A Stabilization Method for Upper-Body Wearable Robotic Limbs

Dr. Ata Otaran
otaran@cs.uni-saarland.de
Human Computer Interaction Lab
Saarbrücken, Germany

Anna Calmbach, Sophie Kunz, David Wagmann
s8ancalm@stud.uni-saarland.de
s8sokunz@stud.uni-saarland.de
s8dawagm@stud.uni-saarland.de
Saarbrücken, Germany

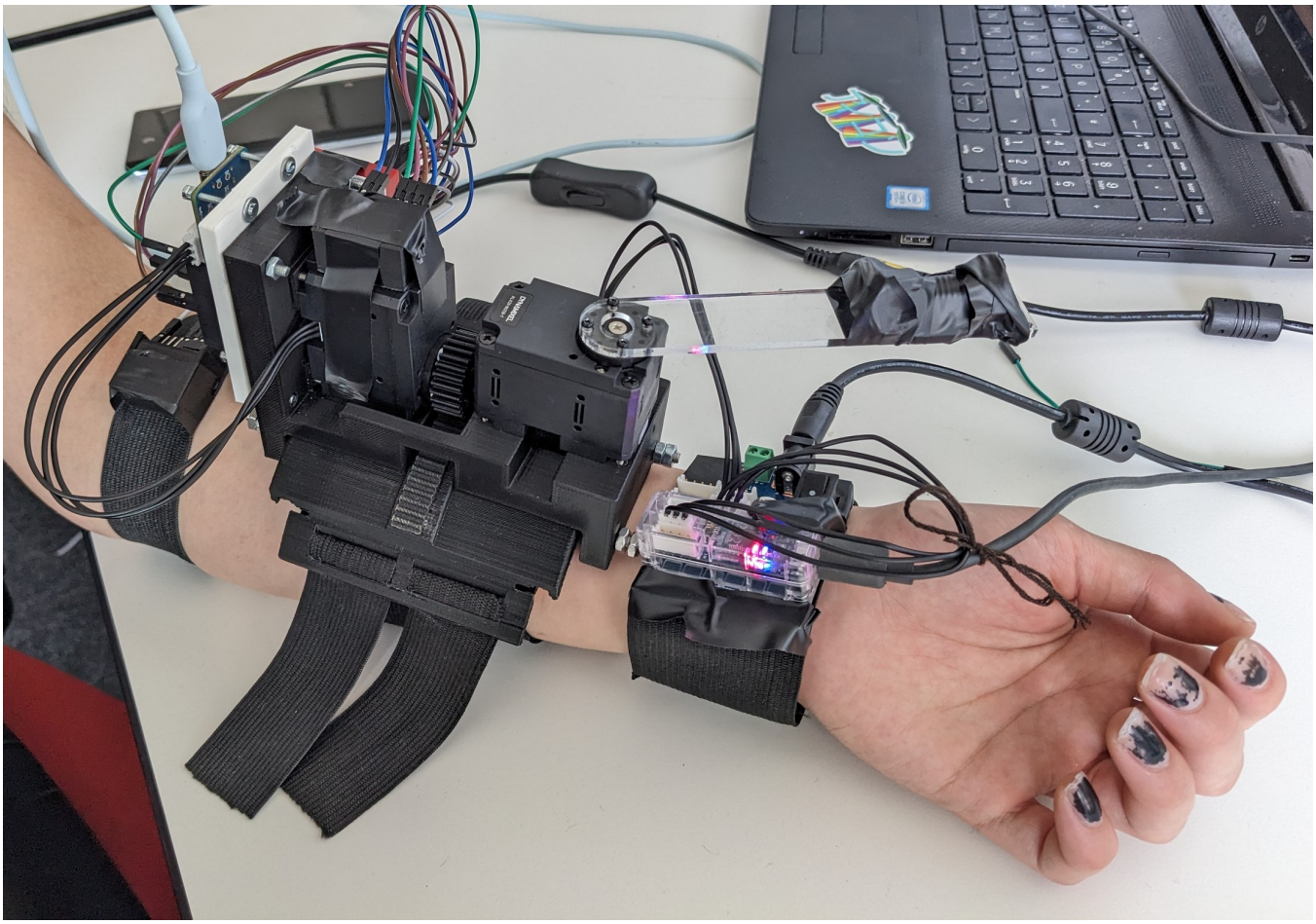


Figure 1: A picture of the final prototype attached to the arm. On the left, we have the reference sensor. The middle shows the balancing prototype with the motors and the base sensor. Lastly, the rightmost strap contains an additional control board for the motors.

ABSTRACT

Wearable robotic limbs, especially those carrying heavier weights, often face stabilization issues. Due to the deformability of the skin and the weight of the robot, small movements of the skin and thus of the robotic limb can happen frequently. Our work focuses on the issue of fixing this skin drift and it consists of two parts. Firstly, a sensing technique using two sensors to detect if and how much skin drift is happening. The second part consists of developing and building two structures to be able to counter the drift. One structure being the rack casing, which is directly attached to the user's arm and holds the rack. The second structure, the motor casing, holds the robotic limb and moves on top of the rack to counter the skin drift, thereby stabilizing it. Within this paper, we present a working first attempt at fixing the skin drift and propose ideas to improve the technique in future work.

1 INTRODUCTION

Wearable robotics is an emerging field of research that is rapidly evolving in recent years. There is a lot of research done on prototypes such as the Supernumary Robotic Limbs [7], Robotic Symbionts [1] and the research of Vatsal et al., who designed a wearable

robotic forearm which provides the user with an assistive third hand [3]. However, when mounting such a robotic limb on the body, there are a lot of things to consider. Wearable robotic limbs need to be accurate and stable in their movement to execute tasks precisely [4]. They also need to be attached to the human body and thus often lay upon the skin. Human skin however is soft and easily deformable, which may result in skin drift. This means that movements of the end-effector may cause the skin to deform and thus falsify the position of the end-effector. This is a problem for executing tasks with a robotic limb. Related research has already implemented mechanisms in order to stabilize wearable robotics. With the Naviarm system [2], the approach to stabilizing the attached limbs is to construct a backpack-like structure as a base for the limb. However, this system is very obstructing and research suggests that users would rather prefer wearing such a limb mounted on the arm or shoulder [5]. The Tomura [5] is a wearable mounting mechanism that enables rotation and quick attachment and detachment of robotic limbs anywhere on the body. In this work, they used a servo motor capable of turning 360°, combined with a stabilizer to change the orientation of the attached limb. Vatsal et al. developed an approach to detect the drift of end-effectors by using eye-tracking cameras [4]. With the data from the cameras, they built disturbance models to detect unwanted drift. However, while this approach balances out the orientation of the end-effector in the light of the movements of its user, this does not take skin drift into account.

We concluded that current methods of mounting the limb to the body are not sufficient to counter unwanted drifts of the end-effector due to skin deformation. This is why our contribution to current research is developing a base structure system that can accurately detect skin drift as well as balance out the end-effector according to the measured drift.

2 PROTOTYPE CONCEPT

Since we wanted our approach to be as modular as possible, we decided against plainly implementing an algorithm that would counter the drift by adjusting the movement of the robotic limb itself. That is why we gathered and refined multiple approaches, that counter skin drift, which can be modularly applied to a robotic limb.

In our first approach, we used the existing mounting base structure of the robotic limb, which can be seen in Figure 2. We thought about adding counterweights on top of this base and shifting the weights in a similar fashion as Shifty [6]. This, however, would make the prototype more uncomfortable to wear due to the added weight. Furthermore, we suspected that latency could be troublesome here as shifting the weight could easily be not fast enough. Another issue we saw was that this method of countering the drift would only work for an upright arm position and could even cause worse drift when used in another position.

Placing the base structure of the robotic limb on top of a Gimbal was a further option we came up with. We quickly had to deem this not feasible in the limited scope of a seminar using DIY materials. However, this could be an interesting approach to explore in future work as it is not only possible to fix skin drift with this, but one would further be able to keep the whole structure in balance. Images of the concept sketches can be found in the Appendix subsection A.10.

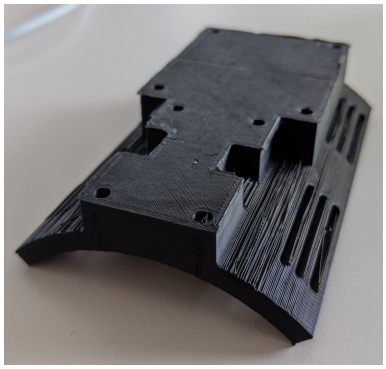


Figure 2: The original base structure of the robotic limb.

3 IMPLEMENTATION

The task of our prototype is to stabilize the rotation of the base structure. Furthermore, our stabilization method must work dynamically while moving the arm. In order to do this, we need a way to sense the rotational data and we must propose an algorithm that processes the input, such that we can adjust the orientation of the base structure as presented before.

In the following, we describe our process, coupled with challenges, pitfalls, and justifications for our design decisions.

3.1 Sensing the Drift

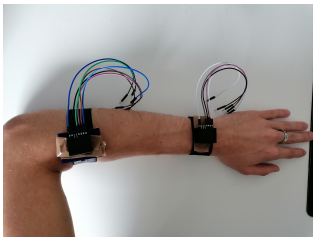
Measuring the drift that arises is the first important part of this work. In order to achieve this we need *sensors*, that deliver the needed data. Furthermore, we discuss where the sensors should best be placed and how to realize the sensing with DIY components.

Sensor Concept.

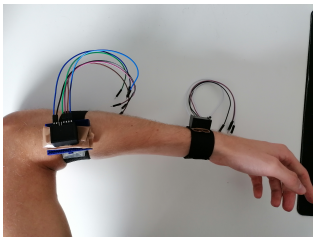
To receive the drift data we need two sensors. One sensor measures the current rotation of the base underneath the robot arm *(base sensor)*, and another sensor keeps track of what rotation the base structure should have without drift *(reference sensor)*. Through both sensors, we are able to calculate the required movement on the rack to negate the drift.

Sensor Placement.

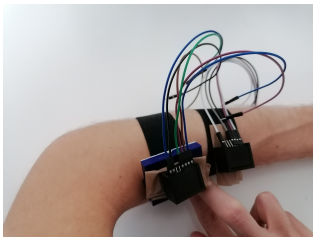
While the placement of the base sensor can be anywhere on the base structure, the position of the reference sensor is challenging. The reason for this is that the sensor must be close enough to the base, such that it rotates equally to the base when the arm moves (Figure 3a & Figure 3b). In addition, the sensor must be placed in a way such that it is not affected by skin drift (Figure 3c). In our work, we tested several positions on the arm to find a suitable place for the reference sensor: Same side - far distance (Ssf), same side - medium distance (Ssm), same side - near (Ssn), and opposite side - near (Osn). Note that all these results are introspective and emerged from internal tests that we did. To get better results one should



(a) Base (left) & reference sensor (right).



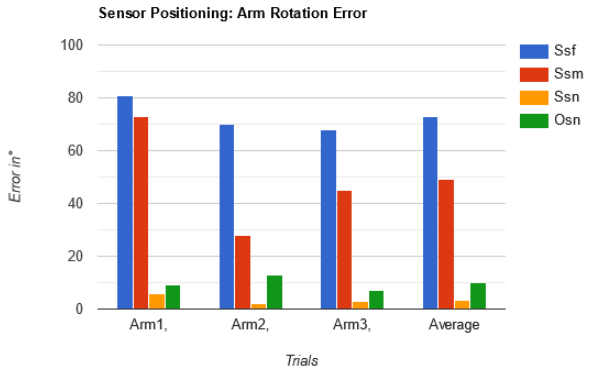
(b) Arm rotation does not rotate both sensors equally.



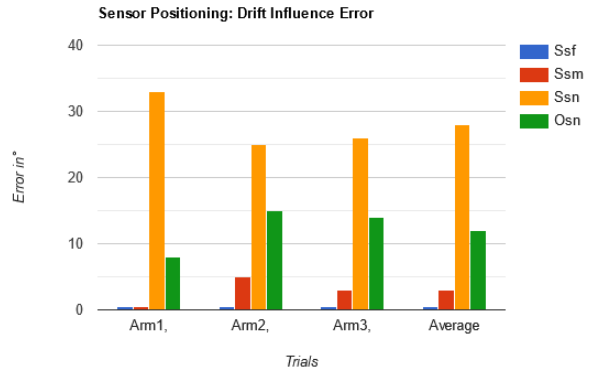
(c) Stretching skin also affects the reference sensor.

Figure 3: Sensor positions on the arm

construct a study with more positions, more participants, multiple trials for every position, and a more stable sensing setup (e.g., turning/bending by x-degree eyesight is not precise). In the scope of testing, we conducted two tasks for each position. Firstly, we rotated the arm by exactly 90° and we measured the error between the sensors, which should be roughly 0°. In the second task, we simulated skin drift, by bending the skin by about 90° and measured the influence on the reference sensor, which should also be 0°. The arm circumferences varied between 14 and 17 cm on the wrist and between 21 to 25 on the arm bend. Every participant used the left arm. For the first task, one can observe that the further the sensors are apart (vertically & horizontally) the more significant the gap when we rotate the arm (Figure 4a). Thus, one should place the sensors closer together to minimize the first error. In the second task, we receive the opposite result (Figure 4b). The further the sensors are apart the lower the influence of the drift on the reference sensor. Following this, one shouldn't place the sensors too close to each other. As both results contradict, we must find a compromise. Because the errors, for the large distances, in the first test are higher (Ssf max = 81°, ssf avg = 73°, ssm max = 73°, ssm avg = 49°) compared to the error of the short distances in test 2 (Ssn max = 33°, ssn avg = 27°, osn max = 15°, osn avg = 12°) we use shorter distances between the sensors. Placing the sensor on a separate strap opposite or next to the base worked well for us. However, people have different arm sizes, etc., and therefore the position of the reference sensor could be worked out further in the future especially since we haven't elaborated on upper arm positions at all.
Additionally to the positioning process we 3D printed cases (Figure 5a & Figure 5b) for the sensors, such that it is easier to attach the sensors to the arm or base structure. At the same time, these cases ensure that the sensors shake less when the arm moves.



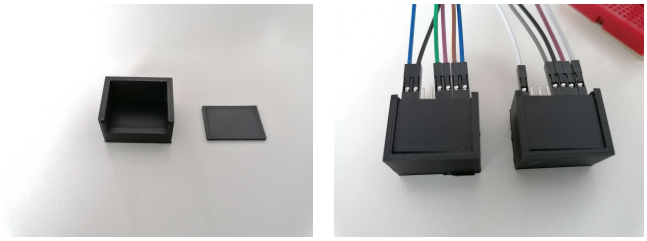
(a) Error between sensors when rotating the arm.



(b) Influence of skin drift on reference sensor.

Figure 4: Sensor positioning: Introspective statistics

Sensor Hardware & Circuit.
In our prototype, we use the MPU-6050 sensor, a 6 DOF accelerometer & gyroscope. Additionally, we employed an Arduino Uno in the test phase, that controls the sensors. For the communication between the sensor and the Arduino, we use the I^2C bus connection. The first MPU-6050 gets connected to VCC, GND, and the bus lines, the SCL (serial clock) & SDA (serial data). However, for installing the second sensor we must use the AD0 pin instead of VCC, such that this sensor gets a different I^2C address (0x68 vs. 0x69). The complete circuit diagram can be seen in Figure 6. Unfortunately, we had trouble with connecting the Arduino shield, that controls the motors. Therefore we switched to the OpenRB-150 for the final prototype. However, the complete software and circuit are applicable to the Arduino-Uno with a functioning shield.



(a) Individual parts. (b) Assembled with sensors.

Figure 5: 3D-printed casing.

Sensor Implementation.
For the Arduino implementation, we used the i2cdevlib library and modified it. Firstly, we removed the complete section for interrupts, as we did not use them in our prototype and they resulted in crashes of the software. Following this, we extended the code to handle both MPUs at the same time, such that we can continue with the calculations based on the sensor data.

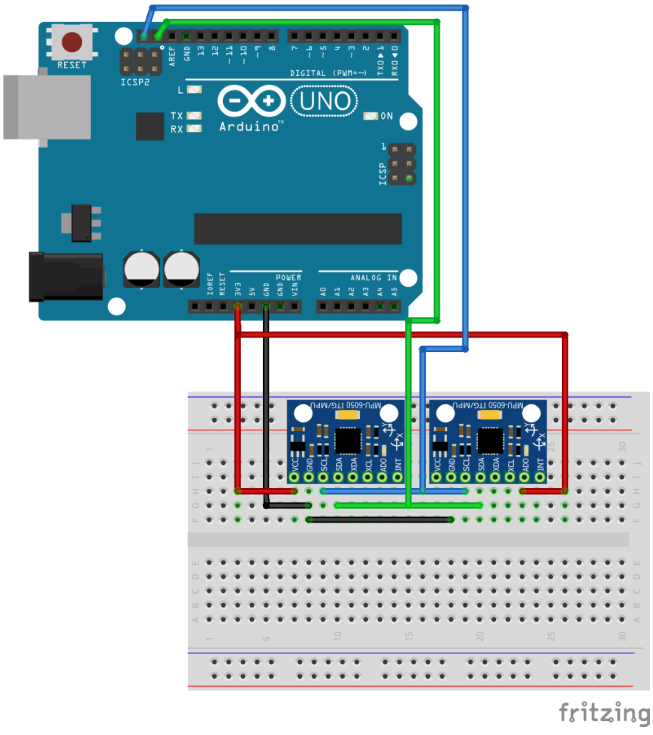


Figure 6: Circuit for connecting 2 MPU-6050[1].

3.2 Processing the Drift

The next challenge is to infer the skin drift from the raw sensor data. Therefore, we need to understand how the different rotation representations work and how calculations with rotations are performed.

[1]Image from https://daniel-ziegler.com/arduino/mikrocontroller/2017/06/11/arduino-mpu6050/

Naive False Calculation.
At this point, we want to present a very simple, but incorrect solution to the problem, which was frequently mentioned during our conception phase. We aim to extract the difference in z-rotation between the two sensors, such that we can describe the angular error α. The angular error describes the difference between two rotations on a specified common axis. For instance, how much do we have to turn rotation 1 to the right (z-axis: [0,0,1], Figure 7) to match rotation 2? Therefore, it is appealing to display our rotations as Euler angles since they grant us direct access to the z-rotation values. An error-prone concept would be to say that the angular error α equals the z-rotation difference, meaning that we can ignore the rotation values of the x and y axes. But this is wrong because Euler angle rotations get applied in a certain order, e.g., x-y-z. Hence previous axis rotations change the local orientation where the current rotation gets applied. For instance, the cubes in Figure 8 point in the same direction (It's not the same orientation, but direction $\leftrightarrow$ The up vectors are identical), although the z rotation is 0 in the first example. Thus, we would incorrectly output that the angular error $\alpha = 0$ on the z-axis, despite the 30° error.

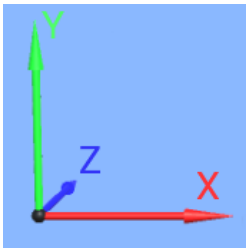


Figure 7: Coordinate System (X,Y,Z).

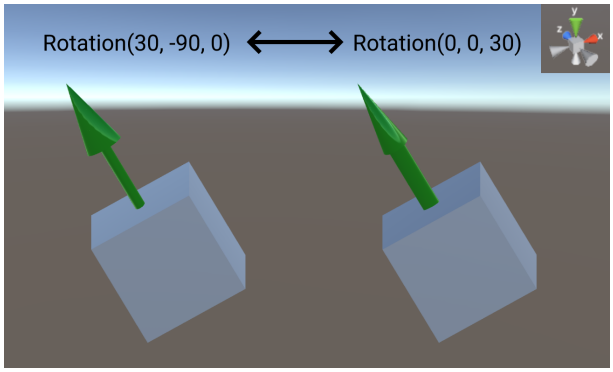


Figure 8: Although the left rotation points in the same direction as the right one, the z rotation is 0.

Rotation Representation.
Moreover, Euler angles have further downsides, because they suffer from gimbal lock. As mentioned before, the rotations get applied in a certain order. This can result in two axes being aligned. Hence, rotating along these axes has the same effect. In summary, we lose one degree of freedom (DOF) through gimbal lock and in order to prevent this we use Quaternions instead. Quaternions are an extension of the complex numbers $\mathbb{C}$, also called Hamiltonian numbers. Unit quaternions (Quaternion of length 1) can be used to describe rotations. They are represented by a 4-dimensional vector of the form: $x_0 + x_1 \cdot i + x_2 \cdot j + x_3 \cdot k$. Here, $[x_1, x_2, x_3]$ describes the rotation axis around which we rotate and x_0 is the rotation angle ϕ.

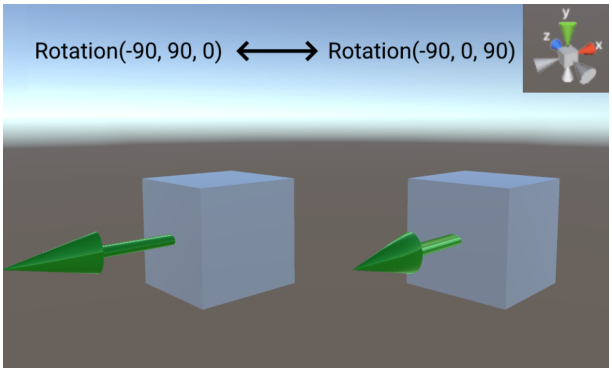


Figure 9: Gimbal Lock: Y & Z axis have the same effect.

Calculations.
Given two rotations in unit quaternion representation $r_1 = [w_1, x_1, y_1, z_1]$ & $r_2 = [w_2, x_2, y_2, z_2]$, we want to calculate the angular error α between both rotations on a predefined axis. For our cause, we want to receive the drift along the z-axis (See Figure 7). In the following, we propose a way to perform these calculations but mark that there might be quicker or more advanced solutions to this problem.
First, we calculate the *rotational difference* (r_d) between *from* (r_1) & *to* (r_2). Therefore, we invert the rotation of *from*:

$$r_1^{-1} = \frac{r_1^*}{\|r_1\|^2} \tag{1}$$

Where r_1^* is the conjugate of r_1. Next, we receive the rotation difference between *from* & *to* by calculating:

$$r_d = r_1^{-1} \cdot r_2 \tag{2}$$

As an auxiliary step, we convert the rotation from the sensor coordinate system to the coordinate system in Figure 7. This can be done by switching the signs and components of the quaternion.

$$r_{adjD} = [r_{d_w}, -r_{d_x}, r_{d_z}, r_{d_y}] \tag{3}$$

Now that we know the adjusted rotational difference (r_{adjD}) we transform the rotation to a position in the local space of the sensors. To do this we take the up vector $\vec{v}_{up}$ = (0,1,0) and rotate it by our r_{adjD}:

$$r_v = [0, \vec{v}_x, \vec{v}_y, \vec{v}_z] = [0, 0, 1, 0] \tag{4}$$

$$\vec{v}_{rot} = r_{adjD} \cdot r_v \cdot r_{adjD}^* \tag{5}$$

r_v is the pure quaternion constructed from our up vector $\vec{v}_{up}$ and $\vec{v}_{rot}$ is the resulting rotated up vector.
In the next step, we project $\vec{v}_{rot}$ to a 2D plane. This means that we set the z-coordinate to 0 which gives us v_{proj}.

$$\vec{v}_{proj} = [\vec{v}_{rot_x}, \vec{v}_{rot_y}, 0] \tag{6}$$

Lastly, we can extract α from $\vec{v}_{proj}$ by calculating the angle between $\vec{v}_{proj}$ and $\vec{v}_{up}$:

$$\theta = acos(\frac{\vec{v}_{proj} \cdot \vec{v}_{up}}{|\vec{v}_{proj}| \cdot |\vec{v}_{up}|}) \tag{7}$$

$$\alpha = \theta \cdot \frac{180}{\pi} \tag{8}$$

Since the acos result is in radians, we must convert it to degrees (8) in order to receive α. This is the angle by which we must rotate the base of the robotic limb, on the z-axis to even out the angular error.

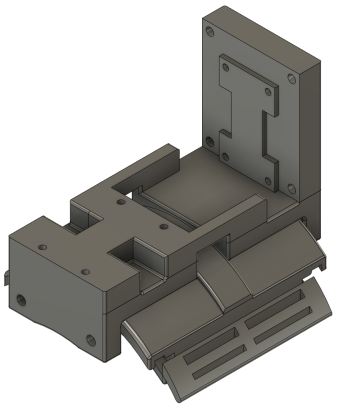
Calibration Stage.
The MPU-6050 has no magnetometer and therefore, we only get rotations of the sensors in their respective local space. Meaning that the measured forward axis depends on the orientation during initialization. This results in the disadvantage that the orientations of both sensors must be aligned during calibration, otherwise we calculate with wrong values.
In our prototype this alignment is necessary, but in future work, one could use sensors that include a north reference. For instance through the use of a magnetometer. That way we can define the orientation offset r_{of} between both sensors during calibration and add this to the calculations. Step 2 would change as follows:

$$r_{of} = r_d = r_1^{-1} \cdot r_2 \quad | \ \textit{During Calibration}$$
$$r_d = r_{of}^{-1} \cdot (r_1^{-1} \cdot r_2) \quad | \ \textit{At Runtime}$$

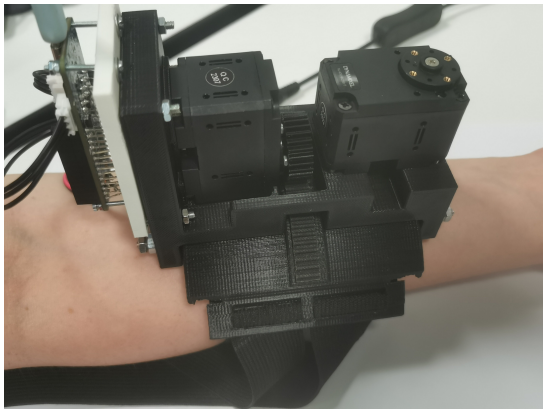
By adding this extra step we can now position the sensors freely on the base and arm, without worrying about their orientations during calibration. Moreover, one could use the same approach to change the goal position during runtime, for instance through an attached slider.

3.3 Development of the Prototype

Now we are able to accurately sense the drift. In order to counter it, we needed to come up with a way to appropriately move the robotic limb.
After some iterations on our previously mentioned concept, we decided to build the two following structures to counter the drift. One structure, the rack casing, is directly attached to the user's arm. As the name suggests, it holds a rack, which is placed in the middle of the casing. The second structure, namely the motor casing, which holds the robotic limb, is mounted on top of the rack casing. By adding a motorized pinion to the motor casing we are able to move the motor casing and thus the robotic limb along the rack. To be able to stabilize the movement of the motor casing, we added small ball bearings to each corner of the casing. These bearings slide alongside a rail that is directly attached to the rack casing, thus allowing the motor casing to only move on one axis on top of the rack casing.
It was especially challenging here, that we needed a robust but also curved structure that would be able to fit the arm of the user. The need for a curved structure further limited our options for off-the-shelf materials to use. Since this was a very specific use case, we decided to model and 3D print both casings as well as a pinion using PLA filament. One can see the 3D models and the final prints of the assembled casings in Figure 10. For the pinion, we used the Autodesk Fusion script *SpurGear* to configure the right size and amount of teeth (see Figure 11).
The pinion is mounted on and rotated by a Dynamixel XL430-W250T. For our rack, as it again needed to be flexible, we utilized a belt originally used for a power tool (see Figure 11). We glued this rack on the elevated part of the rack casing. To control every

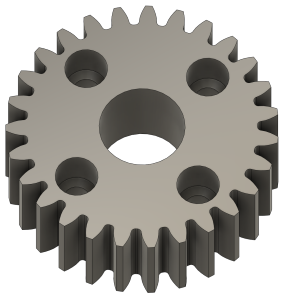


(a)



(b)

Figure 10: a) shows the 3D model of both the rack as well as the motor casing. b) shows the custom 3D prints attached to the arm. On the motor casing, we mounted the Dynamixel with the custom pinion and the OpenRB-150.



(a)

(b)

Figure 11: a) shows the 3D model of the pinion. b) shows the custom 3D print of the pinion using PLA filament. Around it, one can see the flexible rack.

part of this prototype, we use an OpenRB-150 microcontroller. This is attached to the side of the motor casing using screws and nuts. Moreover, we placed the base sensor casing on top of the motor. Additionally, we placed a protective layer of felt underneath the rack casing to increase the comfort of wearing the prototype and protect the user from any sharp edges.

3.4 Algorithm to Fix the Drift

Until now, we are able to measure the value of the drift and we can rotate the base on the z-axis of the arm. In this last step, we propose an algorithm to fix the drift on the z-axis. Moreover, we discuss further calculations and algorithms to solve the same problem on multiple axes.

One Dimensional. As we already covered the calculations for the one dimensional drift we can directly use this value in our fixing algorithm.

Method 1. The most simple approach is to rotate the robotic limb in the counter direction when drift occurs to compensate for this error until the error is small enough. This can be implemented without any further parameters:

Listing 1: Drift Fix

```
/// <summary>
/// Loop every frame
/// </summary>
private void Update()
{
    //Current error
    var alpha = GetAngularError();

    //Drift small enough?
    if(Mathf.Abs(alpha) < threshold)
        return;

    if (alpha > 0)
    {
        //Rotate left
    }
    else
    {
        //Rotate right
    }
}
```

This approach is quick and simple, but a large drawback is that overshooting can happen. If the update rate is low and the rack and pinion move fast, it might happen that we just move the error from one side to the other.

Example:

$$
\begin{aligned}
&\textit{Base moves } 10^\circ/\textit{frame}\\
&\textit{Threshold} = 3^\circ\\
&\alpha = 5^\circ\\
&\Rightarrow \alpha' = 5^\circ - 10^\circ = -5^\circ\\
&\Rightarrow \textit{Never stops overshooting}
\end{aligned}
$$

Hence, we might encounter precision problems with method 1). But as our base movement is not that fast, we still chose this method because of its quick implementation.

Method 2. Beside the first method one can also map the angular error directly to movement on the rack. This can be done by the following function:

Listing 2: Angle Mapping

```
/// <summary>
/// Transforms angular error to movement on the rack
/// </summary>
/// alpha: The angular error
/// angleRange: The range of the rack in degrees
/// rackLen: The length of the rack in cm
/// returns: The length we have to move in cm
private float MapAngleToMovement(float alpha, float
    angleRange, float rackLen){
    float lenPerAngle = rackLen/angleRange;
    return alpha * lenPerAngle;
}
```

The advantage of this method over the first approach is that there won't be any overshooting, because we only move the exactly needed amount. But this comes with the drawback that one must know the correct measurements for the *angleRange* and the *rackLen*. Those values must be precise and if we allow the rack and pinion to change the diameter, for different arm sizes, the angle range will also be inconsistent among various users.

Two Dimensional.
Theoretically, one can extend our approach to more than one axis, under the assumption that there is a physical device that moves the base in more than one axis as well. However, we cannot copy the calculations for the z-axis to the x-axis directly. The reason for this is that a rotation around an axis transforms a point p as follows:

- z-Axis:

$$\begin{vmatrix} cos(\theta) & -sin(\theta) & 0 \\ sin(\theta) & cos(\theta) & 0 \\ 0 & 0 & 1 \end{vmatrix} \cdot \begin{vmatrix} x \\ y \\ z \end{vmatrix} = \begin{vmatrix} x \cdot cos(\theta) - y \cdot sin(\theta) \\ x \cdot sin(\theta) + y \cdot cos(\theta) \\ z \end{vmatrix} = \begin{vmatrix} x' \\ y' \\ z' \end{vmatrix}$$

- x-Axis:

$$\begin{vmatrix} 1 & 0 & 0 \\ 0 & cos(\theta) & -sin(\theta) \\ 0 & sin(\theta) & cos(\theta) \end{vmatrix} \cdot \begin{vmatrix} x \\ y \\ z \end{vmatrix} = \begin{vmatrix} x \\ y \cdot cos(\theta) - z \cdot sin(\theta) \\ y \cdot sin(\theta) + z \cdot cos(\theta) \end{vmatrix} = \begin{vmatrix} x' \\ y' \\ z' \end{vmatrix}$$

- Y-Axis:

$$\begin{vmatrix} cos(\theta) & 0 & sin(\theta) \\ 0 & 1 & 0 \\ -sin(\theta) & 0 & cos(\theta) \end{vmatrix} \cdot \begin{vmatrix} x \\ y \\ z \end{vmatrix} = \begin{vmatrix} x \cdot cos(\theta) + z \cdot sin(\theta) \\ y \\ -x \cdot sin(\theta) + z \cdot cos(\theta) \end{vmatrix} = \begin{vmatrix} x' \\ y' \\ z' \end{vmatrix}$$

Looking at the rotation matrices one can see that the rotation along one axis does influence both values of the other axes. Hence, if we calculate our angular errors α & β the same way on both axes then the resulting combined rotation will be wrong because the rotation of α influences β and the other way around.

Example:

- **Current drift:** EulerAngles(20°, 0, 45°), with an z-x-y order (Figure 12)
- **Resulting x-axis fix value (β):** -20° (Because example input is in euler angles and x gets applied after z)
- **Resulting z-axis fix value (α):** - 46.78° (z-value influenced by x rotation)
- Applying β aligns onto the x-axis (Figure 13a)
- Applying α aligns onto the z-axis (Figure 13b)
- But applying both (α & β) results in an overshoot on the second applied axis (Figure 13c)

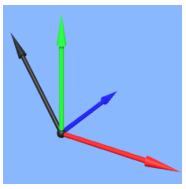
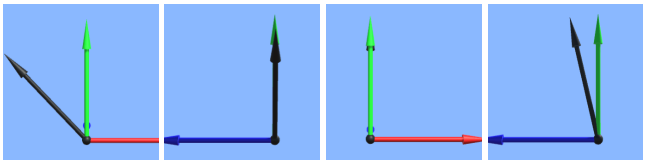
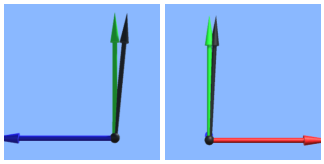


Figure 12: 3D-view: Black arrow shows drift



(a) β applied (Front & side view) **(b) α applied (Front & side view)**



(c) α & β vs. β & α applied: ⇒ Overshoot on x or z-axis

Figure 13: Example why we need to calculate β differently

As a result, we adjust the calculations for the second angle β:

$$r_1^{-1} = \frac{r_1^*}{\|r_1\|^2} \tag{1}$$

$$r_d = r_1^{-1} \cdot r_2 \tag{2}$$

$$r_{adjD} = [r_{d_w}, -r_{d_x}, r_{d_z}, r_{d_y}] \tag{3}$$

$$r_v = [0, \vec{v}_x, \vec{v}_y, \vec{v}_z] = [0, 0, 1, 0] \tag{4}$$

$$\vec{v}_{rot} = r_{adjD} \cdot r_v \cdot r_{adjD}^* \tag{5}$$

$$\vec{v}_{proj} = [\vec{v}_{rot_x}, \vec{v}_{rot_y}, 0] \tag{6}$$

$$\theta = acos(\frac{\vec{v}_{proj} \cdot \vec{v}_{up}}{|\vec{v}_{proj}| \cdot |\vec{v}_{up}|}) \tag{7}$$

$$\alpha = \theta \cdot \frac{180}{\pi} \tag{8}$$

$$\vec{v'}_{rot} = \begin{vmatrix} cos(\theta) & -sin(\theta) & 0 \\ sin(\theta) & cos(\theta) & 0 \\ 0 & 0 & 1 \end{vmatrix} \cdot \vec{v}_{rot} \tag{9}$$

$$\theta = acos(\frac{\vec{v'}_{rot} \cdot \vec{v}_{up}}{|\vec{v'}_{rot}| \cdot |\vec{v}_{up}|}) \tag{10}$$

$$\beta = \theta \cdot \frac{180}{\pi} \tag{11}$$

Steps (1) - (8) are the previous calculations for α. Afterward, we rotate by α around the z-axis (9), to simulate the fix that we apply on that axis. Hence, we can calculate β from this $\vec{v'}_{rot}$ (10) & (11), as it accounts for α, and overshooting won't happen anymore. Note that we must not make a projection as previously for α, because after applying α, the x-value is already 0. These calculations can also be swapped around, depending on the axis that we handle first. One could also begin by fixing the x-axis with our initial calculations and afterward use the second calculation to receive the error on the z-axis. But in that case, one must swap the rotation matrices to account for the right axis.

4 VALIDATION OF THE SYSTEM

It is important to validate the results coming from this prototype. Therefore, we conducted internal tests to verify the functionality of our prototype.

Testing Setup

In order to test the functionality of the algorithm combined with the prototype we added weights onto the horn of a second Dynamixel Servo Motor. With this, we were able to simulate the force of the movement of an attached robotic limb. Since we wanted to control the second Dynamixel using the Dynamixel Wizard, it was not possible to connect the motor to the OpenRB-150 as well. Instead, we connected this motor to a U2D2 and the according power hub. That way we could use the motor with the Wizard and simply rotate the weights that are mounted on the motor to create skin drift and simulate a test case for our prototype. To make this board wearable with the rest of the prototype, we glued it on top of a strap that can be worn around the arm. See Figure 1 for the fully mounted prototype and testing setup.

Results

In our tests, we detected that the velocity, with which the motor turns, can be improved in order to fix the skin drift more efficiently. In the scope of this project, we did not increase the velocity beyond 30, since we did not want to break the prototype. But we are confident that it is able to endure higher velocities. Although the prototype worked as intended, there needs to be more evaluation on how accurately the prototype fixes the drift in future work.

5 LIMITATIONS & FUTURE WORK

Although the prototype works, there are still problems with our approach and improvements that one should realize in future work.

5.1 Concept

The first drawback of this concept is that we try to fix a value that continuously changes. While the drift fix gets applied, meaning that the base moves, one can move the arm and therefore induce even more drift. On the other side, the opposite drift can negate the current drift. Hence, one needs quick update rates such that there is no delay in the rack movement. Moreover, another drawback is that the rack movement during the fix might even create drift by itself.

5.2 Sensing

Firstly, one should use sensors with a north reference in future work, such that the orientation of the sensors during calibration does not matter. This reduces the complexity and the error that arises from misaligned sensors. Furthermore, one should investigate the positioning of the sensors further. The current positions are not optimal and therefore are influenced by drift. An additional structure to mount the reference sensor and a user study should improve this.

5.3 Circuit

Additionally, one should fabricate printed circuit boards (PCBs) and use soldered wires. This prevents the mess with all the jumper wires and reduces the form factor of the prototype.

5.4 Fix the drift

In future work, one should definitely use the angle mapping approach over the threshold drift fix, especially for the drift fix on multiple axes. This approach is more precise and cannot be stuck in an infinite loop. However, it would be optimal to include a way to retrieve the measurements for *angleRange* and *rackLen* faster.

5.5 Prototype

One limitation of the prototype is the range of movement on the arm which can be extended in future work. At the moment the rack supports a range of 40°, thus it is possible to fix 20° of drift in both directions. Furthermore, the diameter of the curve should be optimized in such a way that it can fit more arm sizes better. One could for example look into using flexible materials for 3D printing such as TPU. Additionally, the current attachment of the whole prototype using elastic straps is sometimes too loose and should be improved in the future. Lastly, the velocity of the motor countering the drift is currently set to a low value, since we did not want to break our structure during the demo phase due to the velocity being too high. In the future, the velocity needs to be tested and adjusted accordingly. Overall, the comfort of wearing the prototype should also be improved since the current system is very bulky and quite heavy on top of the arm. Lastly, it would also be of interest to build a prototype that is able to fix drift on 2 axes, as we already proposed the calculations to implement this. However, this prototype will be more complex to realize.

6 CONCLUSION

Concluding this work, we have developed an algorithm for sensing skin drift and built a base structure that is able to move and fix the position of an attached robotic limb. In the end, we have assembled the base structure and sensors into a prototype that is capable of detecting and countering skin drift.
While our prototype seems to be promising in order to fulfill the purpose of fixing skin drift in the field of wearable robotics, one should conduct more evaluation and testing on how accurate this method is.

REFERENCES

[1] Sang-won Leigh, Harshit Agrawal, and Pattie Maes. 2018. Robotic symbionts: Interweaving human and machine actions. *IEEE Pervasive Computing* 17, 2 (2018), 34–43. https://doi.org/10.1109/MPRV.2018.022511241

[2] Azumi Maekawa, Shota Takahashi, MHD Yamen Saraiji, Sohei Wakisaka, Hiroyasu Iwata, and Masahiko Inami. 2019. Naviarm: Augmenting the Learning of Motor Skills Using a Backpack-Type Robotic Arm System. In *Proceedings of the 10th Augmented Human International Conference 2019* (Reims, France) *(AH2019)*. Association for Computing Machinery, New York, NY, USA, Article 38, 8 pages. https://doi.org/10.1145/3311823.3311849

[3] Vighnesh Vatsal and Guy Hoffman. 2017. Wearing your arm on your sleeve: Studying usage contexts for a wearable robotic forearm. In *2017 26th IEEE International Symposium on Robot and Human Interactive Communication (RO-MAN)*. 974–980. https://doi.org/10.1109/ROMAN.2017.8172421

[4] Vighnesh Vatsal and Guy Hoffman. 2019. End-Effector Stabilization of a Wearable Robotic Arm Using Time Series Modeling of Human Disturbances *(Dynamic Systems and Control Conference)*. V001T05A001. https://doi.org/10.1115/DSCC2019-8985

[5] Shigeo Yoshida, Tomoya Sasaki, Zendai Kashino, and Masahiko Inami. 2023. TOMURA: A Mountable Hand-Shaped Interface for Versatile Interactions. In *Proceedings of the Augmented Humans International Conference 2023* (Glasgow, United Kingdom) *(AHs '23)*. Association for Computing Machinery, New York, NY, USA, 243–254. https://doi.org/10.1145/3582700.3582719

[6] Andre Zenner and Antonio Krüger. 2017. Shifty: A weight-shifting dynamic passive haptic proxy to enhance object perception in virtual reality. *IEEE Transactions on Visualization and Computer Graphics* 23, 4 (2017), 1285–1294. https://doi.org/10.1109/TVCG.2017.2656978

[7] Andre Zenner and Antonio Krüger. 2021. Supernumerary Robotic Limbs: a Review and Future Outlook. *IEEE Transactions on Medical Robotics and Bionics* 3, 3 (2021), 623–639. https://doi.org/10.1109/TMRB.2021.3086016

A APPENDIX

A.1 Contribution & Acknowledgement

All members of this project contributed equally to the project. As a group, we refined the idea and concept of this project. Sophie Kunz and Anna Calmbach built the prototype, while David Wagmann implemented the sensing and processing part. Moreover, we want to thank the HCI group and especially Dr. Ata Otaran for their support and help with this project.

A.2 Syntax

To make it easier to differentiate, we write the magnitude of a 3D space vector $\vec{v}$ as $|\vec{v}|$ and the magnitude of a quaternion vector q as $\|q\|$.

A.3 Hamilton Number Rules

A hamiltonian number of the form $x_0 + x_1 \cdot i + x_2 \cdot j + x_3 \cdot k$ matches following rules:

- $i^2 = j^2 = k^2 = -1$
- $ij = k \wedge ji = -k$
- $jk = i \wedge jk = -i$
- $ki = j \wedge ik = -j$

A.4 Quaternion Calculations

In the following, we give the exact formulas for the calculations with quaternions that we used in our approach.

Multiplication.
Given $r_1 = [w_1, x_1, y_1, z_1]$ & $r_2 = [w_2, x_2, y_2, z_2]$:

$$
\begin{aligned}
r_1 \cdot r_2 = [& \\
& w_1 \cdot w_2 - x_1 \cdot x_2 - y_1 \cdot y_2 - z_1 \cdot z_2, \\
& w_1 \cdot x_2 + x_1 \cdot w_2 + y_1 \cdot z_2 - z_1 \cdot y_2, \\
& w_1 \cdot y_2 - x_1 \cdot z_2 + y_1 \cdot w_2 + z_1 \cdot x_2, \\
& w_1 \cdot z_2 + x_1 \cdot y_2 - y_1 \cdot x_2 + z_1 \cdot w_2]
\end{aligned}
$$

Magnitude & Squared Magnitude.
Given $r = [w, x, y, z]$:
$\|r\| = \sqrt{(w^2 + x^2 + y^2 + z^2)}$
$\|r\|^2 = (w^2 + x^2 + y^2 + z^2)$

Normalize.
Given $r = [w, x, y, z]$ & $\|r\|$ the magnitude:
$r_n = [\frac{w}{\|r\|}, \frac{x}{\|r\|}, \frac{y}{\|r\|}, \frac{z}{\|r\|}] \leftrightarrow \|r_n\| = 1$

Conjugate.
Given $r = [w, x, y, z]$:
$r^* = [w, -x, -y, -z]$

Inverse.
Given $r = [w, x, y, z]$:
$r_1^{-1} = \frac{r_1^*}{\|r_1\|^2}$

A.5 Unit Quaternion

A unit quaternion has a magnitude of 1:
$\|r\| = \sqrt{(w^2 + x^2 + y^2 + z^2)} = 1$

A.6 Pure Quaternion

A pure quaternion has a scalar part equal to 0. For example [0, 0.5, 0.25, 0.25].

A.7 GitHub

The code of all Arduino sketches, such as the 3D files and the visualization demo can all be accessed at the following GitHub repository: https://github.com/DavidW2211/Seminar-Robotic

A.8 Do It Yourself

To recreate this work, one must follow these steps. The materials (prints, sketches, ...) can be accessed at GitHub (Appendix A.7):

- 3D print all stl. files of the prototype
- Glue this rack (Link) onto the rack casing or make according changes to your pinion using the *SpurGear* script in AutoDesk Fiuson
- Assemble the 3D prints
- Install the Dynamixel library (Link)
- Load the MotorConfiguration sketch and configure the motor with it (Let it run until the motor stops moving)
- Add the motor to the prototype
- Build the sensor circuit

- Add the MPU6050 plugin to the libraries folder
- Load the I2C_Scanner sketch and write down the I^2C addresses of the sensors
- Insert the previous addresses into the DualSensor sketch (Configuration at top)
- Load the DualSensor sketch onto the microcontroller
- Open the serial monitor. If there is no text or it says "failed" then check all the cables and restart the application. Otherwise, you must press any key to start initialization. Note that the sensors must be aligned during calibration!

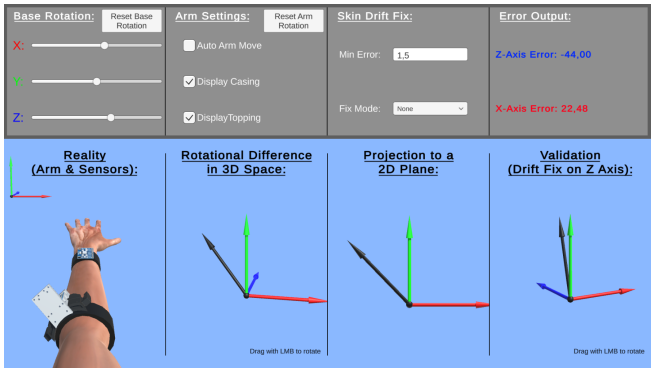
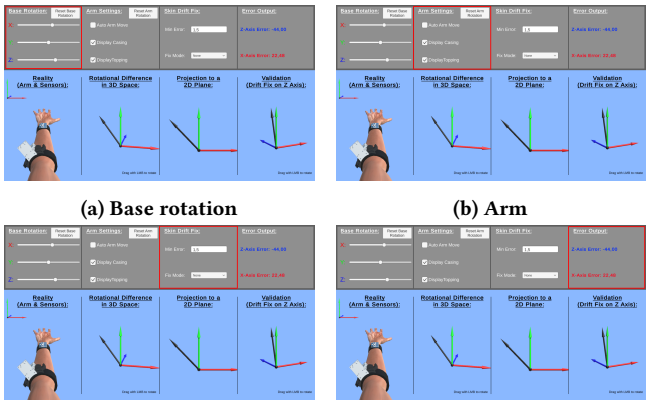


Figure 14: Visualization editor tool

A.9 Visualization Demo

In order to better understand the steps of our prototype (Sensing, processing, fixing) we implemented a visualization demo made in Unity-3D (Figure 14). The demo consists of 2 main parts. In the top part, one can adjust settings, like visuals or the rotation of the robotic limb's base structure on the arm. At the same time, in the lower part, one can see the immediate effect of these settings. In the following, we will go into both parts in more detail.

A.9.1 Settings.
Firstly, one can adjust the rotation of the base structure where the sensor and the robotic limb are mounted (Figure 15a). This simulates the skin drift that happens at the moment. The input is in Euler angles and one can also reset the rotation with a button. The second section is for controlling the arm rotation and visuals (Figure 15b). One can disable the visuals of the casing to see the integrated second sensor. Furthermore, auto-arm movement turns on random movements of the arm. This demonstrates that the arm movement does not influence the measurements. Next are the settings for the skin drift fix (Figure 15c). The min error defines which error still gets tolerated. If the error exceeds the threshold the automatic drift fix happens. The user can select whether the fix should only be applied to one axis (z), or to two axes (x-z). In the last window, we display the measured error on the z and x-axis in degrees (Figure 15d).

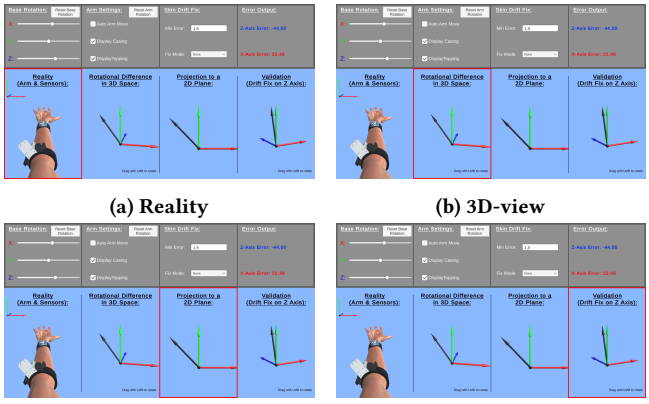


(a) Base rotation **(b) Arm**

(c) Drift fix **(d) Error output**

Figure 15: Visualization settings

A.9.2 Explanation. The left demo shows the arm with both sensors and the applied drift of the base (Figure 16a). This describes the users' view, that one would see in reality. In the next step, show the sensing step where we transform the sensor rotation difference to 3D space (Figure 16b). The third section displays the internal processing step. Here the previous data gets mapped onto a 2D-plane (Figure 16c). Lastly, the fourth window shows the validation, meaning that we apply the calculated z-error to the previous 3D-space vector (Figure 16d). The result will always show that the vector is aligned on the z-axis and thus the error of the drift is removed.



(a) Reality **(b) 3D-view**

(c) 2D-projection **(d) Validation**

Figure 16: Visualization explanation

A.10 Prototype Concept Sketches

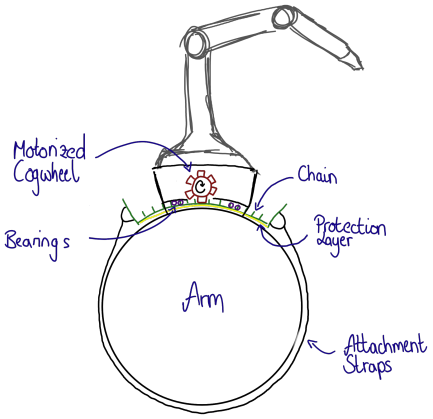


Figure 17: A sketch of the refined concept.

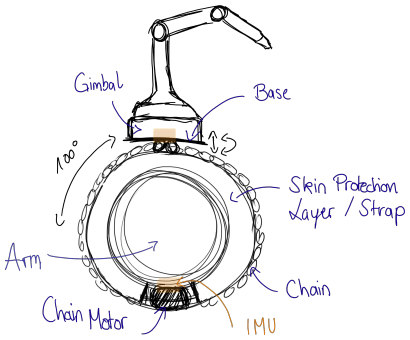


Figure 18: A sketch of the belt and gimbal approach, which was one of our first concept ideas.

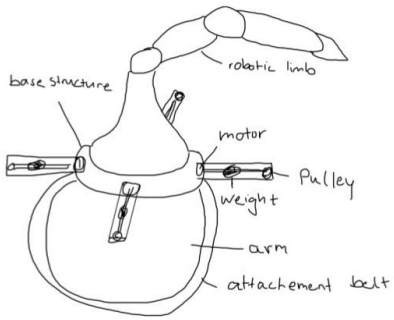


Figure 19: A sketch of the counterweights concept, which was discarded for our project.

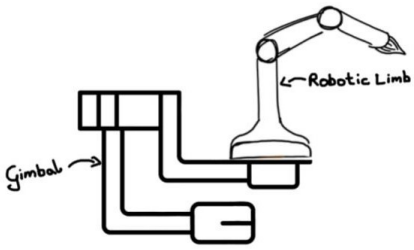


Figure 20: A sketch of the gimbal concept, which was discarded for our project.

submitted 27 of September 2013