Hochschule
Bonn-Rhein-Sieg
University of Applied Sciences

b-it Bonn-Aachen International Center for Information Technology

Human-Centered Interaction in Robotics

# HCIR Assignment-2

*Shrikar Nakhye*
*Trushar Ghanekar*
*Keerthan Shekarappa*

May 2024

# Contents

# 1

# TASK 1

## 1.1 Verbal and Nonverbal Behaviours

Comment on the nonverbal communication capabilities of TIAGo Pro, NAO robot and aibo robot. Focus on kinesics (facial expressions, hand gestures, body movements, eye gaze) and vocalics, from the production and consumption points of view.

### 1.1.1 TIAGo Pro

**Kinesics**

- Body Structure and Movement: Upper part of it resembles a human body, consists of Torso, Arms with a 2-finger gripper Lower part of it is an Autonomous Guided vehicle with 4 wheels
- Like humans, it can adjust its posture to convey different attitudes or intentions. For instance, a straight and upright posture might suggest confidence or readiness, while a slouched posture might indicate relaxation or disengagement.
- Facial Expressions: It portrays a LED screen with a default GUI of a human face having two eyes and a mouth. It can portray some emotions
- Hand Gestures: Its manipulator arm movements can be considered a form of nonverbal communication. For example, extending its arm towards an object might indicate an intention to grasp or manipulate it, while retracting its arm might signal completion of a task.
- Eye Gaze While it doesn't have eyes, its screen can simulate a form of gaze. For example, it might turn its "head" or orient its screen towards objects or people it's interacting with, indicating focus or attention.

**Vocalics**

- Sound: Though not strictly nonverbal, the sounds it produces, such as motor noises or beeps, can also convey information. For instance, different beep patterns might signal different states or actions, such as completion of a task or an error.

### 1.1.2 NAO

**Body Structure and Movement:**

- NAO features 25 joints (i.e. 25 DoF) distributed across its body, allowing for precise movement and coordination, enabling a wide range of body movements.
- The head joints allow for nodding, pitching, and yawing, mimicking human-like head gestures and expressions.
- Its body structure comprises a torso, head, and pairs of arms and legs, resembling a simplified human anatomy. This design enables NAO to perform various actions, such as sitting, crouching, and lying down, enhancing its flexibility and adaptability in different scenarios.

**Facial Expression:**

- Although NAO lacks a movable mouth, it compensates for this limitation with LED eyes capable of blinking.
- The LED eyes can display different colors and durations of blinking, allowing for basic facial expressions.
- Blinking during conversations adds a level of expressiveness to NAO's interactions with users.
- For instance, when the LEDs are blue, it signifies that NAO is actively listening, enhancing its communicative abilities and engagement.

**Verbal Communication:**

- NAO is proficient in speaking 20 languages, making it accessible to users from diverse linguistic backgrounds.
- Two speakers positioned on the left and right sides of NAO's head facilitate clear and directional audio output.
- A broadcast system enhances the projection of sound, ensuring effective communication even in noisy environments.
- Additionally, four omnidirectional microphones capture audio input from the surrounding environment, enabling NAO to listen to and process spoken commands or engage in dialogue.
- Two video cameras located at the front of NAO's head support visual perception and communication, allowing the robot to recognize faces, gestures, and objects, further enriching its interaction capabilities.

### 1.1.3 Aibo

**Body Structure and Movement:**

- Aibo resembles a small dog and features joints for walking, running, sitting, and standing. Its body is made of durable materials to withstand regular interaction.
- Aibo can move it's body by wagging its tail when it's happy or excited, lowering its head when it's tired or sad, or tilting its head to express curiosity or confusion.
- Aibo's face is primarily an LED lights, it cannot display expressions,

**Sounds and Vocalizations:**

- Aibo can produce different sounds and vocalizations to express emotions or communicate certain states. For example, it might bark happily when playing or whine when it wants attention.

## 1.2 Behavior Markup Language

Represent the multimodal communicative behaviour shown in Figure 1 in the Behavior Markup Language format.

```
<bml xmlns="http://www.bml-initiative.org/bml/bml-1.0"character="Tezz" id="bml1">
    <gaze id="gaze1" start="0" target="human" />
    <speech id="speech1" start="1">
        <text>Hello!</text>
    </speech>
    <gesture id="wave" lexeme="hello-waving" start="2" end="5">
        <speech id="speech2" start="wave:end">
            <text>Glad to see you!</text>
        </speech>
    </gesture>
    <head id="nod" lexeme="NOD" start="gaze1:end" end="9" />
    <posture id="happy_swirl" start="nod:end" end="12">
        <stance type="SWIRL" />
    </posture>
</bml>
```

# 2

# Task 2

## 2.1 Realise Multimodal Behaviour on Pepper

In this task, you will use the qiBullet simulation tool to realise the multimodal behaviour shown in Figure-1. Please complete the following subtasks:

1. Specify the shown multimodal behaviour in the form of a dictionary in BML syntax.
2. Extend your source code from Homework #1 to realise the behaviour shown in Figure 1.
3. Add comments in your code to explain your logic.

```
import time
import concurrent.futures
from qibullet import SimulationManager, PepperVirtual
from gtts import gTTS
from playsound import playsound

def wave(pepper):
    for _ in range(2):
        pepper.setAngles("RShoulderPitch", -0.5, 0.5)
        pepper.setAngles("RShoulderRoll", -1.5620, 0.5)
        pepper.setAngles("RElbowRoll", 1.5620, 0.5)
        time.sleep(1.0)
        pepper.setAngles("RElbowRoll", -1.5620, 0.5)

        time.sleep(1.0)


def normal(pepper):
    # Moving pepper to normal position
    pepper.goToPosture("StandInit", 0.6)
    time.sleep(1.0)

def speak():
    time.sleep(1)
    tts = gTTS("Hello")
```

```
    tts.save("message.mp3")
    playsound("message.mp3")

def speak_2():
    time.sleep(1)


    tts = gTTS("Glad to see you")
    tts.save("message1.mp3")
    playsound("message1.mp3")

def head_nod(pepper):
    for _ in range(2):

        pepper.setAngles("HeadPitch", 0.5, 0.5)  # Nod down
        time.sleep(1.0)
        pepper.setAngles("HeadPitch", -0.5, 0.5)  # Nod up
        time.sleep(1.0)

def swirl(pepper):

    for _ in range(1):
        pepper.setAngles("LShoulderPitch",-1.5708,0.7)
        pepper.setAngles("RShoulderPitch",-1.5708,0.7)
        pepper.moveTo(0,0,6,1,1)
        time.sleep(1)
        pepper.setAngles("LShoulderPitch",1.5708,0.7)
        pepper.setAngles("RShoulderPitch",1.5708,0.7)
        time.sleep(1)

def speak_3():
    time.sleep(1)

    # Generate spoken message and play it
    tts = gTTS("Yeeeeaaahhhh Swirl completed")
    tts.save("message2.mp3")
    playsound("message2.mp3")
if __name__ == "__main__":

    simulation_manager = SimulationManager()
    client = simulation_manager.launchSimulation(gui=True)


    pepper = simulation_manager.spawnPepper(client, spawn_ground_plane=True)
    pepper.goToPosture("Crouch", 0.6)
    time.sleep(1)
    pepper.goToPosture("StandInit", 0.6)
    time.sleep(1)

```

```
    # Schedule wave, normal, speak, and nod functions to run concurrently
    with concurrent.futures.ThreadPoolExecutor(max_workers=4) as executor:
        # Submit wave function
        future_wave = executor.submit(wave, pepper)
        future_speak = executor.submit(speak)

        # Wait for wave function to complete
        concurrent.futures.wait([future_wave,future_speak])

        # Submit normal function
        future_normal = executor.submit(normal, pepper)

        # Wait for normal function to complete
        concurrent.futures.wait([future_normal])


        # Wait for speak function to complete
        concurrent.futures.wait([future_speak])

        # Submit speak_2 and nod functions concurrently
        future_speak_2 = executor.submit(speak_2)
        future_nod = executor.submit(head_nod, pepper)

        # Wait for both speak_2 and nod functions to complete
        concurrent.futures.wait([future_speak_2, future_nod])

        #submit swirl function
        future_swirl = executor.submit(swirl,pepper)

        # Wait for swirl function to complete
        concurrent.futures.wait([future_swirl])

        # Submit speak_3
        future_speak_3 = executor.submit(speak_3)

        # Wait for both speak_3 functions to complete
        concurrent.futures.wait([future_speak_3])



    # Stop the simulation
    simulation_manager.stopSimulation(client)
```

Listing 2.1: Python Code for Pepper to do multiple behaviour

# 3

# Task 3

## 3.1 Incremental Dialog Processing

Read the paper (Skantze and Schlangen, 2009) and answer the questions given below:

### 3.1.1 What do you understand by ”incremental dialog processing”?

Incremental dialogue processing refers to a method of processing dialogue in a step-by-step and ongoing manner, where the system continuously analyzes and generates responses while the conversation is still in progress. Instead of waiting for the completion of an entire user's utterance before generating a response, an incremental dialogue system starts processing and responding to input as soon as possible, even if the input is not yet fully formed. This approach mirrors the incremental nature of human conversation, where individuals often start speaking before fully planning their utterances and make use of multiple cues and feedback to determine when to speak and how to adapt their responses. By incorporating incremental processing, a dialogue system can provide faster feedback, allow for back-and-forth interactions, and more closely resemble natural human conversation.

### 3.1.2 How does incremental dialog processing support human-centeredness in interaction?

Incremental dialogue processing supports human-centeredness in interaction by emulating the natural dynamics and characteristics of human conversation. Here are a few ways in which it achieves this:

- Rapid Feedback: By processing dialogue incrementally, the system can provide rapid feedback to the user as they are speaking. This mimics the real-time nature of human conversation, where interlocutors often provide immediate reactions, acknowledgments, or back-channels while the other person is still speaking. Prompt feedback creates a more engaging and interactive dialogue experience, making the system feel more human-like.
- Turn-Taking: Incremental processing enables the system to detect cues for turn-taking, such as pauses, prosodic features, or syntactic cues, allowing it to determine when it is appropriate to take

turns in the conversation. This supports smoother and more natural turn-taking dynamics, similar to how individuals engage in conversations with each other.

- Monitoring User Feedback: An incremental dialogue system can continuously monitor and adapt to user feedback while it is generating its own responses. This means that if the user interrupts or provides feedback during the system's utterance, it can appropriately respond or adjust its output. This capability enhances the system's ability to understand and respond to the user's reactions, making the interaction more human-centric.

- Incremental Understanding: By processing input incrementally, the system can start interpreting and understanding the user's utterance before it is complete. This allows for early recognition of user intentions, faster comprehension, and the ability to provide proactive assistance or clarification. It resembles how humans make sense of partial information and adapt their understanding as the conversation unfolds.

## 3.1 Questions and Answers

### 3.1.3 In the example presented in the paper, how did prosodic analysis contribute to incrementality?

In the example presented in the paper, prosodic analysis contributed to incrementwality by providing additional information about the user's speech, particularly about how phrases and sentences were delivered in terms of pitch and duration. This analysis allowed the system to:

- **Identify Installment Types**: By using delta pitch and duration parameters, the system could distinguish between mid-sequence and end-sequence installments. This distinction is useful for understanding when the user expects feedback.

- **Time Responses Appropriately**: Prosodic features such as rising or falling pitch helped the system plan the timing of its responses, allowing it to provide immediate reactions when the user paused or finished a thought.

- **Improve Turn-Taking**: The prosodic analysis allowed the system to recognize natural breaks in the user's speech, enabling more efficient and responsive turn-taking.

### 3.1.4 In the architecture for spoken dialog systems presented in the paper, why is TTS connected to Discourse Modeller?

In the architecture for spoken dialogue systems presented in the paper, the Text-to-Speech (TTS) module is connected to the Discourse Modeller to allow the system to monitor its own spoken utterances and relate them to the user's responses. This connection enables the system to:

- **Track Its Own Utterances**: The TTS sends information about what it has said back to the discourse modeller, providing feedback about what the system has actually spoken to the user.
- **Update the Discourse Model**: By keeping track of its own spoken output, the system can keep an updated model of the conversation, allowing it to adjust its actions and responses based on the current state of the dialogue.
- **Enhance Grounding**: By informing the discourse modeller about its spoken utterances, the system can track which concepts have been grounded in the dialogue, ensuring a smoother and more coherent conversation.