DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics: Games Engineering

PID-Tuning Framework for Remotely Operated Humanoid Robots

Markus Webel

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics: Games Engineering

PID-Tuning Framework for Remotely Operated Humanoid Robots

PID-Tuning Framework für ferngesteuerte humanoide Roboter

Author:	Markus Webel
Supervisor:	Prof. Gudrun Klinker, Ph.D.
Advisor:	M.Sc. Sandro Weber
Submission Date:	15.12.2019

I confirm that this master's thesis in informatics: games engineering is my own work and I have documented all sources and material used.

Munich, Markus Webel

Abstract

Remotely controlled humanoid robots have countless applications in both industry and science. They can be effectively controlled by making them recreate the pose of a motion-tracked human operator in real-time. The robots often have industrial closed-loop controllers in their joints to perform their movements. PID controllers are among the most common controllers, but they are notoriously hard to tune. This thesis aims to provide tools to assist with PID tuning for both experienced and inexperienced users. It focuses on the specific use-case of remote operation of humanoid robots. To achieve this, a tuning framework consisting of tools, APIs, and UIs is developed. It integrates into an existing remote operation solution, which allows operators to control a virtual, simulated humanoid robot. The tuning framework facilitates both manual and automatic PID tuning. It provides functionality for data collection, motion playback, and performance evaluation. The automatic tuning feature offers multiple heuristics, which are tested on four different robot setups. Their performance is compared to a manual reference tuning. The top heuristics for each setup are identified based on multiple performance metrics. The thesis concludes that the manual tuning generally outperforms all heuristics of the automatic tuning routine. The development of new, purpose-built heuristics for the specific use-case is suggested. Alternative methods of robot control and PID tuning are proposed in the outlook.

Contents

1 Introduction

1.1 Background

Robots have found their way into countless industrial branches, with the automotive and electronics industries as the biggest customers. Their precision, speed, strength, and reliability provides a great increase in efficiency over an entirely human workforce. A key to their success is the very controlled environment in which they are employed: A robot in an automotive factory can, for example, attach hundreds of car doors every day, but it can do so only because its task is completely repetitive; Each door is exactly the same as the last one, and they all go onto the same model of car. Only this high degree of specialization allows the robot to act autonomously, without constant human intervention.

While the automotive sector allows for this level of specialization, other work environments cannot guarantee such predictable workloads. Surgical robots are typically at most semi-autonomous and operate under constant human guidance. This flexibility is necessary due to the uncertain nature of surgical procedures and due to safety concerns over full robot autonomy. But even surgical robots are highly specialized tools; It would be impossible to use a surgical robot for anything other than surgery. Such a narrow specialization is not an exception: General purpose robots are still a rarity.

Humanoid robots have been developed to address this issue. Since every structure and appliance in day-to-day life is built for human interaction, a sufficiently humanoid robot could potentially interact with its environment just like a real person. A common bomb disposal robot, for example, will happily manipulate the inner workings of a complicated explosive, but will be defeated by a closed door or a flight of stairs. In contrast, bipedal locomotion allows robots to traverse uneven terrain and climb staircases. Articulated upper limbs allow robots to pick up, carry, and manipulate objects, for example door handles or the steering wheel of a car.

Despite being humanoid in shape, humanoid robots still retain many of their robotic advantages. They will not experience fatigue or fear and will operate in many environments which are deemed unsafe for human presence. Examples include toxic waste removal, emergency rescue operations, disposal of explosives, and work in zero-atmosphere environments. Robots might also be equipped with superior strength and dexterity compared to a human worker and can be instructed to perform autonomous tasks with high precision. But all this flexibility comes at a price; Humanoid robots are notoriously hard to control. Although the act of walking from A to B might seem trivial for a human, issuing the exact motor commands to all actuators within the robot joints requires complex orchestration. Traditional control schemes like joysticks do not perform well with something as complex as a humanoid robot.

A possible solution for this control problem is remote operation of the robot via human body tracking. Due to their lifelong experience with their own body, human operators require no mental mapping of input to output, given that the robot is able to replicate their pose sufficiently well. Researchers at the Technical University of Munich (TUM) have created a remote operation framework which allows a human operator to control a simulated robot with their body. Tracking technology and inverse kinematics are used to accurately track human limb position and orientation with relatively inexpensive equipment. This pose-data is then relayed to a robot, which replicates the pose as accurately as possible. VR technology is used to further improve the immersion by letting the human operator perceive the world through the "eyes" of the robot.

Arguably the most important part of remote operation is that the robot is able to replicate the operator's pose quickly and accurately. This requires all joints of the robot to receive precise control commands with low latency. Furthermore, the robot needs to be able to deal with obstacles and disturbances without deviating too far from the pose dictated by the operator. Any delays or inaccuracies can have a major detrimental impact on task performance and user experience. To achieve these requirements, many robots utilize closed-loop control systems within their joints, with PID controllers being a popular choice. These controllers need to be carefully tuned to assure the fast and accurate response that is expected for remote operation. The tuning process usually requires in-depth knowledge about the inner logic of the joint controllers, physical properties of the robot, and work environment.

The main objective of this thesis is to make the process of PID tuning more accessible for researchers. This is achieved by developing a tuning framework, which is completely integrated within the remote operation framework from TUM. This tuning framework consists of a collection of tools, APIs, and guidelines for both manual and automatic PID tuning. It provides facilities to test and compare different PID tunings, and to export and share test results among researches.

1.2 Problem Description

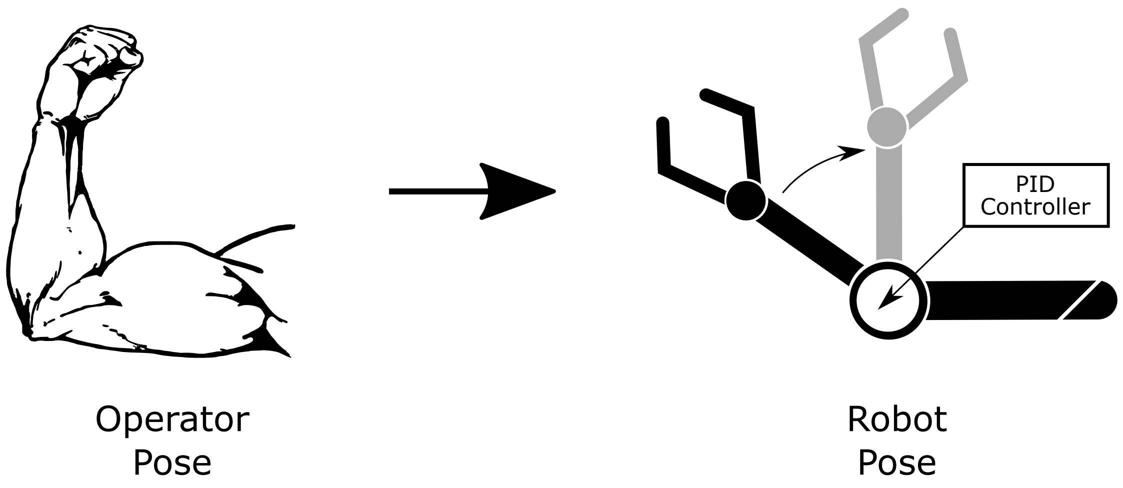


Figure 1.1: PID controller in the robot's right elbow trying to match the operator's pose

Given a target value, also known as set-point, a PID controller attempts to output the right control commands in order to reach this target. An illustration of the process can be seen in Figure 1.1. In this case, the target value is an angle of 90 degrees in the elbow joint. Since the current angle of the robot arm is below 90 degrees, the PID controller applies force to rotate the joint towards it. Well-configured PID controllers will always reach their exact target in a finite amount of time.

PID controllers belong to the family of closed-loop controllers. In contrast to open-loop controllers, they continuously measure their current deviation from the target and act accordingly. This makes it possible for closed-loop controllers to deal with disturbances and obstacles. They are usually used when the environment is not entirely predictable; Turning the elbow joint might not always require the exact same amount of force, depending on the external influences on the robot. The robustness of PID controllers with regard to disturbances comes with a price: They need to be carefully configured to be effective, since misconfigured controllers can lead to a wide array of issues and even damage the robot. The act of finding the right configuration for a PID controller is called *PID tuning*.

PID controllers have three configurable parameters, often called *gains*: K_p, K_i, and K_d. A combination of these gain parameters is also called *(PID) tuning*. Each of the parameters has a different effect on the output of the controller. The complexity of PID tuning stems from the fact that the performance of a controller relies on a good combination of all three parameters; Tuning a single parameter, or just two of them, is often not enough. To find a good tuning, it is necessary to be intimately familiar with the effect of each gain parameter. There is a good reason why PID tuning has often been called an art instead of a science.

A good performance of all joint PID controllers is essential for a successful remote operation solution for a humanoid robot. If the robot cannot replicate the pose of the operator fast enough, task efficiency is reduced drastically. The operator would then be unable to perform tasks like they normally would, defeating the purpose of sending a humanoid robot. On the other hand, it is also possible for the robot to replicate the operator's pose too aggressively: If the robot overshoots the target pose, it can potentially damage itself or its environment. Avoiding these two scenarios is hard, since they are at odds with each other: A very tentative PID tuning will avoid overshoot and safely arrive at the target angle, but will take a long time to do so. An overly aggressive tuning, in contrast, will lead to dangerous oscillations, but will reach the target angle much faster. Both options have advantages and disadvantages. PID tuning is the act of carefully balancing trade-offs until a satisfactory middle ground is reached. Being aware of all trade-offs is not easy, especially because there is usually no objective measure as to how to the robot should perform exactly.

The lack of an objectively 'best tuning' adds even more complexity to the tuning process. While a slightly over-aggressive tuning is probably fine for a robot deployed on a construction site, it would be completely unsafe to use it for a surgical robot. In such an environment, a slow, low-force tuning is much preferred. But not only the robot's environment influences the tuning process: The operator also needs to feel comfortable with the robot's performance. Some operators might be able to anticipate the motion of the robot more quickly, and can apply preemptive corrective adjustments to their pose; Other operators might be overwhelmed by the task. Such considerations add additional complexity to the tuning process.

Since a good PID tuning is such a subjective issue, it is hard to create quantifiable performance goals for the robot. Usually, a human operator needs to be present for the process of PID tuning. Still, the human operator will not be able to repeat their motions accurately during the tuning process. This makes it hard to compare different tuning in a meaningful way: The operator might simply prefer one tuning over the other because they slightly changed their pose between the two test runs. Human operators might also have individual preferences regarding a good PID tuning, which might make it incompatible with other operators. It can be hard to quantify which tunings are suitable for every operator, and which ones are too specialized. Creating a different tuning for each operator is often not practical.

The final difficulty with finding a good PID tuning for a humanoid robot is that it can take a significant amount of time: Each movable joint of the robot has a separate PID controller, which needs to be tuned independently of all other controllers[1]. The different physical properties of all joints make it impossible to re-use a tuning. Using the same controller at the base of an index finger and inside one of the knees is obviously a bad choice for at least one of them, since the force requirements are vastly different. The weight distribution of the limbs which are attached to the joints present a further a complication: The left shoulder joint needs to compensate for the movement of the entire left arm, including the joints in the left elbow and hand. Such non-linear systems are notoriously hard to control with PID controllers [Kan+14].

1.3 Contribution

All the complications mentioned in the last section make PID tuning seem tedious, but it is a necessary evil to make remotely operated humanoid robots perform effectively. The process can be vastly improved by using automation, or at least by giving easy-to-follow instructions on how to achieve a good tuning. It is also of great benefit to have a way of performing accurate, reproducible tests without having to involve a human operator. Most importantly, quickly finding a good PID tuning allows researchers to spend their time on other topics involving remotely controlled humanoid robots. These are the main motivations behind the contribution of this thesis.

This thesis introduces a collection of tools, APIs, UIs, and algorithms related to PID tuning, which will be called *tuning framework* from this point on. It is completely integrated into the remote operation framework that was developed previously at TUM (described in chapter 2). Both manual and automatic tuning of joint controllers is explicitly supported. This section will outline the main capabilities of the tuning framework, with a more detailed explanation available in chapter 3.

Reproducible tests are an important aspect of effective PID tuning. As mentioned before, human operators often will not be capable of accurately repeating motion sequences multiple times. It is also impractical to require a them to be present during the whole tuning procedure. Instead, the tuning framework allows users to replay pre-recorded realistic motion sequences as if a human operator was performing them. This makes it possible to compare different PID tunings on their performance without having to worry about external factors.

The tuning frameworks allows users to record test data about the performance of all PID controllers within the robot in real-time. This recording function is not limited to the motion playback function, but can also record the motions of a human operator. The recording routine is designed to be easily extensible by custom data, allowing researchers to add their own performance measurements with little effort. All data

[1]Symmetric joints (left and right half of the robot) can use the same tuning, however.

that is collected in a recording can be exported to JSON files to be shared with other researchers, or to be used in external software. The PID tuning of all joints can also be exported and imported by the framework, making it possible to share tunings between robot models, as long as they are sufficiently similar.

The framework also provides facilities to analyze the data that was collected during a test run. It includes numerous performance metrics, which can give important clues about the effectiveness of the current PID tuning. These performance metrics also make it possible to define objective targets for the tuning process. This avoids having to resort to numerous human operators for testing and rating every single PID tuning. Performance metrics can exported just like test data. They are also exceptionally useful for iterative manual PID tuning, since the effect of every parameter change on controller performance can be instantly quantified. This helps to guide the user into the right direction and lets them arrive at a satisfactory tuning much faster.

Manual tuning is facilitated by the framework through an extensive user interface (UI). The UI provides visual tools that allow the user to quickly estimate the performance of a PID tuning. The centerpiece of the UI is a real-time graph showing the performance of a controller over time. Every joint PID within the robot can be displayed in this graph. The UI also allows the user to start and stop recording sessions, or to play back pre-recorded motion sequences to the remote robot. It also provides a quick way to change the tuning of any joint, making it possible to iteratively tune PID controllers without any downtime.

The framework also offers automatic PID tuning, where no user interaction at all is required. First, an automated test run measures the response characteristics of a joint. Then, a suitable tuning is calculated via one of the multiple available heuristics. If the user is unhappy with the tuning, they can compare different heuristics until they are satisfied. A special UI was created to make this task more efficient. The automatic tuning routine is especially useful for users that have no experience at all with PID tuning.

The entire framework is designed to be extensible and modular. Future researches are able to re-use parts of the implementation via the well-documented public API. The framework makes its entire functionality available through code; The UI is only a convenient wrapper around the public API. All test and performance data can be exported in the widely used JSON format, which is human-readable and has extensive third-party support.

1.4 Related Work

The concept of remote human presence via robotic surrogates was first proposed in a scientific context by Dr. Susumu Tachi in 1980[2] [Tac+12], who referred to it as *Telexistence*. His definition of the concept was very broad, and allowed for either party (the operator and the robot) to exist in either the real world, or a virtual environment. This means that Telexistence encompasses the use case that is covered by this thesis: An operator in a real-world environment controls a robot within a simulation. This broad definition sets Telexistence apart from *Telepresence*, a concept developed in the USA. Telepresence only encompasses use cases where both operator and surrogate exist in the real world.

In 2001, a cockpit was developed from which a human operator could control a humanoid robot [Tac+03]. The operator has to chose between a simple directional walking mode and direct control mode. In the latter mode, the arms and hands of the robot directly recreate the pose of the human operator. The contribution is especially notable due to its many methods of feedback from the robot to the operator: Visual and audio information was provided via a CAVE-like setup[3], and force-feedback was integrated into the arm-tracking device.

In 2005, the concept of Telexistence was explored again in the form a robotic master-slave arm [Tad+05]. The contribution is notable because of the great care that was taken that the force feedback of the arm was as accurate as possible, and that the robotic arm also applied the same force as the operator. This is a great advantage whenever safe interaction between robot and humans is required. To achieve force equality, the robot arm was controlled via impedance control, a form of force control, in contrast to the more primitive PID control. The robot arm was equipped with multiple force sensors to improve force accuracy.

In 2008, Watanabe et al. introduced *TORSO* [Wat+08]. This system was built to address the lack of neck motion in previous robots. Since humans not only use the rotation of their head, but also its translation, an accurate tracking method for neck movement was developed. A camera was then placed on top of a "robotic neck" that could replicate this kind of motion. Before this method was utilized, the controlled robots would usually move their entire body to compensate for head translation.

In 2012, the *TELESAR V* robot was introduced by Fernando et al. [Fer+12]. This contribution put large emphasizes on the "expansion of his [the operator's] bodily consciousness" into the slave robot. They argue that such a high level of immersion increases the throughput of the operator and thus enables them to finish tasks more quickly. TELESAR V is a anthropomorphic robot that replicates upper body movements

[2]The first publication of Telexistence occurred two years later, in 1982.
[3]VR setup using projectors and room-sized cube [Cru+92]

with high accuracy (52 DOF). It's vision is transmitted in stereo to a VR HMD. The robot is notable for its various forms of feedback: The finger-tips contain haptic and thermal sensors and their measurements are replicated on the finger-tips of the operator. They also discover some requirements for high immersion, and create a connection to psychological experiments like the Rubber Hand Illusion[4].

Pollard et al. contributed an offline approach to human control, meaning that human motion is pre-recorded via MoCap and then later replayed by a robot [Pol+02]. It investigated the much tighter restriction on limb motion in humanoid robots at the time, and developed a method to post-process MoCap data to make it compatible with the robot. They discover issues with pose replication at certain joint angle configurations and provide a workaround. An objective error measure for pose deviations is also introduced in their paper, and is used later in this thesis.

As a midway point between the offline approach and a completely online (real-time) control approach, there have been several contributions which require a learning phase before effective control is possible. Bell et al. developed a control scheme for a semi-autonomous robot using a non-invasive brain-computer interface [Bel+08]. The control is done at a very abstract level, where the brain interface is used to select one of multiple movement choices, which the robot then carries out. They mention that the rate of information that the interface can carry is not yet high enough to facilitate more advanced control schemes. A learning phase of three minutes is determined to be sufficient.

Stanton et al. also require a learning phase of about three minutes for their approach, but use a very unusual method of control [SBR12]. By letting the robot perform poses that the user has to replicate, they train a neural network to generate a mapping between operator pose and robot pose. After this training phase, the robot can then be controlled in real-time with a mean accuracy error of only 5.55%. A major advantage of this approach is that no explicit mathematical model is needed to map between the operator pose and the robot pose, meaning that it can be applied to any model of humanoid robot and any operator. It is noteworthy that their robot has a fairly low number of DOF (16).

Koenemann et al. propose to focus tracking on end-effector positions, meaning hands and feet [KBB14]. The robot can then replicate poses in a manner that is more suitable to its physique, since the position of unimportant joints like elbows can be ignored to some degree. Their work is notable in that they can operate even in single-support mode, meaning that the robot (and operator) can balance on one foot while performing tasks with their hands. This is made possible by not exactly replicating foot position,

[4]A psychological experiment where the subject is tricked into believing a rubber hand to be their own [Kam+09]

but by intentionally deviating from the tracked position to keep the robot in balance. Their approach solves the problem that the robot and human can have vastly different weight characteristics. The tracking setup is similar to the one used in this thesis.

There is also research on the psychological aspects behind Telexistence. Caspar et al. investigates the sense of agency of an operator in a teleoperation scenario [CCH15]. They discover that accurate replication of the operator's motion by the robot comes with a very high sense of agency. Impressively, this sense of agency with a sufficiently good motion replication was indistinguishable from the sense of agency without any form of teleoperation. Even when motion replication was lacking, sense of agency was not completely lost. Almeida et al. discovered that sufficiently high immersion can enable human autonomic responses even through Telexistence [Alm+14]. Such a response is, for example, collision avoidance, even when the robot does not transmit pain back to the operator. They also discover that high immersion considerably decreases mental workload and thus leads to increased performance and user satisfaction.

Nour et al. proposed the use of fuzzy logic control as opposed to PID control for operation of a robot [NOC07]. This type of control also does not require a process model, meaning it can operate as a closed-loop system. They find that fuzzy logic control can consistently outperform PID controllers for certain workloads. They also prove that fuzzy logic controllers are superior with regard to parameter changes, like changes in the mass distribution of the robot. PID controllers require re-tuning in such a situation.

2 Fundamentals

2.1 System Architecture

This thesis builds on top of a *remote operation framework*, which was developed previously at TUM. The framework makes it possible to transmit the pose of a human operator to a robot within a physically correct simulation in real-time. It is also responsible for providing the operator with visual feedback about the robot's virtual environment and pose-tracking performance. The setup consists of distributed software solutions and hardware for tracking and visual feedback. Figure 2.1 provides a rough overview of the data flow within the framework. To avoid clutter, the graphic only includes tracking of the left arm, but the actual framework supports full-body tracking.

The framework follows the common server/client architecture, as seen in Figure 2.2. The client application is mainly responsible for visuals and motion tracking, and is described in further detail in section 2.2. The server application is based on the Neurorobotics Platform [FVW17]. Its main responsibilities are to simulate a humanoid robot using a realistic physics engine, and to respond to motion commands coming from the client application. A more detailed description can be found in section 2.3. Client and server communicate over a network connection using a publisher/subscriber system, which is explained in section 2.4.

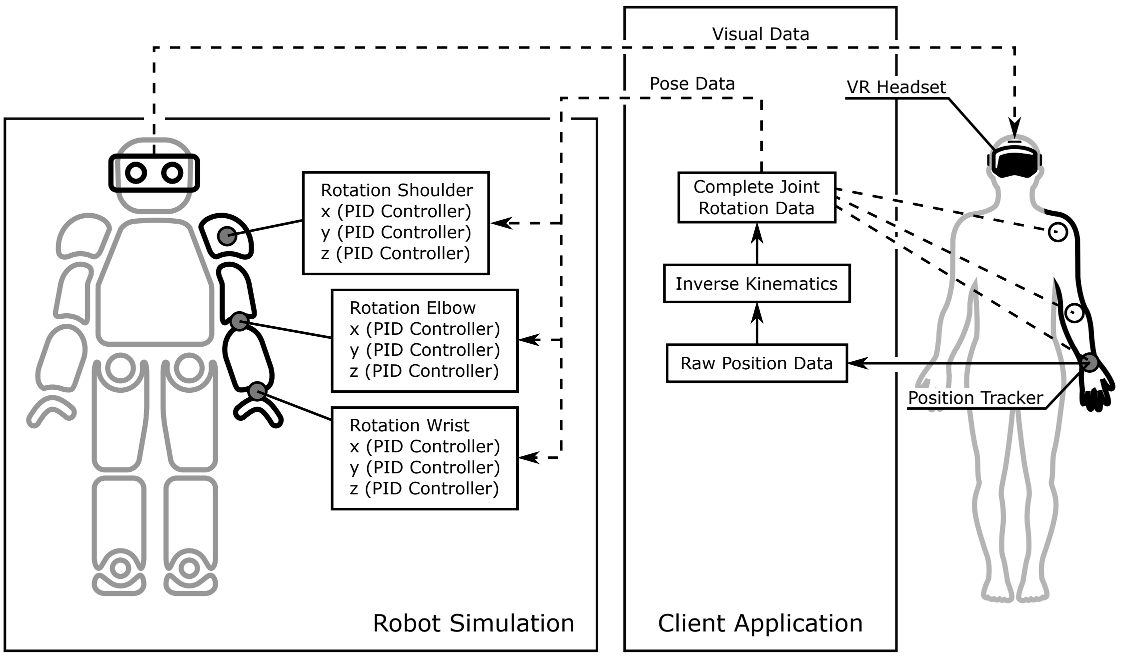


Figure 2.1: Conceptual overview of the data pipeline within both server and client

2.2 Client Application

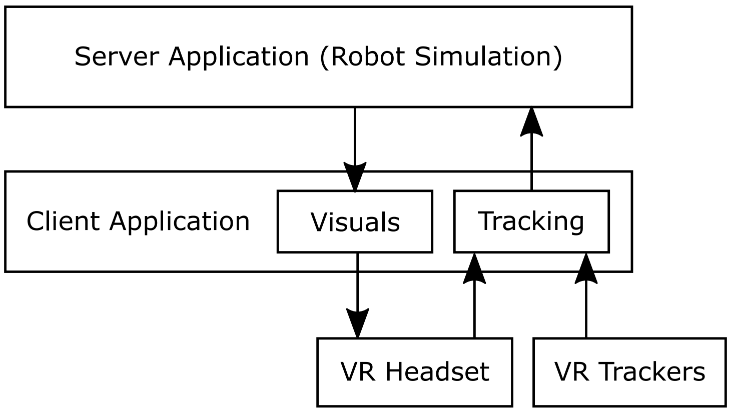


Figure 2.2: Modules within the client application and their communication with external components

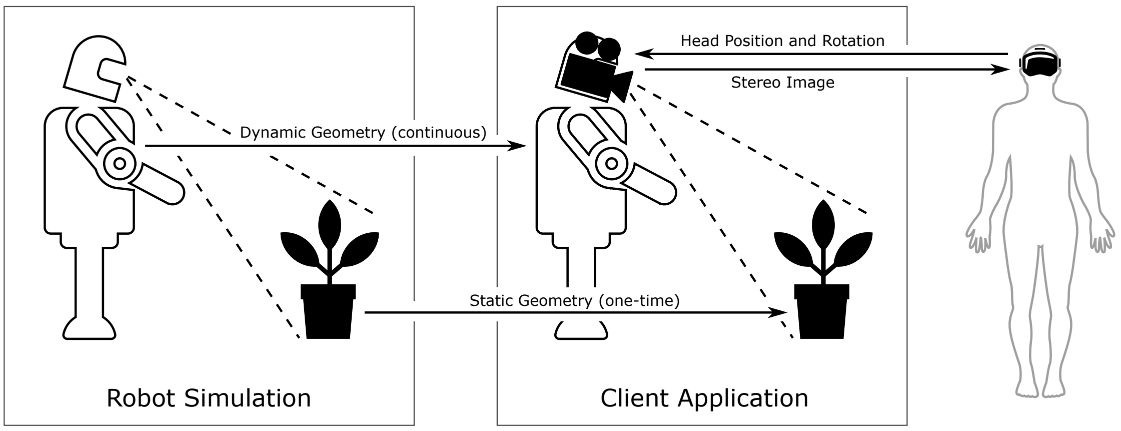

Head Position and Rotation
Stereo Image
Dynamic Geometry (continuous)
Static Geometry (one-time)
Robot Simulation
Client Application

Figure 2.3: The visual transmission subsystem

The client application has multiple responsibilities, like pose tracking, visual feedback, and inverse kinematics. It also needs network capabilities to communicate with the robot simulation server. Since none of these tasks is trivial, the Unity game engine is used as a basis for the client [Uni]. It provides built-in facilities of rendering and pose tracking, as well as inverse kinematics. Furthermore, it allows development on top the the Mono platform. Mono provides the massive .NET standard library, which is used extensively in the client application. Unity also comes with a powerful editor, which makes it possible to quickly iterate over builds of the remote operation framework.

The client contains a collection of C# classes and Unity objects to generate the visuals of the simulation. The goal is to display the point-of-view (POV) of the simulated robot to the operator in real-time. This could be called "seeing the world through the eyes of the robot", and is illustrated in Figure 2.3. All geometry that is present in the robot simulation needs an accurate visual representation in the client application. It is important to make the distinction between a physical representation and a purely visual one: The client application must no simulate the physics of the robot and its environment, since this could lead to a disagreement between client and server; Instead, it just displays the current state of the simulation to the user.

To provide a realistic and immersive sensation, it is beneficial to display a stereo image to the operator. Stereo vision greatly improves depth perception, which is important for the safe operation of a remote robot. This technology is available for consumers in the form of VR headsets. In our case, the HTC Vive VR headset is used to display the robot's environment for the user, since it has a very high refresh rate and resolution.

The image from the robot's 'eyes' is not directly transmitted over the network, since this would cause issues with image quality and latency; Instead, the entire static scene geometry in the robot simulation is transmitted at the beginning of the remote operation and then replicated in the client application, without any collision information. The client can then use local rendering facilities to replicate what the robot sees. Since this method can only be used for static geometry (meaning geometry that will not move during the simulation), dynamic objects, like the robot itself, need special handling. Such objects are updated continuously to account for their movement in the simulated scene. The result is a low-latency, high-quality stereo image that the user sees on their VR headset.

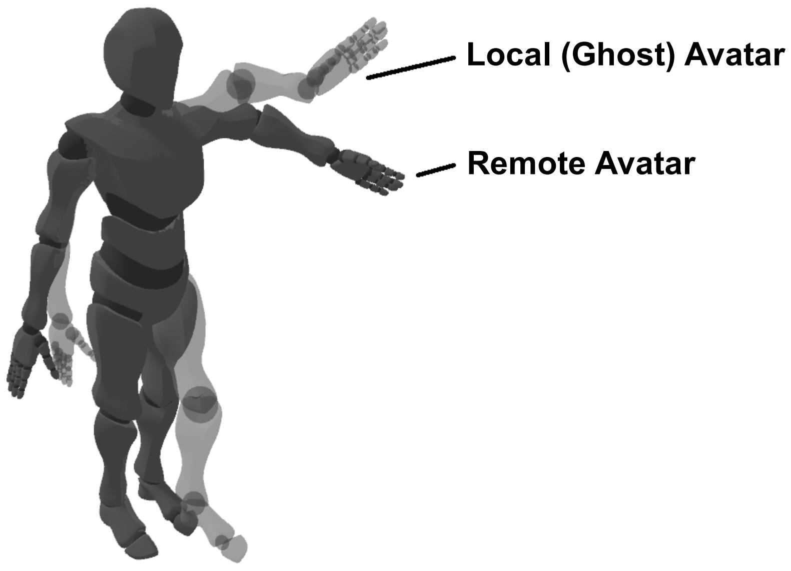


Figure 2.4: Local and remote avatar in the client application

Similarly to a real-time visual representation of the simulation, low latency feedback of the tracking performance of the remote robot is advantageous for the operator. Physical movement should entail immediate visual feedback. This is achieved by displaying a semi-transparent, ghost-like robot in the client application. This robot is called *local avatar* (Figure 2.4), since it directly represents the operator in their current pose. The operator sees this ghost-like avatar in place of their own body through their VR headset. It follows the pose of the operator immediately, unaffected by any network or simulation delays. The local avatar not only provides immediate feedback of the tracking, but also represents the target pose for the simulated robot. Since the robot will necessarily have some delayed response to motion input, the avatar serves as a visual representation of where the robot will eventually move towards.

Although the aforementioned ghost-like local avatar improves user experience, the operator ultimately only cares about the robot's performance. While the robot will always try to approximate the operator's pose as close as possible, this is sometimes not feasible; Possible reasons are self-obstruction or environmental obstruction of the robot. To allow the operator to detect and resolve such situations quickly, the robot's pose is displayed as a fully opaque 3D model in the client at all time. This model is called the *remote avatar* (Figure 2.4). It is emphasized compared to the local ghost avatar since the robot can actually interact with its environment, possibly in an unsafe manner. Any issues need to be resolved quickly in such a situation. The pose of the remote avatar will always be displayed with a certain minimum delay due to network and processing latencies.

An example of a movement sequence illustrates the function of both avatars: When the user moves their arm, they immediately see a ghost-like arm replicate their movement in VR. Then, after some delay, the opaque robot arm will overlap with the ghost arm, indicating that the remote robot has successfully replicated the pose.

Next to the visual subsystem, the tracking subsystem is the second integral part of the client. It uses tracking devices attached to the operator to create a virtual reconstruction of their pose. The HTC Vive provides two controllers, which can be used for tracking the position and orientation of the operator's hands. Additional trackers are available for reconstructing the position of both feet. All tracking occurs in near real-time with very low latency. The setup is light-weight and does not inhibit user motion significantly. The area within which the operator can be reliably tracked is several meters wide and deep, with warnings displayed in the VR headset when they get too close to the edge. Although finger tracking is possible to some degree, it is not utilized yet in the remote operation framework. Instead, the focus was put on the accurate tracking of the operator's limbs.

Since there are only a limited number of trackers (one for each hand, and optionally ankle), inverse kinematics (IK) needs to be applied to estimate position and orientation of other joints. The issue is illustrated in Figure 2.1, on the right. From the tracking data of both hands, ankles, and the current head position and orientation, the IK system can estimate the complete pose of the operator. The pose is directly applied to the local avatar, from which it can be extracted, serialized, and then sent to the robot simulation. This process is described in further detail in section 2.4.

2.3 Server: Neurorobotics Platform + Gazebo

The server application is based on the Neurorobotics Platform [FVW17], which includes the Gazebo robot simulator [KH04]. It provides vast capabilities for AI research concerning humanoid neural networks, but this thesis utilizes only the included robot simulation, called Gazebo. Gazebo is not only a physics simulation, but also contains a graphics engine and various tools and interfaces to inspect and control simulations. The entire framework is available under a open-source license.

This section will explain the most relevant parts of the Gazebo physics engine and the motion characteristics of the virtual robot model. Since Gazebo is a massive project, all parts that are not essential for this thesis were omitted. For a more detailed understanding, it can be helpful to consult the Gazebo documentation [Osra] and the specification of the Simulation Description Format (SDF) [Osrb].

2.3.1 Robot Model

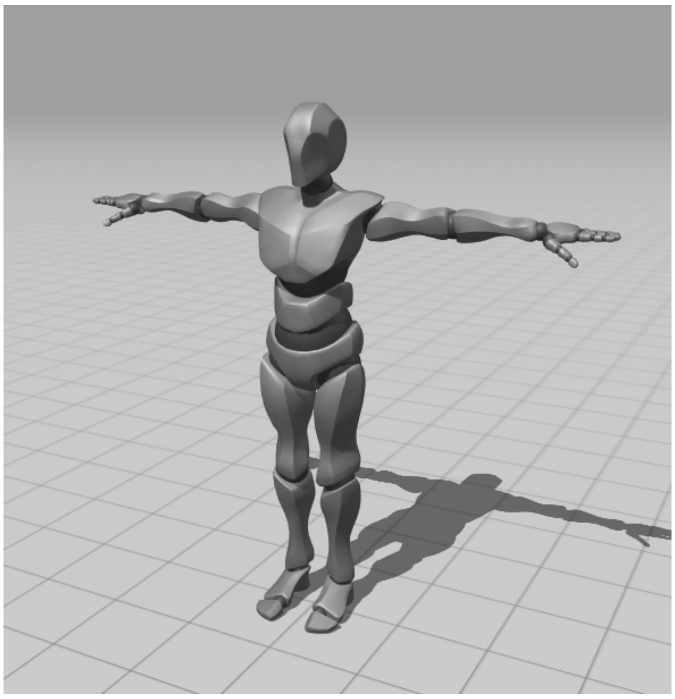


Figure 2.5: 3D model of the simulated robot

A detailed robot model is an essential part of a robot simulation. Since robots come in all shapes and sizes, robot simulators usually provide basic building blocks out of which a more complex robot can be constructed. The resulting robot model needs to describe all relevant physical properties and optionally the appearance of the robot. In addition, robot models also contain information about their movable parts and control mechanisms. In summary, a robot model should provide the simulation with every bit of information that is necessary to replicate the behaviour of a real robot, including all interaction with its environment.

Gazebo uses the SDF format to describe robot models [Osrb]. It is based on the well-known XML format and is easily extensible with custom data. The robot model used in this thesis is fairly standard and uses no special extensions of the format. SDF provides, among many other basic building blocks, links and joints. Links are use to describe rigid (non-flexing) body parts, like upper and lower arms. They also encode physical properties of the body parts, most notably their mass. A realistic mass distribution is essential for an accurate robot simulation, since real robots have mass which can not be arbitrarily distributed. Links also contain collision information and the visual representation of the body part. Since links are all rigid, joints can be added to connect two links, allowing for relative motion. There are many types of joints, like ball joints and hinges. Joints need to specify which two links they connect, how many degrees of freedom they allow, and optionally describe their limits of motion. They are used wherever rigid geometry is not enough to describe the robot, for example at the elbow.

The robot model used in this thesis can be seen in Figure 2.5. It is closely modeled after a male human physique and has enough joints to replicate most human poses, even down to the position of all finger knuckles. This high level of fidelity is not without its downsides: Every joint with at least one degree of freedom needs to be explicitly controlled, or 'driven'. If a joint receives no control input, it is subject to its inertia and to external forces, which can disturb the joint's orientation. Driving every joint, however, is not possible: For our remote control use-case, it would require tracking (or calculating) the entire pose of the human operator. Although we track or calculate some joints (e.g. shoulders, knees, elbows), there are many more which we cannot track with our simple setup. For this reason, some joint simplification is necessary.

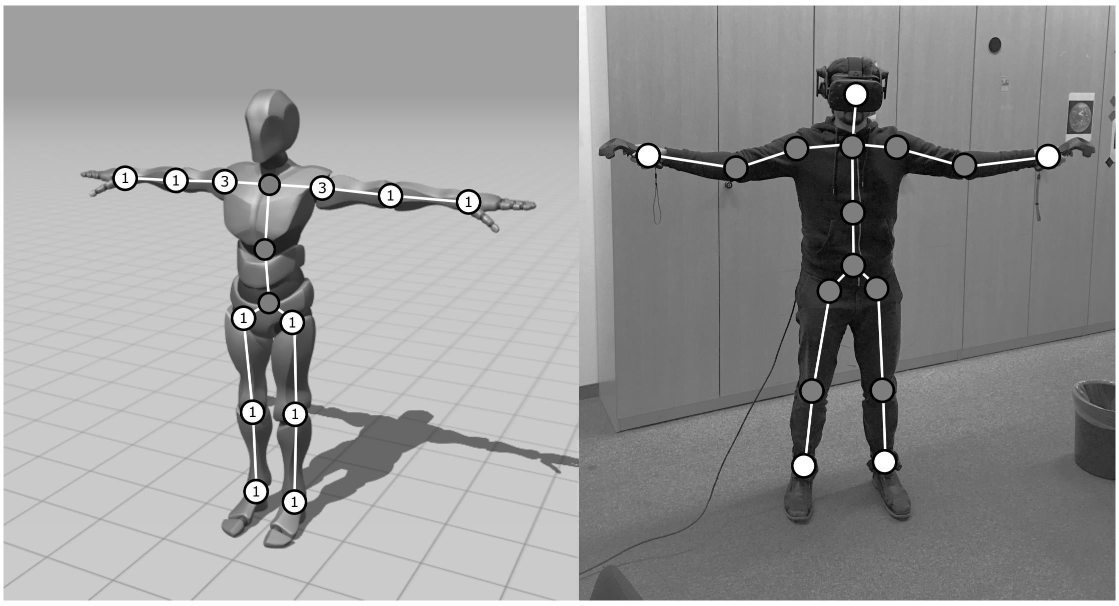


Figure 2.6: Simplified joint structure of robot (left) and operator (right); Gray joints are not actively driven (left) / not actively tracked (right)

The simplified joint structure of the robot model can be seen in Figure 2.6 (left). The number of joints was cut down to just two for every limb. This significantly simplifies the control effort at the cost of toe and finger movement. Note the DOF for each joint: Only the shoulder joints have three DOF, while all other joints have been cut down to one DOF. Ideally, we would retain more DOF, but the the setup is still sufficient to replicate most upper body poses, and basic lower body poses. Figure 2.6 (right) also shows the location of the position trackers on the operator's body. Note that none of the trackers directly measures joint orientation, which is why the client application has to apply inverse kinematics to estimate those values.

The three-DOF shoulder joints can be subject to gimbal lock. Gimbal lock is a phenomenon that occurs for certain angle configurations of the joint. It effectively removes one DOF from the joint, meaning that some rotation commands cannot be successfully executed until the gimbal lock is resolved. Instead of detecting and resolving gimbal lock when it occurs, measures can be taken to avoid it. Our setup doesn't implement any gimbal lock protection as of yet, but it might become necessary at some point. Gimbal lock has caused issues in humanoid robots before and required special handling of the effected joints [Pol+02].

2.3.2 PID Motion Control

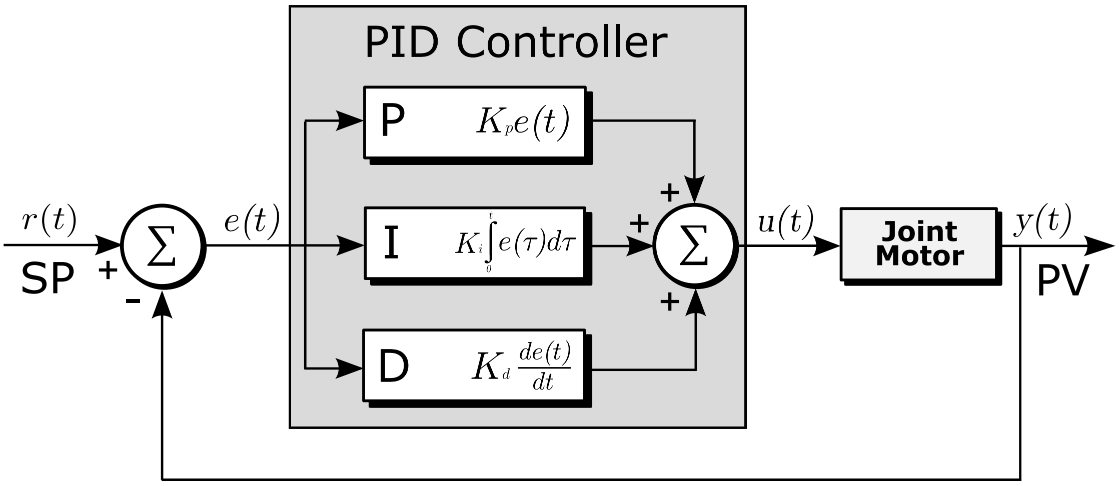


Figure 2.7: A block diagram of a PID controller.

The previous section explained the notion behind the choice of joint positions, but did not answer the question as to how to *control* them. Control is the process of determining which commands to give to the motor inside a joint so that it reaches the desired target orientation in a quick and accurate manner. All joints within the remotely controlled robot are driven by PID controllers, a variant of industrial closed-loop controllers. This section will explain the basics of closed-loop control systems, and in particular systems with PID controllers.

Controlling joints is not a straight-forwards task, since the motion response to control commands depends on external factors. An obvious example is the influence of gravity: Moving a joint downwards, in the direction of the gravitational force, requires less severe controller action than moving the joint upwards. While gravity is a rather predictable force and could potentially be accounted for, there are other disturbances that are more or less unpredictable, like winds, ocean currents, momentum, and obstacles in the path of the joint. This means that the controller output cannot be pre-calculated, and as a result cannot be controlled by an open-loop controller. Instead, we have to resort to a closed-loop controller, which constantly measures the progress of the control action and makes adjustments when necessary.

A closed-loop controller for a joint tries to move the joint towards a target angle when its current angle deviates from that target angle. There is some terminology that is commonly used to describe closed-loop controllers: The *current joint angle* is called process value (PV), while the *target joint angle* is called set-point (SP). In Figure 2.7,

$r(t)$ corresponds to SP, and $y(t)$ corresponds to PV. We can see that the controller does not get SP as its input, but rather the error E ($e(t) = r(t) - y(t)$) between SP and PV. The controller output $u(t)$ serves as input for the joint motors, which rotate the joint until it has settled on $y(t) = r(t)$, meaning that PV has reached SP and is stable. The relationship between controller output and current joint error is proportional: If the joint is far out of position, the controller response will be large.

PID controllers are by far the most widely used closed-loop controller. It is estimated that more than 95% of all industrial controllers are PID controllers [ODw06]. Since they have been used for more than 70 years, a significant amount of research is available on the topic. PID controllers are also widely used in robotics, and are available as default joint controllers in Gazebo. They have a reputation for being among the simplest of the closed-loop controllers. PID controllers have three configurable parameters (K_p, K_i, and K_d), which all have an intuitive effect on the control response. Despite their apparent simplicity, finding a good combination of parameters (called PID configuration, or PID tuning) is a challenging task. PID controllers often have to be configured carefully to meet the requirements of their use case. Examples of such requirements are fast response times, quickly reaching the set-point, and having as little overshoot as possible. One of the most important requirements is stability: Over a sufficiently long period of time, the error E should always decrease, or at least stay constant. If this condition is violated, the error will grow out of control and make effective control impossible.

The architecture of a typical PID controller can be seen in Figure 2.7. The controller takes the current error E as its input and outputs a control value to the joint motor. The error is constantly updated using SP and PV, so that the control response is always appropriate for the current state. A PID controller consists of three sub-terms (P, I, and D), which are added together to calculate the final control response. Each term is scaled by a weight: K_p, K_i, and K_d respectively. These weights are the only part of the controller that can be configured; A combination of these three weights will be called *PID configuration* from now on. Each configuration needs to be carefully balanced in order for the controller to have desirable response characteristics. Since some characteristics, like a fast response time and little overshoot, are at odds with each other, a good PID configuration is somewhat subjective. The next paragraphs will explain each term inside a PID controller in detail.

The P-term scales proportionally with the error. A large and positive error will create a large and positive output. This property makes the P-term useful for bridging most of the gap between SP and PV. It is usually combined with the largest weight constant and contributes the majority of the controller output. Setting K_p too high, however, has the undesirable effect of overshoot. Using the P-term alone (with K_i and K_d at 0) is not enough to drive PV to exactly match SP, as there will often be a residual error (called *steady-state error*). The notion behind this is that the P-term always depends on the current error value; If the error is too small, the P-term cannot output an effective control response. This is the main motivation behind the I-term.

The I-term grows with the integral of the error value over time. This means that it can generate a large response even when the current error value is very small. The idea behind the I-term is that eventually, even a small error will have accumulated a large enough integral to be countered by the control response. This property ensures that a PI controller (even without the D-term) is able to completely eliminate the error — something that a P controller alone could not achieve. The I-term, however, has a downside: Since it relies on the accumulation of past error values, it can generate a control response even when SP equals PV. This will lead to unavoidable overshoot as long as the integral does not reach zero. Usually, there is some oscillation of PV around SP until the controller has settled.

The D-term was introduced to counteract overshoot and scales with the derivative of the error with regard to time. It is based on the notion that the derivative of the error indicates a trend of PV. Based on this trend, the D-term attempts to counter-act the control response of the PI-terms. The result is a control action that has less overshoot and a faster settling time, but also a slower response time. Since the D-term is based on a derivative, it can sometimes generate very large outputs, especially when noise or disturbances are present in the system. In practice, the derivative inside the D-term is often smoothed or limited before the final calculation.

2.3.3 Other Motion Controllers

Although this thesis only covers PID control of the joint angle, other control methods exist. Instead of controlling the joint angle, it is also possible to control the velocity of a joint with a PID controller. These two different types of control are called *positional* and *velocity* PID control respectively.

Velocity control might seem counter-intuitive for a remote operation use-case, since we only care about target positions (meaning joint angles), but it actually has a few advantages [Zel+15]: Velocity control provides smoother joint motion, especially at lower control command frequencies. It also reduces contact forces, making it safer to operate the robot in a cluttered environment. Low contact forces are also a requirement when the robot needs to interact with human workers at some point. Velocity control, however, also comes with a few disadvantages: The most obvious issue is that we care more about matching the target position than matching the intermediate velocity. Usually, it is fine (and even desirable) if the robot's joint moves faster than the corresponding operator's joint, if that means that it reaches the target position earlier. Another disadvantage is a safety concern over connection losses: When the connection to a robot using positional PID control is interrupted, it stops at the last requested target position. A velocity-controlled robot, however, will keep moving at the last requested velocity, which can obviously result in damage to both the robot and its environment. Both these disadvantages contributed to the decision to utilize positional PID control for this thesis. Furthermore, the increased motion smoothness of velocity PID control is much less obvious in our use case, where joints carry significant momentum, smoothing out their trajectory by themselves.

Another alternative to positional PID control is force control. Force control is not PID-based and requires different controllers. As the name suggests, these controllers attempt to influence the force of any joint movement instead of controlling joint position or velocity directly. The need for force controllers becomes obvious when we look at the I-term of a PID controller: When the joint encounters an obstacle, the I-term will keep growing until the joint uses its maximum force. This force can become so high that the robot can easily damage its environment, including human workers.

Impedance controllers are a type of force controller that can operate similar to a positional PID controller. They attempt to move the joint to a given target position, but limit the maximum force that the joint motor will apply. This means that the joint will not use its maximum force, even if that means that the target position will never be reached. A very interesting advantage of impedance controllers are that they can replicate the operators force exactly, meaning that the operator can decide the amount of force that the task requires. This allows for effective control, as demonstrated by Tadakuma et al. [Tad+05]. Unfortunately, the advantages of impedance controllers stop there. Due to their tentative nature, they typically need more time to reach their target position. They also often require force sensors to be mounted on the limbs. These disadvantages influenced the decision in favour of positional PID control for this thesis.

2.4 Network Communication

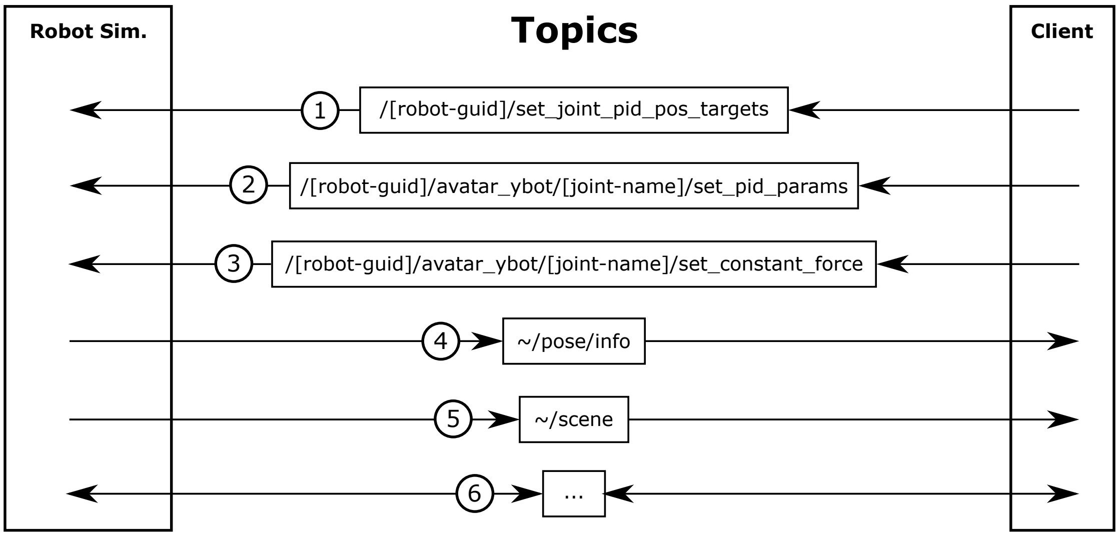

Robot Sim.
Topics
Client
1 /[robot-guid]/set_joint_pid_pos_targets
2 /[robot-guid]/avatar_ybot/[joint-name]/set_pid_params
3 /[robot-guid]/avatar_ybot/[joint-name]/set_constant_force
4 ~/pose/info
5 ~/scene
6 ...

Figure 2.8: Message exchange topics between client and server

All network communication between client and server is based on a publisher/subscriber model, which is integrated into Gazebo. This model allows programmers to create communication channels, which are called *topics*. Topics are identified by a unique string, similar to the concept of a URI. When the need for information exchange arises, a so-called *publisher* can send a message to a suitable topic. All *subscribers* will then receive the message as soon as possible.

The main motivation behind the publisher/subscriber model is that only topics which have at least one subscriber incur network traffic. Similarly, subscribers will typically only subscribe to a small subset of all the topics that the publisher provides. In this manner, the system avoids large amounts of network traffic. This impact is particularly noticeable in the remote operation use-case: The client application cares only about a very small subset of all the information that the robot simulation provides; In particular, only scene geometry and robot pose are transmitted to the client. The simulation, in turn, also subscribes to only a handful of topics that are published by the client: Besides some boilerplate communication, the client mainly publishes the currently desired target pose for the robot, and the PID configuration for all joints. All topics that are relevant to the understanding of this thesis can be seen in Figure 2.8. They will be explained in further detail below.

(1) is the topic responsible for transmitting the target pose from the client to the server. The topic name includes a unique robot ID that is negotiated when the robot in created

within the simulation. This makes sure that the system behaves as expected even in the presence of multiple robots. The content of this message is a mapping of joint names to joint angles. The robot simulation internally calculates the discrepancy between this target pose and the current pose of the robot and feeds it to all PID controllers in the robot's joints. The pose that is transmitted by via this topic is equivalent the the pose of the local avatar, as seen in Figure 2.4.

(2) is a collection of topics that is used to transmit a new PID tuning to replace the existing PID tuning of a joint. Each joint has a separate topic with the unique name of the joint in the topic identifier. The topic accepts messages containing a triplet of values, which directly replace the current values of K_p, K_i, and K_d for the respective joint.

(3) is a collection of topics that allows direct control of the forces affecting each joint. It is special insofar that it was not part of the existing remote operation framework, but was added in the course of this thesis. Again, each joint has an unique topic identifier associated with it. This topic accepts a single value denoting the constant force that should be applied to a joint during each time-step of the physics simulation. This capability is necessary to implement robust automatic PID tuning, which is discussed in subsection 3.5.2.

(4) is used to transmit the robot's current pose to the client so that it can be displayed there. This pose is directly applied to the remote avatar, which can be seen in Figure 2.4. Since a unique object identifier is contained within any message sent via this topic, the topic identifier does not have to include the joint name. This topic is more general than the other topics, since it can refer to simulation objects other than the robot.

(5) transmits information about the simulated scene to the client. This includes information about all static geometry, lighting data, etc. As seen in the illustration for the visual system (Figure 2.3), this information is typically long-lived an does not need frequent updates, unlike the other topics.

(6) includes all topics that are used by the remote operation framework, but only for secondary tasks. Such tasks include actually spawning the robot within the simulation and cleaning up the simulation after the client shuts down. Topics also exist for other housekeeping tasks such as pausing the simulation.

3 Implementation of the PID Tuning Framework

This chapter covers the contribution from this thesis to the remote operation framework. It describes the design, motivation and implementation of a *PID tuning framework*, which is entirely contained within the remote operation framework. Since the PID tuning framework is not necessarily operated by the same person that is controlling the robot, these two roles are distinguished in the text below: The person that is controlling the remote robot with their pose is called the *operator*, while the person that is tuning the PID controllers within the robot is called the *user*. This distinction is especially important since the tuning framework introduces the capability to simulate an operator without having an actual person present. More on this topic can be found in subsection 3.2.2.

3.1 Agenda

The key idea behind the implementation is to facilitate both manual and automatic PID tuning. The framework provides means for both offline and online tuning methods. It is also robust with regard to different humanoid robots, meaning that it can be re-used without too much effort while still providing a consistent workflow. Another key goal is the support of future research: The tuning framework is based on a modular design, where individual modules can be used independently of each other. It also provides facilities to record test data, which can be exported and used by other applications. To achieve all these tasks, the framework needs certain core capabilities, which are explained in this section.

A core concept of the tuning framework was immediate visual feedback of PID performance. Visual feedback is arguably one of the most important guidelines for manual tuning. The user interface, detailed in section 3.4, was built with this notion in mind. The performance characteristics of a PID controller can be quickly determined when looking at the plot of the process variable to set-point error function. For this reason, the real-time error plot has a central position within the UI (Figure 3.11). Changes to the PID tuning of a joint can be quickly evaluated by looking at the changes in the plot. It is invaluable for a rough initial tuning to observe what effect K_p, K_i, and K_d have on the response curve. Certain characteristics of a PID response, like 'acceptable overshoot' and 'good responsiveness', are subjective concepts, so a consistent visual representation of those concepts is often more useful than a misguided objective analysis by code. In summary, the UI is mostly used to guide the user towards a better PID tuning by showing them essential data in a intuitive manner.

A second fundamental concept is data collection: The framework allows the recording of test data at any time, and for any duration. The full extent of all the collected data is described in subsection 3.2.3. It consists principally out of high-frequency measurements (or: *samples*) of the currently desired pose, as well as the actual pose of the simulated robot. All samples are paired with high-precision timestamps to ensure that they paint an accurate picture of the performance test. The test data collection modules were designed with extensibility in mind: Every sample can be augmented by arbitrary key-value pairs, which can be used to carry additional information that is important for further research. Test data can also be exported in the JSON format, providing excellent compatibility with any third party tool, while still remaining readable for a human. The export process additionally saves the PID configuration of all joints, so that a correlation of test data and PID configuration is always readily available. This also makes it possible to share test data between different machines, and hopefully facilitates the creation of external tools for data analysis and visualization.

Another core concept is reproducibility. Since the primary use-case of the tuning framework is the creation of a PID configuration that most accurately tracks a human operator, it is sometimes hard to compare configurations between test runs. Human operators cannot consistently replicate their exact motion between multiple test runs, which could lead to false conclusions about test results. To address this issue, the tuning framework includes an animation playback system (subsection 3.2.2). Full-body motion playback allows users to compare the PID response characteristics of different configurations based on the exact same motion input. Any humanoid animation can be used for a test, and a handful of animations were pre-selected to cover a wide range of motion. They were picked to test slow, fast, accurate and rough motion in all possible combinations. The test data resulting from such a motion playback can be exported as usual.

An essential part of PID tuning are objective performance evaluations. Although the aforementioned visual analysis of a real-time error plot can help a great deal, it lacks the precision of quantifiable performance measurements. Visual analysis of response curves is also not an efficient option in the case of automatic PID tuning. The tuning framework addresses this by calculating various performance metrics after each test run. The evaluation module is explained in further detail in section 3.3. One of the most important metrics is the average absolute error, which is directly related to the PID's tracking performance. Other metrics, like response time, are more situational, and the user has to decide on their importance.

The final core functionality of the framework is the auto-tuning system, explained in subsection 3.5.2. Starting from some initial (reasonable) tuning provided by the user, it uses the relay tuning method [Ber17] to identify the response characteristics of a joint. It then uses those characteristics to calculate new values for K_p, K_i, and K_d, based on some heuristic. Since tuning is, as was mentioned many times, a subjective process, the auto-tuning module allows the user to chose between multiple heuristics. The auto-tuning algorithm was designed to be reliable, even in the case of unusual limb weight characteristics, force limits, or other physical limitations of the joint.

3.2 Architecture

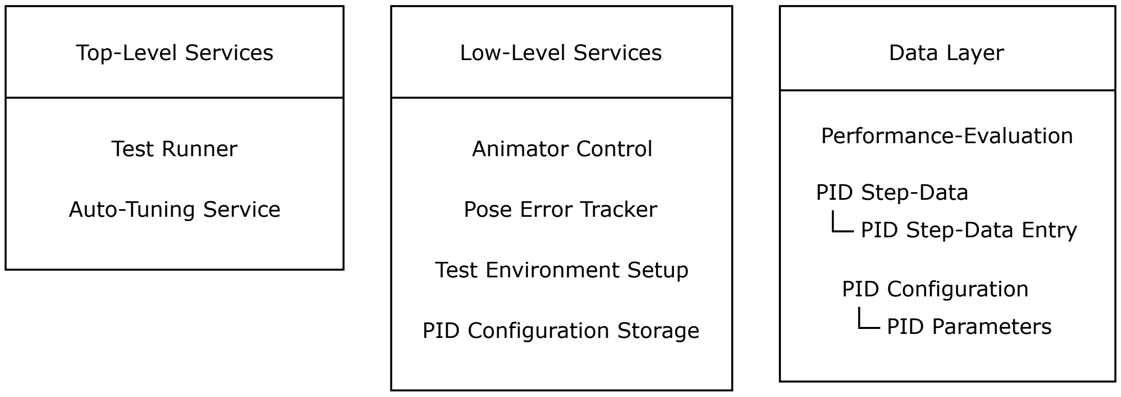


Figure 3.1: Overview of the service structure within the Unity client

The tuning framework is almost entirely contained within the Unity Client. This was a conscious decision, as it uncouples the framework from the robot simulation. The framework should still be useful even if the simulation is replaced by another one, or even a real robot.

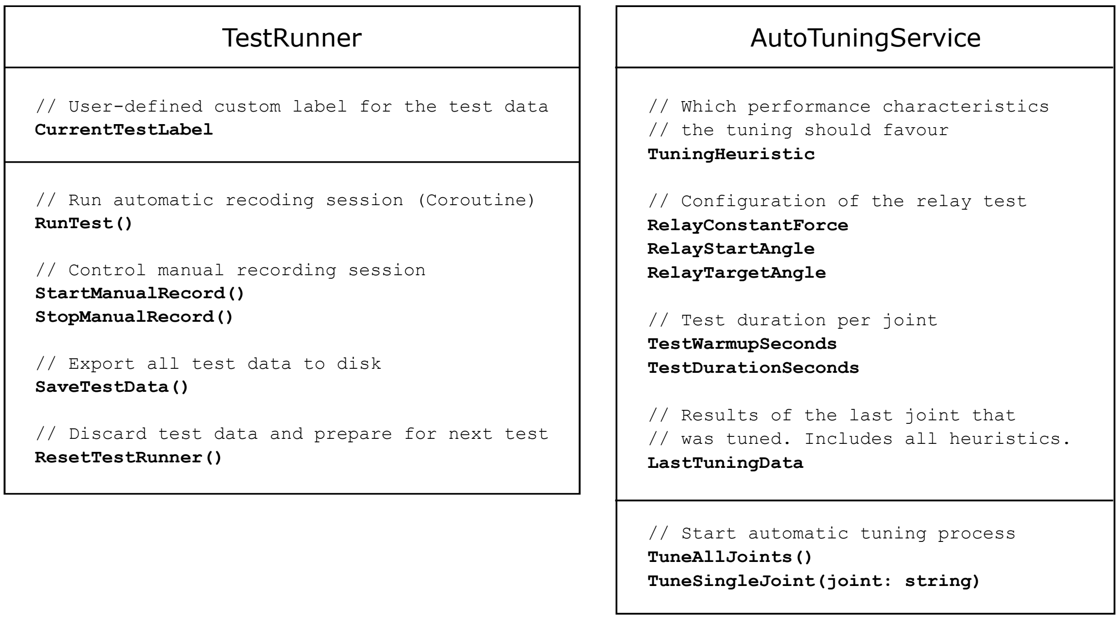
To facilitate code reuse, the framework was organized as a collection of services. Each service encapsulates one or more low-level tasks behind a simple abstraction. The idea behind this design is to allow other code to perform complex testing procedures without having to care for low-level details, like the Unity animation system. This section explains the high-level architecture of the service structure, and the interaction between the services.

The service structure can be seen in Figure 3.1. Services are categorized as either top-level or low-level, depending on the complexity of the task they are performing. This distinction was made to provide a clear entry point to the tuning framework through the two top-level services. These services encapsulate complex and long-running tasks that would be hard to perform using just the low-level services. Most users will only ever interact with the top-level services, and can ignore lower-level services entirely. Top-level services also depend on one or more lower-level services to perform their tasks, while lower-level services were designed not to depend on each other. This allows code to use only certain lower-level services without having to worry about any unintended interactions.

Both service layers make heavy use of the data layer. This layer contains little logic and is intended as a collection of data structures that are specialized for PID tuning. The data layer provides functionality for export and import, and is completely independent from the Unity game engine. This allows all data to be analyzed and visualized by third-party applications.

All services are implemented as `MonoBehaviour` sub-classes, known as *Components* in Unity. This makes it easy to integrate them into existing workflows. The services all reside on a single game object in the scene hierarchy. They can be enabled or disabled individually using the Unity inspector. All services were designed to run with zero overhead as long as they are not used, meaning that the remote operation functionality is not influenced at all by the presence of the tuning framework. Services are generally designed to revert the simulation to its initial state after performing their task, as long as this is reasonably possible.

3.2.1 Top-Level Services



TestRunner

```
// User-defined custom label for the test data
CurrentTestLabel

// Run automatic recoding session (Coroutine)
RunTest()

// Control manual recording session
StartManualRecord()
StopManualRecord()

// Export all test data to disk
SaveTestData()

// Discard test data and prepare for next test
ResetTestRunner()
```

AutoTuningService

```
// Which performance characteristics
// the tuning should favour
TuningHeuristic

// Configuration of the relay test
RelayConstantForce
RelayStartAngle
RelayTargetAngle

// Test duration per joint
TestWarmupSeconds
TestDurationSeconds

// Results of the last joint that
// was tuned. Includes all heuristics.
LastTuningData

// Start automatic tuning process
TuneAllJoints()
TuneSingleJoint(joint: string)
```

Figure 3.2: Public API of `TestRunner` and `AutoTuningService`

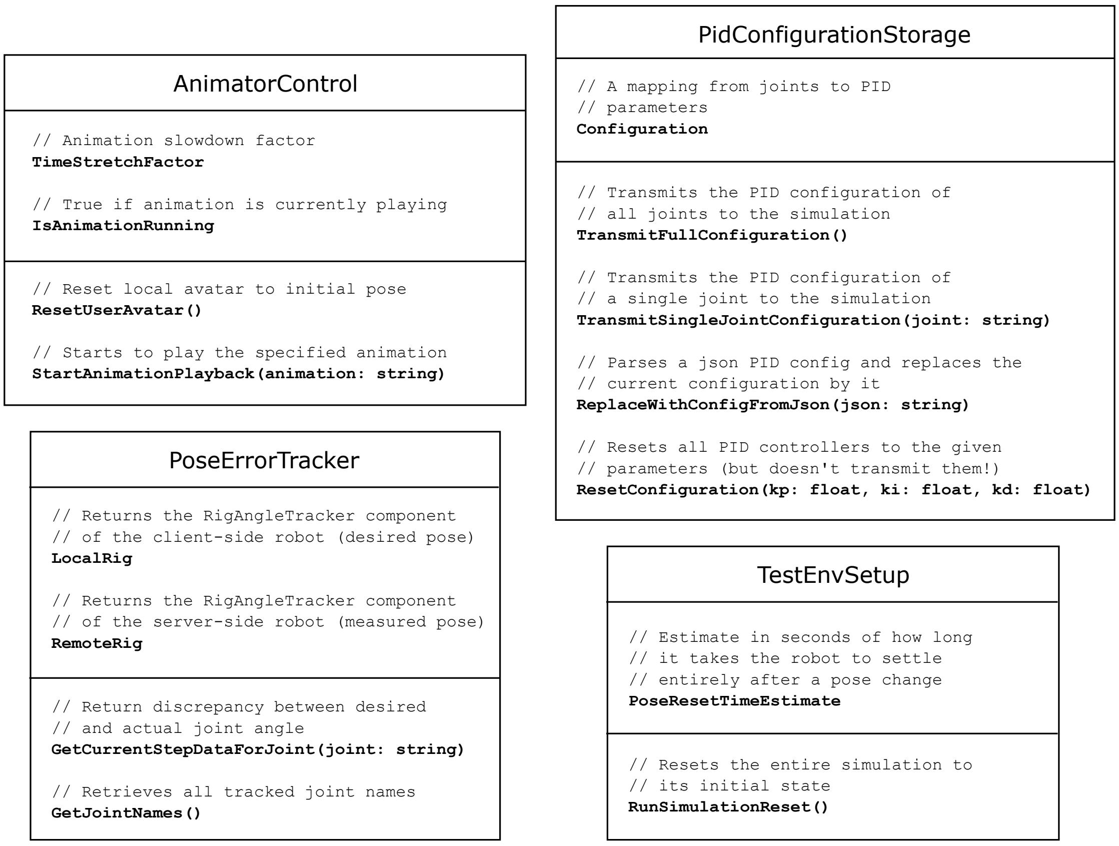
Top-level services are the main entry point for users of the tuning framework. They provide useful functionality for both manual and automatic tuning. Both the top-level services come with their own purpose-built UI. All functionality that is accessible through the UI is also accessible through public APIs. The purpose of the UI is to facilitate quick, iterative PID tuning sessions without having to touch any code.

The first top-level service is `TestRunner`. As the name suggests, it responsible for the entire procedure of testing and recording the response of all PID controllers to input changes. Data collection is the primary purpose of this service. There are two primary modes of operation: Automated and manual data collection. Starting an automated sessions assumes control of the robot, so it no longer replicates the pose of the operator. It resets the robot simulation and plays back a series of user-defined animations. While the robot performs these animations, data about the performance of each PID controller is collected in real-time. At the end of the session, the user has the option to export the recorded data to their hard drive. It includes the PID configuration with which it was recorded, and a performance evaluation for each joint. More information is available in subsection 3.2.3.

Manual recording sessions do not assume control of the robot, although they still allow the user to do that, if they wish. This means they can also be used to record normal sessions with a human operator. Like automatic sessions, they record samples of PID performance for the duration of the recording session. Manual sessions have to be stopped by the user. There is no limit to the length of a recording session except the physical memory of the machine. Sessions can be started through both the UI or through code. Figure 3.2 (left) shows all relevant public properties and methods of the `TestRunner` service.

`AutoTuningService` is the second top-level service. It is responsible for finding reasonable tunings for each joint based on user preference. The process is described in detail in subsection 3.5.2. `AutoTuningService` utilizes `TestRunner` to perform measurements of joint movement limits. It then utilizes these measurements to generate a PID tuning. The user can influence both the automatic test procedure and the resulting PID parameters by modifying public settings of `AutoTuningService`. This service is the only one that is not completely implemented within the Unity client; Instead, some of its code is embedded within a server-side Gazebo plugin. A simple UI is available for this service, and is described in further detail in subsection 3.4.2.

3.2.2 Lower-level Services



```
AnimatorControl

// Animation slowdown factor
TimeStretchFactor

// True if animation is currently playing
IsAnimationRunning

// Reset local avatar to initial pose
ResetUserAvatar()

// Starts to play the specified animation
StartAnimationPlayback(animation: string)
```

```
PidConfigurationStorage

// A mapping from joints to PID
// parameters
Configuration

// Transmits the PID configuration of
// all joints to the simulation
TransmitFullConfiguration()

// Transmits the PID configuration of
// a single joint to the simulation
TransmitSingleJointConfiguration(joint: string)

// Parses a json PID config and replaces the
// current configuration by it
ReplaceWithConfigFromJson(json: string)

// Resets all PID controllers to the given
// parameters (but doesn't transmit them!)
ResetConfiguration(kp: float, ki: float, kd: float)
```

```
PoseErrorTracker

// Returns the RigAngleTracker component
// of the client-side robot (desired pose)
LocalRig

// Returns the RigAngleTracker component
// of the server-side robot (measured pose)
RemoteRig

// Return discrepancy between desired
// and actual joint angle
GetCurrentStepDataForJoint(joint: string)

// Retrieves all tracked joint names
GetJointNames()
```

```
TestEnvSetup

// Estimate in seconds of how long
// it takes the robot to settle
// entirely after a pose change
PoseResetTimeEstimate

// Resets the entire simulation to
// its initial state
RunSimulationReset()
```

Figure 3.3: Public API of all low-level services

Low-level services are usually not be accessed directly by the user, but through one of the top-level services. Still, it is an intentional design decision to expose the functionality of these services through a public API, since this allows future researches to build up their own test frameworks without possessing in-depth knowledge of low-levels details of the game engine or network protocols. All low-level services are completely independent and may even be disabled individually without affecting other services.

`AnimatorControl` is responsible for controlling the movement of the *user-side/local avatar*. Usually, this avatar is directly controlled by the operator and replicates their pose. This pose is then transmitted to the robot simulation, and is in turn replicated by the *remote avatar*. `AnimatorControl` makes it possible to replace real-time motion input from a human operator with pre-recorded animation playback. The human operator becomes entirely unnecessary, which enables the user to perform PID tuning alone, or at least without having to perform manual motion input. `AnimatorControl` also provides the option to slow down (or speed up) animation playback. This is especially useful if the robot simulation cannot be run in real-time due to resource constraints. By adjusting the playback speed to approximately the speed of the simulation, the recorded test data becomes much more meaningful.

`PoseErrorTracker` is responsible for reporting the current discrepancy between the desired pose (set-point, local avatar) and measured pose (process variable, remote avatar). It supplies part of the data that is recorded by `TestRunner`. To measure the angle discrepancy of each joint, `PoseErrorTracker` requires a complete *joint mapping* of both the local and the remote avatar. This mapping is supplied in the form of two `RigAngleTracker` components. These components need to be manually programmed to track the current joint angle of each joint. It is essential that both the `RigAngleTracker` of the local avatar as well as the remote avatar return the same values when both are in the same pose. They should also operate in the same domain (radians, not degrees) to deliver meaningful results. Furthermore, they need to track exactly the same joints - any discrepancy here leads to errors that are hard to detect later on. The UI provides some means to verify a correct implementation of `RigAngleTracker` via the joint angle deviation plot (Figure 3.11).

`TestEnvSetup` (short for test environment setup) is used to reset the simulation to its initial state. This service has very little to do in the current framework, since the test environment is just an empty plane with a robot on it. In the future, the test environment might become more complex, and involve dynamic objects or complex task that the user is asked to perform. `TestEnvSetup` can then be extended to be a central place of control for the entire simulation. If the API is kept stable, such changes will not interfere with other services.

`PidConfigurationStorage` is the final low-level service. It is responsible for providing access to the robot's current PID configuration for all joints, as well as for transmitting that configuration to the robot simulation. The user can also chose to reset all PID controllers to the same configuration, which is useful for creating rough, initial tunings. This service is essential for both manual and automatic tuning; It allows the user to quickly iterate through tunings without having to reset the entire simulation. The service can even be used to change the entire PID configuration, depending on the task at hand. It would be possible to display a prompt for the operator to select their preferred configuration during runtime. To make manual tuning easier, this service has a custom editor UI, which is explained in detail in subsection 3.4.2.

3.2.3 Data Layer

```
{
  "createdTimestamp": "132157393707716735",
  "animation": "Open Door",
  "joint": "mixamorig_LeftArm_z",
  "simulationTimeStretchFactor": "7",
  "data": [
    {
      "timestamp": "132157393737512470",
      "entry": {
        "desired": -0.0001493394,
        "measured": 26.344574
      }
    },
    {
      "timestamp": "132157393738569660",
      "entry": {
        "desired": 4.55049,
        "measured": 34.4379578
      }
    },
    {
      "timestamp": "132157393739598476",
      "entry": {
        "desired": 9.762062,
        "measured": 34.4379578
      }
    }, ...]
}
```

```
{
  "createdTimestamp": "132157390918975488",
  "mapping": {
    "mixamorig_LeftArm_x": {
      "kp": 1000.0,
      "ki": 100.0,
      "kd": 500.0
    },
    "mixamorig_LeftArm_y": {
      "kp": 1000.0,
      "ki": 100.0,
      "kd": 500.0
    },
    "mixamorig_LeftArm_z": {
      "kp": 1000.0,
      "ki": 100.0,
      "kd": 500.0
    },
    "mixamorig_RightArm_x": {
      "kp": 1000.0,
      "ki": 100.0,
      "kd": 500.0
    },
    "mixamorig_RightArm_y": {
      "kp": 1000.0,
      "ki": 100.0,
      "kd": 500.0
    }, ...]
}
```

Figure 3.4: Abbreviated examples of a `PidStepData` (left) and `PidConfiguration` (right) JSON exports

The data layer provides a common basis of communication between services, as well as between the tuning framework and the outside world. It consists of multiple, mostly immutable C# classes. Each class in the data layer provides a `ToJson()` method, which serializes the data to the JSON format. This format is widely used and makes it possible to save data to disk, as well as to share it with other applications. Some classes also implement a `FromJson(...)` method that performs the reverse task. JSON files are also human-readable, so the data can be inspected with a simple text editor. Some classes contain methods that let the user extend the exported data with custom values. The idea behind this is to allow future researchers to extend a data collection without having to re-write large parts of the system. Custom data is also useful to embed meta-information within the data, such as latency measurements or commentary. All data collections were designed to be shared between machines, without any dependencies on the current working environment.

Some data classes contain timestamps, which all adhere to the Windows file time standard: They count the number of 100-nanosecond intervals that have elapsed since 12:00 A.M. January 1, 1601 Coordinated Universal Time (UTC) [Mcl18]. This format was chosen because it provides high precision, while occupying only a single 64-bit integer.

`PidStepData` contains the step-data of a single PID controller within a joint, as produced by the `TestRunner` service. An example of a JSON export can be seen in Figure 3.4 (left). Each sample (`PidStepDataEntry`) contains the desired joint angle (set-point) and the actual joint angle (process variable) in degrees. The range of these values is not clamped in any way, so -180 degrees and 180 degrees are treated as different rotations. Every sample comes with an absolute timestamp: If two step-data instances need to be compared, it is important to subtract the timestamp of the first measurements from all other timestamps. This way, both recordings start at 0, which is usually more convenient than using absolute timestamps.

`PidStepData` also contains some meta-data: A creation timestamp, the name of the animation that was used for the test recording (if any), the joint that the step-data belongs to, and the time stretch factor of the animation. Note that the timestamps of each sample are always collected in real-time, without any respect for time-stretching or time-compression. Nonetheless, `simulationTimeStretchFactor` should always be checked for equality when comparing two `PidStepData` instances, since a discrepancy can render any comparison meaningless.

The `PidConfiguration` class contains the mapping of joint names to PID parameters. A JSON export can be seen in Figure 3.4 (right). PID parameters are saved in *parallel form*, and not *standard form*, which is more common in the industry. The C# class `PidParameters` provides methods to convert between these two formats. `PidConfiguration` instances can also be parsed from JSON files, which enables third-party applications to create tunings for the robot. It also allows researches to send each other PID tunings without having to manually enter them into the framework.

3.3 Performance Evaluation

The `PerformanceEvaluation` class belongs into the data layer, but deserves its own section due to its complexity. This complexity stems from the creation of a `Performance Evaluation` object, not from the content. The data itself can be exported as a small JSON file, just like all other classes in the data layer. The fist subsection explains all individual metrics of `PerformanceEvaluation`, while the second one goes on to describe the process of extracting those metrics from a given collection of test data.

3.3.1 Metrics

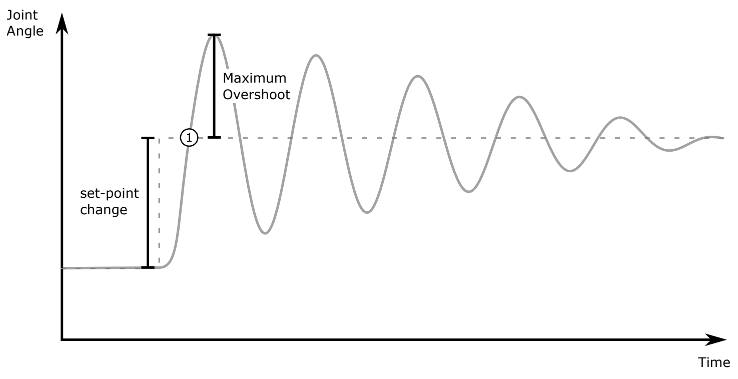


Figure 3.5: Overshoot as seen in the step-data plot of a joint

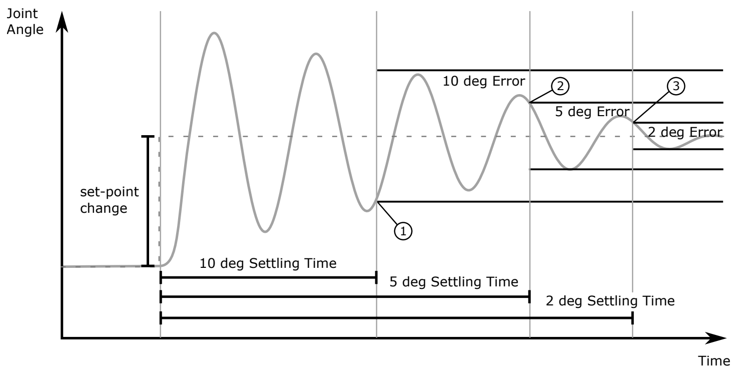


Figure 3.6: Settling time metrics as seen in the step-data plot of a joint

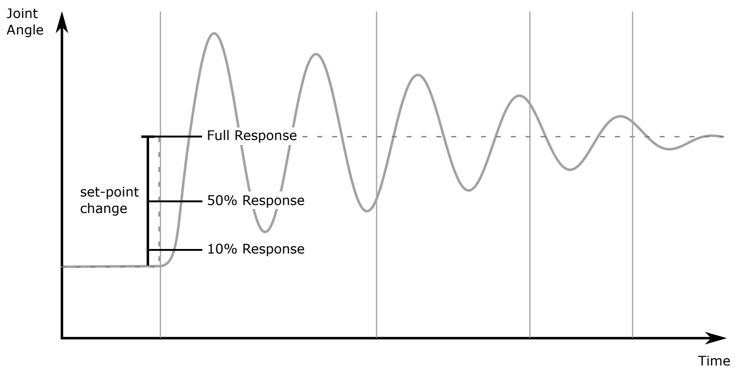


Figure 3.7: Response time metrics as seen in the step-data plot of a joint

`AvgAbsoluteError` is accumulate absolute angle difference between desired joint angle and measured joint angle in degrees, divided by the number of samples. This is arguably the most important performance metric, as it measures the tracking accuracy of the PID controller. Many desirable PID behaviours have a positive influence on this metric: Quick set-point tracking, small overshoot, and few oscillations all decrease `AvgAbsoluteError`. Nonetheless, this metric cannot be blindly trusted: It is possible to get a small absolute error even if tracking performs poorly. This can happen if the set-point coincidentally changes to match the overshooting (or overdamping) behaviour of the controller. Consider, for example, a joint with massive overshoot, where the set-point suddenly changes to the peak of the overshoot. The resulting error will be small, but tracking performance in other scenarios would be very poor. For this reason, it is important to collect many samples from different motion inputs.

`AvgSignedError` is very similar to `AvgAbsoluteError`, but does not remove the sign of the angle difference between desired and measured joint angles. This metric is useful for detecting unsymmetrical response behaviour of a joint, meaning that it is easier to turn a joint in one direction than the other. Gravity is a typical candidate for causing the kind of behaviour. If a joint is heavily skewed towards one direction, it might be necessary to utilize a separate PID controller for the other direction. When a set-point change occurs, the appropriate controller can be picked, to ensure a reasonable response in both directions. `AvgSignedError` is only useful for prolonged measurements, where many set-point changes influence the average. Consider, for example, a test run with only a single set-point change: This set-up would always result in a heavily skewed error into the direction of the change.

`MaxAbsoluteError` is exactly what is says on the lid: Instead of averaging error values, it calculates the maximum absolute discrepancy between desired and measured joint angle over the whole test data. This metric is not as useful for measuring controller performance as it might seem at first glance: For stable processes (where the error decreases over time), the maximum absolute error is usually caused by the largest set-point change. The reason for this behaviour is that a sudden set-point change immediately introduces an error, since the controller did not have any time to react. This means that both a very well tuned and a poorly tuned controller can both have the same value for `MaxAbsoluteError`. This metric is, however, not entirely useless: Instead of giving information about the controller, it gives us information about the motion input. If there are many large set-point changes, we might want to increase to sample frequency of the pose, to avoid these 'surprising' scenarios.

The overshoot of a set-point change is the absolute error of the first peak after crossing the set-point line. This concept is illustrated in Figure 3.5. `MaxOvershoot` contains the maximum overshoot that occurred during the test duration. Note that this metric is not always available: Sometimes, a system is so heavily damped that no overshoot occurs. In other situations, set-point changes occur so rapidly that overshoot cannot be reliably measured. In both the cases, `MaxOvershoot` will be set to `null`. This metric is very useful for determining how aggressive the PID controller is. Large overshoot often comes with a short response time, which might make it acceptable in some situations. For control of a robot by a human operator, overshoot should typically be as small as possible. Otherwise, the robot could damage itself or its environment.

`AvgSettlingTime10Degrees`, `AvgSettlingTime5Degrees`, and `AvgSettlingTime2Degrees` are metrics denoting the settling time (in seconds) of the process after a set-point change. A response is determined to have settled at the point where each further oscillation falls within x degrees of the new set-point. The concept is illustrated in Figure 3.6. Like `MaxOvershoot`, the settling time metrics are optional, and can only be reported when set-point changes are relatively rare within the test data. If they can be determined, settling time metrics are an exceptionally useful metric for tuning PID controllers. For some joints, a rough, fast response is more important than perfect accuracy. 10 degree settling time can be used to tune these joints to the operator's preference. Other joints, like hands or fingers, will sacrifice fast response times for higher accuracy. Here, the controller is expected to settle within 5 or 2 degrees of the new set-point relatively quickly.

`Avg10PercentResponseTime`, `Avg50PercentResponseTime`, and `AvgCompleteResponseTime` measure the time (in seconds) that it takes a controller to respond to a set-point change. A 10 percent response, for example, is the time that it takes the joint to pass 10 percent of the distance between the old set-point and new set-point. Figure 3.7 visualizes the concept. Like settling time and maximum overshoot, response time metrics are not always available. Response time metrics are a much more situational performance indicator than settling time metrics. For precise tasks, response time is usually not important, and operators can live with small delays if that avoids overshoot. On the other hand, imagine a robot that occasionally has to dodge projectiles or obstacles. For such a robot, a fast response time might be much more important than avoiding some overshoot, as a collision could seriously damage it.

After finishing a recording using the `TestRunner` service, `PerformanceEvaluation` objects are automatically created for each joint. Additionally, some cumulative `PerformanceEvaluation` objects are created: For each animation, the evaluations of all joints are combined, to give the user an idea of the overall performance of the robot. All metrics that contain averages are simply averaged over all joints. `MaxOvershoot` will contain the maximum overshoot across all joints. An additional, single `PerformanceEvaluation` is created at the end. This evaluation is an accumulate of the evaluations of every animation. If only one animation (or none) was played, this evaluation is equivalent to the evaluation of all joints combined.

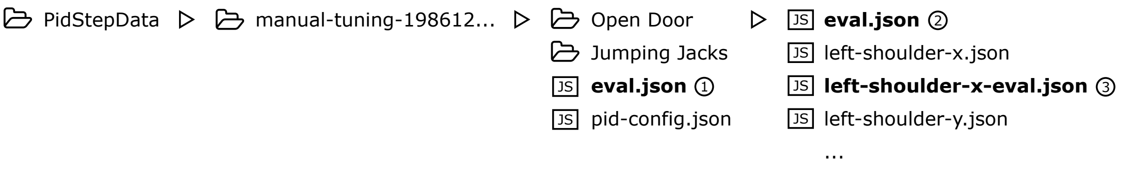


Figure 3.8: Folder structure of a test data export with performance evaluations highlighted

The folder structure is visualized in Figure 3.8. (3) is the evaluation of a single joint. (2) is the accumulate evaluation of all joints combined for the entire "Open Door" animation. (1) is the accumulate evaluation of all animations, or in this case, "Open Door" and "Jumping Jacks".

3.3.2 Metrics Calculation

Some metrics, like `AvgAbsoluteError`, are trivial to compute, but settling times, response times and maximum overshoot all require some special logic. This section outlines the algorithms that are used to calculate these *complex metrics*, as they will be called from now on. All of them are based on the notion of a *set-point change*. In the context of this thesis, I decided to define the notion of a *significant* set-point change: The requirements of a significant set-point change are that the amount of change in degrees is larger than some epsilon[1] and that the new set-point is held constant for at least 2 samples. These requirements are relatively lenient, and all metrics perform further filtering to determine if a given set-point change can be used sensibly in their calculation.

All complex metrics require the detection of significant set-point changes within the test data. Test data from animation playback often contains not even a single significant set-point change, since motion-captured animations usually update the set-point at a very high frequency. This violates the second requirement of a significant set-point change: The new set-point needs to be held constants for at least two samples. `TestRunner` allows users to increase the sample rate, but this will usually not improve the situation by much. PID controllers always need some time to 'react' to a set-point change; If the next change comes too quickly, it's not possible to extract meaningful metrics. The solution is to perform test runs where the set-point is changed at a low frequency. Such test runs can be performed by manually adjusting joint angles, or by playing back specially prepared animations. subsection 3.5.1 gives a more detailed description of the process.

[1]Provided through Unity's `Mathf.Epsilon`

`MaxOvershoot` is the simplest of the complex metrics. The algorithm is split into two phases, which are illustrated in Figure 3.5: First, we need to find the sample where the process variable crosses the set-point for the first time (point (1) in the graphic). If no sample like that exists, the controller is over-damped, and the metric cannot be computed. Otherwise, we just save the maximum absolute error that we encounter from the crossing point until the next set-point change. For a stable response, this maximum is equivalent to the maximum overshoot, as we can expect the oscillations to get smaller. This equivalency becomes problematic if the response turns out to be unstable, meaning that oscillations increase in amplitude. In that case, `MaxOvershoot` will contain the largest unstable oscillation amplitude, and not the amplitude of the first peak after the crossing point. I decided to accept this shortcoming, because an unstable response itself is a much bigger issue than a confusing value for `MaxOvershoot`. Unstable responses can be detected by examining `AvgAbsoluteError`, which will be unusually large in such a scenario. They can also be detected visually by observing that the robot is going crazy.

The settling time metrics (`AvgSettlingTime10Degrees`, etc.) are implemented in a straightforward way: From the last sample, the algorithm searches backwards for the first sample that falls outside of the 2/5/10 degree error band. The concept is illustrated in Figure 3.6. Points (1), (2), and (3) show the position of these samples for each settling time metric. The actual settling times are calculated by subtracting the timestamp of the first sample after the step point change from the timestamps of these points. The 10 degree settling time is `null` if the set-point change was less than 10 degrees, with similar logic for 5 and 2 degree settling times. The algorithm produces wrong results if the last sample it analyzes coincidentally falls into the `x` degree error band, but is actually part of a bigger oscillation. In this case, the settling time metric is available, although it should actually be `null`. An attempt was made to remedy this via peak analysis (as explained in the next section), but the method proved too unreliable.

The response time metrics (`Avg10PercentResponseTime`, etc.) are also relatively simple: As seen in Figure 3.7, we find the respective first samples where the absolute error goes below 90%/50% of the set-point change. From the timestamps of these samples, we then subtract the timestamp of the first sample after the set-point change to calculate the values of our metrics. To avoid inaccuracies from floating point operations and noise, a minimum step-change of 10 degrees is required.

A slightly different method is used to calculate the 100% response time (`AvgComplete ResponseTime`). For this metric, we look for the first sample where the sign of `SignedError` flips, as this indicates that the set-point was crossed by the process variable. The same strategy is utilized in the calculation of `MaxOvershoot`, as described above. This method is slightly susceptible to noise. If significant noise is present in the system, it might become necessary to pre-filter the samples before applying this algorithm.

3.3.3 Peak Detection

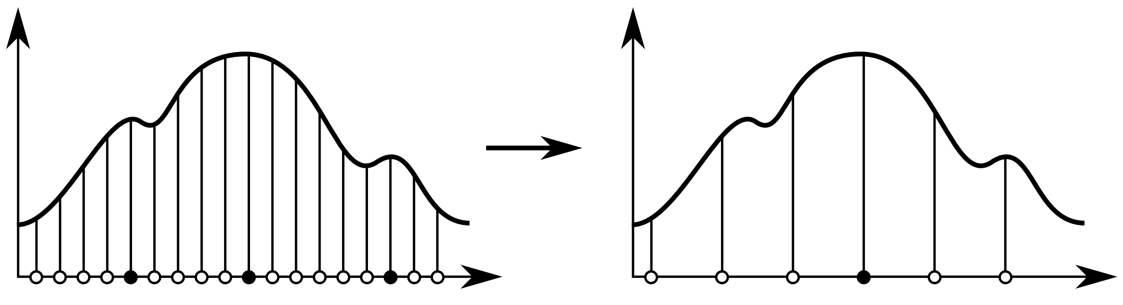


Figure 3.9: Eliminating false positive in the peak detection algorithm by lowering data resolution

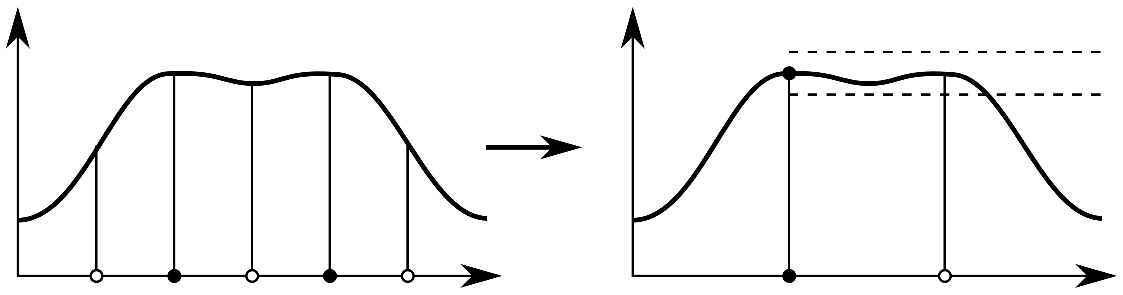


Figure 3.10: Eliminating false positive in the peak detection algorithm by ignoring similar peaks

In the context of this thesis, peak analysis describes the process of finding the time-stamps and amplitude of oscillatory peaks within the test data. It is not directly part of the `PerformanceEvaluation` class, but was included here because of the conceptual similarity. Peaks typically occur after a large set-point change, and can serve as a useful performance metric: An underdamped joint will typically produce more peaks than a well-tuned PID controller. The trend of peak amplitudes can indicate settling times in a more robust way than the actual settling time metrics.

Peak analysis requires a fairly complex algorithm due to the nature of the collected test data. Due to slight noise and floating point errors, peaks cannot be reliably detected by examining just three adjacent points: Approximating the first derivative of the signed error w.r.t. time by finite differences leads to many false positives. Even if the sampling range is expanded, false positives cannot be avoided completely. Figure 3.9 shows some of the problematic scenarios for the detection algorithm. To avoid false positives, a simple filtering algorithm is utilized: By skipping some samples, the resolution of the test data is decreased to a point where false positives are much less likely to exist. The ratio of skipped samples to retained samples can be adjusted in code to account for different sampling rates in the test data. The graphic shows how the low-resolution test data eliminates many false positives.

Although lowering the data resolution helps against some false positives, it does not get rid of another source of errors: Duplicated peaks. When the sampling rate is very high, or oscillations are relatively slow, one peak can be detected multiple times by the algorithm. The problem is illustrated in Figure 3.10. Another filtering step is applied to remove those peaks: All consecutive matches within a small threshold of the last peak are ignored. This method tends to detect the peak 'too early', but the error is usually so small that it can be safely ignored. After both filtering steps, the algorithm outputs a sequence of samples that were determined to be oscillatory peaks.

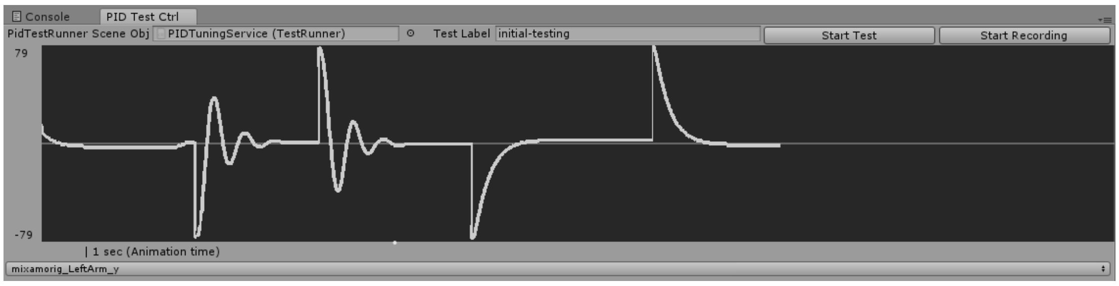
3.4 User Interface

This section explains the function of all UIs in the tuning framework. The UI allows users to access the public API of the tuning services without writing any code. Besides the visual elements, it does not provide any service that cannot also be accessed through the API. This design choice was made to facilitate automatic PID tuning without having to rely on user input. The UI provides only simplified access to the tuning services; If more fine-grained control is needed, the user needs to access the public APIs through code.

All UI elements are created using Unity's immediate mode GUI classes and methods. This allows the framework to be integrated directly within the Unity editor. Unlike external software, Unity's UI system allows access to all of the Unity engine as well as extended editor functionality. It also enables functionality which is expected from a modern UI, like copy-paste, tab navigation and drag-and-drop. Another important feature is that the scene hierarchy is completely accessible for UI elements, enabling the testing framework to display real-time information based on the current client state. One disadvantage of using Unity UI is that it is only available from within the Unity editor, and not in builds. Although using the tuning framework UI is not possible in builds, all APIs are still available and functional.

The UI was designed to be simpler than the public APIs. Error messages are usually printed to the console with a suggestion on how to fix the underlying issue. The UI does not allow the user to enter an invalid state by accident, e.g. starting a test run while another test run is still active. Due to the anything-goes design of Unity, it is not always possible to prevent the user from making mistakes. The UI and the public APIs are not meant to be used in parallel. While most tasks will tolerate user interference through the UI, it can make test results unreliable and compromise the automatic tuning process. In general, commands should be issued either entirely through the UI, or entirely through code.

3.4.1 Test Control Window



PidTestRunner Scene Obj PIDTuningService (TestRunner)
Test Label initial-testing
mixamorig_LeftArm_y

Figure 3.11: The test control window after performing some manual tuning

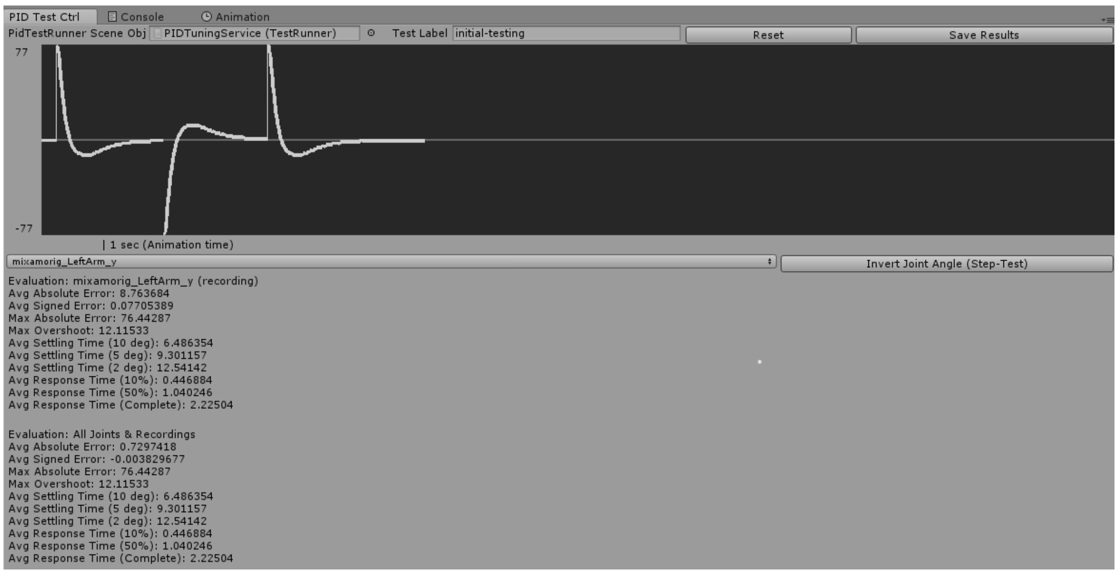
The test control window is the core of the tuning framework UI. It is used to verify correct pose tracking, start manual and automatic test runs, and to view the performance evaluations of test runs. Its most prominent visual element is the joint angle error graph, which is essential to the process of manual PID tuning. The test control window is implemented as an extension of `EditorWindow`, a Unity class that allows the creation of custom, resizable windows. A screenshot of the window can be seen in Figure 3.11. The window needs to be initialized by dragging the `PIDTuningService` game object from the hierarchy into the indicated field after entering play mode.

The joint angle error graph takes up the bulk of the test control window. Its vertical axis shows the signed angle deviation between the desired angle of a joint (set-point) to the measured angle on the remote robot (process variable) in degrees. The active joint can be selected in the drop-down below the graph. The scale of the vertical axis is adjusted automatically so that the graph never goes outside the visible area. The upper and lower limits of the axis are displayed to the left of the graph.

The horizontal axis represents time. For performance reasons, which are explained below, this axis does not have constant scaling. Instead, it compresses if the angle deviation grows too large to be displayed correctly on the graph. For this reason, the axis shows a scale indicator below the graph display. It indicates the width of one second of animation time, according to the setting of `TimeStretchFactor` in the `AnimatorControl` service. If `TimeStretchFactor` is set to one, this indicator displays the width of one second in real-time.

The graph is rendered using `PreviewRenderUtility` from the Unity editor assembly. This class provides an interface to display any Unity scene within an editor window. To render the line that represents the current joint angle deviation, a `GameObject` with a `LineRenderer` component is instantiated within an empty scene. Since the `GameObject` is a prefab, users can customize the appearance of the graph by editing the `LineRenderer` component. To provide real-time performance for the graph display without negatively influencing user experience in the rest of the editor, some optimizations are included: The point list for the `LineRenderer` is extended in-place, and not copied from some source whenever a new sample arrives. This allows Unity to avoid re-allocating and copying the point list after every sample, which is a very costly operation. A further optimization is that existent points are never modified. This allows Unity to also cache large parts of the line mesh. However, this becomes an issue if a new sample falls outside of the current vertical axis limits of the graph, since we would need to 'squish' the mesh to get it to display completely. To avoid this, the vertical FoV of the camera component in `PreviewRenderUtility` is increased to account for the new sample. While this trick avoids a performance penalty, it also introduces unwanted scaling of the horizontal axis. This trade-off is accepted because real-time performance is such an important characteristic of the graph display.

In the top right corner of the test control window, two buttons control the `TestRunner` service. "Start Test" starts an automatic test run, and "Start Recording" starts a manual recording session. The difference between both modes of operations is explained in more detail in subsection 3.2.1. Automatic test runs finish by themselves after playing through all animations, while manual test runs need to be stopped by the user. The "Stop Recording" button appears in the top right once a manual recording session is active.



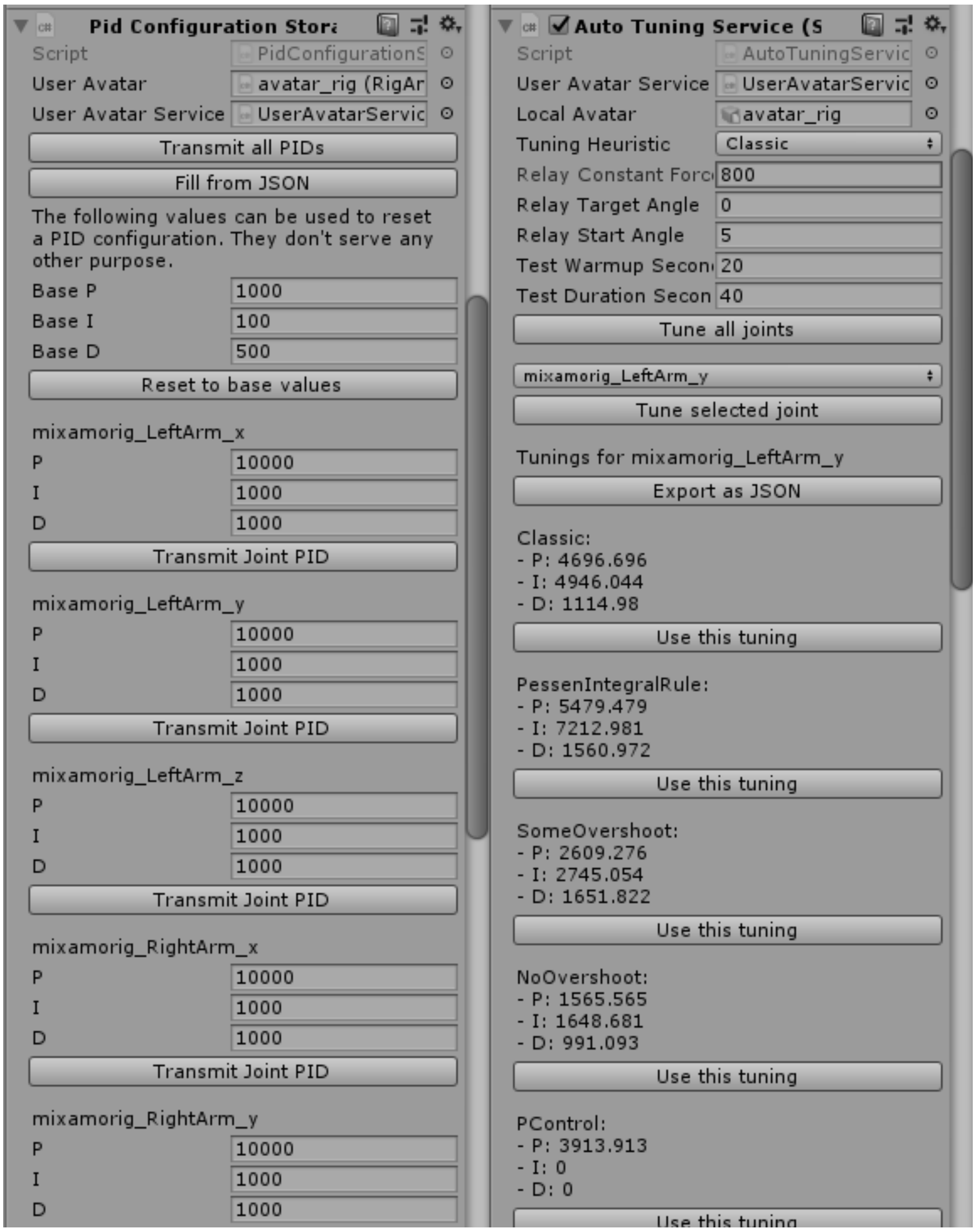
```
Evaluation: mixamorig_LeftArm_y (recording)
Avg Absolute Error: 8.763684
Avg Signed Error: 0.07705389
Max Absolute Error: 76.44287
Max Overshoot: 12.11533
Avg Settling Time (10 deg): 6.486354
Avg Settling Time (5 deg): 9.301157
Avg Settling Time (2 deg): 12.54142
Avg Response Time (10%): 0.446884
Avg Response Time (50%): 1.040246
Avg Response Time (Complete): 2.22504

Evaluation: All Joints & Recordings
Avg Absolute Error: 0.7297418
Avg Signed Error: -0.003829677
Max Absolute Error: 76.44287
Max Overshoot: 12.11533
Avg Settling Time (10 deg): 6.486354
Avg Settling Time (5 deg): 9.301157
Avg Settling Time (2 deg): 12.54142
Avg Response Time (10%): 0.446884
Avg Response Time (50%): 1.040246
Avg Response Time (Complete): 2.22504
```

Figure 3.12: The test control window after recording some step-tests

Once a recording session is over, no matter if it was automatic or manual, two new buttons appear in the top right: "Reset" and "Save Results". Figure 3.12 shows the UI in this state. A reset discards all test data that was collected in the last run. Before any new recording can be made, a reset must be performed. This is a conscious design choice to prevent users from accidentally discarding important data from a test run. The "Save Results" button exports all data to disk. The path names and folder structure is shown in Figure 3.8. If a manual recording was performed, the exported data will contain a single nested folder called "recording" instead of individual folders for each animation.

Because it would be tedious to always export data to access the performance evaluations, they are also displayed below the graph, as seen in Figure 3.12. The topmost evaluation(s) refer to the joint that is selected in the drop-down. One evaluation is displayed per animation. In case of a manual recording, just a single evaluation is displayed for the joint. Below that, the accumulate evaluation of all joints and animations is displayed. This evaluation is mostly useful for a rough, initial tuning, where all PID controllers are set to the same initial configuration. After that phase, it is usually beneficial to concentrate on one joint at a time.

3.4.2 Service UIs



Pid Configuration Stor

Script	PidConfigurationS
User Avatar	avatar_rig (RigAr
User Avatar Service	UserAvatarServic

Transmit all PIDs

Fill from JSON

The following values can be used to reset a PID configuration. They don't serve any other purpose.

Base P	1000
Base I	100
Base D	500

Reset to base values

mixamorig_LeftArm_x

P	10000
I	1000
D	1000

Transmit Joint PID

mixamorig_LeftArm_y

P	10000
I	1000
D	1000

Transmit Joint PID

mixamorig_LeftArm_z

P	10000
I	1000
D	1000

Transmit Joint PID

mixamorig_RightArm_x

P	10000
I	1000
D	1000

Transmit Joint PID

mixamorig_RightArm_y

P	10000
I	1000
D	1000

Auto Tuning Service (S

Script	AutoTuningServic
User Avatar Service	UserAvatarServic
Local Avatar	avatar_rig
Tuning Heuristic	Classic
Relay Constant Forc	800
Relay Target Angle	0
Relay Start Angle	5
Test Warmup Secon	20
Test Duration Secon	40

Tune all joints

mixamorig_LeftArm_y

Tune selected joint

Tunings for mixamorig_LeftArm_y

Export as JSON

Classic:
- P: 4696.696
- I: 4946.044
- D: 1114.98

Use this tuning

PessenIntegralRule:
- P: 5479.479
- I: 7212.981
- D: 1560.972

Use this tuning

SomeOvershoot:
- P: 2609.276
- I: 2745.054
- D: 1651.822

Use this tuning

NoOvershoot:
- P: 1565.565
- I: 1648.681
- D: 991.093

Use this tuning

PControl:
- P: 3913.913
- I: 0
- D: 0

Use this tuning

Figure 3.13: UIs of `PidConfigurationStorage` and `AutoTuningService` during runtime

Although all services in the tuning framework can be configured using Unity's inspector window, there are two services that provide additional UI elements: `PidConfigurationStorage` and `AutoTuningService`. Both of the custom UIs are implemented as subclasses of `UnityEditor.Editor`, which allows user code to extend the default inspector UI. Their extended UI is only visible in play mode, and is only usable when a remote robot has been spawned by the simulation.

`PidConfigurationStorage` is essential for manual tuning, as it allows the user to update the PID configuration of each individual joint without restarting the simulation. The extended UI provides this ability through the Unity editor. Figure 3.13 (left) shows the UI during runtime. "Transmit All PIDs" allows the user to send the entire PID configuration (for each joint) to the simulation. Before a PID configuration is transmitted, any changes are only client-side, and have no effect on the simulation.

"Fill from JSON" opens up a file picker dialog, where the user can select and import a previously exported configuration. Since configurations are saved in the JSON format, such a file could also be produced by external software, including a text editor. The configuration that is extracted from the JSON file replaces the current configuration if it is compatible, i.e. if it provides mappings for all joints. The new configuration is transmitted automatically.

Below the JSON import button, a global PID reset UI can be found. This is meant for rough, initial tunings, where all joints are set to the same configuration. It should only be used before starting a more fine-grained tuning process, where every joint is tuned individually. Since the performance of a single joint, like the shoulder joint, is affected by both the elbow joint and hand joint, it is important to find suitably good starting values when beginning the tuning process. Poorly tuned joints can negatively affect the tuning process of other joints by throwing their momentum around in an unpredictable manner.

Below the reset button, the PID configuration of each individual joint is displayed. For performance considerations, as few joints as possible should be transmitted at once. If you are tuning only a single joint, use the joint-specific "Transmit Joint PID" button instead of constantly transmitting all PID configurations. In rare cases, packet loss has been observed when transmitting all PID parameters at once: Some joints receive their updated mapping, but other joints remain at their old mapping.

`AutoTuningService` is the second service with a custom-built UI and can be seen in Figure 3.13 (right). Its inspector provides multiple options for configuration, which are explained in subsection 3.5.2. Below those, buttons are available which start an automatic tuning process for all joints, or just for the joint that was selected in the drop-down. If no automatic tuning has been performed, this is all you will see in terms of custom UI. After performing auto-tuning for a joint, however, alternative tunings are shown below the buttons. Figure 3.13 (right) shows `AutoTuningService` in such a state. Alternative tunings are based on different heuristics and do not require the auto-tuning session to run again. All options that are listed can also be selected as the default heuristic by changing the "Tuning Heuristic" setting in the inspector. Each heuristic includes a button that replaces the joint's current tuning with the proposed tuning and automatically transmits it to the simulation. The notion behind the different heuristics is explained in subsection 3.5.2.

3.5 Tuning Procedure

This section describes a typical workflow for both manual and automatic tuning. These methods are not at odds with each other; To the contrary, it is often necessary to combine them to get the results you want without spending massive amounts of time on the process. A typical workflow starts with a very rough manual tuning, then runs automatic tuning to refine the PID responses, and finally employs manual fine-tuning to create the perfect, tailor-made configuration.

Both manual and automatic tuning require an initial PID configuration that is at least *stable*. This means that the robot should be at rest, and that it should settle within a reasonable amount of seconds after a set-point change. A good starting point is usually to set I and D to zero and start increasing P. If you find the joint oscillating slowly, or not moving towards its target at all, the PID controller is probably outputting too little force. Double to P-gain until a notable response can be seen. If, on the other hand, the joint oscillates with a quick, jerking motion, try increasing D until the oscillations flatten out. The result is a tuning with a relatively bad response time, but high stability and little overshoot. This is a good starting point for both manual and automatic tuning, as it keeps the disturbance of the joints through the erratic behaviour of other joints to a minimum.

3.5.1 Manual Tuning

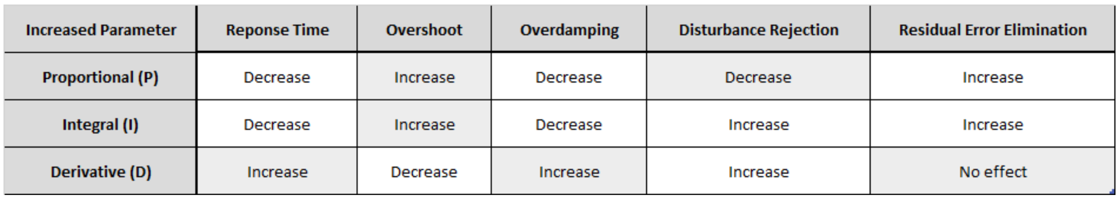
Manual tuning is a viable option for improving the tracking performance of the remote operation use-case. While it is hard to perform for systems with long response times, it is generally much easier for fast-response systems, like the one in our case. Because the robot runs in a simulation instead of being a physical entity, making mistakes doesn't come with the cost of replacing parts of an expensive robot; Instead, we simply reset the simulation. Simulating the robot also allows the user to aggressively test the limits of each joint without having to worry about excessive wear.

Manual tuning is not an exact science: In most cases, some desired response characteristics are at odds with each other. A common example is a quick response time versus little overshoot — Improving one of these metrics negatively influences the other.

It is up to the user to decide on their priorities. Another issue is that all performance characteristics depend on all three of the PID parameters, meaning that a slight change in one of them can completely negate the effect of the other two. Finally, there is the issue of operator preference and task diversity: Some operators might be able to cope with overshoot much better than others. Some tasks require very high precision without much regard for response times. There are countless combinations of preferences and requirements that a proper PID tuning is asked to fulfil; In most cases, you will have to settle for a compromise. This section will provide a guideline for anticipating the effects of changing K_p, K_i, and K_d on the performance characteristics of a joint.

The manual tuning process should be done on a per-joint basis, and not for the whole robot at once. One reason for this are the different weight distributions of the limbs attached to each joint. Driving a finger joint with the same force as an upper leg joint is obviously a bad choice for at least one of them. Another reason is that some joints need to be far more resistant to disturbances than others. A good example is the joint that connects the shoulder to the upper arm: The PID controller has to counteract not only the momentum of the upper arm, but also that of the lower arm and hand, which in turn have their own PID controllers. Since the momentum and response of the whole limb is hard to predict, the shoulder joint usually requires better disturbance rejection characteristics than other joints. It is advisable to start the tuning process at the joints that carry the least weight, e.g. the hand or foot joints.

After creating a rough initial tuning, as described in the introduction to this section, manual tuning can begin. It is usually a good idea to closely watch the joint angle deviation plot while changing PID parameters. Select the joint that you want to tune in the test control window to turn on the graph display. A simple way to gauge the response characteristics of a joint is to perform a step-test: In this test, a large, sudden set-point change is performed. The response of the joint is then observed until the controller has settled on the new set-point. Since step-tests are so common, a special button was added near the joint selection drop-down in the test control window. It switches the sign of the current joint angle, e.g. a joint at 30 degrees will be set to -30 degrees, which amounts to a step test of 60 degrees. Obviously, this only has an effect if the current joint angle is non-zero, so make sure to set it to a suitably large value. Don't forget to turn off the `Animator` component of the local avatar first, since it will override any changes to the joint angle.



Increased Parameter	Reponse Time	Overshoot	Overdamping	Disturbance Rejection	Residual Error Elimination
Proportional (P)	Decrease	Increase	Decrease	Decrease	Increase
Integral (I)	Decrease	Increase	Decrease	Increase	Increase
Derivative (D)	Increase	Decrease	Increase	Increase	No effect

Figure 3.14: Simplified relationship between PID parameters and performance characteristics; Negative influences shown with a grey background

Figure 3.14 contains guidelines for modifying K_p, K_i, and K_d based on the shape of the joint angle deviation plot after a step-test. These guidelines are extremely simplified; To gain an in-depth understanding of the effect of K_p, K_i, and K_d in isolation, the user should perform investigative step-tests on a joint of their choice.

One notion that is not captured in Figure 3.14 is the different effects of K_p and K_i on overshoot. While both can potentially increase overshoot, higher P values will lead to fast oscillations around the new set-point. Higher I values, in contrast, will lead to very slow, drawn-out oscillations. The effect of both parameters on response times is also not straight-forward: Increasing K_p will decrease the initial response time much more effectively than increasing K_i, but a high value for K_i will catch up and out-scale the P-term after a short amount of time. The effects of the I-term are also increased when K_d is increased, since the flattened graph allows for the integral to accumulate error values for a longer amount of time. Another interesting relationship between K_i and K_d can be found in disturbance rejection: While the D-term applies a flattening action to the error graph, the I-term can be interpreted as the 'momentum' of the controller, resisting any sudden changes in output. The D-term can thus be seen as a useful way to counteract small, short-lived disturbances, while the I-term can also deal with prolonged disturbances, like the force of wind. Finally, residual error elimination is influenced in very distinct ways by K_p and K_i. While both decrease the residual error, only the I-term is able to completely eliminate it. Using the P-term alone will always leave some small residual error, which is especially noticeable if there is an external force (like gravity) acting on the joint. In practice, however, this error is often so small that it can be safely ignored.

3.5.2 Automatic Tuning

Automatic tuning performs an automated test to determine the response characteristics of a joint. It then derives a PID configuration based on those measurements and applies it to a joint. The process is called automatic because the only user interaction that is required is to start the process.

The results of automatic tuning often cannot achieve the same level of performance as properly executed manual tuning. It relies on heuristics to calculate ’safe’ PID tunings, usually with high levels of disturbance rejection. This sacrifices other desirable traits, such as response time or overshoot reduction. Still, an automatic tuning routine equipped with suitable heuristics can achieve very good results for little effort. Occasionally, it is also the only option, if the the user is not familiar with the process of manual PID tuning.

Generally, it is useful to run an automatic tuning routine at least once before attempting to manually tune a joint controller, since it requires no user knowledge or effort. By identifying the tuning heuristic that comes closest to your desired result, you can save significant amounts of time by using the automatic tuning routine as a starting point. Advanced users can even provide their own, purpose-built heuristic, which can render manual tuning completely obsolete.

The automatic tuning process can be started by calling the appropriate functions in the public API of `AutoTuningService`, or by using the UI extension the `AutoTuningService` component in the inspector. Although there is a choice to tune all joints at once, this can turn out to be a very long process, so it is recommended to go through the joints one by one. This also allows the user to determine which heuristic is most suitable for the joint. Additionally, the user can exploit symmetries in the robot model to save time: The automatic tuning service has no information about symmetries, so it will, for example, just repeat the automated test for the left arm even when the right arm was already tested. The user can skip the test for the left arm by simply re-using the tuning of the right arm.

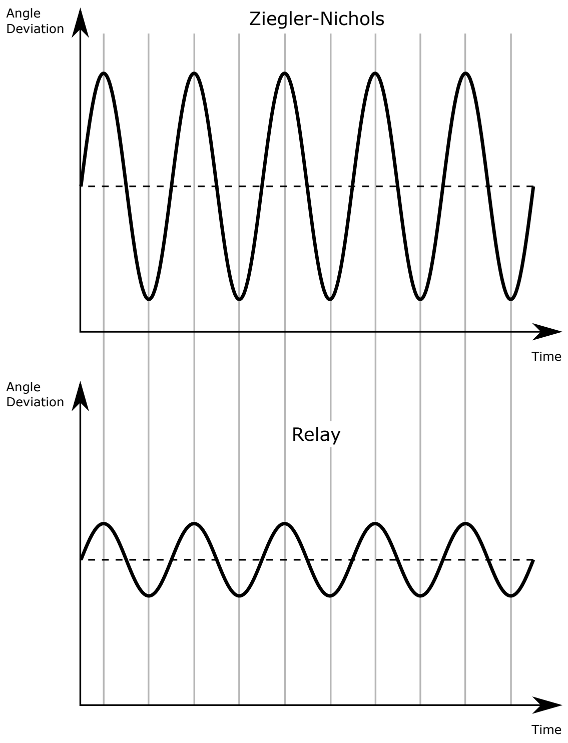


Figure 3.15: Advantage of the relay tuning method in comparison to Ziegler-Nichols

The automated test that is performed by the tuning service is called a relay test [Ber17]. This method of testing was developed out of the necessity to overcome the shortcomings of the Ziegler-Nichols method, which had been (and still is) a de facto standard in PID tuning. Ziegler-Nichols requires the user to stress the PID controller up to the limits of the joint by increasing K_p until the controller produces consistent oscillations, on the verge of becoming unstable. This requirement is not only tricky to achieve, but also poses the risk of the oscillations getting out of hand and damaging the joint. It also increases the wear on the joint. The relay method avoids these pitfalls by creating consistent oscillations at much lower amplitudes; In fact, the relay method allows the user to *control* the amplitude of the oscillations. A comparison between the joint angle deviation graphs during Ziegler-Nichols tuning and during relay tuning can be seen in Figure 3.15. Note that the oscillation *period* remains the same for both methods.

One disadvantage of the relay test is that it requires a slightly more complex setup than a Ziegler-Nichols test: The PID controller needs to be replaced by a so-called relay, which can only create two distinct constant outputs. We chose these outputs to be a large force in one direction, and a equally large force in the other direction. This force can be configured on `AutoTuningService` by modifying the property `RelayConstantForce`. A good starting value is the maximum force that the joint can exert, if that value is known. The relay is the reason why `AutoTuningService` cannot be entirely implemented within the Unity client: It needs direct access to the physics engine in the simulation.

The idea behind the relay test is to switch between these two forces depending on the sign of the error between set-point (desired angle) and process variable (measured angle). This will create oscillations around the set-point, while at the same time making sure that the process never settles. For this method to work, the PID controller needs to be deactivated completely. The tuning framework achieves this by temporarily setting K_p, K_i, and K_d all to zero. The output might need some time to create consistent oscillations: They should neither grow, nor shrink in amplitude, and should have its peaks at a fairly constant distance. `AutoTuningService` provides the setting `TestWarmupSeconds`, which should be set high enough for the process to settle into consistent oscillations before the actual test starts. `RelayStartAngle` can be set to an estimate of the zero-to-peak amplitude of the oscillations. This will lower the time that the relay test needs to achieve consistent oscillations. Due to angle limits, `RelayTargetAngle` sometimes needs to be adjusted to allow the joint to oscillate freely, without hitting any limit. The elbow joints, for example, cannot oscillate into negative degrees.

The measurements from a relay test are used to calculate two metrics: The ultimate period, and the ultimate gain. In the Ziegler-Nichols method, the ultimate period is the period of the oscillations (distance between two same-sign peaks) and the ultimate gain is the amplitude (between positive and negative peaks). For the relay method, the ultimate gain must calculated in a slightly different way, but should yield the same results in a typical scenario [Hor]. Since the relay method produces the same ultimate parameters as Ziegler-Nichols, their heuristics are entirely compatible.

After the warmup phase, the relay test will run for `TestDurationSeconds` seconds. This value should be high enough that at least six peaks (positive or negative) are captured. More peaks will improve the accuracy of the result. Once a test is completed, the tuning service will automatically modify the joint controller according to the test results and transmit the new configuration to the simulation. The tuning that is automatically applied uses the heuristic that was specified in `TuningHeuristic`. If the tuning turns out to be unsatisfactory, other heuristics are available through the `LastTuningData` property of `AutoTuningService`. The same heuristics can also be selected through the UI, which can be seen in Figure 3.13 (right). Note that this selection only concerns the joint the was tuned most recently; Older heuristics are not saved and need to be re-calculated from a test. The remainder of this section will explain all available heuristics. If your interest is in not the design of the heuristics, but in their performance, read section 4.1 instead.

The *Classic* heuristic uses the formulas that were proposed by Ziegler and Nichols in their original publication [ZN93]. It provides quarter-wave decay, meaning that each oscillation will have at most a quarter of the amplitude of the previous one. This heuristic is relatively aggressive; It has a good response time, but comes with large amounts of overshoot and a very long settling times. It was designed with disturbance rejection in mind and does an excellent job in that respect.

The *PessenIntegralRule* heuristic produces even more overshoot due to its very large integral term. Like the Classic heuristic, it produces a fast response time, but it puts even more emphasis on disturbance rejection.

SomeOvershoot and *NoOvershoot* are modifications the original classic rule to eliminate some of the excessive overshoot that it often exhibits. They sacrifice response time and disturbance rejection to achieve this result. Since the proportional gain is very small compared to other heuristics, the settling time is also not among the best, but usually still outperforms rules with higher integral gains.

PControl, *PIControl*, and *PDControl* all disable one or more of the terms in the PID controller. In practice, there are some scenarios where it turns out that the I-term or D-term are detrimental to the controller's performance. The I-term can become unnecessary if there are no long-term disturbances present in the controlled process. An example for such an environment is space, where the robot is not under the influence of gravity or wind. The D-term is counter-productive whenever there are many large short-term disturbances in a system, since it is based on the error derivative. An example for this would be a robot that is mounted on a heavily vibrating vehicle. In such a case, the D-term can cause erratic behaviour of the controller, so P-control or PI-control should be preferred.

4 Results and Conclusion

Due to time constraints, only the automatic tuning performance of the tuning framework was tested. Manual tuning could be tested by conducting a user study and recording the time it takes for users to reach a satisfactory tuning for the robot. Such a study would need a very large sample size, since individuals might have very different experiences with controller tuning, or the tuning process in general. It would also be problematic to separate the effectiveness of the framework from the adaptiveness of the user, which both have a major impact on manual tuning performance. In contrast, the automatic tuning routine can be tested and analyzed entirely according to quantifiable criteria.

4.1 Automatic Tuning Performance Evaluation

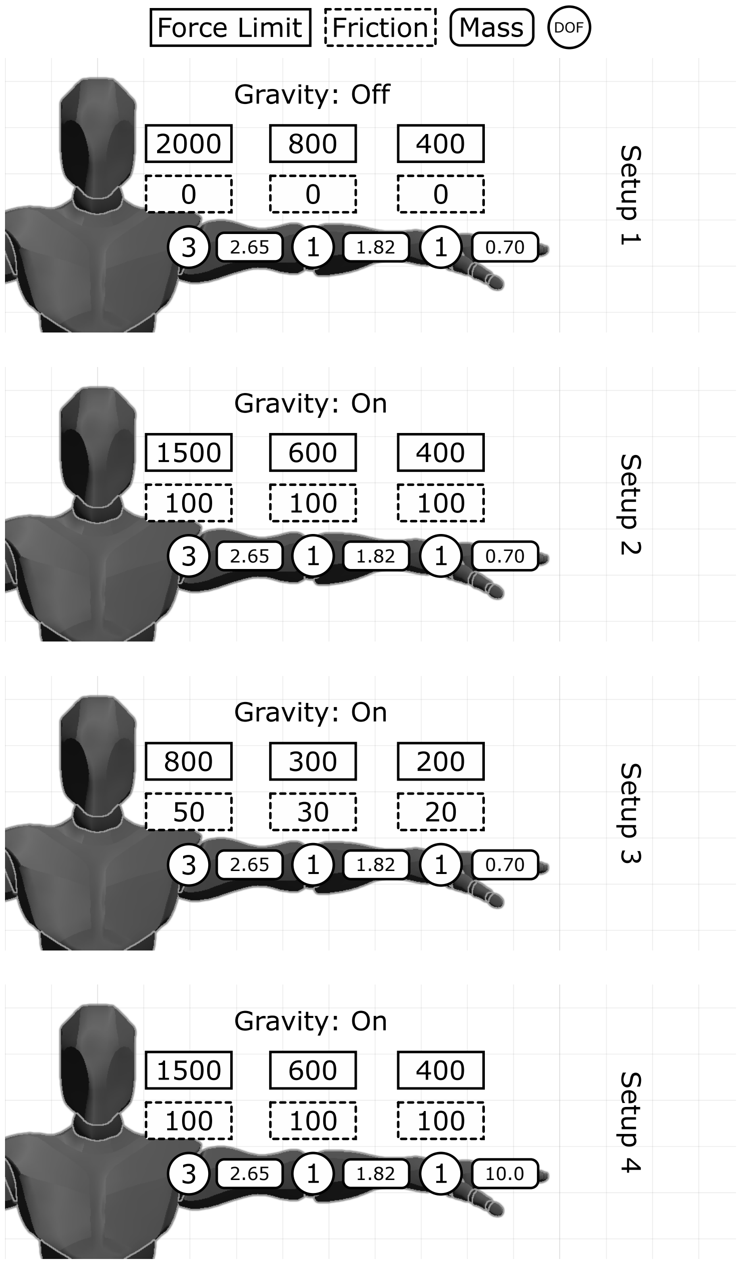


Figure 4.1: Force limits, friction, DOF, and masses of all joints in the four test setups

The main purpose of the automatic tuning service is to give acceptable results for many different robot setups. Finding an *optimal* tuning is not a priority. Heuristics generally favour disturbance rejection over response time and little overshoot, often making the response feel slightly sluggish. Nonetheless, it is important that the automatic tuning service produces usable results. This section compares the performance of different heuristics for the automatic tuning routine, with a carefully attained manual tuning as a baseline. It is important to understand that a professional manual tuning should usually be preferred to automatic tuning, if enough expertise and time are available.

The performance of all heuristics was evaluated based on four major criteria: Average absolute error, 50% response time, 5 degrees settling time, and maximum overshoot. This limitation was imposed so that a graphical representation of the results would not be too cluttered. 50% response time usually gives a good indication of all three response time metrics, and a similar argument can be made for 5 degrees settling time.

The Gazebo test scene is empty with the exception of the robot. Although it would be preferable to test the interaction of the robot with a carefully set up environment, time constraints made this impossible. The only variations between test runs, other than the PID tuning, are the physical properties of the robot, according to its simulation description file (SDF, see subsection 2.3.1). It was modified to represent a wide range of humanoid robots by changing the weight distribution, force limits, and damping characteristics of the entire model. Keep in mind that the different test setups are not even close to being exhaustive: Many architectures for humanoid robots exist, and it would be beyond the scope of this thesis to test them all. Instead, an effort was made to show four representative cases. These different setups are explained at the end of this section, and in Figure 4.1.

All tests were conducted in the same manner: First, each joint within the robot is given a baseline tuning of $K_p = 10000$, $K_i = 1000$, $K_d = 1000$. This tuning achieves an aggressive settling time for all joints in all test cases. Starting from a consistent baseline is important, since the performance of the automatic tuning service relies on having a somewhat stable initial tuning. The second step is to tune all joints in the left arm (three shoulder joints plus one elbow and one hand joint) according to the same heuristic. Joints are tuned in order of decreasing distance from the core of the robot, meaning that the hand joint gets tuned first. After tuning, joints are tested with a fast-paced dance animation, which provides the test result for the *average absolute error in animation* metric. Next, the arm is subjected to a pre-recorded step-test animation, which provides additional response metrics: *Average absolute error in step-test*, *50% response time*, *5 degrees settling time*, and *maximum overshoot*. The animation consists of one 45 degree step change for each joint within the arm, with a pause between each change to prevent interference between the joints. After all tests are concluded, the joints are tuned according to the next heuristic. With a total of 4 setups and 7 heuristics plus one manual tuning, 32 complete test runs are needed in total.

All four setups can be seen in Figure 4.1. The first setup is intended to serve as a baseline: With no gravity and very high force limits, this setup is meant to represent a best-case control scenario without any disturbances. Friction was disabled completely, and the mass distribution of the arm is based on a male human [Pfe10]. This setup is not realistic, although it is not unthinkable that a robot in space might be in a comparable situation due to the lack of gravity and air. Because of the high force limits, the robot should be able to achieve fast settling and response times.

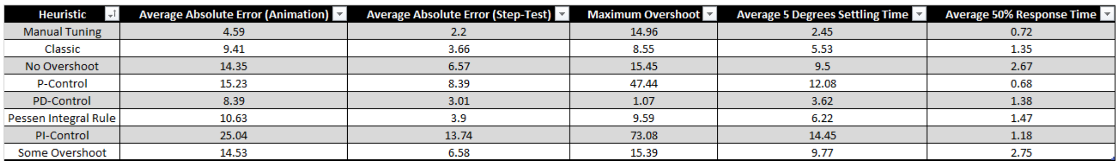
The second setup attempts to create a more realistic scenario: Gravity is enabled, and the joints generally have lower force limits and some amount of friction. The weight distribution is the same as in the first setup. This setup is probably the closest to current humanoid robots, and should generally provide the most widely applicable test results.

In the third setup, all force limits were severely lowered. Gravity is still enabled, and the weight distribution is the same as in the first two setups. Due to the lower force limits, friction was slightly decreased, so movement was still possible at a meaningful speed for all joints. This setup is intended to be a stress-test for the automatic tuning routine: The low force limits primarily increase settling and response times, meaning that the integral term has much more influence than in other setups. The tuning heuristics need to handle this situation by lowering the I-term, or by providing strong counter-action through a high P term. Another point of interest will be how the tuning heuristics handle a controller at its maximum force output for prolonged amounts of time.

The final setup is almost identical to the second setup, but has a much higher mass for the hand. This is intended to simulate the act of carrying a heavy weight. Due to most of the mass being concentrated in the end of the arm, the effect is similar to a pendulum, which is notoriously hard to control. Joints in the upper arm need to be able to counteract the high-inertia overshoot of the hand joint. The hand joint itself should still be able to provide a meaningful control response. In general, this setup should benefit much more from heuristics with higher disturbance rejection characteristics and large I-terms.

The results for all four test setups are shown in both tabular and graphical form on the next pages. The units for the data are degrees for the error and overshoot metrics, and seconds for the settling time and response time metrics. In the chart, the upper half of the bar graph shows maximum overshoot in degrees. The other metrics are shown in the bottom half and share their y-axis. Note that it is usually not meaningful to compare error metrics to time metrics due to their different units.

4.1.1 Setup 1 Results



Heuristic	Average Absolute Error (Animation)	Average Absolute Error (Step-Test)	Maximum Overshoot	Average 5 Degrees Settling Time	Average 50% Response Time
Manual Tuning	4.59	2.2	14.96	2.45	0.72
Classic	9.41	3.66	8.55	5.53	1.35
No Overshoot	14.35	6.57	15.45	9.5	2.67
P-Control	15.23	8.39	47.44	12.08	0.68
PD-Control	8.39	3.01	1.07	3.62	1.38
Pessen Integral Rule	10.63	3.9	9.59	6.22	1.47
PI-Control	25.04	13.74	73.08	14.45	1.18
Some Overshoot	14.53	6.58	15.39	9.77	2.75

Figure 4.2: Tabular test results for setup 1

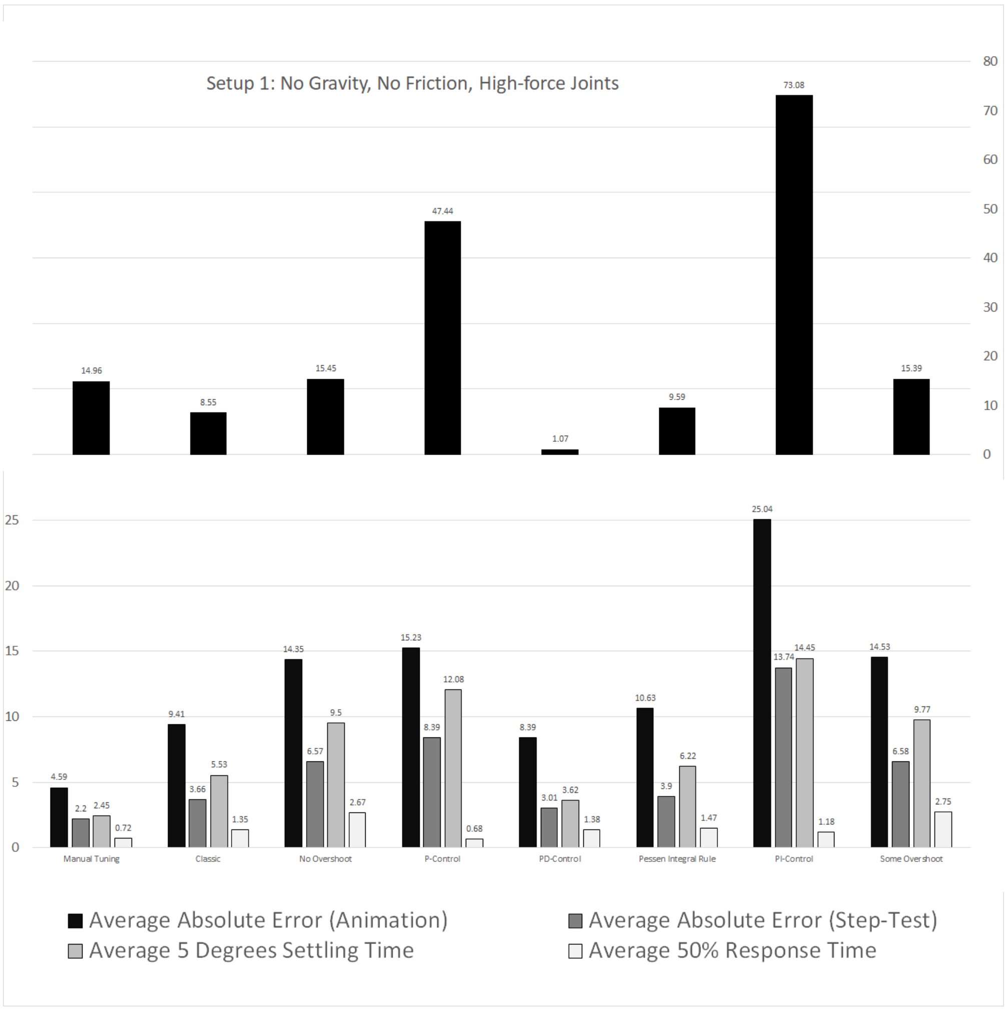


Figure 4.3: Graphical test results for setup 1; Top half shows maximum overshoot, bottom half shows all other metrics

Unsurprisingly, the manual baseline tuning shows superiority over all automatic tuning heuristics in most metrics. Its average error in the dancing animation is almost half of the next-best contender, *PD-Control*. Interestingly, this difference is less pronounced in the step test error: It seems that the automatic tuning heuristics perform better with large, isolated step changes than with continuous set-point tracking. The average absolute error of *PD-Control* in the step test is only 0.8 degrees larger than that of manual tuning, which might be hard to notice during normal operation.

The most obvious losers in this setup are *P-Control* and *PI-Control*. Both heuristics exhibit massive overshoot of over 45 degrees, making them essentially useless in all but the most lenient working environments. Interestingly, *PI-Control* performs measurably worse than pure *P-Control*, even though the heuristic has access to an additional controller term. This means that in this setup, *PI-Control* would be a better heuristic if the I-term was set to zero, basically transforming it into *P-Control*. Both *P-Control* and *PI-Control* have noticeably worse settling times than all other heuristics, with exception of the *No Overshoot* heuristic. Twelve or fourteen second settling times indicate heavy oscillations after a set-point change, basically prohibiting useful operation for at least ten seconds. Response times, in contrast, are generally favourable; *P-Control* even beats manual tuning when if comes to response time. Still the price you pay for this quick response in terms of average error, overshoot, and settling time seems too high to make the trade-off worth it.

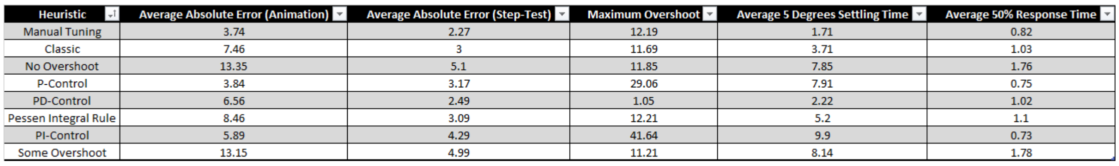
One of the most surprising results is the exceptionally poor performance of the *No Overshoot* and *Some Overshoot* heuristics in terms of overshoot. Despite being designed to minimize that very metric, both fail spectacularly at their intended task. The *Classic* tuning beats out the overshoot heuristics by a factor of almost two. *No Overshoot* and *Some Overshoot* are not only poor at controlling overshoot, they are also the worst overall performers after *P-Control* and *PI-Control*. Although they generally achieve faster settling times and lower average error metrics in the step tests, they still generate fairly unusable tunings. One of the most striking metrics is their 50 percent response time, which is almost twice as long as the next-worst contender, *Pessen Integral Rule*. Joints need over 2.5 seconds to reach the half-way point after a step-change. Such sluggish behaviour is generally not desirable and offers an explanation for the poor tracking performance of both metrics.

The clear winner of all automatic tuning heuristics is *PD-Control*. It has the lowest average error metrics among all contenders, as well as the lowest settling time by a margin of over 50%. Although its tracking performance and step-change error are not much lower than that of the *Classic* heuristic, *PD-Control* is the obvious best choice because of its low settling time, which is within 1.2 seconds of the manual tuning. A very important observation is that *PD-Control* causes almost no overshoot. With only 1 degree maximum overshoot, the tuning proves to be exceptionally useful in environments where precise motion is absolutely necessary, like in surgical robots. *PD-Control* outperforms manual tuning by a factor of fourteen in terms of overshoot. This indicates that the manual tuning is overly aggressive for some workloads, and that the automatic *PD-Control* heuristic actually gives better results here.

The *Classic* heuristic deserves an honorable mention for performing second-best out of all automatic tuning heuristics. In work environments where an integral term is essential due to many external disturbances, it should be favoured over *PD-Control*, which has no I-term at all. *Pessen Integral Rule* performs very similarly to *Classic* in every metric, although always slightly worse. Both metrics clearly beat out the manual tuning in terms of overshoot, but still perform much worse than *PD-Control* in that regard.

In summary, useful tunings were generated by the *PD-Control*, *Classic*, and *Pessen Integral Rule* heuristics in order of decreasing performance. An explanation for the exceptionally good performance of *PD-Control* is that gravity was not present in this setup, meaning that no constant disturbances were affecting the joints. Since this means that there is generally no residual error, the I-term becomes useless and actually decreases controller performance. All results should be viewed with discretion; The setup is not very realistic, and the response of the robot to external forces generated by interaction with the environment were not tested at all. *PD-Control* would most likely perform much worse if the robot was instructed to carry a weight.

4.1.2 Setup 2 Results



Heuristic	Average Absolute Error (Animation)	Average Absolute Error (Step-Test)	Maximum Overshoot	Average 5 Degrees Settling Time	Average 50% Response Time
Manual Tuning	3.74	2.27	12.19	1.71	0.82
Classic	7.46	3	11.69	3.71	1.03
No Overshoot	13.35	5.1	11.85	7.85	1.76
P-Control	3.84	3.17	29.06	7.91	0.75
PD-Control	6.56	2.49	1.05	2.22	1.02
Pessen Integral Rule	8.46	3.09	12.21	5.2	1.1
PI-Control	5.89	4.29	41.64	9.9	0.73
Some Overshoot	13.15	4.99	11.21	8.14	1.78

Figure 4.4: Tabular test results for setup 2

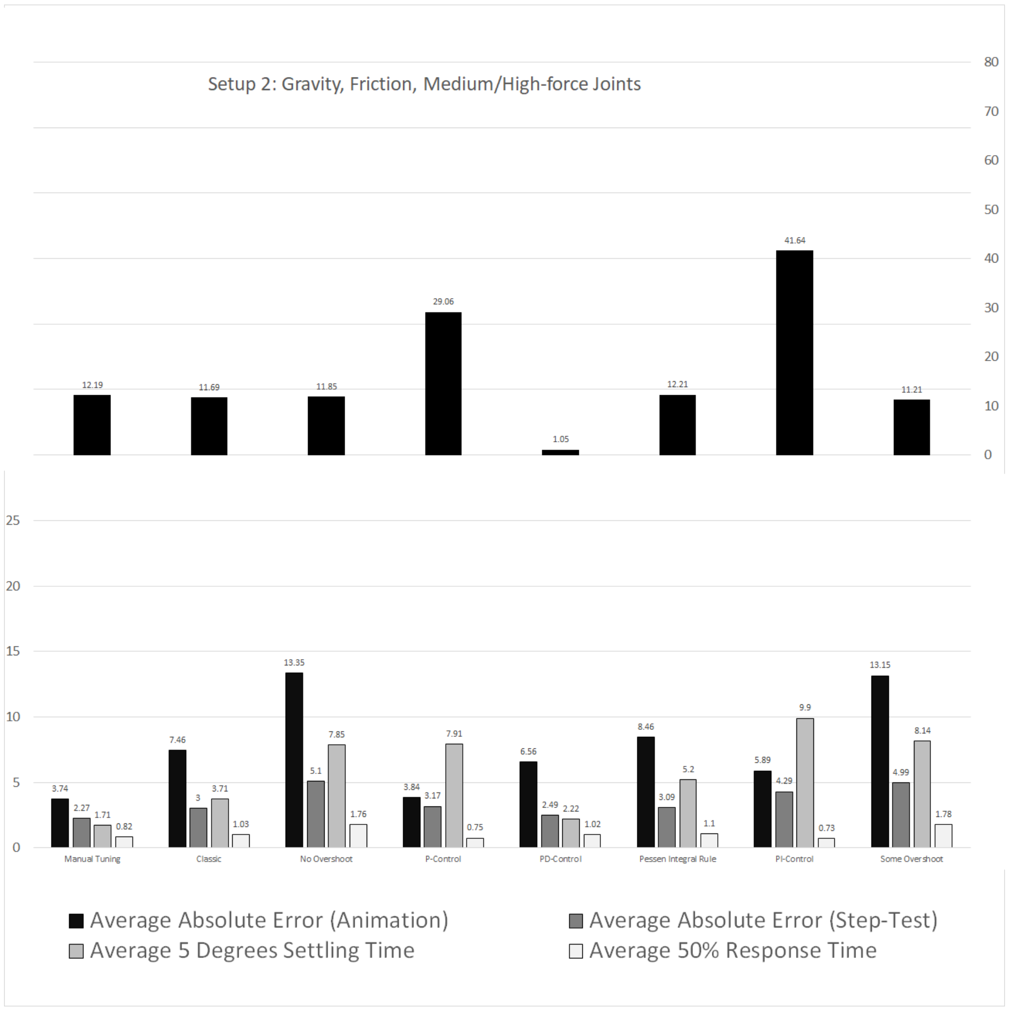


Figure 4.5: Graphical test results for setup 2; Top half shows maximum overshoot, bottom half shows all other metrics

It is immediately obvious that all automatic tuning metrics perform better in setup two than setup one. The biggest winners, by far, are *P-Control* and *PI-Control*: While they had the largest average errors in the original setup, they are now among the top contenders. *P-Control* actually manages to match the animation-tracking performance of the manual tuning to within 0.1 degrees. Step-change tracking is also improved by a factor of more than two, but not as dramatically as animation tracking.

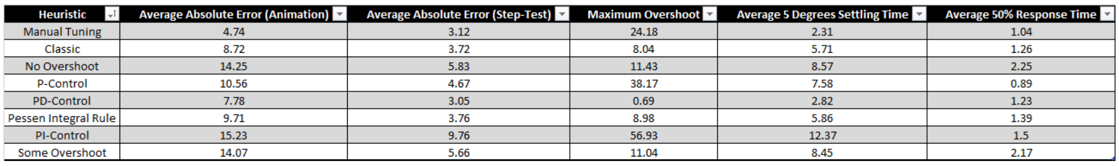
Although *P-Control* and *PI-Control* now provide excellent tracking, they are unfortunately still not very useful: Their tendency to create extreme overshoot and oscillations is still present after the setup change. *P-Control* still has more than twice the overshoot of any other metric except *PI-Control*, which performs even worse. The settling times of both metrics are also among the worst: With over 7.5 seconds, these heuristics cannot even get close to the 1.71 second settling of the manual tuning. Such a difference is very noticeable in remote operation, and basically renders *P-Control* and *PI-Control* useless.

A similar argument can be made against the overshoot heuristics, *No Overshoot* and *Some Overshoot*. Both of them actually show the smallest improvement in terms of animation tracking, with their average error metric decreasing by only 1 to 1.4 degrees. Due to the massive improvements of *P-Control* and *PI-Control* in that area, the overshoot heuristics are now actually the worst performers among all automatic tunings. Again, the *No Overshoot* heuristic ironically produces a tuning that has *more* overshoot than the one created by the *Some Overshoot* heuristic, although the difference is negligible. Both heuristics manage to avoid aggressive oscillations, unlike *P-Control* and *PI-Control*, but their settling time is still among the worst of the bunch. Similarly, their response times are more than twice the benchmark time set by manual tuning, 0.82 seconds. The change in setup was not enough to make *No Overshoot* and *Some Overshoot* viable.

The three top contenders from setup one are still the best performers overall. Generally, all of their metrics showed slight improvement after the setup change, but the overall performance profile remained the same. Interestingly, *PD-Control* is still the clear winner, even though the disturbance caused by gravity favours a non-zero I-term. It seems that *PD-Control* is able to counter-act this disturbance by applying a massive P-term. This decreases the residual error to a level where it is barely noticeable. *Classic* and *Pessen Integral Rule* still give *PD-Control* a run for its money, but cannot quite achieve the same level of performance. The most striking disadvantage of both heuristics is overshoot. *Classic* and *Pessen Integral Rule* are the only heuristics that performed worse in terms of overshoot after the setup change. While 12 degrees of overshoot for a 60 degree set-point change might be acceptable in some conditions, it still cannot match the 1 degree overshoot produced by *PD-Control*. The manual tuning, however, also produces around 12 degrees of overshoot, again showing that response times were favoured over overshoot reduction during the tuning process.

In summary, the change in setup positively affected most automatic heuristics, but the same heuristics as in setup one still come out as the clear winners. It seems that the more realistic conditions in setup two come closer to the conditions that the heuristics were designed for. Although the manual tuning, as expected, performs better in almost every regard, *PD-Control*, *Classic*, and *Pessen Integral Rule* show consistent performance improvements after the change. *PD-Control* even manages to get its set-point tracking performance to within 0.25 degrees of the manual tuning, and its settling time to within 0.5 seconds. It cannot quite match the animation tracking performance, but is a clear winner in terms of overshoot.

4.1.3 Setup 3 Results



Heuristic	Average Absolute Error (Animation)	Average Absolute Error (Step-Test)	Maximum Overshoot	Average 5 Degrees Settling Time	Average 50% Response Time
Manual Tuning	4.74	3.12	24.18	2.31	1.04
Classic	8.72	3.72	8.04	5.71	1.26
No Overshoot	14.25	5.83	11.43	8.57	2.25
P-Control	10.56	4.67	38.17	7.58	0.89
PD-Control	7.78	3.05	0.69	2.82	1.23
Pessen Integral Rule	9.71	3.76	8.98	5.86	1.39
PI-Control	15.23	9.76	56.93	12.37	1.5
Some Overshoot	14.07	5.66	11.04	8.45	2.17

Figure 4.6: Tabular test results for setup 3

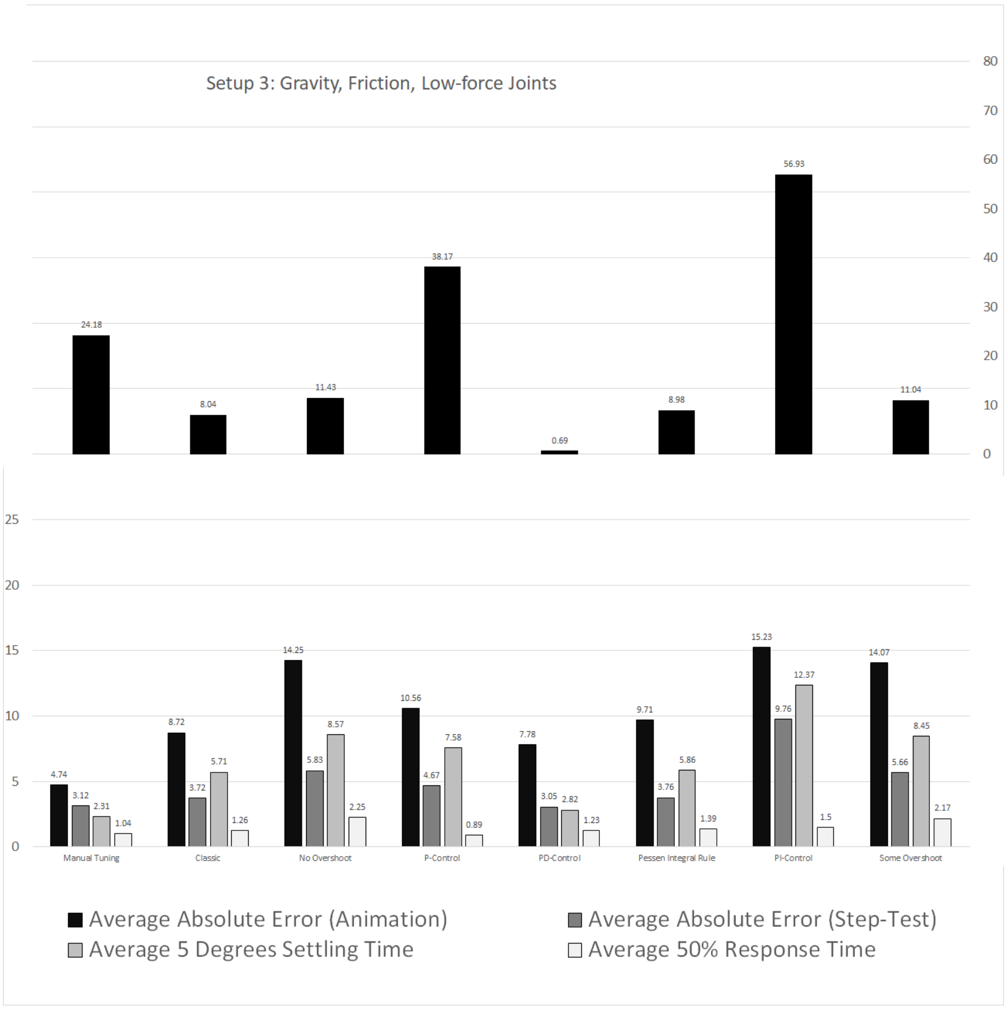


Figure 4.7: Graphical test results for setup 3; Top half shows maximum overshoot, bottom half shows all other metrics

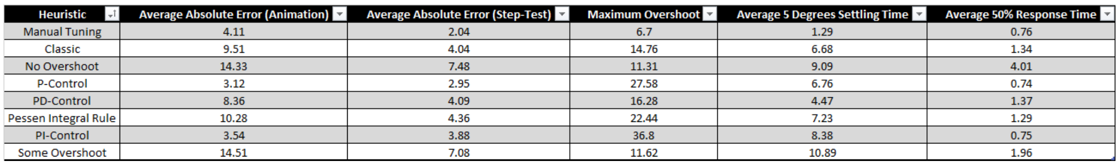
Unsurprisingly, the low force setup decreases the performance of all heuristics in most metrics. *P-Control* and *PI-Control* lose all the progress they made in setup two and go right back to being among the worst of all heuristics. Their overshoot increased dramatically, which can be explained by inertia: While *P-Control* and *PI-Control* manage to accelerate fairly quickly after a set-point change, they have trouble with reversing their output to avoid overshoot. The large I-term of *PI-Control* amplifies this effect, leading to even more overshoot.

P-Control and *PI-Control* also perform much worse in terms of animation tracking than in setup one and two. It seems that their good performance in setup two was owed to low-amplitude oscillations around the set-point, which led to a small average error metric. In setup three, where such oscillations are much harder to achieve due to low force limits, both heuristics oscillate much slower, with much more overshoot. The combination of these effects leads to a significantly worse tracking performance.

No Overshoot and *Some Overshoot* remain consistent in their poor performance. While their overall response profile does not change much between setups, these heuristics are not able to deliver a top-three tuning for any of them. It is however notable that they manage to keep their overshoot roughly the same as in setup two, unlike *P-Control* and *PI-Control*.

Again, and unsurprisingly, *PD-Control*, *Classic*, and *Pessen Integral Rule* are the best-performing heuristics. Remarkably, *PD-Control* even manages to beat the manual tuning in two out of the five metrics, namely overshoot and set-point tracking in the step-test. The manual tuning in this setup might be over-aggressive, since it is beaten by five out of the seven automatic tuning heuristics in terms of overshoot. *Classic* and *Pessen Integral Rule* manage to reduce their overshoot compared to setup two, which is an impressive feat, given that the force limits are now so much lower. These heuristics seem to be able to perform well even in sub-optimal setups.

4.1.4 Setup 4 Results



Heuristic	Average Absolute Error (Animation)	Average Absolute Error (Step-Test)	Maximum Overshoot	Average 5 Degrees Settling Time	Average 50% Response Time
Manual Tuning	4.11	2.04	6.7	1.29	0.76
Classic	9.51	4.04	14.76	6.68	1.34
No Overshoot	14.33	7.48	11.31	9.09	4.01
P-Control	3.12	2.95	27.58	6.76	0.74
PD-Control	8.36	4.09	16.28	4.47	1.37
Pessen Integral Rule	10.28	4.36	22.44	7.23	1.29
PI-Control	3.54	3.88	36.8	8.38	0.75
Some Overshoot	14.51	7.08	11.62	10.89	1.96

Figure 4.8: Tabular test results for setup 4

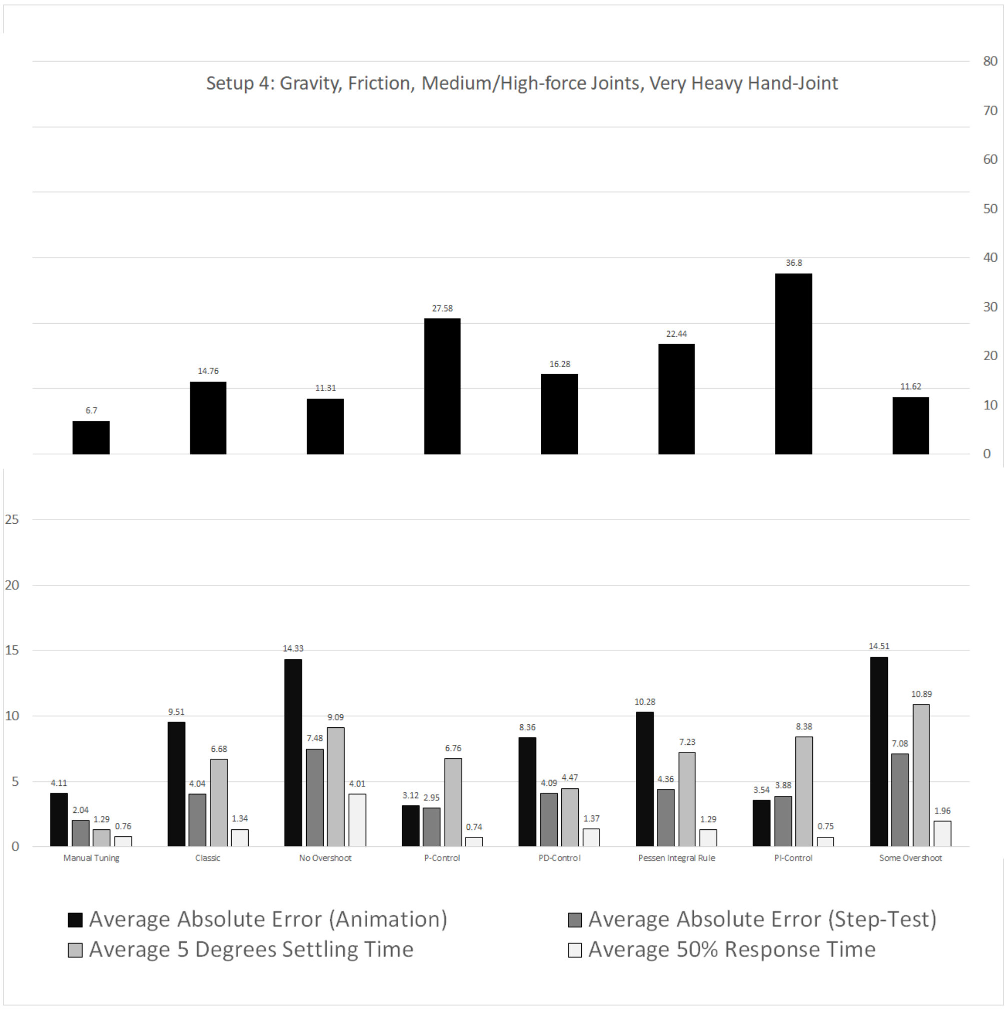


Figure 4.9: Graphical test results for setup 4; Top half shows maximum overshoot, bottom half shows all other metrics

Setup 4 provided some of the most surprising results. While the usual suspects, *No Overshoot* and *Some Overshoot*, delivered their expected sub-par performance, *P-Control* and *PI-Control* both managed to significantly improve compared to the very similar setup 2. Although both heuristics still produce large overshoot and have very long settling times of over 6.5 seconds, they actually manage to beat the manual tuning in terms of animation tracking performance. This feat is not accomplished by any of the other automatic tuning metrics. For use cases favouring fast reaction times over quick settling times, *P-Control* and *PI-Control* might actually outperform the manual tuning.

The biggest loser in this setup is unquestionably *PD-Control*. While usually the top contender, *PD-Control* now suffers from large overshoot. It is outperformed by the *Classic*, *No Overshoot*, and *Some Overshoot* heuristics, and of course the manual tuning. Its set-point tracking performance for animation playback is beaten by *P-Control* and *PI-Control* by a factor of 2, and its tracking performance in the step test is also slightly worse. The step-test tracking metric is matched by *Classic* and *Pessen Integral Rule*, a feat which they were not able to achieve in any of the other setups.

The surprising winners of setup 4 are *P-Control* and *Classic*, unless settling time is an absolute priority; In that case, *PD-Control* still performs best. *P-Control* clearly matches or beats *Classic* in all but the overshoot metric, where *Classic* is the winner by a factor of almost two. Interestingly, *Pessen Integral Rule*, which is usually very close to *Classic*, performs much worse in terms of overshoot. It seems that the massive I-term hurts the metric when it comes to large set-point changes.

In summary, the unpredictability of the pendulum-like setup 4 leads to the most ambiguous results yet. *PD-Control* wins only in terms of settling time, but is clearly beaten by other heuristics in every other metric. There is, however, no general champion among the heuristics: While *P-Control* offers very good tracking and response times, it suffers from massive overshoot. *Classic* remedies the overshoot issue, but performs much worse in terms of tracking performance. *PD-Control* is actually pretty close to *Classic*, but generates slightly more overshoot. *Classic* should be preferred due to its non-zero I-term, which offers much better disturbance rejection.

All automatic heuristics perform poorly in terms of settling time. The best contender, *PD-Control*, is still very far from the 1.29 seconds achieved by manual tuning. It seems that setup 4 leads to excessive amounts of overshoot and oscillations, increasing all settling times significantly. Without inherent knowledge about the exact setup, all automatic tuning routines over-compensate for the inertia caused by the 10 kg hand. Manual tuning can attempt to eliminate excessive hand-motion directly at the hand joint, making tuning of the other joints much easier.

4.1.5 Conclusion

While interpreting these results, it is important to keep in mind that only four setups and a limited number of heuristics were tested. Although these setups were designed to model representative cases, they only cover a small portion of all humanoid robots. The heuristics that were chosen for the test runs are industry standards, and were not modified at all for the remote operation use case. It is reasonable to assume that better heuristics could be developed for this specific scenario. Another shortcoming of the test setups is that they only test tracking performance and motion response, but not interaction with the robot's environment. This is especially important when the robot is supposed to interact with humans, because strict force limits need to be adhered to in such a scenario. All of these caveats limit the applicability of the results; Nonetheless, the general notions should apply to a wide range of use-cases.

Unsurprisingly, the big winner of the test runs was manual tuning, which consistently outperformed almost all of the automatic tuning heuristics. One of the greatest benefits of manual tuning is the ability of the user to adapt the tuning to the wishes of the operator, the work environment, and the specific physique of the robot. This ability is especially important in the use-case of remote operation, since the operator might spend a long amount of time in their surrogate robot body.

Automatic heuristics need to take a careful approach to tuning, as they need to work in many different setups. This typically means that a manual tuning can allow itself to be far more aggressive than any automatic heuristic. One of the most striking results can be found in setup 4: No automatic tuning gets close to manual tuning in terms of settling time, with the best contender being over three times slower. This can be explained by the ability of the manual tuner to see 'the whole picture', meaning that they can tune joints while keeping in mind the performance of other joints. By eliminating strong oscillations at the source (hand joint), the manual tuning allows the elbow and shoulder joints to have less disturbance rejection, decreasing their settling times significantly. Automatic tuning heuristics do not have this luxury; They tune each joint in isolation.

Another interesting observation is that automatic tuning heuristics generally performed better in the step-test than in the animation tracking test. It seems that the heuristics mostly favour the response to large, single set-point changes over animation tracking. This is very detrimental to the remote operation use-case, where continuous tracking performance is favoured. It is not impossible, however, that other heuristics with better tracking performance exist.

Settling time was another clear weakness of all heuristics, which often generated tunings that took more than twice as long as the manual reference tuning to settle within 5 degrees. Again, this is a problematic result for remote operation.

All automatic tuning heuristics performed better in simple, realistic environments, like setup 2. As soon as some complications were introduced in setups 3 and 4, their performance decreased drastically. Interestingly, all heuristics performed worse in setup 1 than in setup 2, despite setup 1 having no disturbances like gravity and friction, and joints with very high force-limits. It can be assumed that the heuristics were designed for more realistic setups, and thus result in overly cautious tunings. This behaviour warrants further investigation.

The best performing heuristic overall was *PD-Control*, a result that is somewhat surprising, as the heuristic does not utilize the I-term of the PID-Controller at all. This implies that the test environment was largely disturbance free, and that the P-term was enough to counter-act the effects of gravity in all joints. *PD-Control* consistently achieved the lowest settling times, and had the lowest overshoot in all but setup 4. This makes the heuristic useful for both precise remote operation tasks, like surgery or bomb disposal, and reactive tasks, like the traversal of uneven terrain. Overall tracking performance, however, was sometimes worse than that of more aggressive tunings, like *P-Control*. Such tunings typically also achieved faster response times. Although *PD-Control* is the overall winner, there might be scenarios where its performance is sub-par; Setup 4 provides an example.

Despite being the subject of frequent criticism, the *Classic* heuristic from the original paper by Ziegler and Nichols performs remarkably well in all setups. It is especially impressive how consistent it performs even in complicated scenarios: Its response profile does not change much between setups, and the heuristic manages to create predictable overshoot in all test cases. The *Classic* heuristic seems to be a viable alternative in setups where *PD-Control* performs poorly. Since it has a fairly large integral term, it will also fare much better in setups with more disturbances. *Pessen Integral Rule* deserves an honourable mention for performing very similarly to the *Classic* heuristic, although never beating it. It might be a worthwhile alternative for some specific setups.

P-Control and *PI-Control* are among the most inconsistent performers. Although their results in setups 1 and 3 are absolutely hopeless, they manage to achieve highly impressive tracking performance in setups 2 and 4. A pattern behind these results does not become apparent from this small sample size of tests. Both *P-Control* and *PI-Control* are consistent in their large overshoot and long settling times, no matter the setup. They are therefore mostly useful for reactive workloads, where response time is more important than settling time. The response times of *P-Control* and *PI-Control* are consistently among the best, often in the sub-second range.

The *No Overshoot* and *Some Overshoot* heuristics were among the worst performers in all setups, but were very consistent in their bad performance. Their overshoot seems to remain within a few degrees in all test runs, while their contenders often show large variations. One of the more puzzling results is the amount of overshoot that both metrics generate: Although they are designed to minimize or even eliminate the overshoot caused by the *Classic* tuning, they regularly perform worse in that regard. Equally puzzling is the fact that the *No Overshoot* heuristic generally generates the same amount of overshoot as the *Some Overshoot* heuristic. It seems that both metrics were designed for fundamentally different setups; They don't seem to be particularly useful for remote operation of a humanoid robot.

4.2 Summary

A tuning framework was developed and integrated into the existing remote operation framework for humanoid robots, developed at TUM. The setup allows operators to control a remote robot by making it replicate their pose in real time. The robot is not physical, but exists within a robot simulation. The tuning framework consists of a collection of C# classes, which provide functionality for both manual and automatic PID tuning, as well as data collection. Following a highly modular design, it allows future researches to re-use and modify parts of the framework with ease, according to their own needs. The tuning framework is accessible through an extensive public API, which is documented in both this thesis and in code. Some of the most commonly used functionality of the public API is accessible through a custom-built UI, which is directly integrated into the remote operation framework.

The process of iterative manual PID tuning is facilitated by both the UI and through APIs. Putting major focus on immediate visual feedback of PID performance, the framework provides a real-time joint angle deviation graph, which can show the performance of any PID controller within the robot. This graph aids users in judging the response characteristics of the current tuning, and helps them decide on the next iteration. PID tunings can be transmitted to the robot during operation, making it possible to implement changes quickly and effortlessly. Once a satisfactory PID tuning is achieved, the user can export it in from of a JSON file. This file can be shared with other researchers, or used by external applications. Tuning files can also be imported back into the tuning framework.

A big issue of the remote operation framework is that it is hard for a human operator to repeat their movement accurately, which makes it difficult to compare PID tunings in a meaningful way. The tuning framework provides facilities to play back pre-recorded motion animations to the robot. These test runs are recorded, and data about the performance of the PID controllers is collected in real-time. The data collection is not limited to animation playback, but can also be used during normal operation. After a test run, the tuning framework automatically creates performance evaluations for each joint within the robot. These evaluations include numerous performance metrics, including the average absolute angle deviation and maximum overshoot of the respective controller. All test results can be exported to the user's hard drive as JSON files.

The tuning framework provides the ability to automatically tune all PID controllers within the robot. This process requires no knowledge about PID tuning from the user, and runs completely autonomously. It is possible to tune single joints in isolation, which is useful for subsequent performance analysis. The procedure starts by subjecting the joint to a relay test, which measures the response characteristics of the PID controller. A tuning is then calculated based on an analysis of the resulting oscillation pattern. Since PID tuning is a subjective process, the user can select between several heuristics for the calculation, which can be switched out without having to repeat the relay test. All available tunings can be exported to a JSON file.

To measure the performance of the automatic tuning procedure, four robot setups were created. The setups differ in terms of the physical properties of the robot's joints and limbs, and are intended to cover a representative selection of realistic scenarios. The automatic tuning service was applied once to every joint in the left arm of the robot. The performance of all automatic tuning heuristics was then measured in two test runs, and finally compared to a baseline manual tuning.

In its conclusion, the thesis identifies the heuristics which lead to the best performance in each setup. It also identifies good all-round performers among the heuristics, as well as good performers for specific use-cases. It is concluded that manual tuning should always be preferred if enough expertise and time are available. It is speculated that the superior performance of manual tuning lies in the ability of the user to see 'the whole picture', instead of tuning each joint independently.

4.3 Future Work and Research

Based on the results of the automatic tuning evaluation, research should be conducted into finding new heuristics. These heuristics should be created with remote operation of a humanoid robot as their primary motivation, as this use-case comes with specific preferences that are not favoured by commonly used heuristics. Although it might be faster to tune a robot according a well-known method, such as the oscillation analysis used in this thesis, it might be worthwhile to investigate more interactive ways of tuning. By including a human operator in the tuning process, tailor-made setups could be created after a short, interactive calibration period. By evaluating operator feedback, iterative improvements could be made to an initial, automatically determined PID tuning.

Another area worth investigating is interaction of the robot operator with the robot's PID configuration. Changing the tuning on the fly would allow an operator to adapt the behaviour of their robot to its environment; An example for this could be a bomb disposal robot, which first has to traverse rough terrain before performing its actual task. Both workloads benefit from different motion characteristics, so switching between PID tunings would provide great benefit. A whole selection of tuning presets could be provided for a robot so that each operator can pick their favourite one.

It might pay off to look at other types of industrial controllers: While PID controllers are ubiquitous and well-researched, promising results have been achieved by using more complex control schemes. Open-loop control could be used to provide the bulk of the motion response, with PID controllers merely eliminating the residual error. Force control, like impedance controllers, could be used to limit the maximum force output of the robot. This is especially important in case the robot has to interact with real humans, or with sensitive appliances which are prone to damage. Force controllers also offer the ability to accurately replicate the force of the operator through the robot.

Interaction between the robot and its environment was barely researched in this thesis. Multiple issues arise when the robot no longer exists within a sterile test setup, but a realistic work environment. Force limits have to be applied to avoid damage to the robot and any potential obstacles. PID tunings might react very differently depending on the current workload of the robot; Carrying a weight, for example, might favour tunings with a higher I-term. Keeping balance is another issue that was not covered in this thesis. Uneven terrain can make it impossible for the robot to both replicate the operators pose and to avoid falling over. Such cases require the development of a safety system.

List of Figures

Bibliography

[Alm+14] L. Almeida, B. Patrão, P. Menezes, and J. Dias. “Be the robot: Human embodiment in tele-operation driving tasks.” In: *The 23rd IEEE International Symposium on Robot and Human Interactive Communication*. Aug. 2014, pp. 477–482. DOI: 10.1109/ROMAN.2014.6926298.

[Bel+08] C. J. Bell, P. Shenoy, R. Chalodhorn, and R. P. N. Rao. “Control of a humanoid robot by a noninvasive brain–computer interface in humans.” In: *Journal of Neural Engineering* 5.2 (May 2008), pp. 214–220. DOI: 10.1088/1741-2560/5/2/012.

[Ber17] J. Berner. “Automatic Controller Tuning using Relay-based Model Identification.” eng. PhD thesis. Lund University, Oct. 2017. ISBN: 978-91-7753-446-4.

[CCH15] E. A. Caspar, A. Cleeremans, and P. Haggard. “The relationship between human agency and embodiment.” In: *Consciousness and Cognition* 33 (2015), pp. 226–236. ISSN: 1053-8100. DOI: https://doi.org/10.1016/j.concog.2015.01.007.

[Cru+92] C. Cruz-Neira, D. J. Sandin, T. A. DeFanti, R. V. Kenyon, and J. C. Hart. “The CAVE: Audio Visual Experience Automatic Virtual Environment.” In: *Commun. ACM* 35.6 (June 1992), pp. 64–72. ISSN: 0001-0782. DOI: 10.1145/129888.129892.

[Fer+12] C. L. Fernando, M. Furukawa, T. Kurogi, S. Kamuro, K. sato, K. Minamizawa, and S. Tachi. “Design of TELESAR V for transferring bodily consciousness in telexistence.” In: *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*. Oct. 2012, pp. 5112–5118. DOI: 10.1109/IROS.2012.6385814.

[FVW17] E. Falotico, L. Vannucci, and S. Weber. *Connecting Artificial Brains to Robots in a Comprehensive Simulation Framework: The Neurorobotics Platform*. https://www.frontiersin.org/articles/10.3389/fnbot.2017.00002/full. Jan. 2017.

[Hor] S. Hornsey. *A Review of Relay Auto-tuning Methods for the Tuning of PID-type Controllers*. https://warwick.ac.uk/fac/cross_fac/iatl/reinvention/archive/volume5issue2/hornsey.

[Kam+09] M. Kammers, F. de Vignemont, L. Verhagen, and H. Dijkerman. "The rubber hand illusion in action." In: *Neuropsychologia* 47.1 (2009), pp. 204–211. ISSN: 0028-3932. DOI: https://doi.org/10.1016/j.neuropsychologia.2008.07.028.

[Kan+14] J. Kang, W. Meng, A. Abraham, and H. Liu. "An adaptive PID neural network for complex nonlinear system control." In: *Neurocomputing* 135 (2014), pp. 79–85. ISSN: 0925-2312. DOI: https://doi.org/10.1016/j.neucom.2013.03.065.

[KBB14] J. Koenemann, F. Burget, and M. Bennewitz. "Real-time imitation of human whole-body motions by humanoids." In: *2014 IEEE International Conference on Robotics and Automation (ICRA)*. May 2014, pp. 2806–2812. DOI: 10.1109/ICRA.2014.6907261.

[KH04] N. Koenig and A. Howard. "Design and use paradigms for Gazebo, an open-source multi-robot simulator." In: *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (IEEE Cat. No.04CH37566)*. Vol. 3. Sept. 2004, 2149–2154 vol.3. DOI: 10.1109/IROS.2004.1389727.

[Mcl18] Mcleanbyron. *File Times - Win32 apps*. https://docs.microsoft.com/en-us/windows/win32/sysinfo/file-times. May 2018.

[NOC07] M. I. H. Nour, J. Ooi, and K. Y. Chan. "Fuzzy logic control vs. conventional PID control of an inverted pendulum robot." In: *2007 International Conference on Intelligent and Advanced Systems*. Nov. 2007, pp. 209–214. DOI: 10.1109/ICIAS.2007.4658376.

[ODw06] A. O'Dwyer. "PI and PID controller tuning rules: an overview and personal perspective." In: *2006 IET Irish Signals and Systems Conference*. June 2006, pp. 161–166.

[Osra] Osrf. *Gazebo Tutorials*. http://gazebosim.org/tutorials.

[Osrb] Osrf. *SDF Specification*. http://sdformat.org/spec.

[Pfe10] R. Pfeifer. *Masseverteilung im Körper*. http://www.arsmartialis.com/index.html?name=http://www.arsmartialis.com/faq/m_anteil.html. Nov. 2010.

[Pol+02] N. S. Pollard, J. K. Hodgins, M. J. Riley, and C. G. Atkeson. "Adapting human motion for the control of a humanoid robot." In: *Proceedings 2002 IEEE International Conference on Robotics and Automation (Cat. No.02CH37292)*. Vol. 2. May 2002, 1390–1397 vol.2. DOI: 10.1109/ROBOT.2002.1014737.

[SBR12] C. Stanton, A. Bogdanovych, and E. Ratanasena. "Teleoperation of a humanoid robot using full-body motion capture, example movements, and machine learning." In: Dec. 2012.

[Tac+03] S. Tachi, K. Komoriya, K. Sawada, T. Nishiyama, T. Itoko, M. Kobayashi, and K. Inoue. "Telexistence cockpit for humanoid robot control." In: *Advanced Robotics* 17.3 (2003), pp. 199–217. DOI: `10.1163/156855303764018468`. eprint: `https://doi.org/10.1163/156855303764018468`.

[Tac+12] S. Tachi, K. Minamizawa, M. Furukawa, and C. Fernando. "Telexistence - From 1980 to 2012." English. In: *IEEE International Conference on Intelligent Robots and Systems*. 2012, pp. 5440–5441. ISBN: 9781467317375. DOI: `10.1109/IROS.2012.6386296`.

[Tad+05] R. Tadakuma, Y. Asahara, H. Kajimoto, N. Kawakami, and S. Tachi. "Development of anthropomorphic multi-D.O.F master-slave arm for mutual telexistence." In: *IEEE Transactions on Visualization and Computer Graphics* 11.6 (Nov. 2005), pp. 626–636. ISSN: 2160-9306. DOI: `10.1109/TVCG.2005.99`.

[Uni] Unity Technologies. *Unity*. `https://unity.com/`.

[Wat+08] K. Watanabe, I. Kawabuchi, N. Kawakami, T. Maeda, and S. Tachi. "TORSO: Development of a Telexistence Visual System Using a 6-d.o.f. Robot Head." In: *Advanced Robotics* 22.10 (2008), pp. 1053–1073. eprint: `https://doi.org/10.1163/156855308X324767`.

[Zel+15] A. Zelenak, C. Peterson, J. Thompson, and M. Pryor. "The Advantages of Velocity Control for Reactive Robot Motion." In: Oct. 2015, V003T43A003. DOI: `10.1115/DSCC2015-9713`.

[ZN93] J. G. Ziegler and N. B. Nichols. "Optimum Settings for Automatic Controllers." In: *Journal of Dynamic Systems, Measurement, and Control* 115.2B (June 1993), pp. 220–222. ISSN: 0022-0434. DOI: `10.1115/1.2899060`. eprint: `https://asmedigitalcollection.asme.org/dynamicsystems/article-pdf/115/2B/220/5546571/220\_1.pdf`.