2 Iconic Gesture Execution Algorithm

Require: $biologicalMotionFlag, actuatorJoint, gestureDuration, gesture\_id$
Ensure: $gestureDuration > 0$
  $gestureID \leftarrow gesture\_id$
  $jointAngleWaypoints \leftarrow$ extractDescriptors($gestureID$) ▷ Extract the waypoints
  $jointClient \leftarrow$ createClient($actuatorJoint$) ▷ Create ROS actionClient
  if $biologicalMotionFlag$ is True **then**
    $Positions, Velocities, Accelerations \leftarrow$ computeTrajectory($jointAngleWaypoints$)
    $status \leftarrow$ moveActuator($jointClient, Positions, Velocities, Accelerations$)
  else
    $status \leftarrow$ moveActuator($jointClient, jointAngle$) ▷ Move the joint
  end if
  return $status$

Work is being done on specifying the iconic and symbolic hand gestures by learning the required motions either by demonstration, using an RGB-D depth camera to determine the joint angles of human gestures in a skeletal model and mapping these to the robot joints. This is documented in D5.5.1.2 Programming by Demonstration.

4.3 Bow Gestures

Upon receipt of the service request specifying a degree to bow, the module processes the input parameters and if activated, the biological motion model computes the trajectory for the motion. The execution phase utilises an action server to translate the computed trajectory into physical movement. This server controls Pepper's hip and knee joints, ensuring the gesture is performed within the specified duration and with the required precision. Throughout the execution, the system continuously monitors the gesture's progress. The flow of the gesture execution of a bow gesture is shown in Figure 4 below. Given an angle $theta\_degrees$ in degrees required to bow, the algorithm for the system is listed in Algorithm 3 below.

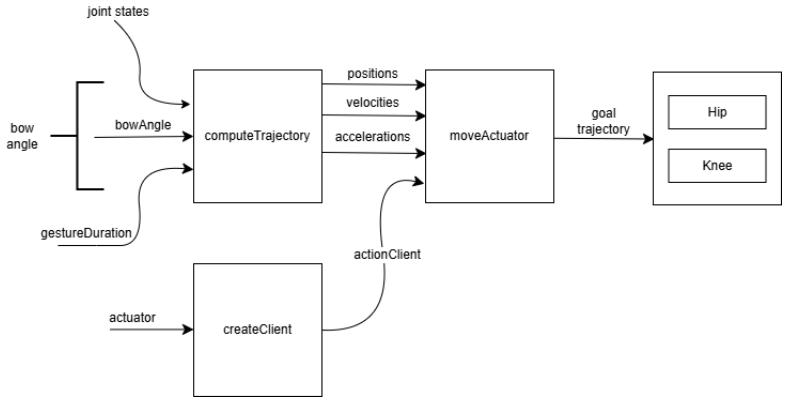

Figure 4: Architecture of the Gesture Control System for Bow Gestures

Algorithm 3 Bow Gesture Execution Algorithm

Require: $biologicalMotionFlag, actuatorJoint, gestureDuration, theta\_degrees$
Ensure: $gestureDuration > 0$
  $jointAngle \leftarrow theta\_degrees$
  $jointClient \leftarrow$ createClient($actuatorJoint$) ▷ Create ROS actionClient
  if $biologicalMotionFlag$ is True **then**
    $Positions, Velocities, Accelerations \leftarrow$ computeTrajectory($jointAngle$)
    $status \leftarrow$ moveActuator($jointClient, Positions, Velocities, Accelerations$)
  else
    $status \leftarrow$ moveActuator($jointClient, jointAngle$) ▷ Move the joint
  end if
  return $status$

4.4 Nod Gestures

Upon receipt of the service request specifying a degree to nod, the module processes the input parameters and if activated, the biological motion model computes the trajectory for the motion. The execution phase utilises an action server to translate the computed trajectory into physical movement. This server controls Pepper's hip and knee joints, ensuring the gesture is performed within the specified duration and with the required precision. Throughout the execution, the system continuously monitors the gesture's progress. The flow of the gesture execution of a nodding gesture is shown in Figure 5 below. Given an angle $theta\_degrees$ in degrees required to nod, the algorithm for the system is listed in Algorithm 4 below.

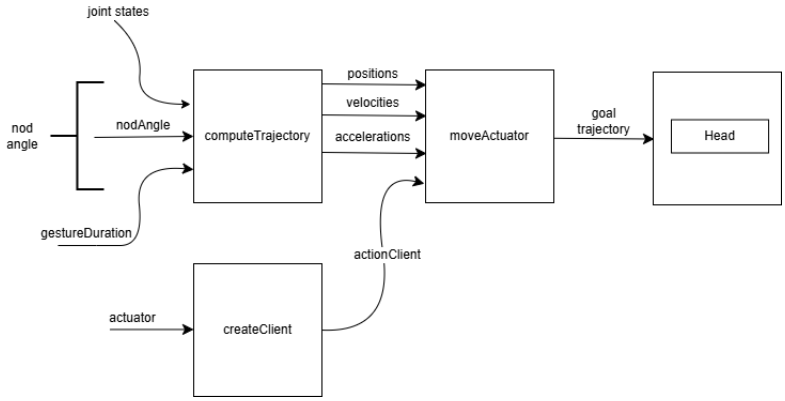

Figure 5: Architecture of the Gesture Control System for Nod Gestures

Algorithm 4 Nod Gesture Execution Algorithm

Require: $biologicalMotionFlag, actuatorJoint, gestureDuration, theta\_degrees$
Ensure: $gestureDuration > 0$
  $jointAngle \leftarrow theta\_degrees$
  $jointClient \leftarrow$ createClient($actuatorJoint$) ▷ Create ROS actionClient
  if $biologicalMotionFlag$ is True **then**
    $Positions, Velocities, Accelerations \leftarrow$ computeTrajectory($jointAngle$)
    $status \leftarrow$ moveActuator($jointClient, Positions, Velocities, Accelerations$)
  else
    $status \leftarrow$ moveActuator($jointClient, jointAngle$) ▷ Move the joint
  end if
  return $status$

5 Implementation

File Organization

The source code for executing the gestures is structured into three primary components: gestureExecutionApplication, gestureExecutionImplementation, and pepperKinematicsUtilitiesImplemmentation. The gestureExecutionImplementation component encapsulates all the essential functionality required for executing the gestures. This includes diectic, iconic, bow, and nod gestures. The gesture execution node is also equipped with the functionality to process various files critical for the execution process, which include configuration files, gesture descriptor files, and topic files.

On the other hand, the gestureExecutionApplication invokes those functions for the execution process. It is tasked with the execution of functions defined within the gestureExecutionImplementation and pepperKinematicsUtilitiesImplementation, effectively managing the gesture execution operations.

The file structure of the gesture execution node in the `cssr_system` package is shown in Figure 6 below.

```
cssr_system
└── gestureExecution
    ├── config
    │   └── gestureExecutionConfiguration.ini
    ├── data
    │   ├── gestureDescriptors.dat
    │   ├── lArmShakeGestureDescriptors.dat
    │   ├── lArmWelcomeGestureDescriptors.dat
    │   ├── pepperTopics.dat
    │   ├── rArmShakeGestureDescriptors.dat
    │   ├── rArmWelcomeGestureDescriptors.dat
    │   ├── simulatorTopics.dat
    │   └── waveGestureDescriptors.dat
    ├── include
    │   └── gestureExecution
    │       ├── gestureExecutionInterface.h
    │       └── pepperKinematicsUtilitiesInterface.h
    ├── launch
    ├── msg
    │   └── Gesture.msg
    ├── src
    │   ├── gestureExecutionApplication.cpp
    │   ├── gestureExecutionImplementation.cpp
    │   └── pepperKinematicsUtilitiesImplementation.cpp
    ├── srv
    │   └── performGesture.srv
    ├── README.md
    └── CMakeLists.txt
```

Figure 6: File Structure for the Gesture Execution Node

Configuration File

The operation of the gestureExecution node is determined by the contents of the configuration file that contains a list of key-value pairs as shown below.

The configuration file is named `gestureExecutionConfiguration.ini`

Table 2: Configuration file for the gesture execution node

Key	Value	Description
`interpolation`	`linear` or `biological`	Specifies the interpolation type. This indicates how the joint angles that define the trajectory in joint space between the current joint angles and the gesture joint angles are computed for body gesture and hand deictic gestures and between waypoints for iconic and symbolic gestures. The two options are: (a) independent linear interpolation of each joint angle, and (b) biological motion, selecting the sequence of joint angular velocities and joint accelerations to form a trajectory in time and joint space that mimics biological movement.
`gestureDescriptors`	`gestureDescriptors.dat`	Specifies the filename of the file in which the gesture descriptors are stored. This file contains the information about each iconic gesture descriptor, which includes the ID, the arm to be used and the filename of the file containing the descriptors for the gesture.
`robotTopics`	`robotTopics.dat`	Specifies the filename of the file in which the physical Pepper robot sensor and actuator topic names are stored.
`simulatorTopics`	`simulatorTopics.dat`	Specifies the filename of the file in which the simulator sensor and actuator topic names are stored.
`verboseMode`	`true` or `false`	Specifies whether diagnostic data is to be printed to the terminal.

Input File

There is no input data file for the gesture execution node. The gestures are executed based on the service request provided by a client node.

Output Data File

There is no output data file for the gesture execution node. The result of the gesture execution is returned as a response to the client that invoked the service and diagnostic messages are printed on the screen, depending on the value of `verboseMode` key in the configuration file.

Topics File

For the node, a selected list of the topics for the robot and simulator is stored in the topics file. The topic files are written in the .dat file format. The data file is written in key-value pairs where the key is the actuator name and the value is the topic

The topics file for the robot is named `robotTopics.dat` and the topics file for the simulator is named `simulatorTopics.dat`.

Topics Subscribed

This node subscribes to one topic, published by `robotLocalization` node, which provides the pose of the robot.

Table 3 lists the topic to which the `gestureExecution` node subscribes.

Table 3: Topic Subscribed to by the gestureExecution node.

Topic	Node	Platform
/robotLocalization/pose	robotLocalization	Physical robot

Topics Published

Table 4 lists the topics to which the `gestureExecution` node publishes. These are specified in the files identified by the `robotTopics` and `simulatorTopics` key-value pairs in the configuration file.

Table 4: Topics Published by the gestureExecution node.

Topic	Actuator	Platform
/pepper_dcm/LeftArm_controller/ follow_joint_trajectory	LShoulderPitch, LShoulderRoll, LElbowYaw, LElbowRoll, LWristYaw	robot
/pepper_dcm/RightArm_controller/ follow_joint_trajectory	RShoulderPitch, RShoulderRoll, RElbowYaw, RElbowRoll, RWristYaw	robot
/pepper_dcm/LeftHand_controller/ follow_joint_trajectory	Left Hand	robot
/pepper_dcm/RightHand_controller/ follow_joint_trajectory	Right Hand	robot
/pepper_dcm/Pelvis_controller/ follow_joint_trajectory	HipRoll, HipPitch, KneePitch	robot
/pepper_dcm/cmd_moveto	Wheels	robot
/pepper/LeftArm_controller/ follow_joint_trajectory	LShoulderPitch, LShoulderRoll, LElbowYaw, LElbowRoll, LWristYaw	simulator
/pepper/RightArm_controller/ follow_joint_trajectory	RShoulderPitch, RShoulderRoll, RElbowYaw, RElbowRoll	simulator
/pepper/Pelvis_controller/ follow_joint_trajectory	HipRoll, HipPitch, KneePitch	simulator
/pepper/cmd_vel	Wheels	simulator

Services Supported

This node provides and advertizes a server for a service `/gestureExecution/perform_gesture` to initiate the performance of a required gesture. It uses a package-specific msg, `Gesture.msg`. The message has several fields, as follows:

Table 5: Fields in the `Gesture.msg` of the gestureExecution node.

Field	Field Value	FIeld Type	Units
gesture_type	iconic, symbolic deictic, bow, nod	String	
gesture_id	<number>	Integer	
gesture_duration	<number>	Integer	milliseconds
bow_nod_angle	<number>	Integer	degrees
location_x	<number>	Real	metres
location_y	<number>	Real	metres
location_z	<number>	Real	metres

If the `perform_gesture` request is successful, the service response is "1"; if it is unsuccessful, it is "0". The service is called by the `behaviorController` node.

Table 6 summarizes the services supported.

Table 6: Service supported by the gestureExecution node.

Service	Message Value	Effect
/gestureExecution/perform_gesture	iconic, symbolic deictic, bow, nod	Perform an iconic, symbolic, deictic, bow, or nod gesture

Services Called

This node calls the `/overtAttention/set_mode` service as shown in Table 7.

Table 7: Service called by the gestureExecution node.

Service	Message Value	Effect
/overtAttention/set_mode	mode ("location"), location_x location_y, location_z	Invoke the attention subsystem to look at a location in the world

The type of variable that is passed as an argument to the `overtAttention/set_mode` service and the type of the service call return value is defined in D5.3 Overt Attention. Specifically, the location mode of the overtAttention node is invoked and the required point coordinates are passed as request in this service call.

6 Executing the Gestures

The implementation of the gesture execution node on the Pepper robot is realized as a ROS service `/gestureExecution/perform_gesture`, which is hosted and can be called with specific parameters to request the execution of gestures. The service provides a flexible and convenient interface for controlling the robot's gestures, allowing for customization of gesture type, duration, angles of bowing and nodding, as well as the target coordinate for pointing in the world.

The ROS service is invoked with the following parameters:

- Gesture Type: the type of gesture to be executed (e.g., `deictic`, `iconic`, `bow`, `nod`).
- Gesture ID: the ID of an iconic gesture to be executed (e.g., `01`, `02`, `03`).
- Duration: the duration of the gesture, controlling the speed at which the gesture is performed.
- Angle of Bowing: the angle at which the robot should bow if the gesture is bowing.
- Angle of Nodding: the angle at which the robot should nod if the gesture is nodding.
- Target Coordinate: the target coordinate in the world that the robot should point to, if the gesture is a deictic pointing gesture.

To run the node, the user must run the following command (after waking the robot):

```
rosrun cssr_system gestureExecution
```

NOTE: The node requires the availability of the `/rootLocalization/pose` topic and the `overtAttention/set_mode` services to run. This services can be made available by one of two ways, outlined below:

Option 1: Running the `overtAttention` and `robotLocalization` nodes in `cssr_system` package (if available)

```
rosrun cssr_system robotLocalization
```

```
rosrun cssr_system overtAttention
```

Option 2: Running the `gestureExecutionDriver` and `gestureExecutionStub` nodes in the `unit_tests` package and providing the actual robot pose in the physiscal environment

```
rosrun unit_tests gestureExecutionStub
```

```
rosrun unit_tests gestureExecutionDriver <robot_x> <robot_y>
 <robot_theta>
```

After the node is run, the `/gestureExecution/perform_gesture` service is available and can be invoked by running the following command:

```
rosservice call /gestureExecution/perform_gesture -- <gesture_type> <
   gesture_id> <gesture_duration> <bow_nod_angle> <location_x> <
   location_y> <location_z>
```

To execute a deictic gesture at a point `x_p, y_p, z_p` for `T ms`, the user must invoke the service with the command below (replacing the duration and the pointing coordinates with the actual coordinates):

```
rosservice call /gestureExecution/perform_gesture -- \
deictic 01 T 0 x_p y_p z_p
```

To execute an iconic gesture with gesture ID `gesture_id` for `T ms`, the user must invoke the service with the command below (replacing the duration and the gesture ID placeholder with the actual parameters):

```
rosservice call /gestureExecution/perform_gesture -- \
iconic gesture_id T 0 0 0 0
```

To execute a bow gesture at `theta_degrees` for `T ms`, the user must invoke the service with the command below (replacing the duration and the bow angle placeholder with the actual bow angle):

```
rosservice call /gestureExecution/perform_gesture -- \
bow 01 T theta_degrees 0 0 0
```

To execute a nod gesture at `theta_degrees` for `T ms`, the user must invoke the service with the command below (replacing the duration and the nod angle placeholder with the actual nod angle):

```
rosservice call /gestureExecution/perform_gesture --  \
bow 01 T theta_degrees 0 0 0
```

For executing gestures on the physical robot, the NAOqi DCM (Device Communication Manager) driver is used to control the robot's actuators. The driver provides a hardware interface to connect to Alderban's robot Nao, Romeo, and Pepper robots. The module is designed to move the joint actuator of the robot to a specified position by defining the trajectory goal and sending it to a control server via a ROS topic.

First, the module initializes a client for interacting with the ROS actions server. The function is a structured attempt to establish a connection with the server. The connection to the server is attempted multiple times before giving up and throwing an error. The function `ControlClientPtr create_client(const std::string& topic_name)` takes in the topic name and returns the actionClient pointer.

After the client is created, the module commands the robot to move to a specified position by defining a trajectory goal and sending it to the control server via the ROS action client. First, the module defines a goal message for a joint trajectory action, which is part of the `control_msgs` package. The `follow_joint_trajectory` action is used to generate a more complex motion control mechanism, often used for executing predefined paths or trajectories for a set of joints.

The components of this topic include:

- `/goal`: used to send a "FollowJointTrajectoryGoal", which includes a trajectory comprising multiple points (position, velocities, acceleration, and/or efforts for each joint) and the time at which those points should be reached.
- `/cancel`: can cancel a currently executing trajectory.
- `/feedback`: provides real-time feedback about the current state of the trajectory execution.
- `/result`: provides the outcome of the trajectory execution after completion.
- `/status`: provides status information about the goal, such as if it's active, succeeded, or aborted

The trajectory goal is sent to the action server through the client. The server, presumably a part of a motion control system, interprets this goal and commands the robot arm to move accordingly. The system then waits for a fixed duration for the movement to complete before proceeding.

7 Unit Tests

To start the unit tests, the user must first install the necessary software packages as outlined in Deliverable D3.3. The operation of the unit test is controlled by a configuration file `gestureExecutionTestConfiguration.ini` which contain key-value pairs required to execute the unit tests as shown below.

Table 8: Configuration file for the gesture execution unit tests node

Key	Value	Description
`iconic`	`true` or `false`	Specifies whether to run the iconic gestures test
`deictic`	`true` or `false`	Specifies whether to run the deictic gestures test
`bow`	`true` or `false`	Specifies whether to run the bow gestures test
`nod`	`true` or `false`	Specifies whether to run the nod gestures test
`verboseMode`	`true` or `false`	Specifies whether diagnostic data is to be printed to the terminal.

Referring to Table 8 above, the user must specify which gestures to test using the key-value pairs, with the gesture type (`deictic`, `iconic`, `bow` and `nod`) as the key and the status (`True` or `False`) as the value. To launch the gesture execution test, the user must run the following commands:

```
# Launch the physical robot
roslaunch unit_tests gestureExecutionLaunchRobot.launch \
robot_ip:=<robot_ip> roscore_ip:=<roscore_ip> \
network_interface:=<network_interface>
```

```
# Launch the physical robot
roslaunch unit_tests gestureExecutionLaunchTestHarness.launch \
launch_drivers_stubs:=true launch_test:=true initial_robot_x:=<robot_x>
initial_robot_y:=<robot_y> initial_robot_theta:=<robot_theta>
```

The above commands will launch the test for the robot. The unit tests wake up the robot, launch the gesture execution node (to make the `gestureExecution/perform_gesture` service available), launch a driver for the robotLocalization node (to publish the pose of the robot) and launch a stub for the `/overtAttention/set_mode` service which randomly returns success or failure when the gesture execution node invokes the service. Based on the status of the gesture being executed, this result is saved in a file `gestureExecutionTestOutput.dat`. This file stores the input requirement for each gesture and the status of the gesture (either `Success` or `Failure`).

NOTE: Setting the argument `launch_test` to `true` runs the tests based on the configuration, while setting it to `false` only launches the node and its dependencies. This is particularly useful if the user wants to run unique tests on the node, which are not available in this document. Setting the argument `launch_drivers_stubs` to `true` launches the drivers and stubs required to drive the `gestureExecution` node from the `unit_tests` package, while setting it to `false` launches the actual nodes (if available) which include `faceDetection`, `soundDetection`, `overtAttention`, and `robotLocalization` from the `cssr_system` package. The default values are `true`. Ensure that the robot pose matches the exact pose of the robot in the world frame of reference.

References

[1] C. Bartneck, T. Belpaeme, F. Eyssel, T. Kanda, M. Keijsers, and S. Sabanovic. *Human-Robot Interaction – An Introduction*. Cambridge University Press, 2020.

[2] Markus Huber, Markus Rickert, Alois Knoll, Thomas Brandt, and Stefan Glasauer. Human-robot interaction in handing-over tasks. *RO-MAN 2008 - The 17th IEEE International Symposium on Robot and Human Interactive Communication*, pages 107–112, 2008.

Principal Contributors

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

Adedayo Akinade, Carnegie Mellon University Africa.
David Vernon, Carnegie Mellon University Africa.

Document History

Version 1.0

First draft.
Adedayo Akinade.
10 October 2024

Version 1.1

Renumbered to D5.5.1.1 to facilitate an additional deliverable for Task 5.5.1: D5.5.1.2 Programming by Demonstration.
David Vernon.
6 January 2025.

Version 1.2

Removed all references to simulator platform, provided additional information on running the unit tests, and added hyperlinks to relevant external documentation.
Adedayo Akinade.
4 February 2025.