EduExo - The Robotic Exoskeleton Kit

Handbook and Tutorial

First Edition

2017

English

Beyond Robotics GmbH

www.beyond-robotics.com

EduExo - The Robotic Exoskeleton Kit. Handbook and Tutorial.

For all inquiries please contact:
Beyond Robotics GmbH
8706 Meilen, Switzerland
www.beyond-robotics.com
info@beyond-robotics.com

First Edition, 2017

Preface

Welcome to the handbook of the EduExo, the robotic exoskeleton kit that introduces you, hands-on, into the fascinating world of robotic exoskeletons. This handbook and tutorial are used in combination with the EduExo exoskeleton, an elbow exoskeleton that you assemble and program yourself. The EduExo kit will teach you different aspects of robotic exoskeleton technology. Topics discussed are human anatomy and exoskeleton mechanics, electronics and software, controls systems, and virtual reality and computer games. Each chapter of this handbook will discuss one of the topics and starts by introducing you to the theoretical background. The second part of the chapters consists of a tutorial in which you can immediately apply your new knowledge to improve the EduExo exoskeleton.

The EduExo is designed for students (high school or university level), makers and hobbyists who are interested in robotics, electronics, haptics, force-feedback systems, control systems, computer games or virtual reality interfaces. The EduExo is designed for people without a background in robotics. Basic programming and electronics knowledge will facilitate the use of this kit. Depending on you prior knowledge, some of the explanations might not teach you something new and some might not be detailed enough. But if you don't shy away from learning by doing and are willing to use the internet for some further readings if necessary, there should be nothing in this kit that you cannot understand and master.

This kit is developed for an international audience, therefore, we mostly use the International System of Units (SI) and follow the International Electrotechnical Commission (IEC) standards. In case your country uses a different system, you may find some units and symbols that are different to the ones you are used to. But this does not affect the science and technology behind the EduExo, and should not really complicate its use.

When you finish this kit, you will know about exoskeleton applications, how they are designed, how to connect the sensors and read the sensor data, how to control the motor, how to program a microcontroller and how to design a control system that defines the interaction between exoskeleton and user. Additionally, you will have learned how to create a computer game and how to use the exoskeleton as an input device to interact with the virtual environment of the game. But this is hopefully just the beginning and, after this introduction into the world of exoskeletons, you will find it as interesting as we do. Based on your new knowledge, you can start expanding the exoskeleton and experiment further,

add sensors, program new games or learn about mechanical design and adapt or extend the exoskeleton. Please note that complementary to this handbook, we also created the website www.eduexo.com to provide additional information, instructions and multimedia content related to the EduExo robotics kit. We would also invite you to share the results of your EduExo projects through this page with other EduExo users.

The EduExo development would not have been possible without the support of many people. Therefore, we would like to express our deepest gratitude to everyone who supported this project along the journey. A special thanks goes to all our Kickstarter backers. Your support and trust made it possible to convert a fun project into the educational kit in front of you that will enable many more people to get involved into this fascinating and highly promising field.

We hope you will enjoy the EduExo and that it will help you learn about robotic exoskeleton technology to get you started into the world of wearable assistive robots!

Zürich, July 2017 *The EduExo Team*

Contents

Chapter 1

Introduction

Summary

This chapter will introduce you to the field of robotic exoskeletons. A brief look at exoskeleton history, examples of their applications and their functionality will help you understand the technology that exists today. This is the only chapter without a hands-on tutorial section, but the last sections of this chapter will introduce you to the EduExo hardware and explain the necessary preparations to get started.

1.1 Robots and Robotic Exoskeletons

Robots have a huge impact in our everyday lives. They have been used in industry for mass production for decades. Today, they can be found in applications ranging from lawn mowing to warehouse logistics to autonomous cars. The technology is still developing and today includes driving robots, underwater robots, flying drones and even quadrupedal and bipedal walking robots that can operate in many different scenarios.

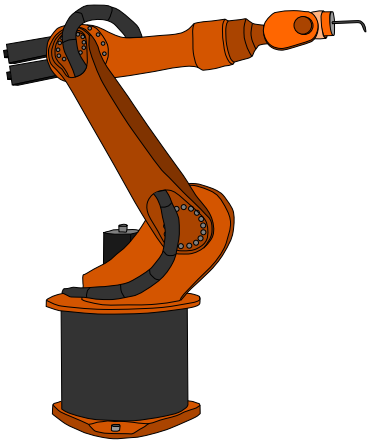

Figure 1.1: A typical industrial robot.

Typical industrial robots are stationary articulated robot arms (Figure 1.1) that are very good at executing pre-programmed, repetitive movements very fast and accurately for 24 hours, 7 days a week. They are used, for example, in assembly lines, where they execute tasks like welding or spray painting parts of new cars. As these systems can be very strong and fast, they are often too dangerous to work together with humans, and are therefore separated from human workers.

Figure 1.2: A vacuum cleaner robot.

On the other end, service robots are made to interact with humans; a typical example are vacuum cleaner robots (Figure 1.2). They are mobile robots with an integrated vacuum cleaner that move around through apartments to clean them. They are usually shaped as small and flat disks, which enables them to navigate even in very messy apartments and clean underneath furniture. They relieve us from the tedious task of vacuuming our apartments by doing the task for us, preferably when we are not at home.

Robotic exoskeletons, that is, wearable robots that resemble the shape of the human body and therefore appear to be an outer (greek: exo) shell or skeleton, are comparatively new. If this is the first time you are introduced to robotic exoskeletons, your ideas about them might be highly influenced by science fiction movies such as Iron Man, Aliens or Elysium. While the portrait of the technology in those movies may be a little over-the-top, resulting in unrealistic expectations, the main idea is not wrong. The exoskeleton enables the heroes to solve a difficult task that they would otherwise not be able to solve (for example, defeating the alien queen), and this main motivation translates to today's real world robotic exoskeletons. In contrast to their robot colleagues that assemble cars or clean our apartments, exoskeletons are not developed to relieve us from a task by solving it for us, but rather to support us humans in executing a physical task that we cannot do independently.

The need for physical support can occur in many situations. Maybe a task is too difficult for the average person, as when parts in a shipyard are too heavy to carry. Or someone is unable to execute an activity because of a physical impairment. In either case, the robotic exoskeleton provides the required physical assistance. Typical applications are industry jobs that require lifting and carrying heavy objects, rehabilitation of stroke survivors or gait restoration in people that suffered spinal cord injuries. Although these systems are not yet as widely spread as other types of robots, the field of robotic exoskeletons is growing quickly, and more and more companies and research institutions are developing new systems (for more on exoskeleton history, see next section).

Robotic exoskeletons have shown especial potential in the rehabilitation of movement impairments. Different medical conditions can lead to movement impairments. One example are spinal cord injuries that can cause paraplegia, which is the paralysis (inability to move) of the lower extremities. Another example are stroke survivors that suffer from hemiplegia, a form of paralysis on one side of the body due to brain damage. In all cases of paralysis, the use of the paralyzed limb is limited or even impossible. This is were robotic exoskeletons can be useful, as they can be attached to the paralyzed limb and provide the support necessary to move it (Figure 1.3).

In case of a stroke or an incomplete spinal cord injury (spinal cord partially severed), patients often lose some, but not all, ability to move on their own. In some cases, the residual movement can improve over time if the patient follows an intensive rehabilitation and training program.

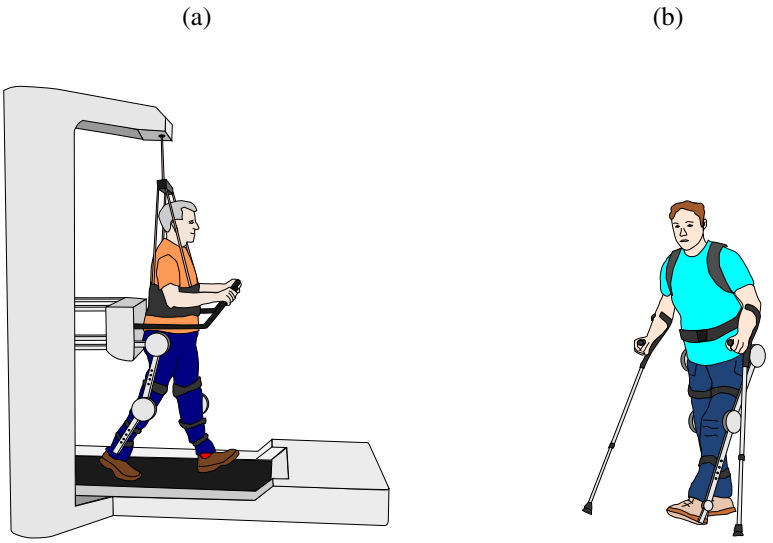

Figure 1.3: Examples of exoskeleton applications in medicine: (a) a rehabilitation exoskeleton used for gait training of stroke and spinal cord injury patients; and (b) a lower extremity exoskeleton for functional gait restoration of paraplegic users.

If the impairments are too severe and the patients cannot use their arms or legs without support, they require assistance to follow their rehabilitation program. One possibility to provide patients with the necessary assistance are robotic exoskeletons for rehabilitation (Figure 1.3(a)). Such systems are usually placed in hospitals and are used to train multiple patients per day. In the example (Figure 1.3(a)), patients follow a treadmill-based gait training as part of a gait rehabilitation program. Patients are connected to the device and their leg movements are supported by the exoskeleton. The support can usually be adjusted to the impairment level of each individual patient. Rehabilitation exoskeletons have several potential advantages. They relieve the therapy staff of the exhausting task of manually supporting the patient by holding the patient's arms or legs and moving them, enabling longer and more intensive therapy sessions. The exoskeleton's integrated sensor systems can be used to measure the amount of support needed by the patient, and can automatically record the recovery process over time. The exoskeleton can be easily used in combination with computer games to keep the patient motivated during week- and month-long therapy programs (more on this subject in chapter 5).

If healing is not possible, e.g., a complete spinal cord injury resulting in permanent paraplegia, smaller exoskeletons can be used to enable walking again by substituting the lost function of the legs (Figure 1.3(b)). They can also help to prevent or reduce negative effects of prolonged seating, such as osteoporosis (weakening of the bone) or decubitus (damage of the skin and soft tissues due to prolonged exposure to pressure). These systems provide the strength and stabilization to move the legs of paralyzed users. Some devices even allow stair climbing and other 'advanced' movements, and enable the user to regain locomotion capabilities that exceed the ones provided by a wheelchair, which is the common way to restore locomotion in paralyzed patients.

Outside of the medical field, robotic exoskeletons are used to augment people's physical performance. Robotic exoskeletons can increase the strength of the user, increasing endurance and enabling lifting of heavy objects (Figure 1.4(a)). This makes them especially useful for physically demanding labor and can prevent injuries typically related to heavy labor (injuries of the back, knees, etc.). Passive exoskeletons (without motors) are used to provide relief from the user's own body weight (Figure 1.4(b)) or from the weight of an external payload (Figure 1.4(c)).

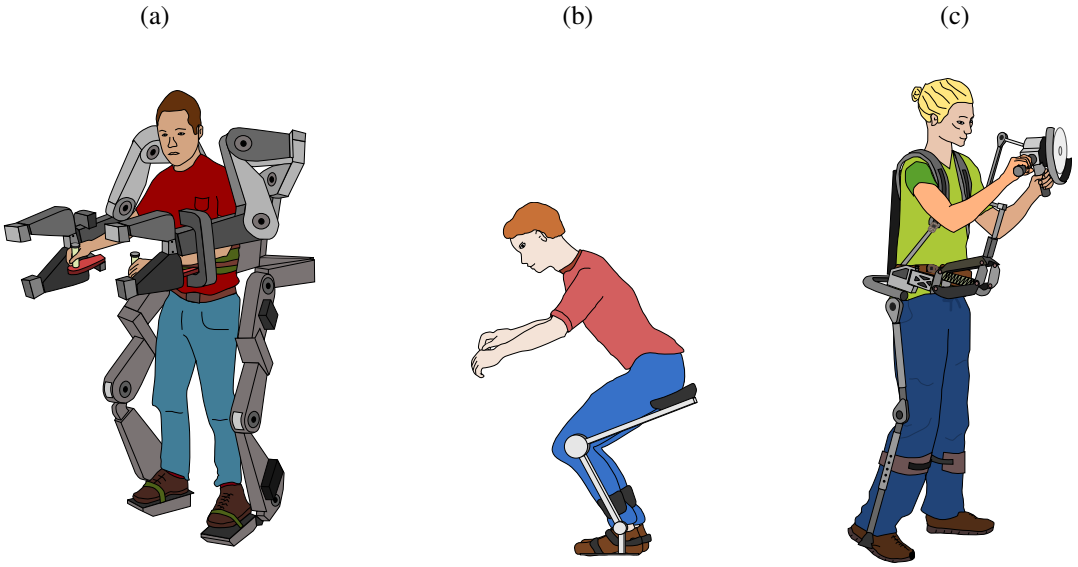

Figure 1.4: Examples of non-medical exoskeleton applications: (a) for strength augmentation; (b) for mobile body weight support; (c) for power tool support;

The examples above already illustrate that exoskeleton designs are variable and depend a lot on the intended task and the required amount of support. First, we can distinguish between stationary systems and mobile systems. Stationary systems (e.g., the gait trainer in Fig. 1.3(a)) are often permanently installed in a room, and the users come to the system to use it. Mobile systems, on the other hand, enable the user to move around. While moving around is required for many applications, it also requires that all components such as power supply, actuation and computers are integrated into the system.

Most exoskeletons are also not 'full-body-systems', that is, they do not cover the entire body and support all limbs. It is much more common that they only cover the parts of the body where the support is needed, for example, the legs of a paraplegic user or the affected arm after a stroke. Also, single articulated devices that only support one joint can be very useful, e.g., hip support for heavy lifting, or ankle joint support for running.

1.2 A Brief History of Robotic Exoskeletons

The development of robotic exoskeletons began already in the second half of the 20th century. Around 1965, General Electric (USA) began developing the Hardiman, a large

full-body exoskeleton designed to augment the user's strength to enable the lifting of heavy objects. The first exoskeletons for gait assistance were developed at the end of the 1960s at the Mihajlo Pupin Institute (Serbia), and in the early 1970s at the University of Wisconsin-Madison (USA). Because of the technical limitations of their time and the lack of experience and knowledge, it still took several decades until the technology matured and the first exoskeletons were ready to be used.

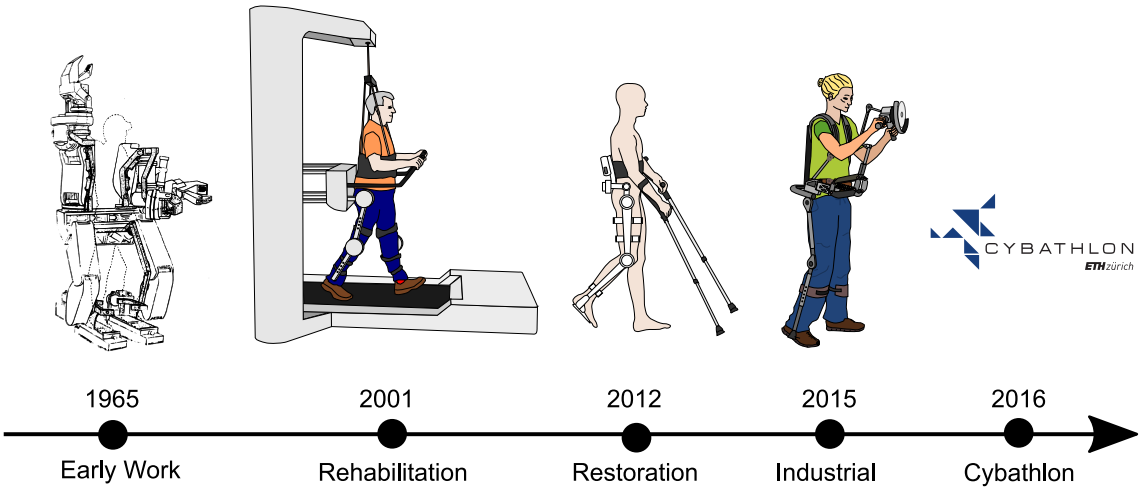

Figure 1.5: Timeline of exoskeleton developments (not in scale).

With the beginning of the 21st century, the first exoskeleton products made their way into the market and became accessible to an increasing number of users. One of the first applications was gait rehabilitation in stroke and spinal cord injured patients. An example is the gait rehabilitation exoskeleton Lokomat® (Hocoma AG, Switzerland), that was released in 2001 and has since been used in hospitals and rehabilitation centers worldwide. In 2013, Hocoma announced the shipment of the 500th device. Development continued in the first decade of the 21st century in an increasing number of research labs and companies. Since 2010, several gait assistance and restoration exoskeletons have been presented and gradually introduced to the consumer market. Most of them are designed to enable paraplegic users to leave the wheelchair and walk upright with the support of the device. Examples are the ReWalk™ (ReWalk Robotics, Israel) and the Indego® (Parker Hannifin, USA). An increasing number of these medical exoskeletons are being certified, for example, CE certification in Europe or FDA certification in the US, for clinical use and also for home use, allowing them to be used outside of medical facilities. In September 2016, ReWalk Robotics announced the 100th exoskeleton delivered for home use.

Besides medical applications, several manufacturers started developing exoskeletons for industrial use and just recently (around 2014-2015) introduced their first systems. Passive systems (without motors) are increasingly popular, as actuators are not always required to relieve the exoskeleton user of a payload or their body weight. For certain applications, even single articulated exoskeletons that support only one joint are sufficient to provide support. This makes them lighter and cheaper than their actuated (and thus larger) counterparts.

In addition to all the development efforts, an increasing number of manufacturers started to promote their systems to a wider audience to demonstrate their capabilities and increase awareness of the technology. In 2012, Claire Lomas, who has paraplegia, used a ReWalk to participate in the London Marathon and crossed the finish line after 17 days. In 2016, she participated in a half-marathon and finished after 5 days. In 2014, an paraplegic exoskeleton user executed a symbolic kick-off at the FIFA World Cup in Brazil. He was assisted by a 'mind-controlled' (by measuring brain activity) exoskeleton that was developed as part of the Walk Again project. In October 2016, ETH Zurich in Switzerland hosted the first Cybathlon, a competition in which, among other disciplines, exoskeleton-assisted paraplegic users, called pilots, raced each other in an obstacle course. At this demonstration of pilot skills and technology, the exoskeleton users had to solve tasks such as sitting down on a couch and standing up again, walking up and down slopes, walking over stones like you would cross a shallow mountain river, and conquer stairs. At this first edition, none of the participating pilots was able to solve all obstacles, and it took even the fastest teams more than 8 minutes to finish the 50 m long obstacle course while solving most of the obstacles. In comparison, an unimpaired young adult is able to complete all tasks in less than a minute without being in a rush. The next event will take place in 2020. It will highlight what progress will have been made between the two events.

This look back shows that exoskeletons have been around for quite some time, but just recently really took-off with the first systems becoming available outside of research labs. While today many systems are still limited in their performance, they have great potential, and it will be very exciting to see where the technology is going and what opportunities it will present in the future.

1.3 User-Exoskeleton Interaction

The general task of all exoskeletons is to provide the right amount of support at the right time. To ensure that, the interaction between exoskeleton and user has to be bidirectional. It can include the exchange of a variety of input and feedback modalities, such as forces, movements, and biosignals (Figure 1.6).

Robotic exoskeletons provide support by transferring the power of their actuators to the user. In addition, their rigid structure can provide stabilization, protection, and can transfer the weight of a payload to the ground, relieving the users from carrying it themselves. A robotic exoskeleton consists of the same components as any 'normal' articulated robot. A rigid frame, joints and actuators enable movements. Sensors, electronics, computers and software are used to control the device.

Because robotic exoskeletons do not act independently but rather in coordination and synchronized with the user, several additional components are necessary. Mechanical interfaces connect exoskeleton and user to transfer support. Adjustment mechanisms can adapt the length of the exoskeleton segments to fit different users. Several approaches are possible to control the exoskeleton. Sometimes, exoskeletons are directly controlled by the user through buttons or steering body movements. Or additional sensor systems can be

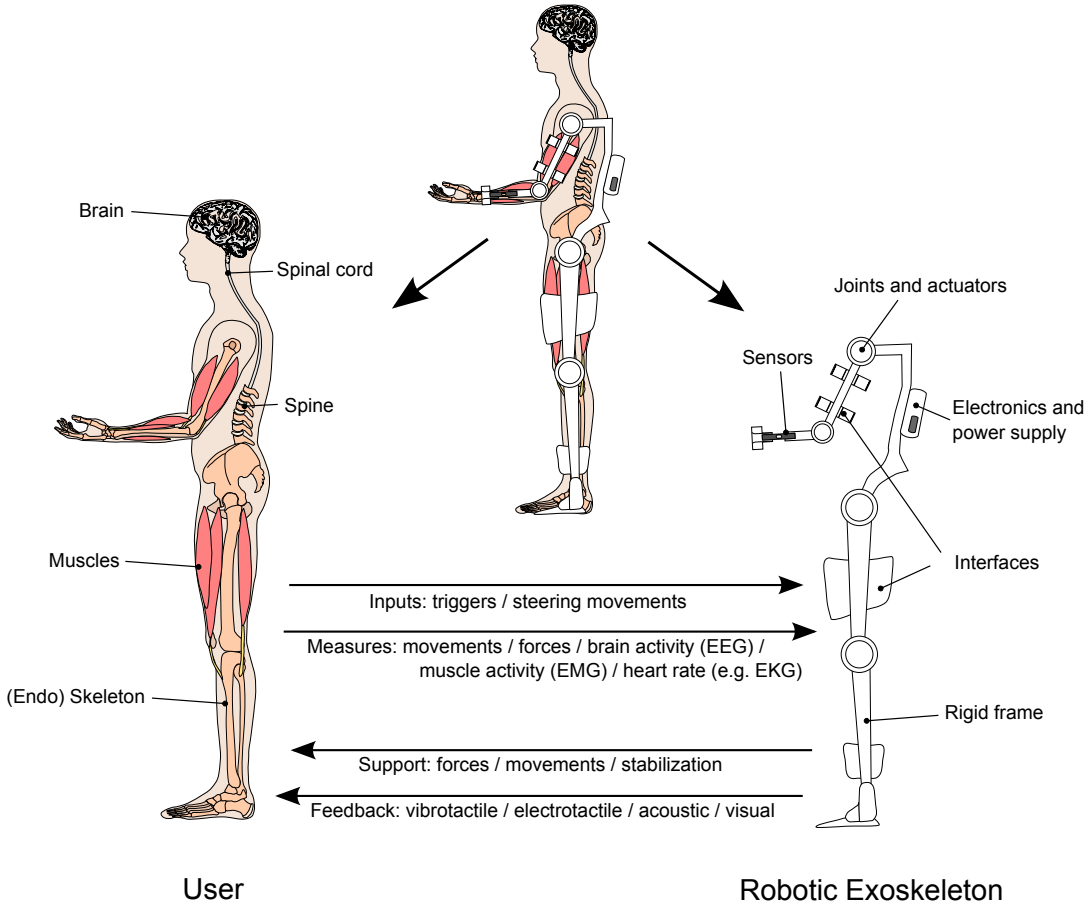

Figure 1.6: Illustration of user-exoskeleton interaction.

integrated to improve the exoskeleton's autonomy, for example, by measuring biosignals such as muscle activity, brain activity or heart rate from the user, allowing the exoskeleton to better anticipate the user's intentions and to react accordingly. In the other direction, feedback systems such as sounds (acoustic feedback), blinking lights (visual feedback) or vibrations similar to a mobile phone (tactile feedback) can be used to inform the user about upcoming exoskeleton actions to improve usability.

1.4 Limitations and Constraints

After reading these first sections, you might think that robotic exoskeletons are the solution to many problems and will augment the physical capabilities of everyone who uses them. But, if we look closer at the field and the technology, it also becomes apparent that today's robotic exoskeletons still have several limitations and are, in many cases, simply not good enough. Aside from providing support, certain constraints and disadvantages can occur at the physical connection between the exoskeleton and the human user.

1.4.1 Mass

The first potential problem is the exoskeleton's mass (Figure 1.7). This mass may have to be carried by the user. Or, the mass can considerably modify the person's center of mass, inducing altered and unnatural poses. Additionally, the exoskeleton's mass has to be moved and accelerated by the user when moving around. Especially distal masses – at the end of the arms and legs – are exposed to high accelerations when the human is performing fast movements, thus significantly increasing the load on the human. As a result, the exoskeleton user may fatigue faster or may have reduced strength.

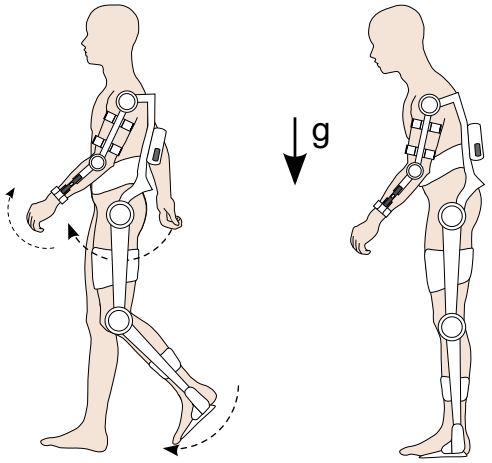

Figure 1.7: The user has to carry and move the exoskeleton's mass.

Of course, it is not always the case that the user has to carry, move and balance the exoskeleton's entire mass. Many exoskeletons, especially stationary ones that are connected to an external structure and the ground, can compensate and support at least a part of their own mass. Nevertheless, it is very likely that a high exoskeleton mass will negatively affect the user.

1.4.2 Kinematic Mismatch and Misalignment

Another potential problem is the exoskeleton's rigid structure (Figure 1.8). This structure may cause problems if the exoskeleton does not perfectly resemble the users' anatomy, e.g., spanning a human joint without incorporating a corresponding joint itself. For example, human hip joints have three degrees-of-freedom: flexion/extension, abduction/adduction and internal/external rotation. However, many hip exoskeletons typically only move along the flexion/extension axis. This restricts the human's movements, similar to a ski boot that constrains your ankle movements (Figure 1.8(a)).

Another source of constraints related to the exoskeleton's rigid structure may come from misalignment between the axes of the exoskeleton joint and the corresponding human joint (Figure 1.8(b)). Misalignment causes constraints due to the offset between the centers of rotation of two bodies that are mechanically connected. It originates mainly from inaccurate alignment during the setup, or slippage of the exoskeleton along the limb during operation.

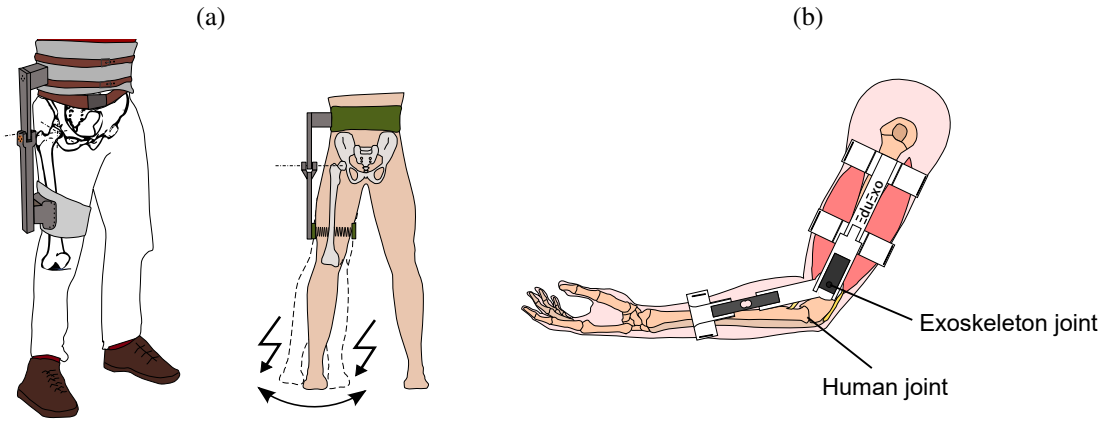

Figure 1.8: The exoskeleton's rigid structure can constrain the user's movements: (a) a one degree-of-freedom exoskeleton hip joint connected to a three degree-of-freedom human hip joint; and (b) misalignment between the user's and the exoskeleton's joints can cause constraints.

1.4.3 Interfaces and Size

The interfaces that connect exoskeleton and human may already negatively affect the user (Figure 1.9(a)). Most exoskeleton interfaces are designed as braces or textile cuffs that are wrapped around the human limbs and torso. A tight fit is often required to enable the transfer of the loads and avoid slipping and shifting of the interface on the skin. These interfaces may not fit perfectly, and therefore cause discomfort or even pain and injury when performing movements. If the fit is too tight, it limits the increase of muscle diameter during contraction, creating high compression on the soft tissue that causes soreness or skin irritation. The interfaces may also cover large parts of the skin, like an additional thick layer of clothing, and cause overheating of the user. The interfaces can also span multiple joints, for example, a torso interface over multiple vertebrae, limiting the user's mobility.

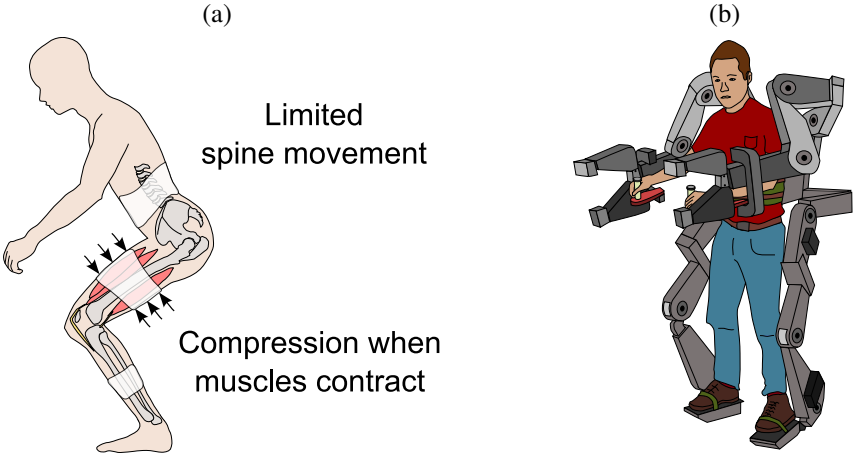

Figure 1.9: Other potential sources of constraints: (a) the exoskeleton interfaces can cause discomfort or limit movements; and (b) the size of the exoskeleton may cause collisions with the environment.

Another potential cause for constraints is the size of the devices themselves (Figure 1.9(b)). The rigid structure, large actuators, mobile power supply and the control computers all add volume to the human-exoskeleton system. This added volume can increase the risk of a collision with the environment or between parts of the exoskeleton, e.g., the left and the right legs. It may also restrain interaction with the environment, such as preventing the user from sitting down on a chair because it is not wide enough for the exoskeleton.

1.4.4 Sum of Benefits and Constraints

With these limitations in mind, the performance of an exoskeleton can be considered as the sum of the benefits it provides minus the constraints it imposes on the user (Figure 1.10). If the benefits are dominant, it will reduce the effort to conduct a specific task. If, on the other hand, the constraints outweigh the benefits, using the exoskeleton will make the task more difficult.

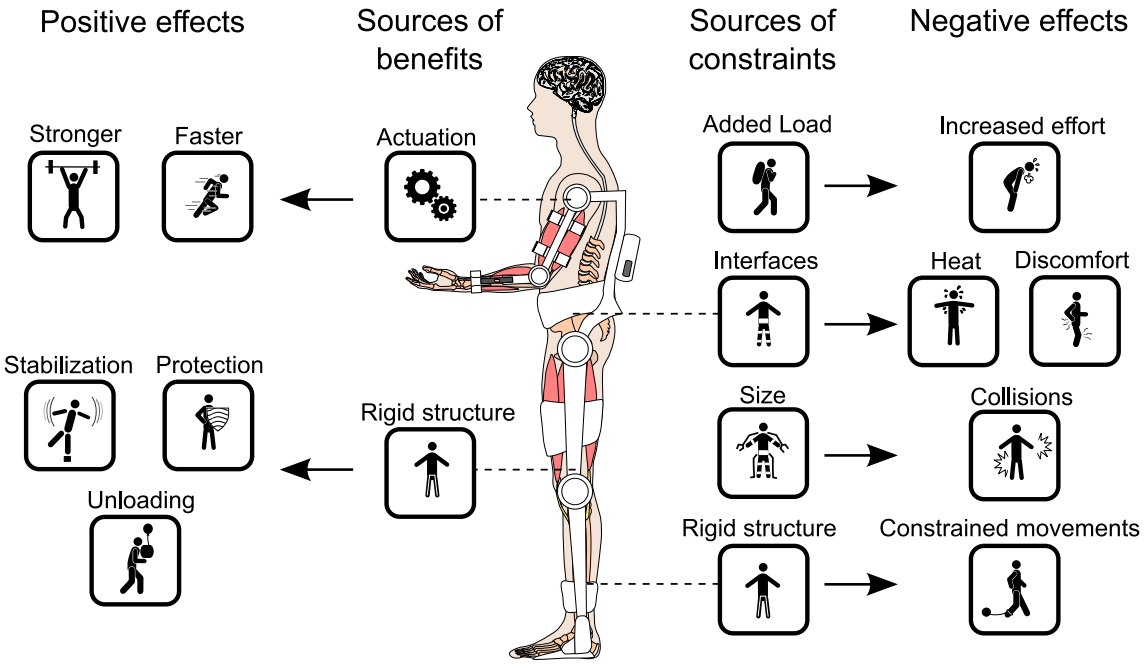

Figure 1.10: The overall performance of a robotic exoskeleton can be considered as the sum of its benefits minus the constraints it causes.

The challenge for developers is to design a system that provides the right amount of support while imposing as few constraints as possible on the user. The dilemma is that it is often impossible to optimize every aspect of the exoskeleton simultaneously because some of them trade-off others. For example, an exoskeleton with a stronger motor usually has an increased mass. Or, the larger interface required to transfer higher forces could impose additional interface constraints. The exoskeleton will offer a benefit only if we achieve a good balance. Therefore, the development can be a bit of a balancing act between increasing benefits and reducing constraints.

1.4.5 Task and User

To make things even more complicated, it is important to understand that benefits and constraints are not absolute, but rather depend on several factors. One factor is the activity that is performed while wearing an exoskeleton. Running involves higher accelerations and forces than walking. Climbing stairs requires a larger range of motion in the joints than level walking, potentially causing larger constraints due to kinematic mismatch, joint misalignment or interfaces. Another factor is the user. A strong, well-trained worker will react differently than an elderly person to the same exoskeleton weight. An impaired user might perceive a rigid structure as more supportive than a healthy user, who might perceive it as constraining. As a consequence, robotic exoskeletons are usually highly specialized devices that are only suitable for a narrowly defined user group and application, and can be completely useless for other applications.

1.5 The EduExo Exoskeleton

Now that you have some basic understanding of the field of robotic exoskeletons, let's start building our own exoskeleton by using the hardware provided together with this handbook. The EduExo exoskeleton (Figure 1.11) covers many aspects also found in 'professional' exoskeletons to teach you relevant knowledge.

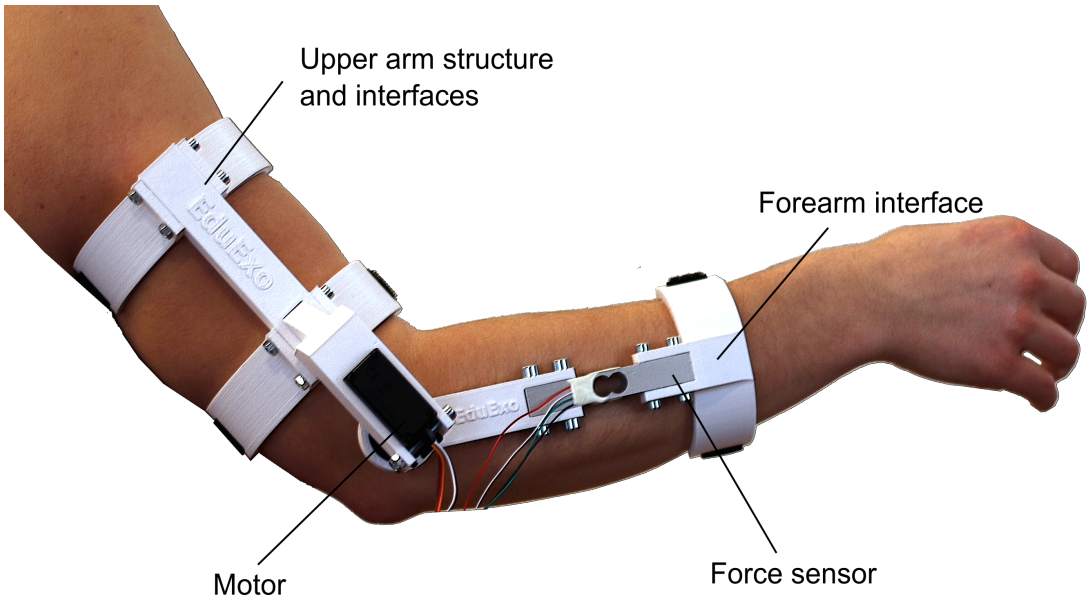

Figure 1.11: Overview of the EduExo hardware.

The EduExo is, of course, subject to some limitations regarding power and accuracy to keep it affordable and safe.

To make sure that your expectations match our intentions, here are quick facts about the EduExo kit:

EduExo Quick Facts

What it is:

- An educational robotics kit that will help you learn about exoskeleton technology.
- A do-it-yourself kit that requires active participation and willingness to learn new things.

What it is not:

- A medical device that is intended to be used for any kind of medical application.
- An exoskeleton that will make you super strong. The actuation is only intended to illustrate basic exoskeleton principles, and is weak. You will not be able to do more chin-ups. It rather provides a gentle guidance of your voluntary movements.
- A robot that works out of the box. You have to make it work, that is a feature!

For whom it is:

- For high school and college students who want to learn about robotic exoskeletons.
- For makers and hobbyists who are looking for a fun project in a fascinating field.
- For teachers and professors who want to set up exoskeleton courses or labs.

The EduExo exoskeleton is designed to support a user executing elbow movements. It spans the human elbow joint and is connected to the upper and lower arms of the user through mechanical interfaces. One single motor at the elbow joint guides the elbow flexion and extension movement. The motor of the EduExo is not very powerful; it rather demonstrates exoskeleton-user interaction while being safe to use without any risk of injury. An angle sensor integrated in the motor measures the elbow angle. A force sensor in the lower arm segment measures the interaction force between the exoskeleton and the user. The EduExo is connected to an Arduino microcontroller, on which the control system is programmed. The microcontroller itself can be connected to a PC to communicate with a computer game. The following chapters will introduce, step-by-step, the components and their functionality. With the help of the tutorial section in each chapter, you will assemble and program the exoskeleton, add functionality, and test new concepts.

The EduExo is designed for users that are new to robotic exoskeletons. Basic programming and electronics skills are beneficial, but you can learn everything you need while using the kit. We encourage you to solve the tutorials on your own, but they also include sample solutions to help you in case you get stuck.

1.6 Preparation and Tools

In order to use the EduExo, some preparations are necessary and you will need some tools and equipment (Figure 1.12).

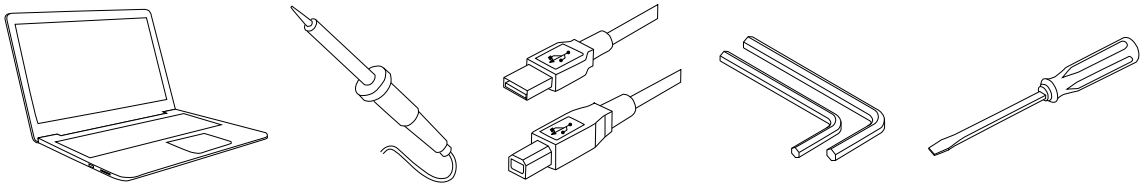

Figure 1.12: Tools and equipment required to build and program the EduExo in addition to the hardware provided in the kit: a computer, a soldering iron (and some solder), a USB A/B cable, hex keys and a flat-blade screwdriver.

To assemble the EduExo with the parts provided in the kit (Chapter 2), you need hex keys sizes 2.5 mm and 3 mm, and a flat-blade screwdriver. To program the EduExo and create a computer game (Chapters 3, 4 and 5), you need a computer. You also need a USB A/B cable to connect the Arduino microcontroller and your computer. Parts of the electronics (Chapter 3) can be connected without soldering, with the breadboard and jumper cables that are included to get you started. The force sensor requires soldering to connect it; for this, you need a soldering iron and some solder. Do not worry If you have not done this before, the equipment is not expensive and we include a step-by-step explanation in the tutorial. Some additional electronic equipment and tools (multimeter, gripper,...) will facilitate your work at some point, but are not necessary. If you are a proud owner of the Maker Edition and not the boxed EduExo version, you will need to additionally order the components and manufacture all parts yourself. For that, you need a 3D printer and sewing equipment, either needle and thread or a sewing machine (Figure 1.13).

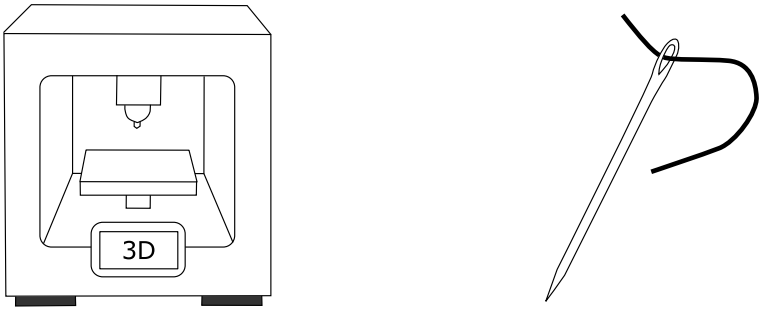

Figure 1.13: If you start with the maker edition, you additionally need a 3D printer and sewing equipment.

To program the exoskeleton's microcontroller (Chapters 3, 4 and 5) you need to install the Arduino IDE. It is free and can be downloaded on the Arduino website for several operating systems. For the 'virtual reality and video games' tutorial (Chapter 5), you need to install the Unity 3D game engine. It is also free for private and non-commercial use, but requires registration. So far, we have only tested the Windows (7 and 10) versions and

cannot promise that the communication between exoskeleton and game works similarly on other platforms.

Due to shipping regulations, no battery is included. For all the tutorials, we use the Arduino's power out to supply the sensors and the motor. The Arduino itself has to be battery-powered when wearing the exoskeleton for safety reasons.

Chapter 2

Anatomy and Mechanics

Summary

In this chapter, you will learn the basics of force generation in humans and robotic exoskeletons, and the interaction between the user and the exoskeleton. Then, we will introduce you to the mechanical components of the EduExo and explain its design. The tutorial will guide you through the manufacturing (if you have the Maker Edition) and assembly of the exoskeleton.

2.1 Human Anatomy and Strength

Our musculoskeletal system enables us to move: run, jump, grasp objects and so on. It consists of bones, joints, muscles, tendons and a few other tissues. Bones give structure to our bodies. The connection between bones is called a joint. The geometry of our joints, and how they are arranged and connected, define the kinds of movement we can execute. Movements are generated by muscles that produce forces. When muscles contract, they pull (but they cannot push!). Muscles are connected to bones through tendons. Muscles and tendons span our joints; when muscles contract, they pull on the bones they are attached to, bringing the bones together. If we take a closer look at the joint that concerns us for our exoskeleton – the human elbow joint –, we can identify certain characteristics that define its functionality (Figure 2.1).

The elbow joint is a hinge joint. This means that we have one possible movement (often referred to as degree-of-freedom or DoF), which is the flexion and extension of the elbow. When talking about human joint movements, the direction of the movement is usually specified. 'Elbow flexion' means that you move your elbow in such a way that your hand goes towards your shoulder (presenting your impressive biceps); 'elbow extension' is the reverse movement, when you straighten your elbow joint. Accordingly, the muscles that generate elbow flexion are called (elbow) flexor muscles, and the muscles that extend the joint are called (elbow) extensor muscles. Examples of muscles that generate movement at the elbow joint are the biceps (flexor) and the triceps (extensor). To move a joint,

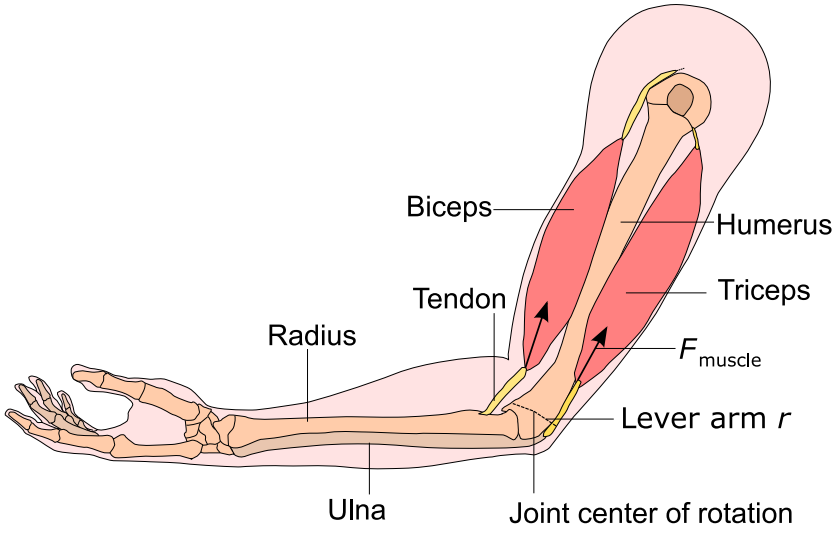

Figure 2.1: The human elbow joint and some of the major muscles responsible for elbow movement.

a muscle has to span that joint (Fig. 2.1). For example, the biceps can also move the shoulder joint; however, this movement is not important for us here.

When muscles contract, they create a pulling force F_{muscle}. Because of a lever arm r between muscle force and joint center of rotation, the force results in a torque T_{muscle} at the joint level that induces a joint rotation (Equation 2.1). If we want to move back, the muscle relaxes and a muscle on the other side of the joint (antagonist) contracts to pull the joint back.

$$T_{\text{muscle}} = F_{\text{muscle}} \cdot r \tag{2.1}$$

When we look at our elbow joint, we realize that the movement is limited. Usually, full extension of the elbow (0° flexion) forms a straight arm, and we can flex the elbow until the lower arm touches our upper arm at about 150° flexion. This range of angles is called the range of motion (RoM) of the elbow. The RoM is different for every joint. Also, there can be variations for the same joint between different people. It is important to know the RoM of joints when designing an exoskeleton, for example, so the exoskeleton does not move beyond these natural RoMs for safety. Otherwise, it could hurt the user when it is equipped with strong motors.

2.2 Robotic Kinematics and Force

Similar to how humans have joints and bones, robots have joints and segments. The way the joints of a robot are arranged, and the length of the segments that connect the joints, are described by the robot's kinematic configuration. This kinematic configuration and the shape of exoskeleton should appropriately resemble the human anatomy to best fit and support its users (Figure 2.2). For example, an exoskeleton that is supposed to support a certain human movement should be able to execute the same movement itself. As a

counter-example, an exoskeleton that spans a human joint without having a corresponding joint itself, will very likely constrain the user's movement.

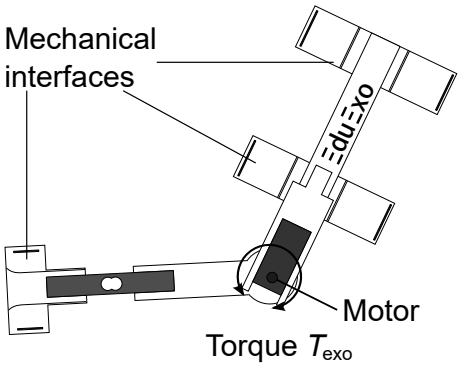

Figure 2.2: The exoskeleton mechanics: the motor at the joint creates a torque T_{exo} that is transferred through the mechanical interfaces to the human inside.

Passive (non-actuated) joints are used to allow free joint movements, while still transferring certain forces and loads across the joint. Therefore, they can be very useful in applications where the exoskeleton unloads the user (e.g., Figure 1.4(c)) without actively supporting the user's joint movements. However, the joint of an exoskeleton is often equipped with an actuator. The most common actuators are electric motors that can create a torque T_{exo} around the joint axis. As mentioned before, to avoid injuries, it should be ensured that the exoskeleton joint can not exceed the range of motion (RoM) of the corresponding human joint. This can be achieved by mechanical end-stops that limit the range of motion of the exoskeleton's joint, ensuring safe operation even if the actuator does not work properly.

The control of the exoskeleton is usually done by the control system, which is implemented on a computer. The control system of exoskeletons is programmed to support the user. It does this by acquiring the information provided by the exoskeleton's sensors, and responds by using the exoskeleton's actuators to create forces and movements. You will learn more about control systems later (Chapter 4).

2.3 Human Robot Interaction

To support the human, the exoskeleton is connected to the human (Fig. 2.3). The joint axes of the exoskeleton should coincide, if possible, with the corresponding joint axes of the user to ensure 'kinematic compatibility'.

The torques created by the exoskeleton's actuators T_{exo} are transferred through interaction forces in the interfaces F_{exo} to the user, and, ideally, reduce the loads on the human muscles. The torque total in the human elbow joint is then the sum of the torque T_{muscle} created by the human muscles and the torque T_{exo} provided by the exoskeleton (Equation 2.2).

$$T_{\text{joint}} = T_{\text{muscle}} + T_{\text{exo}} \tag{2.2}$$

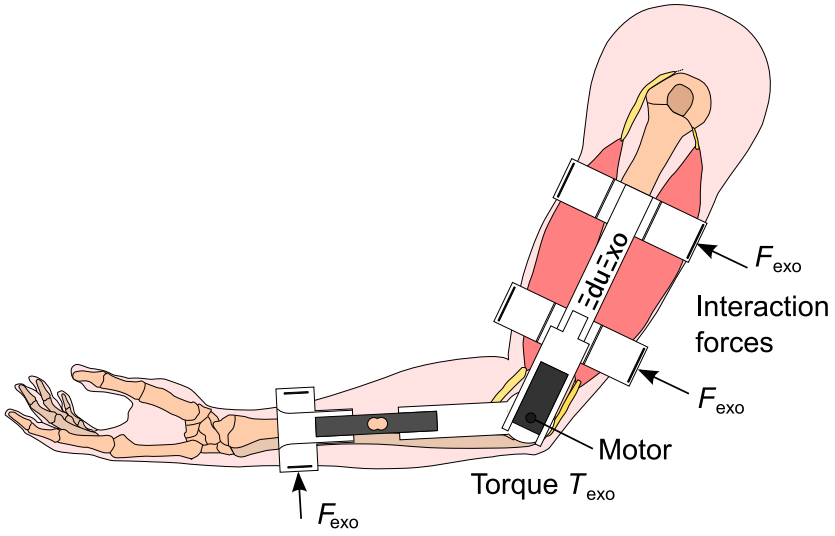

Figure 2.3: The exoskeleton connected to the human arm. If the exoskeleton applies a torque, this torque is added to the human's torque.

If both the human and the exoskeleton push in the same direction (which should be ensured by the control system), the human can utilize the exoskeleton's force to execute movements or carry load.

As discussed in the introduction, for the design of an exoskeleton, it is important to consider that such a device can also create additional loads or constraints besides providing support. To minimize these effects, the exoskeleton should be adjustable in size to the user, provide all the joints required to execute the desired movements, and be as lightweight as possible.

2.4 Tutorial Mechanics

2.4.1 Maker Edition

If you have the Maker Edition of the EduExo, you will start by manufacturing the parts and ordering all the extra components. If you bought the EduExo kit, you can skip this section and immediately start assembling the device (go to section 2.4.2).

Ordering the parts

Your first job is to order all the parts you need (Table 2.1). Depending on your location, it might take some time to get the parts delivered, so it may be a good idea to start with that. You can find a continuously updated list of possible suppliers on the EduExo website (www.eduexo.com).

Table 2.1: List and description of all EduExo parts you have to order. See also list on www.eduexo.com with links to suppliers.

Part	Type
Motor	Analog feedback DC servo motor (Type 1404).
Microcontroller	Arduino UNO.
Force sensor	Load Cell 10kg; Straight Bar (TAL220).
Sensor amplifier	Instrumentation amplifier (INA125).
Resistor	60.3 Ohm through-hole resistor. Other values close to 60.3 Ohm are OK.
Breadboard	Any breadboard large enough to implement the amplifier circuit (Figure 3.16).
Cables	10 jumper cables. Two 4-wire extension cables for motor and force sensor.
Heat shrink tubing	For the soldered wires. Diameter 1.6 mm. Length around 25 cm.
Hook and loop fastener	For the cuffs. Around 1 meter of loops and around 10 cm of hooks. Width 2 cm. Elastic loop strips make the connection more flexible and comfortable (if available). Also, self-adhesive hook strips will make handling easier.
Screws and nuts	Wood screws: M2x12mm (4 pcs). Socket head screws: M4x10 mm (4 pcs); M4x30 mm (2 pcs); M4x35 mm (4 pcs); M5x10 mm (4 pcs). Nuts: M4 (6 pcs).

3D printing

Next, you will have to print all the parts of the exoskeleton structure (Table 2.2, Figure 2.4).

Table 2.2: List and description of all EduExo parts you have to print,

Part	Description
UpperArmSegment	The upper arm segment that connects the two upper-arm cuffs and motor adapter.
UpperArmCuff	The two upper-arm cuffs (this part has to be printed twice).
MotorAdapter	The part that connects and fastens the motor.
InterfaceMotorSensor	Part between motor and force sensor.
LowerArmSegment	Lower-arm cuff and interface to force sensor.

You can load these files on your printer to print them. All parts were designed to be printable with any extrusion (Fused Deposition Modeling - FDM) printer without support material. In case you have a printer that uses another method, we would be happy if you inform us about your results. If you run into any problems with the printing or the parts (e.g., stability), let us know and we can work together on a solution.

Figure 2.5 shows all the parts loaded into the software of a 3D printer. Make sure that

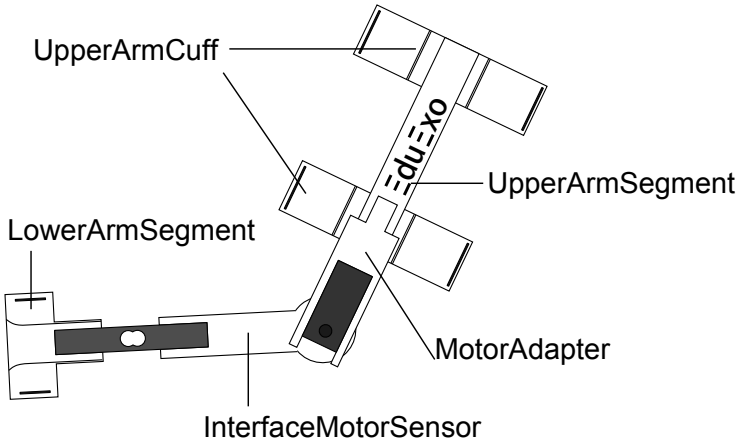

Figure 2.4: The five different parts of the EduExo that have to be printed. The part 'UpperArmCuff' has to be printed twice.

the correct side is facing up (as illustrated) to ensure smooth printing. Note that you have to print the part 'UpperArmCuff' twice.

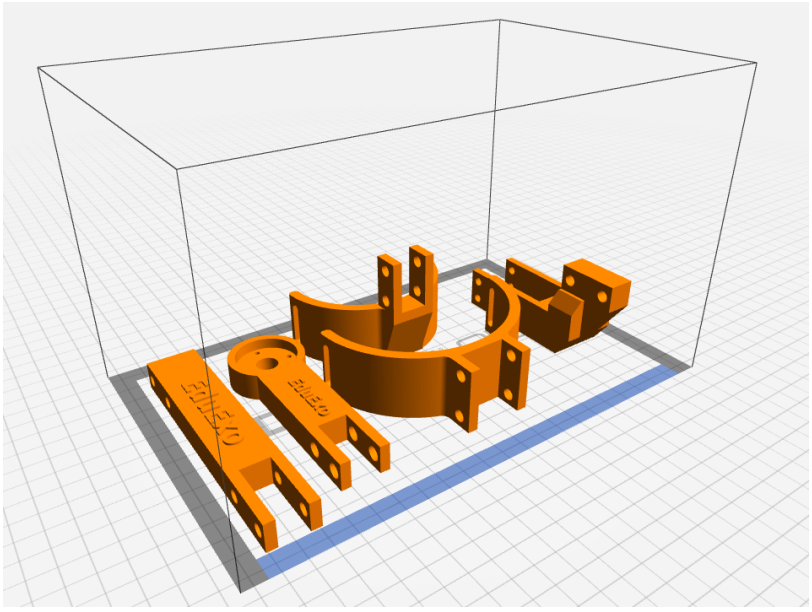

Figure 2.5: The STL files loaded to a 3D printer software. Make sure that the correct side is facing upwards before you print.

Manufacturing of the cuffs

Once you printed all parts, your next task will be to finalize the cuffs. The cuffs are the interface that connect user and exoskeleton. Therefore, they should ideally enable quick and easy donning and doffing of the exoskeleton, and fit a certain range of limb diameters, while ensuring a good and stable interface to transfer forces between exoskeleton and user.

To realize all that, the EduExo cuffs combine the (printed) rigid interface segments with hook-and-loop fasteners. As you already printed the rigid parts of the interface, it is now time to add the fastener.

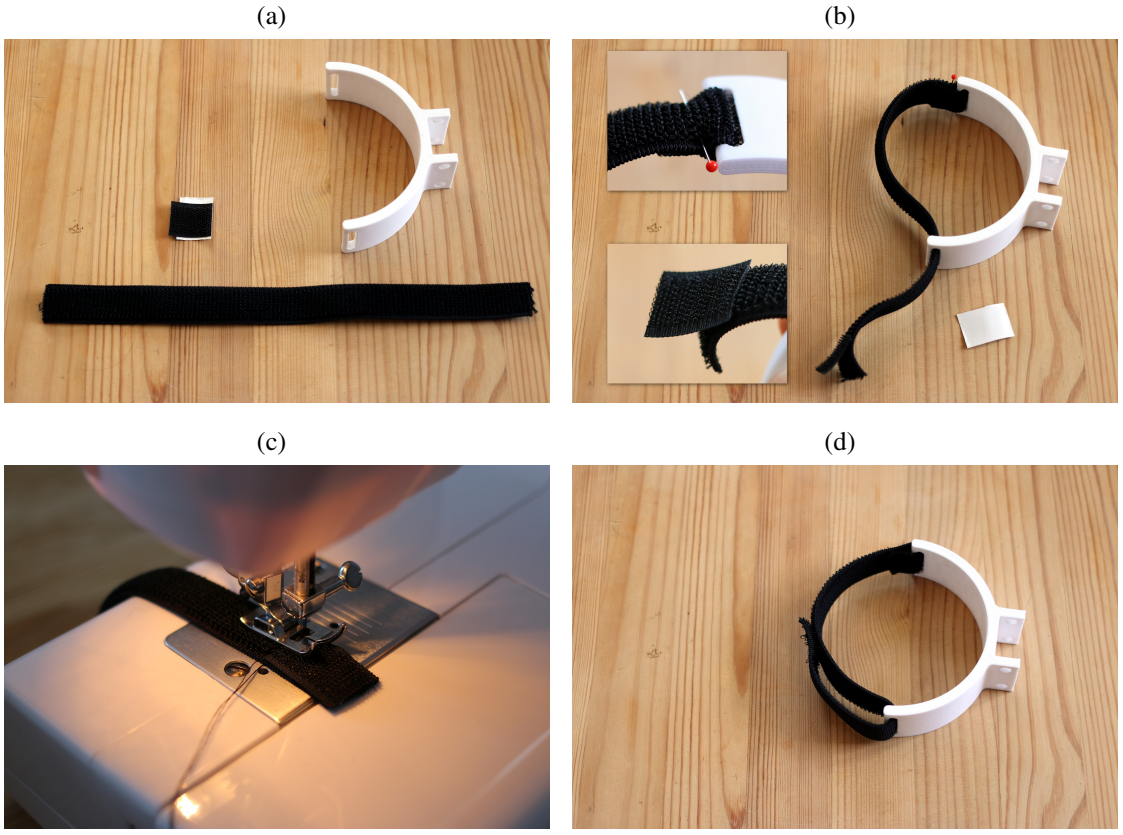

Figure 2.6: Finalizing the cuffs: (a) the parts you need: the 3D printed interface parts, a long strip of loops and a short strip of hooks for the fastener; (b) how to connect the parts; (c) sewing the strips; (d) the finalized cuffs.

This involves a few steps (Figure 2.6) and requires sewing. You can do it manually with a needle, or with a sewing machine, if you have access to one. The first step is to cut the fastener to the appropriate length. The long segment is a loop strip, the short one is a hook strip. You can use an elastic loop strip to better adjust the interface to changing arm diameters (e.g., if you flex a muscle). A self-adhesive hook strip will make the sewing simpler as you can glue the parts together before sewing; otherwise, you can use a pin to hold them together while sewing. In the illustration (Figure 2.6(b)), we routed the loop strip through the printed part and then fixed it with a pin, and connected the hooks by gluing the strips together. Next, it is time for sewing (Figure 2.6(c)). You have to close the loop strip on the one side, and sew the hooks on the loop strip at the other end. When this is done (Figure 2.6(d)), you can remove the pins and repeat the steps for the other cuffs.

2.4.2 Assembling

Attention!

We know it is very tempting to test those braces and other EduExo parts and squeeze them a little to see how robust the are. Be aware that they are 3D printed plastic and that applying too much force will break them.

With all the parts ready, you can now assemble the EduExo, connect it to your arm and move it against the passive resistance of the unpowered motor. You will find all the components that you need in the kit (Figure 2.7). Although you can do most of the assembly without tools and relying on your strong grip, a set of hex-keys, a screwdriver (flat-blade) and pliers will make your life much easier.

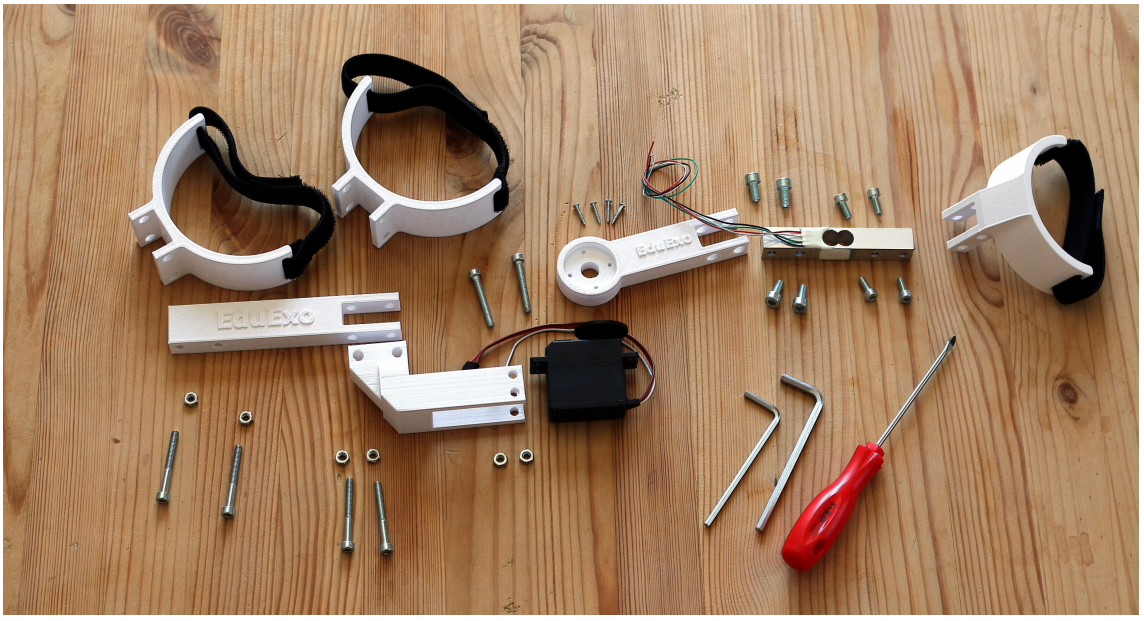

Figure 2.7: The disassembled mechanical exoskeleton components out of the box along with some tools to assemble it (tools not included).

For the mechanical assembly, you have to execute several steps (Figure 2.8). Start by assembling the upper arm segment by connecting the UpperArmCuffs with the UpperArmSegment and the MotorAdapter (Figure 2.8(a)). Next, prepare the motor and connect it to the InterfaceMotorSensor part (Figure 2.8(b)). The motor has an integrated end-stop and cannot rotate 360°. We will use this end-stop by connecting lower arm and motor in a way that it prevents the elbow joint from hyper-extending (moving beyond a fully extended, straight arm). To find the end-stop, you can use the round black motor adapter provided together with the motor: put the adapter onto the motor output shaft and turn it manually until it does not move anymore. Remove the round adapter again and connect it to the InterfaceMotorSensor with the four small M2 wood screws (Figure 2.8(b). Then, put the InterfaceMotorSensor part together with the round adapter on the motor shaft in a way that the elbow joint is straight when the motor is at the end-stop and can move

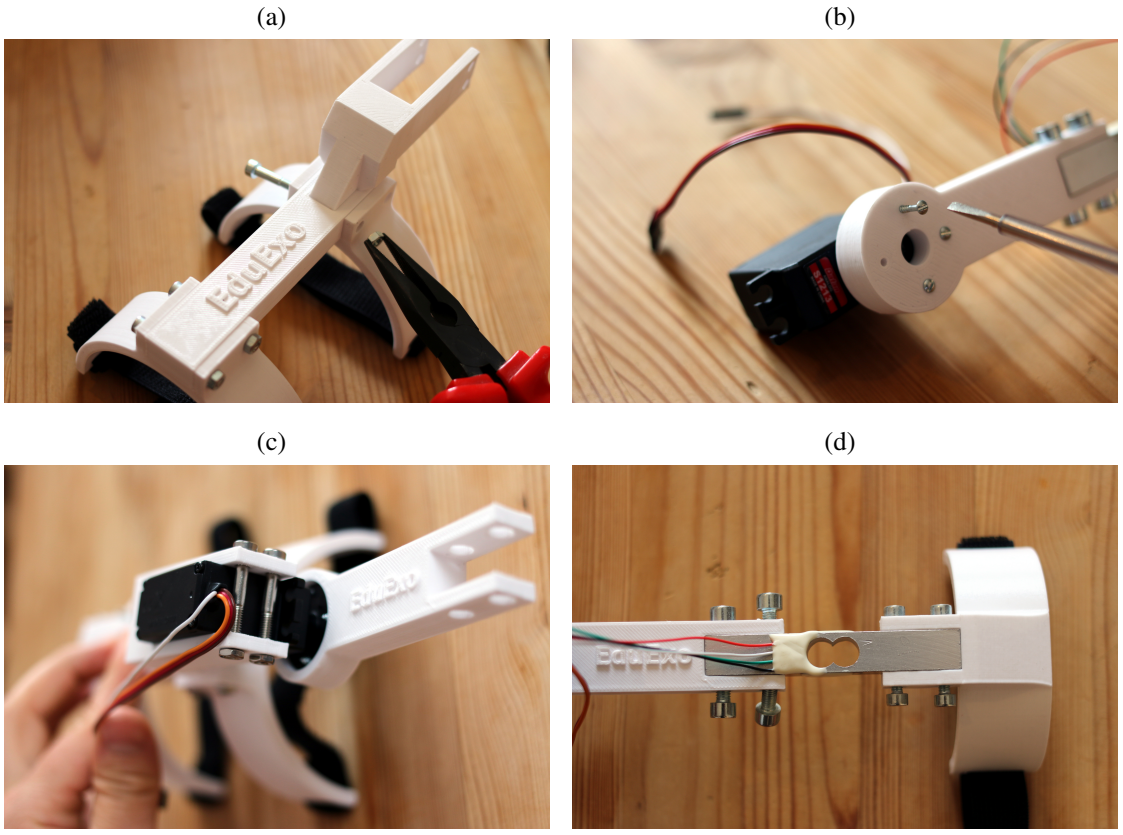

Figure 2.8: Assembling the EduExo: (a) use the long M4x35 mm screws and M4 nuts to connect the cuffs, the upper arm segment and the motor adapter; (b) to connect the motor to the lower arm segment, you have to first install the round black adapter with the black screw (not shown, both provided in the motor bag) and then connect the part 'InterfaceMotorSensor' with the four M2 wood screws; (c) the motor is installed into the motor adapter by clamping it with two M2x30 mm screws and nuts; (d) the force sensor and the part 'LowerArmSegment' are installed with a total of 8 short screws.

upwards when flexing the elbow. Don't forget to fix motor and lower arm segments with the black screw you find in the motor bag. When you have done that, you can connect the lower arm parts to the upper arm parts by clamping the motor in the printed MotorAdapter piece with two screws (Figure 2.8(c)). Finally, you can the install the LowerArmSegment and the force sensor (Figure 2.8(d)).

2.4.3 First Test

When you executed all assembly steps, it is time to put it on you arm and (carefully) test it. Slip in, tighten the cuffs and flex your elbow joint (Figure 2.9). You should be able to move your elbow with only little resistance of the passive motor.

You can already use the passive exoskeleton as a tool to explore some of the negative aspects of exoskeleton use we discussed in the introduction. Take a closer look at your elbow joint and ensure that the exoskeleton joint and your elbow joint are aligned to avoid the aforementioned misalignment problems. Now move ahead and intentionally misalign

Figure 2.9: The fully assembled exoskeleton.

the exoskeleton by sliding it up and down along your arm. Carefully flex you elbow. Do you feel a difference? Is the exoskeleton constraining your movements when it is strongly misaligned or is the effect negligible? Just be careful not to break the exoskeleton by taking your first experiment too far.

How about other constraints, such as the weight or the size of the exoskeleton? The EduExo it not very powerful and therefore relatively small and lightweight. Lift and move your arm with and without the EduExo attached. Do you think wearing it on your arm would affect your stamina or increase the chance of a collision with the environment? What about the interfaces? As they are not custom designed to fit your arm, there is the chance they are a little too small or too big. We can adjust the cuffs to a certain degree with the hook and loop fasteners and our soft tissue will deform to adjust to the shape of the cuffs. But is it comfortable, or is it tight? Is it squeezing your arm when you flex it?

Simply use the possibility of having an exoskeleton at hand to consciously experience and feel what it means to wear an exoskeleton. Explore how design choices can influence the use of such a wearable robot and think about what you would (or will) improve. After all, this is what this kit is all about!

Chapter 3

Electronics and Software

Summary

In this chapter, we will introduce the theory behind the EduExo's electronic components, and discuss their functionality. In the tutorial, we will start by connecting the components and writing a basic software. Then, you will learn how to read the force and the joint position signals, and we will calibrate the sensors. This step is important and should be done before implementing the control systems in the next chapter.

3.1 Overview and Functionality of the Components

The main electronic components of the EduExo (Figure 3.1) are the motor with integrated angle sensor, the force sensor, an amplifier for the force sensor and the microcontroller. The microcontroller is connected to a computer for programming.

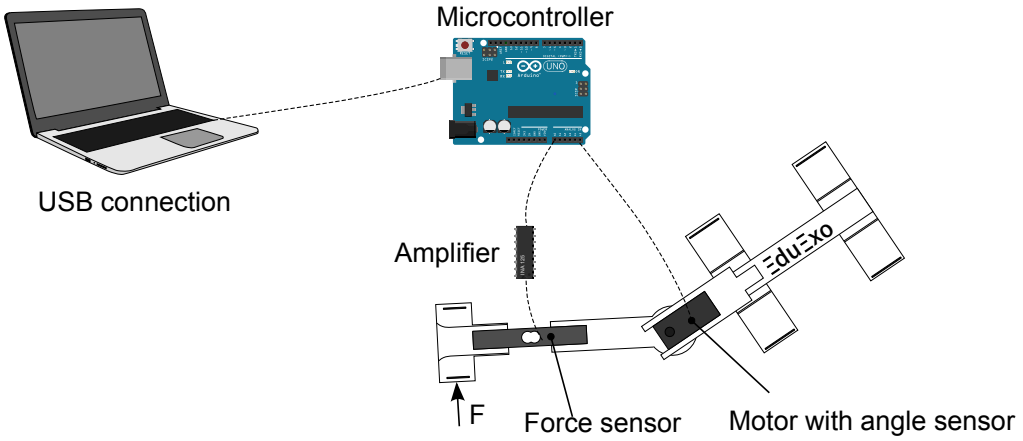

Figure 3.1: The main electronics components of the EduExo.

Attention!

Please be careful when handling the electronic components. You can cause damage by just touching the components – especially the pins of the chips – because of electrostatic discharge (ESD). We advise you to quickly read up on ESD before handling the electronic components. There are some basic rules that you should look up and follow (e.g., do not walk over that nice rug before handling electronics, and discharge yourself by touching grounded metal components like your heater - but don't burn yourself!).

In order to start operating the exoskeleton, we first have to connect the components. Then, we can start programming the Arduino software. This is usually done on an external computer, which is connected to the Arduino. The program is compiled on the computer and then uploaded to the Arduino.

3.1.1 Microcontroller

The main component of the exoskeleton electronics is the microcontroller. Microcontrollers are essentially miniature computers that have all components in one board. Usually, they have additional components and interfaces to directly connect to other devices (e.g., analog-to-digital converters and a variety of communication interfaces and buses). The microcontroller used for the exoskeleton is the Arduino UNO (Figure 3.2).

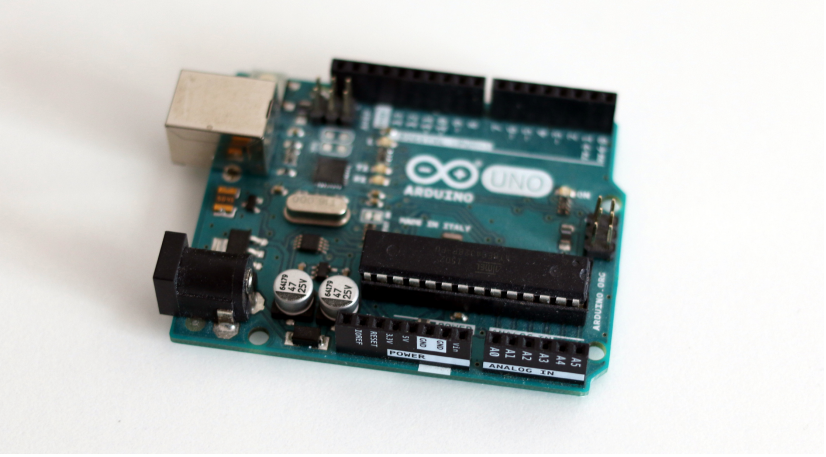

Figure 3.2: The Arduino microcontroller that is used to control the EduExo.

We will use it to read signals from the position and the force sensors, and to control the movements of the motor. The entire Arduino microcontroller family has a very large user base and, in case you run into any trouble or have questions that are not answered in this handbook, you can find a lot of help and tutorials online. You can start, for example, on their official website (www.arduino.cc).

3.1.2 Motor and Angle Sensor

The actuator used in this kit is an electric servo motor with a gear and control electronics within the motor housing. The servomotor also contains an integrated potentiometer to measure the rotation angle (position) of the motor's shaft. In total, the servomotor has four wires: two inputs for the power supply (red and brown), one input for the control signal to the servo motor (orange) and one output for the angle signal from the integrated position sensor (white). It is common that servomotors have an integrated position sensor, but not all servomotors allow access to the angle signal (with an extra wire).

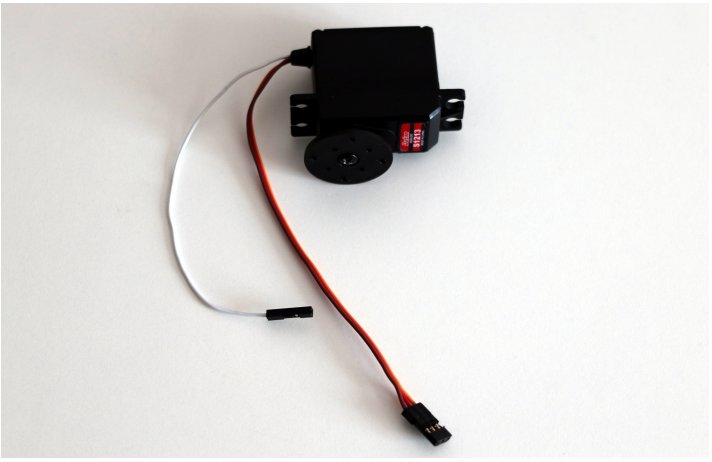

Figure 3.3: The analog feedback servomotor that is used as an actuator and joint angle sensor of the exoskeleton. The brown wire is the ground (GND), the red wire is the supply voltage, the orange wire is the position control signal and the isolated white wire is the position feedback signal.

The servomotor's integrated control system can already perform the basic (low-level) control, for example, move to a desired angle when the command is sent on the control wire (e.g., from the main microcontroller). The desired angle is encoded using pulse width modulation (PWM). In PWM encoding, the desired angle of the motor is proportional to the width (duration) of an electric pulse that is sent to the servomotor on the control wire. This signal is sent at regular time intervals to keep the motor at the desired angle (Figure 3.4).

For example, a pulse width of 1.5 ms every 20 ms could set the angle to zero degrees, a pulse width of 2 ms every 20 ms could set the angle to 90 degrees, and so on. Note that there are some variations of PWM that we will not worry about for now.

Additionally, the signal from the angle sensor can be sent to the microcontroller. The angle sensor is an analog potentiometer, configured as an adjustable voltage divider with three contacts (Figure 3.5).

Two contacts are connected to the supply voltage: one to V_s, and one the ground. The middle contact is a wiper that is rotating along with the motor axis. If the wiper is moved (in our case, by moving the elbow joint), R1 and R2 change: one increases while the other drops. Thus, the ratio between R1 and R2 changes. This leads to a change in the voltage V_{out} measured between the wiper and ground, which is the output signal (Equation 3.1).

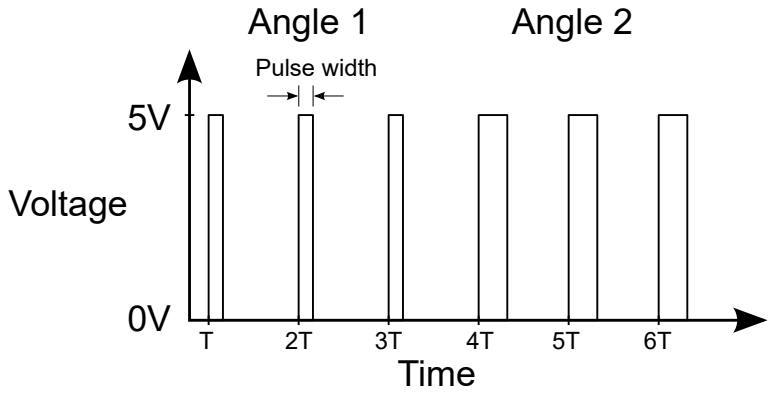

Figure 3.4: Illustration of how the desired motor angle is encoded using pulse width modulation (PWM) from the Arduino to the servo. The width (duration) of a pulse is proportional to the desired angle. In this example, the desired angle remains constant for the first three time periods, and then changes to a larger desired angle.

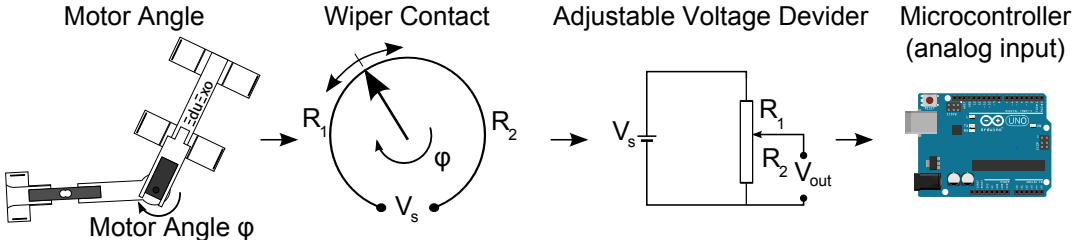

Figure 3.5: The setup to measure the exoskeleton's joint angle. A potentiometer is integrated in the servomotor. It is connected between the ground and the supply voltage V_s. When the joint moves, the wiper of the potentiometer moves along. As a result, the ratio between the resistors R1 and R2 changes, and the output voltage V_{out} changes accordingly. This voltage can be measured at an analog input pin of the microcontroller, and used to calculate the joint angle.

$$V_{\text{out}} = \frac{R_2}{R_1 + R_2} \cdot V_s \tag{3.1}$$

The output voltage is therefore proportional to a ratio between the two resistors, which is related to the position of the wiper. Thus, we can use the output voltage to measure the motor's angle.

The servomotor combines the functionality of an actuator and an angle sensor, and we will use both features later on. To use these functions, we will connect the servomotor's input (control) channel to a PWM pin of the main microcontroller. This pin will send PWM-encoded desired angles to the servomotor's integrated position control system, which 'takes over' and moves the motor towards the desired angle. The output of the potentiometer will be connected to an analog input port of the microcontroller to measure the joint angle. With these two connections established, we can send new desired position values to the servomotor and read out the joint angle.

3.1.3 Force Sensor

The force sensor (Figure 3.6) measures the interaction force between exoskeleton and human, and can be used to record the interaction and to control the exoskeleton.

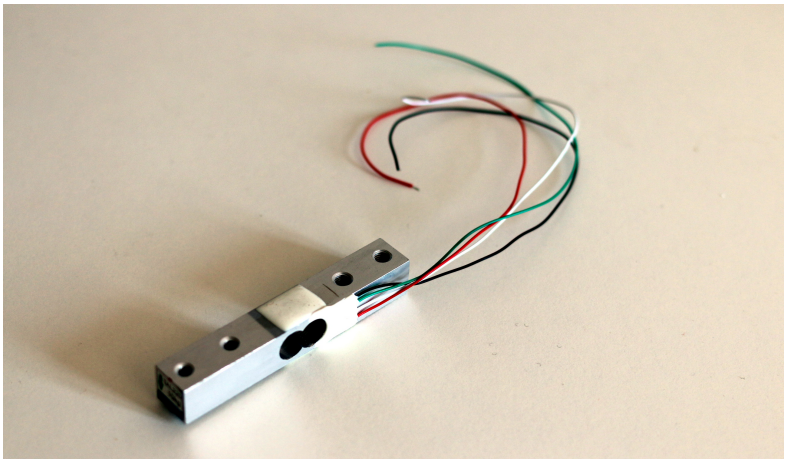

Figure 3.6: The force sensor used to measure the exoskeleton-user interaction forces.

To measure the interaction force, we first need to convert the physical force (in Newton [N]) into an electric signal (voltage measured in Volts [V]) that can then be recorded by the microcontroller. This is done by the force sensor (Figure 3.7).

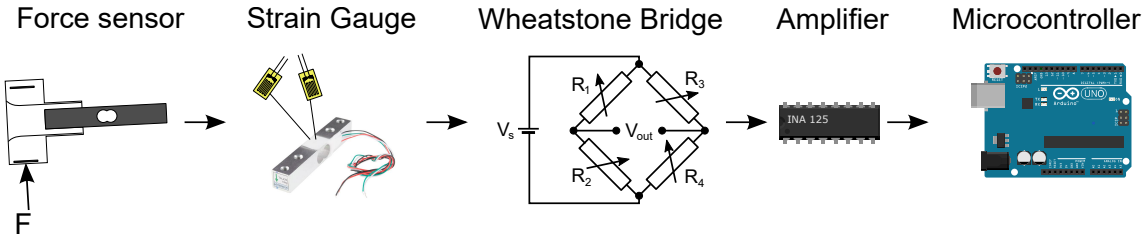

Figure 3.7: The setup to measure the user-exoskeleton interaction force. The force sensor is integrated in the exoskeleton so it is directly affected by interaction forces. When force is applied, the metal beam and attached strain gauges are deformed, changing the electric resistance of the strain gauges. These changes in resistance are measured in a Wheatstone bridge circuit: two of the wires of the sensor are for the supply voltage V_s, the other two are the output voltage V_{out} (the sensor signal). Because the output voltage is very small, it is connected to an amplifier chip. This chip is then connected to the Arduino microcontroller that then reads the amplified signal.

The force sensor is basically a piece of metal that has special electric resistors called strain gauges glued onto it. Strain gauges change their electric resistance when they are strained (pulled), even by very small strains. When a force is applied to the force sensor, the metal deforms by very small amounts, which are usually not visible to the naked eye. The strain gauges change their length along with the metal, which causes a change in their electric resistance. Typically, several of these resistors are connected in a special electric circuit, called a Wheatstone bridge, that allows precise measurements of changes in electric resistance. The output voltage of this circuit changes when the resistors change

their values. This voltage is the output signal of the force sensor, and is proportional to the applied force. Because the output voltage is very small, it is connected to an amplifier chip. This chip is then connected to the Arduino microcontroller that then reads the amplified signal.

3.2 Preparation

3.2.1 Installing the Arduino IDE

Before you can start programming the microcontroller, you have to install the Arduino Integrated Development Environment (IDE). You can download it for several operation systems on the Arduino website (www.arduino.cc).

When you install and open the IDE (Figure 3.8), you will find everything you need to write code and upload it to the Arduino board.

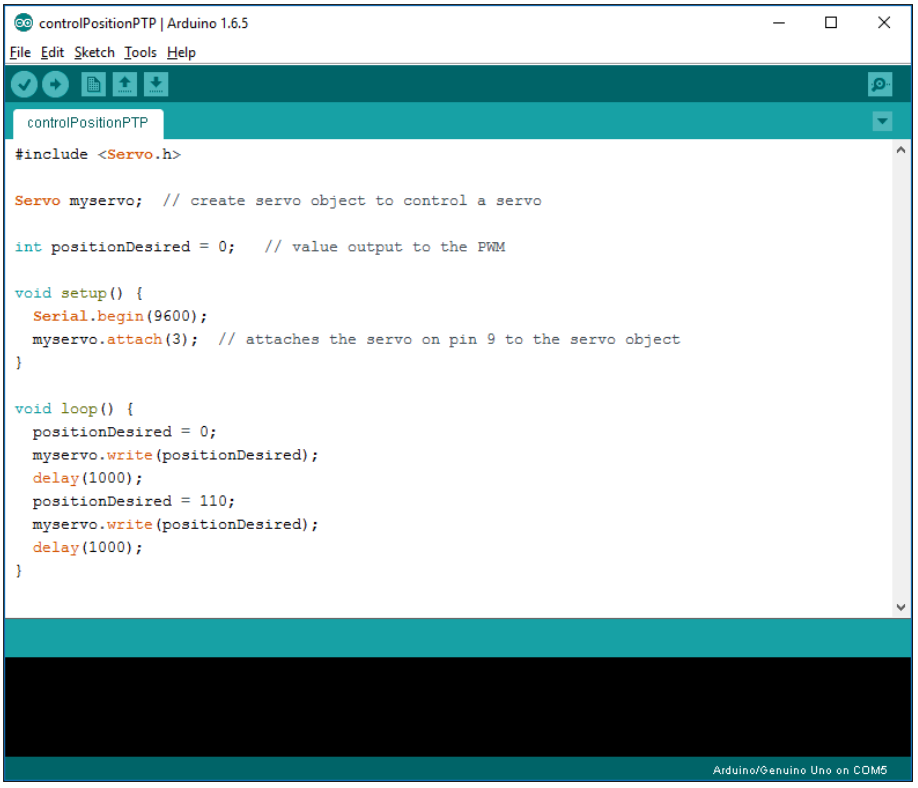

Figure 3.8: The Arduino IDE (Windows version).

This includes an editor with syntax highlight to improve code readability, a compiler to translate the source code into machine-readable files, all the tools you need to connect the Arduino and a serial monitor that comes in handy to read out the serial communication stream. To program the Arduino microcontroller, you have to connect it to your computer with a USB A-B cable. The programming language we use to implement the EduExo software is C.

3.2.2 Soldering

Soldering is a technique to physically connect two parts. The basic idea of soldering is to use (melted) metal as glue: metal is heated until it melts, then it is poured along the connecting interfaces of two parts, and, when the metal cools down and solidifies, a connection is made. Soldering can be used to mechanically connect two parts (to transfer forces) and to electrically connect two conductors (to transfer electric current). Regarding the connections of the EduExo, you can connect the motor without soldering, but soldering is required to connect the force sensor.

The only steps you have to do are to remove insulation from cables and wires, solder them together and apply heat shrink tubing to the connection. For that, you need a soldering iron and solder (the metal alloy that you melt), heat shrink tubing (provided in the EduExo kit) and, optionally, a helping hand tool or similar fixation aids for the wires. As you need both hands to solder, anything that prevents the wires from moving will make soldering easier. Additionally, a multimeter is one of the most basic devices when you work on anything electric. You can get a simple multimeter for a few dollars/euros and it will help you a lot finding short circuits, bad soldering, etc. If you never soldered, we would advise you to take a quick look at some basic introductory videos online. We will also provide additional information and video material on our website. Videos are simply much better than the written word to explain this skill, therefore, we will not include a lengthy text explanation about soldering technique at this point. Just be very careful with the soldering iron: it is very hot and can easily burn or melt objects around it. Thus, we advise you always work on a clear working space!

3.2.3 Power Supply

There are multiple possibilities to provide power to your exoskeleton.

Attention!

Do not use unprotected grid power supply! Exoskeletons are wearable devices that are attached directly to humans, and should **never** be connected to the electrical grid while being worn. Although the interfaces are insulating against electric current, always use a power supply that is protected or limited (battery). Stay safe!

For the content of this handbook, we use the power outlet of the Arduino board to supply the motor and the force sensor. So, you only need one power supply, to power the Arduino board. As the EduExo is a wearable device, a safe power supply is required because you cannot simply run away from the exoskeleton in case something goes wrong, like you could if, for example, your microwave were malfunctioning. A safe power supply is usually limited in voltage and current. You should never use unprotected grid power; rather, use batteries as they are intrinsically limited in their output. You can also power the Arduino by a USB cable from you laptop when it is running on its batteries – and **not when the laptop is plugged into the wall**. There are also certified power supply units that are used in medical devices, but they are rather expensive.

3.3 Tutorial Motor

Attention!

Please be sure that you disconnected any power supply as well as the USB connection from your computer to the Arduino while wiring the components. All those loose wires can easily create a short circuit that could harm you, your computer, the EduExo components, or your power supply.

3.3.1 Connecting the Servomotor

Hint

It is important to correctly wire the exoskeleton; otherwise, you might end up with an error that is really hard to find.

First, we will connect the motor to the Arduino to read its angle and to control its position. In total, four wires have to be connected: two for the power supply, one for the angle sensor and one for the position control (Figure 3.9).

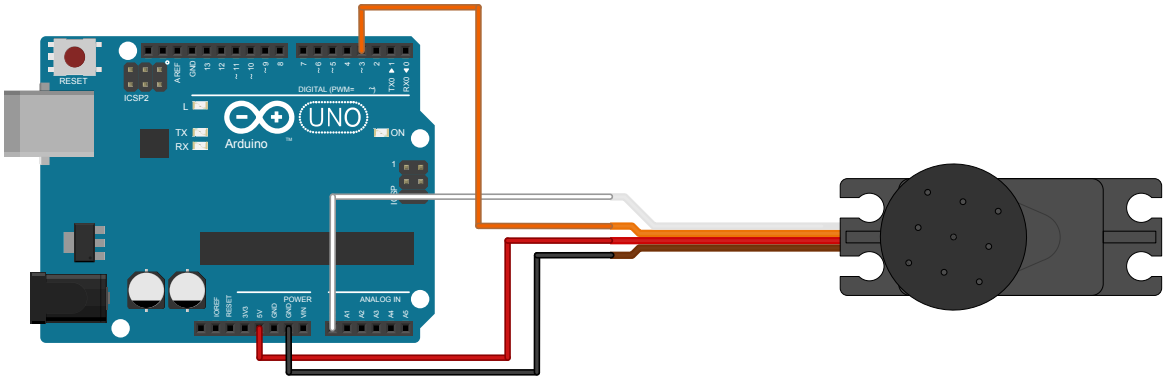

Figure 3.9: The schematics of how to connect the servomotor to the microcontroller. This will enable you to read the servomotor angle (white wire) through one of the Arduino's analog inputs, and to control the servomotor angle (orange wire) through one of the Arduino's PWM outputs. The power supply is connected to the Arduino's power output (red wire to 5V and brown wire to GND)

For a first test, simply connect the servomotor using the jumper wires included in the kit (Figure 3.10). For the motor power supply, we connect the red wire of the motor to the 5V power outlet of the Arduino, and the brown wire to one of the two ground outlets. Then, we connect the white wire of the potentiometer angle sensor to one of the analog input pins of the Arduino (A0-A5). As the angle signal is an analog voltage, the analog input can be used to read the signal and use it in the control system later. Last, we connect the control signal wire to one of the Arduino's PWM outputs (3, 5,6). The PWM-capable output pins are marked with a ~ symbol.

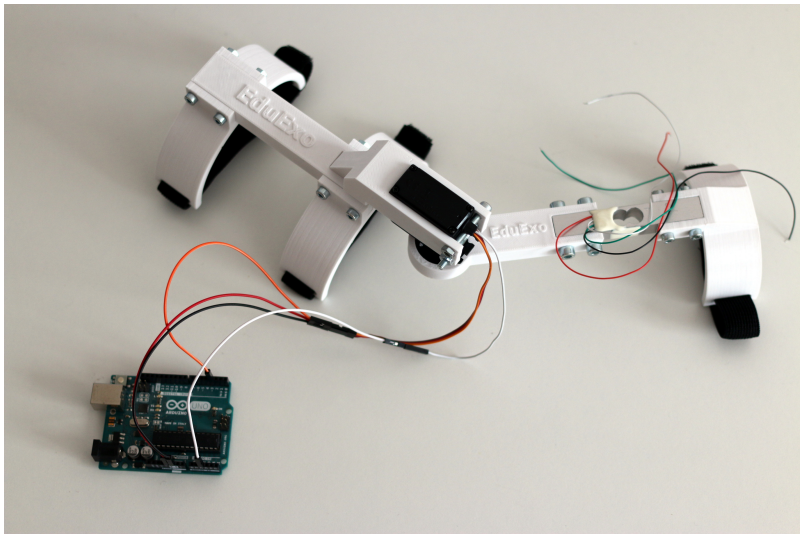

Figure 3.10: The servomotor-Arduino connection implemented with the jumper cables included in the kit.

3.3.2 Cable extension

As mentioned in the previous section, while the cabling with the 4 jumper cables is very convenient, jumper cables are not very suitable to span large distances and are also not very robust connections. Therefore, we included cables with four wires that you can use to extend the connection to your Arduino. This step requires soldering (Figure 3.11), so if you prefer to start programming now or still need to get soldering equipment, you can come back to this later.

(a) (b)

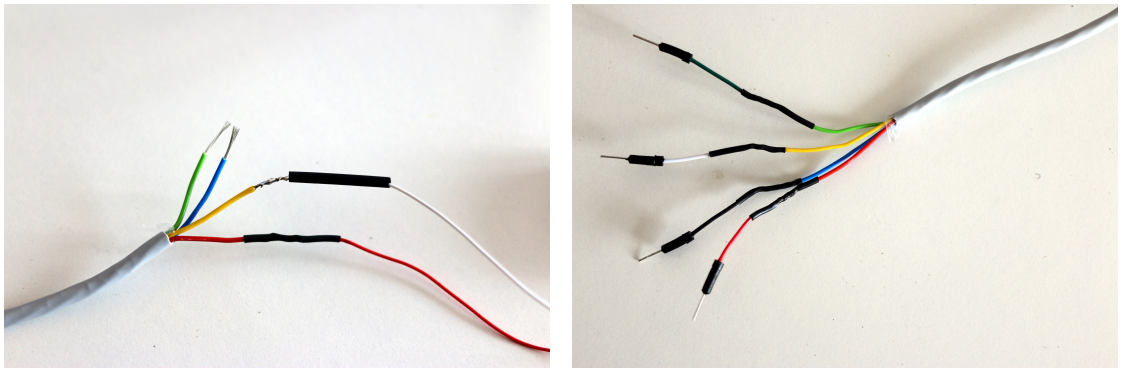

Figure 3.11: How to extend the jumper cables: cut them in the middle, remove insulation, and put the extension cable with the four wires in between the two halves.

To extend the cables, you cut the jumper cables in the middle and put one of the four wire cables in between the two halves of the jumper cables. For that, you have to cut the wires, remove insulation, solder the wires together and add the heat shrink tubing. You can find a more detailed explanation in section 3.4.1, where you have to execute similar steps to connect the force sensor.

3.3.3 Reading the Motor Angle

Now, it is time to write our first program and implement the first functionality in the EduExo. An Arduino program is usually called a 'sketch'. Sketches contain one or several source code files that can be compiled, uploaded and run on the Arduino. Our first program (Listing 3.1) will read the position from the analog feedback sensor to measure the exoskeleton's elbow joint angle.

```
1  int servoAnalogInPin = A0;
2  int posIs;
3
4  void setup(){
5      Serial.begin(9600);
6      delay(1000);
7  }
8
9  void loop(){
10     posIs = analogRead(servoAnalogInPin);
11     Serial.print("Position:");
12     Serial.println(posIs);
13     delay(10);
14 }
```

Listing 3.1: Arduino program to read the position sensor.

As this is our first program, let's walk through it step by step. If you are completely new to Arduino, we strongly advise you to read the 'getting started' section that you find in the Arduino IDE in the menu bar under 'Help'. Although we explain the main steps below, the IDE introduction provides more information and more details than we can put in this handbook.

First, you have to open the Arduino IDE and create a new file. Copy the code in Listing 3.1 and save it. Note that the Arduino IDE creates a folder with the same name as the file.

Hint

Arduino sketches have to be stored in a folder with the same name as the main file. If you have the .io file in a different folder and want to open it with the Arduino IDE, you will receive an error message.

Before we compile the sketch and load it on the microcontroller, let's take a look at the code. First, you can see that the entire program consists of three main sections.

The first section (lines 1 and 2) are used to define our variables. Each variable has a type, a name and a value (which you can assign later). We have two integer (whole number) variables. The first integer `servoAnalogInPin` is used to store the number of the analog input port (A0) to which our position signal is connected. If you connected your angle sensor to a different analog input port, change the value of `servoAnalogInPin`. The second integer `posIs` is the variable that will store the sensor data. No value is assigned yet as we do not know the initial position. You can change the name of the

variables as you wish, but use something that is easy to understand, especially for other people looking at your code. If you are interested, there are standards to ensure code readability, which you can read more about online.

The second section (lines 4-7) is the `setup()` function. Generally, a function is a named piece of code that can be used from elsewhere in a sketch. There are many predefined functions in Arduino that we can use. We can access many additional functions by including libraries, and we can define our own libraries. The `setup()` and the following `loop()` functions are special, as they are part of every Arduino sketch. You need to include both functions in your sketch, even if you do not need them for anything. The `setup()` function is called once, when the sketch starts. It is a good place to do setup tasks like setting pin modes or initializing libraries.In our case, we initialize serial communication that we will use later to send the motor position to our PC (line 5). Note that this is not a hardware serial connection with a serial cable like back in the day; it is a software serial interface that uses the USB cable. The `delay()` function (line 6) makes the Arduino wait for the specified number of milliseconds before continuing on to the next line. There are 1000 milliseconds in a second, so the function in line 6 causes a one second delay. We added it here to give the serial communication enough time to be established.

The third section (lines 9-14) of our sketch is the `loop()` function that contains the code of our main program loop. It is called over and over until the program is stopped, and is the heart of most sketches. Here, we implement the functionality of our exoskeleton. In line 10, we use a function once again; this time, to acquire the value (voltage)of the signal in the analog input pin that we specified earlier, and store this value in the variable `posIs`. Because this line is within the main loop, this code is executed every cycle. So, if the angle of the exoskeleton changes, the variable `posIs` is updated in the next loop. Lines 11 and 12 are are used to send this value to the serial port, so it can be displayed on the computer screen. This will let us monitor the position values output by the servomotor while the sketch is running. Line 11 sends and displays the string '`Position:`'.Line 12 sends and displays the most recent position value stored in the variable `posIs`. Note the difference between the `Serial.print()` and `Serial.println()` functions: `Serial.println()` automatically ads a line break after printing the variable's value. You will see this difference later, when we look at the output. As any code requires some time to execute, the code together with intentionally implemented delays (e.g., with the `delay()` function) define how much time each cycle of the loop requires. This will be of special interest later on when we implement the control system, as a high control frequency (many loops per second) requires fast execution of the code.

When you have written all that code, it is time to compile it and load it onto the Arduino. First, you have to connect the Arduino to you computer with a USB cable. This also powers the Arduino, so make sure that all cables are properly connected. Next, it may be necessary to install some drivers if you never used an Arduino on your computer. This should happen automatically and will enable you to establish communication between the Arduino and your computer.

With your Arduino connected to your computer, you can setup your IDE to the correct settings to program the Arduino Uno. First, you have to tell the program which board you

have: on the menu bar, go to Tools → Board and select the Arduino Uno. Then, you have to tell the program where the board is connected: go to Tools → Ports and then select the COM port to which the Arduino is connected (the port with the Arduino UNO should be highlighted in the port list).

Now, the settings should be correct and you can compile and upload your sketch using the upload icon located on the upper left corner, below the menu bar in your IDE. The Arduino IDE now will automatically compile your source code into machine code (zeros and ones) and then upload it onto the Arduino. If everything works out, you will receive a message 'Done uploading', together with some information about how much of the Arduino Uno resources are used by your program. Again, if you run into any troubles connecting the Arduino, compiling, or uploading the program, you can find a lot of help online starting with your favorite search engine.

When successfully uploaded, the program will start immediately, and your EduExo is now measuring the motor angle and sending it to the serial interface many times per second. To take a look at the data output, you can open the 'Serial Monitor' in the Arduino IDE (Tools → Serial Monitor). When opened, you should see the serial output writing the exoskeleton's position very quickly one line after the other. Now, pick up the exoskeleton, move the joint and see how the value changes. The value will be somewhere between 0 and 1023. The reason for that range is that the analog-to-digital (A/D) converter of the Arduino that converts the analog voltage at the input pins into a digital signal that can be used by the microcontroller, has a range of 10 bits. This means it can discretize the full range of the analog input into $2^{10} = 1024$ values.

3.3.4 Calibrating the Servomotor's Angle Sensor

Because 10 bit values are not really intuitive and can be confusing later on, we will calibrate the position sensor. This will provide us with an angle value (in degrees) instead of a value between 0 and 1023. For the calibration, we will move the exoskeleton joint to two known positions, read out the potentiometer voltage value, and come up with the math that relates the sensor's digitized value and the joint angle. The easy two positions are 0° and 90° (Figure 3.12). You can simply move the joint to these positions by hand, this is accurate enough for us.

Open the serial monitor (tools → Serial Monitor) and write down the sensor value it shows for both positions. Let's call them a_0 and a_{90}. We can use simple linear interpolation to come up with the equation that expresses the joint angle φ_{elbow} (in degrees) as a function of the digitized sensor value (Equation 3.2).

$$\varphi_{elbow} = \frac{90° - 0°}{a_{90} - a_0} \cdot (pos_{is} - a_0) \tag{3.2}$$

In our testing, the parameters were $a_0 = 80$ and $a_{90} = 255$. Now, we can include this calibration in our program to return the angle in degrees (Listing 3.2).

First, you can see that we added a new variable (line 3) `posIsDeg` of data type float. Float variables have decimal places and are, therefore, more precise than integers. The

(a) (b)

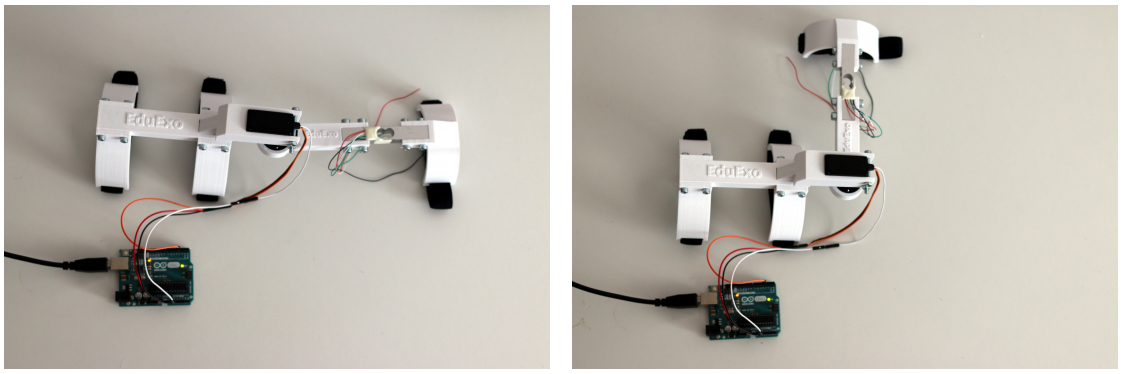

Figure 3.12: Calibration of the servomotor angle sensor. Move the exoskeleton to 0° and 90°. Read out the signal from the angular sensor at each position using the Serial Monitor of the Arduino IDE.

```
1  int servoAnalogInPin = A0;
2  int posIs;
3  float posIsDeg;
4
5  void setup(){
6     Serial.begin(9600);
7     delay(1000);
8  }
9
10 void loop(){
11    posIs = analogRead(servoAnalogInPin);
12    posIsDeg = (90.0/(255.0-80.0))*(posIs-80);
13    Serial.print("Position (in degree):");
14    Serial.println(posIsDeg);
15    delay(10);
16 }
```

Listing 3.2: Arduino program to read and calibrate the position sensor.

drawback is that they require more dynamic memory (RAM) and have a slower computation speed.

We also added code that includes the linear interpolation (line 12) in the program loop. We directly added Equation 3.2 to make it easier to follow. For example, it is clear that the value of the denominator is the difference between two specific numbers (255 and 80), instead of wondering where the value (175) comes from. This is a way to improve readability of the code. Compare, for example, with the numerator - it is not clear whether 90.0 came from 90-0, or 150-60. Knowing this can help understand the thought process behind the code, which can be very helping when debugging or modifying the code.

Of course, it could make sense to actually solve the equation to a degree, to reduce computation time, since the math has to be solved again in each cycle. You might also wonder about why we use the decimal points in the equation. It is because we have to avoid a problem that is linked to the use of integer variables. Without that point, Arduino considers these values integers, with the point they are considered floats. If you divide two integers, everything after the decimal point is lost because integers do not have decimal points; so, 9 / 2 = 4 and 7 / 9 = 0. You can try it without the decimal points to see what happens. You can also change the type of the variable `posIsDeg` from float to integer and see how it affects the output and the memory requirements. You can see this information in the Arduino IDE after successful compilation.

3.3.5 Controlling the Motor Angle

Now that we are able to read the position sensor data, our next step will be to control the movements of the motor. As described above, the servo motor that we use has its own internal position controller. Therefore, controlling its angle from the Arduino is quite simple. We have to send a desired angle from the Arduino to the motor and it will take care of the rest. To send the desired angle, we have to encode it into a signal using pulse-width modulation (PWM). Thankfully, there is an Arduino library that provides us with the PWM functionality (Listing 3.3).

```
1  #include <Servo.h>
2
3  Servo myservo;
4
5  int positionDesired = 40;
6
7  void setup() {
8    Serial.begin(9600);
9    myservo.attach(3);
10 }
11
12 void loop() {
13   myservo.write(positionDesired);
14 }
```

Listing 3.3: Arduino program to set a motor position.

Create a new sketch and write the program as listed. Be careful when you compile and upload the code, as the exoskeleton will move immediately. In this program, you fill find several new concepts. First (line 1), we include the Arduino servo library that allows an Arduino board to control servomotors as the one we use in the EduExo. Then, we create a Servo object called `myservo` (line 3). From now on, this object will be used to address our servomotor in the Arduino program. Next, we define an integer variable (line 5) to store the desired motor position.In the `setup()` function (line 9), we use a function from the servo library to define which pin the servo is attached to (one of the PWM pins; in the example, we use number 3). The last new command is the servo `write()` function (line 13), which sends the desired position in degrees to the servomotor.

When the sketch is uploaded, the motor will move to the desired position and stay there. You can now put the exoskeleton on your arm (this works much better if you have already added the extension cable). You will see that you can easily move away from the desired position, because the motor is quite weak as it is of limited power. Nevertheless, the exoskeleton will support you moving towards the desired position and hinder you moving away from it.

Hint

It may happen that the movement range of the servomotor and the exoskeleton joint are somehow out of the range that you require at this point. For example, the command to move to position '0' can be in the middle of the range you need. You can easily adjust this by removing the lower arm segment from the servo. This can be done by loosening the black screw of the servo mount. Rotate the lower arm segment a little bit and fixate it again as you need. If you did that, you have to repeat the calibration we did above to be able to read the angle values in degrees.

3.3.6 Mapping Sensor Position and Motor Position

You might have already noticed that the motor position does not necessarily exactly represent the value of the angle sensor. This is because the motor uses for its own internal coordinate system in the movement control, which might not correspond with the coordinate system of the exoskeleton. Later, we will need to record movements and replay them. For that, we have to know which angle sensor value corresponds to what motor position.

We will send two position commands to the servo, let it move there, and then record the angle sensor value (Listing 3.4). This is very similar to our manual calibration before, where we mapped the sensor signal to an angle value. But, instead of moving the exoskeleton to different angles manually, we send a motor command to move it.

Create the sketch, upload the program and open the serial monitor. The program will display four values: the first two are the position command values we defined to send to the servo motor, the other two are the sensor values we read at those two motor positions. As the program is repeated several times, you will find that the sensor values may change a little bit each time. This can happen due to inaccuracies in the servo control, the angle


```
#include <Servo.h>

Servo myservo;

int servoAnalogInPin = A0;
int posServo1 = 0;
int posServo2 = 100;
int posSensor1;
int posSensor2;

void setup(){
    Serial.begin(9600);
    myservo.attach(3);
    delay(1000);
}

void loop(){
    myservo.write(posServo1);
    delay(2000);
    posSensor1 = analogRead(servoAnalogInPin);
    myservo.write(posServo2);
    delay(2000);
    posSensor2 = analogRead(servoAnalogInPin);
    Serial.println(posServo1);
    Serial.println(posServo2);
    Serial.println(posSensor1);
    Serial.println(posSensor2);
}
```

Listing 3.4: Program to extract two value pairs to map angle sensor signal to the corresponding servo position.

sensor or noise in the signals. For each variable, simply choose one value or calculate the mean over several measurements (samples). Write the four values down. With these two value pairs, we will correlate angle sensor position and servo position and create a function to map them. We could do this again as we did with our first calibration, by writing down the math and programming it into the source code. A simpler way to do so is the `map()` function (Listing 3.5).

```
posIs = analogRead(servoAnalogInPin);
int posIsServo = map(posIs, posSensor1, posSensor2, posServo1,
    ↪ posServo2);
```

Listing 3.5: The map function to calculate a servo position that corresponds to a measured angle sensor position.

This function will map any angle sensor value `posIs` to the corresponding servo control value. In other words, if we want to return to any given position `posIs`, this function

provides us with the servo position value that we have to send to the servo. The function has five parameters. First, the sensor value that should be mapped to another range. The next two are the ranges of the input signal: these are the lower and upper sensor values that we measured. The last two parameters are the range of the output signal that is returned by this function; these are our lower and upper servo positions. You can fill in the four values you wrote down before, and we will use this function again later on.

3.4 Tutorial Force Sensor

3.4.1 Connecting the Force Sensor

The force sensor is a standard strain gauge-based design. As previously mentioned, the force sensor is a simple design that enables to measure the interaction force between exoskeleton and human. Because the signal of the force sensor is very small, an amplifier chip is used so the Arduino microcontroller can read it. Therefore, we will connect the force sensor to the amplifier, and then the amplifier to one of the Arduino's analog inputs. This requires that you solder the force sensor wires to a 4 wire cable and jumper cable ends to be able to connect them to the breadboard. The preparation includes several steps (Figure 3.13).

First, you have to prepare the cables (Figure 3.13(a)). Remove the coating from the cables so you can see the individual wires. Then, remove the insulation from the end of all wires (about 1 cm). Both steps can be done with a wire stripper or carefully with a cutter or similar. Just be careful not to cut into the copper wires inside. If that happens, simply shorten the cable end and try again. Next, put some heat shrink tubing on each of the four force sensor cables, as shown in figure 3.13(a). Push the heat shrink tubes as far away from the soldering spot as possible, otherwise the heat of the soldering iron might cause them to shrink in the wrong place. Then, twist the copper wires and apply the solder. When the solder has solidified, move the heat shrink tubes over the soldered connection (Figure 3.13(b)) and heat it up using, e.g., a lighter or a hair dryer. Be careful not to melt the insulation around the rest of the cables! Repeat that for all eight connections (4 from the force sensor, 4 with the jumper cables). When you finish, you will have a long cable connecting each of the four wires of the force sensor to half of a jumper cable. With this setup, you can easily plug the wires into the breadboard.

Next, we will connect force sensor, amplifier and Arduino on the breadboard using jumper cables. A breadboard is a tool that you can use to quickly create circuits without soldering (Figure 3.14).

On the long sides of the breadboard, you find two bus strips. They are normally used to provide power to the electronic components. Perpendicular to and between the bus strips, we have shorter terminal strips that are separated by a notch in the center. The notch is a standard size that allows easy placement of chips (like our amplifier) such that each pin of the chip is connected to a different terminal strip. All the pins in a strip are electrically connected. This enables us to connect different components by simply plugging them into the same strip.

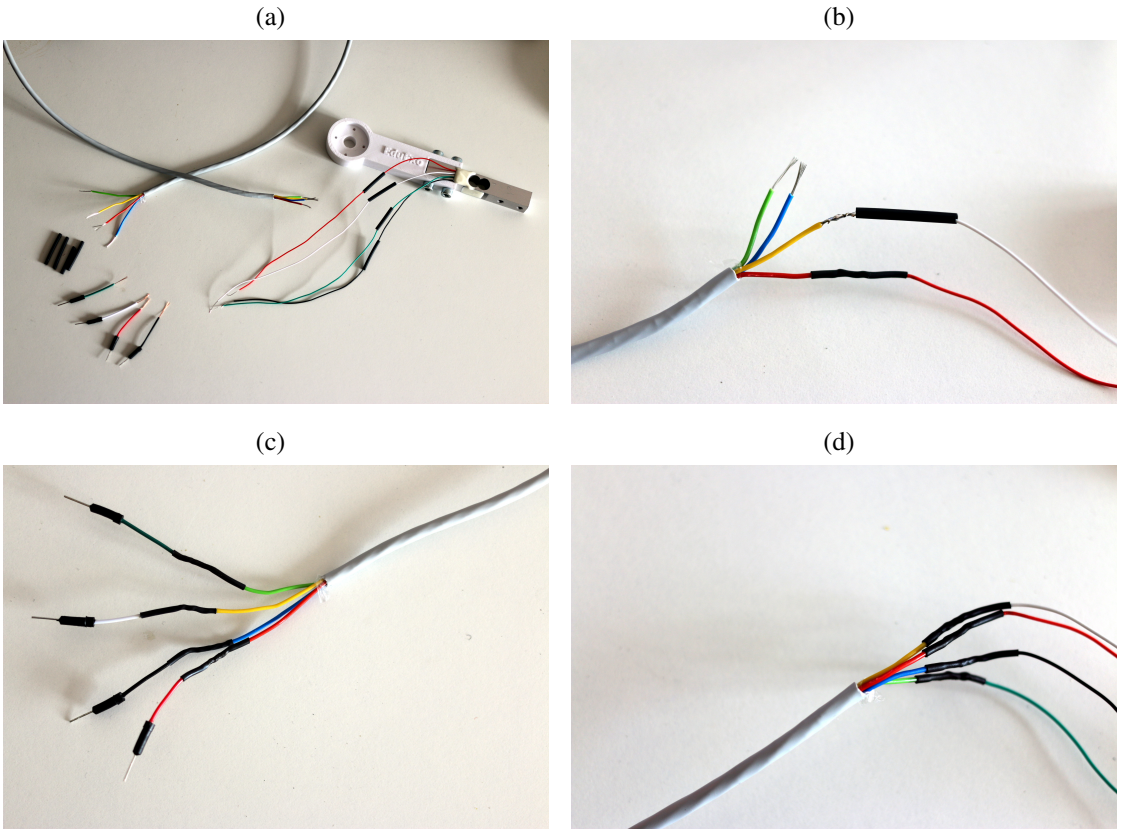

Figure 3.13: Preparation of the force sensor to connect it to the amplifier on a breadboard: (a) Cut the cables and wires and remove insulation; (b) solder the wires together, don't forget to put the heat shrink tubing over the cable before soldering! When soldered, push the heat shrink tubing over the connection and heat it up (e.g., with a lighter); (c) what the finalized end for the breadboard should look like; (d) the finalized connection between the force sensor wires and the cable.

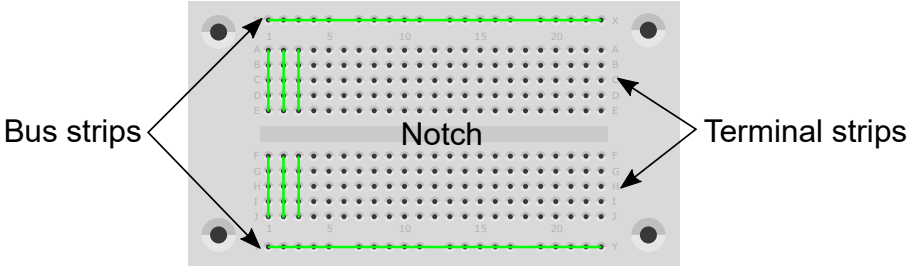

Figure 3.14: A breadboard with the internal electrical connections highlighted.

Let's start implementing our amplifier circuit by taking a look at the schematics (Figure 3.15). On the left side, you find the Wheatstone bridge that is in the force sensor. In the center is the amplifier chip with its 16 pins. Note the small mark (half-circle) on the top of this chip: this will help you identify the pins and correct chip orientation later. On the right side, you find the analog input pins and the power output of the Arduino.

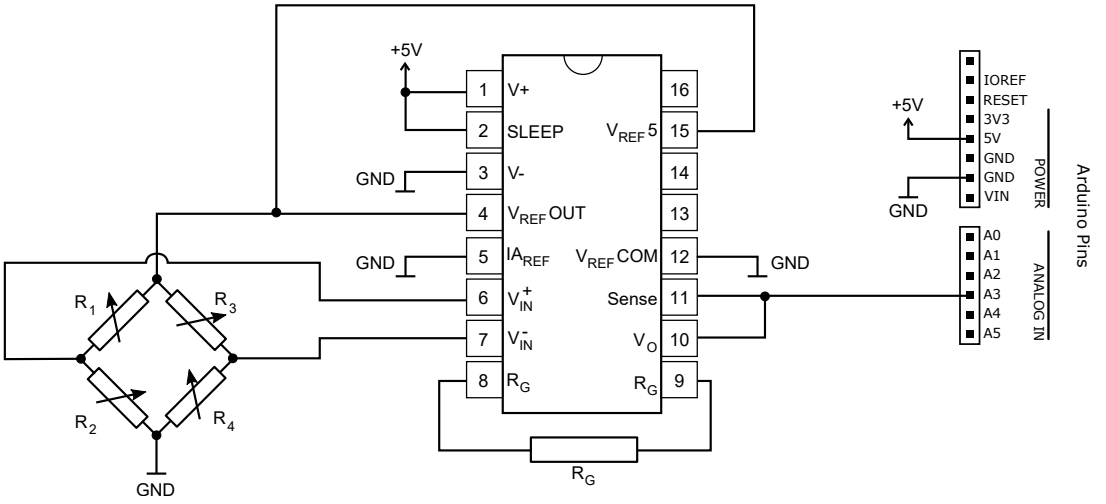

Figure 3.15: Schematics of connections between force sensor, amplifier chip and Arduino to enable the acquisition of the force signal.

At the Arduino output, you find the two symbols for the power supply (+5V and GND). You find these symbols again at different pins of the chip and at the Wheatstone bridge. Pins with the same symbols are connected together. The other pins are connected as indicated by the lines between them. When crossing lines are connected with a black dot, it indicates an electrical connection. When two lines cross but one of the crossing lines has a half-circle, as if it were passing over the other line, it means they are not connected. You can see this right above the Wheatstone bridge.

Attention!

When connecting multiple pins together (i.e., to the same potential), avoid making connection loops. Loops are especially bad if they happen on the electrical ground (GND); you can find a lot of information about ground loops, and why they should be avoided, online. A simple solution is to always use a star configuration, that is, always connect the grounded pins directly to the ground bus. For example, in our circuit, pins 3, 5 and 12 of the amplifier chip should all be connected to the ground: you should connect 3 $\rightarrow$ GND, 5 $\rightarrow$ GND, and 12 $\rightarrow$ GND (making a star with GND in the middle) and NOT 3 $\rightarrow$ 5 $\rightarrow$ 12 $\rightarrow$ GND $\rightarrow$ 3 (which makes a loop connecting all the pins). If you always follow the star configuration, you will not make loops in your circuit!

Lets take a detailed look at the circuit. Pins 1 and 2 of the amplifier are connected to the supply voltage. Pin 1 is the chip's power supply, pin 2 activates the chip if set to 'high',

meaning 5V. We can directly connect it to the supply voltage, this will automatically activate the chip once the Arduino is switched on. Pin 3 is the ground of the amplifier's power supply. Pin 4 is the supply voltage for the force sensor. It is connected to pin 15 to set the reference voltage to 5V. This provides an accurate voltage source for our Wheatstone bridge on the force sensor. The other wire of the Wheatstone supply (bottom) is grounded. Pin 5 of the amplifier is the instrumentation amplifier reference (IA_{REF}) that we set to ground. Pins 6 and 7 are connected to the force sensor wires that carry the output signal of the Wheatstone bridge. Between pins 8 and 9, an external resistor R_G is connected.

The resistance value of R_G is used to set the gain of the amplifier. The resistor provided in the EduExo kit has resistance of 60.4 Ohm, which sets the amplifier's gain to around 1000. A higher gain (smaller resistor) will enable you to measure smaller signals, but it will also amplify the noise. Pins 10 and 11 are connected and provide the amplified output signal. This signal is then connected to one of the analog input pins of the Arduino. The reference voltage pin 12 is connected to ground. The remaining pins (13, 14, and 16) should stay open, that is, should not be connected to anything. You can find additional information about the amplifier in its data sheet (simply search for INA125 amplifier online). Next, we will implement this circuit with the help of the breadboard (Figure 3.16).

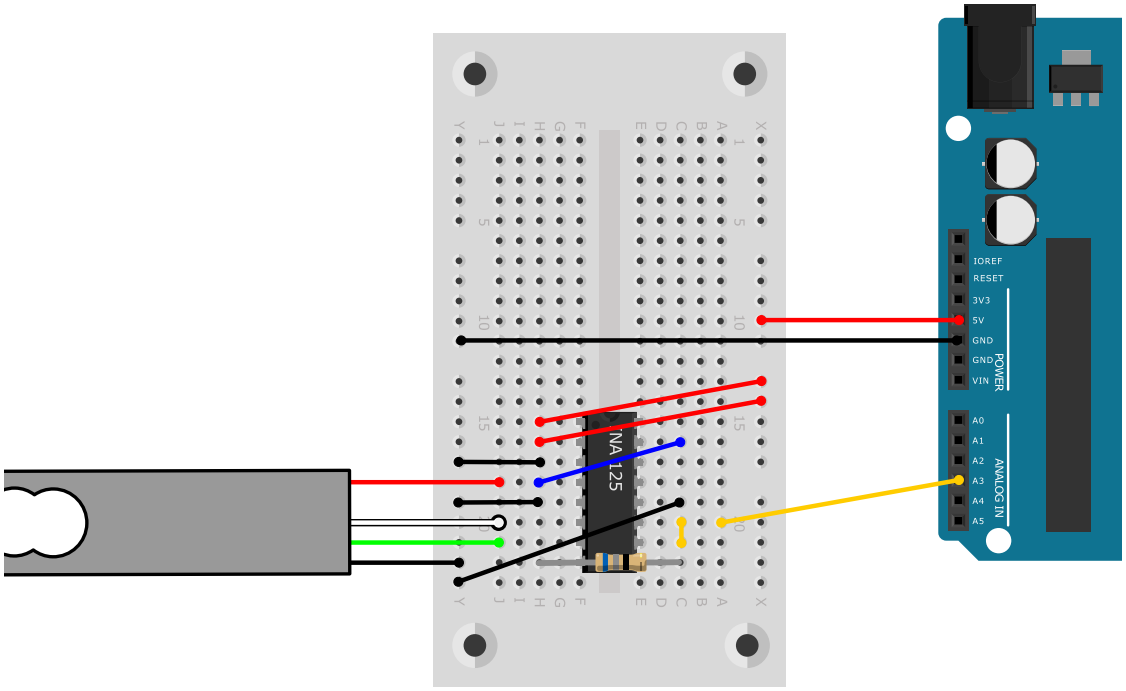

Figure 3.16: Illustration of how the circuit can look on the breadboard.

This might look a little confusing at first. But, simply try to follow the schematics and diagram, one connection at a time. Also, try to color-code the signals, e.g., GND is always black and 5V supply voltage is always red. This will help you not get lost in all the wires!

3.4.2 Reading Force Sensor Data

With the force sensor connected, we can now implement the program which will read the force sensor and display the measured value on the screen using the serial monitor. Reading the force data is similar to reading the motor angle. The amplified signal is connected to an analog input pin, which can be read by the following code (Listing 3.6).

```
1  int forceAnalogInPin = A3;
2  int forceIs;
3
4  void setup(){
5    Serial.begin(9600);
6    delay(1000);
7  }
8
9  void loop(){
10     forceIs = analogRead(forceAnalogInPin);
11     Serial.print("Force: ");
12     Serial.println(forceIs);
13     delay(10);
14 }
```

Listing 3.6: Arduino program to read force sensor.

3.4.3 Calibrating the Force Sensor

Similar to the angle sensor, it would be nice if the values that we measure were more intuitive and correspond to the actual load, not just represented by a 10 bit value. To achieve that, first we will offset the force value (Listing 3.7).

```
1  int forceAnalogInPin = A3;
2  int forceIs;
3  const int forceOffset = 439;
4
5  void setup(){
6    Serial.begin(9600);
7    delay(1000);
8  }
9
10 void loop(){
11     forceIs = analogRead(forceAnalogInPin);
12     forceIs -= forceOffset;
13     Serial.print("Force: ");
14     Serial.println(forceIs);
15     delay(10);
16 }
```

Listing 3.7: Force sensor program with the offset value subtracted from the force signal.

If you take a look at the force signal in the serial monitor, you will see that you only get positive values. So, you don't even know the direction of the force (towards flexion or extension) as it is a value without a sign. To change that, put the force sensor on the table without any force applied and read the value using the serial monitor. This value equals no force.

The first change in our sketch is to store this offset value in a constant integer value (line 3). Variables defined by the keyword `const`, cannot be changed during runtime. This makes sense here, as this offset value will not change. We can subtract this value from the input signal (line 12). The operator we use (`-=`) is just a shorter version of (`forceIs = forceIs - forceOffset`) to save space. It changes the value of the integer variable by subtracting whatever is on the right side of the operator.

In the result, you will see that the force signal changes signs when you apply forces in opposite directions. If you have a scale at hand or some known weights, you can also calibrate the force sensor to calculate the actual force (e.g. in Newton) out of the force signal. This step is very similar to the position sensor calibration (Chapter 3.3.4). Apply a known force, e.g., with a known weight, read out the sensor signal, and come up with the math that correlates force and digitized sensor signal. Just be careful to not put too much weight, as you might break the EduExo!

Chapter 4

Control Systems

Summary

In this chapter, we will develop and implement different control systems that define the exoskeleton's behavior. To do so, we will define a set of rules of how the robot should act and react when worn by the human, pack those rules into some equations, and implement them on our Arduino microcontroller.

4.1 Background Control Systems

4.1.1 Introduction

Control systems are generally a set of rules and algorithms that define the behavior of technical systems. They can be found in many technical systems to control many kinds of variables and keep it to a desired value. They control the fast and precise movements of the robot that assembled your car. They come along in the form of the autopilot that flies you to the Caribbean. And, they keep your self-driving car aligned in the streets at a constant speed. In our case, the control system is responsible for the amount and timing of support provided to the user by the exoskeleton.

In order for a control system to work, sensors and actuators are needed. Sensors provide information about the state of the system (for example, the pose and movement velocity of the robotic exoskeleton), and actuators can be used to react to the measured state following a predefined rule, the control algorithm. Typically, control systems are represented by a control diagram (Figure 4.1).

On the left is the desired value of a variable or system state, such as position, force, or velocity. This is compared to the values measured by a sensor (bottom of diagram, feedback loop), and the difference is calculated. This difference is then sent to the controller that adjusts, for example, the motor current, opens and closes a valve, or executes an action to 'push' the system (called 'plant') towards the desired value. After this input goes through the plant, the sensors measure again if the desired value is achieved, and the

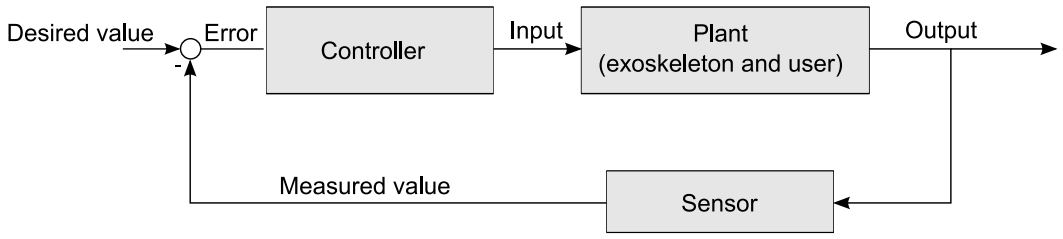

Figure 4.1: Control diagram of a generic control loop.

controller reacts accordingly. This is repeated over and over again to put and keep the system at the desired value. The desired value can change over time, and the job of the controller is to make the system follow these changes.

Often, you can distinguish between high-level and low-level control systems. High-level controllers (sometimes called control strategy) are a set of rules that define, for example, in which situation the exoskeleton provides assistive force or executes a guiding movement. The high-level control can include a user intention detection to ensure that the exoskeleton support suits the user's momentary needs. The low-level control system is responsible for the individual joints and actuators providing that desired support.

It is not uncommon in robotic exoskeletons that the user or a second person remains at least partially in control of the high-level control system. For example, by controlling the exoskeleton manually by pressing control buttons. However, there are several approaches for the exoskeleton to autonomously detect the user's movement intentions and support needs, for example, by measuring the user-exoskeleton interaction forces, or the user's muscle activities. The two following examples will help illustrate how the behavior of a robotic exoskeleton is defined by its control system.

4.1.2 Example: Control of a Gait Restoration Exoskeleton

To control an exoskeleton for gait restoration in users with paraplegia, manual control inputs combined with a position controller are used. As the user's legs are completely paralyzed, the exoskeleton has to provide all the power required to move them.

In this case, the high-level control is implemented as manual control by the user. Buttons located in the handles of crutches (that are required anyway for balancing) are used to select and initiate movements manually. Different types of movements are prerecorded and stored in the exoskeleton's control systems. These recordings provide the desired angle trajectories for each of the actuated exoskeleton joints. When a movement is triggered by the user, the low-level joint position controllers execute the recorded joint angles, replaying the movement. Then, the exoskeleton automatically continues to execute consecutive steps until stopped by the user, or each step can be triggered manually by the user, who then remains in command.

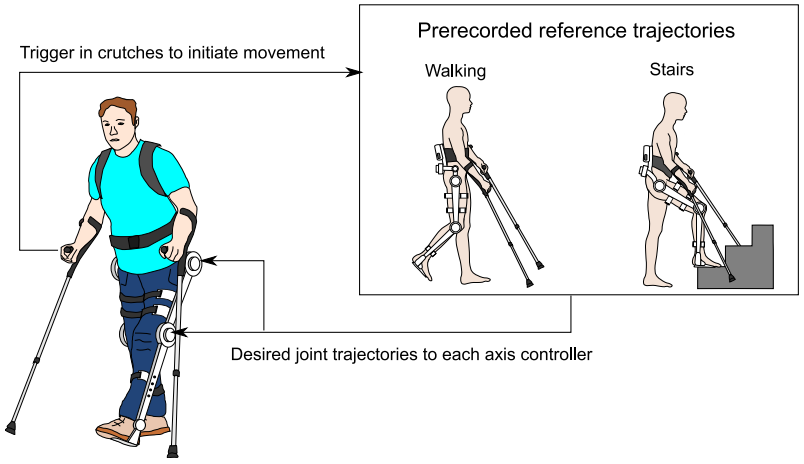

Figure 4.2: Control of a gait restoration exoskeleton.

4.1.3 Example: Control of a Work Assist Exoskeleton

The second example is an upper-body exoskeleton that supports the user when lifting heavy objects (Figure 4.3). The device is connected to the user's back and hip like a backpack, and relieves the user's arms when lifting and carrying a payload.

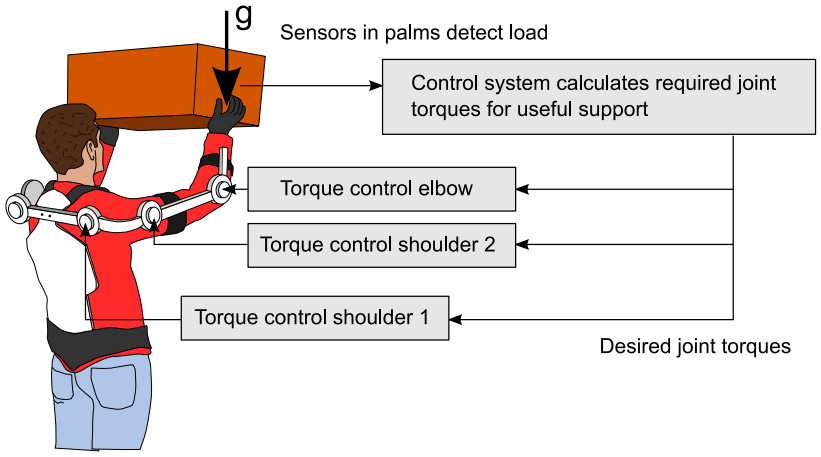

Figure 4.3: Control of a lifting support exoskeleton.

To detect when the user needs support, sensorized gloves are used in the high-level control. A pressure sensor in each palm measures when the exoskeleton user is carrying a heavy object, and only then a desired vertical assistive force is activated. The high-level controller calculates which joint torques are necessary in each arm joint to provide the desired support. These desired joint torques are then sent to the low-level torque controllers of the joints, and they take it from there.

As in every control system, this cycle is then repeated to reevaluate if the situation has changed. So, again: measure the force, decide what support is necessary and update the input to the low-level controllers accordingly. This is usually done very quickly and repeated many times every second to ensure that the exoskeleton will react quickly to

changing user needs. Otherwise, you can end up with a slow device that provides inadequate support. How often such a cycle is repeated is defined by the control frequency, which is usually somewhere between a few hundred or a few thousand times per second (Hertz, Hz). Typically, the low-level control is faster (higher frequency) than the high-level control, to give the joint controllers time to execute a command before a new command is received.

4.2 Common Control Approaches

As part of this brief introduction into the very large world of control systems, let's take a look at common low-level control approaches used for exoskeletons and other haptic devices. They mostly differ in which variable or physical value is controlled. For closed-loop control, we can only use variables that we can influence and also measure.

4.2.1 Position Control

A position controller controls a position, in our case, the joint angle of the exoskeleton. The controller focuses on only one aspect, to move the exoskeleton joint to a desired angle. It will change the motor's torque until this desired angle is reached. A control diagram for a position controller is shown in Figure 4.4.

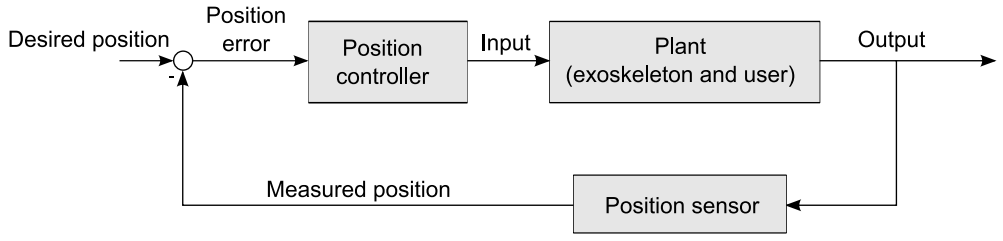

Figure 4.4: Control diagram of a position control system.

This kind of controller is implemented in our servomotor, and takes care of the low-level joint position control. This is why we can simply send a desired position to the servomotor; the onboard controller makes sure the motor moves to the commanded position. Position control systems are very useful to support rigidly guided movements, or to limit the range of motion of a user.

4.2.2 Force/Torque Control

A force (torque) controller directly controls a force (torque). Torque controllers try to keep a predefined (desired) torque without regards to the angle (Figure 4.5). Thus, in our case, if the user applies a torque that is higher than the desired joint torque, the exoskeleton will move.

A closed-loop torque controller requires force or torque sensors. It also requires the ability to directly change the force applied by the plant, e.g., by increasing or decreasing

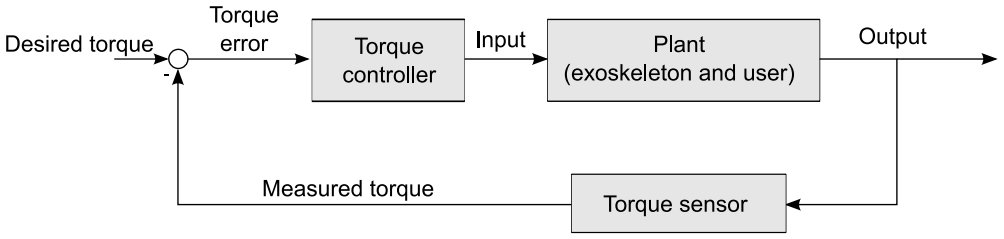

Figure 4.5: Control diagram of a torque control system.

motor current. Force and torque control systems can be very useful to provide support, e.g., an upward force to assist lifting, without restricting the movements of the user.

4.2.3 Impedance and Admittance Control

Impedance and admittance controllers are somewhere in between position and force controllers. They are often used when a robot interacts with other objects; in our case, a human user inside the robot. They usually define a relation between a desired position and a force (impedance), or between a desired force and a position (admittance). In an impedance controller, movement (position) is measured and force is controlled. In an admittance controller, force is measured and the movement is controlled accordingly (Figure 4.6. In the EduExo, we have a force sensor to measure the interaction force, and the low-level servo controller is a position controller. Thus, we will use admittance control in our exoskeleton.

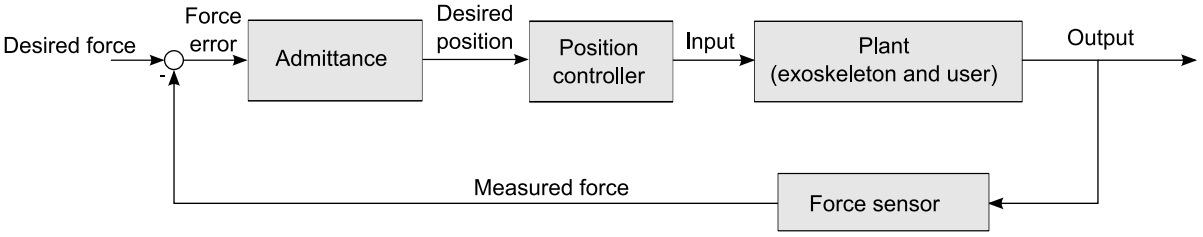

Figure 4.6: Control diagram of an admittance control system.

In such an admittance controller, the mathematical rule that defines the relation between force and position is called the admittance or virtual admittance. This rule often imitates the behavior of a mechanical system (spring, damper, mass). As a simple example, you can think of a controller simulating a mechanical spring. When no force is applied and measured, the robot stays where it is. When a force is applied, the robot moves from its initial position by a distance that is proportional to the force. Thus, the robot behaves as a mechanical spring with a defined stiffness, even though there is no physical spring.

4.3 Tutorial Control Systems

Now that you have been introduced to the basics, we can start implementing different control systems on the exoskeleton, and finally make it do something! All of the controllers

presented in the following will be integrated into our Arduino software. As mentioned earlier, only variables that we can influence and measure can be controlled. In our exoskeleton, the values measured are the joint angle (with the motor's potentiometer, section 3.1.2) and the interaction force (with the force sensor, section 3.1.3). The values we can control with the motor's integrated servo controller are related to the motor's movements (position, velocity). Consequently, for our exoskeleton, we can only implement a position controller and an admittance controller. If we would like to implement an impedance controller, we need a motor that allows direct torque control. Keep these dependencies between hardware design and control system development in mind. We will see – and feel! – in the following sections the kind of exoskeleton behaviors we can create by using these two approaches and by combining them.

Attention!

After implementing a new controller, always test it first without your arm in the exoskeleton! The motor is not very strong and should not be able to hurt you, but better be safe than sorry. Simply fix the exoskeleton's upper arm segment to a table or another convenient object,switch the exoskeleton on and see what the controller does. You can still move it around by hand, holding it at the lower arm interface. Always test your controllers first to make sure you did not, e.g., accidentally invert a sign and the exoskeleton starts moving in the wrong direction.

4.3.1 Tutorial Position Control

Let's start by implementing two position controllers. The first one will simply move the exoskeleton from $0°$ to $90°$ elbow flexion and back. For the second controller, we will record and replay a trajectory.

Point-to-point movement

The first position controller that we will implement is a simple back-and-forth movement between two different positions (Listing 4.1).

In this sketch, you find several new elements. First, we define two new variables. The variable `counter` (line 6) stores how many point-to-point movements we have already executed, and the variable `reps` (line 7) is the number of repetitions we want to execute. Otherwise, the program would run forever until you unplug the power supply!

The `setup()` function is the same as before. The `loop()` function was extended to execute the point-to-point movement (lines 15-20). A `delay()` between the movement commands defines how fast the movement sequence is executed. In line 21, there is an increment operator ++. Applying it to our variable counter will increment (increase) it by one (there is also a − operator).

Next, we compare our counter variable with the reps variable using the == operator. Note the difference between a single equal sign and a double equal sign: a single equal sign assigns a value to a variable (see lines 5-7), while the double equal sign compares

```
1  #include <Servo.h>
2
3  Servo myservo;
4
5  int positionDesired;
6  int counter = 0;
7  int reps = 5;
8
9  void setup() {
10   Serial.begin(9600);
11   myservo.attach(3);
12 }
13
14 void loop() {
15   positionDesired = 0;
16   myservo.write(positionDesired);
17   delay(1000);
18   positionDesired = 90;
19   myservo.write(positionDesired);
20   delay(1000);
21   counter++;
22   if(counter == reps)
23   {
24     exit(0);
25   }
26 }
```

Listing 4.1: Arduino program for point-to-point movement control.

two values and returns 'true' if they are the same, or 'false' if they are not the same. We use this comparison in an `if()` statement (line 22). Because of the `if()` statement, the following block of code (lines 23-25) is only executed when the `if()` statement condition is true. In our case, until the number of repetitions reaches the value we defined in the beginning of the sketch. When that number is reached, the `exit(0)` command will end the program. If you want to restart the program after reaching the desired number of repetitions, you can push the reset button on the Arduino board.

For the first test, we strongly advise you to mount the exoskeleton to a table or similar stable support beforehand. Only after you check that the control works properly, and that the movement is as intended, you can put the exoskeleton on and test it. Just let your arm be guided by the exoskeleton. You will realize that the motor power is limited and that you can easily stop the movement if you wanted to. This is perfectly fine, as a motor of this size will always be limited in power, but it is much safer to operate and you can feel the interaction and test different controllers. Professional exoskeletons work in a similar way to this position controller, but with stronger and much more expensive actuators that are able to move paralyzed limbs, and assist with high payloads.

Prerecorded trajectory

You may have already asked yourself why an exoskeleton would execute a point-to-point movement. While this is perfectly acceptable behavior for an industrial robot that executes a pick-and-place task, the applications for human movement support are limited. It makes much more sense to record the trajectory of a 'real' human movement and replay it with the exoskeleton on demand. To do this, we will write a program (Listing 4.2) that records a trajectory and saves it in the Arduino's memory. Then, the program will replay this trajectory.

This program introduces many new aspects of the Arduino microcontroller, most of them linked to memory and data management. Let's start with a quick look at the Arduino Uno's three memory types:

1. Flash memory: where the compiled Arduino sketch is stored (32 kilobytes available);
2. RAM (random access memory): where the program creates and manages variables when it runs (2 kilobytes available); and
3. EEPROM (electrically erasable programmable read-only memory): memory that you can use to store information that is saved when the board is turned off (512 bytes available).

You probably know at least some of these memory types from your computer or smartphone. But, in those devices, you usually have much more memory available to store data or for your programs to execute. In a PC or a phone, memory sizes of megabytes (10^6 bytes), gigabytes (10^9 bytes) or terabytes (10^{12} bytes) are much more common than the few kilobytes (10^3 bytes) or hundreds of bytes we have available on a typical microcontroller. This is because microcontrollers are striped down to the very basics we need to execute our small programs. Microcontrollers do not have any operating system (that already requires gigabytes of memory), and we also do not store hundreds of vacation pictures on them.

You use and access these memory types of the Arduino in different ways. The flash memory is used to store the compiled program. You access it through the Arduino IDE when you load your compiled sketch onto the microcontroller. The program is stored there safely, even if we unplug the Arduino. This is the reason why the exoskeleton will immediately run your last program when you switch it on.

The RAM memory is used to store information while the program is running ('runtime'). All the variables that we define in our program are stored here. Different variables require different amounts of space (bytes). For example, a byte variable requires 1 byte of memory, an integer variable requires 2 bytes , etc. Thus, if you define too many variables, you might run out of memory. The RAM memory is erased when the Arduino is switched off, as it needs power to store the information.

The EEPROM memory can be accessed from the program to store data as well, but in contrast to the RAM memory, the data stored will remain there when you switch the Arduino off. Therefore, it is suited to store data you plan to reuse in another session.

```
#include <Servo.h>
#include <EEPROM.h>

#define SAMPLE_DELAY 25
#define SAMPLES 200

Servo myservo;
int servoPin = 3;
int servoAnalogInPin = A0;

void setup() {
    Serial.begin(9600);
}

void loop() {
    Serial.println("Trajectory recording starts in 3 seconds");
    delay(3000);
    recordTrajectory();
    delay(3000);
    replayTrajectory();
 }

 void recordTrajectory() {
    Serial.println("Recording");
    for (int addr=0; addr<SAMPLES; addr++){
       int posIs = analogRead(servoAnalogInPin);
       byte posIsServo = map(posIs, 119, 332, 0, 100);
       EEPROM.write(addr, posIsServo);
       delay(SAMPLE_DELAY);
    }
    Serial.println("Done recording");
}

void replayTrajectory() {
  myservo.attach(servoPin);
  Serial.println("Playing");
  for (int addr=0; addr<SAMPLES; addr++){
      byte positionDesired = EEPROM.read(addr);
      myservo.write(positionDesired);
      delay(SAMPLE_DELAY);
  }
  Serial.println("Done replaying");
  myservo.detach();
}
```

Listing 4.2: Arduino program to record and replay a trajectory.

When you compile and upload your program, you can find information on how much of the flash memory and dynamic memory (RAM) are used by your program, and how much are available, within the Arduino IDE.

With the memory discussed, let's walk through the program step by step. First, we include the EEPROM library (line 2), which provides the functions to access the EEPROM memory. Next, we define two constants with a preprocessor directive `#define` (lines 4-5). This is a little different to defining a constant variable (e.g., `const int samples = 200`). When using the `#define` directive, a preprocessor replaces all mentioning of the defined constant in the source code with the defined value before the IDE sends the code to the compiler. By replacing it in the source code before compilation, the constant is hard-coded into the program that is stored in the flash memory. This saves RAM during runtime. A potential disadvantage of using preprocessor directives is that it can create bugs that are very hard to find, especially in larger programs where you might accidentally reuse a previously defined expression in your code as a variable name, which is then replaced by a constant value. If memory is not an issue for your application, you can use 'normal' constants. But, now you have a choice!

The constants we define are a fixed sampling period `SAMPLE_DELAY`, equal to 25 ms. This means that every 25 ms we record a new trajectory point. The second constant `SAMPLES` defines how many trajectory points we record. If you do the math, you will find that we record 5 seconds worth of trajectory. You can adjust these values, but keep in mind that the EEPROM capacity is limited to 512 bytes and we need one byte per trajectory point.

Next, you will find that not all of the code is in the main loop (lines 15-21). Rather, we define two functions and call them from within the main loop. You can already see that, by doing so, the structure of the program is easier to follow.

Hint

You can also store functions in separate files, they do not need to be in the same sketch as the main program. This can help increase code readability as you only have the essential source code in the main file. It also allows you to reuse functions in other programs.

If we take a look at the `recordTrajectory()` function (lines 23-32), there is a `for()` loop inside, which is repeated once for every sample we want to record. That is done by using the loop counter `addr`, which is initialized to zero. In every cycle, if the condition `(addr<SAMPLES)` is true, the code inside the `for()` loop is executed and the loop counter is incremented by one (`addr++`). This is repeated until the condition is reached, that is, until we record all the trajectory samples.

Within the loop, the position sensor value is read (line 26). In line 27, you will find our mapping function between angle sensor voltage and angle in degrees (compare to Section 3.3.6). The idea is to not store the angle sensor value in the EEPROM memory, but rather the corresponding motor command that we have to send later to replay the trajectory. As the mapping function returns a value that is between 0 and 100 (or similar values, depending on your mapping function), we can store the motor position value in a byte

variable (8 bits). This byte is stored in the EEPROM in order. In general, the memory is accessed by a number (its address) that defines which of its bytes we want to write or read. Then, we wait for the defined delay time before we record the next sample (line 29). This function finishes by displaying when it is done recording the trajectory (line 31).

Our second function, `replayTrajectory()`, (lines 34-44) is very similar. It read outs the recorded trajectory sample by sample, and sends it to the motor at defined time intervals. The only novelty in the for loop is that we read the stored byte at the momentary EEPROM address (line 38) and then send it to the motor (line 39). When the entire trajectory is replayed, we have to detach the motor (line 43) to deactivate it. Otherwise, its servo position control system would remember the last position and stay there, preventing us from recording another trajectory.

As a general comment, note that this program only provides the very basic functionality of writing and reading EEPROM that we need for our example. Several features can and should be added to improve the code, such as preventing exceeding the address space of the EEPROM, in case you accidentally record more than 512 data points. Consider it your homework to do these improvements when you use the EEPROM storage capabilities!

4.3.2 Tutorial Admittance Control

Next, we will implement an admittance controller. As discussed before, the basic idea is that we measure the interaction force and then control the exoskeleton's movements accordingly. In a simplified way, you can see it like this: if the force is too high (more than we desire), we move away from the user. If the force is not high enough, we move toward the user.

An implementation of such an admittance controller (Listing 4.3) will measure the force as before (section 3.4.2), and then control the motor's movements.

In the sketch, you will first find that we now have two force variables, `forceIs` (line 6) and `forceDesired` (line 7). The first one we know already, this is our measured force. The second one is the force that we want the exoskeleton to impose on the user to provide support. In line 8, we have the variable `positionDesired` that we initialize with an arbitrary value within our range of motion. In line 9, we have our control gain. The higher it is, the more responsive the controller becomes. This can be good as it reacts more actively to adjust the force. On the other hand, a high control gain can quickly result in unstable system behavior.

The loop starts with our force calibration (line 17). At this point, it is important that we offset the force to know its direction. You can also include any additional calibration you might have prepared in a previous chapter. Next, we add a delay of 10 ms (line 18). This delay will affect the control frequency, that is, how often the entire control loop is executed. You can change it later for testing. Line 19 is our admittance control algorithm. You can see that we calculate the difference between the force that we measure and the force that we desire, multiply this difference by our gain, and then adjust our desired position accordingly. Last, we send the new desired position to the position controller of the servomotor (line 20).


```
1  #include <Servo.h>
2
3  Servo myservo;
4  int forceAnalogInPin = A3;
5  const int forceOffset = 439;
6  int forceIs;
7  int forceDesired = 0;
8  int positionDesired = 40;
9  float gain = 0.2;
10
11 void setup(){
12   myservo.attach(3);
13   delay(1000);
14 }
15
16 void loop(){
17   forceIs = analogRead(forceAnalogInPin)-forceOffset;
18   delay(10);
19   positionDesired -= gain*(forceIs-forceDesired);
20   myservo.write(positionDesired);
21 }
```

Listing 4.3: Arduino program for a very basic admittance control system.

Hint

By design, the force sensor is located very close to the lower arm cuff. This should ensure that the interaction force can be measured. If you interact with the exoskeleton in a way that the force sensor cannot measure it, e.g., by trying to move the exoskeleton at the 'interfaceMotorSensor' part, the admittance control will not react.

If you want the exoskeleton to just move out of the way, set the desired force to zero. In this 'zero force control' or 'transparent mode', the user should be able to move freely without being hindered by the exoskeleton. The exoskeleton is basically trying to move out of the way. The term 'zero force control' is a little misleading because you actually have to measure some force to know the intention of the user to move in a certain direction. Usually, the quality of such zero force controllers are judged on how little resistance the user feels, i.e., how small the interaction forces are in this control mode. The quality of this mode, and the control system in general, is affected by many factors, including mechanical design (the mass, friction, etc., of the exoskeleton), the speed of the actuator, and the frequency of the control system. You will find that, with our setup, this zero force control reacts rather poorly. Try to increase or reduce the delay (line 18) to test how the control frequency affects the behavior and quality of the control system. Remember: always test first without your arm in the exoskeleton! Also increase and reduce the gain

(line 9) and feel what happens. Next, you can change the desired force to another value and see how the exoskeleton behaves.

4.3.3 Tutorial Virtual Wall

Now, we will implement a controller that will allow you to move in a certain range, but will prevent you from moving beyond that desired RoM. Thus, the exoskeleton will try to limit and guide your movement to stay within a certain range. This can be very useful, for example, in rehabilitation or motor learning, to prevent users from executing undesired or even harmful movements. We will implement this controller by using a zero force controller and two threshold values (one on each side of the RoM) that limit the possible movement in each direction (Listing 4.4).

```
1  #include <Servo.h>
2
3  Servo myservo;
4  int forceAnalogInPin = A3;
5  const int forceOffset = 439;
6  int forceIs;
7  int forceDesired = 0;
8  int positionDesired = 40;
9  float gain = 0.2;
10 int maxAngle = 70;
11 int minAngle = 20;
12
13 void setup(){
14   myservo.attach(3);
15   delay(1000);
16 }
17
18 void loop(){
19   forceIs = analogRead(forceAnalogInPin)-forceOffset;
20   delay(10);
21   positionDesired -= gain*(forceIs-forceDesired);
22   if(positionDesired > maxAngle) {positionDesired = maxAngle;}
23   else if(positionDesired < minAngle) {positionDesired =
       ↪ minAngle;}
24   myservo.write(positionDesired);
25 }
```

Listing 4.4: A virtual wall controller implemented as a combination of a zero force admittance controller with two output thresholds.

The thresholds are defined by two integer variables (lines 10-11). Adjust these values in your setup if necessary. The virtual wall behavior is implemented with two `if()` statements (lines 22-23) that check if the new desired position out of the admittance controller exceeds the defined thresholds. If the desired position is out of the defined range, the

'virtual walls' become active and adjust the desired position to the maximum or minimum value before they are sent to the servo control. The resulting behavior will feel like 'walls', as the exoskeleton will prevent you from moving beyond the predefined range. You can see that the second statement is an `else if()` statement, which groups it together with the `if()`. This means the `else if()` will only be evaluated if the `if()` is not already true. This makes sense here, because only one of the two conditions can be true at a time. This can save computation time, because if there were two simple `if()` statement, both would always be evaluated in every cycle.

4.3.4 Further Control Approaches

As seen in the last section, you can combine different control approaches to define a control strategy. By combining different algorithms for when to provide support, when to prevent a (possibly dangerous) movement, etc., you will define when and how the exoskeleton assists the user. It is important to note that the control systems presented here are just the beginning, and are simple examples of what can be done. There are many more approaches out there on how to control an exoskeleton to assist the user that go beyond the scope of this handbook.

Chapter 5

Virtual Realities and Video Games

Summary

In this chapter, you will learn how computer games and exoskeletons are used together in rehabilitation, or just for fun. In the tutorial, we will create a computer game/virtual reality, connect the exoskeleton to your computer, and use it as an interface to the virtual world.

Now that we know how to use, program and control our exoskeleton, we can connect it to a PC, create a computer game, and play the game with the exoskeleton. What may sound like fooling around (and it certainly is fun) is also not uncommon in rehabilitation exoskeletons, where virtual reality (VR) and games are well-established methods used in combination with exoskeletons. The purpose of the games is to keep patients motivated and engaged during their rehabilitation. The adjustable support by the exoskeleton and the levels in the games are used to adapt the training difficulty to the patient's abilities. The goal is to facilitate recovery. But, besides medical applications, exoskeletons can also be used to interact in different ways with virtual objects or to control video games.

5.1 Background: VR/Games and Exoskeletons

We previously discussed how exoskeletons can be valuable tools for the rehabilitation of incomplete paraplegia or hemiplegia after a spinal cord injury or a stroke. They provide adjustable support to enable training at a difficulty level that optimizes the recovery of the patient. Typical applications are leg exoskeletons that support gait training, and arm exoskeletons that support the training of arm and hand movements such as reaching and grasping. Most of these devices are stationary and placed in hospitals, where multiple patients can use them every day.

Rehabilitation programs often stretch over weeks and months, and involve many training sessions that are very repetitive. This can quickly become boring, tedious, and, if progress is slow, demotivating. But, for optimal recovery, a high training intensity with active participation of the patient are crucial.

This is were games and VR can be valuable assets. Instead of walking in the exoskeleton on a treadmill staring at a wall, you control an avatar in a game with your movements and run in a stadium collecting points for every 100 steps completed. Or, you steer an airplane by moving your partially paralyzed arm in the exoskeleton to keep the plane from crashing into obstacles. The exoskeleton becomes your interface to the game, making the training more exciting.

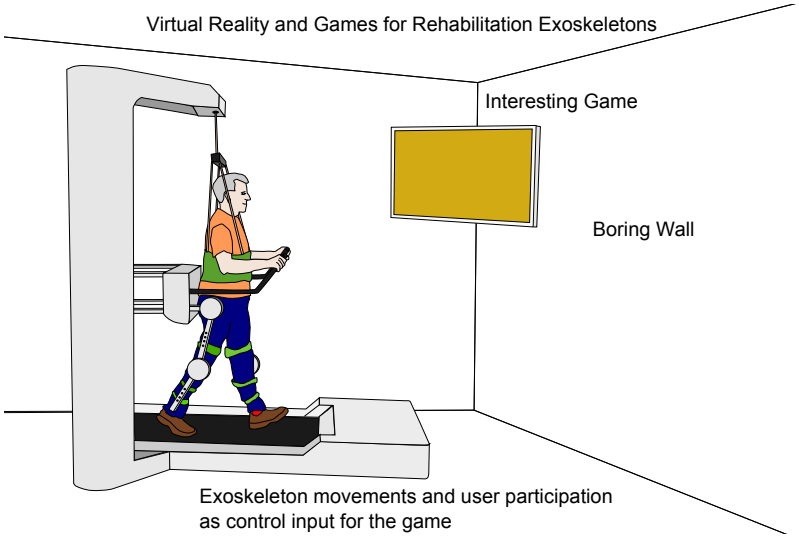

Figure 5.1: Concept of the use of VR and computer games in combination with exoskeletons for gait rehabilitation.

The application is, of course, serious (they are actually called 'serious games' and the field is called 'serious gaming' if you want to read more about it), but the main motivation is similar to games for entertainment: to be fun and to keep you engaged. As in any other game, the difficulty level has to match the user's skill level to be fun and motivating. Otherwise, it might be too simple or too overwhelming. In the context of serious games, the skill level is usually connected to the impairment level of the user. This difficulty adjustment can be done in the game settings (as in any other game), or, and this is special about exoskeleton-based games, by adjusting the support provided by the exoskeleton.

The games can also be designed to directly contribute to relearning activities of daily living (ADLs). They often simulate tasks in a virtual environment (e.g., reaching and grasping virtual objects) that will help patients restore abilities they need in daily life.

In addition, the exoskeleton can become an assessment device to monitor and quantify the patient's performance and progress (like a fitness tracker, but with more information). The exoskeleton can record data during therapy, allowing patient, therapist and doctor to compare, for example, today's session to yesterday's session, or the session two weeks ago.

5.2 Preparation for VR and Games

If you have not done so already, we will use the Unity 3D game engine. You can download it from their website at http://unity3d.com/get-unity. It is free for non-commercial use or small businesses, but requires registration. Execute the installer, launch Unity, and create an account with a personal license.

You might have to pay later on only if you want to sell your awesome exoskeleton games on a big scale.If you are completely new to Unity, we suggest you first take a look at one of the beginner tutorials to get familiarized with it. We will explain many details below step-by-step, but it will certainly be easier if you go through their tutorials: you can quickly learn how to create some simple and fun games, and already get inspired on what you can do later, once you know how to integrate the exoskeleton to games.

5.3 EduExo VR Setup

In the next tutorial, we will implement an exoskeleton gaming application with the EduExo. Our goal is to connect the EduExo to a computer that runs a video game, and use the EduExo as a gaming interface. In a first application, we will use the EduExo as a simple control input device to play a game with steering movements from your elbow joint. In a second application, we will provide feedback from the game to guide your movements with the exoskeleton when playing. The hardware setup for this tutorial (Figure 5.2) is similar to what we used before. The new component is a video game that runs on the computer and communicates with the Arduino and, therefore, the exoskeleton.

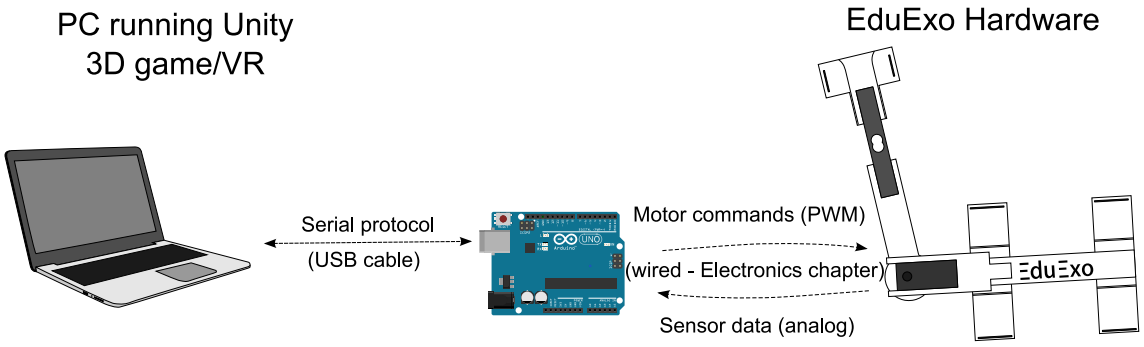

Figure 5.2: Overview of the EduExo VR setup.

5.4 Tutorial: Create a Game

Our first step will be to create a very simple Unity 3D game that we can later control with our exoskeleton. This first game will be a simple pong-style game that we can control with the left and right arrow keys (Figure 5.3). Later, we will replace the arrow control with our exoskeleton control.

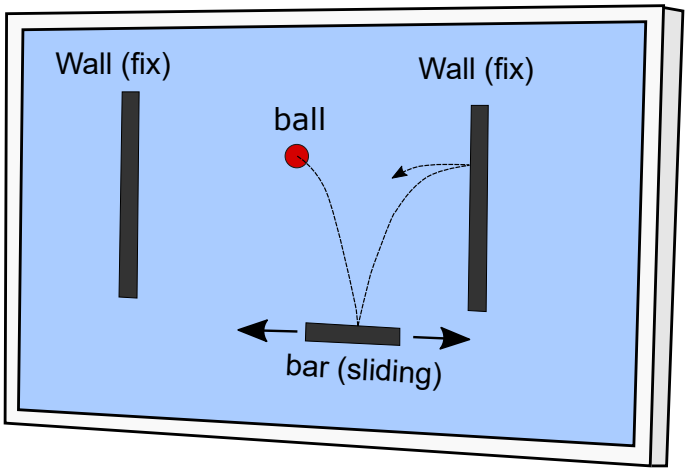

Figure 5.3: The concept for our EduExo game. You have to keep the bouncing ball from falling through the bottom of the screen by controlling the bar at the bottom.

Hint

Feel free to implement any other game you would like! Just consider a design that you can play with the EduExo as an interface device.

5.4.1 Create a Unity Project

Let's start by launching Unity and creating a new project (Figure 5.4).

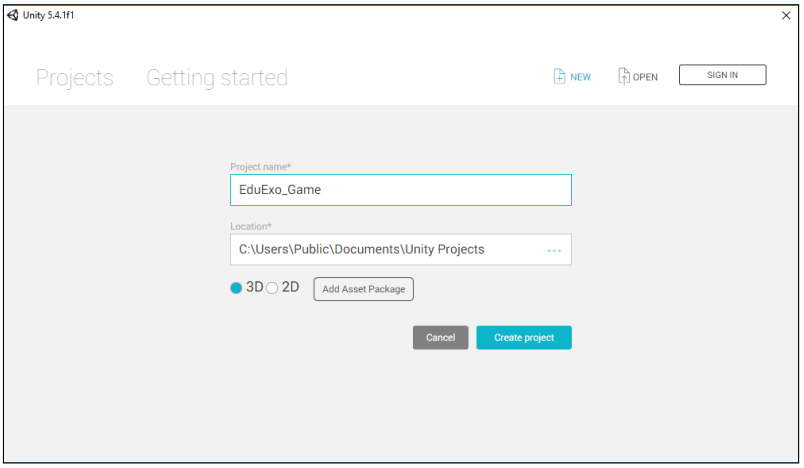

Figure 5.4: Creating a new project in Unity.

We chose a 3D game project even though we only have a 2D game design. This will enable you to expand the game into the 3 rd dimension later, if you would like. Give the project a name, change the folder if necessary, and click 'Create project'. This brings you directly into the Unity editor screen (Figure 5.5).

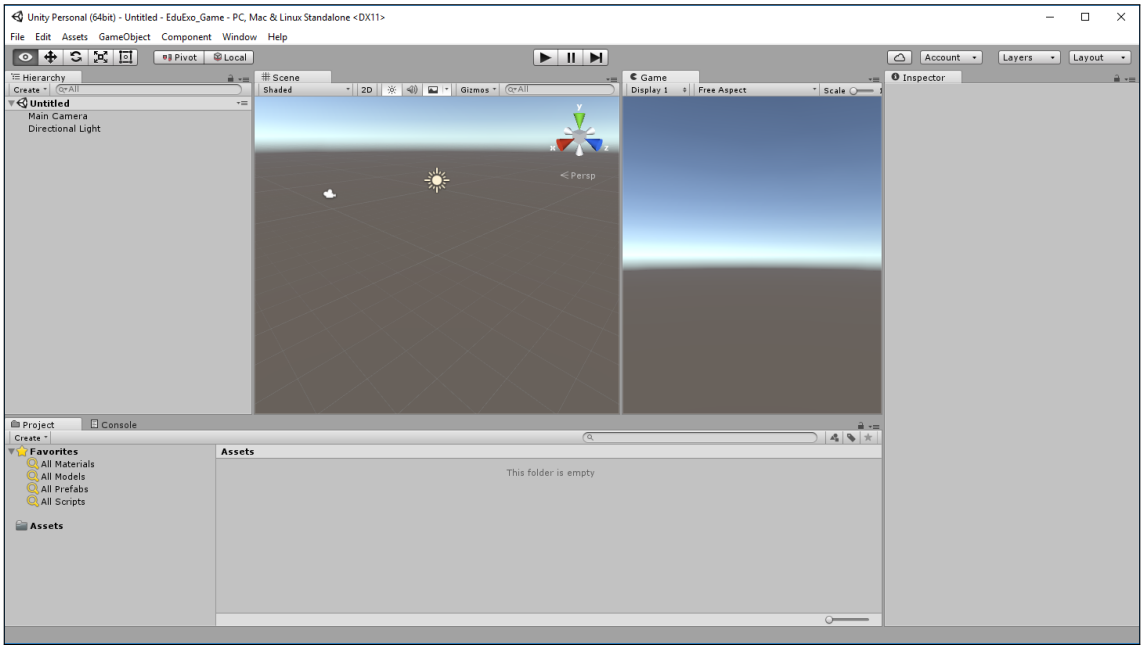

Figure 5.5: The Unity editor.

In this editor, we will create our game by adding objects to our scene, and adding assets (materials, control scripts, etc.) to these objects. In the Hierarchy tab (usually on the left), you can see that our game scene only has two default objects, a game camera (that defines what is displayed on the screen when the game is executed), and a light source that illuminates the scene (otherwise, it would be dark).

5.4.2 Adding Game Objects to the Scene

We will start developing our game by adding one of the walls to the scene. In the menu bar, select 'GameObject' → '3D Object' → 'Cube'. You will see that a small cube appears in our game scene. You will also find it in our Hierarchy tab and, if it is selected, you find its properties in the Inspector tab (usually on the right side). The Inspector always shows us the properties of the selected object; you can also select the camera or the light (by clicking on them in the Hierarchy tab or directly in the scene) to look at or change their properties. Let's change the name of the cube to 'Wall_left'. Next, let's turn it into a wall: simply change the Y-Scale value from 1 to 8 in the Inspector. Then, move the to the left side and upwards in the game window by setting the X-Position value to -7, and the Y-Position value to 5 (Figure 5.6).

If you now click on the play symbol (top-center in the Unity window), the game will start and you will see the scene rendered through the camera. As we have not yet changed the camera position, the wall may not be completely visible in the game window. To adjust that, stop the game execution, select the camera in the hierarchy window and set its Y-Position to 5. When you launch the game again by clicking on the play symbol, the wall should be on the left side and completely visible. If it isn't, keep adjusting the

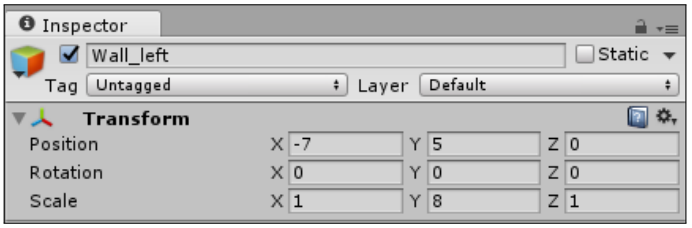

Figure 5.6: The Inspector view of the left wall settings.

camera position (or other parameters, such as 'field of view') until you are happy with the result.

Next, we will add a rigid body to the wall. This rigid body component will add physical attributes to the wall that are required for the physical simulation. Select the wall, go to the menu bar and select 'Component' → 'Add..' and then select 'Rigidbody' in the drowdown menu that appears in the Inspector. When you have done that, a rigid body is assigned to the wall and you can find it in the Inspector. You will now be able to set the physical parameters of the Rigidbody and, therefore, of the wall (Figure 5.7).

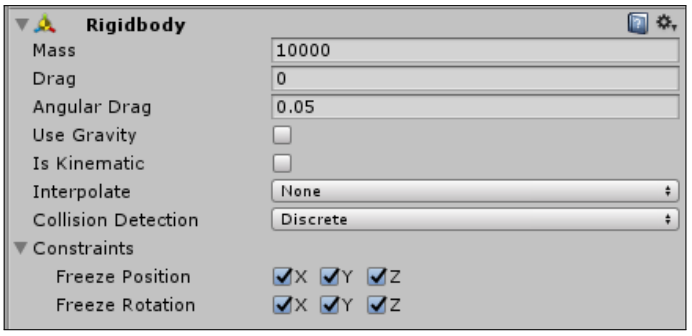

Figure 5.7: The Rigidbody setting for the wall.

If you start the game without any adjustment, the wall will start falling down. To prevent that, stop the game, go to the Inspector and un-check the 'use gravity' checkbox. Additionally, we will set the 'Mass' value of the wall very high, to around 10.000. This will be necessary for the ball to bounce back nicely from the wall and prevent the wall from being pushed away when hit by the ball. Another way to do this is by adding constraints to the wall's movement by freezing the position and rotation of the Rigidbody assigned to the wall. You find the options to do this in the 'constraints' section, at the very bottom of the Rigidbody settings in the Inspector (Figure 5.7). For our wall, we will freeze all six components as we do not want it to move at all. However, freezing the movements alone without increasing the mass might cause strange behavior from the physics simulation.

When our first wall is finished, we will duplicate it (right click on it in the Hierarchy tab and select duplicate), rename it 'Wall_right' and set the x-position of the duplicated wall to 7.

Our next step is to create the bar that we will control later on. Simply repeat the same steps as you did for the walls: create a 3D object, set the scale to make it a bar (e.g., scale

x/y/z = 6/1/1) and put it at the bottom position (position x/y/z = 0/0/0). Add a Rigidbody, deactivate gravity, and increase the mass to 10.000. In contrast to the walls, we only freeze the Y- and Z-Positions and the three Rotations. Make sure that you do NOT freeze the X-Position of the bar, as we want to move it left and right later on.

As the last object, we will add the ball to our scene. In the menu bar, select: 'Game Object' → '3D Object' → 'Sphere'. Rename it to 'Ball', set its position to 0/6/0, leave the scale as it is (1/1/1). Then, add a Rigidbody to it: keep the mass at one and, this time, leave the box 'use gravity' checked. Also, do not add any constraints except for freezing the Z-Position. Your scene should now look similar to the one in Figure 5.8.

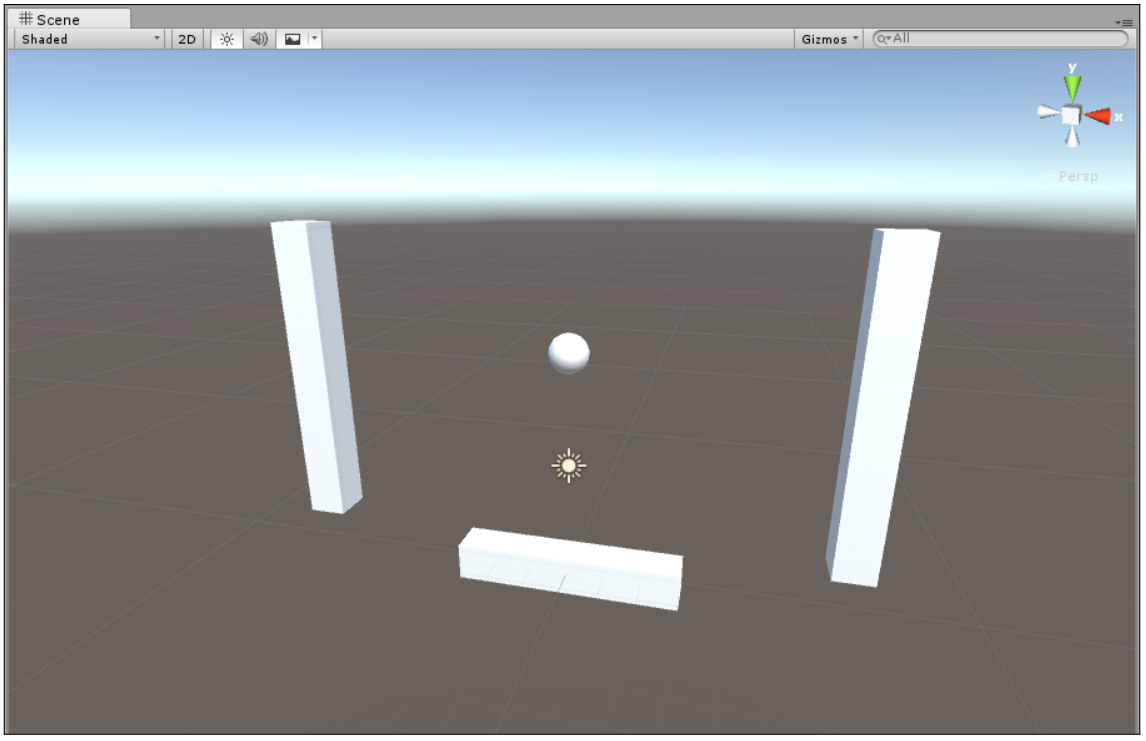

Figure 5.8: The scene with the four game objects.

5.4.3 Adding Physical Behavior to the Objects

When you finish the scene, click on the play button. If everything is set up correctly, the ball will fall down and then stick to the bar. Of course, this is not the physical behavior that we would like to see. To change that, we need to change the physical behavior of all objects to make them 'bouncy'. To do so in Unity, we first create a new 'Physical Material' (menu bar: 'Assets' → 'Create' → 'Physical Material'). The new material will appear in the assets list in the Project tab (usually at the bottom of the Unity window), and you can name it (e.g., BouncyMaterial). When you select the material in the Assets list, you will see its parameters in the Inspector tab: static friction, dynamic friction and bounciness (Figure 5.9).

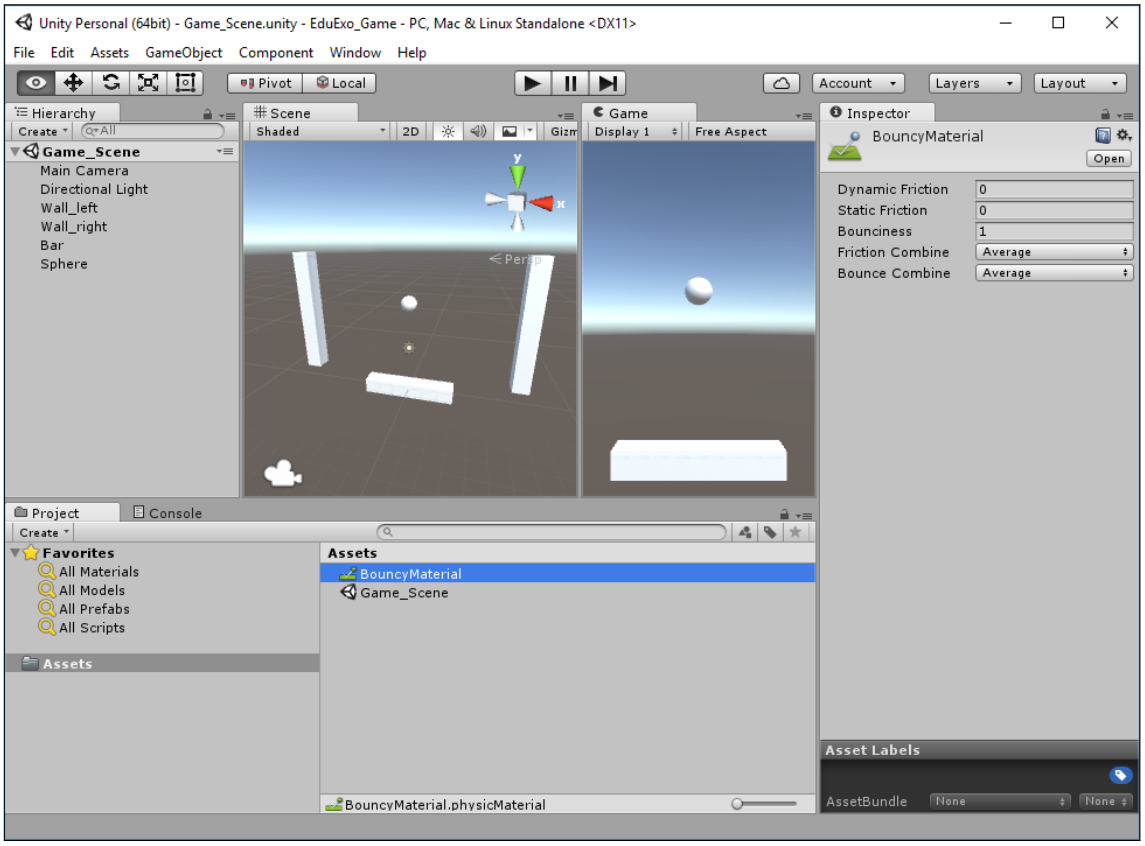

Figure 5.9: The new material in the Assets list and its properties in the Inspector tab.

Set the friction values to zero, and increase the bounciness value to one. This way, the ball will bounce forever. Next, you have to assign this material to the game objects. Simply drag and drop the BouncyMaterial from the Assets list onto all game objects in the scene window. Now, if you select an object, you will see in the Inspector tab that the material is assigned to the object's collider. (Figure 5.10)

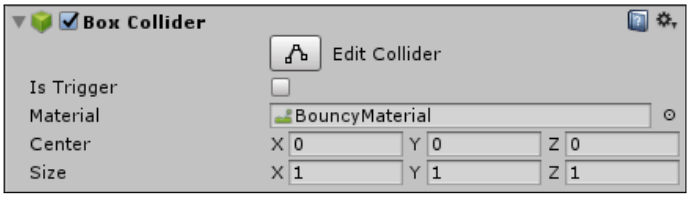

Figure 5.10: The material assigned to one of the walls.

Start the game again. This time, the ball should keep bouncing up and down on the bar. Of course, this is not very exciting yet and you cannot do anything. To change that, it is time to start programming some behavior into our components. In Unity, this can be done by adding scripts to the objects.

5.4.4 Adding Scripts

First, we will create two scripts: one to make the movements of the ball more interesting, and one to enable moving the bar left and right.

Hint

The programming language we use for the Unity scripts in this tutorial is C#. Unity supports C# and JavaScript as scripting languages. If you never heard of either language, now is a good time to quickly read up on them.

In the menu bar, select: 'Assets' → 'Create' → 'C# Script'. Name the script 'Bar-Control' and assign it to the bar by dragging it on the bar object in the scene window. By assigning it to an object this way, you can use the script to define the behavior of the object. In the Assets list, double-click on the script to open it. The script will open in a new editor window. Either the Mono development environment or another development environment if installed, e.g., Visual Studio. You will find that the script is not empty, but already has the general structure prepared for you (Listing 5.1).

```
1  using UnityEngine;
2  using System.Collections;
3
4  public class NewBehaviourScript : MonoBehaviour {
5
6      // Use this for initialization
7      void Start () {
8
9      }
10
11     // Update is called once per frame
12     void Update () {
13
14     }
15 }
```

Listing 5.1: Example of a new C# script for Unity. We will complete this script to define our object's behavior.

You will find that the Unity scripts are not that different from the Arduino code that we wrote before. First, you include libraries that you need with the `using` operator (lines 1-2). In the main part of the program, you have a `Start()` function for the initialization (line 7). As indicated by the comment in the source code (the text after `//` is a comment), this is executed once when the script is launched. Then, you have an `Update()` function that is called once per frame when the game is executed (line 12). With this template, we will now implement the functionality that allows us to control the bar with the left and right arrow keys (Listing 5.2).

The first thing we added is the variable `speed` (line 5). It defines how fast the bar moves when you press the arrow keys. If you reduce it, the game may become more


```
1  using UnityEngine;
2  using System.Collections;
3
4  public class BarControl : MonoBehaviour {
5      float speed = 5f;
6
7      void Start () {
8
9      }
10
11     void Update () {
12         if (Input.GetKey(KeyCode.LeftArrow))
13         {
14             transform.position += Vector3.left * speed * Time.
                   ↪ deltaTime;
15         }
16         if (Input.GetKey(KeyCode.RightArrow))
17         {
18             transform.position += Vector3.right * speed * Time.
                   ↪ deltaTime;
19         }
20     }
21 }
```

Listing 5.2: The C# script for manual control of the bar using the left and right arrow keys.

difficult as the bar is not able to catch up with the bouncing ball. Don't be confused by the expression `5f` (or `5.0f`); the 'f' behind the value simply ensures that C# treats the right hand side of the allocation as a float value. We don't have any initialization going on, so the `Start()` function remains empty for now.

The next new elements are the keyboard evaluation in the `Update()` function. The `if()` statements check if one of the arrow keys (left or right) is pressed. If one of these conditions is true, the corresponding code segment (in the curly brackets) is executed. The operator += then increases the current position by the increment defined on the right hand side. This position increment is calculated by multiplying the speed with the time increment `Time.deltaTime` (remember from your physics lectures?). When you finish the script, save it and start the game. You should now be able to move the bar left and right by pressing the arrow keys. While this makes the game interactive, it doesn't make it more fun, as the ball is still just bouncing up and down. To change that, we will randomize the ball's initial velocity. To do so, we will create another script, name it 'BallControl' and assign it to the ball. Open it and complete the code template (Listing 5.1) according to the example (Listing 5.3).

This script will randomize the ball's initial speed (x and y velocities), making the game more interesting and challenging. As this is required only once (at the start), we put the randomization code in the `Start()` function (lines 6-10), and leave the `Update()`

```
1  using UnityEngine;
2  using System.Collections;
3
4  public class BallControl : MonoBehaviour {
5
6      void Start () {
7          float speedx = Random.Range (5, 7);
8          float speedy = Random.Range (2, 5);
9          GetComponent<Rigidbody>().velocity = new Vector3 (speedx
               ↪ , speedy, 0);
10     }
11
12     void Update () {
13
14     }
15 }
```

Listing 5.3: This script randomizes the initial velocity of the ball.

function empty. First, we create randomized x-velocity (line 7) and y-velocity (line 8). This is done by the `Random.Range(a,b)` function, where `a` and `b` are the lower and upper limits of the random value. You can adjust them to make the game easier or more difficult. When both random speeds are generated, we assign them to the ball by assigning them to the ball's Rigidbody velocity variable (line 9). When you finish, execute the game and enjoy playing our very exciting Unity 3D game!

5.5 Tutorial: Exoskeleton as an Input Device

Now, it is time to prepare everything to play a game with the exoskeleton. In a first step, we will use the exoskeleton to control the bar in our previously developed game, i.e., play the game by moving our elbow. More precisely, we will use the measured exoskeleton elbow angle to control the bar's movement (Figure 5.11).

To connect the exoskeleton to the game, plug the USB cable into the Arduino the same way you do to program the Arduino. To use the exoskeleton to control the bar, we have to add two new features: first, we prepare an Arduino program that measures the joint angle and sends it out through the serial interface in a way that we can read it in the Unity game. Second, we will adjust the Unity game in a way that it reads in the serial communication interface to get the angle values, and then uses them to control the bar's position. Let's start with the Arduino software (Listing 5.4).

While you already know most of this code, there are a few things we added and changed to enable the serial communication with Unity. First, you find the new variable `sendValue` (line 3). Its type (byte) means that it stores an 8-bit unsigned number, from 0 to 255. While this is less accurate than the 10-bit value we can read from the analog-to-digital converter, it requires less memory and, therefore, less capacity for the

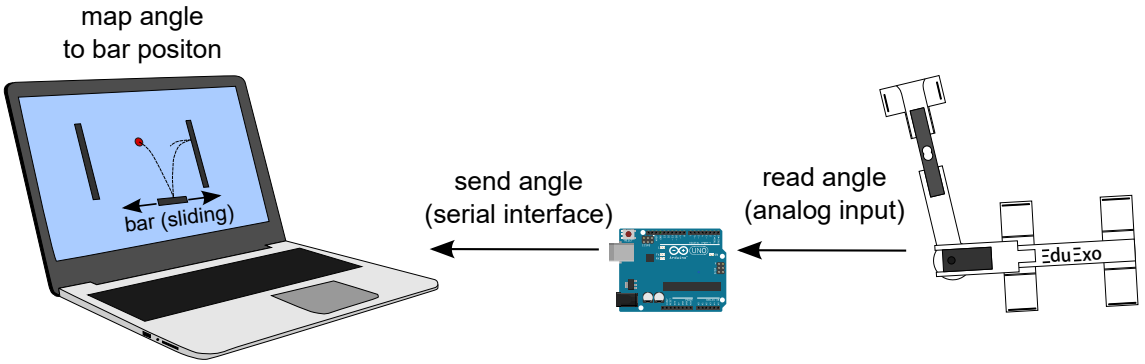

Figure 5.11: The setup and the data exchange between the devices. The Arduino reads the motor angle and sends it to the Unity game, where it is used to control the position of the sliding bar.

serial communication. We assign the angle value in line 12, where we map the raw input data to the range of the byte value.

```
1  int servoAnalogInPin = A0;
2  int posIs;
3  byte sendValue;
4
5  void setup() {
6   Serial.begin(9600);
7   delay(1000);
8  }
9
10 void loop() {
11   posIs = analogRead(servoAnalogInPin);
12   sendValue = map(posIs,80,350,0,255);
13   Serial.write(sendValue);
14   Serial.flush();
15   delay(20);
16 }
```

Listing 5.4: Arduino code to read the angle, put it into a byte variable and send it through the serial port.

The range of the input data (in the example, 80-350) was tested before by moving the exoskeleton joint through the range of motion of interest, and looking at the output data in the serial monitor (compare Listing 3.1).

Next, we send the byte value to the serial interface using the `write()` function (line 13). We use `write()` because we send a binary byte value, and not a string as we did before with the `println()` function. Then, we ensure that the transmission of the outgoing data is completed by using the `flush()` function in line 14. Last, set the delay to 20 ms (line 15). This ensures that we do not send too many values to be transmitted and processed by Unity. If the serial communication causes problems and results in strange bar behavior (e.g., jumping from left to right), try increasing this value even more to

reduce the amount of data sent. Just be aware that this affects how often data is send to the Unity game and, thus, your control frequency. If your delays are too long, the bar will start to jitter.

Now, we connect the Arduino and the computer, and upload the code as before. In the lower right corner of the Arduino IDE, you should see the COM port number to which the Arduino is connected (e.g., COM5). Write this down or remember it, we will need it in the next step in Unity.

Hint

If you reconnect the Arduino and the computer another time, the COM port number might change and communication between the game and the Arduino might not work anymore. If that happens, open the Arduino IDE again and double check the COM port number. If it changed, adjust it in the Unity script as well.

Next, we add the input functionality to the Unity game. This is done by adding a few lines to our script that controls the bar movements, which, until now, used the left and right arrow keys. Go to Unity, open the script 'BarControl', and add the serial read functionality (Listing 5.5).

In this script, we removed the arrow key control to shorten the script, but you can keep it if you want to keep the option of playing with the keyboard.Regarding the new code segment, you will first see that we included libraries that are needed for the serial communication (lines 3-5). Next, we declared our serial port object (line 8), then initialized and opened the port in the `Start()` function (lines 11-13). Here, you have to enter the COM port number you got from the Arduino IDE. Make sure the communication speed (or baud rate) matches the one from the Arduino program (in the example 9600, Listing 5.4). If you have problems with the serial communication, you can try increasing the baud rate; just ensure that you increase the Arduino's serial baud rate as well. Within the `Update()` function, we now added the serial read function within a `try-catch` statement. This `try-catch` statement prevents our entire game from stopping in case the serial read function does not work, e.g., because no data is being sent by the Arduino. We first read the bytes that are sent from the Arduino and store them locally in the integer variable `value` (line 19). Next, we map the input into our Unity coordinate system and assign it to the float variable `positionUnity` (line 20). We use a float variable because we need the position resolution of the decimal places, otherwise the bar would just hop between a few discrete positions (try using integer instead of float and see what happens). This mapping function might require a little adjustment as it depends on your game and how you assembled the exoskeleton.

Hint

If you do not see the bar when you launch the game, it is very likely that it is out of the camera's field of view (to the left or right) because the mapping function does not match your setup. Go back to your scene setup in Unity, and adjust the camera position by moving it farther away from the game objects. This will help you find the bar in the game and adjust your mapping function in the BarControl script.


```
1  using UnityEngine;
2  using System.Collections;
3  using System.IO.Ports;
4  using System.IO;
5  using System;
6
7  public class BarControl : MonoBehaviour {
8      SerialPort sp;
9
10     void Start () {
11         sp = new SerialPort("COM5", 9600);
12         sp.ReadTimeout = 10;
13         sp.Open();
14     }
15
16     void Update () {
17         if (sp.IsOpen) {
18             try {
19                 int value = sp.ReadByte();
20                 float positionUnity = (10-((float)value/10));
21                 transform.position = new Vector3(positionUnity,
                       ↪  transform.position.y, transform.position.
                       ↪ z);
22
23             }
24             catch(System.Exception e) {
25             }
26         }
27     }
28 }
```

Listing 5.5: The extended BarControl script including serial communication with the Arduino.

The last thing we do is assign the mapped position to the bar (line 21). We simply update the game object's x-position with the new value at every frame. The `catch` statements will be activated and provide an error handling message in case the serial communication does not work. Save the script and go back to the Unity editor.

Before we launch the game, we have to make sure that the Unity build settings are correct to enable the serial communication (see Figure 5.12). In the menu bar, select: 'File' → 'Build Settings' → 'Player Settings ...'. This opens the inspector in the main window; scroll down to 'Optimization' and select '.Net 2.0' for API compatibility level. We need this to have serial port support.

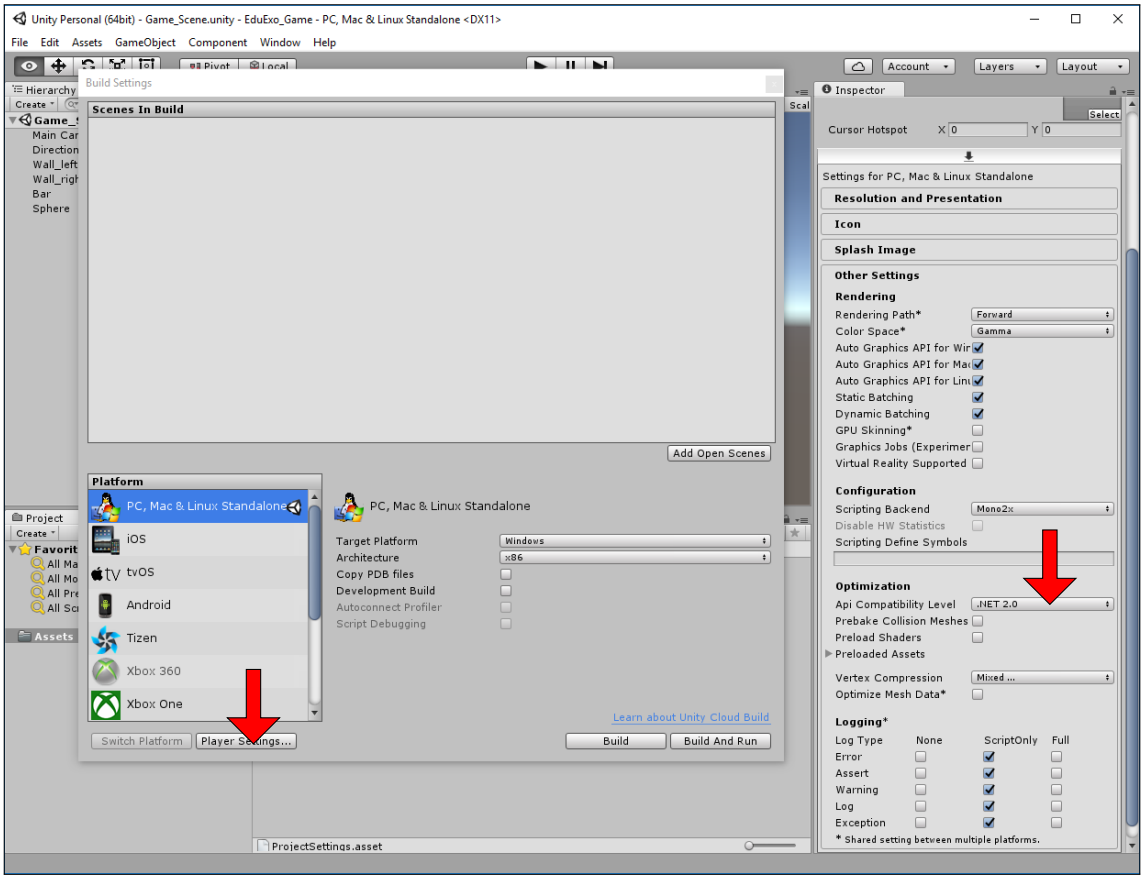

Figure 5.12: Changing the build settings for full .NET compatibility, required for serial communication.

Hint

Make sure that you closed the serial connection between the Aduino IDE and the Arduino microcontroller. Otherwise, you may encounter problems with the serial communication between Unity and the Arduino. Closing the serial monitor in the Arduino IDE should do that.

Now we can launch the game. If everything was set up correctly, you should now be able to move the bar around by moving the EduExo's joint. Note that we now directly control the bar's position and not its velocity, as we did before with the arrow keys.

Hint

Sometimes, it can take a little time before everything is initialized, including the serial communication. If the game starts before you can control the bar with the exoskeleton, add a delay at the beginning of the game.If you restart the game and have a strange behavior, it can help to also restart the Arduino: just press the small reset button on the Arduino board.

5.6 Tutorial: Exoskeleton as a Feedback Device

While playing the game with the exoskeleton is fun, so far we use the exoskeleton only as a motion tracking device. As discussed before, a robotic exoskeleton has more capabilities than that, and we will use them in this section.

What we will implement now is called haptic guidance, which will help us play the game. If we are not able to follow the ball's movements, the exoskeleton will push us in the right direction. As before, the exoskeleton's joint angle is measured, and then (after some transformations) sent to Unity. There, this angle controls the bar's position. As a new functionality, Unity now compares the bar's horizontal position with the horizontal position of the ball. If the bar is not underneath the ball, Unity will send simple commands (left and right) to the Arduino to correct the joint angle. Arduino will receive this information and control the motor position to push the user a little bit so that the bar stays underneath the ball. The technical novelty is that our Unity game will now also send information to the exoskeleton so that the exoskeleton knows when and how to support the user. To add this new functionality, we have to adjust the 'BarControl' script in Unity and the Arduino software to enable them to talk to each other. Let's start with the Unity script. (Listing 5.6).

The first new element is the definition of a write timeout value in our `Start()` function for our serial port connection (line 13). This should be done because we now also send data through our serial port, and not only receive it. The first part of the `Update()` function (lines 20-22) you already know. Here we receive the angle information and use it to control the bar position. We added the new function `sp.DiscardInBuffer()` in line 23. This function will delete any remaining data in the serial receive buffer. We add it here to improve stability of the serial communication as it helps us prevent data accumulation in the input buffer. This could happen, for example, when our Arduino sends the data faster than we process it in Unity.

The next block of code (lines 26-31) is our new send data functionality. First, we calculate the horizontal difference between the center of our ball and the center of our bar (line 27). We can access the ball object from the 'BarControl' script with the command `GameObject.Find("Ball")`. It enables us to access the data from other game objects, in our case, the x-position of the ball. After calculating the position offset, we use it to decide which command we send to Unity (lines 28-30). If the difference is lower than -3, we send the char 'L' for 'left' through the serial port. If the difference is higher than 3, we send 'R' for 'right'. If it is between -3 and 3, we send 'O' for off. As you might remember, the bar has a width of 6. Therefore, a distance between -3 and 3 means that the center of the ball is still above the bar. In that case, we don't want any support, we are good enough by ourselves. Of course, you can change these values to adjust when the exoskeletons starts supporting you.

```
1  using UnityEngine;
2  using System.Collections;
3  using System.IO.Ports;
4  using System.IO;
5  using System;
6
7  public class BarControl : MonoBehaviour {
8      SerialPort sp;
9
10     void Start () {
11        sp = new SerialPort("COM5", 19200);
12        sp.ReadTimeout = 5;
13        sp.WriteTimeout = 5;
14        sp.Open();
15     }
16
17     void Update () {
18        if (sp.IsOpen) {
19         try {
20            int positionSensorValue = sp.ReadByte();
21            float positionUnity = (10-((float)positionSensorValue
                  ↪ /10));
22            transform.position = new Vector3(positionUnity,
                  ↪ transform.position.y, transform.position.z);
23            sp.DiscardInBuffer();
24         }
25         catch(System.Exception e) {}
26         try {
27            float positionDifference = transform.position.x -
                  ↪ GameObject.Find("Ball").transform.position.x;
28            if(positionDifference < -3) {sp.WriteLine("L");}
29            else if(positionDifference > 3) {sp.WriteLine("R");}
30            else {sp.WriteLine("O");}
31         }
32         catch (System.Exception e) {}
33        }
34     }
35 }
```

Listing 5.6: The extended BarControl script including the bi-directional serial communication and the active position feedback.

Next, we have to adjust our Arduino script to receive the movement commands from Unity and apply them to control the exoskeleton (Listing 5.7).

To receive the command from Unity, we create a variable of type `char` and call it `unityCommand` (line 7).

Within the loop, we first check if Unity has sent a new command by checking if incoming serial data is available (line 16). If we have received data, we first answer by sending the current motor position as we did before (lines 17-21). Answering with a position only after having received data from Unity is another measure to keep the serial communication stable. This way, we can assume that Unity had time to process the last position we sent.

Next, we read the the first byte of incoming serial data available with the command `Serial.read()` and store it in our variable `unityCommand` (line 23). Next, we interpret the data with a `switch-case` statement (lines 24-43). The switch-case statement compares the content of the variable to predefined possibilities and executes the code segment in the corresponding case-section. If we receive an 'O' from Unity, we will deactivate the motor if it is active. If we receive and 'L' or 'R' from Unity, we activate the motor if necessary and then send it to a fixed position on the left or right (in the example code, 0 and 80). By doing so, the motor will push towards the ball. As the motor is not very powerful, we can simply use position control to implement this kind of assistance. Note the `break;` commands at the end of each case section. This is necessary to exit the entire case block after we found a match. Without the break command, all the code below our first match would be executed.

The last thing that we do after the `switch-case` statement is to erase the Arduino's serial input buffer (line 44) with a simple loop that reads out all existing bytes. This is again a measure to ensure stable communication. It prevents the serial input buffer on the Arduino side to fill up in case Unity sends commands faster than our Arduino program can process them.

When you have completed the code, upload the sketch to the Arduino and launch the game. You should be able to play the game as before by moving the passive motor to control the bar's position. But, the moment the ball leaves the area above the bar, the exoskeleton should start pushing you into the direction of the ball. You can also start the game without your arm in the exoskeleton and watch it play the game by itself.

Congratulations, you implemented your first real exoskeleton-based video game! Of course, this is a relatively simple feedback solution that does not anticipate the ball's movements, it just follows the ball around. This could certainly be expanded and improved. At this point, you have the knowledge of how it can be done and it is up to you and your creativity to add features as you like!

This also brings us to the end of the EduExo tutorial. But this does not mean that your excursion into the world of exoskeletons has to stop here. There are infinite possibilities to continue from here, and you might already have your own ideas of what you want to try next.

```
#include <Servo.h>

Servo myservo;
int servoAnalogInPin = A0;
int posIs;
byte sendValue;
char unityCommand;

void setup() {
 Serial.begin(19200);
 myservo.attach(3);
 delay(1000);
}

void loop() {
    if(Serial.available() > 0){
      posIs = analogRead(servoAnalogInPin);
      sendValue = map(posIs,80,350,0,255);
      Serial.write(sendValue);
      Serial.flush();
      delay(10);

      unityCommand = Serial.read();
      switch(unityCommand)
       {
         case 'O':
           {
              if(myservo.attached())  myservo.detach();
           }
           break;
         case 'L':
           {
            if(!myservo.attached())  myservo.attach(3);
            myservo.write(0);
           }
           break;
         case 'R':
           {
            if(!myservo.attached())  myservo.attach(3);
            myservo.write(80);
           }
           break;
       }
    while(Serial.available() > 0) {Serial.read();}
    }
  }
```

Listing 5.7: Arduino code for bi-directional serial communication with Unity.

Chapter 6

Beyond the Handbook

We hope you enjoyed reading this handbook and implementing the tutorial, and you had fun developing your expertise in robotic exoskeletons. But, as we mentioned before, you don't have to stop here: you can use the EduExo as the starting point for your own ideas and experiments. Here are a couple of ideas you could start with:

- Develop new games: the example we developed was very basic and not the most inspiring game or VR application. Wouldn't you prefer to steer a spaceship through an asteroid belt using the exoskeleton while being chased by aliens?
- Explore new applications: how about a exoskeleton curl counter for your dumbbell training? Maybe exoskeleton-based fitness training can help you to achieve your New Year's resolution?
- Adjust the exoskeleton hardware: why not redesign the exoskeleton to better fit your arm? For example, you could design new cuffs that fit perfectly to your arm. Learn about mechanical design and how to use a CAD (computer aided design) software. You can print new parts yourself if you own a 3D printer, or you can check if there is a printing option in your area or an online service.
- Implement better control systems: as mentioned before, the field of exoskeleton control systems and strategies is quite large. Maybe you find an approach that is better suited for you ideas. Or, you can develop your own control strategy.
- Changing the game feedback: is the feedback good enough to allow you to play the game with your eyes closed? If not, can you achieve that by adjusting the feedback?
- Improve the communication between Unity and Arduino: you can implement a proper protocol with plausibility checks. Right now, we do not really 'ask' for data, we simply send it and hope it will be received.
- Implement filters to improve the force and position signal: signals can be quite noisy, which affects the control system. Can you implement a software filter on the Arduino, or an analog hardware filter on the breadboard, to improve that?

But, we are sure that you have many great ideas of your own. If you implement and test something new with the EduExo kit, we would love to hear about it! If you would like, we can also feature the best ideas on our EduExo website (www.eduexo.com). On this website, you can find complementary material to this handbook, including multimedia content.

If you have any wishes or ideas for the EduExo kit, handbook or extensions, we would be very happy if you share them with us. We will be adding several extensions to the EduExo in the future, so check out our website from time to time and subscribe to our mailing list to stay updated. For more information also on future products, make sure you also visit our company website www.beyond-robotics.com.

Thank you very much for using the EduExo kit and all the best from our side for your projects and career!

The EduExo Team