MarimbaBot

Carl von Heyden[1], Yunlong Wang[2], Christian Salamut[3],
Imran Ibrahimli[4], Berk Gungor[5], Juliane Röscheisen[6], Florian Vahl[7] and Tom Schmolzi[8]
Technical Aspects of Multimodal Systems
Universität Hamburg
Hamburg, Germany
{carl.von.heyden[1], yunlong.wang[2], christian.salamut[3]}@studium.uni-hamburg.de,
{imran.ibrahimli[4], berk.gungor[5], juliane.roescheisen[6], florian.vahl[7], tom.schmolzi[8]}@uni-hamburg.de

***Abstract*— Our project focuses on the development of a robotic system capable of playing musical notes and chords on a marimba. The system integrates various technologies, including optical music recognition, audio command processing, and precise motion control. It employs a custom vision model to read musical notes from a whiteboard and utilizes voice recognition to interpret human commands, and finally evaluates the robot's performance with audio feedback. The project encompasses hardware components, including 3D-printed compliant mechanisms and servomotors, to mimic human-like mallet strikes. This multifaceted project demonstrates the fusion of robotics, music, computer vision, and audio processing, showcasing a robot capable of performing a complex musical instrument with precision. The entire project, including code and documentation, can be found on GitHub[1].**

Index Terms—robot, music, vision, speech

I. Introduction

Humans play various musical instruments. The automated operation of such traditional instruments by non-human players has been of interest for over 250 years. As an early example, *the musician* from Jaquet-Droz, is a mechanical android, that utilizes 10 fingers to manipulate a customized organ [12]. With the advancement of technology, the field of modern robotics emerged. Music-playing robots could be used to showcase different technologies. This could be done as part of a band to facilitate human-robot interaction. Alternatively, they can independently perform a song or a melody for entertainment or other use cases like music therapy. Normally, musicians use sheet music to communicate and document pieces. Reading notes and performing the sequence of notes on an instrument are key competencies of performers. A similar process can be executed by robots as well.

In our project, we created a system that utilizes a six-degree-of-freedom robot (Universal Robotics UR5) equipped with an additional mallet holder, as shown in Figure 1. This system employs a camera in conjunction with a self-trained optical music recognition neural network to perceive music pieces. After generating a suitable motion trajectory, the piece can be played on the marimba. It does so using standard mallets within a specially designed actuated compliant mallet holder. The actuated single degree of freedom allows for dynamic spacing between the two mallets. Voice commands control the actions of the robot and the played notes are evaluated on the fly to generate a playing accuracy score. A physics simulation environment was used as a digital twin in the development. The trajectory generation allows the robot to play pieces at differing speed and loudness configurations while preserving the relative timing of the notes. The fields of music knowledge, engineering, and computational intelligence were combined to design a music-playing robot.

[1]`https://github.com/UHHRobotics22-23/MarimbaBot`

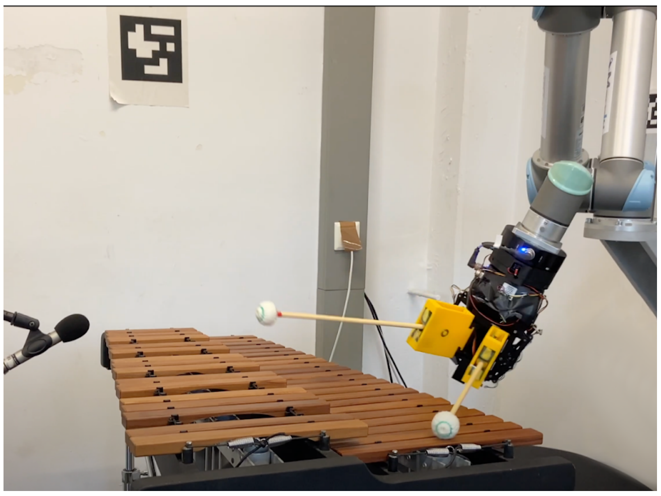


Fig. 1: The MarimbaBot playing the Marimba. The following components are visible: The Marimba, the UR5 robot arm, the mallet assembly, and the audio feedback microphone. The whiteboard, camera, and computer are not visible.

II. Related Work

In the field of robotics and dexterous manipulation, the pursuit of human-like dexterity in robotic hands has been a significant challenge. "ROBOPIANIST: Dexterous Piano Playing with Deep Reinforcement Learning" by Zakka et al. [14] is a notable and recent endeavor in this domain. This work explores controlling bi-manual anthropomorphic robotic hands in a simulation environment to play the piano, a task demanding high spatial and temporal precision, coordination, and planning. The resulting policies exhibit highly dexterous behaviors and produce visually and acoustically pleasing performances.

Of particular relevance to our project, is the dexterity exhibited by ROBOPIANIST. Our project focuses on a similar goal but distinguishes itself by using two mallets directly instead of hands, designed specifically for playing the marimba. This drastically reduces the dimensionality of the task. We also chose a non-learning-based technique for our motion and deployed it on a real robot platform. Moreover, our project incorporates speech recognition, vision input for interpreting music notes, and interaction with humans, demonstrating a comprehensive fusion of robotics, music, and human-machine interaction.

Exploration in the domain of marimba-playing robots has been conducted by Hoffman and Weinberg [2], who introduced "Shimon", a marimba player designed to collaborate and improvise with human musicians. Shimon is equipped with four arms affixed to a rail capable of traversing the marimba's length. Each arm is furnished with two mallets, one designated for "black" keys and the other for "white" keys. This design showcases an innovative method for enhancing marimba performance, a concept that resonates with our objective of using robotic arms with two mallets explicitly tailored for marimba play. Our project extends beyond conventional marimba playing, integrating speech recognition, vision input for interpreting musical notes, and interactions with human collaborators, ultimately exemplifying a holistic fusion of robotics, music, and human-machine interaction. Moreover, our project is not only more human-like but also more general-purpose, built using a multi-purpose robot arm.

III. Project Goal

The project aimed to create a robotic system capable of visually interpreting musical notes from a whiteboard and using an industrial robot arm to play them on a marimba with two mallets. Additionally, the system was designed for voice control, enabling natural human interaction.

IV. Implementation

The Implementation of the MarimbaBot can be separated into seven different chapters music audio processing, command recognition, vision, motion, simulation, behavior, and hardware. The whole system is controlled by voice commands, therefore the implementation of an audio setup was crucial for controlling the system. The command recognition and motion are connected via the behavior state machine. With a voice command, the visual part of the framework is activated. The Camera captures an image of the notes, which are subsequently extracted via a neural network. In the motion, the trajectories for hitting the right notes at the right time with the two mallets are calculated. During development, we utilized a physics simulation setup. In the end, the trajectories are executed using a UR5 robot and the 3D-printed mallet assembly.

A. Vision

The MarimbaBot should be capable of visually reading the notes, as defined in our project goals. Notes are presented to the robot on a whiteboard. They can be represented by black note-shaped magnets or drawn manually using a whiteboard marker. A webcam (*Logitech StreamCam*) is capturing the whiteboard content for further processing. After preliminary experiments with existing OMR (Optical Music Recognition) solutions, we decided to develop a custom end-to-end neural network solution. Previous OMR techniques (like Mozart[2] or Oemer [13]) largely rely on handcrafted logic to visually parse the image even if they utilize some learning based techniques. In contrast to that, our method bases itself on recent advances in image-to-text models, mainly used for image captioning or visual document understanding. We train a model to predict a textual representation of the music directly from the given image. As a representation, a subset of the LilyPond [9] syntax is used. Normally, LilyPond is used for rendering music notation based on a markup language. We train the model to perform the inverse of the rendering operation.

In the beginning, a minimal setup for testing purposes was created. Limiting the key to C major, not including chords, dynamics, and many other things allowed us to see if the approach is viable and detect issues without directly solving a too complex problem.

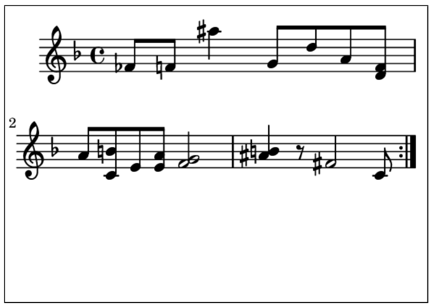
After solving said task sufficiently well, the notation was extended to include other keys, dynamics, chords as well and articulations.

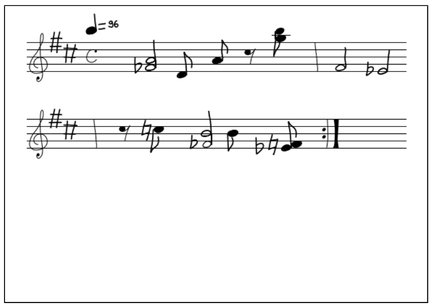
1) Dataset: Using the LilyPond [9] software for generating artificial training samples aligns with our choice of the LilyPond markup language as the target format for our model output. This approach facilitates the straightforward creation of an extensive dataset comprising synthetic samples. The generation of these samples originates from random LilyPond notations, distinct from any pre-existing musical compositions, ensuring precise control over the data distribution. This level of control empowers the efficient coverage of edge cases, essential in music-related applications.

The data generation process outlined in this study capitalizes on a Python script harnessing the capabilities of the Abjad[3] library to construct synthetic music scores in LilyPond format. The process commences with script configuration, granting users the ability to define various parameters, such as the number of samples and the minimum note durations.

To execute this, we initiate the generation by randomly sampling sets of bars comprising notes, chords (two notes), and rests. Subsequently, these musical elements are consolidated into a LilyPond string, which is further subjected to random modifications based on the pre-defined configuration. This configuration allows for the inclusion or exclusion of several musical components, including tempo, slurs, dynamics, articulations, repeats, and major or minor scales. Following these operations, the resulting data is stored in the form of LilyPond ".ly" files, rendered PNG images, and

[2]https://github.com/aashrafh/Mozart
[3]https://github.com/Abjad/abjad


(a) A randomly generated sample rendered using LilyPond.


(b) A randomly generated sample using the handwritten style rendering.

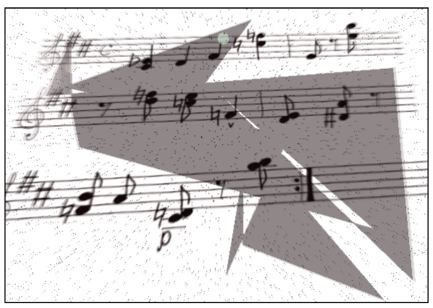

(c) Sample with random data augmentation applied to it.

Fig. 2: Different artificial dataset samples.

textual LilyPond notation strings, serving as ground truth references.

The incorporation of multiprocessing capabilities within the script expedites dataset generation by enabling parallel processing. Significantly, the script ensures the integrity of the generated samples, thereby averting redundancy. Achieved through judicious use of random sampling, this data generation process harmonizes flexibility with efficiency, culminating in substantial datasets relevant to diverse music-oriented applications. A representative generated sample is illustrated in Figure 2a.

However, using only LilyPond for data generation has several drawbacks. First of all, some symbols significantly differ between the whiteboard magnets and the font utilized by LilyPond. For example, successive eighth notes in LilyPond's font are always represented with a connected stroke as seen in Figure 2a, while our whiteboard magnets only provide the notes as individual symbols with associated individual flags as seen in Figure 4. Both variants are valid musical notations, but the differences can lead to erroneous or ambiguous recognitions in a vision model. There are also concerns regarding more subtle overfitting on the exact shape of the notes in LilyPond. This is especially relevant if some of the notation on the whiteboard is manually drawn.

To approach this issue, a custom rendering engine supporting our subset of the features present in LilyPond was created. It samples each symbol from a number of different hand-drawn ones, creating a diverse dataset. Our handwritten notation consists of 26 separate symbol classes, which are used complementary to construct the final result. In the case of notes, for example, we differentiate between note head and note stem in order to construct complex note structures, while using only a small number of symbol classes consisting of four to six hand-drawn PNG images. All 110 symbol images are displayed in Figure 3. Additionally, the horizontal distance between symbols and the vertical distance between staff lines can be randomized by defining minimal and maximum vertical and horizontal distances. Like the LilyPond render, our handwritten script also incorporates basic music notation rules, such as bar separation, dot placement, ledger lines, stem and flag direction, and adaptation of accidentals affected by key signatures or previous accidentals within the same bar. A sample created with this renderer can be seen in Figure 2b. Both presented methods to render the notation without any perspective or lighting changes. To account for that, a number of data augmentations are applied to both data sources. These augmentations include:

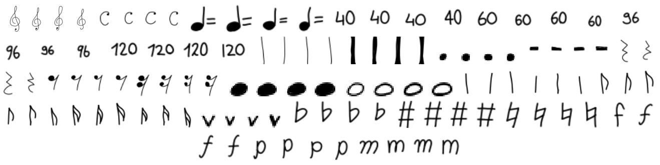


Fig. 3: The full symbol set for the automated handwritten data generation consists of 110 handwritten images.

- Affine transformations
- Perspective transformations
- Brightness changes
- Contrast changes
- ”Shadow”
- ”Sun flare”
- Pixel dropout
- Color shift
- Zoom blur

An augmented sample can be seen in Figure 2c. The synthetic part of the dataset consists of 200000 of these samples, evenly split between the LilyPond and ”handwritten” rendering. To further improve the domain shift into the real-world a small set of annotated real-world images is added. Due to the high effort required for the manual real-world data collection, this set includes only 127 distinct pieces. Each piece is captured from multiple angles, resulting in 608 samples. Capturing the same sequence too many times could result in overfitting on the sequence of notes, so we're capturing a maximum of five samples for each piece. We also include a number of real-world negative samples, to avoid hallucinations in cases where no notes are present on the whiteboard.

2) Model: The model itself is based on the Donut [3] model. We use the provided model as a starting point but perform a few major modifications. Mainly, we noticed that switching the model's tokenizer to a custom one improved the overall performance. The original model comes with a tokenizer that was optimized for natural language tokenization, but LilyPond data benefits from character-level tokenization.

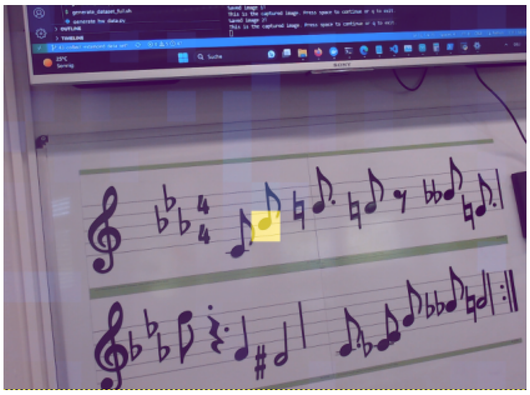


Fig. 4: Real-world whiteboard sample with cross attention map overlay, showing the token in the vision transformer that was most influential for the generation of the `g` token.

Here, we illustrate a tokenization of the following LilyPond string:

- String: `b'8 b'4`
- Normal Tokenizer: `b | ' | 8 | b | '4`
- New Tokenizer: `b | ' | 8 | b | ' | 4`

The normal tokenizer combines the `'` denoting the octave and `4` denoting the note's length. Due to patterns learned during the training on natural language, this combination is not consistent across notes and other notes might be split differently. This makes it harder for the model to disentangle these concepts (e.g. the octave and duration), as sometimes both are predicted at once in an inconsistent manner. Using character-level tokenization fixes this issue, but it also increases the resources needed to represent longer keywords like `\tempo`, so they are explicitly added to the tokenizer.

The vision encoder of the Donut model is based on the Swin [8] transformer [11] architecture. A BART [7] transformer [11] decoder is used to generate the textual representation by querying features from the vision backbone via cross-attention. A visualization of the cross attention for a given token can be seen in Figure 4. Overall, the model has 143 million trainable parameters. During training, a pure cross-entropy classification loss is applied for the next token prediction. No handcrafted domain-specific loss function is utilized.

B. Command Recognition

One main objective of the robot's functionality is to adeptly receive and process human commands conveyed through speech, thus enabling users to interact with the robot naturally and facilitating reciprocal communication.

In the operational pipeline of this subsystem, the initial phase involves keyword spotting to identify and activate upon detecting the designated keyword signals. Subsequently, human voice activity detection tools are deployed to precisely determine the end of speech. Following this, the whisper [10] model is utilized to transcribe the spoken content into text format. Finally, the system undertakes the crucial task of command extraction from the transcribed text, after which the extracted command is executed by the robot.

Therefore, the command recognition subsystem encompasses four core components:

- Keyword spotting
- Voice activity detection
- Speech transcription
- Command extraction

1) Keyword spotting: The keyword spotting is based on this GitHub Repository[4]. It is a lightweight, RNN-based signal classifier. Basically, it first uses sliding windows on the chunks of raw data to extract the Mel Frequency Cepstral Coefficients (MFCC) features, and then the recurrent neuron network will be used for signal classification to figure out, if data chunks belong to the keyword speech signal or not. The dataset of keyword spotting is collected manually, it consists of a wake-word set and a not-wake-word set. And the dataset of background noise, Public Domain Sounds Backup[5], is involved, as a part of the not-wake-word dataset.

2) Voice activity detection: In conjunction with keyword spotting, voice activity detection plays a pivotal role. Its primary purpose is the identification of speech termination within the audio stream. This feature greatly assists in segmenting the sentence of speech accurately, since our speech transcription model is context-dependent.

3) Speech transcription: Subsequent to voice activity detection, the subsystem employs a multilingual speech recognition model, which is the Whisper [10] model, to transcribe spoken language into textual form. This transformation ensures the subsequent processing of spoken commands in text format. The Whisper architecture presents a straightforward end-to-end methodology, employing an encoder-decoder Transformer model. Initially, input audio is divided into 30-second segments, which are subsequently transformed into log-Mel spectrograms and processed through an encoder. A decoder is trained to predict text captions corresponding to the audio. With the incorporation of special tokens, the model can execute multiple tasks, including handling multilingual speech transcription, speech-to-English translation, and so on.

4) Command extraction: The final stage involves the extraction of the essential command from the transcribed text. This phase entails regular expression to extract actionable commands effectively.

C. Behavior

The integration of the various modules is important for seamless and efficient task execution. Therefore, we implemented a *Behavior* module that acts as the central decision-making unit that orchestrates the interaction between the different modules of the system. Figure 5 showcases the communication between the Behavior module and the other four main modules within the system.

[4]`https://github.com/MycroftAI/mycroft-precise`
[5]`http://pdsounds.tuxfamily.org/`

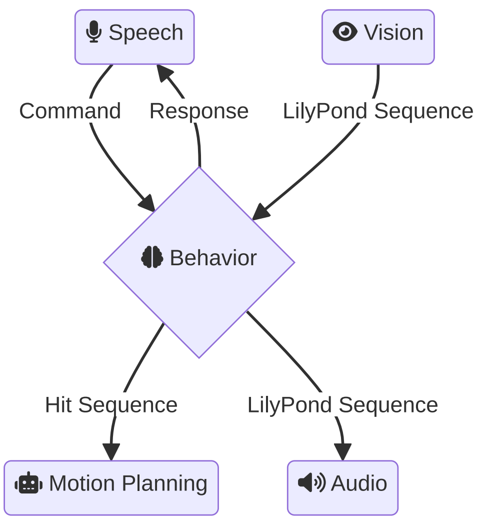


Fig. 5: The communication flow between the Behavior node and the MarimbaBot system.

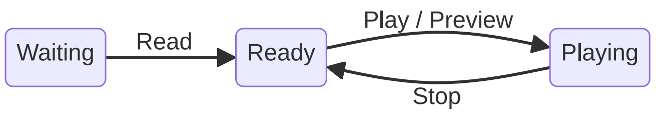


Fig. 6: The three states of the Behavior module.

1) Commands: The recognized commands consist of three fields; 'Command', 'Action', and 'Parameter'. There are four possible values for the 'Command' field; 'Read', 'Play', 'Preview', and 'Stop'. Actions are only available for the 'Play' and 'Preview' commands and include increasing, decreasing, or specifying the speed or volume, and the option to repeat the performance in a loop. Further, all actions besides looping can also be provided with a 'Parameter' value, defining to what extent the action is to be executed.

2) States and state transitions: Upon receiving a command from the Speech module, the Behavior module analyzes the context and formulates a response strategy, depending on the current state. There are three main states of the Behavior: 'Waiting', 'Ready', and 'Playing'. The three states and the state transitions using the appropriate 'Command' values can be seen in Figure 6.

Initially, the system starts in the 'Waiting' state. Once the 'Read' command is issued, the latest recognized LilyPond sequence from the Vision module is forwarded to an *interpreter* script based on the LilyPond parser provided by Abjad[6] and the MIDI handling tool pretty_midi[7], which translates the LilyPond sequence into a *Hit Sequence*. The Hit Sequence breaks down the LilyPond notation to only the audible tones and assigns each tone the corresponding start-time in relation to the entire piece, which starts at 0, as well as the duration and volume level of the tone. This format is crucial for the interaction between the Behavior module and the Motion Planning module, facilitating the generation of accurate strike timings for the marimba bars. If this translation fails due to the received sequence not being in parse-able format, a response message is sent to the Text-to-Speech module and we stay in the 'Waiting' state. However, if the translation was successful, we move to the 'Ready' state. The Behavior now knows a playable LilyPond sequence which we call the *active LilyPond sequence*, and a corresponding Hit Sequence. Every time the active LilyPond sequence is modified, a new Hit Sequence is calculated in response.

In the 'Ready' state, the commands 'Play' and 'Preview' can now be carried out. The 'Play' command will forward the current Hit Sequence to the Motion Planning, where a sufficient trajectory is calculated for our robot to play the piece on the marimba. Similarly, the 'Preview' command will forward the active LilyPond sequence to an Audio module, which plays the sequence as MIDI on the connected speaker. Now in the 'Playing' state, the system can still recognize and process commands, however, giving another 'Play' command while still in the 'Playing' state will cause the system to send a response to the Text-to-Speech module, notifying the user that the current play or preview process needs to be aborted first via the 'Stop' command. The 'Stop' command aborts all current operations and moves the system back to the 'Ready' state. There is, however, one exception in which the system can execute 'Play' / 'Preview' commands while still in the 'Playing' state; If the 'Playing' state was entered using a 'loop' action, further actions that request tempo or volume adjustments will be handled without responding to the user in the way described earlier. The changes will then be applied in the next iteration of the playing loop.

Actions are handled by first adjusting the active LilyPond sequence to fit the request, and then executing the main 'Command' value. In case of looping, the base command execution will be repeated once the preview or motion plan is finished, and can only be ceased using the 'Stop' command.

3) Speech response: In the event that the MarimbaBot system cannot perform a command due to constraints, technical issues, or other factors, the Behavior module ensures a swift and informative response. This response is relayed to the Text-to-Speech module, which communicates the status to the human operator. For example, if a command like "Play faster" cannot be executed due to physical limitations, the Behavior Module informs the operator of this. Similarly, if a command was successfully executed, but the execute does not produce audible feedback such as in the case of the 'Read' command, a response notifying the user of the successful state transition is forwarded to the Text-to-Speech module.

D. Motion

Our task requires robust and dynamic motion skills, that allow the robot to strike the correct keys on the marimba at the right point in time.

We utilize a standard MoveIt-based ROS motion planning setup to calculate and execute trajectories according to the music notes and timing information. The motion planning interfaces with the rest of our software stack by providing a ROS action server.

For every note we want to hit, a goal position is queried from the robot model using the TF2 [1] forward kinematics

[6]`https://github.com/Abjad/abjad`
[7]`https://github.com/craffel/pretty-midi`

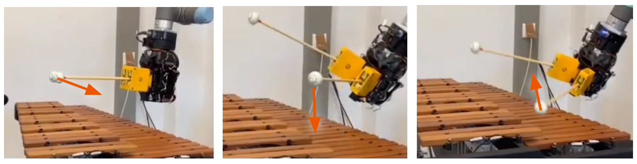

(a) (b) (c)

Fig. 7: Different key points during the hit motion.

library. The height of the goal position is slightly offset using a parameter allowing for adjustments needed with the flexible mallet setup. To generate the trajectory that needs to be performed to hit a given note, we divide the problem into three phases. **In the approach phase**, the robot is moved from its home position or a previous note's retreat position to the new note's approach point. **In the hit phase**, the mallet tip approaches the goal position, starting at an approach point. Located above the hit point, the approach point can also be adjusted using parameter offsets. **The retreat phase**, starts out at the hit point and moves the mallet back to a retreat point similar to the approach point.

We solve the robot's inverse kinematics at each phase transition using the BioIK[8]. The transition points for the hit motion can be seen in Figure 7. Interpolation during the phase is done in joint space using the Pilz Industrial Motion Planner[9]. After adjusting the timing in each phase to match the speeds required to match the required melody or loudness, the trajectories are concatenated and sent to the robot controller.

To play chords, we search for notes that need to be played at the same time. We assign the left note to the left mallet and the right note to the right mallet. The hit motion now includes two approaches, hit and retreat points.

Using a trajectory optimization approach, considering joint space as well as Cartesian space constraints would lead to a more optimal trajectory, allowing for faster and more natural playing. However, after some preliminary experiments, this direction was abandoned in favor of the simpler approach described earlier. We implemented another approach for faster two-mallet playing, which generates a trajectory for each mallet in Cartesian space, which is later translated into joint space. While theoretically allowing for more complex behaviors, like playing two notes in fast succession with different mallets. The approach was subject to many edge cases where e.g. joint space contains might be violated and was therefore abandoned after some testing.

1) Inverse Kinematics: We employ BioIK as our inverse kinematic solver. It is able to search for a valid robot joint configuration that satisfies a number of constraints. Due to our expanded objectives, analytical inverse kinematics solutions, typically applicable to the UR5 [5], are no longer realistic. We therefore use a numerical method like BioIK. To move a single or both mallets to a goal position, the following goals are considered in the optimization:

- Position Goals: The goal position of one or both mallet heads.
- Link on Plane of wrist: We want to keep the robot's wrist on a horizontal plane above the marimba. This way the robot is discouraged from hitting with the whole arm and uses rotations of the wrist instead.
- Link on Plane of right mallet: We keep the right mallet above the left mallet if we only play with one of them (the left is the default). The distance is configurable and needed to prevent the right mallet from accidentally hitting the marimba.
- Joint Goal: We want to keep the robot's wrist 3 joint fixed at zero if only one mallet is utilized. This makes the robot less overactuated and results in more consistent solutions.
- Joint Goal: To move the right mallet out of the way during single mallet strokes, the mallet finger joint can be fixed to 70 degrees.
- Minimal Displacement Goal: For more consistent solutions a minimal displacement goal relative to the robot's home position (idle position above the marimba) is applied.

2) Timing: Ensuring that the robot hits the marimba bars at the right points in time is crucial for an accurate performance. The timing of the robot's trajectory also influences how hard or soft a given key is hit. In order to manage this process, the speed of each of the sub-trajectories is adjusted. Based on the commanded volume, the hit trajectories duration are adjusted. The duration of the approach trajectory is also altered based on the commanded timing. Here we also need to consider that the hit itself isn't instant, so the durations of the hit trajectories count towards the approach times. The robot only slows down the plans from the Pilz Industrial Motion Planner, which is configured to procure the fastest possible plan. Speeding them up will exceed the robot's acceleration or velocity limits, so instead the approach time is clamped and a warning is issued. The robot could also reject the goal, but we explicitly opted for a best-effort behavior. Interpolation is applied to keep the trajectory at a certain density after significant slowdowns.

In a more formal manner, the timing can be expressed by

$$\hat{t}_d = t_d + (1 - l) \cdot t_{max}$$
$$\hat{t}_a = max(\Delta t_n - \hat{t}_d - t_r, t_a)$$

where $\hat{t}_d$ denotes the duration of the down stroke trajectory, $\hat{t}_a$ represents the duration of the approach. t_d, t_a, and t_r represent the durations of the initially planned trajectories, which serve as a lower bound of the trajectory duration. Δt_n represents the time span between the previous and subsequent note, while l describes the goal loudness in the interval $[0, 1]$. The down stroke duration extension of the most silent note is defined by t_{max}.

In the end, the full trajectory is executed via MoveIt, which executes it on the robot using a JointTrajectoryController.

E. Simulation

The simulated environment consists of the UR5 arm, the compliant double mallet, and the model of the marimba. All of the objects in the scene are modeled in the Unified

[8]https://github.com/TAMS-Group/bio_ik
[9]https://github.com/PilzDE/pilz_industrial_motion

Robotics Description Format (URDF). The Gazebo physics engine [6] is used to simulate forces, contacts, as well as other physical aspects of the environment. The URDF files also include the `<inertial>` blocks that specify the inertial properties for each link, and `<transmission>` blocks for joints to allow control in Gazebo. The joints use a transmission with EffortJointInterface as a hardware interface.

The URDF description of the marimba is modified in the simulated environment to allow detection of bar hits: the bars are considered mounted on a prismatic joint instead of the usual fixed one, to enable detection of contact velocities. The prismatic joints have a range of ± 0.1 cm and the bar is considered hit if the joint moves with a velocity above a certain threshold value (0.5 cm/s).

In addition to the aforementioned changes, the simulation also includes PID values for all of the actuated joints, since unlike the real setup, we do not have the real robot controller and must therefore run separate PID controllers. Some simplifying assumptions are made about the physical properties, such as assuming diagonal inertia matrices.

The simulation is tightly integrated with the other components of the project: the planning node is launched at the start of the simulation and controls the UR5 and the double mallet attachment; the URDF descriptions are shared with RViz visualization in the real robot setup. Controlling the robot is done through the same interface as the real setup, i.e. using the action server and the planning node. While the simulation is running, the environment is visualized in the Gazebo GUI, RViz with the MoveIt motion planning plugin, as well as optionally launching PlotJuggler[10] to plot the bar joint velocities (and therefore the raw values used to detect contacts) in real-time. Another node is running to monitor the bar joint velocities, perform contact detection, and write the hits to a MIDI file at the end of the simulation.

F. Hardware

The hardware assembly allows for the controlled and compliant usage of two mallets in the UR5's three-finger gripper [11].

1) Design and 3D-Printing: The ergonomics of the human hand are used as a model for the development of the 3D model. This 3D Model, which can also be found on GitHub[12], full-fills two requirements. Firstly a compliant mechanism to create proper sound on the marimba. Secondly an assembly that could hold two mallets and vary the distance between the two mallet heads. The mallet is mounted in a mallet-car and secured using screws. This mallet-car is attached to the housing by an axle to allow the mallet to swing in the vertical direction. To limit compliance, strong rubber bands and a shock absorbing sponge are attached to the mallet-car and housing. By the choice of the rubber bands, one can choose the degree of compliance. The intention is to imitate the human hand as it holds the mallets loosely and allows them to swing in the hand. Two of these assemblies are attached to the main housing with one of the attachment points being actuated by a servo while the other remains rigid. Using the servo joint the distance of the two mallet can be adjusted. The connection was designed as a quick-exchange plate to allow different mallet-housings to be mounted on the main body.

2) Electronic setup: The electronic setup was designed to fulfill the following requirements:

- independent power supply
- compact design
- wireless control
- reliability

Independent power is provided by a 7.4 volt, 2200 mAh battery. The use of a Raspberry Pico W with integrated wifi enables a compact design. By using the wifi as a communication interface, wireless and fast control is made possible. The components have been securely soldered and installed inside the case to ensure the reliability of the components. For a detailed overview of the circuit see the hardware documentation in the repository [12].

3) Software: Communication between the hardware extension and ROS takes place via a wifi-based UDP connection. The UDP connection ensures that the data is transferred quickly. The transmitted informatio can be separated into the classes limits, current position, and position commands. The protocol used for the communication between ROS and the microcontroller is based on simple string commands. At first, the ROS side is expected to request the bounds and resolution using the "l" command. This allows for multiple devices with different bounds to be used interchangeably. The "p" command is used in the read phase of the hardware interface. Lastly, the "s" command allows the ROS side to give an integer-valued command position to the microcontroller. A detailed description is contained in the hardware repository [12].

G. Music Note Detection

Evaluating the robot's marimba-playing performance is vital. To aid in this assessment, a dedicated submodule has been developed to evaluate the robot's music using audio feedback. This submodule detects Western musical notes from raw audio, providing visual representations in MIDI figures and the Constant-Q Transform (CQT) spectrum. Additionally, it offers a recall score to assess overall motion quality.

In the subsystem's pipeline, raw data first undergoes Constant-Q Transform (CQT) to create a spectrogram. CQT is a time-frequency representation with evenly spaced frequency bins on a logarithmic scale, ensuring consistent ratios of center frequencies to bandwidths (Q-factors) across all bins. Next, the amplitude spectrogram is converted into a dB-scaled version, which serves for computing the spectral flux onset strength envelope. Following this step, onset events are localized using the peak-pick method. To determine the pitch of these onset events, a monophonic pitch classifier

[10]`https://github.com/facontidavide/PlotJuggler`

[11]`https://robotiq.com/products/3-finger-adaptive-robot-gripper`

[12]`https://github.com/UHHRobotics22-23/hardware_design`

called CREPE [4], which operates directly on the CQT spectrum using Convolutional Neural Networks (CNN), is employed. Lastly, we compare the detected musical notes with ground truth provided by vision recognition to calculate the recall score, offering a quantified measurement of the robotic motion.

V. Results

In the following Section, we highlight some evaluations of the different subsystems presented in Section IV.

A. Vision Evaluation

For the evaluation of our vision pipeline we consider three different models. All of them use the same architecture, but their training data and tokenizer differ. A detailed comparison against a third-party baseline is not performed. We tried to include both Mozart[13] and Oemer [13] as our baseline, but both had runtime errors on all presented whiteboard samples. Also, the long runtime of 3-5 minutes in the case of Oemer ruled it out for our interactive use case. The first model we want to evaluate is the *Base* model, which is one of our first models. It is trained on a reduced set of notation (C Major, no dynamics and accents, no chords) and serves as a proof of concept. Training it on extended LilyPond data resulted in the *Extended* model. The next iteration featured a character-level tokenization as well as an extended handwritten dataset. It is called *Extended_Char*.

TABLE I: Edit distance on real whiteboard data

Model	Whiteboard_Extended	Whiteboar_Basic
Extended_Char	$\mathbf{0.1 \pm 0.3}$	0.57 ± 0.92
Extended	17.9 ± 4.76	1.1 ± 0.54
Base	67.6 ± 3.98	$\mathbf{0.24 \pm 0.42}$

TABLE II: Edit Distance on generated handwritten notation

Model	Handwritten_Extended	Handwritten_Basic
Extended_Char	$\mathbf{2.41 \pm 6.34}$	$\mathbf{0.045 \pm 0.45}$
Extended	50.92 ± 17.12	1.56 ± 1.54
Base	64.16 ± 21.38	0.845 ± 1.44

TABLE III: Edit Distance on data generated using LilyPond

Model	LilyPond_Extended	LilyPond_Basic
Extended_Char	$\mathbf{0.735 \pm 0.735}$	$\mathbf{0.04 \pm 0.4}$
Extended	2.77 ± 5.62	0.765 ± 0.79
Base	59.88 ± 20.64	0.12 ± 0.61

All of these models were evaluated on six different test datasets. Table I shows the evaluation on data collected in the real-world on the whiteboard. It is split into two datasets, one containing basic C Major notation and one which also includes the extended notation. As expected, the *Base* model which is not trained on the extended notation fails to predict the extended notation. Also, due to the high manual effort involved with the real-world data collection the size of the extended dataset is very limited. It only includes 10 images, while the basic one contains 30. So we need to consider this part of the evaluation as less reliable, as many edge cases are not covered in the data. Still, similarly to the other datasets, we are able to observe a significant improvement with the *Extended_Char* model for the extended notation. Interestingly, it is slightly outperformed by the *Base* model for the basic notation. This could be the case because the number of basic notation real-world training samples takes up a larger fraction of the training data if no extended notation is present in the artificial dataset.

[13]`https://github.com/aashrafh/Mozart`

In Table II the models are evaluated on test data which was generated using the handwritten style data generation. The datasets include 200 samples each. We can see that the *Extended_Char* model also scores very well in the evaluation. It can also be observed, that the *Extended* model fails to generalize to the handwritten notation from extended LilyPond data alone.

The *Extended_Char* model version also leads the evaluation on data which was generated using LilyPond (see Table III). Similar to the handwritten test datasets, the LilyPond ones also include 200 samples each. As in all other cases, the *Base* model fails on the extended notation. While not performing best, the performance of the *Extended* model is significantly better on the LilyPond data.

As our evaluation metric, we utilized the edit distance (Levenshtein distance) of the predicted LilyPond strings.

Some limitations of our vision approach include:

- The data includes only a maximum of 3 bars, so the model is not capable of processing pieces much longer than that reliably.
- The models space and compute complexity concerning the output sequence length grow by the power of two due to the transformer's self-attention layers. Currently, we only have up to 60 tokens per piece. This is a result of the limited number of bars in our dataset. The effect is not yet relevant at this comparably small context size but could impact the text generation speed for longer pieces.
- The training data only includes notation in landscape format and the dimensions roughly resemble the notation on the whiteboard. Therefore, a zero-shot generalization to a more classical optical music recognition task can not be expected. A less domain-specific dataset would be needed to approach this task.

B. Motion Evaluation

This Section is mainly concerned with the influence of three major parameters in the trajectory generation on the note's loudness. To perform this evaluation, a microphone has been placed next to the $C4$ key. Due to the pitch-dependent perception of loudness by the human ear, only relative measurements from one pitch are compared. First of all a human baseline has been collected by hitting the key as hard as possible. All following measurements are compared against this baseline. It needs to be noted that the robot plays significantly softer than the human player. The

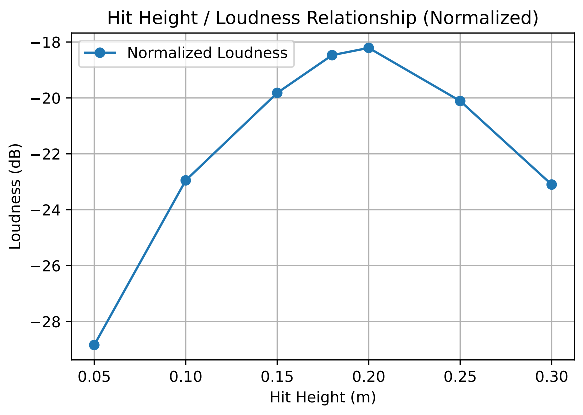


Fig. 8: Effect of the approach point height over the marimba on the loudness relative to a human baseline.

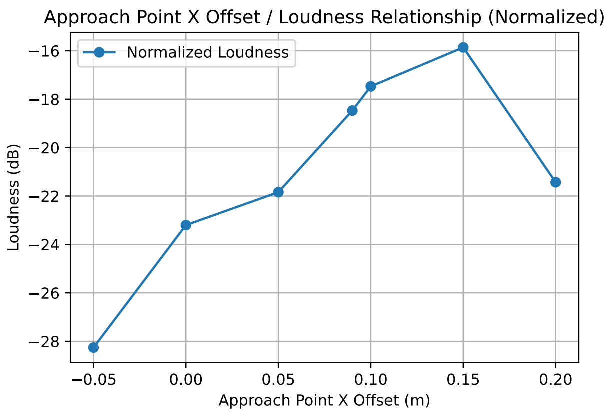


Fig. 9: Influence of the approach points offset in the X direction on the loudness relative to a human baseline.

robot's acceleration and velocity limits have been adjusted to maximize the loudness. While an in-depth ablation of these parameters would be possible, our focus lies on the trajectory generation parameters. Three major ones include:

- Approach point height (Z Offset)
- Approach point X offset
- Hit point Z offset

Figure 8 shows the influence of the height of the approach point on the loudness. Higher approach points allow for a longer acceleration period, but can nevertheless lead to a soft tone because of the dynamics of the compliant mallets during the acceleration and deceleration phases. Too short approaches also result in a very faint note, as the short acceleration phase leads to a low impact velocity of the mallet head. The optimum seems to be an approach from circa $20cm$ above the Marimba. It also needs to be considered, that increasing the stroke distance leads to a slower maximum playing speed with given acceleration and velocity constraints.

Offsetting the approach and retreat point to the front leads to a more circular mallet head trajectory during the approach, which is more closely aligned with both the passive and actively driven joints in the wrist and mallet mount. This

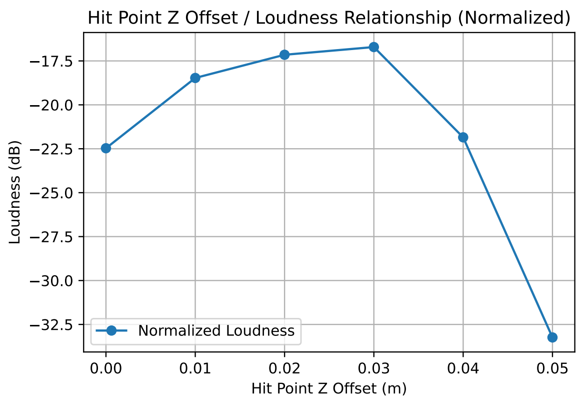


Fig. 10: Relation between the height of the hit point relative to the surface of the key and the loudness relative to a human baseline.

leads to less translational and more rotational movement in the wrist, enabling faster peak speeds. Tuning this offset can significantly influence the loudness as seen in Figure 9. An offset of about $15cm$ seems to be optimal. This could very much be dependent on the approach height, but both variables are analyzed independently for now.

Due to the compliance of the mallet holder assembly, offsetting the hit point might influence the loudness. Breaking slightly above the marimba could still lead to a short impact, but the mallet is not pressed down dampening the note shortly after the initial impact. On the other hand, breaking too early might result in no impact at all. This can be seen in Figure 10. While it suggests that breaking above the marimba is beneficial, one can also observe that for an offset of $5mm$, the mallet fails to substantially impact at all, approaching the experiment's noise floor of $-39.58db$.

In addition to a purely loudness-based evaluation, one might also want to evaluate if a given note was hit at all at the right time during a music piece. While we are able to perform such an evaluation for each played piece at runtime, no specific numbers are included in this report. This is mainly the case because the recall of the onset detection, especially for high notes, is quite low. Also, the collection of a human baseline is much more difficult, as it highly depends on the skill of the subject and we have no professional players on our team.

Some limitations of our current motion pipeline include:

- Non-optimal trajectories due to handcrafted motion heuristics.
- While the mallets are able to hit at the same time, they are not able to hit independently in short succession.
- Only the left mallet is used for single mallet playing. Utilizing the right one might improve the maximum playing speed.
- Compared to a human player, the robot plays comparably soft. This can be observed in e.g. Figure 8.
- Chords including both levels of the marimba might lead to significant and slow arm movement.
- As only one robot arm is utilized, the capabilities of a human with two arms can not be matched.

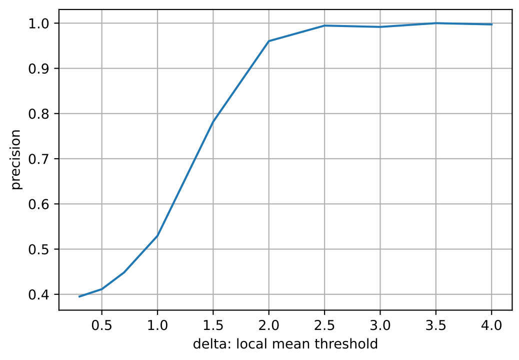


Fig. 11: Local mean threshold of onset detection.

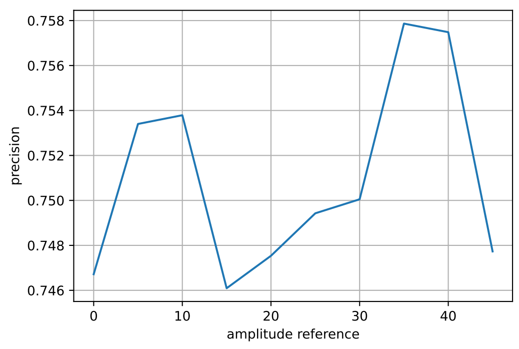


Fig. 12: Amplitude reference for envelope calculating

C. Audio Evaluation

This section will dive into the evaluation of monophonic music note detection performance, with a primary focus on examining the impact of two key factors: The local mean threshold offset represented by the delta (δ), increasing the (δ) will discard more peak with small amplitude. And the amplitude reference expressed in decibels (dB) within the context of envelope generation based on the CQT spectrum, increasing the reference amplitude at the proper level will increase the sensitivity of sensing the music note will higher frequency, even under a nosing background.

To conduct our evaluation, we initially gathered music audio data through manual human performance. In essence, we had individuals play each music note individually, capturing their audio as the ground truth for our music note detection system. In a controlled experiment, we utilized the same mallet that the robot employs, thus replicating the robot's striking effect accurately. However, it's worth noting that human motion offers greater flexibility, resulting in significantly louder sound production compared to the robot. For calculating the precision, we compare the percentage of recognized music notes regarding the ground truth. For the impact of those two factors refer to Figure 11 and Figure 12. As we can see, the most proper value of the amplitude reference is 35 dBs, which quite makes sense, since increasing the reference value, on the one hand, will increase the sensitivity of detecting the music note with high frequency, on the other hand, it will reduce the sensitivity of sensing the music note with lower frequency. Regarding the local mean threshold δ, it seems to be as big as better, a most possible explanation for that is, that increasing this value will discard the false positives of the onset detection.

The limitations of current music note detection of the MarimbaBot are:

- False negative detection due to the background noise and robot motor noise
- False positive detection due to the noise and the sound with the same frequency of music.
- Only capable of detecting the notes from monophonic music
- Since there are multiple vibrating chambers of the marimba, it is hard to attach a contact microphone.
- The music note $C\#7$ can not be detected because of the bug from CREPE.

D. Command Recognition Evaluation

Regarding the command recognition subsystem, we developed a keyword spotting model aimed at triggering the speech recognition system, effectively streamlining the system's operation. To create our training dataset, we performed a manual collection of 226 wake-word and 220 non-wake-word audio samples, splitting them into a 2:8 ratio. Furthermore, we augmented the training dataset with additional background conversation and noise data. The test dataset consists of 28 wake words and 44 not-wake words. The model achieved 98.61% precision at the test dataset, detailed results refer to Table IV.

TABLE IV: The testing result of keyword spotting

	True	**False**
Positive	27	0
Negative	44	0

Limitations of the command recognition:

- Lack of complexity of command input, since the command extractor is built by regular expression.
- False positive keyword spotting due to background talking
- Weird output of speech transcription because faulty of prompt

VI. Conclusion

In conclusion, our multifaceted project has successfully developed a robotic system capable of playing musical notes and chords on a marimba, integrating technologies like computer vision, audio processing, and precise motion control. It demonstrates the fusion of robotics, music, and artificial intelligence, showcasing a robot capable of performing a complex musical instrument with precision. Specifically, it provides unique insights into the intersection of robotics and music. However, the system still has several limitations and areas open for further research. Including a lack of dynamic adjustments, improving audio recognition, and vision detection, i.e. generalization and adaptation to different music notations.

References

[1] Tully Foote. tf: The transform library. In *Technologies for Practical Robot Applications (TePRA), 2013 IEEE International Conference on*, Open-Source Software workshop, pages 1–6, April 2013.

[2] Guy Hoffman and Gil Weinberg. Interactive improvisation with a robotic marimba player. *Autonomous Robots*, 31:133–153, 2011.

[3] Geewook Kim, Teakgyu Hong, Moonbin Yim, JeongYeon Nam, Jinyoung Park, Jinyeong Yim, Wonseok Hwang, Sangdoo Yun, Dongyoon Han, and Seunghyun Park. Ocr-free document understanding transformer. In *European Conference on Computer Vision*, pages 498–517. Springer, 2022.

[4] Jong Wook Kim, Justin Salamon, Peter Li, and Juan Pablo Bello. Crepe: A convolutional representation for pitch estimation. In *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 161–165. IEEE, 2018.

[5] Adam L Kleppe and Olav Egeland. Inverse kinematics for industrial robots using conformal geometric algebra. *Modeling, Identification and Control*, 37, 2016.

[6] N. Koenig and A. Howard. Design and use paradigms for gazebo, an open-source multi-robot simulator. pages 2149 – 2154 vol.3, 04 2004.

[7] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Ves Stoyanov, and Luke Zettlemoyer. Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. *arXiv preprint arXiv:1910.13461*, 2019.

[8] Ze Liu, Yutong Lin, Yue Cao, Han Hu, Yixuan Wei, Zheng Zhang, Stephen Lin, and Baining Guo. Swin transformer: Hierarchical vision transformer using shifted windows. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 10012–10022, 2021.

[9] Han-Wen Nienhuys and Jan Nieuwenhuizen. Lilypond, a system for automated music engraving. In *Proceedings of the XIV Colloquium on Musical Informatics (XIV CIM 2003)*, volume 1, pages 167–171. Citeseer, 2003.

[10] Alec Radford, Jong Wook Kim, Tao Xu, Greg Brockman, Christine McLeavey, and Ilya Sutskever. Robust speech recognition via large-scale weak supervision. In *International Conference on Machine Learning*, pages 28492–28518. PMLR, 2023.

[11] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[12] Bianca Westermann. The biomorphic automata of the 18th century. mechanical artworks as objects of technical fascination and epistemological exhibition. *Figurationen*, 17(2):123–137, 2016.

[13] Yoyo, Christian Liebhardt, and Sayooj Samuel. Breezewhite/oemer: v0.1.7, October 2023.

[14] Kevin Zakka, Philipp Wu, Laura Smith, Nimrod Gileadi, Taylor Howell, Xue Bin Peng, Sumeet Singh, Yuval Tassa, Pete Florence, Andy Zeng, and Pieter Abbeel. Robopianist: Dexterous piano playing with deep reinforcement learning. In *Conference on Robot Learning (CoRL)*, 2023.