NAO, Let's play the Xylophone

1st Zhiyu Wang
Robotics, Cognition, Intelligence
Technical University of Munich
Munich, Germany
ge87wax@mytum.de

2nd Yijia Qian
Robotics, Cognition, Intelligence
Technical University of Munich
Munich, Germany
go69jek@mytum.de

3rd Yuan Cao
Robotics, Cognition, Intelligence
Technical University of Munich
Munich, Germany
ge83kin@mytum.de

***Abstract*—This project presents the development of a system enabling the NAO humanoid robot to autonomously play melodies on a xylophone after listening to a music snippet. The robot utilizes its integrated microphone to record the played notes, processes the audio data to extract pitch and timing information, and reproduces the melody on the xylophone. The system combines human-robot interaction via speech and tactile interfaces, autonomous grasping of the xylophone sticks, pitch detection, and real-time performance evaluation. Key technical achievements include robust manipulation control using NAO Cartesian space APIs, accurate pitch mapping for melody learning, and error-corrective playback. This work demonstrates the integration of perception, motor control, and audio processing in a humanoid robot for interactive musical tasks.**

***Index Terms*—Humanoid Robotics, NAO Robot, Xylophone Playback, Human-Robot Interaction, Audio Processing, Pitch Detection, Vision-Guided Localization, Modular System Design**

I. Introduction

The ability of robots to interact with their environment and perform complex tasks such as music playback demonstrates the potential of combining perception, manipulation, and control systems. This project aims to develop a system for the NAO humanoid robot to listen to a short music snippet, extract the melody, and reproduce it on a xylophone. The task integrates audio signal processing, robotic manipulation, and human-robot interaction into a unified framework.

This project involves multiple technical challenges, including integrating audio signal processing for pitch detection, implementing reliable robotic grasping, and ensuring precise timing control during musical performance.

II. System Architecture

This section introduces the design and implementation of a humanoid robotic system that enables the NAO robot to autonomously play the xylophone. The system adopts a task-based architecture, where each function is divided into sequential tasks, such as grasping mallets, computing key positions, playing melodies, and listening to notes. The system architecture is shown in Figure 1. Each module is designed to complete specific tasks in an orderly manner, reflecting characteristics similar to the piano-playing humanoid robot Kato.

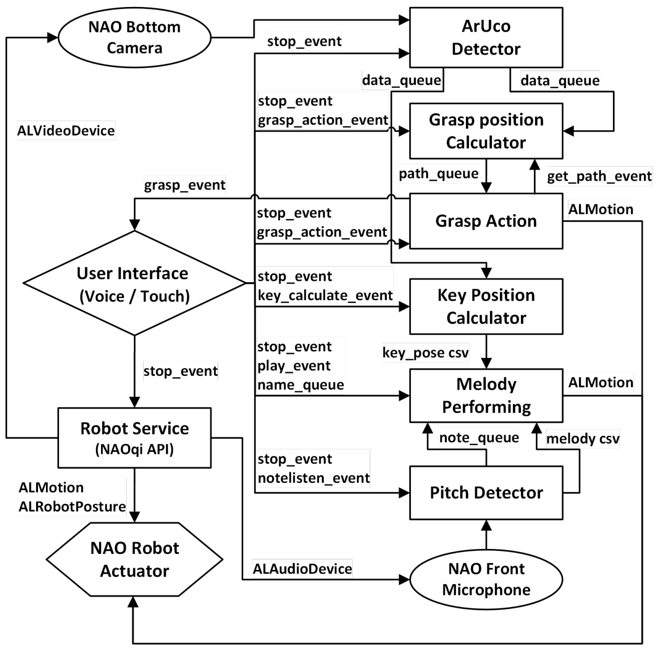

Fig. 1. System Architecture Diagram

- **Main Client and Stop Control**: The primary program orchestrates all threads and encompasses the robot's interactive interface. Interaction is facilitated through two primary modalities: voice commands and touch button controls. All threads continuously monitor for a `stop_event` and possess the highest level of trigger loop detection to ensure operational safety.
- **Robot Service**: This thread is responsible for initializing the robot's posture. Upon receiving a `stop_event`, it cancels all subscriptions, issues a rest command, and releases all stiffness to safely deactivate the robot's actuators.
- **ArUco Marker Detection**: After subscribing to the NAO robot's bottom camera, the ArUco marker detection process is continuously executed, utilizing exclusively ArUco markers `5x5_1000: 11, 21, 31, 41`, each with a size of 60 mm. The pose of each ArUco marker is transformed into the robot's torso frame and placed into a `data_queue` for access by other threads. The detection results are simultaneously displayed in an OpenCV (cv2) window, refreshed every second to facilitate real-time observation.
- **Key Position Computation**: Upon receiving and pre-

processing the results from ArUco markers [11, 21, 31], this module maps these markers to the corresponding positions of the robot's hands required for striking each xylophone key. The computed positions are exported as a CSV file. This thread operates under the control of the key_calculate_event, executing tasks when the event is set and remaining idle when the event is cleared.

- **Grasp Position Computation**: When the get_path_event is set, this module retrieves the result from ArUco marker [41] from the data_queue and preprocesses it. Utilizing predefined transformation relationships, it calculates the grasping position and places the result into a path_queue. Subsequently, it clears the get_path_event, awaiting the next set signal to execute.
- **Grasp Action**: This module first verifies whether the grasping action has been completed via the grasp_event flag and determines if re-grasping is necessary. It sets the get_path_event and awaits the retrieval of grasping results. If the grasp is unsuccessful, it reinitiates the get_path_event. Upon successful grasping, it sets the grasp_event to indicate completion.
- **Melody Performing**: When the play_event is set, this module reads the melody name from the name_queue, locates the corresponding CSV file, and parses the key position data. It maps each musical note to the corresponding hand positions, forming a dictionary for execution. During playback, the module compares detected pitch feedback to ensure accuracy, making real-time corrections if necessary. Upon completing the performance, it clears the play_event and terminates the playback sequence.
- **Pitch Detection**: After subscribing to the NAO robot's front microphone, this module decodes the audio input and continuously detects pitch, providing ongoing feedback to the melody playback module. If the notelisten_event flag is set, it initiates recording and exports the detected pitches to a CSV file for further analysis.

This structured and modular approach ensures that each component of the system operates cohesively, facilitating the autonomous and accurate performance of musical tasks by the NAO robot.

III. User Interface and Security Mechanisms

Through the integration of multiple user interaction methods and the design of robust security mechanisms, the NAO robot is capable of providing a rich interactive experience while performing tasks such as playing the xylophone, ensuring the safety of both the user and the robot itself. The combination of touch and voice interactions allows the robot to respond flexibly to user needs, while safety measures like emergency stops, event management, and confirmation mechanisms provide solid security assurances during the interaction process. These designs not only enhance the practicality and reliability of the robotic system but also offer valuable references for future human-robot interaction systems.

A. Touch Interaction

The NAO robot is equipped with multiple touch sensors, enabling users to trigger specific functions by touching different parts of the robot. These touch events are handled by the ReactToTouch class, implemented as follows:

- **Event Subscription and Handling**: Utilizing the ALMemory proxy [1], the robot subscribes to the TouchChanged event. Once a touch event is detected, the onTouched method is invoked to parse the touched body part and execute the corresponding action.
- **Functions Triggered by Touch**:
  - **Left Foot Touch (”LFoot/Bumper/Left”)**: Stops all text-to-speech (TTS) outputs and speech recognition tasks, ensuring that users can interrupt the robot's speech feedback at any time.
  - **Right Foot Touch (”RFoot/Bumper/Right”)**: Activates the emergency stop mechanism by setting the stop_event, immediately terminating all ongoing tasks to safeguard both the user and the robot.
  - **Right Hand Touch (”RHand/Touch/Back”)**: Initiates the grasping action by setting the grasp_event and grasp_action events, prompting the robot to execute the grasp sequence.
  - **Left Hand Touch (”LHand/Touch/Back”)**: Triggers the playing function by setting the play_event, causing the robot to start playing the specified melody.
  - **Head Touch**:
    * **Front (”Head/Touch/Front”)**: Starts the key position calculation by setting the key_calculate_event, initiating the computation of key positions.
    * **Middle (”Head/Touch/Middle”)**: Activates the speech recognition feature by calling the speechrecognition method to recognize user commands.
    * **Rear (”Head/Touch/Rear”)**: Initiates the note listening feature by setting the notelisten_event, and automatically disables listening after a set duration to conserve resources.

B. Voice Interaction

The NAO robot integrates speech recognition and text-to-speech functionalities, allowing users to interact with the robot through voice commands. The implementation details are as follows:

- **Speech Recognition Setup**: Within the initialization method of the ReactToTouch class, the ALSpeechRecognition proxy [2] is configured to use English as the recognition language. A vocabulary list is defined, including commands such as “play”, “replay”, “listen”, “grasp”, “rest”, “check”, “yes”, and “no”.

- **Command Processing Workflow**:
  1) **Command Recognition**: Users initiate voice commands by touching the middle part of the robot's head, which triggers the `speechrecognition` method to listen for user commands.
  2) **Confirmation Mechanism**: Upon recognizing a command, the robot uses the `ALTextToSpeech` proxy [3] to verbally confirm the user's intent by asking, for example, "Do you want me to play, yes or no?". The user's confirmation is captured through another round of voice input.
  3) **Command Execution**: If the user confirms the command, the robot executes the corresponding action, such as playing a specified song, performing a grasp action, entering a rest state, or listening to notes.

C. Security Mechanisms

To ensure the safety of both the user and the robot, the code incorporates multiple layers of security mechanisms, primarily focusing on the following aspects:

1) Emergency Stop: The robot is designed with an emergency stop feature. When the right foot touch event (`"RFoot/Bumper/Right"`) is detected, the `stop_event` is set, immediately terminating all running threads and tasks. This mechanism ensures that users can quickly halt the robot's operations in case of an emergency, preventing potential hazards.

2) Event Management and Thread Control: The code employs a multi-threaded and event-driven architecture [4] to ensure that each functional module operates independently without causing conflicts. By utilizing `threading.Event` objects such as `stop_event` and `grasp_event`, the robot can systematically start and stop various tasks, preventing resource contention and task interference that could lead to safety issues.

3) Voice Recognition Confirmation Mechanism: Before executing any voice command, the robot verifies the user's intent through a confirmation step. The `check_answer` method ensures that commands are only carried out when the user explicitly confirms them. This reduces the risk of unintended operations caused by misrecognition of voice inputs, enhancing the overall safety of the interaction.

4) Resource Access Control: The robot manages resource access through appropriate proxy subscriptions and unsubscriptions. For instance, when performing grasp actions, the robot clears related events to ensure the exclusivity and safety of the grasp task. Additionally, video and audio streams are captured and processed in separate threads, preventing interference with the main control flow and maintaining system stability.

IV. ArUco Marker Processing

ArUco marker-based localization is a commonly employed method in robotic development due to its robustness and ease of integration. In the context of the NAO robot, several strategies were implemented to enhance the accuracy of pose estimation when utilizing ArUco markers. By meticulously calibrating the camera, selecting optimal marker sizes and dictionaries, employing multiple marker triangulation, implementing smoothing techniques, and utilizing OpenCV's ArUco detection with the PnP method, the NAO robot achieves high-accuracy pose estimation using ArUco markers. These strategies collectively enhance the robot's ability to interact seamlessly with its environment, perform precise actions, and maintain robust performance even under challenging conditions.

A. Camera Calibration

Accurate pose estimation using ArUco markers necessitates precise camera calibration. To achieve this, the bottom camera of the NAO robot was calibrated to obtain the intrinsic matrix and distortion coefficients. The calibration process involved capturing multiple images of a checkerboard pattern under various orientations and using OpenCV's calibration functions to compute the necessary parameters [5] [6]. Additionally, simulations of task scenarios under extreme conditions were performed to ensure the robustness of the calibration (fig. 2. The calibration parameters are stored in the `camerapara_dict` and `cameradist` variables within the code, facilitating accurate correction of lens distortions and precise pose estimation.

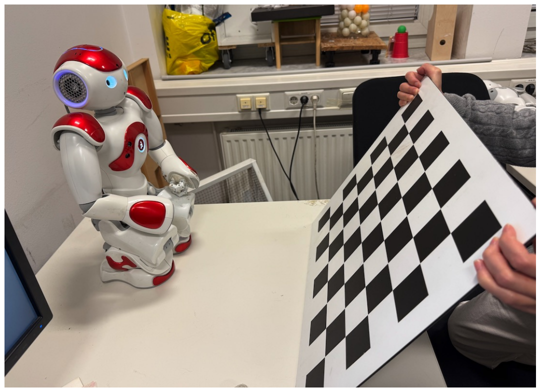
Fig. 2. Camera Calibration under extreme conditions

B. Resolution Selection

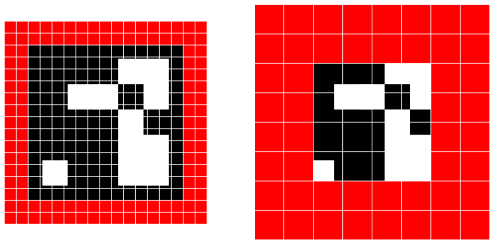
Fig. 3. Boundary Detection Error of Aruco Marker at Different Resolutions

Different camera resolutions were evaluated to balance accuracy and processing efficiency. The performance of

ArUco marker detection was assessed across multiple resolutions—320p, 640p, and 960p. The 960p resolution provided the highest accuracy in pose estimation but suffered from lower processing efficiency. Conversely, the 640p resolution offered a favorable trade-off, maintaining minimal pose estimation errors while ensuring acceptable processing speeds. Consequently, the 640p video stream was selected as the optimal resolution. However, the system retains the flexibility to adjust the resolution based on specific requirements. This decision is reflected in the `resolution` parameter within the `main` function, where 640p (denoted by `resolution = 2`) is chosen for optimal performance.

C. ArUco Marker Size and Dictionary Selection

The size and type of ArUco markers significantly impact detection accuracy and robustness. Two approaches were considered for deploying ArUco markers:

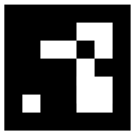

Fig. 4. Aruco Marker with ID 31 from the 5x5_1000 dictionary

- **Small Markers**: Placing smaller markers directly at the execution points proved susceptible to higher error margins due to their reduced size, which diminishes detection reliability.
- **Large Markers with Transformations**: Utilizing larger markers (60mm) fixed at relative positions, supplemented by additional pose transformations, enhanced detection robustness. The larger size facilitates more reliable detection and reduces the likelihood of errors inherent in smaller markers.

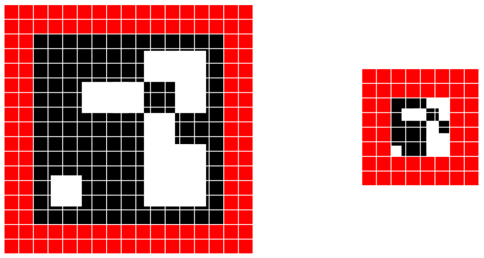

Fig. 5. Boundary Detection Error of Aruco Marker at Different Sizes

Based on these observations, the decision was made to employ larger 60mm ArUco markers using a more complex 5x5 dictionary. This choice enhances the system's robustness by improving detection accuracy and minimizing pose estimation errors. The selection of the 5x5 dictionary is optimized for larger markers, ensuring that the markers are easily distinguishable and reducing the probability of misidentification.

D. Mean Calculation and Smoothing

To address the variability and jitter inherent in real-time marker detection, statistical smoothing techniques were employed. The detection algorithm often experiences fluctuations due to environmental noise and rapid movements. To counteract this, the system aggregates the results of 20 consecutive detections and computes their mean values. This averaging process smooths out transient errors and provides a more stable and reliable output. In the code, this is managed by the `capture_video` function, which continuously captures frames, detects markers, and enqueues the results. The `calculate_grasp_positions` and related functions then process these queued results by computing their stable value (See Section VI-B, ArUco Marker Detection and Pose Estimation, for more details) before proceeding with further actions. This approach ensures that the robot's movements and interactions remain smooth and precise, even in the presence of occasional detection inconsistencies.

E. Detection Methodology

The detection of ArUco markers was performed using the OpenCV library, specifically leveraging the ArUco module's detection capabilities. The Perspective-n-Point (PnP) method was employed to estimate the pose of the markers relative to the camera [7] [8]. This approach involves solving the PnP problem, which determines the position and orientation of the camera with respect to the detected marker based on the 2D-3D point correspondences provided by the marker's corners.

The implementation utilizes the `cv2.aruco.detectMarkers` function [9] to identify markers within each captured frame. Once the markers are detected, the `cv2.solvePnP` function [10] is used in conjunction with the previously obtained calibration parameters (`camera_matrix` and `distortion_coefficients`) to compute the rotation and translation vectors that define the marker's pose. These vectors are then used to transform the marker's coordinates into the robot's torso frame, enabling accurate localization and interaction.

Integrating the PnP method within the detection pipeline ensures that the pose estimation is both accurate and efficient, facilitating real-time interactions and movements by the NAO robot based on the detected ArUco markers.

F. Implementation Overview

The integration of these methodologies is evident in the `client.py` script. The `capture_video` function is responsible for continuously capturing frames from the NAO robot's bottom camera, utilizing the calibrated intrinsic and distortion parameters to correct the images. The `ArucoDetector` class processes each frame to detect and transform ArUco markers, applying the chosen resolution and marker size configurations. Detected marker positions are then enqueued for further processing, where mean calculation and triangulation enhance the accuracy and stability of the robot's interactions.

Furthermore, the calibration script provided facilitates the initial setup by capturing calibration images and computing the necessary camera parameters. This calibration is crucial for ensuring that the pose estimations based on ArUco markers are as accurate as possible, thereby underpinning the reliability of the robot's operational tasks such as grasping and playing the xylophone.

V. FFT-Based Pitch Detection Method and Implementation

This section outlines the pitch detection and note recognition methodology employed using the robot's microphone input. It details the entire process from subscribing to audio data, preprocessing, fundamental frequency estimation, note mapping, duration quantization and normalization, to the final CSV output. Although the `librosa` library [11] offers more advanced and robust audio processing capabilities, the compatibility limitations necessitated the use of Fast Fourier transform (FFT) through `numpy` and `scipy` [12]. This approach effectively meets the project's requirements, providing accurate pitch estimation and reliable note recognition suitable for the NAO robot's applications in humanoid robotic systems.

A. Audio Data Acquisition and Preprocessing

1) Subscribing to Robot Microphone Data: Audio data is continuously streamed from the robot's microphones by subscribing to the `ALAudioDevice` module [13]. The subscription specifies the sampling rate (16 kHz) and the number of channels (mono). Once subscribed, the robot's microphones send audio buffer data to the registered processing module in real-time. The initialization involves setting the sampling rate, number of channels, and module name, ensuring the correct configuration for subsequent data handling.

2) Conversion from 16-bit Little-Endian to Float32: The robot's internal audio data is transmitted in a 16-bit little-endian format. To facilitate numerical computations in Python, this data is converted to `float32` type. The conversion process involves reading each audio sample as a signed 16-bit integer, normalizing the values to the range $[-1, 1]$ by dividing by 32768.0, and casting the data to `float32`. This normalization is essential for accurate frequency analysis and efficient processing in subsequent steps.

B. Pitch Detection and Frequency Estimation

1) FFT Method: The project utilizes the Fast Fourier Transform (FFT) for real-time and robust pitch detection. The process involves the following steps:

1) **Audio Segment Selection:** A segment of audio data (e.g., 2 seconds) is extracted for analysis.
2) **FFT Execution:** The FFT is applied to the selected audio segment to obtain its frequency spectrum.
3) **Fundamental Frequency Identification:** Ignoring the zero-frequency (DC) component, the frequency with the maximum magnitude in the spectrum is identified as the fundamental frequency.

Mathematically, for an input audio signal $x[n]$ of length N, the FFT is defined as:

$$X[k] = \sum_{n=0}^{N-1} x[n] e^{-\frac{j2\pi kn}{N}}, \quad k = 0, 1, \ldots, \left\lfloor \frac{N}{2} \right\rfloor.$$

After computing the magnitude spectrum $|X[k]|$, the index k corresponding to the highest magnitude (excluding $k = 0$) is converted to its corresponding frequency f using:

$$f = \frac{k \cdot f_s}{N},$$

where f_s is the sampling frequency.

2) Note Recognition: To map the detected fundamental frequency to a musical note, the following logarithmic relationship is used:

$$\text{note\_number} = 69 + 12 \times \log_2 \left(\frac{f}{440\,\text{Hz}} \right),$$

where f is the fundamental frequency. The calculated MIDI note number is then rounded to the nearest integer and mapped to the corresponding note name from the sequence {C, C#, D, D#, E, F, F#, G, G#, A, A#, B}. The octave is determined based on the MIDI note number, providing a standard musical note representation [14] [15].

C. Duration Quantization and Normalization

1) Duration Quantization: To differentiate between notes of varying lengths, the duration of each detected note is quantized. When a valid pitch is sustained beyond a certain threshold, it is recognized as a note. If the pitch changes significantly or the amplitude drops below a threshold, the current note is terminated, and its duration is calculated. 6This duration is then mapped to predefined standard values such as whole notes (1.0 s), half notes (0.5 s), quarter notes (0.25 s), eighth notes (0.125 s), etc. The mapping is performed by selecting the standard duration closest to the actual measured duration.

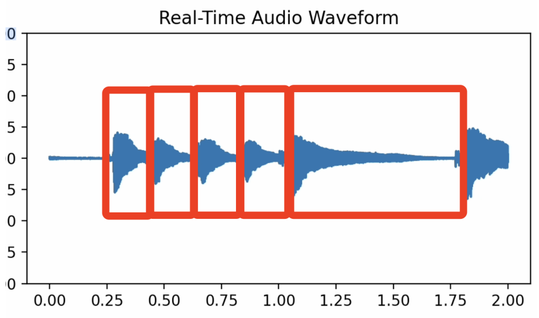

Fig. 6. Capturing the duration of notes

2) Duration Normalization: Considering the playback limitations of the NAO robot, the durations of the quantized notes are normalized to ensure consistency. The normalization process involves:

1) Identifying the shortest quantized note duration $\min_d$.

2) Scaling each note's duration d_i using the formula:

$$d_{\text{normalized}} = \frac{d_i}{\min_d} \times 0.5\,\text{s}.$$

This ensures that the shortest note duration is set to 0.5 seconds, and other durations are proportionally scaled, maintaining the relative timing between notes.

D. Result Output

1) CSV File: After detecting and quantifying the notes and their durations, the results are exported to a CSV file for further processing or playback. The CSV file contains two columns: the note name and its corresponding normalized duration. When a new recording session begins, the existing CSV file is overwritten to prevent data redundancy and ensure that only the latest session's data is retained. This facilitates easy integration with other modules that may require the note data for tasks such as music playback or analysis.

2) Data Queue: Real-time detected note data will be immediately inserted into the `note_queue`, serving as a reference for performance detection and verification.

E. Limitations of Pitch Detection in Xylophone-Based Systems

The pitch detection system for the xylophone is affected by:

1) Xylophone-Specific Issues: Toy xylophones cannot be precisely tuned, and each key exhibits varying degrees of pitch deviation. These factors make it difficult for the pitch detection system to accurately distinguish musical notes. Furthermore, xylophone bars do not produce pure sine waves but generate complex overtones (harmonics). The *Fast Fourier Transform (FFT) method* may mistakenly identify harmonics instead of the fundamental frequency. Additionally, the instrument's natural resonance causes frequency mixing, making it challenging to isolate the correct pitch.

2) NAO Microphone Hardware Issues: The frequency resolution of FFT-based pitch detection depends on the **buffer size and sampling rate**. The **16-bit PCM audio data** converted to **Float32 format** cannot compensate for **low-gain distortion and high-gain signal loss**. Furthermore, with a **fixed buffer size of 85.3125 ms**, the system operates with a **sampling rate of 16,000 Hz** and a **block length of 1365 samples**, leading to a frequency resolution of:

$$\frac{16,000\text{ Hz}}{1365\text{ samples}} + \text{Error (5Hz)} \approx 16.73\text{ Hz}$$

This resolution is insufficient to distinguish adjacent musical notes, particularly in the low-frequency range where pitch differences between notes are smaller. As a result, the system introduces ambiguity in pitch detection.

Given these limitations, to ensure smooth operation and prevent performance interference due to insufficient precision, the pitch detection function was not utilized in real-time feedback adjustments during performance. For listening to a melody and reproducing it, human intervention may be needed to correct some of the erroneous detections.

VI. Vision-Guided Grasping System

A. System Overview

The NAO humanoid robot implements a vision-guided grasping system (See Fig .7) for retrieving a mallet from a fixed holder. The system utilizes ArUco marker detection combined with precise coordinate transformations to achieve reliable grasping performance. The setup consists of a xylophone with an ArUco marker and mallet holder mounted at fixed relative positions.

B. ArUco Marker Detection and Pose Estimation

1) Stability-Based Pose Estimation: To ensure reliable transformation computation, the system first implements a sophisticated stability monitoring mechanism:

$$\Delta_{max-min} = \max_{w\in W} x_w - \min_{w\in W} x_w \leq \epsilon \tag{1}$$

where:

- W is the measurement window of size n (typically $n = 30$)
- x_w represents ArUco marker pose measurements
- ϵ is the stability threshold (typically 0.6)

Once stability is achieved, the system produces:

- Translation vector $\mathbf{t}^{opt} \in \mathbb{R}^3$ in the camera optical frame
- Rotation vector $\mathbf{r}^{opt} \in \mathbb{R}^3$ in the camera optical frame

C. Frame Transformation Chain

These stable pose estimates are then transformed through multiple reference frames to obtain the marker position in the robot's torso frame.

a) 1. Optical to Camera Frame Transform: The fixed transformation matrix $\mathbf{T}_{opt}^{cam}$ from optical frame to camera frame is:

$$\mathbf{T}_{opt}^{cam} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2}$$

b) 2. Camera to Torso Transform: The dynamic transformation matrix $\mathbf{T}_{cam}^{torso}$ is obtained through NAO's `motion_proxy.getTransform` function [16], which implements a real-time kinematic chain:

$$\mathbf{T}_{cam}^{torso} = \mathbf{T}_{neck}^{torso} \cdot \mathbf{T}_{head}^{neck} \cdot \mathbf{T}_{cam}^{head} \tag{3}$$

where each component represents:

- $\mathbf{T}_{cam}^{head}$: Transform from camera_bottom to head frame
- $\mathbf{T}_{head}^{neck}$: Transform from head to neck frame
- $\mathbf{T}_{neck}^{torso}$: Transform from neck to torso frame

Each transformation matrix is computed in real-time based on the joint angles. This hierarchical transformation chain ensures that the final camera position relative to the torso remains accurate even during head movements, as each component transform is updated in real-time based on the current joint angles.

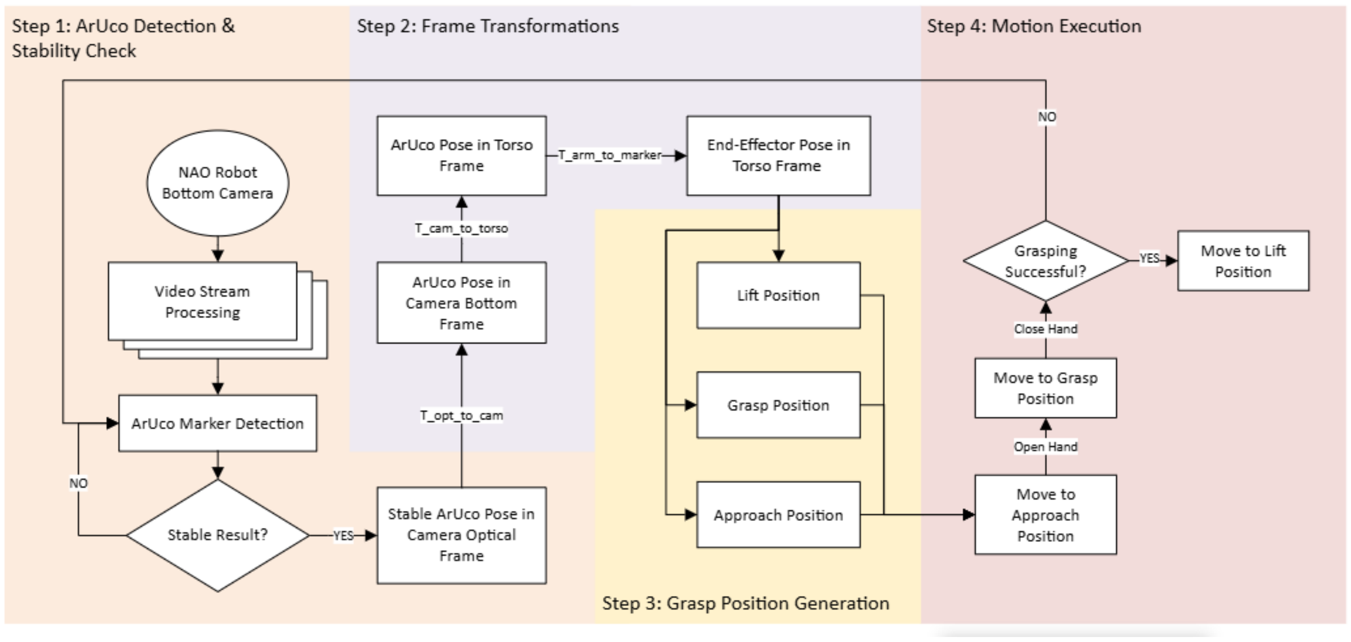

Fig. 7. Flowchart of the grasping task

c) 3. Complete Transform Chain: The complete transformation sequence is computed as:

$$\mathbf{T}^{torso}_{marker} = \mathbf{T}^{torso}_{cam} \cdot \mathbf{T}^{cam}_{opt} \cdot \mathbf{T}^{opt}_{marker} \tag{4}$$

where:

$$\mathbf{T}^{opt}_{marker} = \begin{bmatrix} \mathbf{R}(\mathbf{r}^{opt}) & \mathbf{t}^{opt} \\ \mathbf{0}_{1\times3} & 1 \end{bmatrix} \tag{5}$$

with $\mathbf{R}(\mathbf{r}^{opt})$ being the rotation matrix obtained through Rodrigues' formula.

D. Grasping Position Computation

1) Semi-Automatic End-Effector Calibration: The end-effector transformation relative to the ArUco marker was determined through a semi-automated script, `measure_ee_to_marker_euler.py`. In this approach, we loosen the stiffness of the right arm ("RArm") or left arm ("LArm"), manually place the corresponding hand to the desired grasping position near the ArUco marker, and let the NAO robot automatically retrieve:

- $\mathbf{T}^{\text{torso}}_{\text{arm}}$ via NAO's `motion_proxy.getTransform` function,
- $\mathbf{T}^{\text{torso}}_{\text{marker}}$ via ArUco detection and transformations to the torso frame.

We then compute

$$\mathbf{T}^{\text{marker}}_{\text{arm}} = \left(\mathbf{T}^{\text{torso}}_{\text{marker}}\right)^{-1} \cdot \mathbf{T}^{\text{torso}}_{\text{arm}}. \tag{6}$$

Hence, the user only needs to position the hand physically; the final transformation matrix is obtained automatically. This script is repeated for both the right hand and the left hand, allowing us to measure $\mathbf{T}^{\text{marker}}_{\text{rarm}}$ and $\mathbf{T}^{\text{marker}}_{\text{larm}}$.

The end-effector pose in the torso frame is then:

$$\mathbf{T}^{\text{torso}}_{\text{arm}} = \mathbf{T}^{\text{torso}}_{\text{marker}} \cdot \mathbf{T}^{\text{marker}}_{\text{arm}}. \tag{7}$$

From this transformation, we extract:

- Translation vector $\mathbf{t}^{\text{torso}}_{\text{arm}}$,
- Rotation matrix $\mathbf{R}^{\text{torso}}_{\text{arm}}$, converted to Euler angles $[\psi, \theta, \phi]$ in `xyz` order (roll about x, pitch about y, yaw about z) [17].

2) Grasping Position Generation: Using the computed end-effector pose in the torso frame, the system generates three key positions and orientations for the grasping sequence, with slight positional offsets for the right and left arms.

a) Right Arm:

$$\begin{aligned} \mathbf{p}_{\text{approach}} &= \mathbf{t}^{\text{torso}}_{\text{rarm}} + \begin{bmatrix} -0.04 \\ -0.04 \\ 0 \end{bmatrix}, \\ \mathbf{p}_{\text{grasp}} &= \mathbf{t}^{\text{torso}}_{\text{rarm}}, \\ \mathbf{p}_{\text{lift}} &= \mathbf{t}^{\text{torso}}_{\text{rarm}} + \begin{bmatrix} 0 \\ 0 \\ 0.08 \end{bmatrix}. \end{aligned} \tag{8}$$

b) Left Arm:

$$\begin{aligned} \mathbf{p}_{\text{approach}} &= \mathbf{t}^{\text{torso}}_{\text{larm}} + \begin{bmatrix} -0.04 \\ 0.04 \\ 0 \end{bmatrix}, \\ \mathbf{p}_{\text{grasp}} &= \mathbf{t}^{\text{torso}}_{\text{larm}}, \\ \mathbf{p}_{\text{lift}} &= \mathbf{t}^{\text{torso}}_{\text{larm}} + \begin{bmatrix} 0 \\ 0 \\ 0.08 \end{bmatrix}. \end{aligned} \tag{9}$$

The orientation remains constant throughout each sequence, maintaining the initial Euler angles $[\psi, \theta, \phi]$ computed for each arm. The complete pose for each position is formed by concatenating the position vector with the orientation angles.

E. Motion Control Implementation

1) Execution Sequence: The grasping motion follows a state machine pattern:

$$S = s_{approach}, s_{open}, s_{grasp}, s_{close}, s_{lift} \tag{10}$$

with transitions governed by:

$$T(s_i, s_{i+1}) = \begin{cases} 1 & \text{if } |\mathbf{p}_{\text{current}} - \mathbf{p}_{\text{target}}| \leq \delta, \\ 0 & \text{otherwise.} \end{cases} \tag{11}$$

2) Error Handling: Success verification through hand stiffness:

$$\text{grasp}_{\text{success}} = \begin{cases} 1 & \text{if } k_{\text{hand}} = 1.0, \\ 0 & \text{otherwise.} \end{cases} \tag{12}$$

where k_{hand} represents the hand stiffness parameter.

F. System Architecture

The implementation employs a multi-threaded architecture with:

- Thread τ_1: Marker detection and pose stability monitoring
- Thread τ_2: Grasp position computation
- Thread τ_3: Motion execution and error handling

Inter-thread communication is managed through:

$$Q = q_{path}, q_{status} \tag{13}$$

where q_{path} carries computed trajectories and q_{status} carries system state information.

G. Recovery Behavior

In case of grasp failure ($k_{hand} < 1.0$), the system implements a retry mechanism:

$$\text{retry}_{\text{condition}} = \begin{cases} \text{true} & \text{if } k_{\text{hand}} < 1.0 \land n_{\text{attempts}} < N_{\text{max}}, \\ \text{false} & \text{otherwise.} \end{cases} \tag{14}$$

where N_{max} is the maximum number of retry attempts.

VII. Vision-Based Xylophone Key Localization and Melody Play

VIII. Overview

To enable the NAO humanoid robot to play the xylophone accurately, this system localizes key positions using **ArUco markers**. The process begins with **stability detection**, followed by loading **relative positions (CSV)** and computing key locations using **3-Sphere LSQ optimization**. After applying the Transformation matrix to the key positions, we got the Arm positions and orientations. The results are converted into **Euler angles** for precise movement

The system then reads musical notes and determines the appropriate arm for striking. The robot **adjusts the wrist angle, moves to the key, and plays the note**, repeating until all notes are played. The complete process is illustrated in Fig. 8, ensuring accurate key localization and coordinated movement for melody execution.

A. System Setup and Marker Configuration

The localization system leverages three pre-calibrated ArUco markers (IDs: 11, 21, and 31) placed at fixed, known positions relative to the xylophone. These markers are detected using the NAO robot's bottom camera, with their poses transformed into the robot's torso frame using real-time kinematic data.

1) Marker Triangulation: Relying on a single ArUco marker for pose estimation can introduce significant errors due to potential misdetections or environmental factors. To address this, the system employs a robust multi-marker triangulation strategy using three ArUco markers. The relative positions of these markers with respect to known measurement points were predetermined, enabling precise computation of their positions within the torso frame of the robot. By detecting all three markers and applying triangulation, the system combines positional data from all markers, significantly reducing the impact of individual marker detection errors and ensuring reliable and precise key localization.

The corresponding implementation is encapsulated within the `ArucoDetector` class in the `aruco_marker.py` module. The `detect_and_transform` method processes the detected markers, computes their spatial relationships, and facilitates accurate localization within the robot's coordinate system. This approach enhances the overall reliability of the pose estimation process, even under challenging environmental conditions or single-marker occlusion.

B. Relative Position Mapping

Relative positions between the ArUco markers and the xylophone keys were predefined based on geometric measurements. Two sets of relative positions were defined:

- **Standard Keys**: These keys are located on the first row and are playable by the NAO robot.
- **Extra Keys**: Located on the second row, these keys are beyond the robot's reach but are included for completeness.

For both sets, incremental offsets along the x-axis and y-axis were applied to generate positions for multiple keys, ensuring consistent spacing and alignment.

C. Least-Squares Optimization for Absolute Positioning

To transform the relative positions of keys into the robot's torso frame, a least-squares optimization algorithm was used. This algorithm solves spherical equations representing the distance constraints between the detected ArUco marker positions and the xylophone keys.

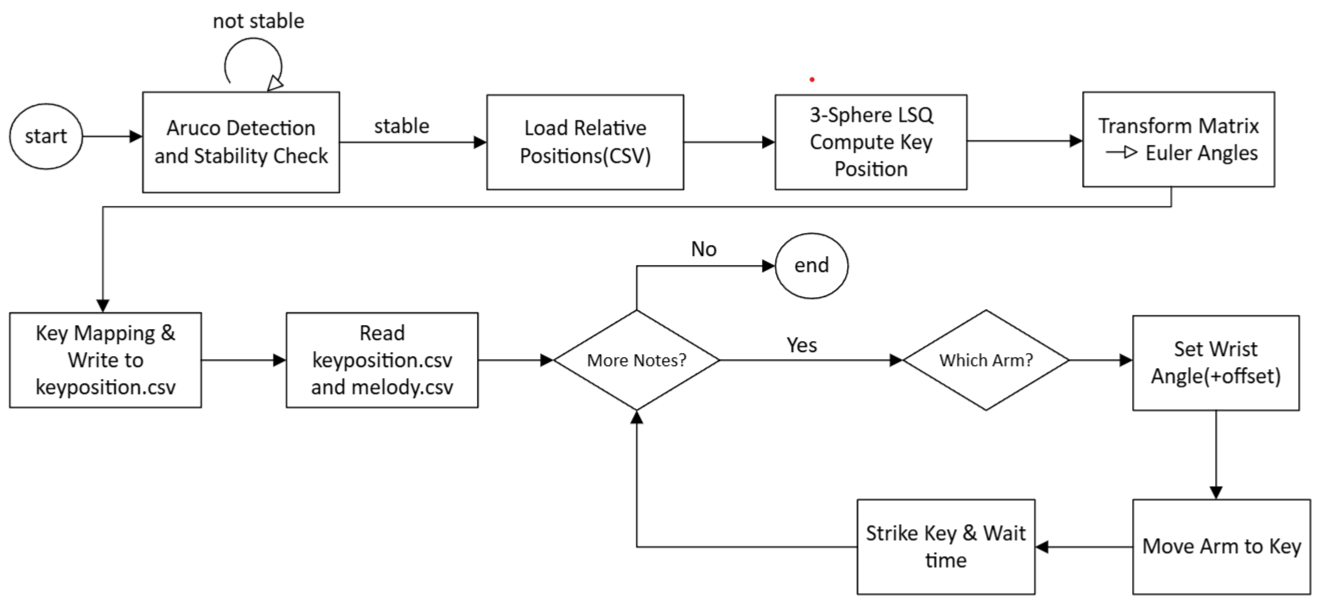
not stable
start
Aruco Detection and Stability Check
stable
Load Relative Positions(CSV)
3-Sphere LSQ Compute Key Position
Transform Matrix —▷ Euler Angles
Key Mapping & Write to keyposition.csv
Read keyposition.csv and melody.csv
More Notes?
No
end
Yes
Which Arm?
Set Wrist Angle(+offset)
Move Arm to Key
Strike Key & Wait time

Fig. 8. Flowchart of the key localization and melody play

1) Spherical Constraints: The constraints are defined as:

$$(x - x_i)^2 + (y - y_i)^2 + (z - z_i)^2 = d_i^2$$

where (x, y, z) represents the xylophone key's position, (x_i, y_i, z_i) are the positions of the ArUco markers, and d_i are the Euclidean distances computed from the relative positions.

2) Optimization Process: The SciPy library's `least_squares` function [18] was used to solve for the key positions. Initial guesses and marker positions served as inputs, and the algorithm iteratively refined the solutions to minimize positional errors.

D. Calculation of Hand Position and Orientation Relative to the Torso

After determining the positions of the xylophone keys, a series of rotation matrices and transformations are applied to calculate the hand's final position and orientation relative to the torso. This ensures the hand and stick can accurately strike the xylophone keys.

1) Key Parameters:

- **Stick Length**: The stick length is fixed at 18.8 cm, which defines the hand's position relative to the xylophone key.
- **Hand Assignment**: Keys 1 to 7 are played using the left hand, while keys 8 to 15 are played using the right hand.
- **Wrist Angle**: The wrist angle is set to -1.57 radians for the left hand and 1.57 radians for the right hand.

2) Rotation Matrix Calculation: For the right hand, rotation matrices are used to simulate the relative position and orientation of the hand with respect to the torso. The sequence of rotations, in Fig. 9, is as follows:

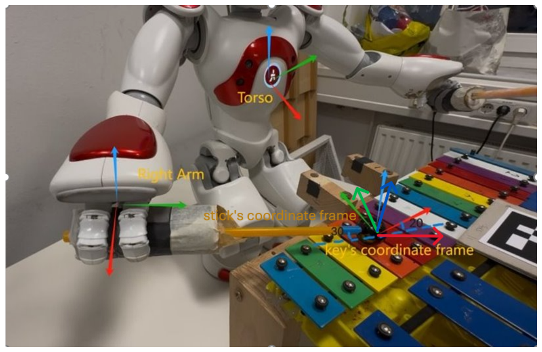

Fig. 9. The relation between 3 coordinate frame

1) **Rotate 90° around the torso's z-axis**: This transforms the key's position from the torso frame to the key's local coordinate frame.
2) **Rotate -20° around the z-axis of the key's coordinate frame**: This simulates the tilt of the stick during the strike, ensuring the stick is properly aligned.
3) **Rotate 30° around the y-axis of the key's coordinate frame**: This further adjusts the stick's tilt for optimal striking angles.
4) **Rotate 90° around the z-axis of the key's coordinate frame**: This determines the final orientation of the hand (Arm frame) to ensure proper alignment with the stick and the striking motion.

The total rotation matrix is expressed as:

$$R_{\text{total}} = R_{z1} \cdot R_{z2} \cdot R_y \cdot R_{z3}$$

where:

- R_{z1}: Rotation around the torso's z-axis.

- R_{z2} and R_y: Rotations in the key's local coordinate frame for tilt adjustment.
- R_{z3}: Rotation in the stick's coordinate frame to align the hand with the stick's direction. (The hand's coordinate frame's y-axis aligns with the stick's coordinate frame's x-axis.)

a) Hand Coordinate System Characteristics::

- The x-axis always points forward along the arm.
- The y-axis aligns with the stick's direction.
- The z-axis is perpendicular to the back of the hand and points outward.

3) Hand Position Calculation: The final position of the hand is calculated as:

$$\begin{bmatrix} x_s \\ y_s \\ z_s \end{bmatrix} = R_{z1} \cdot R_{z2} \cdot R_y \cdot \text{stick\_direction} + \begin{bmatrix} x_e \\ y_e \\ z_e \end{bmatrix}$$

where:

- stick_direction $= \begin{bmatrix} -L \\ 0 \\ 0 \end{bmatrix}$ represents the stick's direction in the stick's coordinate frame.
- $\begin{bmatrix} x_e, y_e, z_e \end{bmatrix}$ is the position of the xylophone key in the torso coordinate frame.

4) Hand Orientation Calculation: The orientation of the hand is obtained by extracting Euler angles from the total rotation matrix R_{total}:

- ϕ: Roll angle (rotation around the x-axis), representing the hand's lateral tilt.
- θ: Pitch angle (rotation around the y-axis), adjusting the forward-backward tilt of the stick.
- ψ: Yaw angle (rotation around the z-axis), determining the hand's horizontal rotation.

E. Key Mapping and Output

The computed key labels were mapped to musical notes (e.g., *Point_1* to *G7*) based on their sequence. The hand position, orientation and other parameters were then exported to a CSV file, providing a structured dataset for subsequent use in robotic arm movement planning and melody playback.

F. Melody Playback Process

1) Arm and Wrist Preparation: Before playing the melody, the robot's arm and wrist are positioned based on the data from the key position CSV file:

- **Arm Positioning**: The arms are moved to the starting positions corresponding to the keys `G6` and `G7` using the positional data stored in `key_dict`.
- **Wrist Initialization**: The wrist angles are pre-adjusted to a "pre-strike" position, allowing smooth and precise motion during the striking phase.

2) Melody Execution: The robot plays each note in the melody by sequentially performing the following steps:

- Retrieve the current note and its corresponding parameters (e.g., arm position, wrist angle, strike timing) from the melody CSV file.
- Determine which arm (left or right) is responsible for striking the key based on the `hand` parameter.
- Move the selected arm to the designated key position using `setPositions`.
- Adjust the wrist angle to the pre-strike position and strike the xylophone key by applying the computed `strike_wrist_angle`.
- Introduce precise delays between notes using the `lasting_time` and `time_to_strike` parameters to ensure correct timing and rhythm.

3) Post-strike Adjustment: After striking a key, the wrist is reset to its pre-strike position to prepare for the next note:

- This ensures smooth transitions between consecutive notes and prevents unnecessary strain on the robot's joints.
- The reset mechanism is implemented using the `setAngles` method.

G. Error Mitigation and Robustness

The system incorporates several strategies to ensure robust and accurate melody playback:

- **Multi-marker Triangulation**: Reduces potential errors in key localization by combining positional data from multiple markers.
- **Timing Synchronization**: Simulations under various conditions validate the accuracy and timing of the arm movements and key strikes.
- **Dynamic Arm Adjustment**: Adaptive control ensures that the arm positions are corrected in case of small misalignments during playback.

H. Output and Integration

The entire melody playback process is encapsulated in the `playMelodyFromCSV` function, which integrates:

- Arm and wrist control.
- Melody timing and rhythm synchronization.
- Structured logging and error reporting for debugging and performance analysis.

The combination of these elements allows the NAO robot to perform melodies on the xylophone with high precision and fluid motion, ensuring a seamless and robust performance.

SUMMARY

This research developed a NAO humanoid robotic system capable of playing the xylophone, showcasing integration of human-robot interaction, audio processing, motion control, and perception. The system achieves its objectives through the following key functionalities:

1. Human-Robot Interaction and Interface

- Supports both voice and touch control, enabling intuitive operation.
- Equipped with safety mechanisms such as emergency stop and command confirmation to ensure user and machine safety.

2. Autonomous Grasping and Performance

- Utilizes visual feedback and kinematic control to autonomously grasp xylophone mallets.
- Executes single-note playback accurately and reproduces melodies based on predefined sequences.

3. Melody Learning and Pitch Detection

- Implements real-time pitch detection using the FFT method on audio data from the robot's microphone.
- Extracts melodies from audio snippets and adjusts playback speed to accommodate mechanical constraints.

4. Vision-Guided Localization

- Employs ArUco markers for environmental localization and high-precision pose estimation.
- Enhances robustness through multi-marker triangulation and smoothing techniques.

5. System Architecture and Modularity

- Features a task-driven modular design, enabling efficient collaboration among components.
- Independent modules handle specific functions such as pitch detection, grasping, melody playback, and self-assessment.

By integrating multidisciplinary technologies, this research successfully demonstrates the autonomous behavior of robots in complex tasks and provides valuable references for the design of human-robot interaction and robotic systems.

References

[1] Aldebaran Documentation Team, "ALMemory - NAOqi Core Documentation," [Online]. Available: http://doc.aldebaran.com/2-1/naoqi/core/almemory.html

[2] Aldebaran Documentation Team, "ALSpeechRecognition - NAOqi Audio Documentation," [Online]. Available: http://doc.aldebaran.com/2-1/naoqi/audio/alspeechrecognition.html

[3] Aldebaran Documentation Team, "ALTextToSpeech API - NAOqi Audio Documentation," [Online]. Available: http://doc.aldebaran.com/2-1/naoqi/audio/altexttospeech-api.html

[4] E. D. Kafura, "Threads vs Events," Presentation for CS5204: Advanced Topics in Operating Systems, Virginia Tech, [Online]. Available: https://courses.cs.vt.edu/cs5204/fall09-kafura/Presentations/Threads-VS-Events.pdf

[5] R. Tsai, "A versatile camera calibration technique for high-accuracy 3D machine vision metrology using off-the-shelf TV cameras and lenses," *IEEE Journal on Robotics and Automation*, vol. 3, no. 4, pp. 323–344, 1987, doi: 10.1109/JRA.1987.1087109.

[6] OpenCV Documentation, "Camera Calibration and 3D Reconstruction (calib3d module)," [Online]. Available: https://docs.opencv.org/3.4/d9/d0c/group__calib3d.html

[7] V. Lepetit, F. Moreno, and P. Fua, "EPnP: an accurate $O(n)$ solution to the PnP problem," *International Journal of Computer Vision*, vol. 81, no. 2, pp. 155–166, Feb. 2009, doi: 10.1007/s11263-008-0152-6. [Online]. Available: http://hdl.handle.net/2117/10327.

[8] X.-S. Gao, X.-R. Hou, J. Tang, and H.-F. Cheng, "Complete solution classification for the perspective-three-point problem," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 25, no. 8, pp. 930–943, 2003, doi: 10.1109/TPAMI.2003.1217599.

[9] OpenCV Documentation, "ArUco Marker Detection," [Online]. Available: https://docs.opencv.org/3.4/d9/d6a/group__aruco.html#ga061ee5b694d30fa2258dd4f13dc98129

[10] OpenCV Documentation, "Perspective-n-Point (PnP) pose computation," [Online]. Available: https://docs.opencv.org/4.x/d5/d1f/calib3d_solvePnP.html. Accessed: Jan. 21, 2025.

[11] Librosa Documentation, "Librosa: Python Library for Audio and Music Analysis," [Online]. Available: https://librosa.org/doc/latest/index.html. Accessed: Jan. 21, 2025.

[12] Audio Apartment, "What Is Fourier Transform (FFT) in Audio?" [Online]. Available: https://audioapartment.com/techniques-and-performance/what-is-fourier/

[13] Aldebaran Documentation Team, "ALAudioDevice API - NAOqi Audio Module," [Online]. Available: http://doc.aldebaran.com/2-1/naoqi/audio/alaudiodevice-api.html

[14] Melodics Support, "Making Sense of MIDI Notes and Values," [Online]. Available: https://support.melodics.com/en/articles/9889452-making-sense-of-midi-notes-and-values

[15] Computer Music Resource, "MIDI Note/Key Number Chart," [Online]. Available: https://computermusicresource.com/midikeys.html

[16] Aldebaran Documentation Team, "Control Cartesian API - NAOqi Motion Module," [Online]. Available: http://doc.aldebaran.com/2-1/naoqi/motion/control-cartesian-api.html

[17] Aldebaran Documentation Team, "ALMath Library Overview," [Online]. Available: http://doc.aldebaran.com/2-8/ref/libalmath/overview.html

[18] SciPy Documentation, "scipy.optimize.lsq_linear: Solve a Linear Least-Squares Problem with Bound Constraints," [Online]. Available: https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.lsq_linear.html