Saarland Informatics Campus SIC | HCI Human Computer Interaction Lab Saarland University

EXOKIT

A TOOLKIT FOR RAPID PROTOTYPING OF INTERACTIONS FOR ARM-BASED EXOSKELETONS

USER MANUAL

VERSION 1.0 (NOV 2024)

MARIE MUEHLHAUS, ALEXANDER LIGGESMEYER, JÜRGEN STEIMLE

Project Webpage (including video, link to 3d models, code etc):
https://hci.cs.uni-saarland.de/exokit/

Reference:
If you use this toolkit, please reference the scientific publication as follows:
Marie Muehlhaus, Alexander Liggesmeyer, Jürgen Steimle. ExoKit: A Toolkit for Rapid Prototyping of Interactions for Arm-based Exoskeletons. In Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems (CHI '25). Yokohama, Japan.
DOI: https://doi.org/10.1145/3706598.3713815

1 Table of Contents

1 Introduction

1.1 Overview

Exoskeletons open up exciting possibilities for novel interactions that assist, modify, or constrain physical body movements. These wearable devices can be used in Human-Computer Interaction (HCI) research for applications such as motion capture, delivering haptic feedback in virtual reality (VR), and facilitating skill transfer between individuals. For instance, an exoskeleton might serve as a wearable tutor, teaching novices to perform sports like learning a perfect golf swing, or it might enhance safety in hazardous environments, such as preventing accidents on construction sites.

However, the development of functional human-exoskeleton interactions is complex, requiring expertise in mechanical design, motion control, and interaction design. ExoKit is a do-it-yourself toolkit designed to empower designers, makers, and hobbyists in prototyping low-fidelity, functional arm-based exoskeletons. Targeted at users with basic programming and electronics skills, ExoKit offers:

- **Modular hardware components** for sensing and actuating shoulder and elbow joints. These components are designed to be easy to fabricate, assemble, and reconfigure for customized applications.
- **Programming interfaces** that simplify programming interactive behaviors.

ExoKit specifically supports users in the **early stages of prototyping** to creatively explore diverse ideas. Once a promising interaction concept is identified, users can transition to higher-fidelity hardware and software solutions.

ExoKit is an open-source toolkit for low-fi prototyping. We invite the community to use, modify, improve and extend.

Chapter Overview

Chapter 1 provides an introduction to ExoKit's purpose, some relevant background to human anatomy, application examples to inspire novel users, and recommendations for safe use.

Chapter 2 provides an overview of the required software and hardware, the fabrication and installation steps, and a quick start guide that guides novel users through the basic setup and usage of the hardware in conjunction with the software.

Chapter 3 introduces the modular hardware components, how to combine them, and how to wire the associated sensors and actuators.

Chapter 4 gives an overview of the offered software abstractions and libraries.

Chapter 5-7 explain the usage of the Processing GUI, library, and Arduino API, respectively. It contains guidance on the setup, all function signatures, parameter explanations, sample code and troubleshooting guides. It is recommended to start with the GUI, then move on to the other libraries. It is not necessary to read a chapter in a linear fashion, but to get an overview of the available functions and then use them as needed.

Chapter 8 provides the code of four application examples presented in the published paper.

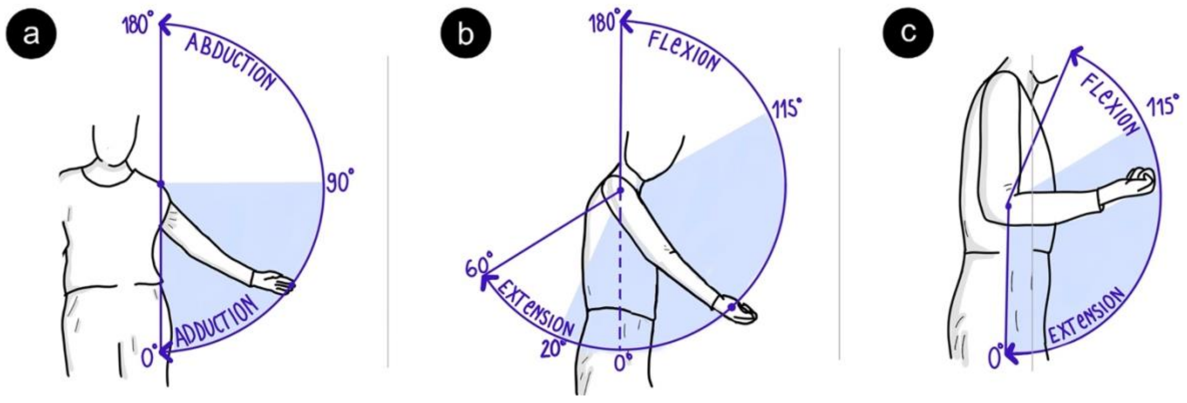
The appendices include a glossary, a debugging script when working with position encoders to ensure they are set up correctly, a sample script for streaming sensor data including load cell data, a cheat sheet of compatible motor properties and the terminal commands for the command line interface.

1.2 Brief Background on Human Anatomy

Understanding the biomechanics of the human upper limbs is crucial for designing effective exoskeletons. Several key properties of the upper limb's anatomy and movement capabilities are particularly relevant in this context. These include the joints' degrees of freedom (DoFs), range of motion (RoM), and torque settings.

The shoulder joint possesses three **DoFs**, enabling the following movements: abduction-adduction, flexion-extension, and medial-lateral rotation. The elbow, on the other hand, has two DoFs, allowing for flexion-extension and rotation of the forearm. ExoKit currently supports flexion-extension movements in both the shoulder and elbow (see below Figure b, c), as well as abduction-adduction in the shoulder (Figure a).

The **RoM** of a joint defines the maximum arc through which the joint can move. These ranges can vary based on factors such as individual anatomical differences and the targeted population. For instance, typical ranges of motion of adults include shoulder abduction and flexion with up to 180 degrees, while shoulder extension reaches up to 60 degrees. Elbow flexion typically spans 150 degrees, with extension returning to a neutral 0-degree position. The RoM supported by ExoKit is visually highlighted in the figure below.



Another critical factor is the **torque** required to assist or replicate human joint movements. As such, different applications require different levels of support that are tailored to the user needs. For example, rehabilitation exoskeletons might adjust the amount of support to the patient's progress and muscle strength whereas industrial applications may demand higher torques to handle heavy loads. Directly affecting the user's body, torques should be carefully selected depending on the intended use case.

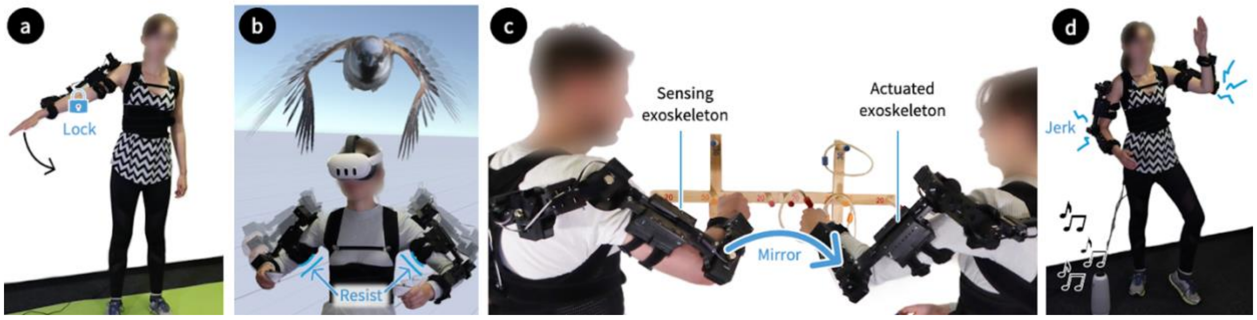
1.3 Application Examples

These are some ideas to spark your creativity of what to do with ExoKit:

- **Improve athletic performance**: Use ExoKit to guide or correct your body movements for tasks like throwing a ball. For example, it could help you adopt the right starting position or restrict joint angles to prevent overextension.
- **Collaborative art**: Work with the exoskeleton to paint on a shared canvas**.**
- **Control a digital game**: Use the exoskeleton as a motion-based controller for interactive games. Explore open-source games, such as those available on OpenProcessing, and adapt them to your exoskeleton setup.
- **Compose music through motion**: Turn your exoskeleton into a wearable musical instrument using the Processing Sound Library. For instance, you could map joint angles to different notes and movement speed to pitch or volume to create dynamic soundscapes.
- **Encourage healthy movement**: Design an application that reminds you to move after long periods of inactivity, promoting better posture or overall physical well-being.
- **Create a two-player game**: For instance, imagine a competitive game where players race to achieve a specific pose. The first to succeed scores a point, while the slower player's exoskeleton locks briefly as a playful penalty.

In **Chapter 8**, we present the code of four application cases implemented with ExoKit:

- Designing **motion guidance strategies** for physical therapy (Figure a)
- Creating **kinesthetic feedback** to enhance immersion in virtual reality (Figure b)
- Developing a **collaborative, body-actuated play experience** (Figure c)
- Augmenting a user's **dance style** with motion modification (Figure d)



1.4 Safety & Best Practices for Safe Use

Safety is a critical consideration when working with a device that can actuate the body. ExoKit includes **three layers of safety**—mechanical, software, and user-centered—specifically designed to support novice developers. These layers aim to minimize risks during prototyping:

Mechanical Safety:

- ExoKit provides mechanical safety features to limit the exoskeleton's range of motion. Designers can attach **3D-printed mechanical restrictions** to the joints using a simple plug-and-play mechanism (see Chapter 3.1.2). These restrictions physically block the motors from exceeding a predefined range of motion and can be added, removed, or exchanged as needed.

Software Safety:

ExoKit incorporates software routines to monitor the exoskeleton's range of motion in real time. If the system detects that the configured limits are exceeded, it will **instantly shut down the system** and terminates the current program.

User-Centered Safety:

To give users control over safety, users can include a **panic button**. When pressed, it **instantly shuts down the system** and terminates the current program. This feature is especially useful during early development or debugging, allowing users to quickly stop the exoskeleton in case of unexpected behavior.

Other Best Practices

In addition to the built-in safety features, we recommend following these guidelines to ensure safe operation:

- **Test interactions off-body first**: Before wearing the exoskeleton, test the implemented behavior in a safe, controlled environment to understand how it operates. For instance, lay the exoskeleton on a table or attach it to a mannequin.
- **Keep the panic button and/or power switch close**: Always keep the panic button close at hand or stay near the power supply, so you or someone nearby can shut off the system quickly if needed.
- **Perform proper calibration**: During setup, ExoKit will guide you through calibration steps. Follow these carefully to ensure the exoskeleton is properly configured.

2 Getting Started

2.1 Hardware Components

- 1 ***Arduino Mega***
- 1 ***Dynamixel Shield*** for Arduino
- 1 ***Sparkfun serial basic breakout*** *(CH340C)*
- 1-6 ***motors of the Dynamixel XM series***
- 1-3 ***12 Volt Dynamixel power supplies*** *(Robotis SMPS 12V 5A);* depends on the number of motors; we used 3 supplies when 6 motors were involved.
- 1-2 ***SMPS2Dynamixel adapter*** (only if several motors are involved)

- **Cables**:
    - USB cable to connect Arduino to PC
    - USB cable to connect serial breakout board to PC
    - Dynamixel cables to control motors (must match the selected motors)
    - Connectors to connect Dynamixel power supply to the Dynamixel shield (e.g., https://www.amazon.de/Connctors-Connector-Security-Surveillance-Transmission/dp/B07QLTWQ66/ref=sr_1_23?__mk_de_DE=ÅMÅŽÕÑ&crid=3CR4GU3YRNQK6&keywords=DC+Power+Connector+5.5x2.5mm&qid=1688727928&sprefix=dc+power+connector+5.5x2.5mm%2Caps%2C145&sr=8-23)
    - If SMPS2Dynamixcel adapter used, additional Dynamixel convertible cables are needed to connect the adapter to the XM motors (X-series-to AX-series adapter cable)
    - Jumper wires

- 1 ***U2D2***. Required for initial setup of the Dynamixel motors in the Dynamixel Wizard. Please refer to the Dynamixel user manuals.

- ***Screws***:
    - *M2x10mm*: 2 per small XM motors (2 each)
    - *M2.5x16mm*: 2 per big XM motor and 2 per passive joint module
    - *M3x60mm*: 4 per shoulder and 1 per passive joint module
    - *M4x40mm*: 1 per shoulder
    - *M4x60mm*: 2 per connected chain segments, arm rail (1-2, depends on setup), 4 per arm cuff
    - *M5X35mm*: 2 per connection between joint module and rigid rod
    - *M4x35-60mm*: 4 per load cell (important: cross-check that they fit with the specific load cell model)

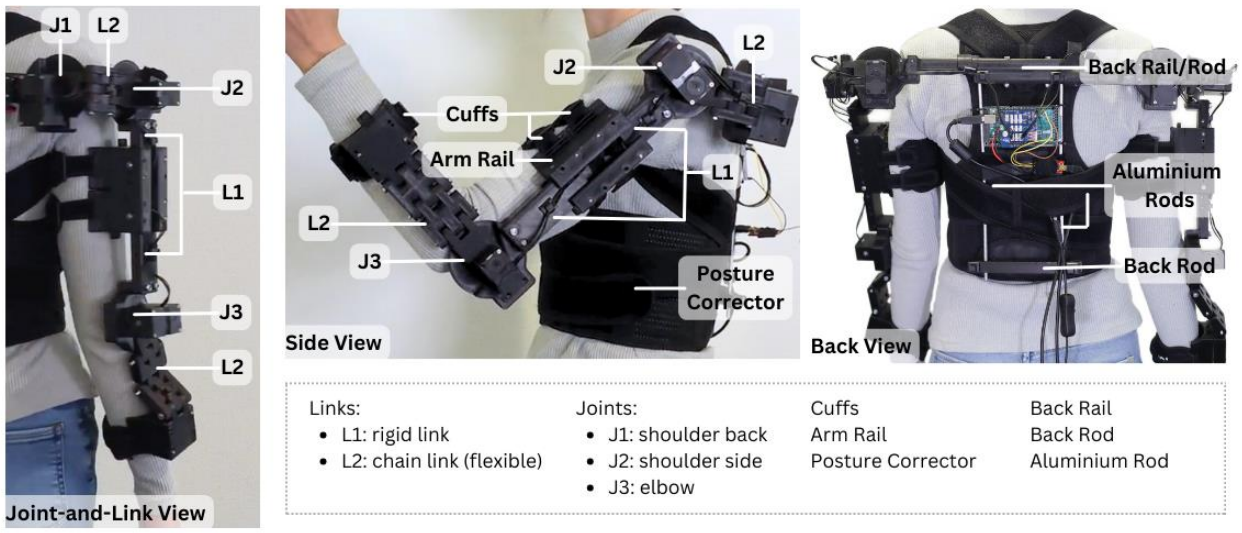
- 2 ***aluminium rods*** (5mm diameter, 30 mm length) for the back construction (see chapter 2)
- ***Posture corrector***. For a stable attachment, the posture corrector should have a stable and big back part (see chapter 2)
- ***Velcro tape*** for attaching the exoskeleton back construction to the posture corrector (see chapter 2)
- Textile ***arm cuffs***. The cuffs must fit through the holes of the printed arm cuffs (max. 3cm width)
- ***Filament***, e.g., ABS

- Optional - *bearings*, e.g.: https://www.kugellager-express.de/miniatur-kugellager-mr-74-2rs-4x7x2-5-mm. They can be inserted in the chain segments for a smoother rotation:
  - 2 bearings per joint rod
  - 2 bearings per chain adapter at the forearm cuff
  - 6 bearings per female chain segment
  - 5 bearings per male chain segment

- Optional - *sensors*:
  - 1 Arduino-compatible *pushbutton* to be used as the emergency button
  - Adafruit-based load cells with HX711 ADC chips. Important: should support at least 80 Hz
  - Adafruit-based panel mount 10K potentiometer for angle tracking

- Optional – *mobile setup*. Motors can be powered with the Dynamixel power supplies or a portable power bank. For the latter, we used:
  - 1 battery (e.g., https://zeeebattery.com/products/zeee-4s-lipo-battery-3300mah-14-8v-50c-xt60?srsltid=AfmBOorQjAq9dsYPO9lusj896r7ObA3ebrmTTRoyuW0HyrSZbV6QOapf)
  - 1-3 voltage converters to power the motors & shield (e.g., https://www.amazon.de/gp/product/B09CYZTM96)
  - 1 voltage converter to power the Arduino (e.g., https://www.amazon.de/gp/product/B09B7XZYJQ?th=1)
  - Lipo voltage tester
  - Optional to cover the voltage converters & co: a plastic box large enough to fit the components

2.2 Hardware: Fabrication



Step 1: 3d printing

The 3d models were designed with Autodesk Fusion 360. We fabricated the 3D-printed parts with an Ultimaker S5 printer, using ABS filament, 100% infill with an octet infill pattern, a normal support structure and 2 mm wall thickness.

To prepare the **back construction**, print:

- `2x back_rail`
- `1x lower_back_rail`

To build a **3 DoF actuated left arm** (right arm analogous), print:

- Joints: `1x joint_elbow_l, 1x joint_shoulder_back_l, 1x joint_shoulder_side_l, 3x rotor`
- Cuffs: `2x arm_cuff_upper_arm, 1x forearm_cuff` (the cuff for the forearm is smaller than the upper arm cuff)
- Links: `1x arm_rail, 2x joint_rod_long, 1x forearm_chain_adapter`, multiple `chain_mf` and `chain_mm` (we recommend at least two chain segments of each type)

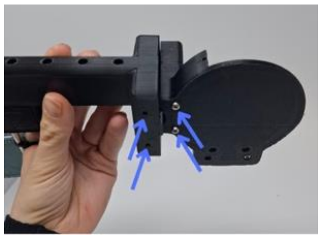
If **passive joint modules** (cf., Chapter 3) are needed, print:

- `1x servo_dummy_outer`
- `1x servo_dummy_rotor`
- `1x rotor`

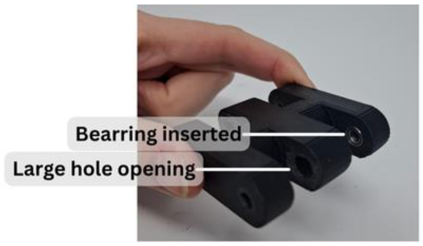
If a **load cell** should be deployed at the upper arm, also print:

- `1x loadcell_adapter_joint`
- `1x loadcell_adapter_rail`

Step 2: Preparing the fabricated parts

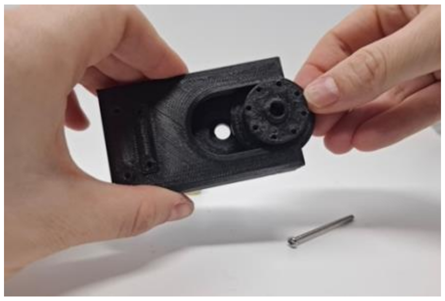
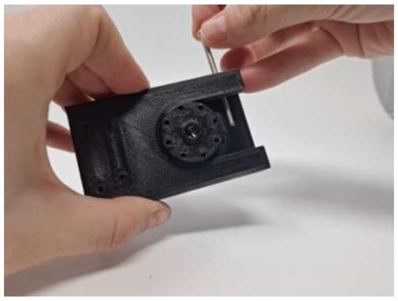


Stabilization of the shoulder: Insert screws at the left and right ends of the back rail (4 *M3x60mm* screws per side). These enhance stability of the back construction.

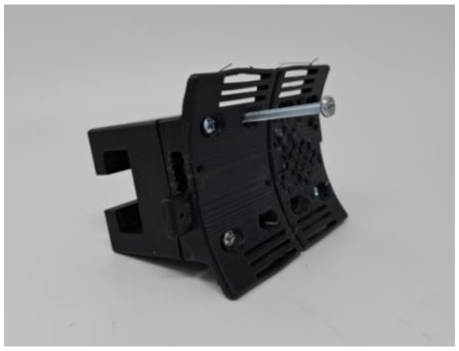
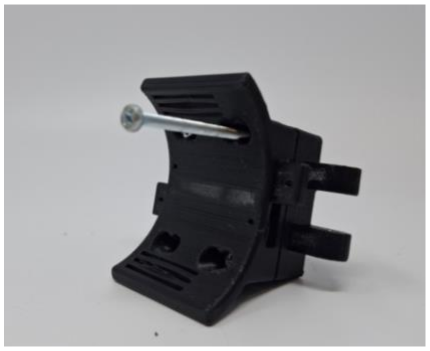


Chain segments: The chain segments and forearm chain adapter are designed to accommodate bearings. The use of bearings helps the chain to move more smoothly. Insert the bearings by pushing them into the holes. Pliers might help to insert the bearings.

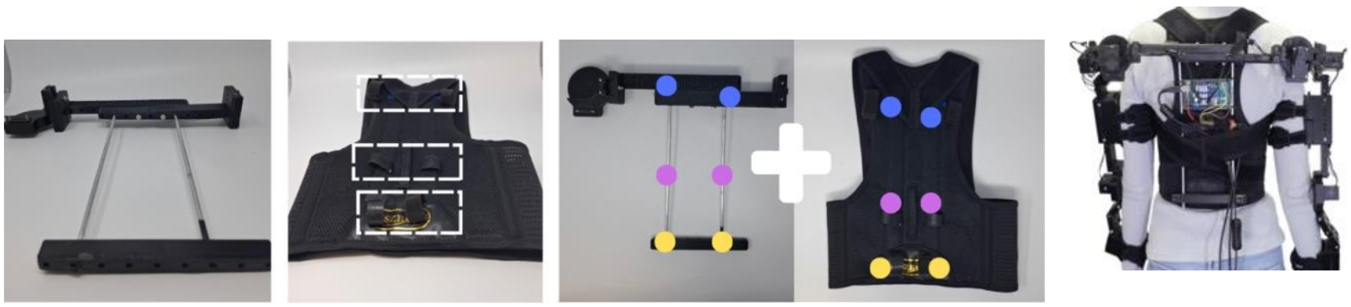
Preparing the passive joint modules: First, insert the `servo_dummy_rotor` into the `servo_dummy_outer`. Next, prevent it from slipping out with the help of a M3x60mm screw.

Arm cuffs: Use *M4x60mm* screws to screw the upper arm cuffs to the arm rail and the forearm cuff to the forearm chain adapter (see photos below). Finally, attach the textile arm cuffs to the 3d printed cuffs.

Assembling the back scaffold: The exoskeleton is held in place through a back construction that is attached to the posture corrector with the help of sewn Velcro tapes. The 3d printed back rails are connected through aluminium rods that were inserted through the holes of the printed rails. Note that the width of the back construction can be adjusted based on which holes the rods are inserted (this affects the alignment of the shoulder joints). The figures below show how the rail and aluminium rods were connected and where the Velcro tapes were sewn to hold the scaffold in place.



2.3 Software: Installation

Processing GUI

1. To set up the Processing GUI, download the GitHub code from the folders “Processing GUI” and “Arduino API”.
2. Download Processing. The authors used version 4.3.
3. Follow the installation guide for the Arduino API.
4. Open processing_gui.pde.
5. Chapter 5 explains the GUI usage.

Processing library

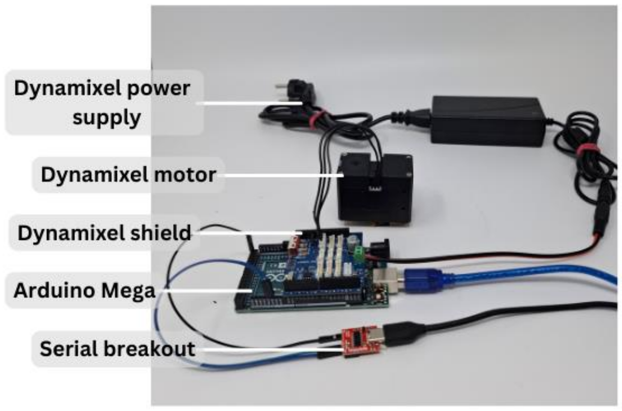
1. To set up the Processing library, download the GitHub code from the folders “Processing library” and “Arduino API”.
2. Download Processing. The authors used version 4.3.
3. Install the library.
4. Follow the installation guide for the Arduino API.
5. You can find an example program in the download folder (processing_sample).
6. Chapter 6 explains the library usage.

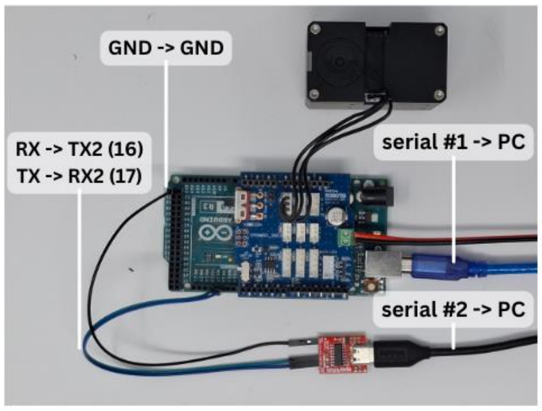
Arduino API

1. To set up the Arduino API, only download the GitHub code from the folder “Arduino API“
2. Follow existing online guides to the platformIO extension for CLION. Test the successful installation with a simple Arduino program (you can find examples online).
3. Install the required plugins in CLION:
   - Serial Port Monitor Plugin: https://plugins.jetbrains.com/plugin/8031-serial-port-monitor
   - PlatformIO Plugin for CLion: https://plugins.jetbrains.com/plugin/13922-platformio-for-clion
4. Open the project as a platformIO project.
   - To work with the Arduino Mega, PlatformIO must be set up for megaatmega2560. We provide the suitable platformio.ini file as part of the downloaded code. The platformio.ini contains further library dependencies, such as Dynamixel2Arduino, DynamixelShield, HX711, LinkedList, TaskManagerIO.
5. Build the project.
   - In case of an error message about undefined references to loop and setup functions, make sure that one (and only one!) of the example programs in the src folder is commented in.
   - In case of error messages about libraries not being available for the system, the library dependencies listed in platform.ini (“lib_deps”) might need to be adjusted to fit your system.
6. You can find a video for setting up the platformIO project (software_installation) in the repo’s video folder (“Software Installation PlatformIO”).
7. Details about the API can be found in chapter 7.

2.4 Quick Start Guide: Basic Setup

This section guides you through setting up the Arduino Mega, Dynamixel shield, and serial communication. The basic setup is shown below:





Components

- **Arduino Mega**: The main microcontroller for controlling the system.
- **Dynamixel Shield**: Manages communication with Dynamixel motors.
- **Serial Communication:**
  - **Serial #1**: USB connection to the Arduino for uploading code. ⚠ Note: This port is blocked during motor operation as it is used for motor communication.
  - **Serial #2**: External serial connection through the serial breakout board from the Arduino to a terminal, allowing commands and data exchange while the Arduino communicates with the shield.

Wiring

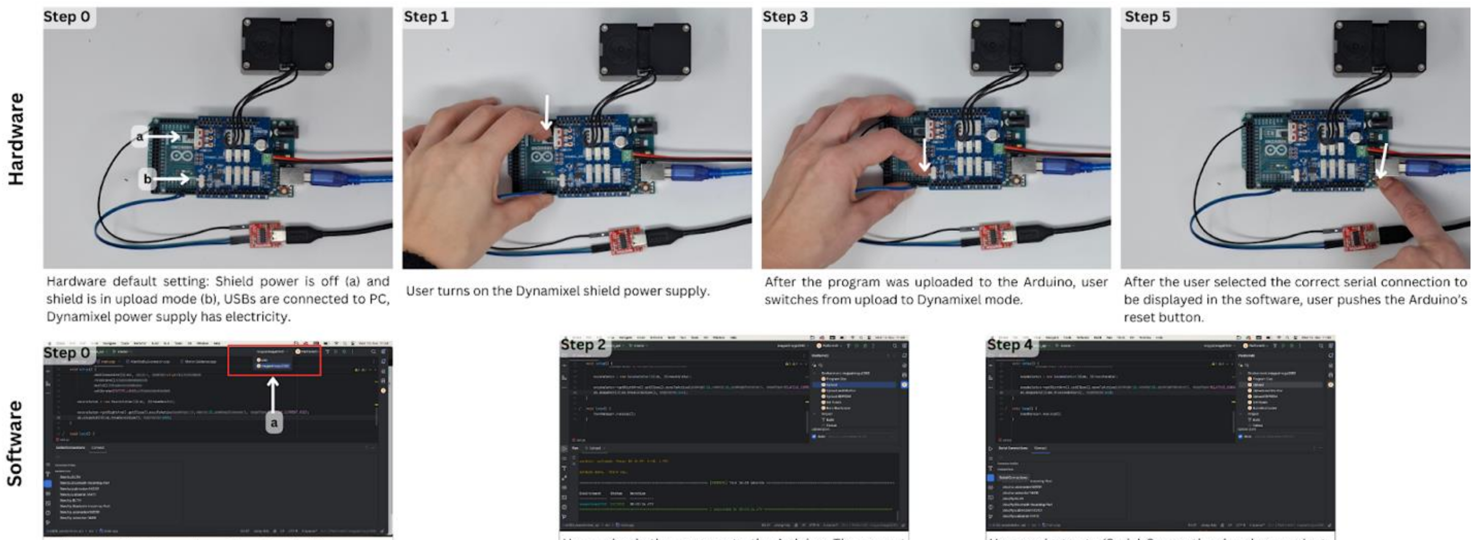
- Connect **RX** of the breakout board to pin **16 (TX2)** on the Arduino Mega.
- Connect **TX** of the breakout board to pin **17 (RX2)** on the Arduino Mega.
- Connect **GND** of the breakout board to any free **GND** pin on the Arduino Mega.
- Ensure both USB connections (Serial #1 and Serial #2) are connected to the computer.
- Connect the motor directly to the shield.
- Connect an **additional power supply** to the shield to power the motors later. ⚠ Be cautious when handling the power supply to avoid damaging the shield or motors.
- While not shown in the photos, you can add an additional Adafruit button, which will serve as an emergency switch. To wire the button, connect one button pin to a digital pin on the Arduino Mega (e.g, **pin 50**), and the other button pin to any free **GND** pin on the Arduino Mega.

Turning On and Connecting the Hardware

Before starting, check the following:

- **Hardware:** Ensure the Dynamixel shield power is off, the Dynamixel shield is in upload mode and the Dynamixel power supply is plugged in. Noth USB cables (Serial #1 and Serial #2) are connected to the computer.
- **Software**: The software is set up correctly with PlatformIO installed, the ExoKit library imported, and the correct board (Arduino Mega) selected.

1. Turn on the **Dynamixel shield power**. ▨ A red LED on the shield should light up (if not already on from a previous session).
2. In the software (e.g., CLion or Arduino IDE), **upload the program** (for testing the basic setup, see sample code below). Ensure the **correct port for uploading code to Arduino is indicated** (serial #1, not the one from the breakout board). Wait until the upload succeeded.
3. Once the upload is confirmed, switch the Dynamixel shield to **Dynamixel mode**.
4. In the software, now select the other serial connection (serial #2) to read the Arduino output and communicate with the board.
5. Press the **reset button** on the Arduino.
6. A calibration dialog should appear in the terminal. When testing the basic setup, simply confirm with **y**.



Hardware default setting: Shield power is off (a) and shield is in upload mode (b), USBs are connected to PC, Dynamixel power supply has electricity.

User turns on the Dynamixel shield power supply.

After the program was uploaded to the Arduino, user switches from upload to Dynamixel mode.

After the user selected the correct serial connection to be displayed in the software, user pushes the Arduino's reset button.

Software default setting: The ExoKit library was downloaded and imported into CLION, PlatformIO works, the correct board is selected (a), and the correct program for execution is open.

User uploads the program to the Arduino. The correct upload port (serial #1) must be selected.

User navigates to 'Serial Connections' and now selects the correct port (serial #2) to read the exo's ouput while the exo is in the Dynamixel mode.

Testing the Basic Setup

To verify the setup, run the following script in PlatformIO. This script moves the motor by 10 degrees.

⚠ **Note:** Before running the script, ensure the **Dynamixel ID** matches the motor. If needed, refer to the **Dynamixel Wizard** to configure unique IDs and the correct baud rate for your motors (recommended: **1,000,000**).

```
#include "event/simpleactionbuilder/ActionBuilder.h"
#include "exo/ConfiguredEncoder.h"
#include "DynamixelShield.h"
#include "exo/builder/ExoskeletonBuilder.h"
#include "event/simpleactionbuilder/Exoskeleton.h"

DynamixelShield dxl;
ExoskeletonHandle* exoHandle;
Exoskeleton* exoskeleton;
ActionBuilder ab;
const float DXL_PROTOCOL_VERSION = 2.0;

void setup() {
   Serial.begin(1000000);
   dxl.begin(10000001);
   dxl.setPortProtocolVersion(DXL_PROTOCOL_VERSION);
   dxl.ping(1);
   DEBUG_SERIAL.begin(115200);

   // TODO change the indicated Dynamixel ID (currently 4) to your
      //Dynamixel's ID
   exoHandle = &ExoskeletonBuilder()
           .addRightArm()
           .addElbowJoint(dxl, 3, nullptr)
           .finishArm()
           .addEmergencyShutdownButton(50,true)
           .build()
           .calibrate(CREATE);

   exoskeleton = new Exoskeleton{ab, *exoHandle};

   exoskeleton->getRightArm().getElbow().moveToActive(10,20,5,
RELATIVE_CURRENT_POS);
   ab.dispatch(ab.trueCondition(), 100);
}

void loop() {
   taskManager.runLoop();
}
```

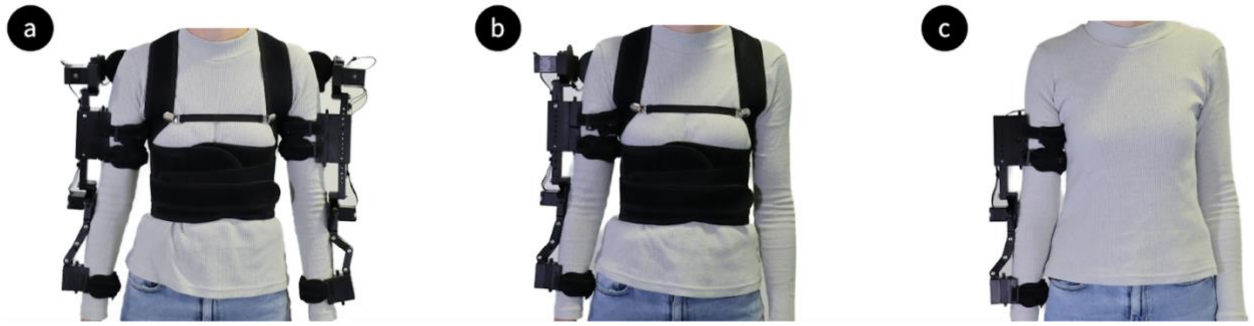
Troubleshooting

If the setup does not work, check the following:

- Is the power turned on?
- Is the Dynamixel shield in **upload mode** (for uploading code) or **Dynamixel mode** (for motor operation)?
- Do the Dynamixel IDs match the IDs indicated in the program? Is the baud rate set to **1,000,000**? Use the Dynamixel Wizard and a U2D2 board for reconfiguration if needed. Refer to the Dynamixel's documentation for more details.
- Is the correct USB serial port selected? If the terminal shows cryptic signs, you may be connected to the wrong port.

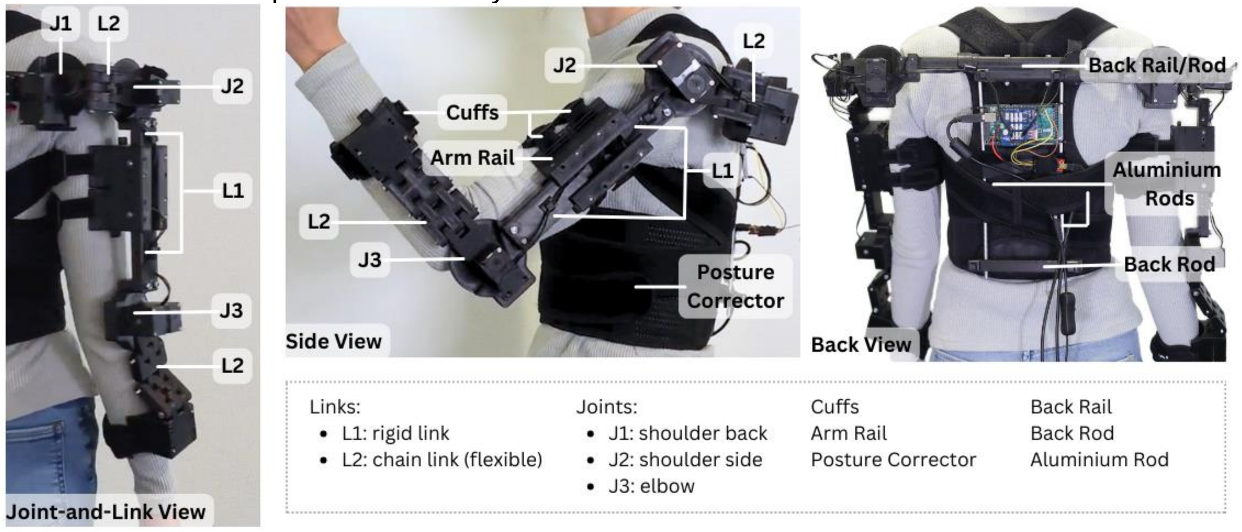
3 Using the Toolkit: Modular Hardware Components

ExoKit allows the designer to easily adjust the hardware prototype by detaching joints to combine and exchange components, adjust the side and remove or add DoFs to fit designer's specific project requirements. For instance, designers can construct exoskeletons that support both arms (a), only one arm (b) or the elbow (c). In the following, we detail on exoskeletons key components (link and joint types), and illustrate the assembly of the modular components.



3.1 Components Overview

The exoskeleton comprises several key elements:

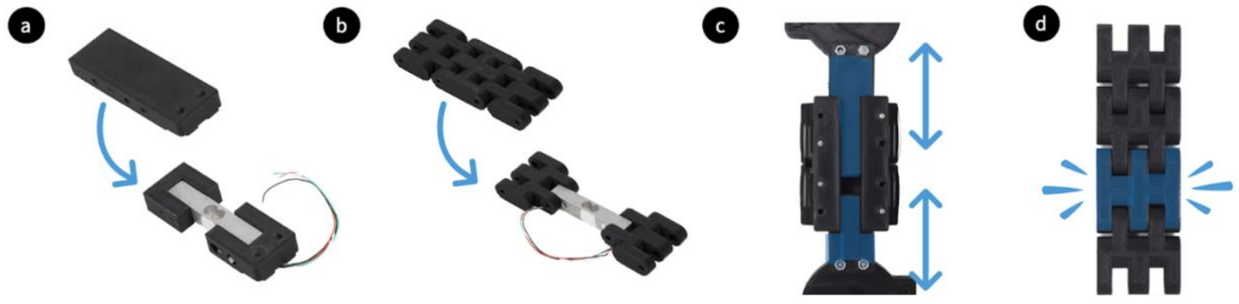


Link types

Exoskeleton links are structural components that connect and transmit forces between joints. In ExoKit, we provide two types of 3D-printable passive links. The first is a simple rigid rod, used to stably connect the shoulder and elbow joints (`joint_rod_long`). The second is a flexible link made of 3d printed chain segments (`chain_mm, chain_mf`).

Integration of additional sensors

If desired, both the rigid and chain links can be equipped with an additional load cell that captures the torques applied to the joints as they are common sensors deployed in exoskeletons (a, b).



Size-adjustability

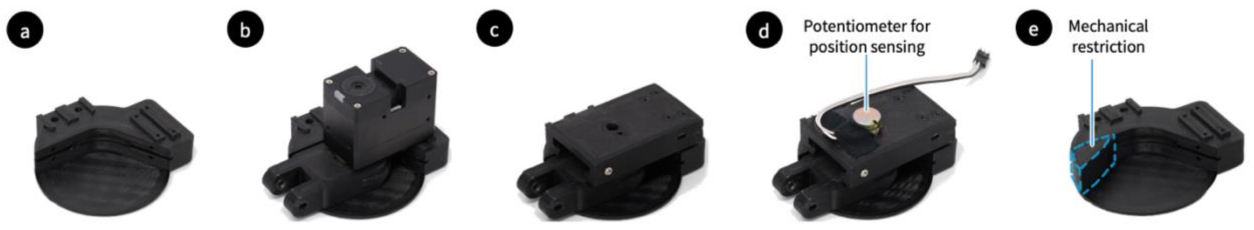
The rigid links on the upper arm and back use adjustable mechanisms to accommodate a range of body sizes (c). The chain link can be adjusted to different shoulder circumferences and lower arm lengths by simply attaching or detaching chain segments (d). Finally, the arm cuffs which connect the links to the human arm come in two different sizes, and can be stacked together to provide a larger area that holds the exoskeleton in place.

Joint types

As exoskeletons can be either actuated, act as a sensing device, or be passive, it is crucial for designers to have the flexibility to choose the functionality for their application themselves. To enable hands-on reconfigurability, ExoKit offers three types of interchangeable joint modules:

- **Actuated joints** (b) comprise an actuator. A joint base (a) serves for mounting a motor. The base is designed to be compatible with all motors of the Dynamixel XM series. These motors inherently offer sensing capabilities (position, velocity). If desired, developers can modify the design of the joint base to include other motors with different dimensions.
- **Passive joints** (c) simply follow the user's motions, realizing a passive DoF.
- **Sensing-only joints** (d) measure the exoskeleton angle configuration, through which further motion properties, including velocity and acceleration, can be inferred.

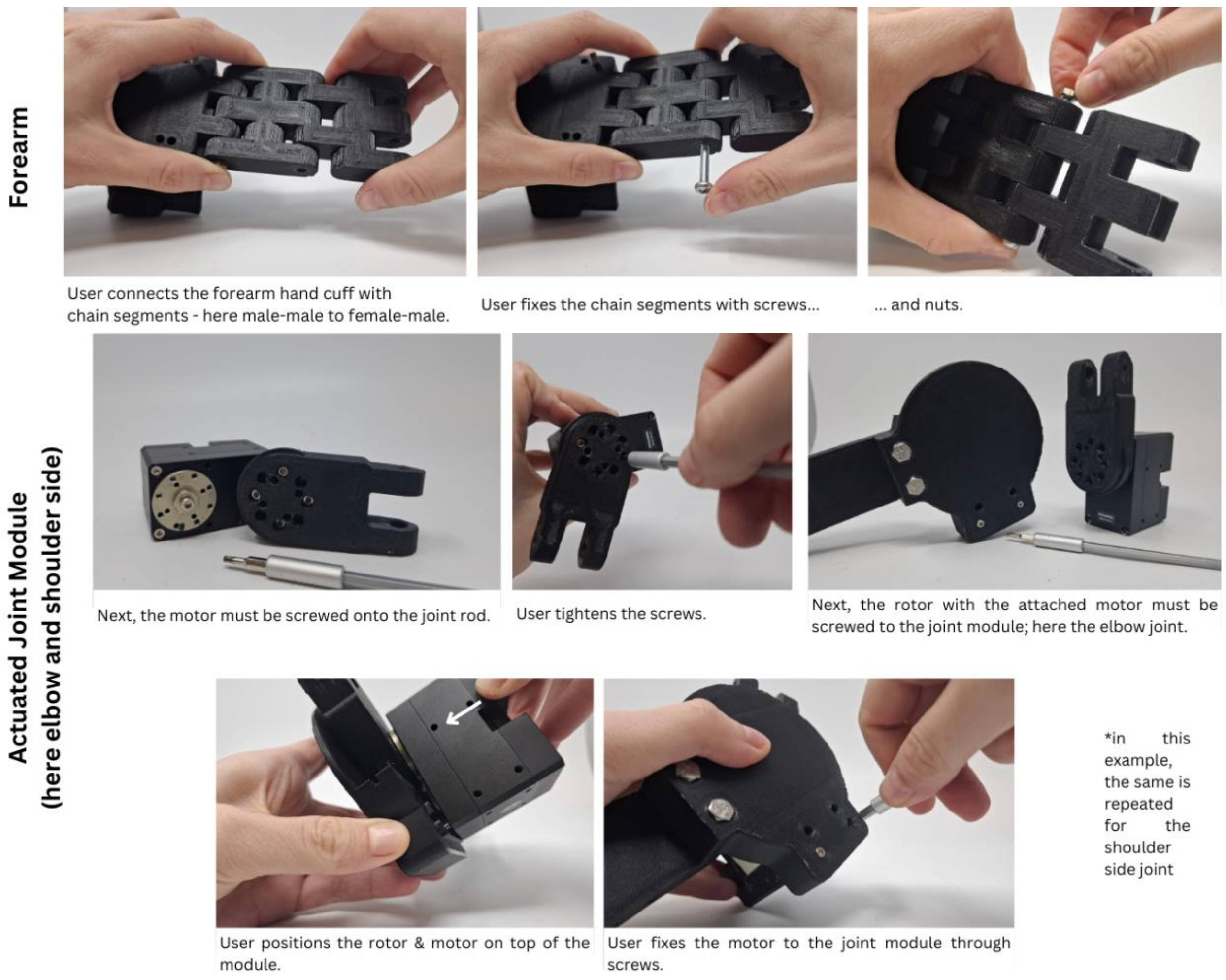
To realize sensing-only and passive joints, ExoKit provides a module that can be mounted on top of the joint base instead of a motor (c,d). This module offers a space for an optional position encoder, hence can be repurposed as a sensing-only joint when the position encoder is attached.



Mechanical restrictions for adjustability

Designers can add 3d printed mechanical restrictions to the base joints (e, `restriction_20/30/45`). Through a simple plug-and-play mechanism, these mechanical restrictions physically prevent the motor from exceeding the defined range of motion. They are offered in 3 different sizes (restricting the motion by 15, 30 or 45° on both sides) and can be added, removed, or exchanged after fabrication as needed.

3.2 Putting the Exoskeleton Together - Assembly steps



User connects the forearm hand cuff with chain segments - here male-male to female-male.

User fixes the chain segments with screws...

... and nuts.

Next, the motor must be screwed onto the joint rod.

User tightens the screws.

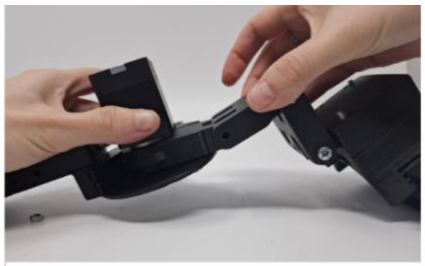
Next, the rotor with the attached motor must be screwed to the joint module; here the elbow joint.

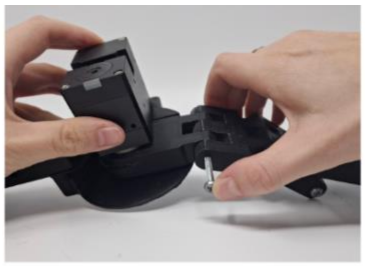
User positions the rotor & motor on top of the module.

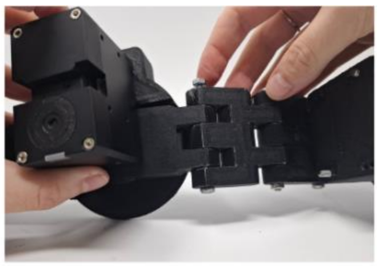
User fixes the motor to the joint module through screws.

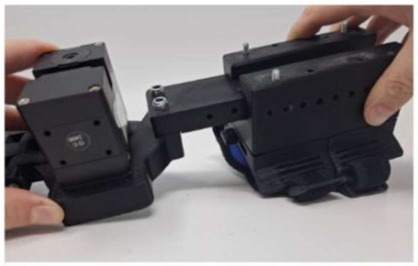
*in this example, the same is repeated for the shoulder side joint

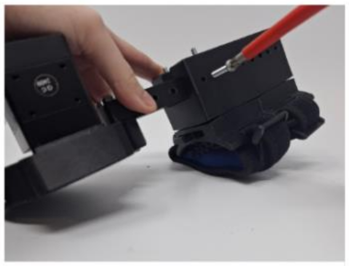
Upper Arm

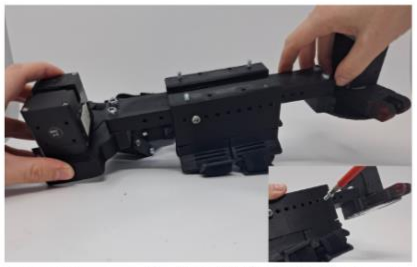

The rotor is now connected to the chains & forearm cuff.


User fixes the chain segments with screws...

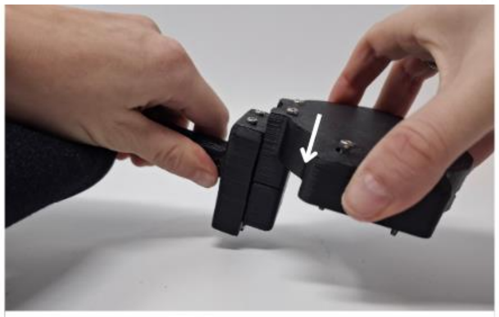

... and nuts.

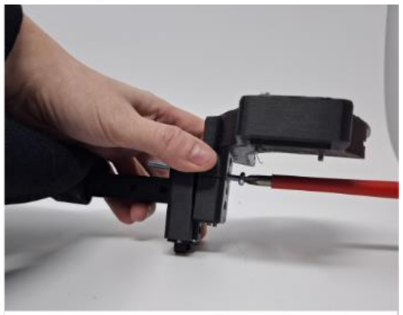

Insert the rigid link attached to the other side of the elbow joint into the arm rail.


User fixes the link at a suitable height with a screw.

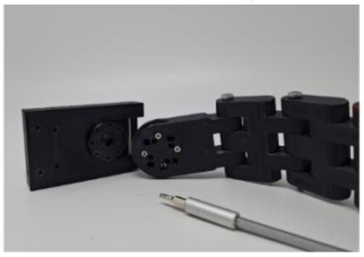

Repeat the same for the rigid link attached to the shoulder side joint module.

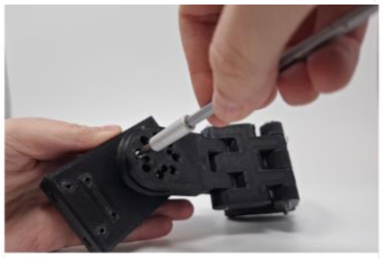
Shoulder

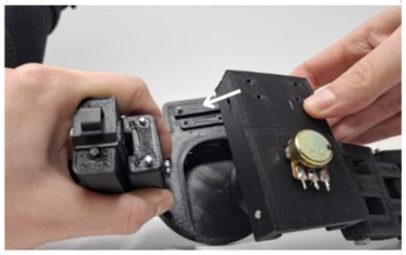

Connect the shoulder back module to the back rail.

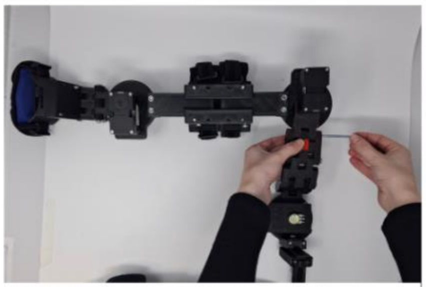

Fix it with a screw and nut.

Passive/Sensing Joint Module (here shoulder back joint)


To connect the passive/sensing joint module, it must be screwed onto the joint rod.

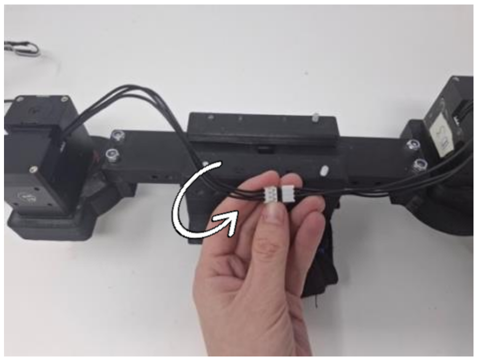

User tightens the screws.


User positions the rotor with the passive/sensing joint module on top of the module.

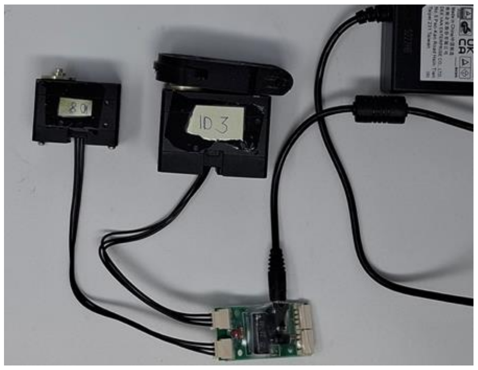

Finally, the user connects the shoulder side and shoulder back joints through chains.

3.3 Wiring Motors and Sensors

Connecting multiple motors



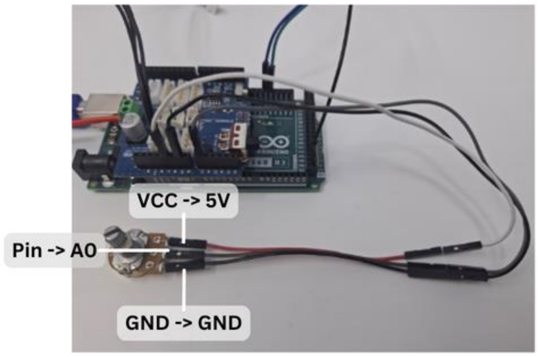
Connect all motors that are part of the same exoskeleton arm (up to 3 motors) to one long daisy-chain. If the cable between the two motors is too short, you can connect two shorter Dynamixel cables with each other. However, one of the two cables must be rotated by 180° before connecting them.



When connecting at least 3 motors through a daisy-chain, it is recommended to add additional power inputs. You can add a SMPS2Dynamixel adapter in between two motors and supply the adapter with extra power from the Dynamixel power supply.

Some more helpful material: https://www.youtube.com/watch?v=LN2XjlSr1kM

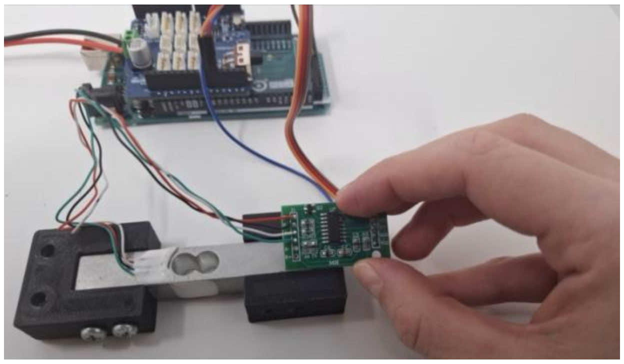
Connecting position encoders

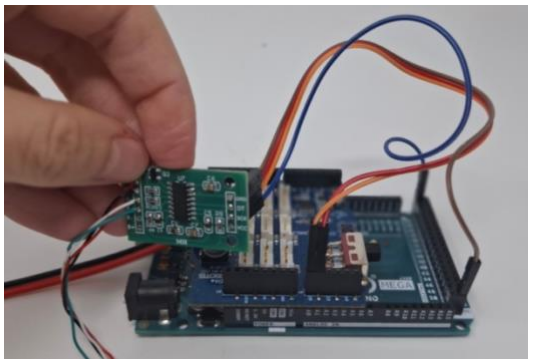


- Connect the **VCC pin** of the potentiometer to any free **5V pin** on the Arduino Mega.
- Connect the **analog pin** of the potentiometer to any free analog pin on the Arduino Mega (e.g., A0).
- Connect the **GND** of the potentiometer to any free **GND** pin on the Arduino Mega.

Connecting the load cell

- Connect the **DT and SCK pins** of the HX711 module to any free **analog pins** on the Arduino Mega (e.g., A1 and A0)
- Connect the **GND pin** of the HX711 module to any free **GND** pin on Arduino
- Connect the **VCC pin** of the HX711 module to any free **5V pin** on the Arduino Mega.





4 Using the Toolkit: Software Components

4.1 Overview of ExoKit's Augmentation Strategies

ExoKit is designed to facilitate the creation of interactions for arm-based exoskeletons by providing functional abstractions that can be combined into meaningful motion interactions. These abstractions allow designers to implement both basic operations and advanced motion augmentation strategies.

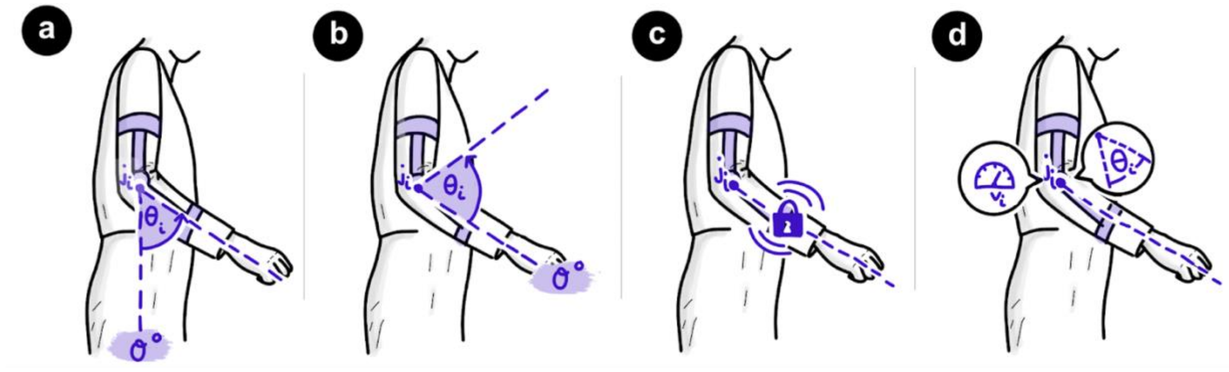
Exoskeletons built with ExoKit are composed of either one or two arms. Each arm consists of up to three joints ji, $i \in \{1, 2, 3\}$, which each can be either passive, sensing, or actuated. As a prerequisite for the following actuation functionalities, the designer must register the exoskeleton's configuration and calibrate the absolute zero-degree position in software for each actuated or sensing joint ji once at system startup.

For ease of programming, all functions are provided for controlling an individual joint only, a set of joints, or the entire arm at once.

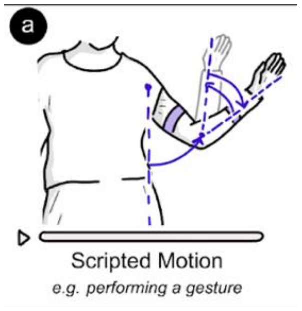
Basic Functions

ExoKit offers various basic funtions: **Moving a joint ji** to an angle θi expressed as (a) **absolute** w.r.t. ji's calibrated zero-degree angle, or (b) **relative** to the joint's current angle; (c) **locking** joint ji in place, or (d) **collecting real-time sensor data** of selected joints. ExoKit provides real-time sensor data about the exoskeleton's physical angle configuration, motion velocity, acceleration, and applied torques. ExoKit can continuously stream this sensor data from selected or all joints through a serial port connection, making the data easily accessible to external applications.
Functions:

- `move To`
- `lock`
- `sensing`



Scripted Motions

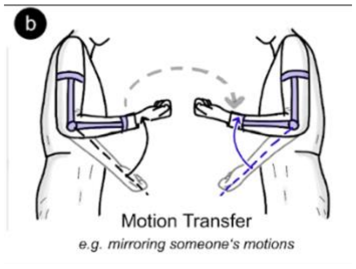


Designers can create and playback scripted sequences of motion, which are defined as a time series of basic move commands. Scripted body motions can be helpful to realize a specific repetitive motion with fixed parameters, or to play back motion sequences that serve as gestures (e.g., waving). For illustrative purposes, we provide functions for executing a **waving gesture** with the arm and for generating a **vibrating** sensation through rapid back-and-forth movements of adjustable amplitude and frequency. The latter serves as a haptic notification for the user.

Functions:

- `gesture`
- `vibration`

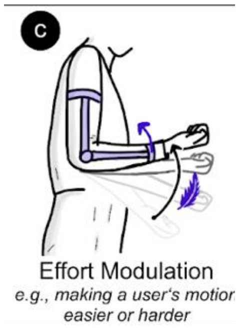
Motion Transfer



The concept of motion transfer involves transferring human motion in real-time from one joint to another joint, from one arm to another arm, or even from one user to another person. For interaction designers, transferring motions offers flexible means for applications. These involve guiding another user's motion, such as for motion teaching and skill transfer but also for creating shared kinesthetic experiences between users. ExoKit offers functions that facilitate motion transfer between two exoskeleton instances exo1 and exo2, where in every time interval, the state of the sensing joints is read and the mapped actuated joints are moved accordingly. Designers can **mirror** the input signal from a sensing exoskeleton arm exo1 onto an actuated counterpart exo2 and optionally **scale the motion up or down** by a user-defined factor f, while the system internally ensures that the scaled motion does not exceed the user's calibrated limits of their range of motion.
Functions:

- `mirror`

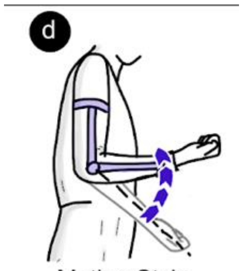
Modulating the Motion Effort



Modulating the motion effort comprises interaction strategies in which the exoskeleton applies torques τ in or opposite to a user's inherent motion, while preserving the user's ability to freely navigate in space. The resulting feeling of physical assistance or resistance can be leveraged by designers to modulate how users perceive their own motion, with stronger torques τ resulting in a stronger effect. ExoKit provides two pre-implemented functions which designers can use to increase or decrease the motion effort, respectively, selectively for individual joints or applying to the entire exoskeleton. These functions include parameters that allow designers to adjust the **assistive** or **resistive torque** τ and to specify whether the strategy should be **continuously active or only triggered when the user moves in a certain direction**.
Functions:

- `amplify`
- `resist`

Modulating the Motion Style



Motion Style
e.g., making a user's motion smoother

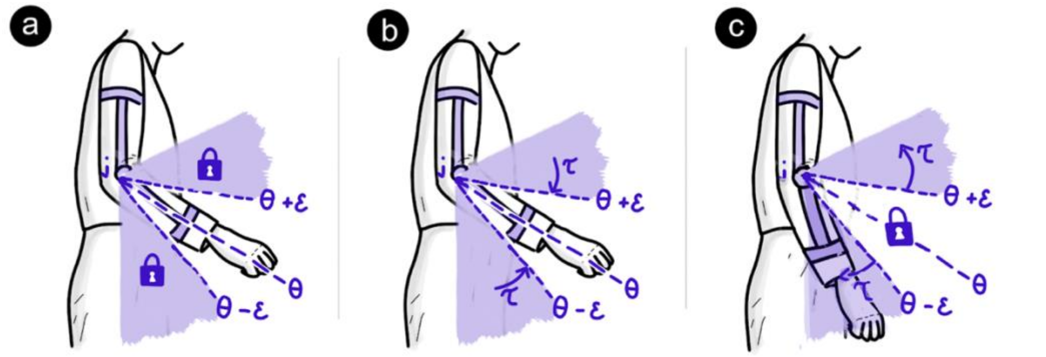
Modulating the motion style refers to interaction strategies that modify the observable characteristics of a user's movement, such as speed or path, while keeping the user in control of the overall motion. These style modifications can range from subtle adjustments to more pronounced changes. ExoKit provides two pre-implemented functions to facilitate exploring modulations of motion style: (1) The exoskeleton tries to **keep a user's motion speed within a pre-defined range**, by applying assistive torques τ if they are too slow or resisting ones if they are moving too fast. Designers can fine-tune these effects by setting the velocity range and the forces applied to maintain it. (2) As an example of an artistic style modulation, the exoskeleton can **introduce jerks to augment a user's motion path** for a more technical or robot-like motion style. A jerk consists of a short back-and-forth movement that creates a brief distortion in the user's movement.
Functions:

- `filter speed`
- `add jerks`

Motion Guidance

ExoKit provides three complementary functions: (a) The exoskeleton can **constrain the range of motion** of a joint ji to an area around an absolute angle θi with range ϵ. As soon as the user attempts to move beyond this range, the exoskeleton tries to move the user back to the boundary with maximum torque, thereby keeping the joint within the specified limits. (b) The exoskeleton can **guide the user towards an area** centered around θi with range ϵ. Here, the exoskeleton applies assistive torques τ to ji when moving towards the desired area and resists otherwise (c) Lastly, the exoskeleton can **keep a joint ji away from the area** around θi, applying resistance with torque τ if the user is approaching the area and assistive torques otherwise. In addition to defining angle θi and range ϵ, designers can also specify the magnitude of the amplifying and resisting forces.
Functions:

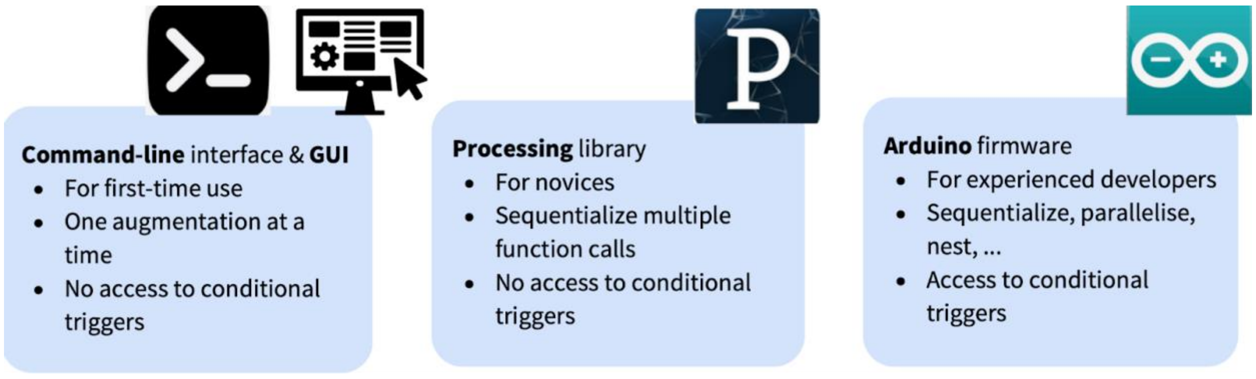
- `constrain to`
- `guide towards`
- `guide away`



constraining user to a range   guiding user towards a range   guiding user away from a range

4.2 Overview of ExoKit’s Programming Interfaces

ExoKit provides a variety of programming interfaces designed to cater to users with different levels of expertise. By offering options for novices and advanced users, ExoKit follows a recommended practice in end-user robot programming and makes interaction programming more accessible and flexible. Below, we outline the key interfaces available.


Command-line interface & **GUI**
For first-time use
One augmentation at a time
No access to conditional triggers
Processing library
For novices
Sequentialize multiple function calls
No access to conditional triggers
Arduino firmware
For experienced developers
Sequentialize, parallelise, nest, ...
Access to conditional triggers

Command-Line Interface (CLI) & Graphical User Interface (GUI) for Novices

For users in the early stages of exploration, ExoKit offers a simple command-line interface (CLI). This interface allows the execution of basic functions and higher-level augmentation strategies while abstracting away complex programming concepts such as loops or conditional statements.

As an alternative to the terminal, novices can use a graphical user interface (GUI) programmed with Processing. The GUI enables easy execution of basic functions at the touch of a button. Sliders with labeled minimum and maximum values help guide users in selecting suitable parameters for various functions. The intuitive design of the GUI lowers the barrier for those with minimal programming experience.

Processing Library for Intermediate Users

For users looking to define sequences of motions, ExoKit provides a Processing library. This supports the creation of moderately advanced behaviors without requiring in-depth knowledge of firmware programming. It further allows to easily couple motion augmentations to graphical Processing applications.

Arduino Firmware for Advanced Users

Experienced users can directly program ExoKit's Arduino firmware to implement custom and complex behaviors. The firmware uses a trigger-action programming model built on an object-oriented, event-driven architecture. This design allows users to combine basic functions and augmentation strategies, define sequences, nested functions, and parallelized actions, and use sensor data as conditional triggers for more responsive interactions. Furthermore, users can define their own manual triggers, such that they can trigger a certain augmentation strategy when sending the self-configured command through the terminal. This allows to control the hardware also with external programs, such as with Python scripts or Processing applications.

4.3 Available Functions

These are the functions implemented in each of the interfaces:

	Functionality	**Processing GUI**	**Processing Library**	**Arduino Library**
Basic Functions	`move To`	✓	✓	✓
	`lock`	✓	✓	✓
	`sensing`	✗	✗	✓
Scripted Motions	`gesture (wave)`	✓	✓	✓
	`vibration`	✓	✓	✓
Motion Transfer	`mirror`	✓	✓	✓
Modulating the Motion Effort	`amplify`	✓	✓	✓
	`resist`	✓	✓	✓
Modulating the Motion Style	`filter speed`	✓	✓	✓
	`add Jerks`	✓	✓	✓
Motion Guidance	`constrain To`	✓	✓	✓
	`guide towards`	✓	✓	✓
	`guide away`	✓	✓	✓

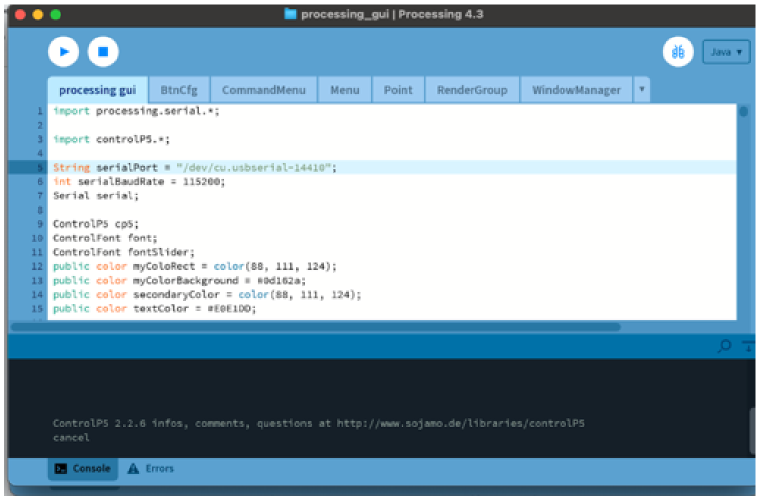
5 First Steps: Using the Processing GUI

The Processing Graphical User Interface (Processing GUI) is the recommended starting point for first-time users of ExoKit. The GUI provides a streamlined interface to access and control ExoKit's core functionalities in a structured manner.

5.1 Setup

To set up ExoKit's Processing GUI, follow these steps:

1. **Prepare the Arduino Library:**
   - Copy the provided script (see next page) into PlatformIO.
   - Replace the necessary motor Ids and pin for the emergency shutdown button in the script based on your exoskeleton configuration.
     - In the provided sample script, we calibrated a left-arm exoskeleton with an active elbow joint (motor ID 3). An emergency shutdown button connected to pin 50 on the Arduino Mega. The assigned exoskeleton ID is 1.
     - If you use a different exoskeleton configuration, update the code accordingly. For more details, see Chapter 7.
   - Build and upload the script to the Arduino (cf., Chapter 2.4 Quick Start Guide).
   - Connect to serial#2 and follow the instructions for calibration provided in the Arduino programming interface terminal.
   - After calibration, disconnect the serial port in the Arduino terminal to avoid conflicts in later steps.
2. **Configure the Processing GUI:**
   - Open the Processing GUI file.
   - Identify the correct serial port (serial #2) for communication. For example:
     - On **Windows**, the port might be COM11.
     - On **Linux/macOS**, you can find a list of your serial devices using the command "ls /dev/tty.*".
   - Provide the correct serial port in the code:



```
import processing.serial.*;

import controlP5.*;

String serialPort = "/dev/cu.usbserial-14410";
int serialBaudRate = 115200;
Serial serial;

ControlP5 cp5;
ControlFont font;
ControlFont fontSlider;
public color myColoRect = color(88, 111, 124);
public color myColorBackground = #0d162a;
public color secondaryColor = color(88, 111, 124);
public color textColor = #E0E1DD;
```

```
ControlP5 2.2.6 infos, comments, questions at http://www.sojamo.de/libraries/controlP5
cancel
```

This is the code that must be uploaded to the Arduino programming interface:

```
#include <com/SerialCmdReader.h>
#include "event/simpleactionbuilder/ActionBuilder.h"
#include "exo/ConfiguredEncoder.h"
#include "DynamixelShield.h"
#include "exo/builder/ExoskeletonBuilder.h"
#include "event/simpleactionbuilder/Exoskeleton.h"

DynamixelShield dxl;
ExoskeletonHandle* exoHandle;
SerialCmdReader terminalInteraction(DEBUG_SERIAL);
const float DXL_PROTOCOL_VERSION = 2.0;

void setup() {
   Serial.begin(1000000);
   dxl.begin(1000000l);
   dxl.setPortProtocolVersion(DXL_PROTOCOL_VERSION);
   dxl.ping(1);
   DEBUG_SERIAL.begin(115200);

   // TODO change the indicated Dynamixel ID (currently 3) to your Dynamixel's
//ID
   // TODO change the emergency button pin number or remove it if not desired
   exoHandle = &ExoskeletonBuilder()
           .addLeftArm()
           .addElbowJoint(dxl, 3, nullptr)
           .finishArm()
           .addEmergencyShutdownButton(50,true)
           .build()
           .calibrate(PREFER_LOAD);
   exoHandle->setId(1);

   terminalInteraction.enableSerialControlCommands();
}

void loop() {
   taskManager.runLoop();
}
```

5.2 Usage

After launching the Processing GUI, the main interface appears (see Figure below, left). It consists of two clusters:

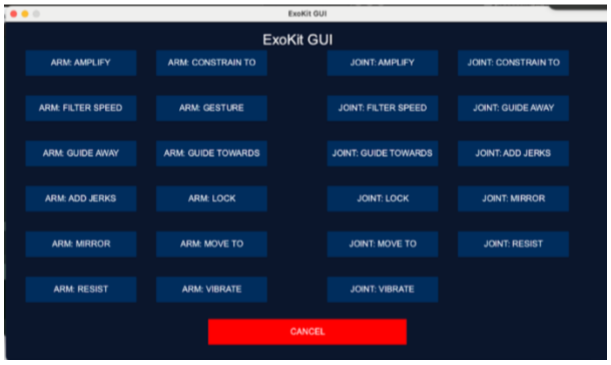
1. **Full-Arm Functions (left cluster):** Functions applied to the entire arm are accessible here.
2. **Individual Joint Functions (right cluster):** Functions applied to a specific joint only.

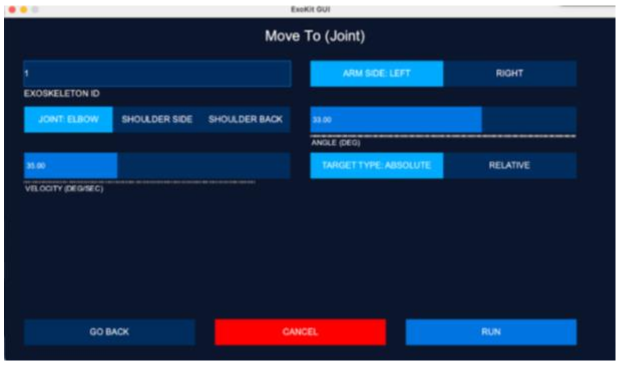
Executing Functions:

- Click on the desired function. A secondary interface will appear (see Figure below, right).
- Adjust the function's parameters as needed:
  - Specify the ID of the calibrated exoskeleton (e.g., exoID 1 in our setup).
  - Select the target arm or joint for the function.
  - The GUI provides default values for the other functions for convenience. Change them as you like.

  💡 **A detailed explanation of the function parameters can be found in the next chapter.**

- Run the function by clicking *Run*.
- The function will execute until you either click *Cancel* or execute a new function.





EXOSKELETON ID: 1
ARM SIDE: LEFT
JOINT: ELBOW
ANGLE (DEG): 33.00
TARGET TYPE: ABSOLUTE
VELOCITY (DEG/SEC): 35.00

💡 *Tip: If you prefer command-line control, ExoKit functions can also be invoked directly using terminal commands in CLION. Refer to the Appendix for detailed instructions.*

5.3 Troubleshooting

Here are some common issues and solutions:

Busy serial port in Processing:

- Check CLION to ensure the port is disconnected.
- Reconnect the port in Processing if necessary.

Commands do not work anymore:

- The system may have triggered an emergency shutdown. This can occur due to several reasons:
    - The calibrated motion ranges were exceeded, and the system automatically performed an emergency shutdown. Reset the Arduino using the restart button and restart the Processing GUI, too.
    - The motors overheated. This can happen if too much load was applied to them. Check for a blinking red light on the motor which indicative of overheating. Resolve this issue by ensuring the motors have cooled down and, resetting the Arduino using the restart button, then restarting the Processing GUI.
    - Loose connections. Sometimes, this problem occurs, for instance, in case of a loose connection with the serial breakout board.

The Wave functionality does not work:

- The wave function only supports fully actuated arms. Ensure all required actuators are connected and functional.

6 Next Steps in Design: Implement Motion Sequences with the Processing Library

The Processing library helps users to define sequences of exoskeleton motions.

6.1 Setup

To leverage ExoKit's Processing Library, follow these steps:

1. **Prepare the Arduino Library (analog to Chapter 5.1):**

    - Copy the provided script (see next page) into PlatformIO.
    - Replace the necessary motor Ids and pin for the emergency shutdown button in the script based on your exoskeleton configuration.
        - In the provided sample script, we calibrated a left-arm exoskeleton with an active elbow joint (motor ID 3). An emergency shutdown button connected to pin 50 on the Arduino Mega. The assigned exoskeleton ID is 1.
        - If you use a different exoskeleton configuration, update the code accordingly. For more details, see Chapter 7.
    - Build and upload the script to the Arduino (cf., Chapter 2.4 Quick Start Guide).
    - Connect to serial#2 and follow the instructions for calibration provided in the Arduino programming interface terminal.
    - After calibration, disconnect the serial port in the Arduino terminal to avoid conflicts in later steps.

```
#include <com/SerialCmdReader.h>
#include "event/simpleactionbuilder/ActionBuilder.h"
#include "exo/ConfiguredEncoder.h"
#include "DynamixelShield.h"
#include "exo/builder/ExoskeletonBuilder.h"
#include "event/simpleactionbuilder/Exoskeleton.h"

DynamixelShield dxl;
ExoskeletonHandle* exoHandle;
SerialCmdReader terminalInteraction(DEBUG_SERIAL);
const float DXL_PROTOCOL_VERSION = 2.0;

void setup() {
   Serial.begin(1000000);
   dxl.begin(10000001);
   dxl.setPortProtocolVersion(DXL_PROTOCOL_VERSION);
   dxl.ping(1);
   DEBUG_SERIAL.begin(115200);

   // TODO change the indicated Dynamixel ID (currently 3) to your
//Dynamixel's ID
  // TODO change the emergency button pin number or remove it if not
//desired
   exoHandle = &ExoskeletonBuilder()
           .addLeftArm()
           .addElbowJoint(dxl, 3, nullptr)
           .finishArm()
           .addEmergencyShutdownButton(50,true)
           .build()
           .calibrate(PREFER_LOAD);
   exoHandle->setId(1);

   terminalInteraction.enableSerialControlCommands();
}

void loop() {
   taskManager.runLoop();
}
```

2. **Configure the Processing GUI:**

   - Open a new Processing file.
   - Import the Processing library. For instance, you can simply drag-and-drop the ExoKit.jar file into the open Processing window.
   - Identify the correct serial port (serial #2) for communication. For example:
     - On **Windows**, the port might be COM11.
     - On **Linux/macOS**, you can find a list of your serial devices using the command “ls /dev/tty.*“.
   - Provide the correct serial port in the code.

Below is an example Processing code that you can use for testing:

```
import processing.serial.*;
int serialBaudRate = 115200;
Serial serial;

void setup() {
  serial = new Serial(this, "/dev/cu.usbserial-14410", 115200);
  Exokit exo = ExokitLibrary.openConnection(serial);

  exo.jointMoveTo(1,  Exokit.ArmSide.LEFT,  Exokit.JointType.ELBOW,  50,  50,
Exokit.MoveToTargetType.ABSOLUTE);

  delay(1000);
  exo.jointMoveTo(1,  Exokit.ArmSide.LEFT,  Exokit.JointType.ELBOW,  -50,  30,
Exokit.MoveToTargetType.RELATIVE);
}

void draw() {
}
```

First, we connect to the serial port ("/dev/cu.usbserial-14410") and then open a connection to ExoKit. Afterwards, we defined a simple motion sequence. In this example, the exoskeleton’s left elbow joint is moved to an angle of 50 degrees with a velocity of 50 degrees/second. The program then waits for one second and then sends another command. This command makes the exoskeleton’s left elbow joint move 50 degrees back with a velocity of 30 degrees/second.

6.2 Usage

You can use ExoKit's Processing library to design and implement motion sequences. For the motion sequences you can use the functions which we present in the following subsections. Most functions offer two variations: Only apply to one joint, or apply to one complete arm. If default values for parameters are given, this indicates the parameter is optional.

Below, we start with an overview of parameters that almost every function uses:

`int exoId`	The ID of the exoskeleton as defined in the complementary Arduino script. The exoID in our provided sample scripts is always `1`.
`ArmSide side`	The targeted arm of the exoskeleton. Options: • `Exokit.ArmSide.LEFT` • `Exokit.ArmSide.RIGHT`
`JointType joint`	The targeted joint of the selected arm. Options: • `Exokit.JointType.ELBOW` • `Exokit.JointType.SHOULDER_SIDE` • `Exokit.JointType.SHOULDER_BACK`
`ExoArmAngles angles`	`angles` is an instance of the ExoArmAngles class. It stores user-specified angles for one exoskeleton arm and comprises the targeted angle for the motors attached to the elbow, shoulder side and shoulder back.

ExoArmAngles

A new object of type `ExoArmAngles` can be created through

```
ExoArmAngles angleConfig1 = new ExoArmAngles();
```

The following functions are offered:

```
public void delElbowAngle()
public void delShoulderSideAngle()
public void delShoulderBackAngle()
```

Deletes a previously defined angle for a joint.

```
public void setElbowAngle(float angle)
public void setShoulderSideAngle(float angle)
public void setShoulderBackAngle(float angle)
```

Sets the angle for a joint.

```
public float getElbowAngle()
public float getShoulderSideAngle()
public float getShoulderBackAngle()
```

Gets the angle for the joint.

```
public bool isHasElbow()
public bool isHasShoulderSide()
public bool isHasShoulderBack()
```

Checks if the angle for the joint has been defined already.

Cancel running functions

Call the following function to cancel the function that that the exoskeleton currently executes:

```
public void cancelRunningAction()
```

Basic Functions

Move To

To move an exoskeleton joint or arm to a desired position with a desired velocity, call the functions below. The motion of a joint can either be (1) absolute w.r.t. a pre-determined 0 reference-position or (2) relative to the user's current position.

These are the signatures for the joint- and arm-based function:

```
public void jointMoveTo(int exoId, ArmSide side, JointType joint, float angle, float velocity, MoveToTargetType targetType);

public void armMoveTo(int exoId, ArmSide armSide, ExoArmAngles angles, float velocity);
```

`float angle` (joints) or `ExoArmAngles angles` (arm)	The angle(s) that the joint/arm should be moved to/by in degrees.
`float velocity`	The velocity that the joint(s) is moved with in degrees/second. ⚠ **Note:** `0` means maximum velocity.
`MoveToTargetType targetType` (joints)	Determines whether the exoskeleton the exoskeleton moves by `angle` degrees (relative to the current position) or to the `angle` (absolute position w.r.t. the calibrated 0 degree angle) Options: • `MoveToTargetType.ABSOLUTE` • `MoveToTargetType.RELATIVE` ⚠ **Note:** For arm motions, all motions are currently interpreted as absolute values.

Lock

Locks the user's joint/arm in place.

These are the signatures for the joint- and arm-based function:

```
public void jointLock(int exoId, ArmSide side, JointType joint);

public void armLock(int exoId, ArmSide side);
```

Scripted Motions

Gesture

Performs a pre-defined gesture.

Function Signature:
```
public void armGesture(int exoId, ArmSide side, Gesture gesture);
```

`Gesture gesture`	Options: • `Exokit.Gesture.WAVE` (performs a wave gesture). • Default = `Exokit.Gesture.WAVE` ⚠ **Note:** The waving function only works for a fully actuated exoskeleton arm!

Vibrate

Gives vibrotactile feedback.

These are the signatures for the joint- and arm-based function:
```
public void jointVibrate(int exoId, ArmSide side, JointType joint, int frequence, float amplitude);

public void armVibrate(int exoId, ArmSide side, int frequency, float amplitude);
```

`int frequency`	Frequency of the vibration in Hz.
`float amplitude`	The amplitude of the vibration in deg.

⚠ **Note:** Don't let the motors run for too long at a very high frequency as this might cause damage.

Motion Transfer

Mirror

Mirrors movements from one arm/joint (**source**) to another arm/joint (**target**) while scaling the movement by a specified factor.

These are the signatures for the joint- and arm-based function:
```
public void jointMirror(int targetExoId, ArmSide targetArmSide, JointType targetJoint, int sourceExoId, ArmSide sourceArmSide, JointType sourceJoint, float scaleFactor);
public void armMirror(int targetExoId, ArmSide targetArmSide, int sourceExoId, ArmSide sourceArmSide, boolean mirrorElbow, boolean mirrorShoulderSide, boolean mirrorShoulderBack, float scaleFactorElbow, float scaleFactorShoulderSide, float scaleFactorShoulderBack);
```

`float scaleFactor` (joint)	The factor by which the movement of the source gets scaled up/down when being replayed on the target. Default = `1.0`
`boolean mirrorElbow` (arm)	Should the elbow be mirrored?
`boolean mirrorShoulderSide` (arm)	Should the shoulder side be mirrored?
`boolean mirrorShoulderBack` (arm)	Should the shoulder back be mirrored?
`float scaleFactorElbow` (arm)	The factor by which Elbow movements get scaled. Default = `1.0`
`float scaleFactorShoulderSide` (arm)	The factor by which shoulder (side) movements get scaled. Default = `1.0`
`float scaleFactorShoulderBack` (arm)	The factor by which shoulder (back) movements get scaled. Default = `1.0`

⚠ **Note:** Be careful with this function. Ensure there are no loose connections with the position encoder. Always test the functionality first on the table/on a mannequin before wearing it yourself. Ensure you are close to the power switch or emergency button. If there are loose connections in the position encoder, sudden peaks in the measurements can occur. As they are replayed at another joint, these sudden jerks might hurt the user.

Modulating the Motion Effort

Effort modulation comprises interaction strategies in which the exoskeleton applies forces in or opposite to a user's inherent motion, while preserving the user's ability to freely navigate along space. The resulting feeling of assistance or resistance can be leveraged by designers to change a user's internal motion perception, with stronger forces resulting in a stronger effect.

Amplify

Makes it easier for the user to move.

These are the signatures for the joint- and arm-based function:

```
public void jointAmplify(int exoId, ArmSide side, JointType joint, float torquePercentage, JointMovementDirection amplifyDirection, int startingVelocity, int maxVelocity);

public void armAmplify(int exoId, ArmSide side, float torquePercentage, JointMovementDirection amplifyDirection, int startingVelocity, int maxVelocity);
```

`float torquePercentage`	Percentage (from 0 to 1) of the motor's maximum available torque which will be used to amplify the user's motion.
`JointMovementDirection amplifyDirection`	The exoskeleton will only amplify the motion if the user moves in the indicated direction. Options: • `Exokit.JointMovementDirection.BOTH` • `Exokit.JointMovementDirection.ABDUCTION_OR_FLEXION` • `Exokit.JointMovementDirection.ADDUCTION_EXTENSION`
`int startingVelocity`	The threshold velocity in deg/sec with which the user should move the joint/arm to trigger the amplification. Can be used to fine-tune the sensitivity of the function. Default = `30`
`int maxVelocity`	The maximum velocity in deg/sec to which the function will accelerate the arm. Default = `0` ⚠ **Note:** 0 means unlimited.

Resist

Makes it harder for the user to move.

These are the signatures for the joint- and arm-based function:

```
public void jointResist(int exoId, ArmSide side, JointType joint, float torquePercentage, JointMovementDirection resistDirection, float minVelocity);

public void armResist(int exoId, ArmSide side, float torquePercentage, JointMovementDirection resistDirection, int minVelocity);
```

`float torquePercentage`	Percentage (from 0 to 1) of the motor's maximum available torque which will be used to resist the user's motion.
`JointMovementDirection resistDirection`	The exoskeleton will only resist the motion if the user moves in the indicated direction. Options: • `Exokit.JointMovementDirection.BOTH` • `Exokit.JointMovementDirection.ABDUCTION_OR_FLEXION` • `Exokit.JointMovementDirection.ADDUCTION_EXTENSION`

	Default = `Exokit.JointMovementDirection.BOTH`
`int minVelocity`	The minimum velocity in deg/sec to which the function will slow down the arm. Default = `0`

Modulating the Motion Style

Motion style refers to interaction strategies that modify the observable characteristics of a user's movement, such as speed or path, while keeping the user in control of the overall motion. These style modifications can range from subtle adjustments to more pronounced changes.

Filter Speed

Keeps the speed of the user's motion within a desired range. The exoskeleton will amplify the user's motion if the user is slower than desired and resist if they are faster.

These are the signatures for the joint- and arm-based function:

```
public void jointFilterSpeed(int exoId, ArmSide side, JointType joint, float minSpeed float maxSpeed, JointMovementDirection onDirection, float amplifyFlexTorquePercentage, float resistFlexTorquePercentage, float amplifyExtensionTorquePercentage, float resistExtensionTorquePercentage);

public void armFilterSpeed(int exoId, ArmSide side, float minSpeed, float maxSpeed, JointMovementDirection onDirection, float amplifyFlexTorquePercentage, float resistFlexTorquePercentage, float amplifyExtensionTorquePercentage, float resistExtensionTorquePercentage);
```

`float minSpeed`	The minimum speed in degrees/second at which the joint should move. If the joint's speed is slower, the exoskeleton will amplify.
`float maxSpeed`	The maximum speed in degrees/second at which the joint should move. If the joint's speed is faster, the exoskeleton will resist.
`JointMovementDirection onDirection`	The exoskeleton will only filter the speed if the user moves in the indicated direction. Options: • `Exokit.JointMovementDirection.BOTH` • `Exokit.JointMovementDirection.ABDUCTION_OR_FLEXION` • `Exokit.JointMovementDirection.ADDUCTION_EXTENSION` Default = `Exokit.JointMovementDirection.BOTH`
`float amplifyFlexTorquePercentage`	Percentage (from 0 to 1) of the motor's maximum torque which will be used to amplify the motion. Will be

	applied if the joint moves too slowly and the user is doing a flexion/abduction movement. Default = `0.05`
`float resistFlexTorquePercentage`	Percentage (from 0 to 1) of the motor's maximum torque which will be used to provide resistance. Will be applied if the joint moves too high and the user is doing a flexion/abduction movement. Default = `0.05`
`float amplifyExtensionTorquePercentage`	Percentage (from 0 to 1) of the motor's maximum torque which will be used to amplify the motion. Will be applied if the joint moves too slowly and the user is doing an extension/adduction movement. Default = `0.05`
`float resistExtensionTorquePercentage`	Percentage (from 0 to 1) of the motor's maximum torque which will be used to provide resistance. Will be applied if the joint moves too fast and the user is doing an extension/adduction movement. Default = `0.05`

Jerk

Makes the user's motion jerky: Twitches consisting of forth-and-back motions of a randomized size and within randomized time intervals will be added to the user's motion.

These are the signatures for the joint- and arm-based function:

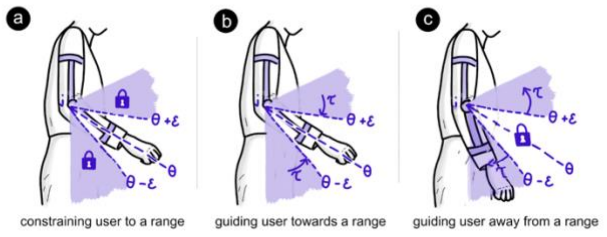
```
public void jointJerk(int exoId, ArmSide side, JointType joint, float minJerkAngle, float maxJerkAngle, int minJerkIntervalMs, int maxJerkIntervalMs, float maxAccumulatedMovementsLeft, float maxAccumulatedMovementsRight, float velocity, int nrJerks);

public void armJerk(int exoId, ArmSide side, float minJerkAngle, float maxJerkAngle, int minJerkIntervalMs, int maxJerkIntervalMs, float maxAccumulatedMovementsLeft, float maxAccumulatedMovementsRight, float velocity, int nrJerks);
```

`float minJerkAngle`	The minimum size of a jerk in degrees.
`float maxJerkAngle`	The maximum size of a jerk in degrees.
`int minJerkIntervalMs`	The minimum time in between two jerks in milliseconds. ⚠ **Note:** Timer runs only if joint/arm is moving

`int maxJerkIntervalMs`	The maximum time in between two jerks in milliseconds. ⚠ **Note:** Timer runs only if joint/arm is moving
`float maxAccumulatedMovementsLeft`	How far should jerks be able to move the joint/arm to the left if all jerks are summed up in degrees.
`float maxAccumulatedMovementsRight`	How far should jerks be able to move the joint/arm to the right if all jerks are summed up in degrees.
`float velocity`	The velocity of the jerks in deg/sec. Default = `0.0` ⚠ **Note:** 0 means unlimited.
`int nrJerks`	The number of jerks that will be performed. Default = `0` ⚠ **Note:** 0 means unlimited.

⚠ **Note:** Jerks only occur if the arm is in motion.

Motion Guidance


a
θ+ε
θ-ε
θ
constraining user to a range
b
τ
guiding user towards a range
c
guiding user away from a range

Motion guidance comprises strategies in which the exoskeleton applies forces that correct, guide, or constrain a user's movement. As the forces increase and the unconstrained range of motion decreases, the user's motion becomes more controlled, up to being fully moved or locked. Conversely, weaker forces and larger unconstrained areas allow the user to move more freely.

Constrain To

Restricts a user's motion range to a specified area. The area is determined by a range ϵ (here called radius) around a center angle θ.

These are the signatures for the joint- and arm-based function:

```
public void jointConstrainTo(int exoId, ArmSide side, JointType joint, float angle, float radius);

public void armConstrainTo(int exoId, ArmSide side, ExoArmAngles angles, float radius);
```

`float angle` (joint) or `ExoArmAngles angles` (arms)	The center of the area in degree.

`float radius`	The radius around the center angle in degrees.

Guide Away

Guides a user's away from a specified area. The area is determined by a range ϵ (here called radius) around a center angle θ. The user cannot enter the area at all.

These are the signatures for the joint- and arm-based function:

```
public void jointGuideAway(int exoId, ArmSide side, JointType joint, float angle, float radius, float amplifyTorquePercentage, float resistTorquePercentage);

public void armGuideAway(int exoId, ArmSide side, ExoArmAngles angles, float radius, float amplifyTorquePercentage, float resistTorquePercentage);
```

`float angle` (joint) or `ExoArmAngles angles`(arms)	The center of the area in degree.
`float radius`	The radius around the center angle in degrees.
`float amplifyTorquePercentage`	Percentage (from 0 to 1) of the motor's maximum torque which will be used to amplify the motion. Will be applied if the joint moves away from the forbidden area. Default = `0.05`
`float resistTorquePercentage`	Percentage (from 0 to 1) of the motor's maximum torque which will be used to resist the motion. Will be applied if the joint moves towards the forbidden area. Default = `0.05`

Guide Towards

Guides a user towards a specified area. The area is determined by a range ϵ (here called radius) around a center angle θ. Within the area, the user can freely move.

These are the signatures for the joint- and arm-based function:

```
public void jointGuideTowards(int exoId, ArmSide side, JointType joint, float angle, float radius, float amplifyTorquePercentage, float resistTorquePercentage);

public void armGuideTowards(int exoId, ArmSide side, ExoArmAngles armAngles, float radius, float amplifyTorquePercentage, float resistTorquePercentage);
```

`float angle` (joint) or	The center of the area in degree.

`ExoArmAngles angles(`arms)	
`float radius`	The radius around the center angle in degrees.
`float amplifyTorquePercentage`	Percentage (from 0 to 1) of the motor's maximum torque which will be used to amplify the motion. Will be applied if the joint moves towards the desired area. Default = `0.05`
`float resistTorquePercentage`	Percentage (from 0 to 1) of the motor's maximum torque which will be used to resist the motion. Will be applied if the joint moves away from the desired area. Default = `0.05`

6.3 Troubleshooting

Here are some common issues and solutions:

Busy serial port in Processing:

- Check CLION to ensure the port is disconnected.
- Reconnect the port in Processing if necessary.

Commands do not work anymore:

- The system may have triggered an emergency shutdown. This can occur due to several reasons:
  - The calibrated motion ranges were exceeded, and the system automatically performed an emergency shutdown. Reset the Arduino using the restart button and restart the Processing GUI, too.
  - The motors overheated. This can happen if too much load was applied to them. Check for a blinking red light on the motor which indicative of overheating. Resolve this issue by ensuring the motors have cooled down and, resetting the Arduino using the restart button, then restarting the Processing GUI.
  - Loose connections. Sometimes, this problem occurs, for instance, in case of a loose connection with the serial breakout board.

The Wave functionality does not work:

- The wave function only supports fully actuated arms. Ensure all required actuators are connected and functional.

7 Deep Dive: Implementing Complex Interactions with the Arduino Library

The Arduino programming interface is recommended for more advanced users that are familiar with the working principles of the pre-implemented functionalities and/or want to define more interactive behavior. To provide a powerful interface, it relies on ist own control structure on which we detail in this chapter:

7.1 Core Structure

The following code provides a scaffold to initialize libraries that ExoKit depends on. This is the main.cpp of a standard Arduino script using ExoKit:

```
// main.cpp

#include "TaskManagerIO.h"
#include "util/DebugSerial.h"
#include "com/SerialCmdReader.h"
#include "DynamixelShield.h"

const float DXL_PROTOCOL_VERSION = 2.0;
SerialCmdReader cmdReader(DEBUG_SERIAL);
DynamixelShield dxl;

void setup() {
   dxl.begin(1000000l);
   dxl.setPortProtocolVersion(DXL_PROTOCOL_VERSION);
   dxl.ping(1);

   DEBUG_SERIAL.begin(115200);

  // <EXOSKELETON CONFIGURATION & CALIBRATION (CHAPTER 7.2)>
  // <CONDITION-ACTION PROGRAMMING (CHAPTER 7.3)>
}

void loop() {
   taskManager.runLoop();
}
```

setup()-function

The setup function is the function in which you will define most of the exoskeleton's interactive behavior.
Before doing this, we first must setup up the communication with the motors and the PC. In order to be able to use Dynamixel servos, the Dynamixel shields needs to be set up, which includes setting the baudrate for the communication between the motors and the microcontroller. The

Arduino Mega can communicate with the servos using a bauddrate of up to 1.000.000 bd. Next we enable the DEBUG_SERIAL console, which is used to exchange information between a PC and the exoskeleton. It communicates over Serial #2 (Pin 16 and 17) on an Arduino Mega.
Afterwards follows the code with which the physical exoskeleton setup will be registered and calibrated (see Chapter 7.2). Then the interactive behavior will be programmed (see Chapter 7.3).

loop()-function

ExoKit runs all actions and conditions using tasks that are managed by TaskManagerIO and registered in the setup() function. For this reason, the only method called in the loop()-function is taskManager.runLoop(), which schedules and executes ExoKits functions.

7.2 Exoskeleton Configuration and Calibration

```
#include <com/SerialCmdReader.h>
#include "event/simpleactionbuilder/ActionBuilder.h"
#include "exo/ConfiguredEncoder.h"
#include "DynamixelShield.h"
#include "exo/builder/ExoskeletonBuilder.h"
#include "event/simpleactionbuilder/Exoskeleton.h"

DynamixelShield dxl;
ExoskeletonHandle* exoHandle;
Exoskeleton* exoskeleton;
ActionBuilder ab;
const float DXL_PROTOCOL_VERSION = 2.0;
ExoArmAngles angleConfig1;

void setup() {
    Serial.begin(1000000);
    dxl.begin(1000000l);
    dxl.setPortProtocolVersion(DXL_PROTOCOL_VERSION);
    dxl.ping(1);

    DEBUG_SERIAL.begin(115200);

    exoHandle = &ExoskeletonBuilder()
        .addLeftArm()                      // returns an ArmBuilder& object
            .addElbowJoint(dxl, 1, nullptr) // returns an ArmBuilder& object
            .addShoulderSideJoint(dxl, 2, nullptr)
            .addShoulderBackJoint(dxl, 3, nullptr)
            .finishArm()                    // returns an ExoBuilder& object
        .addEmergencyShutdownButton(50,true)// returns an ExoBuilder& object
        .build()                   // returns an ExoskeletonCalibrator& object
        .calibrate(PREFER_LOAD);       // returns an ExoskeletonHandle& object

    exoskeleton = new Exoskeleton{ab, *exoHandle}; // create the exoskeleton

    // <CONDITION-ACTION PROGRAMMING (CHAPTER 7.3)>
}

void loop() {
    taskManager.runLoop();
}
```

The code snipped above shows an example on how to configure an exoskeleton instance. It comprises one left arm with three active joints and an emergency shutdown button.

For an example of how to configure an exoskeleton with a load cell or a position encoder, see the example scripts in Appendix B and C.

The following methods are available for configuring the exoskeleton:

ExoskeletonBuilder

```
ArmBuilder& ExoskeletonBuilder::addLeftArm();
```

```
ArmBuilder& ExoskeletonBuilder::addRightArm();
```
Starts the configuration of the selected arm.

```
ExoskeletonBuilder&
ExoskeletonBuilder::addEmergencyShutdownButton(uint8_t buttonPin, bool
isPullUp);
```
Configures the emergency shutdown button. If the corresponding button is pressed, the exoskeleton disables all configured motors and shuts down.

```
ExoskeletonCalibrator ExoskeletonBuilder::build();
```
Completes the configuration step. It returns an ExoskeletonCalibrator object.

ArmBuilder

Given an ArmBuilder object, joints can be added to the arm. To add an active joint (a joint with a motor) to the arm, use the following function:

```
ArmBuilder& ArmBuilder::addElbowJoint(DynamixelShield& dxl, uint8_t
dxl_id, HX711* loadCell = nullptr, uint8_t lower_mechanical_restriction
= 0, uint8_t upper_mechanical_restriction = 0);

ArmBuilder& ArmBuilder::addShoulderSideJoint(DynamixelShield& dxl,
uint8_t dxl_id, HX711* loadCell = nullptr, uint8_t
lower_mechanical_restriction = 0, uint8_t upper_mechanical_restriction
= 0);

ArmBuilder& ArmBuilder::addShoulderBackJoint(DynamixelShield& dxl,
uint8_t dxl_id, HX711* loadCell = nullptr, uint8_t
lower_mechanical_restriction = 0, uint8_t upper_mechanical_restriction
= 0);
```

`DynamixelShield& dxl`	A pointer to the the instance of the Dynamixel shield to which the motors are attached.
`uint8_t dxl_id`	The ID motor attached to the joint module.
`HX711* loadCell`	A loadcell object. Default = `nullptr`
`uint8_t lower_mechanical_restriction`	The degree (positive) by which the joint's range of motion has been reduced at the lower end of the joint module through a mechanical restriction. Default = `0`
`uint8_t upper_mechanical_restriction`	The degree (positive) by which the joint's range of motion has been reduced at the lower end of the joint module through a mechanical restriction.

	Default = 0

To add a passive joint (either without any sensor and actuator or with a position encoder), use the following functions:

```
ArmBuilder& ArmBuilder::addElbowJoint(ConfiguredEncoder* encoder = nullptr, HX711* loadCell = nullptr, uint8_t lower_mechanical_restriction = 0, uint8_t upper_mechanical_restriction = 0);

ArmBuilder& ArmBuilder::addShoulderSideJoint(ConfiguredEncoder* encoder = nullptr, HX711* loadCell = nullptr, uint8_t lower_mechanical_restriction = 0, uint8_t upper_mechanical_restriction = 0);

ArmBuilder& ArmBuilder::addShoulderBackJoint(ConfiguredEncoder* encoder = nullptr, HX711* loadCell = nullptr, uint8_t lower_mechanical_restriction = 0, uint8_t upper_mechanical_restriction = 0);
```

Configures the elbow joint as a passive joint. All parameters are optional.

`ConfiguredEncoder* encoder`	A potentiometer object that is used to read the joint's position and velocity. Default = `nullptr`
`HX711* loadCell`	A loadcell object. Default = `nullptr`
`uint8_t lower_mechanical_restriction`	The degree (positive) by which the joint's range of motion has been reduced at the lower end of the joint module through a mechanical restriction. Default = 0
`uint8_t upper_mechanical_restriction`	The degree (positive) by which the joint's range of motion has been reduced at the lower end of the joint module through a mechanical restriction. Default = 0

To complete the configuration process of the current arm, call:

```
ExoskeletonBuilder& ArmBuilder::finishArm();
```

ExoskeletonCalibrator

The calibrate-method starts the calibration of the exoskeleton and returns an ExoskeletonHandle. During the calibration the 0-degree positions of all joints are configured and persisted in the Arduino's EEPROM.
The DEBUG_SERIAL console is used to guide the user through the calibration process.

```
ExoskeletonHandle& calibrate(ConfigLoadBehaviour configLoadBehaviour =
ASK_CREATE);
```

There are three different options for the calibration process:

`ConfigLoadBehaviour.CREATE`	Triggers a calibration after each restart of the Arduino. It does not ask the user if they want to persist the created calibration. Hence, this mode always requires interaction through the serial console and is not suited for mobile usage in which the Arduino is not connected to the PC's serial port.
`ConfigLoadBehaviour.ASK_CREATE`	If the Arduino is restarted, the system asks if the user wants to perform a new calibration of the exoskeleton, or if the persisted calibration parameters should be applied. As the previous mode, it requires interaction with the serial console.
`ConfigLoadBehaviour.PREFER_LOAD`	Automatically loads the persisted calibration parameters from the EEPROM. If the persisted parameters were determined for a different configuration, the system asks the user to perform a new calibration. This mode does not require no interaction with the serial console if configured once for the correct exoskeleton configuration. Hence, this mode should be used for a mobile setup.

7.3 Defining the Program Flow With Condition-Action Programming

ExoKit follows an event-based architecture, in which interactive behavior is modeled with `Condition` and `Action` classes (see Chapters 7.4 and 7.5). Conditions serve as runtime triggers that control the flow of actions, evaluating to true or false based on the exoskeleton's current state. This evaluation is checked in a control loop that runs at a fixed frequency. Actions encapsulate basic functions and augmentation strategies (see prior Chapters). ExoKit allows to combine the actions sequentially or in parallel either for individual joints, a set of joints, or the arm for complex control flows.

The latter is achieved with the help of two classes: `ActionBuilder` and `Exoskeleton`.

After the desired condition-action sequences are defined, they need to be dispatched, using the ActionBuilders dispatch function:

```
ActionBuilder::dispatch(ICondition& condition, unsigned long loopInterval, boolean multipleTrigger)
```

`ICondition& condition`	The dispatch function receives a condition. The dispatched sequence is triggered if the condition evaluates to true.
`unsigned long loopInterval`	Defines the frequency in milliseconds in which the task that runs the action sequence will be called. A smaller number means that the exoskeleton reacts faster to changes as it checkst he condition more often.
`boolean multipleTrigger`	After the dispatched function has finished, this parameter decides whether the function can be triggered again if the condition evaluates to true. If false, then once the function was triggered, it won't be executed again even if the condition would evaluate to true.

An example scaffold:

```
#include <com/SerialCmdReader.h>
#include "event/simpleactionbuilder/ActionBuilder.h"
#include "exo/ConfiguredEncoder.h"
#include "DynamixelShield.h"
#include "exo/builder/ExoskeletonBuilder.h"
#include "event/simpleactionbuilder/Exoskeleton.h"

DynamixelShield dxl;
ExoskeletonHandle* exoHandle;
Exoskeleton* exoskeleton; // create a global Exoskeleton object
ActionBuilder ab; // create a global ActionBuilder object
const float DXL_PROTOCOL_VERSION = 2.0;

void setup() {
    Serial.begin(1000000);
    dxl.begin(1000000l);
    dxl.setPortProtocolVersion(DXL_PROTOCOL_VERSION);
    dxl.ping(1);

    DEBUG_SERIAL.begin(115200);

    // <EXOSKELETON CONFIGURATION & CALIBRATION (CHAPTER 7.2)>

    exoskeleton = new Exoskeleton{ab, *exoHandle}; // initialize the
                                                      exoskeleton        object
    // <CONDITION-ACTION PROGRAMMING (CHAPTER 7.3)>

    ab.dispatch(ab.trueCondition(), 80); // the sequence is immediately
                                          executed. It runs at a frequency
                                           of         80         milliseconds
}

void                                loop()                               {
    taskManager.runLoop();
}
```

Sequential execution of actions

Sequences are programmed the same way as one would write a normal program. The classes ActionBuilder and Exoskeleton provide various methods can be used to program the exoskeletons behaviour.
Calling one behaviour defining method `A` first and another method `B` second, means that the behaviour defined by `B` executes after `A` finished.

Example:

```
Exoskeleton->getLeftArm().getElbow().resist(0.05,
 ExoskeletonJointHandle::BOTH,0, ab.timeoutCondition(5000));

ab.delay(5000);

exoskeleton->getLeftArm().getElbow().amplify(0.05,
  ExoskeletonJointHandle::BOTH);
```

The code above first lets the exoskeleton resist elbow movements for 5 seconds, then does nothing for 5 seconds and amplifies elbow movements afterwards.

Parallel execution of actions

The parallel block makes the exoskeleton execute two functions in parallel, which is for example useful if two different joints should perform two different functions at the same time.
The parallel block finishes if both actions defined in the two parallel functions are finished.

```
void      parallel(void      (*firstParallel)(ActionBuilder&),      void
(*secondParallel)(ActionBuilder&));
```

Example:

```
void firstParallelBlock(ActionBuilder& ab) {
   exoskeleton->getLeftArm().getShoulderSide().moveToActive(60, 20);
}

void secondParallelBlock(ActionBuilder& ab) {
   exoskeleton->getLeftArm().getElbow().moveToActive(30, 20);
}

void setup() {
    <SETUP/CONFIGURATION/CALIBRATION>

   ab.parallel(firstParallelBlock, secondParallelBlock);
   ab.dispatch(ab.trueCondition(), 1, true);
}
```

If-then-else

The ifThenElse block takes a condition and two functions defining the then and else blocks. If the condition evaluates to true, the sequential behaviour that is defined in the thenBlock function is executed. Otherwise the behaviour defined in the elseBlock function is executed.

```
void          ifThenElse(ICondition&          condition,          void
(*thenBlock)(ActionBuilder&), void (*elseBlock)(ActionBuilder&));
```

Example:

```
void thenBlock(ActionBuilder& ab) {
   exoskeleton->getLeftArm().getElbow().moveToActive(60, 20);
}

void elseBlock(ActionBuilder& ab) {
   exoskeleton->getLeftArm().getElbow().moveToActive(30, 20);
}


void setup() {
       <SETUP/CONFIGURATION/CALIBRATION>

   ab.ifThenElse(ab.trueCondition(), thenBlock, elseBlock);
   ab.dispatch(ab.trueCondition(), 1, true);
}
```

Loop

The loop block corresponds to a while call. It takes a loopCondition and one function, the inLoopBlock, comprising a sequence of actions that corresponds to the body of the loop. If the loopCondition evaluates to true, the actions defined in the inLoopBlock are executed. Just like a while block, this loop block checks the loopCondition again after the code in the inLoopBlock-function finishes. If the condition evaluates to false, the loop block finishes.

```
void          loop(ICondition&          loopCondition,          void
(*inLoopBlock)(ActionBuilder&));
```

Example:
```
void loopBlock(ActionBuilder& ab) {
   exoskeleton->getLeftArm().getElbow().moveToActive(30, 20);
   exoskeleton->getLeftArm().getElbow().moveToActive(60, 20);
}

void setup() {
       <SETUP/CONFIGURATION/CALIBRATION>

   ab.loop(ab.trueCondition(), loopBlock);
   ab.dispatch(ab.trueCondition(), 1, true);
}
```

The code above would let the exoskeleton move the user's elbow up and down in a never ending loop.

Breakable

The breakable block continuously evaluates the breakIf condition while executing the actions defined in the block-function. If the breakIf condition evaluates to true, the in the block-function defined action gets aborted mid run.
```
void breakableBlock(ICondition& breakIf, void (*block)(ActionBuilder&));
```

Important function calls and classes

As in the Processing library, indicating the ExoID, targeted arm side of the exoskeleton as well as the joint are important for telling the API where it should execute an action and monitor conditions.

Getting the arm and joint objects

Getting the arm and joint objects is important for defining conditions and actions. The ExoskeletonArm object contains all actions that can be performed directly on an arm. The ExoskeletonJoint object contains all actions and conditions that can be performed or evaluated on a joint.
To get these objects, you can use the following functions:

```
ExoskeletonArm &Exoskeleton::getLeftArm()
```
Gets the left arm of an exoskeleton.

```
ExoskeletonArm &Exoskeleton::getRightArm()
```
Gets the right arm on an exoskeleton.

```
ExoskeletonJoint& ExoskeletonArm::getElbow();
```
Gets the elbow joint from an arm.

```
ExoskeletonJoint& ExoskeletonArm::getShoulderBack();
```
Gets the shoulder back joint of an arm.

```
ExoskeletonJoint& ExoskeletonArm::getShoulderSide();
```
Gets the shoulder side joint of an arm.

ExoArmAngles

As in the Processing library, `ExoArmAngles` is a class that helps the user store the goal angles for an exoskeleton arm.

Here methods for the elbow joints are getting explained, but methods for ShoulderSide and ShoulderBack work the same:

```
void ExoArmAngles::setElbowAngle(float angle)
```
Sets the angle for the elbow.

```
float ExoArmAngles::getElbowAngle()
```
Gets the angle for the elbow.

```
bool ExoArmAngles::isHasElbow()
```
Checks if the angle for the elbow has been defined.

```
void ExoArmAngles::delElbowAngle()
```
Deletes a previously defined angle for the elbow.

Example:

```
#include <com/SerialCmdReader.h>
#include "event/simpleactionbuilder/ActionBuilder.h"
#include "exo/ConfiguredEncoder.h"
#include "DynamixelShield.h"
#include "exo/builder/ExoskeletonBuilder.h"
#include "event/simpleactionbuilder/Exoskeleton.h"

DynamixelShield dxl;
ExoskeletonHandle* exoHandle;
Exoskeleton* exoskeleton;
ActionBuilder ab;
const float DXL_PROTOCOL_VERSION = 2.0;
ExoArmAngles angleConfig1;

void setup() {
    Serial.begin(1000000);
    dxl.begin(10000001);
    dxl.setPortProtocolVersion(DXL_PROTOCOL_VERSION);
    dxl.ping(1);
    DEBUG_SERIAL.begin(115200);

    // TODO change the indicated Dynamixel ID (currently 1, 2, ,3) to your
Dynamixel's ID
    exoHandle = &ExoskeletonBuilder()
            .addLeftArm()
            .addElbowJoint(dxl, 1, nullptr)
            .addShoulderSideJoint(dxl, 2, nullptr)
            .addShoulderBackJoint(dxl, 3, nullptr)
            .finishArm()
            .addEmergencyShutdownButton(50,true)
            .build()
            .calibrate(PREFER_LOAD);
    exoskeleton = new Exoskeleton{ab, *exoHandle};

    angleConfig1.setElbowAngle(10);
    angleConfig1.setShoulderSideAngle(10);
    angleConfig1.setShoulderBackAngle(10);
    exoskeleton->getLeftArm().moveTo(angleConfig1,20, nullptr);

    ab.dispatch(ab.trueCondition(), 80);
}

void loop() {
    taskManager.runLoop();
}
```

7.4 Conditions

Conditions are used for various purposes within ExoKit‘s API. For example, they can be used to trigger an action sequence, to end the execution of an action or as deciders for control flow blocks. Conditions evaluate at runtime and return boolean values “True” or “False”. They can be composed using boolean operations.

Conditions that observe a joint or an arm can be generated using the ExoskeletonJoint object. Example:

```
exoskeleton->getLeftArm().getElbow().isAtPosition(/* ..*/);
```

The code above generates a condition that checks if the motor at the left elbow is at a specific position.

Conditions that are not related to a specific joint or arm are generated using the ActionBuilder object. Example:

```
ab.trueCondition()
```

The above example always evaluates to true.

To compose conditions, the boolean operators “!”, “||” and “&&” are used directly in the code. Example:

```
ICondition& c = exoskeleton->getLeftArm().getElbow().isAtPosition(/*
..*/) && !exoskeleton->getLeftArm().getElbow().isMoving();
```

The above example evaluates to true if the elbow is at a specific position and is not moving.

true

Always evaluates to true.

```
ICondition& ActionBuilder::trueCondition();
```

false

Always evaluates to false.

```
ICondition& ActionBuilder::falseCondition();
```

timeout

After being checked for the first time, the condition starts a timer. It evaluates to true after the timer expires. The conditions resets when the action finishes. The same instance can be used for multiple actions/triggers.

```
ICondition& ActionBuilder::timeoutCondition(unsigned long wait_ms);
```

`unsigned long wait_ms`	The time which the condition should wait in milliseconds.

manualCondition

A condition that can be set manually either through custom code or by using a custom command that can be sent through the serial console. The latter is useful to trigger actions through an external application that sends out user-defined commands through the serial port. To use this condition, create an instance of *ManualConditionTriggerCmd* with a command name and the condition as a parameter and an instance of SerialCmdReader. Register the manual condition trigger using the *add*-function of your instance of the *SerialCmdReader* .

```
ICondition& ActionBuilder::manualCondition(bool unfulfillOnRestore = true, bool armOnFirstEval = false);
```

`bool unfulfillOnRestore`	Determines if the condition should be set back to false if the action it triggered has finished.
`bool armOnFirstEval`	Determines if the condition should ignore it being set to 'true' if the action it was assigned to didn't evaluate it yet. An action evaluates a condition at its start and continuously while it runs.

isVelocity

Compares the velocity of a joint to a user-defined threshold or range.

```
VelocityCondition& ExoskeletonJoint::isVelocity(SpeedCompareType compare, float degreePerSecond, float tolerance = 0)
```

`SpeedCompareType compare`	The compare operation that is used. The measured sensor data of the joint is always on the left side of the operation and the threshold `degreePerSecond` is always on the right side. Options: • `FASTER_THAN` • `SLOWER_THAN` • `WITHIN` E.g., the function reads as 'the current velocity of the joint is `FASTER_THAN` the user-specified threshold `degreePerSecond`'
`float degreePerSecond`	The value to compare the joint's velocity with in degrees/second. If `compare == WITHIN`, this value indicates the center of the range in which the condition should evaluate to true.
`float tolerance`	The tolerance for `degreePerSecond` in degrees/second. If `compare == WITHIN`, this defines the size of the range around the center in which the condition should evaluate to true. Default = `0`

isSpeed

Compares the speed of a joint to a user-defined threshold or range.

```
SpeedCondition& ExoskeletonJoint::isSpeed(SpeedCompareType compareType,
float thresholdDegreePerSecond, float tolerance = 0);
```

`SpeedCompareType compare`	The compare operation that is used. The measured sensor data of the joint is always on the left side of the operation and the threshold `degreePerSecond` is always on the right side. Options: • `FASTER_THAN` • `SLOWER_THAN` • `WITHIN`
`float degreePerSecond`	The value to compare the joint's velocity with in degrees/second. If `compare == WITHIN`, this value indicates the center of the range in which the condition should evaluate to true.
`float tolerance`	The tolerance for `degreePerSecond` in degrees/second. If `compare == WITHIN`, this defines the size of the range around the center in which the condition should evaluate to true. Default = `0`

isMovingTowards

Evaluates to true if a joint is moving towards the specified direction.

```
IsMovingTowardCondition&
ExoskeletonJoint::isMovingTowards(ExoskeletonJointHandle::MovementDire
ction direction, float thresholdVelocity = 5);
```

`ExoskeletonJointHandle::MovementDirection direction`	The movement direction that the condition should observe. Options: • `ExoskeletonJointHandle::MovementDirection.BOTH` • `ExoskeletonJointHandle::MovementDirection.ABDUCTION_OR_FLEXION` • `ExoskeletonJointHandle::MovementDirection.ADDUCTION_EXTENSION`
`float thresholdVelocity`	How fast the joint needs to move at a minimum in degrees/second to be considered as moving. Default = `5`

isMoving

Evaluates to true if the joint is moving with a speed greater than 0 deg/sec.

```
IsMovingCondition& ExoskeletonJoint::isMoving();
```

isAcceleration

Compares the acceleration of a joint to a user-defined threshold or range.

```
AccelerationCondition&
ExoskeletonJoint::isAcceleration(SpeedCompareType  compareType,  float
accleration, float tolerance = 0);
```

`SpeedCompareType compare`	The compare operation that is used. The value of the joint is always on the left side of the operation and thresholdDegreePerSecond is always on the right side. Options: • `FASTER_THAN` • `SLOWER_THAN` • `WITHIN`
`float acceleration`	The value to compare the joint's velocity with in degrees/second². If `compare == WITHIN`, this value indicates the center of the range in which the condition should evaluate to true.
`float tolerance`	The tolerance for `acceleration` in degrees/second². If `compare == WITHIN`, this defines the size of the range around the center in which the condition should evaluate to true. Default = `0`

isAccelerating

Evaluates to true if the joint's speed is increasing.

```
IsAcceleratingCondition&        ExoskeletonJoint::isAccelerating(float
accelerationThreshold = 1);
```

`float accelerationThreshold`	How much the joint needs to accelerate at a minimum in degrees/second² to be considered as accelerating. Default = `1`

isForce

Compares the force measurement of a load cell registered with this joint to a user-defined threshold or range.

```
ForceCondition&
ExoskeletonJoint::isForce(ForceCondition::ForceCompareType compareType,
float force, float tolerance = 0);
```

`ForceCondition:: ForceCompareType compareType`	Options: • `STRONGER_THAN` • `WEAKER_THAN`

	• `WITHIN`
`float force`	The value to compare the applied force. If `compare == WITHIN`, this value indicates the center of the range in which the condition should evaluate to true.
`float tolerance`	The tolerance for `force`. If `compare == WITHIN`, this defines the size of the range around the center in which the condition should evaluate to true. Default = `0`

isAtPosition

Evaluates to true if the joint is in the position of the specified angle

```
AtPositionCondition&        ExoskeletonJoint::isAtPosition(CompareType
compareType, float angle, float tolerance);
```

`CompareType compareType`	Options: • `EQUAL` • `LOWER_THAN` • `HIGHER_EQUAL_THAN`
`float angle`	The absolute angle in degrees to compare to.
`float tolerance`	The tolerance range in degrees around the goal angle in which the joint should still being considered to be in the position.

isAtExoArmAngle

Evaluates to true if all arm joints capable of tracking are in the desired position for a specific duration. This can be used for pose detection.

```
ICondition& ExoskeletonArm::isAtExoArmAngle(ExoArmAngles  exoArmAngle,
float tolerance, unsigned long duration);
```

`ExoArmAngles exoArmAngle`	The angles of the arm.
`float tolerance`	The tolerance range in degrees around the goal angle in which the joint should still being considered to be in the position.
`unsigned long duration`	How long the users needs to stay in the position before the conditions evaluates to true in milliseconds.

7.5 Actions

As for the conditions, the actions can be called by their corresponding functions offered by the joint object.

The actions do the same as in the Processing library. Some just offer access to more optional arguments. These allow you to further fine-tune the interaction.

Common arguments

These are arguments that are often found in action calls.

`ICondition* alternativeFinishCondition`	Condition that if it evaluates to true finishes the action prematurely

WaitFor

Waits for the specified condition to evaluate to true.

```
void ActionBuilder::waitFor(ICondition& condition);
```

`ICondition& condition`	The condition to wait for.

Delay

Waits for the specified number of milliseconds

```
void ActionBuilder::delay(unsigned long millis);
```

`unsigned long millis`	The number of milliseconds that the action should wait.

Basic Functions

MoveTo

To move an exoskeleton joint or arm to a desired position with a desired velocity, call the functions below. The motion of a joint can either be (1) absolute w.r.t. a pre-determined 0 reference-position or (2) relative to the user's current position.

Applied to an arm:

```
void   ExoskeletonArm::moveTo(ExoArmAngles   pos,   float   maxVelocity,
ICondition* alternativeFinishCondition = nullptr);
```

Applied to a single joint with a motor:
```
void ExoskeletonJoint::moveToActive(float goalAngle, float maxVelocity, float goalAngleTolerance = 5, TargetType targetType = ABSOLUTE, ICondition* alternativeFinishCondition = nullptr)
```

Applied to a passive joint (this action only waits until the user moves the joint to the specified position):
```
void ExoskeletonJoint::moveToPassive(float goalAngle, float goalAngleTolerance = 5, TargetType targetType = ABSOLUTE, ICondition* alternativeFinishCondition = nullptr);
```

`float goalAngle` (joint) or `ExoArmAngles pos` (arms)	The angle(s) that the joint/arm should be moved to/by in degrees.
`float maxVelocity`	The velocity that the joint(s) is moved with in degrees/second. ⚠ **Note:** `0` means maximum velocity.
`float goalAngleTolerance`	The tolerance around the goal angle (in degrees) that defines the area in which the goal position for the joint is being considered as reached.
`TargetType targetType`	Determines whether the exoskeleton the exoskeleton moves by `angle` degrees (relative to the current position) or to the `angle` (absolute position w.r.t. the calibrated 0 degree angle) Options: • `ABSOLUTE` • `RELATIVE_STARTING_POS` ⚠ **Note:** For arm motions, all motions are currently interpreted as absolute values.

Lock & Unlock

Locks a joint and holds it in place.

Applied to a joint:
```
void ExoskeletonJoint::lock(unsigned long lockId, ICondition* alternativeFinishCondition = nullptr);
```

`unsigned long lockId`	An id set by the user that identifies the lock. Use the same id on an action that runs in parallel to disable the lock again.

To unlock a joint, call:
```
void ExoskeletonJoint::unlock(unsigned long lockId);
```

`unsigned long lockId`	An id set by the user that identifies the lock.

Sensing

To stream data from a joint:

```
void ExoskeletonJoint::streamSensorData(bool streamPosition, bool
streamVelocity, bool streamSpeed, bool streamLoadCell, ICondition*
alternativeFinishCondition =nullptr);
```

All selected sensing data is printed in one line separated by a space.

`bool streamPosition`	If true prints the absolute angle in degrees for the joint.
`bool streamVelocity`	If true prints the velocity in degrees/second for the joint.
`bool streamSpeed`	If true prints the speed in degrees/second for the joint.
`bool streamLoadCell`	If true prints the force applied to the joint's associated load cell.

Scripted Motions

Gesture

Performs a pre-defined gesture.

Function Signature:

```
void ExoskeletonArm::gesture(GestureType gesture, ICondition*
alternativeFinishCondition = nullptr);
```

`GestureType gesture`	Options: • `WAVE` (performs a wave gesture) ⚠ **Note:** The waving function only works for a fully actuated exoskeleton arm!

Vibrate

Provides vibrotactile feedback.

These are the signatures for the joint- and arm-based function:

```
void ExoskeletonJoint::vibrate(uint8_t frequency, float amplitude,
ICondition* alternativeFinishCondition = nullptr);
```

```
void ExoskeletonArm::vibrate(uint8_t frequency, float amplitude, ICondition*
alternativeFinishCondition = nullptr);
```

`uint8_t frequency`	Frequency of the vibration in Hz.
`float amplitude`	The amplitude of the vibration in deg.

⚠ **Note:** Don't let the motors run for too long at a very high frequency as this might cause damage.

Motion Transfer

Mirror

Mirrors movements from one arm/joint (**source**) to another arm/joint (**target**) while scaling the movement by a specified factor.

These are the signatures for the joint- and arm-based function:

```
void  ExoskeletonJoint::mirror(ExoskeletonJoint  sourceJoint,  float
scaleFactor, float velocity, ICondition* alternativeFinishCondition =
nullptr);

void  ExoskeletonArm::mirror(ExoskeletonArmHandle  &exoArmSource,  bool
mirrorElbow, bool mirrorShoulderSide, bool mirrorShoulderBack, float
scaleFactorElbow  =  1.0,  float  scaleFactorShoulderSide  =  1.0,  float
scaleFactorShoulderBack = 1.0, ICondition* alternativeFinishCondition =
nullptr);
```

`ExoskeletonJoint sourceJoint` (joint) or `ExoskeletonArmHandle &exoArmSource` (arm)	The joint/arm that should act as the source.
`float scaleFactor` (joint) or `float scaleFactor(Elbow/ShoulderSide/ShoulderBack)` (arm)	The factor by which the source joint movements are scaled. Default = `1.0` for arm
`float velocity` (joint)	The velocity in degrees/second with which the arm follows the source joints position. ⚠ **Note:** `0` means maximum velocity.

`bool mirror(Elbow/ShoulderSide/ShoulderBack)`	Should the joint be included in the mirroring?

⚠ **Note:** Be careful with this function. Ensure there are no loose connections with the position encoder. Always test the functionality first on the table/on a mannequin before wearing it yourself. Ensure you are close to the power switch or emergency button. If there are loose connections in the position encoder, sudden peaks in the measurements can occur. As they are replayed at another joint, these sudden jerks might hurt the user.

Modulating the Motion Effort

Effort modulation comprises interaction strategies in which the exoskeleton applies forces in or opposite to a user's inherent motion, while preserving the user's ability to freely navigate along space. The resulting feeling of assistance or resistance can be leveraged by designers to change a user's internal motion perception, with stronger forces resulting in a stronger effect.

Amplify

Makes it easier for the user to move.

These are the signatures for the joint- and arm-based function:

```
void ExoskeletonJoint::amplify(float amplifyPercentage, ExoskeletonJointHandle::MovementDirection amplifyDirection = ExoskeletonJointHandle::MovementDirection::BOTH, float startingVelocity = 30, float maxVelocity = 0, ICondition* alternativeFinishCondition = nullptr);

void ExoskeletonArm::amplify(float amplifyPercentage, ExoskeletonJointHandle::MovementDirection amplifyDirection = ExoskeletonJointHandle::MovementDirection::BOTH, float startingVelocity = 30, float maxVelocity = 0, ICondition* alternativeFinishCondition = nullptr);
```

`float amplifyPercentage`	Percentage (from 0 to 1) of the motor's maximum available torque which will be used to amplify the user's motion.
`ExoskeletonJointHandle::MovementDirection amplifyDirection`	The exoskeleton will only amplify the motion if the user moves in the indicated direction. Options: • `ExoskeletonJointHandle::MovementDirection.BOTH` • `ExoskeletonJointHandle::MovementDirection.ABDUCTION_OR_FLEXION` • `ExoskeletonJointHandle::MovementDirection.ADDUCTION_EXTENSION`

`float startingVelocity`	The threshold velocity in deg/sec with which the user should move the joint/arm to trigger the amplification. Can be used to fine-tune the sensitivity of the function. Default = `30`
`float maxVelocity`	The maximum velocity in deg/sec to which the function will accelerate the arm. Default = `0` ⚠ **Note:** 0 means unlimited.

Resist

Makes it harder for the user to move.

These are the signatures for the joint- and arm-based function:

```
void        ExoskeletonJoint::resist(float        resistancePercentage,
ExoskeletonJointHandle::MovementDirection      resistDirection       =
ExoskeletonJointHandle::MovementDirection::BOTH,
           float minVelocity = 0, ICondition* alternativeFinishCondition
= nullptr);

void           ExoskeletonArm::resist(float           resistPercentage,
ExoskeletonJointHandle::MovementDirection      resistDirection       =
ExoskeletonJointHandle::MovementDirection::BOTH, float minVelocity = 0,
ICondition* alternativeFinishCondition = nullptr);
```

`float resistancePercentage`	Percentage (from 0 to 1) of the motor's maximum available torque which will be used to resist the user's motion.
`ExoskeletonJointHandle::MovementDirection resistDirection`	The exoskeleton will only resist the motion if the user moves in the indicated direction. Options: • `ExoskeletonJointHandle::MovementDirection.BOTH` • `ExoskeletonJointHandle::MovementDirection.ABDUCTION_OR_FLEXION` • `ExoskeletonJointHandle::MovementDirection.ADDUCTION_EXTENSION`
`float minVelocity`	The minimum velocity in deg/sec to which the function will slow down the arm. Default = `0`

Modulating the Motion Style

Motion style refers to interaction strategies that modify the observable characteristics of a user's movement, such as speed or path, while keeping the user in control of the overall motion. These style modifications can range from subtle adjustments to more pronounced changes.

Filter Speed

Keeps the speed of the user's motion within a desired range. The exoskeleton will amplify the user's motion if the user is slower than desired and resist if they are faster.

These are the signatures for the joint- and arm-based function:

```
void    ExoskeletonJoint::filterVelocity(float    minVelocity,    float
maxVelocity,   ExoskeletonJointHandle::MovementDirection   direction   =
ExoskeletonJointHandle::MovementDirection::BOTH,                   float
amplifyFlexPercentage = 0.05, float resistFlexPercentage = 0.05, float
amplifyExtensionPercentage  =  0.05,  float  resistExtensionPercentage  =
0.05, ICondition* alternativeFinishCondition = nullptr);

void     ExoskeletonArm::filterVelocity(float     minVelocity,    float
maxVelocity,   ExoskeletonJointHandle::MovementDirection   direction   =
ExoskeletonJointHandle::MovementDirection::BOTH,                   float
amplifyFlexPercentage = 0.05, float resistFlexPercentage = 0.05, float
amplifyExtensionPercentage  =  0.05,  float  resistExtensionPercentage  =
0.05, ICondition* alternativeFinishCondition = nullptr);
```

`float minVelocity`	The minimum speed in degrees/second with which the joint should move. If the joint's speed is slower, the exoskeleton will amplify.
`float maxVelocity`	The maximum speed in degrees/second at which the joint should move. If the joint's speed is faster, the exoskeleton will resist.
`ExoskeletonJointHandle::MovementDirection direction`	The exoskeleton will only filter the speed if the user moves in the indicated direction. Options: • `ExoskeletonJointHandle::MovementDirection.BOTH` • `ExoskeletonJointHandle::MovementDirection.ABDUCTION_OR_FLEXION` • `ExoskeletonJointHandle::MovementDirection.ADDUCTION_EXTENSION`
`float amplifyFlexPercentage`	Percentage (from 0 to 1) of the motor's maximum torque which will be used to amplify the motion. Will be applied if the joint moves too slowly and the user is doing a flexion/abduction movement. Default = `0.05`

`float resistFlexPercentage`	Percentage (from 0 to 1) of the motor's maximum torque which will be used to provide resistance. Will be applied if the joint moves too high and the user is doing a flexion/abduction movement. Default = `0.05`
`float amplifyExtensionPercentage`	Percentage (from 0 to 1) of the motor's maximum torque which will be used to amplify the motion. Will be applied if the joint moves too slowly and the user is doing an extension/adduction movement. Default = `0.05`
`float resistExtensionPercentage`	Percentage (from 0 to 1) of the motor's maximum torque which will be used to provide resistance. Will be applied if the joint moves too fast and the user is doing an extension/adduction movement. Default = `0.05`

Jerk

Makes the user's motion jerky: Twitches consisting of forth-and-back motions of a randomized size and within randomized time intervals will be added to the user's motion.

These are the signatures for the joint- and arm-based function:

```
void ExoskeletonJoint::jerk(float minJerkAngle, float maxJerkAngle, long minJerkInterval, long maxJerkInterval, float maxAccumulatedMovementsLeft, float maxAccumulatedMovementsRight, float velocity, long nrJerks = 0, ICondition* alternativeFinishCondition = nullptr);

void ExoskeletonArm::jerk(float minJerkAngle, float maxJerkAngle, long minJerkInterval, long maxJerkInterval, float maxAccumulatedMovementsLeft, float maxAccumulatedMovementsRight, float velocity, long nrJerks = 0, ICondition* alternativeFinishCondition = nullptr);
```

`float minJerkAngle`	The minimum size of a jerk in degrees.
`float maxJerkAngle`	The maximum size of a jerk in degrees.
`long minJerkInterval`	The minimum time in between two jerks in milliseconds. ⚠ **Note:** Timer runs only if joint/arm is moving
`long maxJerkInterval`	The maximum time in between two jerks in milliseconds. ⚠ **Note:** Timer runs only if joint/arm is moving

`float maxAccumulatedMovementsLeft`	How far should jerks be able to move the joint/arm to the left if all jerks are summed up in degrees.
`float maxAccumulatedMovementsRight`	How far should jerks be able to move the joint/arm to the right if all jerks are summed up in degrees.
`float velocity`	The velocity of the jerks in deg/sec. Default = `0.0` ⚠ **Note:** 0 means unlimited.
`long nrJerks`	The number of jerks that will be performed. Default = `0` ⚠ **Note:** 0 means unlimited.

Motion Guidance

Motion guidance comprises strategies in which the exoskeleton applies forces that correct, guide, or constrain a user's movement.

Constrain To

Restricts a user's motion range to a specified area. The area is determined by a range ϵ (here called radius) around a center angle θ.

These are the signatures for the joint- and arm-based function:

```
void   ExoskeletonJoint::constrainTo(float   angle,   float   radius,
ICondition* alternativeFinishCondition = nullptr);

void   ExoskeletonArm::constrainTo(ExoArmAngles   pos,   float   radius,
ICondition* alternativeFinishCondition = nullptr);
```

`float angle` (joint) or `ExoArmAngles pos` (arm)	The center of the area in degree.
`float radius`	The radius around the center angle in degrees.

Guide Away

Guides a user's away from a specified area. The area is determined by a range ϵ (here called radius) around a center angle θ. The user cannot enter the area at all.

These are the signatures for the joint- and arm-based function:

```
void ExoskeletonJoint::guideAway(float angle, float radius, float amplifyPercentage = 0.05, float resistPercentage = 0.05, ICondition* alternativeFinishCondition = nullptr);

void ExoskeletonArm::guideAway(ExoArmAngles pos, float radius, float amplifyPercentage, float resistPercentage, ICondition* alternativeFinishCondition = nullptr);
```

`float angle` (joint) or `ExoArmAngles pos` (arm)	The center of the forbidden area in degree.
`float radius`	The radius around the center angle in degrees.
`float amplifyPercentage`	Percentage (from 0 to 1) of the motor's maximum torque which will be used to amplify the motion. Will be applied if the joint moves away from the forbidden area. Default = `0.05`
`float resistPercentage`	Percentage (from 0 to 1) of the motor's maximum torque which will be used to resist the motion. Will be applied if the joint moves towards the forbidden area. Default = `0.05`

Guide Towards

Guides a user's towards a specified area. The area is determined by a range ϵ (here called radius) around a center angle θ. The user can freely move inside.

These are the signatures for the joint- and arm-based function:

```
void ExoskeletonJoint::guideTowards(float angle, float radius, float amplifyPercentage, float resistPercentage, ICondition* alternativeFinishCondition = nullptr);

void ExoskeletonArm::guideTowards(ExoArmAngles pos, float radius, float amplifyPercentage, float resistPercentage, ICondition* alternativeFinishCondition = nullptr);
```

`float angle` (joint) or	The center of the area in degree.

`ExoArmAngles pos` (arm)	
`float radius`	The radius around the center angle in degrees.
`float amplifyPercentage`	Percentage (from 0 to 1) of the motor's maximum torque which will be used to amplify the motion. Will be applied if the joint moves towards the desired area. Default = `0.05`
`float resistPercentage`	Percentage (from 0 to 1) of the motor's maximum torque which will be used to resist the motion. Will be applied if the joint moves away from the desired area. Default = `0.05`

7.6 Troubleshooting

Here are some common issues and solutions:

Upload of a new program did not work:

- Check if the Dynamixel shield is in upload mode.

Program stopped working:

- The system may have triggered an emergency shutdown. This can occur due to several reasons:
    - The calibrated motion ranges were exceeded, and the system automatically performed an emergency shutdown. Reset the Arduino using the restart button and restart the Processing GUI, too.
    - The motors overheated. This can happen if too much load was applied to them. Check for a blinking red light on the motor which indicative of overheating. Resolve this issue by ensuring the motors have cooled down and, resetting the Arduino using the restart button, then restarting the Processing GUI.
    - Loose connections. Sometimes, this problem occurs, for instance, in case of a loose connection with the serial breakout board.

The Wave functionality does not work:

- The wave function only supports fully actuated arms. Ensure all required actuators are connected and functional.

8 Example Applications

8.1 Motion Guidance for Strength Exercises

A promising area of exoskeletons is rehabilitation and home therapy. We used ExoKit to assist in

the correct execution of an upper arm exercise aimed at strengthening muscles. In this exercise, the user stretches their arm out sideways and moves it up and down repetitively, ensuring the arms don't leave the body plane. At the same time, the elbow must remain fully extended throughout the exercise. The exoskeleton's role is to help the user maintain proper form during execution.

In the following we describe a potential design process:

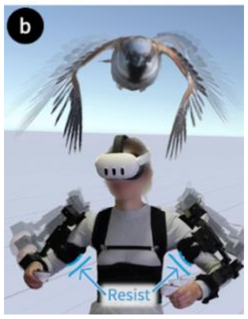
We began by addressing how to keep the user's shoulder movement aligned within the body's side plane. To do this, we built an exoskeleton with an actuated joint at the shoulder (on the side) to modify sideway shoulder motions, along with a passive joint at the back. Initially, we used the `constrainTo` function to lock the user's movement within a narrow area inside the desired body plane. However, we noticed that this reduces the user's agency substantially. To give the user more control, we switched to the `guideTowards` function, which guides the user back toward the desired plane. Since the shoulder can exert significant forces during this motion, we leveraged a stronger Dynamixel motor (XM540-W270-T). This motor provides enough power to apply both assisting and resisting forces at the shoulder and we fine-tuned the torques to be effective yet gentle.

Once satisfied with the shoulder motion, we integrated an actuated joint for the elbow, using the a weakest Dynamixel motor (XM430-W210-T) and added a `lock` function to ensure that the elbow does not drift from the desired position as the user moves.

This is the code:

```
#include "event/simpleactionbuilder/ActionBuilder.h"
#include "exo/ConfiguredEncoder.h"
#include "DynamixelShield.h"
#include "exo/builder/ExoskeletonBuilder.h"
#include "event/simpleactionbuilder/Exoskeleton.h"

DynamixelShield dxl;
ExoskeletonHandle* exoHandle;
Exoskeleton* exoskeleton;
ActionBuilder ab;
const float DXL_PROTOCOL_VERSION = 2.0;


void guideTowardsShoulderSide(ActionBuilder& ab) {
   exoskeleton->getRightArm().getShoulderSide().moveToActive(0,50,3);
   exoskeleton->getRightArm().getShoulderSide().guideTowards(0,             5,
0.03,0.03,nullptr);
}

void lockElbow(ActionBuilder& ab) {
   exoskeleton->getRightArm().getElbow().moveToActive(0,50,3);
   exoskeleton->getRightArm().getElbow().lock(1, nullptr);
}

void setup() {
   Serial.begin(1000000);

   dxl.begin(1000000l);
   dxl.setPortProtocolVersion(DXL_PROTOCOL_VERSION);
   dxl.ping(1);

   DEBUG_SERIAL.begin(115200);

   // TODO enter your Dynamixel IDs
   exoHandle = &ExoskeletonBuilder()
           .addRightArm()
           .addElbowJoint(dxl, 4, nullptr)
                   //.addShoulderSideJoint(dxl, 3, nullptr)
           .addShoulderSideJoint(dxl, 1, nullptr)
           .finishArm()
           .build()
           .calibrate(PREFER_LOAD);

   exoskeleton = new Exoskeleton{ab, *exoHandle};

   ab.parallel(guideTowardsShoulderSide, lockElbow);
   ab.dispatch(ab.trueCondition(), 100);
}

void loop() {
   taskManager.runLoop();
}
```

8.2 Kinesthetic Feedback for Avatar Embodiment in VR



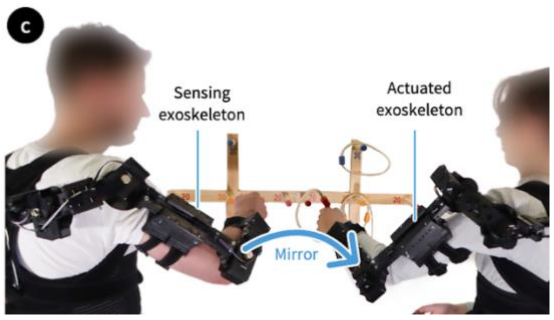
An important research area in VR is providing haptic feedback. We implemented a VR environment that uses ExoKit to create immersive kinesthetic feedback for embodying the motion of an avatar. In our application, a flying game, the user can control the motion of an avatar's wings with the VR setup, by moving one's arms, and feel the corresponding kinesthetic real-time feedback through ExoKit. For instance, when embodying a dragon, characterized by heavy, powerful movements, a designer can adjust the exoskeleton's 6 actuated DoFs with the `resist` function to increase resistance, simulating large, forceful wing beats.

This is the code:

```
#include "event/simpleactionbuilder/ActionBuilder.h"
#include "exo/ConfiguredEncoder.h"
#include "DynamixelShield.h"
#include "exo/builder/ExoskeletonBuilder.h"
#include "event/simpleactionbuilder/Exoskeleton.h"

DynamixelShield dxl;
ExoskeletonHandle* exoHandle;
Exoskeleton* exoskeleton;
ExoskeletonArm* leftArm;
ExoskeletonArm* rightArm;
ActionBuilder ab;
const float DXL_PROTOCOL_VERSION = 2.0;

void rightResist(ActionBuilder& ab) {
   rightArm->resist(0.05);
}

void leftResist(ActionBuilder& ab) {
   leftArm->resist(0.05);
}

void parallelResist(ActionBuilder& ab) {
   ab.parallel(leftResist, rightResist);
}

void setup() {
   Serial.begin(1000000);

   dxl.begin(1000000l);
   dxl.setPortProtocolVersion(DXL_PROTOCOL_VERSION);
   dxl.ping(1);

   DEBUG_SERIAL.begin(115200);

   // TODO correct the dynamixel IDs
    exoHandle = &ExoskeletonBuilder()
   .addLeftArm()
       .addElbowJoint(dxl, 4, nullptr)
```

```
        .addShoulderSideJoint(dxl, 5, nullptr)
        .addShoulderBackJoint(dxl, 2, nullptr)
    .finishArm()
    .addRightArm()
        .addElbowJoint(dxl, 3, nullptr)
        .addShoulderSideJoint(dxl, 6, nullptr)
        .addShoulderBackJoint(dxl, 1, nullptr)
    .finishArm()
    .build()
    .calibrate(PREFER_LOAD);

   exoskeleton = new Exoskeleton{ab, *exoHandle};
   leftArm = &(exoskeleton->getLeftArm());
   rightArm = &(exoskeleton->getRightArm());

   ab.loop(ab.trueCondition(), parallelResist);
   ab.dispatch(ab.trueCondition(), 100);
}

void loop() {
   taskManager.runLoop();
}
```

8.3 Collaborative Body-actuated Play



Body-actuated play leverages body-actuating technologies for novel kinds of creative, shared bodily experiences. Using ExoKit, we developed a collaborative game that demonstrates how designers can craft interactions between multiple players each wearing an exoskeleton. In this game, one player (P1) controls the body movements of another player (P2). The goal is to solve a physical color sorting game by making P2 sort as many rings correctly as time permits. P1 is wearing a 3-DoF exoskeleton arm with sensing capabilities that captures her movements. P2 wears a fully actuated 3-DoF exoskeleton that mirrors P1's motion in real time. This behavior was rapidly implemented with the toolkit's `mirror` function, which supports customizable body remapping strategies in few lines of code. To make the game more demanding for advanced players, we developed a second level that adjusted how the transferred motions are scaled by changing one parameter in the software. Finally, we implemented a third level, that goes beyond a direct 1:1 mapping of body parts and instead maps P1's sideways shoulder motions onto P2's elbow motions, and P1's elbow motions onto P2's shoulder joint to challenge coordination.

```
#include "event/simpleactionbuilder/ActionBuilder.h"
#include "exo/ConfiguredEncoder.h"
#include "DynamixelShield.h"
#include "exo/builder/ExoskeletonBuilder.h"
#include "event/simpleactionbuilder/Exoskeleton.h"

DynamixelShield dxl;
ExoskeletonHandle* exoHandle;
ConfiguredEncoder encoderElbow(A2, 300);
ConfiguredEncoder eEncoderShoulderBack(A0, 300);
ConfiguredEncoder encoderShoulderSide(A1, 300);
ActionBuilder ab;
Exoskeleton* exoskeleton;
const float DXL_PROTOCOL_VERSION = 2.0;

void mirrorElbow2ShoulderSide(ActionBuilder& ab) {
      exoskeleton->getLeftArm().getShoulderSide().mirror(exoHandle-
>getRightArm().getElbow(), 1, nullptr);
}

void mirrorShoulderSide2Elbow(ActionBuilder& ab) {
      exoskeleton->getLeftArm().getElbow().mirror(exoHandle-
>getRightArm().getShoulderSide(), 1, nullptr);
}

void mirrorShoulderBack(ActionBuilder& ab)
{
      exoskeleton->getLeftArm().getShoulderBack().mirror(exoHandle-
>getRightArm().getShoulderBack(), 1, nullptr);
}
void mirrorOtherJoints(ActionBuilder& ab)
{
   ab.parallel(mirrorShoulderSide2Elbow,mirrorShoulderBack);
}

void setup() {
   Serial.begin(1000000);

   dxl.begin(1000000l);
   dxl.setPortProtocolVersion(DXL_PROTOCOL_VERSION);
   dxl.ping(1);

   DEBUG_SERIAL.begin(115200);

   exoHandle = &ExoskeletonBuilder()
           .addLeftArm()
           .addElbowJoint(dxl, 4, nullptr)
           .addShoulderSideJoint(dxl, 1, nullptr)
           .addShoulderBackJoint(dxl, 2, nullptr)
           .finishArm()
           .addRightArm()
           .addElbowJoint(&encoderElbow, nullptr)
           .addShoulderSideJoint(&encoderShoulderSide, nullptr)
           .addShoulderBackJoint(&eEncoderShoulderBack, nullptr)
           .finishArm()
           .build()
           .calibrate(ASK_CREATE);

   exoskeleton = new Exoskeleton{ab, *exoHandle};
   // collaborative body-actuated play - level 1
   //exoskeleton->getLeftArm().mirror(exo->getRightArm(), true, true, true);

   // collaborative body-actuated play - level 2
   //exoskeleton->getLeftArm().mirror(exo->getRightArm(), true, true, true, 2,
2, 2);

   // collaborative body-actuated play - level 3
   ab.parallel(mirrorElbow2ShoulderSide, mirrorOtherJoints);

   ab.dispatch(ab.trueCondition(), 100);
}

void loop() {
   taskManager.runLoop();
}
```

8.4 Artistic Performances

Exoskeletons also offer interesting applications for artistic performances. Using ExoKit, we developed an arm exoskeleton that artistically modifies a dancer's movements in real time. The dancer defines and executes the overall motion of the arms, while the exoskeleton enhances the style of the motion to be more jerky and robot-like. By leveraging the pre-implemented `jerk` function, the designer can introduce variability in the dancer's movements, making them more abrupt. Designers can easily adjust jerk parameters, such as amplitude, frequency, and velocity, to suit the performance, from subtle modifications to exaggerated effects. Moreover, the toolkit allows designers to easily implement specific body gestures that serve as triggers for starting and stopping the motion augmentation. We implemented an ``arm stretched out'' gesture (elbow is at 0 degree angle) for triggering the exoskeleton, ensuring the augmentation aligns with the dancer's intent and preferences.

```
#include "com/SerialCmdReader.h"
#include "event/simpleactionbuilder/ActionBuilder.h"
#include "exo/ConfiguredEncoder.h"
#include "DynamixelShield.h"
#include "exo/builder/ExoskeletonBuilder.h"
#include "com/ManualConditionTriggerCmd.h"
#include "event/simpleactionbuilder/Exoskeleton.h"

SerialCmdReader cmdReader(DEBUG_SERIAL, 10);
ActionBuilder ab;
DynamixelShield dxl;
ExoskeletonHandle* exoHandle;
Exoskeleton* exoskeleton;
ExoskeletonArm* leftArm;
ExoskeletonArm* rightArm;
ManualCondition moveCond;
ManualConditionTriggerCmd triggerMove("do1jerk", moveCond);

const float DXL_PROTOCOL_VERSION = 2.0;

void jerkLeftArm(ActionBuilder& ab)
{
   leftArm->jerk(20,20,1500,1500,100,100,0);
}

void jerkRightArm(ActionBuilder& ab) {
   rightArm->jerk(20,20,1500,1500,100,100,0);
}

void parallelJerk(ActionBuilder& ab)
{
   ab.parallel( jerkLeftArm, jerkRightArm);
}



ICondition& startPose()
{
   return (leftArm->getElbow().isAtPosition(EQUAL, 0, 20)
           && rightArm->getElbow().isAtPosition(EQUAL, 0, 20));
}

void setup() {
   Serial.begin(1000000);

   dxl.begin(1000000l);
   dxl.setPortProtocolVersion(DXL_PROTOCOL_VERSION);
   dxl.ping(1);

   DEBUG_SERIAL.begin(115200);
   cmdReader.registerCmd(triggerMove);
   moveCond.configure(true, false);

   // TODO correct the dynamixel IDs
   exoHandle = &ExoskeletonBuilder()
           .addLeftArm()
           .addElbowJoint(dxl, 4, nullptr)
                   //.addShoulderSideJoint(dxl, 3, nullptr)
           .addShoulderBackJoint(dxl, 2, nullptr)
           .finishArm()
           .addRightArm()
           .addElbowJoint(dxl, 3, nullptr)
                   //.addShoulderSideJoint(dxl, 3, nullptr)
           .addShoulderBackJoint(dxl, 1, nullptr)
           .finishArm()
           .build()
           .calibrate(PREFER_LOAD);

   exoskeleton = new Exoskeleton{ab, *exoHandle};
   leftArm = &(exoskeleton->getLeftArm());
   rightArm = &(exoskeleton->getRightArm());

   parallelJerk(ab);
   ab.dispatch(startPose(), 10);

}

void loop() {
   taskManager.runLoop();
}
```

Appendix A: Glossary

Degrees of Freedom (DoF)*:* The number of independent movements a joint can perform, such as flexion/extension or abduction/adduction.

Range of Motion (RoM): The maximum arc through which the joint can move. Measured in degrees.

Torque: Torque refers to the rotational force applied to an object, such as a joint in an exoskeleton. Measured in newton-meters (Nm).

Velocity: Velocity is the speed of an object in a specific direction. In the context of exoskeletons, it often refers to how quickly a joint moves or rotates. Measured in units like meters per second (m/s) or degrees per second (°/s).

Load cell: A sensor that measures weight and force.

Links: Exoskeleton links are structural components that connect and transmit forces between joints.

Joints: Exoskeleton joints are structural components and can be either active or passive. Active joints are actuated, e.g., through a motor. Hence, they can be used to actively control user motion.

Appendix B: Debug Script for Working With Position Encoders

```
#include <com/SerialCmdReader.h>

#include "event/simpleactionbuilder/ActionBuilder.h"
#include "exo/ConfiguredEncoder.h"
#include "DynamixelShield.h"
#include "exo/builder/ExoskeletonBuilder.h"
#include "event/simpleactionbuilder/Exoskeleton.h"

ConfiguredEncoder  encoder(A0,  300);  //  create  a  position  encoder  object

DynamixelShield dxl;
ExoskeletonHandle* exoHandle;
Exoskeleton* exoskeleton;
ActionBuilder ab;
SerialCmdReader terminalInteraction(DEBUG_SERIAL);
const float DXL_PROTOCOL_VERSION = 2.0;


void setup() {
   Serial.begin(1000000);
   dxl.begin(1000000l);
   dxl.setPortProtocolVersion(DXL_PROTOCOL_VERSION);
   dxl.ping(1);
   DEBUG_SERIAL.begin(115200);

   // TODO change the indicated Dynamixel ID (currently 4) to your Dynamixel's
ID
   // TODO remove button if you don't want one
   exoHandle = &ExoskeletonBuilder()
       .addLeftArm()
           .addElbowJoint(&encoder) // configure a passive elbow joint with a
position encoder
           .addShoulderSideJoint(dxl, 1, nullptr)
           .finishArm()
       .addEmergencyShutdownButton(50,true)
       .build()
       .calibrate(PREFER_LOAD);

   exoskeleton = new Exoskeleton{ab, *exoHandle};
   exoskeleton->getLeftArm().getShoulderSide().mirror(exoskeleton-
>getLeftArm().getElbow(),1,0);

   ab.dispatch(ab.trueCondition(), 100);

}

unsigned long lastPrint{0};
void loop() {
   taskManager.runLoop();
   unsigned long val = millis() % 30000;
   if(millis() - lastPrint >= 500)
   {
       lastPrint = millis();
       DEBUG_SERIAL.print(F("EP/EV -- B/S: "));
       DEBUG_SERIAL.print(exoskeleton-
>getLeftArm().getElbow().unwrap().getPresentPosition());
       DEBUG_SERIAL.print(F(" / "));
       DEBUG_SERIAL.print(exoskeleton-
>getLeftArm().getElbow().unwrap().getVelocity());
       DEBUG_SERIAL.print(F(" -- "));
       DEBUG_SERIAL.print(exoskeleton-
>getLeftArm().getShoulderBack().unwrap().getPresentPosition());
       DEBUG_SERIAL.print(F(" / "));
       DEBUG_SERIAL.print(exoskeleton-
>getLeftArm().getShoulderSide().unwrap().getPresentPosition());
       DEBUG_SERIAL.print(F(" / "));
       DEBUG_SERIAL.print((val % 10000) / 1000);
       DEBUG_SERIAL.println();
   }
}
```

Appendix C: Example Script for Streaming Sensor Data & Using Load Cell

```
#include <com/SerialCmdReader.h>
#include "event/simpleactionbuilder/ActionBuilder.h"
#include "exo/ConfiguredEncoder.h"
#include "DynamixelShield.h"
#include "exo/builder/ExoskeletonBuilder.h"
#include "event/simpleactionbuilder/Exoskeleton.h"

DynamixelShield dxl;
ExoskeletonHandle* exoHandle;
Exoskeleton* exoskeleton;
ActionBuilder ab;
HX711 loadCellElbow;
const float DXL_PROTOCOL_VERSION = 2.0;

void loopblock(ActionBuilder& ab) {
    exoskeleton->getLeftArm().getShoulderSide().streamSensorData(true, true,
true, true,
        &(exoskeleton-
>getLeftArm().getShoulderSide().isForce(ForceCondition::WITHIN, 0, 100)));
}

void setup() {
    Serial.begin(1000000);
    dxl.begin(10000001);
    dxl.setPortProtocolVersion(DXL_PROTOCOL_VERSION);
    dxl.ping(1);
    DEBUG_SERIAL.begin(115200);

    // TODO change the indicated Dynamixel ID (currently 1) to your
Dynamixel's ID
    exoHandle = &ExoskeletonBuilder()
            .addLeftArm()
            .addShoulderSideJoint(dxl, 1, &loadCellElbow)
            .finishArm()
            .addEmergencyShutdownButton(50,true)
            .build()
            .calibrate(PREFER_LOAD);
    exoskeleton = new Exoskeleton{ab, *exoHandle};
    loadCellElbow.begin(A1, A0);
    loadCellElbow.tare();

    ICondition& loopcondition = !exoskeleton-
>getLeftArm().getShoulderSide().isForce(ForceCondition::WITHIN, 0, 100);
    ab.loop(loopcondition,loopblock);
    ab.dispatch(ab.trueCondition(), 80);
}

void loop() {
    taskManager.runLoop();
}
```

Appendix D: Motor Cheat Sheet

XM430-W210
- Torque, i.e., ‘maximum strength’ of the motor: 3.0 Nm
- Weight: 82 g
- Maximum speed: 77 [rev/min]
- for details regarding performance graphs etc:https://emanual.robotis.com/docs/en/dxl/x/xm430-w210/

XM430-W350
- Torque, i.e., ‘maximum strength’ of the motor: 4.1 Nm
- Weight: 82 g
- Maximum speed: 46 [rev/min]
- for details regarding performance graphs etc: https://emanual.robotis.com/docs/en/dxl/x/xm430-w350/

XM540-W150
- Torque, i.e., ‘maximum strength’ of the motor: 7.3 Nm
- Weight: 165 g
- Maximum speed: 53 [rev/min]
- for details regarding performance graphs etc: https://emanual.robotis.com/docs/en/dxl/x/xm540-w150/

XM540-W270
- Torque, i.e., ‘maximum strength’ of the motor: 10.6 Nm
- Weight: 165 g
- Maximum speed: 30 [rev/min]
- for details regarding performance graphs etc: https://emanual.robotis.com/docs/en/dxl/x/xm540-w270/

Appendix E: Terminal Function Calls

- Set up the right program in the Arduino programming interface by enabling serial commands.

- Each function call has mandatory and/or optional parameters. Those parameters indicated by *(.)* are mandatory, parameters indicated by *[.]* optional. The default value of optional parameters is indicated in the function description.
- You can only call one function at a time.Try to call at least each joint-function once and play around with the parameters to familiarize yourself with the effects.

Basic Functions

To lock an exoskeleton joint or arm in place, call:

- **jointlock** (ExoID) (armSide) (jointType)
- **armlock** (ExoID) (armSide)

cancel must be called to end this actuation.

To move an exoskeleton joint or arm to a desired position with a desired velocity, call the functions below. The motion of a joint can either be (1) absolute w.r.t. a pre-determined 0 reference-position or (2) relative to the user's current position.

- **jointmoveto** (ExoID) (armSide) (jointType) (angle) (velocity) (*absolute*|*relative*)
- **armmoveto** (ExoID) (armSide) (elbowAngle|*false*) (shoulderSideAngle|*false*) (shoulderBackAngle|*false*) (velocity)

Parameters (lock):

- ExoID: the ID assigned to the exoskeleton (done in firmware). For the exploration, use the pre-configured ID 1.
- armSide: determines whether the left or right arm should be actuated; {*left*, *right}.* In the current setup, only the right arm can be actuated.
- jointType: determines which joint should be actuated; {*elbow, shoulderback, shoulderside*}

Parameters (moveto):

- ExoID: the ID assigned to the exoskeleton (done in firmware). For the exploration, use the pre-configured ID 1.
- armSide: determines whether the left or right arm should be actuated; {*left*, *right}.* In the current setup, only the right arm can be actuated.
- jointType: determines which joint should be actuated; {*elbow, shoulderback, shoulderside*}
- angle, elbowAngle, shoulderSideAngle, shoulderBackAngle: in degree
- velocity: in degree / second

Warnings moveTo:

- velocity=0 equals maximum velocity. Don't do this. A good start is velocity=20.

Example calls:

```
> jointlock 1 right elbow
> jointmoveto 1 right elbow 90 20 absolute
```

Scripted Motions

Scripted motions are predefined sequences of exoskeleton movements.

To perform a pre-defined movement sequence, call the functions below. At the moment, we only provide a WAVE gesture as an example. This actuates the full arm

- **armgesture** (ExoID) (armSide) *[gesture=WAVE]*

To provide a vibration feedback, call:

- **jointvibrate** (ExoID) (armSide) (jointType) (frequency) (amplitude)
- **armvibrate** (ExoID) (armSide) (frequency) (amplitude)

cancel must be called to end this actuation.

Parameters (gesture):

- ExoID: For the exploration, use the pre-configured ID 1.
- armSide: {*left*, *right*}
- gesture*:* the gesture to perform; *{WAVE}. We* intend to offer more pre-defined gestures in the future

Parameters (vibrate):

- ExoID: For the exploration, use the pre-configured ID 1.
- armSide: {*left*, *right*}
- jointType: {*elbow, shoulderback, shoulderside*}
- frequency: the vibration frequency in Hz. We recommend values <= 65
- amplitude: the amplitude of the vibration in degree. We recommend values in [1,3]

Warnings vibrate:

- amplitude is given in degrees. Don't take a high amplitude, since this will quickly and heavily shake your arm. Recommended to start with a value like 2.

Example calls:

```
> armgesture 1 right
> jointvibrate 1 right elbow 10 2
```

Motion Transfer

Motion transfer involves strategies where the exoskeleton follows a specific trajectory, either to reach a target position or to demonstrate a motion to the user.

To transfer motions from sensing joints or arms onto actuated joints or arms in real-time, call:

- **jointmirror** (targetExoID) (targetArmSide) (targetJointType) (sourceExoID) (sourceArmSide) (sourceJointType) *[scaleFactor=1.0]*

- **armmirror** (targetExoID) (targetArmSide) (sourceExoID) (sourceArmSide) *[mirrorElbow=true|(true|false)]* *[mirrorShoulderSide=true|(true|false)]* *[mirrorShoulderBack=true|(true|false)]* *[scaleFactorElbow=1.0]* *[scaleFactorShoulderSide=1.0]* *[scaleFactorShoulderBack=1.0]*

cancel must be called to end this actuation.

Parameter units:

- ExoID:For the exploration, use the pre-configured ID 1.
- armSide: {*left*, *right}.* In the current setup, the right arm can be actuated, the left one contains sensors (shoulderside and elbow)
- jointType: {*elbow, shoulderback, shoulderside*}
- mirrorElbow, mirrorShoulderSide, mirrorShoulderBack: indicate whether the motion should be transferred from the sensing elbow/shoulderside/shoulderback joint onto the actuated counterpart
- scaleFactorElbow, scaleFactorShoulderSide, scaleFactorShoulderBack: indicates the factor by which the sensed motions are scaled up/down before transferring them onto the actuated joint. A scale factor of 1 means cloning the received signal, a factor < 0 to shrink the transferred motion, a factor >0 to enlarge the motion.

Warnings:

- **Make sure to move both arms to equal positions before starting this action.** The exoskeleton will move to the source position with maximum velocity. Also at the start!
- while armmirror works in theory with our prototype, our shoulder sensor currently is broken. **Please only use joint mirror with the elbow!**

Example calls:

```
> jointmirror 1 right elbow 1 left elbow 1
```

Effort Modulation

Effort modulation comprises interaction strategies in which the exoskeleton applies forces in or opposite to a user's inherent motion,
while preserving the user's ability to freely navigate along space. The resulting feeling of assistance or resistance can be leveraged
by designers to change a user's internal motion perception, with stronger forces resulting in a stronger effect.

To make it easier for the user to move, call:

- **jointamplify** (ExoID) (armSide) (jointType) (torquePercentage) [direction=*BOTH|(abduction|flexion|adduction|extension|both)]* [startingVelocity=30] [maxVelocity=0]
- **armamplify** (ExoID) (armSide) (torquePercentage) [direction=*BOTH|(abduction|flexion|adduction|extension|both)]* [startingVelocity=30] [maxVelocity=0]

cancel must be called to end this actuation.

To make it harder to move, call:

- **jointresist** (ExoID) (armSide) (jointType) (torquePercentage) [direction=*BOTH*|(*abduction*|*flexion*|*adduction*|*extension*|*both*)] [minVelocity=0]
- **armresist** (ExoID) (armSide) (torquePercentage) [direction=*BOTH*|*(abduction*|*flexion*|*adduction*|*extension*|*both)*] [minVelocity=0]

cancel must be called to end this actuation.

Parameters (amplify):

- ExoID: For the exploration, use the pre-configured ID 1.
- armSide: {*left*, *right}*
- jointType: {*elbow, shoulderback, shoulderside*}
- torquePercentage: A float value between 0 and 1 where 0 is 0% and 1 is 100%. Applies this torque as a percentage of the maximum torque that the servo is allowed to apply. (Which is the maximum. **Don't set it to 1; we recommend values around 0.03.**)
- direction: the movement direction in which the user's motion shall be augmented; *{both, abduction, flexion, adduction, extension}*
- startingVelocity: in degree/seconds. Exo starts amplifying if this value is surpassed.
- maxVelocity: in degree/seconds. Exo stops amplifying if velocity surpasses this value.

Parameters (resist):

- ExoID: For the exploration, use the pre-configured ID 1.
- armSide: {*left*, *right}*
- jointType: {*elbow, shoulderback, shoulderside*}
- torquePercentage: A float value between 0 and 1 where 0 is 0% and 1 is 100%. Applies this torque as a percentage of the maximum torque that the servo is allowed to apply. (Which is the maximum. **Don't set it to 1; we recommend values around 0.03.**)
- direction: the movement direction in which the user's motion shall be augmented; *{both, abduction, flexion, adduction, extension}*
- minVelocity: in degree/seconds. Exo stops resisting if velocity falls below this value.

Warnings:

- velocity=0 equals maximum velocity.
- **torquePercentage should be set to a very low value.** 1 will apply a torque of 100%. Start with 0.03.

Example calls:

```
> jointamplify 1 right elbow 0.03 flexion
> jointresist 1 right elbow 0.03 flexion
```

Motion Style

Motion style refers to interaction strategies that modify the observable characteristics of a user's movement, such as speed or path, while keeping the user in control of the overall motion. These style modifications can range from subtle adjustments to more pronounced changes.

To keep the velocity of the user's motion within a desired range, call the function below. The exoskeleton will amplify the user's motion if the user is slower than desired and resist if they are faster.

- **jointfilterspeed** (ExoID) (armSide) (jointType) (minSpeed) (maxSpeed) [direction=BOTH|(*abduction*|*flexion*|*adduction*|*extension*|*both*)] [amplifyFlexTorquePercentage=0.05] [resistFlexTorquePercentage=0.05] [amplifyExtensionTorquePercentage=0.05] [resistExtensionTorquePercentage=0.05]
- **armfilterspeed** (ExoID) (armSide) (minSpeed) (maxSpeed) [direction=*BOTH|(abduction|flexion|adduction|extension|both)*] [amplifyFlexTorquePercentage=0.05] [resistFlexTorquePercentage=0.05] [amplifyExtensionTorquePercentage=0.05] [resistExtensionTorquePercentage=0.05]

cancel must be called to end this actuation.

To make the user's motion jerky (i.e., make the user's limbs twitch in specified time intervals), call:

- **jointjerk** (ExoID) (armSide) (jointType) (minJerkAngle) (maxJerkAngle) (minJerkIntervalMs) (maxJerkIntervalMs) (maxAccumulatedMovementsLeft) (maxAccumulatedMovementsRight) [velocity=0] *[nrJerks=0]*
- **armjerk** (ExoID) (armSide) (minJerkAngle) (maxJerkAngle) (minJerkIntervalMs) (maxJerkIntervalMs) (maxAccumulatedMovementsLeft) (maxAccumulatedMovementsRight) [velocity=0] [nrJerks=0]

cancel must be called to end this actuation.

Parameters (filterspeed):

- ExoID: For the exploration, use the pre-configured ID 1.
- armSide: {*left*, *right}*
- jointType: {*elbow, shoulderback, shoulderside*}
- minSpeed: in degree/second. Triggers the augmentation (amplification) if the user is slower than this. Here 0 actually means 0.
- maxSpeed: in degree/second. Triggers the augmentation (resistance) if the user is faster than this
- direction: the movement direction in which the user's motion shall be augmented; *{both, abduction, flexion, adduction, extension}*
- amplifyFlexTorquePercentage, resistFlexTorquePercentage, amplifyExtensionTorquePercentage, resistExtensionTorquePercentage: [0,1]: The percentage with which the exoskeleton resists or amplifies

Parameters (jerk):

- ExoID: For the exploration, use the pre-configured ID 1.
- armSide: {*left*, *right}*
- jointType: {*elbow, shoulderback, shoulderside*}
- minJerkAngle: in degree/second. The minimum amplitude of a jerk.
- maxJerkAngle: in degree/second. The maximum amplitude of a jerk.
- minJerkIntervalMs: in milliseconds. The minimum time between two jerks
- maxJerkIntervalMs: in milliseconds. The maximum time between two jerks
- maxAccumulatedMovementsLeft: the maximum number of times that a jerk towards the left might take place in a row
- maxAccumulatedMovementsRight: the maximum number of times that a jerk towards the right might take place in a row
- velocity: in degree/second. The velocity that the jerk should be performed with.
- nrJerks: Number of jerks to be performed by the action

Warnings:

- velocity=0 equals maximum velocity. Don't do this.

Example calls:

```
> jointfilterspeed 1 right elbow 30 50 flexion
> jointjerk 1 right elbow 10 20 1000 2000 0 0 0 5
```

Motion Guidance

Haptic guidance comprises strategies in which the exoskeleton applies forces that correct, guide, or constrain a user's movement. As the forces increase and the unconstrained range of motion decreases, the user's motion becomes more controlled, up to being fully moved or locked. Conversely, weaker forces and larger unconstrained areas allow the user to move more freely.

To restrict a user's motion range to an area with a radius r centered around a specified angle, call:

- **jointconstrainto** (ExoID) (armSide) (jointType) (angle) (radius)
- **armconstrainto** (ExoID) (armSide) (elbowAngle|*false*) (shoulderSideAngle|*false*) (shoulderBackAngle|*false*) (radius)

cancel must be called to end this actuation.

To guide a user towards an area with a radius r centered around a specified angle, call the function below. The exoskeleton will amplify the user's motion if they are moving towards the area and resist it otherwise.

- **jointguidetowards** (ExoID) (armSide) (jointType) (angle) (radius) [amplifyTorquePercentage=0.05] [resistTorquePercentage=0.05]
- **armguidetowards** (ExoID) (armSide) (elbowAngle|*false*) (shoulderSideAngle|*false*) (shoulderBackAngle|*false*) (radius) [amplifyTorquePercentage=0.05] [resistTorquePercentage=0.05]

cancel must be called to end this actuation.

To guide a user away from an area with a radius r centered around a specified angle, call the function below. The exoskeleton will amplify the user's motion if they are moving away from the area and resist it otherwise.

- **jointguideaway** (ExoID) (armSide) (jointType) (angle) (radius) [amplifyTorquePercentage=0.05] [resistTorquePercentage=0.05]
- **armguideaway** (ExoID) (armSide) (elbowAngle|*false*) (shoulderSideAngle|*false*) (shoulderBackAngle|*false*) (radius) [amplifyTorquePercentage=0.05] [resistTorquePercentage=0.05]

cancel must be called to end this actuation.

Parameters (constrainto):

- ExoID: For the exploration, use the pre-configured ID 1.
- armSide: {*left*, *right*}
- jointType: {*elbow, shoulderback, shoulderside*}
- angle, radius, elbowAngle, shoulderSideAngle, shoulderBackAngle: in degree/second

Parameters (guidetowards & guideaway):

- ExoID: For the exploration, use the pre-configured ID 1.
- armSide: {*left*, *right}*
- jointType: {*elbow, shoulderback, shoulderside*}
- angle, radius, elbowAngle, shoulderSideAngle, shoulderBackAngle: in degree/second
- amplifyTorquePercentage, resistTorquePercentage: A float value between 0 and 1 where 0 is 0% and 1 is 100%. Applies this torque as a percentage of the maximum torque that the servo is allowed to apply. (Which is the maximum. **Don't set it to 1; we recommend values around 0.03.**)

Example calls:

```
> jointconstrainto 1 right elbow 45 15
> jointguidetowards 1 right elbow 45 5
> jointguideaway 1 right elbow 45 5
```