FACULTY OF ENGINEERING AND ARCHITECTURE

Design and control of a simulated anthropomorphic robotic finger using differentiable programming

Killian Storm
Student number: 01500259

Supervisors: Prof. dr. ir. Francis wyffels, Prof. dr. Joris Nikolaas Leijnse
Counsellor: Rembert Daems

Master's dissertation submitted in order to obtain the academic degree of
Master of Science in Computer Science Engineering

Academic year 2019-2020

GHENT UNIVERSITY

FACULTY OF ENGINEERING AND ARCHITECTURE

Design and control of a simulated anthropomorphic robotic finger using differentiable programming

Killian Storm
Student number: 01500259

Supervisors: Prof. dr. ir. Francis wyffels, Prof. dr. Joris Nikolaas Leijnse
Counsellor: Rembert Daems

Master's dissertation submitted in order to obtain the academic degree of
Master of Science in Computer Science Engineering

Academic year 2019-2020

GHENT UNIVERSITY

Preamble

This thesis was made during the COVID-19 pandemic of 2020. This resulted in some restrictions on the experiments done with the physical setup of the robotic finger. I did have the chance to install it at home, but in the last few weeks several problems occurred with the hardware. These issues restricted me from generating more trajectories to be compared to the simulator.

Acknowledgements

It all started when I had the chance to do a challenging final project during my high school years. I chose to create a functional robotic arm and hand controlled by a depth camera and flex sensors. This sparked my interest in the anatomy of human hands and the field of robotics. It let me to the decision of starting my own project on the creation of anatomically correct robotic hands, which I have been developing over the last few years. When I saw the work done by Leijnse et al. I was immediately convinced to research this subject as it could help me provide insight in improving my own project.

These five years of university have been a long and exciting journey. I cannot thank my parents enough for letting me explore all my interests and giving me all the love I need. To my friends whom I have shared so many experiences with. I also want to show my sincere appreciation for the continuous guidance and support of Rembert Daems and Prof. dr. ir. Francis wyffels. And lastly, I would like to thank Prof. dr. Joris Leijnse for providing me with his astounding knowledge of hand anatomy.

Killian Storm, June 15, 2020

Admission to Loan

Killian Storm, June 15, 2020

Design and control of a simulated anthropomorphic robotic finger using differentiable programming

Killian Storm

Supervisors:
Prof. dr. ir. Francis wyffels
and Prof. dr. Joris Nikolaas Leijnse

Counsellors:
Rembert Daems

Master's dissertation submitted in order to obtain the academic degree of Master of Science in Computer Science Engineering

Academic year 2019 – 2020

Faculty of Engineering
and Architecture
Ghent University

Abstract

As of today the human hand still has undiscovered secrets about its anatomy and its dynamic behaviour. Although surgeons are well equipped to operate tendon injuries in the human hand, they lack knowledge of its dynamics. Predicting the outcomes of operations on complex tendon injuries is a tough task. A profound knowledge of this would make surgeons superior. Recently, a robotic anthropomorphic physical replica of a human finger has been manufactured at the University of Ghent. It has been researched with several control strategies. The downside of a physical replica is that it cannot be used to simulate the thousands of iterations needed to predict the outcomes of a modification in the tendons of a patient. On the other hand, a simulated version of this physical setup is perfectly suited for this task. This thesis focuses on the design and implementation of the simulator based on Lagrangian mechanics. Its accuracy compared to the physical setup is measured and several reference trajectories are approximated by training the parameters of a neural network. This includes simple trajectories like a grasp, but also loaded control like pressing a piano key. After applying the proposed strategy to improve the simulator's accuracy, the end result is a simulator and control strategy that could be used to predict the outcomes of a tendon modification.

Keywords

robotics, anthropomorphic, simulation, force control, loaded control, CTRNN, CPG, trajectory approximation, differentiable programming, JAX

Design and control of a simulated anthropomorphic robotic finger using differentiable programming

Killian Storm
Prof. dr. ir. Francis wyffels and Prof. dr. Joris Nikolaas Leijnse
Rembert Daems

Abstract As of today the human hand still has undiscovered secrets about its anatomy and its dynamic behaviour. Although surgeons are well equipped to operate tendon injuries in the human hand, they lack knowledge of its dynamics. Predicting the outcomes of operations on complex tendon injuries is a tough task. A profound knowledge of this would make surgeons superior. Recently, a robotic anthropomorphic physical replica of a human finger has been manufactured at the University of Ghent. It has been researched with several control strategies. The downside of a physical replica is that it cannot be used to simulate the thousands of iterations needed to predict the outcomes of a modification in the tendons of a patient. On the other hand, a simulated version of this physical setup is perfectly suited for this task. This thesis focuses on the design and implementation of the simulator based on Lagrangian mechanics. Its accuracy compared to the physical setup is measured and several reference trajectories are approximated by training the parameters of a neural network. This includes simple trajectories like a grasp, but also loaded control like pressing a piano key. After applying the proposed strategy to improve the simulator's accuracy, the end result is a simulator and control strategy that could be used to predict the outcomes of a tendon modification.

Index Terms robotics, anthropomorphic, simulation, force control, loaded control, CTRNN, CPG, trajectory approximation, differentiable programming, JAX

Introduction

A common injury is focal dystonia in the hands of musicians [7] which is a neurological motor disorder characterised by involuntary contractions of those muscles involved in the play of a musical instrument. It is seen as task-specific and only impairs the voluntary control of highly practiced musical motor skills. That means the hands are still functional, but not for those specific movements. Sometimes this injury can be treated by making adjustments to the tendon configuration of the patient. The problem is that predicting the outcome of this modification is a tough task. Having a robotic finger that allows modifications of its tendons and can approximate certain given reference trajectories would be a valuable asset in this research. For that reason, a physical replica of the human finger has been developed at the University of Ghent. It has been studied by using several control strategies: a mixed force

and position control with a proportional-integral-derivative (PID) controller, which showed promising results in its accuracy, but lacked the support for fast movements and there were no universal PID parameters for each trajectory. The other control strategy was pure force control with only two out of the four muscles being actuated. The problem with force control is that it requires training to approximate a certain trajectory, as it is less forgiving as position control because small force changes can result in a different trajectory. The forces were generated by a Hopf-based central pattern generator (CPG) [13] [15] and its parameters were trained with a CMA-ES algorithm [6]. A CPG [3] is a biologically inspired neural network capable of generating rhythmic output without any input. The Hopf-based CPG lacked a lot of flexibility and showed no support for sudden force changes. This algorithm also required a lot of iterations before converging, which resulted in significant wear of the physical setup. For these reasons, this thesis focuses on the design of a simulated finger controlled by a pure force control strategy with a more advanced CPG and optimisation algorithm based on differentiable programming. The corresponding animations and source code can be found on the GitHub repository: `https://github.com/killianstorm/simulated-anthropomorphic-finger`.

Anatomical foundation

To design a simulated version of the physical setup, knowledge of human finger anatomy is essential. The human hand consists of three regions: the carpals (wrist), the metacarpals (palm) and the phalanges (fingers). The fingers (except for the thumb) consist of respectively from the beginning of the palm to the fingertip of the proximal phalanx, the middle phalanx and the distal phalanx. Each of these phalanges is interconnected by joints, which are displayed in Fig. B. The metacarpophalangeal joint (MCP) connects the palm to the proximal phalanx. The proximal interphalangeal joint (PIP) connects the proximal phalanx to the middle phalanx, and the distal interphalangeal joint (DIP) connects the middle phalanx to the distal phalanx. Fig. B also shows the tracks of each tendon connected to a muscle. There are four muscles necessary for full 2D finger control: the extensor digitorum (ED), the flexor digitorum profundus (FP), the flexor digitorum superficialis (FS) and the interosseus (IO). The lumbrical (LU) has the same end tendon track as the IO and is seen as redundant in primary finger control [9]. The FS is seen as redundant in unloaded finger control, which means the ED, FP and the IO suffice to do all unloaded trajectories [10]. The ED consists of an M-string, which connects to the middle phalanx and several T-strings which connect to the distal phalanx. This subdivision results in a changing moment arm on the DIP joint in function of the angle of the PIP joint which is illustrated in Fig. C. The T-strings slide across the lateral side of the PIP joint causing only one T-string to be taut depending on the angle of the PIP joint. This principle is called the interphalangeal joint (IPJ) coupling and can be described by $d\theta_3 = \frac{r_{\mathrm{M_{PIP}}} - r_{\mathrm{T_{PIP}}}}{r_{\mathrm{T_{DIP}}}} * d\theta_2$ [11], where $r_{\mathrm{M_{PIP}}}$, $r_{\mathrm{T_{PIP}}}$ and $r_{\mathrm{T_{DIP}}}$ are respectively the moment arms of the M-string at the PIP and T-strings at the PIP and DIP.

Physical setup

Leijnse et al. [8] have designed a physical copy of a human finger with 3 DoF on a scale of 2:1. Fig. A shows a closeup of the finger. The robotic finger contains the ED, FP, FS and the IO and also the IPJ-coupling principle. The FS is not actuated as only unloaded control was studied by Leijnse et al. Each moment arm in the human finger is realised by using pulleys. Three infrared reflectors are placed on each phalanx to be measured by an OptiTrack capture system [12] to record a full trajectory. The tendons are actuated by Dynamixel MX106 servomotors [5], which are capable of position and force control. The force on each tendons is measured by a load cell which is a metal ring with several strain gauges placed inside.

Dynamic model

In order to calculate the necessary forces to reproduce a certain trajectory when using a pure force control strategy, a model describing the forces and energies of each body is necessary. This is exactly what a dynamic model is, it takes a set of forces on a certain time instant as input and outputs a trajectory traveled by each body. There are several ways of constructing a dynamic model such as classical Newtonian mechanics and Lagrangian mechanics. The latter is chosen for its simplicity. The equations of motion of each body can be found by solving what is called the Euler-Lagrange equations: $\frac{d}{dt}(\frac{\partial L}{\partial \dot{q_i}}) - \frac{\partial L}{\partial q_i} = F_i$, where q_i are the generalised coordinates of body i, and F_i the acting forces on body i. L is called the Lagrangian and is defined by $L = T - V$, where T and V are respectively the kinetic energy and the potential energy of the system. Fig. D shows the representation of the finger with its absolute angles (θ_i), relative angles (α_i), torques (τ_i) and coordinate system. The first step is to define the dynamic variables as the absolute angles and their velocities. The static variables are e.g. the lengths and masses. The torques and relative angles can be calculated from the dynamic variables. These variables can then be used to calculate the potential and kinetic energy to construct the Lagrangian. The kinetic energy of the system is given as $T = \sum_{i=1}^{3}(\frac{m_i(\dot{x_{ci}}^2+\dot{z_{ci}}^2)}{2} + \frac{\dot{\theta_i}^2 I_i}{2})$, and the potential energy as $V = \sum_{i=1}^{3} m_i g z_{ci}$, where i ranges from 1 to 3 to respectively map on the proximal, middle and distal phalanx, m_i the mass, (x_{ci}, z_{ci}) the coordinates of the center of gravity, I_i the inertia and g the gravitational constant. The only remaining things to construct the Euler-Lagrange equations are the acting forces F_i. These are defined by the torques generated by the force on each tendon, the friction and the ligaments. Fig. E shows the locations of each tendons track in the physical setup. The torque generated by the tendons on a certain joint can be seen as the sum of the force of the tendons passing through the joint multiplied by the moment arm of the joint for each specific tendon. The torque on the MCP joint becomes: $\tau_{\text{MCP}} = r_{\text{FS}_{\text{MCP}}} F_{\text{FS}} + r_{\text{IO}_{\text{MCP}}} F_{\text{IO}} + r_{\text{FP}_{\text{MCP}}} F_{\text{FP}} - r_{\text{ED}_{\text{MCP}}} F_{\text{ED}}$. The torque on the PIP joint: $\tau_{\text{PIP}} = r_{\text{FS}_{\text{PIP}}} F_{\text{FS}} - r_{\text{IO}_{\text{PIP}}} F_{\text{IO}} + r_{\text{FP}_{\text{PIP}}} F_{\text{FP}} - r_{\text{ED}_{\text{PIP}}} F_{\text{ED}}$ and the torque on the DIP joint: $\tau_{\text{DIP}} = r_{\text{FP}_{\text{DIP}}} F_{\text{FP}} - \tau_{\text{ED}_{\text{DIP}}}$. The torque generated by the ED on the DIP joint (the T-strings) is not as straightforward, since it has a changing moment arm depending on the relative angle of the PIP joint. It can be simplified by normalising the PIP angle range to a range of 0 and 1. This normalised range is then multiplied by the torque generated by the ED on the DIP joint. If e.g. the PIP angle is at its minimum boundary, the torque on the DIP joint generated by the ED will be 0. If it is halfway it will be half of the torque and if it is at its maximum limit, the full torque will be applied: $\tau_{\text{ED}_{\text{DIP}}} = F_{\text{ED}} \frac{\alpha_{\text{PIP}} - \alpha_{\text{PIP}_{\text{min}}}}{\alpha_{\text{PIP}_{\text{max}}} - \alpha_{\text{PIP}_{\text{min}}}} l_{\text{dp}}$, where l_{dp} is the distance on which the T-strings are attached on the proximal phalanx. The joint friction is modelled as $\tau_{fr} = -c_{fr}\dot{\alpha_i}$, where $\dot{\alpha_i}$ is the velocity of the relative angle of joint i, and c_{fr} the friction coefficient. The higher it is, the higher the friction and vice-versa. The ligaments stop the finger from over bending, these can be defined as a damper that activates once an angle exceeds its boundaries. However, during the computational experiments it proved to be very inefficient and they are instead modelled as counter-torques that activate once a joint exceeds a certain boundary. The Euler-Lagrange equations can now be constructed by passing in the torques generated by the force on each tendon, the friction and the ligaments. The result is a set of ordinary set of equations (ODEs) of the form $F = MA$, where F, M and A are respectively the force, mass and acceleration matrix. The accelerations which describe the trajectory are unknown so the equations to be solved becomes: $A = M^{-1}F$. These equations can now be solved using an ODE solver.

Simulator accuracy

The accuracy of the simulator is compared against the physical setup. This is done by running predefined trajectories on the physical setup and measuring the forces on the tendons. These forces are then passed into the simulator to find the simulated trajectory with the same forces. Both of these trajectories are then visually and quantitatively inspected. The tested trajectories are a full finger grasp

(flexion), a full flexion with an extended PIP joint, a full flexion with an extended MCP joint and a complex trajectory. More trajectories could not be tested due to the consequences of the COVID-19 pandemic during the period of this thesis. Fig. F and Fig. G show the measured forces and the comparisons for the full grasp trajectory. The trajectories are scattered as dots on the figures. If many subsequent dots are close to each other, it means that the velocity of the end-effector is greater than when the subsequent dots are more spread. The simulated trajectory shows serious deviations but the overall movement is globally the same. The deviations can be explained due to factors which haven't been considered in the model like tendon inertia and measurement errors in the capture system. Fig. H shows the results for the extended PIP trajectory. The trajectories differ but show close resemblance. The reason why the end-effector first goes up and then down is because the PIP bends over the dorsal side and reaches its ligament boundary before bending the MCP joint to the palmar side. The PIP joint itself is kept extended only by muscle action and is otherwise unconstrained (except for its ligaments), so this means only a slight error in the friction coefficient or measured forces can show a deviant trajectory like this. This slight deviation can be solved by increasing the friction coefficient on the PIP joint, thus preventing the PIP joint from over extending. Fig. I shows that it now closely resembles the physical trajectory. Fig. J shows the results for the extended MCP trajectory. The trajectories differ greatly in distance, but show close resemblance in the actual shape. The accuracy can also be improved by increasing the friction on the MCP joint (Fig. K). Fig. L shows the results for the complex trajectory. There is little resemblance between the two trajectories. Changing the friction coefficients did not result in a more accurate simulation. This indicates that the model is not yet accurate enough for complex trajectories. The quantitative comparison between the trajectories is illustrated in Table 1.

	Full grasp	Extended PIP	Extended MCP	Complex
Global friction coefficients	0.1903	0.1858	0.1853	0.1864
After fine-tuning friction coefficients	/	0.0934	0.1021	/

Table 1: Overview of the root mean squared error (RMSE) values between the trajectories done on the physical setup and the trajectory produced by passing the measured forces into the simulator. Changing the friction coefficients results in a much lower loss and thus a better resemblance of the reference. There is however no global configuration of friction coefficients to result in a RMSE decrease for all of the trajectories.

Improving the accuracy

Most of the time, the accuracy could be improved by changing the friction coefficients. This indicates that the immeasurable parameters (e.g. the phalanx inertia, joint friction) of the model are not entirely correct. Therefore, a strategy is discussed in order to optimise these parameters. The strategy uses a differentiable programming approach called gradient descent, which is an optimisation algorithm used to minimise a certain loss function. The loss function is defined as the root mean square error (RMSE) of either the coordinates of each phalanx, its angles, velocities or the end-effector trajectory. The choice of this loss function has yet to be decided as this strategy was not applied due to the strict time span, but it is proposed as future work. A number of iterations n is defined, the gradient of this loss function is then taken and each parameter to be optimised is then subtracted by its partial derivative found in the gradient. This can be seen as each parameter taking a step in its deepest descent of the loss function. This goes on until the desired minimum for the loss function is achieved.

Automatic differentiation computer library

The calculation of the gradients is a numerically intensive process. There are several Python packages that provide efficient methods for calculating gradients like Autograd [1]. Until recently it used to

be the fastest implementation available. Since 2019 it was superseded by JAX [2]. JAX combines Accelerated Linear Algebra (XLA) [16] and Autograd. Due to its high efficiency and execution speed, JAX is the preferred method for calculating gradients. It also allows the use of Just-In-Time compilation (JIT) and a Graphics Processing Unit (GPU). The GPU functionality is not used as the optimisation algorithm does not benefit from the parallelisation the GPU offers. The first time a JAX function is executed, it has to be compiled. This can take some time, but the overall achieved speedup for the next iterations compensates for this.

Approximating unloaded trajectories

The most interesting part about the simulator is that it can now be used to approximate a reference trajectory along with a powerful CPG. The previous Hopf-based CPG did not show the desired results. A continuous time recurrent neural network (CTRNN) is chosen instead as it is also capable of producing rhythmic output along a time interval [4]. A CTRNN is given by $\tau_i \dot{y_i} = -y_i + (\sum_{j=1}^{N} w_{ji}\sigma(y_j + \theta_j)) + I_i$, where y is the state of each neuron, τ is a time constant, w is the weight of an incoming connection, σ is the sigmoid activation function, θ is the bias term or firing threshold of the node and I is an external input which is not used since no external input is registered. The CTRNN consists of four neurons mapping its output on the four muscles. Fig. 1 shows the working principle of the optimisation algorithm to train the parameters of the CTRNN so that it generates the necessary forces to approximate a reference trajectory. The optimisation algorithm uses the *minimize* function of the SciPy [14] package which minimises a certain loss function. Two loss functions have been tested, a loss function based on the angle positions and their velocities ($\text{opt}_{\text{angles}}$) and another loss function based on the coordinates of the end-effector ($\text{opt}_{\text{end-eff}}$). Several experiments have been done on different kinds of trajectories to test the CTRNN's performance on generating oscillations and handling sudden force changes. Fig. M and Fig. O show the approximation of a full grasp trajectory for both loss functions. Fig. N and Fig. P show the approximation of a sinusoidal trajectory. Fig. S shows a perfect circle trajectory. Fig. Q and Fig. R show a trajectory with sudden force changes. The CTRNN proved to not be capable of generating the necessary oscillations to reproduce the sine and circle trajectory. It did prove to be capable at approximating simple trajectories and trajectories that require sudden force changes. Table A shows the benchmarks for the optimisation algorithm for each reference trajectory and each loss function. If the complexity of the trajectory increases, so does the time needed per iteration. For some combinations of trajectories and loss functions, precision errors started to occur, forcing the optimisation algorithm to stop. Both loss functions have their own optimal type of trajectory, but the angle positions and their velocities loss function ($\text{opt}_{\text{angles}}$) performs best overall.

Approximating loaded trajectories

Research on loaded trajectories is the most beneficial to the prediction of the outcome of tendon modifications. As mentioned before, the FS proved to be redundant in unloaded control, which means it should be actively used when the finger is loaded. A piano key has been modelled to provide an example of the optimisation algorithm's performance on the stroking of a piano key without knowing the necessary forces. The piano key can be modelled as a spring that exerts force when it is pressed. This force is then exerted on the finger which result in an extra torque on each joint. This torque has to be mitigated by the CTRNN in order to fully press the piano key. The piano key exerts a force of 10N when it is fully pressed. Fig. T shows the algorithm's performance on the reference trajectory of stroking a piano key. As can be seen from the used forces (Fig. U), the FS is actively used when pressing the piano key. When it is released, it is no longer used. This fully supports the theory of Leijnse et al. The CTRNN was already capable of finding the forces to fully press the piano key after 25 iterations. The time needed per iteration is comparable to the unloaded trajectories which means the addition of a load does not cause any performance impacts.

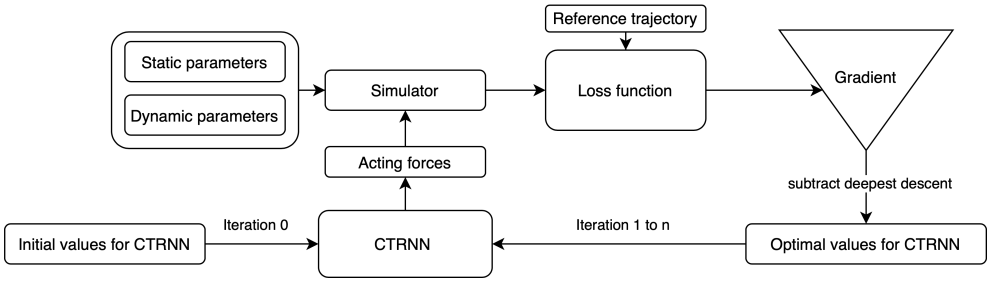


Figure 1: Diagram showing the optimisation algorithm to train the CTRNN parameters. The purpose of training these parameters is to let the CTRNN generate the necessary forces to reproduce a certain reference trajectory. The initial values for the CTRNN parameters are chosen and are passed to the CTRNN, which produces a matrix of approximated forces. These force are passed to the simulator along with the static and dynamic variables defining the model. The simulator outputs an approximated trajectory which is passed into the loss function along with the reference trajectory to return a number indicating the similarity between both trajectories. The gradient is taken, and each CTRNN parameter is updated by subtracting a fraction of its corresponding derivative. These updated parameters are then fed into the CTRNN to begin the whole process again. This is repeated for n iterations or until a desired loss is achieved.

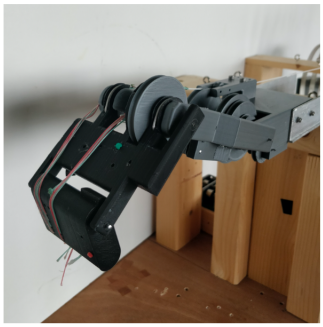
Conclusion

The previous research done on the physical setup raised the question on how the pure force control strategy would perform with a more advanced CPG. A continuous time recurrent neural network CTRNN was chosen instead of the Hopf-based CPG. Its parameters have been trained using gradient descent to generate the necessary forces to reproduce a certain reference trajectory. But in order to do this, a dynamic model describing the energies and forces of the system had to be designed. It allows the calculation of the finger trajectory by passing in the forces on each tendon. Its accuracy compared to the physical setup has also been discussed, it showed some deviations due to some simplifications and immeasurable parameters like the joint frictions. A strategy to improve its accuracy has also been proposed but not executed. It used gradient descent to optimise the immeasurable parameters. To test the CTRNN and the optimisation algorithm on their capability of approximating a certain trajectory, several reference trajectories have been tested. The CTRNN proved to not be capable of handling oscillations due to precision errors in the optimisation algorithm, but it did handle sudden force changes very well. An experiment on loaded control has also been done. A piano key has been modelled and after only 25 iterations, the CTRNN was already capable of generating the forces necessary to fully press the piano key. More research is necessary on the CTRNN to allow it to generate oscillations.

Future work

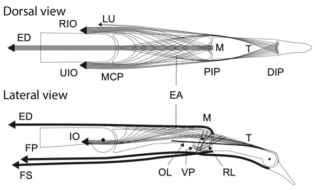
This thesis was mostly an exploration of the capabilities of a simulated anthropomorphic finger and a CTRNN. The simulator is not perfect, therefore more research has to be done in order to increase the accuracy of the simulator by applying the proposed optimisation strategy. There is also still much room for improvement for the CTRNN as it proved to be incapable of performing complex oscillations. A neural network with multiple layers could be the answer. Once this is done, the dynamic model can be used to verify if a patient suffering from musician's focal dystonia can fully press a piano key by modelling the individual tendon couplings of the patient for a single finger.The dynamic model should also be extended to incorporate multiple fingers and the full movement of an MCP joint, as only dorsal and palmar bending is considered in the current model.

Figure A



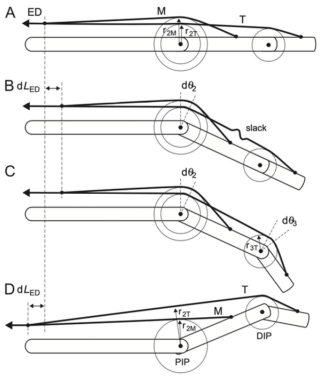
Close-up view of the robotic finger

Figure B



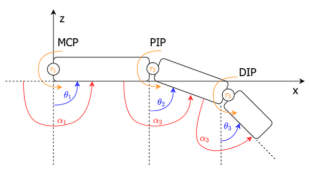
Schematic of the anatomy of the human finger with tendons, joints and ligaments [9]. The LU has the same end tendon track as the RIO and is seen as redundant in primary finger control [8] [7]. The ED consists of two parts: an M-string which connects to the middle phalanx and several T-strings which connect to the distal phalanx.

Figure C



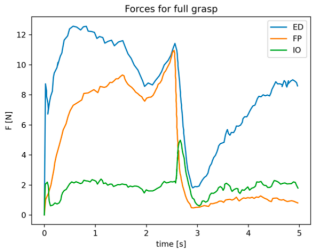
IPJ coupling visualisation [9]. If $r_{2T} < r_{2M}$ and $dL_{ED} > 0$, the PIP flexes (A to B), causing the M-string to take up a displacement m (Eq. 2.1). However, the displacements of the M-string and the T-strings should be equal, as they are driven by the same muscle (the ED). Which results in a flex of the DIP joint (C) with respect to its moment arm and the displacement (as given in Eq. 2.2). Here the DIP and PIP joints rotate in the same sense. If $r_{2T} > r_{2M}$ and $dL_{ED} < 0$, the PIP hyperextends (D), causing dorsal bowstringing. Slack now accumulates in T, causing the DIP to flex. Here the DIP and PIP joints rotate in the opposite sense.

Figure D



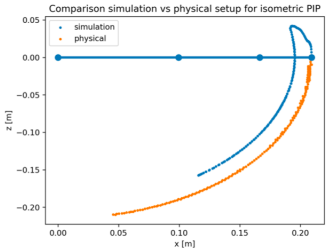
Representation of the finger with its absolute angles (θ_i), relative angles (α_i), torques (τ_i) and coordinate system.

Figure F



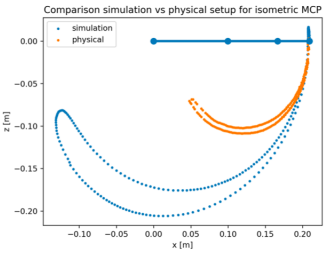
Forces for the grasp trajectory. Only the ED, FP and IO are used for this trajectory as it is not a loaded control. The IO is kept at 2N by the force PID-controller in the physical setup. The ED varies from 8N to 12N and then a sudden drop to 2N after which it goes back up to 8N over the course of two seconds. The FP roughly follows the same path except for a maximum of 8N and once it drops it stays around 1N. The sudden drop in ED and FP is to allow the finger to fully flex. The reason for the ED to be the only active force at $t = 3$ is because the finger has to fully extend again.

Figure H



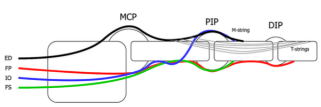
Comparison of the simulation and physical setup for the extended PIP trajectory which is a full flexion of the MCP joint while keeping the PIP joint fully extended. The trajectories differ but show close resemblance. The reason why the end-effector first goes up and the down is because the PIP bends over the dorsal side and reaches its ligament boundary before bending the MCP joint. The velocities of the end-effector for the physical and simulated trajectory are almost equal.

Figure J



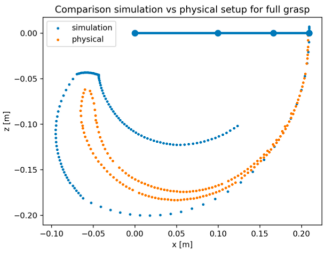
Comparison of the simulation and physical setup for the full flexion but with extended MCP trajectory. The trajectories differ greatly in distance, but show close resemblance in the actual shape. The force generated by FP is too high for the simulated MCP joint since it immediately starts bending to the palmar side. The velocity of the simulated end-effector trajectory is similar to the physical setup.

Figure E



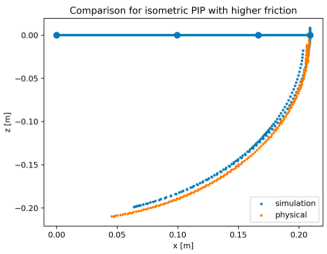
Finger representation with the tendon tracks. For the MCP joint, the FS, IO and FP pass through the palmar and the ED through the dorsal. For the PIP joint, the FS and FP pass through the palmar and the IO and ED (the M-string) through the dorsal. For the DIP joint, the FP passes through the palmar and the ED (the T-strings) through the dorsal. This figure is not on scale.

Figure G



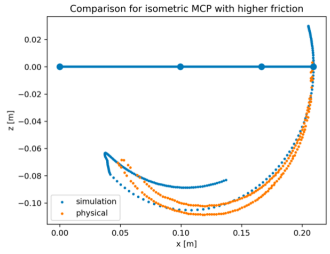
Comparison of the simulation and physical setup for the full grasp trajectory. The finger is represented by three blue dots and lines. The most left being the MCP joint, the middle the PIP joint and the most right the PIP joint. The blue scatter represents the simulated trajectory. The orange scatter the physical trajectory.
The reference trajectory has the same overall end-effector velocity. While flexing, the velocity of the simulation trajectory is too high, resulting in a fast swing and thus preventing the PIP from fully bending. Once the force on the ED and FP drops, the finger remains stationary at a semi-flexed state. Once force on the ED rises again, the finger starts flexing. The PIP joint is however not extending enough, resulting in an extension where the PIP stays fully bent. This is because the IO does not go back to 2N but instead remains around 1N.

Figure I



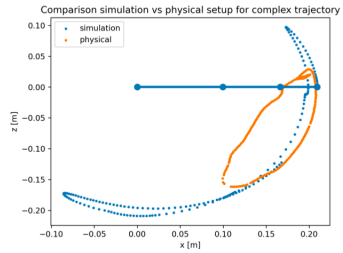
Comparison of the simulation and physical setup for the extended PIP trajectory with a higher friction on the PIP joint. The problem with the overextending of the PIP joint can be solved by using a higher friction coefficient on the PIP joint. It shows an almost equal reproduction of the trajectory of the physical setup.

Figure K



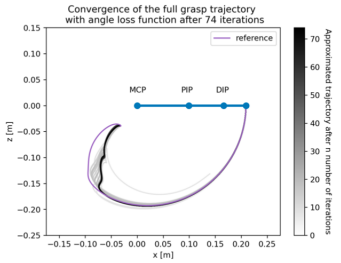
Comparison of the simulation and physical setup for the full flexion but with extended MCP trajectory with a higher friction on the MCP joint. The problem of the MCP bending to the palmar side can be prevented by increasing the friction on the MCP joint. The flexion shows a much closer resemblance to the physical setup, the extension however is much slower thus preventing the finger from going back to its original full extended state. It shows that the problem does not lie in the friction but in other not considered factors.

Figure L



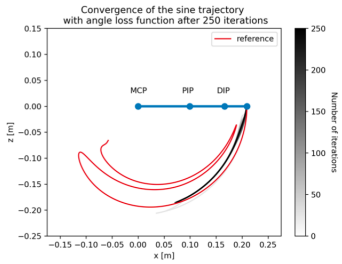
Comparison of the simulation and physical setup for the complex trajectory. The trajectories do not resemble at all. Fine-tuning the friction did not result in any improvements on the accuracy of the simulator. It indicates that complex trajectories are not accurate and should thus be avoided. The accuracy can be improved by taking the factors mentioned in section 5.2 into consideration.

Figure M



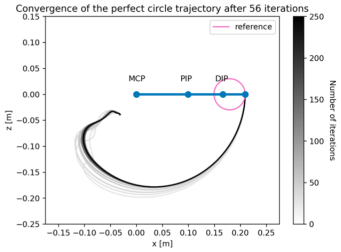
Approximated full grasp trajectory convergence for the end-effector loss function. The approximated trajectory never fully resembles the reference. The optimisation stopped after 74 iterations due to precision errors. This figure shows a gradient bar on the right from white to black. Each end-effector trajectory for a certain iteration has a specific color in this gradient, e.g. the first iteration is almost white and the last iteration is almost black. It illustrates the way the optimisation approaches the reference.

Figure N



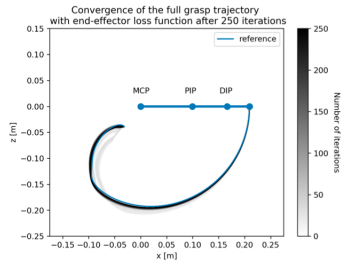
Approximated sine trajectory convergence for the end-effector loss function. The approximated trajectory never fully resembles the reference. The optimisation stopped after 114 iterations due to precision errors. The finger flexes with an extended PIP and DIP and suddenly stops. There is no oscillatory behaviour. This figure shows a gradient bar on the right from white to black. Each end-effector trajectory for a certain iteration has a specific color in this gradient, e.g. the first iteration is almost white and the last iteration is almost black. It illustrates the way the optimisation approaches the reference.

Figure S



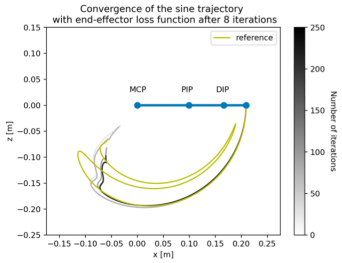
Approximated perfect circle trajectory convergence for the angles/angle velocities loss function. The approximated trajectory never fully covers the reference. The optimisation stops after 56 iterations due to precision errors. The finger does not follow the shape of a circle irregardless of its radius. This figure shows a gradient bar on the right from white to black. Each end-effector trajectory for a certain iteration has a specific color in this gradient, e.g. the first iteration is almost white and the last iteration is almost black. It illustrates the way the optimisation approaches the reference.

Figure O



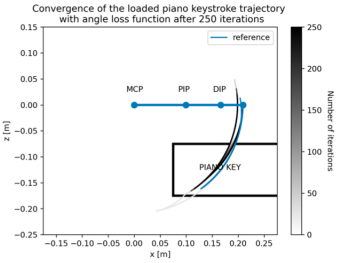
Approximated full grasp trajectory convergence for the end-effector loss function. After 25 iterations, the approximated trajectory almost perfectly resembles the reference.

Figure P



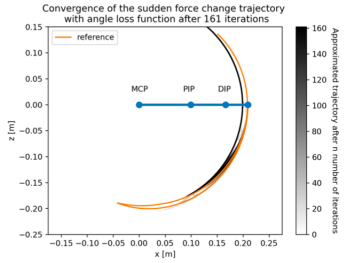
Approximated sine trajectory convergence for the end-effector loss function. The approximated trajectory never fully covers the reference. The optimisation stopped after only 8 iterations due to precision errors. The finger flexes with an extended PIP and DIP and does a full swing, after which a very small oscillatory behaviour occurs.

Figure T



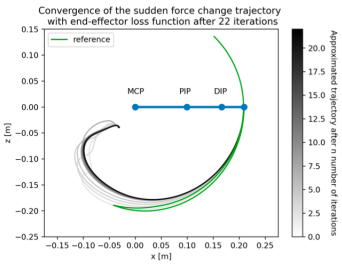
Approximated piano key stroke trajectory convergence for the angles/angle velocities loss function. The approximated trajectory greatly resembles the reference after only 10 iterations as was expected by inspecting the loss convergence. However, it never perfectly resembles the reference as the MCP bends over to the dorsal side when the piano key is pressed. But the overal shape is the same. This figure shows a gradient bar on the right from white to black. Each end-effector trajectory for a certain iteration has a specific color in this gradient, e.g. the first iteration is almost white and the last iteration is almost black. It illustrates the way the optimisation approaches the reference.

Figure Q



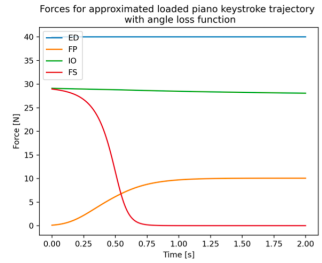
Approximated sudden force change trajectory convergence for the angles/angle velocities loss function. The approximated trajectory greatly resembles the reference after 60 iterations as was expected by inspecting the loss convergence. However, it never fully resembles the reference as the MCP does not bend enough to the dorsal side, but the overal shape is the same. This figure shows a gradient bar on the right from white to black. Each end-effector trajectory for a certain iteration has a specific color in this gradient, e.g. the first iteration is almost white and the last iteration is almost black. It illustrates the way the optimisation approaches the reference.

Figure R



Approximated sudden force change trajectory convergence for the end-effector loss function. The approximated trajectory never even comes close to the reference, instead it does a full grasp. The optimisation stopped after only 22 iterations due to precision errors.

Figure U



Approximated forces for the piano key stroke trajectory convergence for the angle/angle velocities loss function. The most interesting thing is that the FS is actively used when the piano key is pressed. When it is released, it is not used anymore. This experiment fully supports the theory of Leijnse et al. which stated that the FS muscle is necessary in loaded control.

Table A

Trajectory	# iterations without errors		Average time per iteration		Total time	
Loss function	$opt_{end\text{-}eff}$	opt_{angles}	$opt_{end\text{-}eff}$	opt_{angles}	$opt_{end\text{-}eff}$	opt_{angles}
Full grasp	74	no errors	8.21 s	7.22 s	10 min	30 min
Sine	114	7	22 s	38 s	41 m	5 min
Perfect circle	57	/	4 min	/	3 h 47 min	/
Force change	160	22	16 s	35 s	45 min	13 min

Overview of the benchmarks for each unloaded trajectory. If the complexity increases, the average time needed per iteration will also increase.

References

[1] *Autograd*. https://github.com/HIPS/autograd. Accessed: 2020-05-25.

[2] James Bradbury et al. *JAX: composable transformations of Python+NumPy programs*. Version 0.1.55. 2018. URL: http://github.com/google/jax.

[3] D. Bucher. "Central Pattern Generators". In: *Encyclopedia of Neuroscience*. Ed. by Larry R. Squire. Oxford: Academic Press, 2009, pp. 691–700. ISBN: 978-0-08-045046-9. DOI: https://doi.org/10.1016/B978-008045046-9.01944-6. URL: http://www.sciencedirect.com/science/article/pii/B9780080450469019446.

[4] Ángel Campo and José Santos. "Evolution of adaptive center-crossing continuous time recurrent neural networks for biped robot control". In: (Jan. 2010).

[5] *Dynamixel MX-106T*. http://emanual.robotis.com/docs/en/dxl/mx/mx-106/. Accessed: 2020-05-25.

[6] Nikolaus Hansen and Anne Auger. "CMA-ES: Evolution Strategies and Covariance Matrix Adaptation". In: *Proceedings of the 13th Annual Conference Companion on Genetic and Evolutionary Computation*. GECCO '11. Dublin, Ireland: Association for Computing Machinery, 2011, pp. 991–1010. ISBN: 9781450306904. DOI: 10.1145/2001858.2002123. URL: https://doi.org/10.1145/2001858.2002123.

[7] Abbruzzese G. Front Hum Neurosci. Konczak J. "Focal dystonia in musicians: linking motor symptoms to somatosensory dysfunction". In: *Journal of Pharmaceutical Sciences* ().

[8] J. Leijnse et al. "Developing an anthropomorphic robot finger. Part 2: Design and functional evaluation of a prototype with multi- string interphalangeal joints coupling mechanism". In: *Transactions on Mechatronics* (Oct. 2019).

[9] J.N.A.L. Leijnse. "Why the lumbrical muscle should not be bigger - a force model of the lumbrical in the unloaded human finger". In: *Journal of Biomechanics* 30 (1997), pp. 1107–1114.

[10] J.N.A.L. Leijnse and J.J. Kalker. "A two-dimensional kinematic model of the lumbrical in the human finger". In: *Journal of Biomechanics* 28 (1995), pp. 237–249.

[11] Joris Leijnse and C Spoor. "Reverse engineering finger extensor apparatus morphology from measured coupled interphalangeal joint angle trajectories - a generic 2D kinematic model". In: *Journal of biomechanics* 45 (Nov. 2011), pp. 569–78. DOI: 10.1016/j.jbiomech.2011.11.002.

[12] *Motive OptiTrack*. https://optitrack.com/products/motive/. Accessed: 2020-05-25.

[13] Ludovic Righetti and A J Ijspeert. "Pattern generators with sensory feedback for the control of quadruped locomotion". English (US). In: *IEEE International Conference on Robotics and Automation. ICRA 2008*. 2008, pp. 819–824.

[14] Pauli Virtanen et al. "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python". In: *Nature Methods* 17 (2020), pp. 261–272. DOI: https://doi.org/10.1038/s41592-019-0686-2.

[15] Xingming Wu et al. "CPGs with Continuous Adjustment of Phase Difference for Locomotion Control". In: *International Journal of Advanced Robotic Systems* 10.6 (2013), p. 269. DOI: 10.5772/56490. eprint: https://doi.org/10.5772/56490. URL: https://doi.org/10.5772/56490.

[16] *XLA: Optimizing Compiler for Machine Learning*. https://www.tensorflow.org/xla. Accessed: 2020-05-25.

Contents

List of Figures

List of Tables

List of acronyms

CMA-ES Covariance Matrix Adaptation Evolution Strategy.
CTRNN Continuous Time Recurrent Neural Network.

DIP Distal Interphalangeal.
DoF Degree(s) of Freedom.

EA Extensor Apparatus.
ED Extensor Digitorum.

FP Flexor Digitorum Profundus.
FS Flexor Digitorum Superficialis.

IO Interosseus.
IPJ Interphalangeal Joint.

LU Lumbrical.

MCP Metacarpophalangeal.

OL Oblique Retinacular Ligament.

PID Proportional–Integral–Derivative.
PIP Proximal Interphalangeal.

RIO Radial Interosseus.
RL Retinacular Ligament.

UIO Ulnar Interosseus.

Chapter 1

Introduction

As of today the human hand still has undiscovered secrets about its anatomy and its dynamic behaviour. Although surgeons are well equipped to operate tendon injuries in the human hand, they lack knowledge of its dynamics. A well profound knowledge of this would make them superior in predicting the outcomes of operations on complex injuries.

For example, a common injury is focal dystonia in the hands of musicians [13] which is a neurological motor disorder characterised by involuntary contractions of those muscles involved in the play of a musical instrument. It is seen as task-specific and only impairs the voluntary control of highly practiced musical motor skills. That means the hands are still functional, but not for those specific movements.

Sometimes these injuries can be resolved by making adjustments to the coupling of individual tendons. Each individual person has a unique set of interconnections between their tendons. The prediction of which interconnection should be severed to allow the task-specific movements to be done again is very complex. Therefore a robust and accurate replica of the human hand with options to model unique tendon couplings would be seen as a big step forward to help construct these sorts of operations.

Several existing simulators already exist. Avilés Sánchez et al. [2] have designed a simulator of the human hand with 5 degrees of freedom. Each finger only has one degree of freedom, thus only allowing it to flex and extend. The minimum degree of freedoms necessary per finger to reproduce all the movements in the 2D-plane is at least 3. No other simulator incorporates the coupling between the proximal phalanx and the distal phalanx which is a fundamental criterion to make the simulator anatomically correct.

1.1 Project goals

Leijnse et al. [14] [5] [9] have developed a physical set-up which replicates the behaviour of a single human finger. It has been studied using a motorized approach which enabled several complex trajectories to be reproduced and accurately analysed. The study focused on position and force control to move the finger along predefined trajectories.

With position control, the desired trajectories are translated to tendon displacements using a kinematic model to be fed to the servo motors. The positions are regulated by a proportional-integrate-derivative (PID) controller which showed promising results and accurate behaviour.

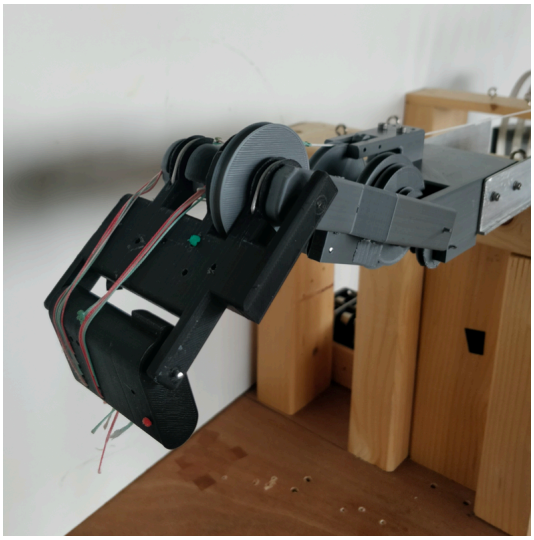


Figure 1.1: Closeup of the anatomically correct replica (scale 2:1) of a human finger. It has 3 degrees of freedom.

With force control, the trajectories are defined by their forces and not the tendon displacements. The forces were generated using a Hopf-based oscillator which is a central pattern generator (CPG) [4]. However, this oscillator does not allow complex changes in the forces thus restricting the possible trajectories. The parameters for the oscillator were determined using a CMA-ES [11] algorithm which takes hundreds of physical tries to approximate the desired trajectory.

1.1.1 Thesis goals

The previous study on force control begs the question how the system would perform with a more advanced oscillator. Recent work on robotic control has showed that recurrent neural networks can behave as CPGs, e.g. to mimic crab-like and biped robot locomoting [7] [23].

However a physical setup is not capable to perform thousands and thousands of iterations in order to let the neural network converge into an acceptable approximation of the necessary forces. An accurate simulator of the physical setup would be a very promising attribute in the training of this neural network as it allows parallelisation, is resistant to wear and can be done virtually anytime and anywhere.

To optimise the parameters of the neural network, a technique called differentiable programming is used. It requires a differentiable loss function which can be minimised by taking steps into the deepest descent of each parameter. This technique is also better known as gradient descent.

Several numerical computer libraries allow the efficient calculation of gradients. Autograd [1] is one of those, until recently it used to be the fastest implementation. Since 2019 a new Python package called JAX [3] has been designed to use a combination of accelerated linear algebra (XLA) [25] and Autograd. JAX is by far the most efficient currently available package for calculating gradients and is

for that reason the method used in this thesis.

In order to efficiently determine which forces should be applied at which time instant, the kinematic model designed by Leijnse et al. [14] is not sufficient as it only returns the necessary tendon displacements to reproduce a trajectory. A more suited approach would be to use a dynamic model which is defined by the energies and active forces on the bodies. It also allows the use of differentiable programming to efficiently reach the optimal forces to reproduce the desired trajectory.

Leijnse et al. only focused on unloaded control. The next step is to focus on loaded control which is much more interesting as the patients suffering from tendon injuries have to be able to do loaded trajectories again.

This thesis focuses on the design and analysis of the simulator, advanced oscillator and optimising algorithms in order to reproduce desired loaded and unloaded simulated finger trajectories.

1.2 Chapter summary

Chapter 2 and Chapter 3 focus on the work done by the Leijnse et al. [14] [5] [9]. Chapter 4 on the construction of the dynamic model. This model is then simulated and analysed in Chapter 5. Chapter 6 introduces differentiable programming and a strategy to use it to optimise the accuracy of the model. Chapter 7 discusses the performance of the optimisation algorithm used to approximate certain unloaded and loaded trajectories with the help of a neural network. Chapter 8 concludes all results of the aforementioned chapters and the appendices contain valuable information such as the source code and instructions for reproducing certain results.

Chapter 2

Setting and background

This chapter is dedicated to explaining the necessary principles of finger anatomy and dynamics in order to understand the requirements for the robotic finger.

2.1 Anatomy of the human finger

The human hand is a complex tool, therefore only the necessary topics will be highlighted. As can be seen in Fig. 2.1, there are three regions:

- *Carpals* in the wrist
- *Metacarpals* in the palm
- *Phalanges* in the fingers

The bones in the finger are respectively from lower to upper called *proximal phalanx, middle phalanx* and *distal phalanx*. Each of these bones are connected by joints and ligaments and are driven by muscles which are shown in Fig. 2.3. The joints are called:

- *Metacarpophalangeal joint (MCP):* between the metacarpals (palm) and the proximal phalanx
- *Proximal interphalangeal joint (PIP):* between the proximal phalanx and the middle phalanx
- *Distal interphalangeal joint (DIP):* between the middle phalanx and the distal phalanx

The phalanges are controlled by six muscles which are connected to tendons:

- *Extensor digitorum (ED)*
- *Flexor digitorum superficialis (FS)*
- *Flexor digitorum profundus (FP)*
- *Radial interosseus (RIO)*
- *Ulnar interosseus (UIO)*
- *Lumbrical (LU)*

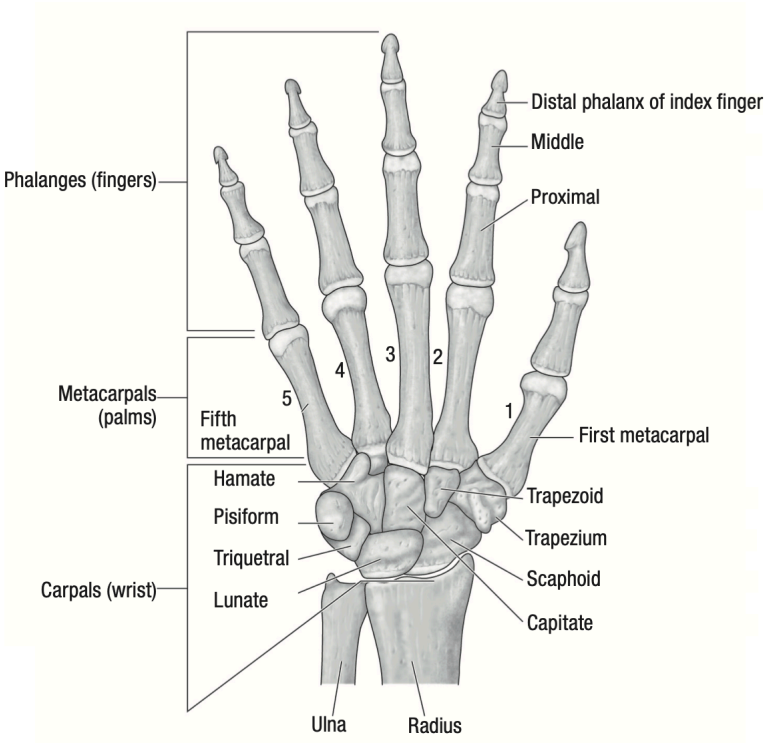


Figure 2.1: Schematic showing the anatomy of the human hand [18]. There are three regions the carpals, metacarpals and the phalanges. The fingers (except for the thumb) consist of a proximal, middle and distal phalanx.

Fig. 2.2 shows an actual dissection of a middle finger with the tendons, joints and ligaments [12]. The ED, FS and FP lie in the forearm, RIO, UIO and LU lie in the metacarpals region. The LU has the same end tendon track as the RIO and is seen as redundant in primary finger control [16] [15], which means it can be omitted in the replica of the finger [9].

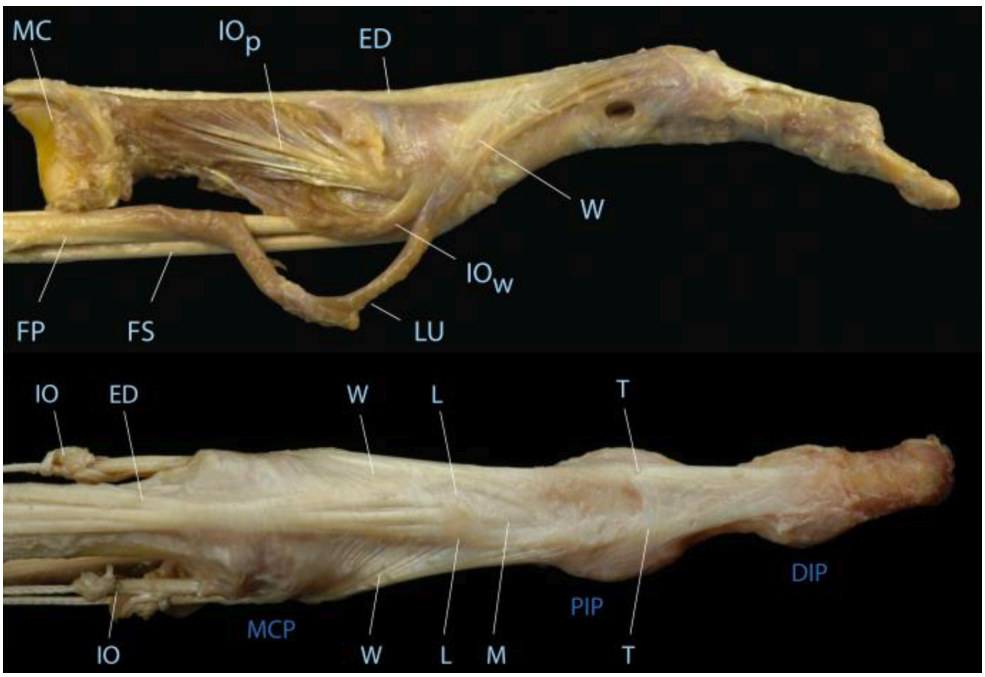


Figure 2.2: Dissection of a middle finger showing the tendons, ligaments and joints [12].

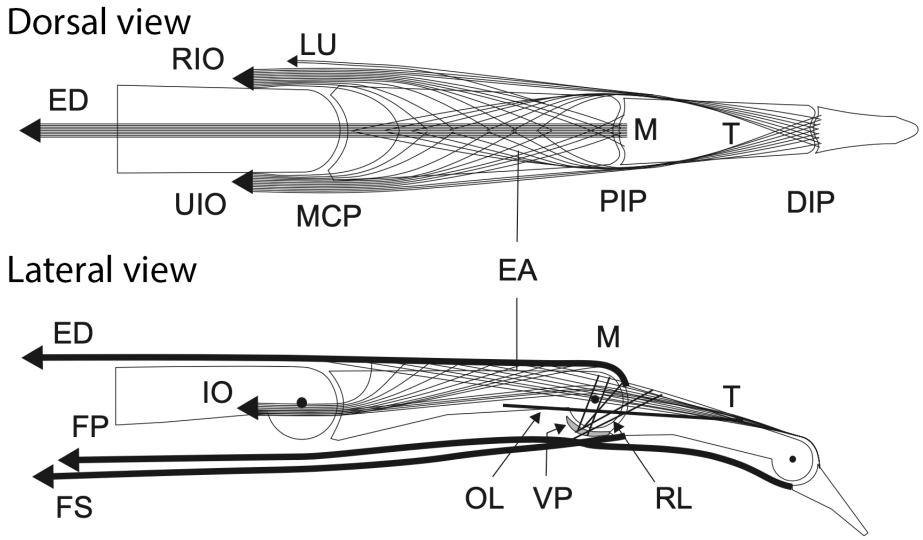


Figure 2.3: Schematic of the anatomy of the human finger with tendons, joints and ligaments [17]. The LU has the same end tendon track as the RIO and is seen as redundant in primary finger control [16] [15]. The ED consists of two parts: an M-string which connects to the middle phalanx and several T-strings which connect to the distal phalanx.

2.1.1 Interphalangeal joints (IPJ) coupling

All muscles have straightforward behaviour on how they move the phalanges, for example the FP will cause the MCP, PIP and DIP to bend according to their moment arms. Except for the ED, which has a certain coupling between the PIP and DIP joint called the interphalangeal joints (IPJ) coupling which is discussed below.

Fig. 2.3 shows that the ED consists of two parts: an M-string which connects to the middle phalanx and several T-strings which connect to the distal phalanx.

The total displacement of the M-string is given as the sum of the displacement of the M-string on the MCP and PIP joint [17]:

$$dL_{\mathrm{ED_M}} = \frac{\partial L_{\mathrm{ED_M}}}{\partial \theta_1} + \frac{\partial L_{\mathrm{ED_M}}}{d\theta_2} = r_{\mathrm{ED_{MCP}}} d\theta_1 + r_{\mathrm{M_{PIP}}} d\theta_2, \tag{2.1}$$

where $r_{\mathrm{ED_{MCP}}}$ and $r_{\mathrm{M_{PIP}}}$ are respectively the moment arms of the ED at the MCP joint and the M-string at the PIP joint.

The total displacement of the T-strings are then given as the sum of the displacements of the T-strings on the MCP, PIP and DIP joint:

$$dL_{\mathrm{ED_T}} = \frac{\partial L_{\mathrm{ED_T}}}{d\theta_1} + \frac{\partial L_{\mathrm{ED_T}}}{d\theta_2} + \frac{\partial L_{\mathrm{ED_T}}}{d\theta_3} = r_{\mathrm{ED_{MCP}}} d\theta_1 + r_{\mathrm{T_{PIP}}} d\theta_2 + r_{\mathrm{T_{DIP}}} d\theta_3, \tag{2.2}$$

where $r_{\mathrm{T_{PIP}}}$ and $r_{\mathrm{T_{DIP}}}$ are respectively the moment arms of the T-strings at the PIP and DIP.

When the ED is taut and remains taut, the displacements of the M-string (Eq. 2.1) and the displacement of the T-strings (Eq. 2.2) should be equal:

$$r_{\mathrm{ED_{MCP}}} d\theta_1 + r_{\mathrm{M_{PIP}}} d\theta_2 = r_{\mathrm{ED_{MCP}}} d\theta_1 + r_{\mathrm{T_{PIP}}} d\theta_2 + r_{\mathrm{T_{DIP}}} d\theta_3, \tag{2.3}$$

which results in what is called the interphalangeal joints (IPJ) coupling [17]:

$$d\theta_3 = \frac{r_{\mathrm{M_{PIP}}} - r_{\mathrm{T_{PIP}}}}{r_{\mathrm{T_{DIP}}}} * d\theta_2. \tag{2.4}$$

Leijnse and Spoor [17] determined that an IPJ coupled trajectory can only be approximated by using **multiple** T-strings, each of which would be taut over a certain part of the PIP motion range.

Fig. 2.4 shows the IPJ coupling principle [17].

If $r_{2\mathrm{T}} < r_{2\mathrm{M}}$ and $dL_{\mathrm{ED}} > 0$, the PIP flexes (A to B), causing the M-string to take up a displacement m (Eq. 2.1). However, the displacements of the M-string and the T-strings should be equal, as they are driven by the same muscle (the ED). Which results in a flex of the DIP joint (C) with respect to its moment arm and the displacement (as given in Eq. 2.2). Here the DIP and PIP joints rotate in the same sense.

If $r_{2\mathrm{T}} > r_{2\mathrm{M}}$ and $dL_{\mathrm{ED}} < 0$, the PIP hyperextends (D), causing dorsal bowstringing. Slack now accumulates in T, causing the DIP to flex. Here the DIP and PIP joints rotate in the opposite sense.

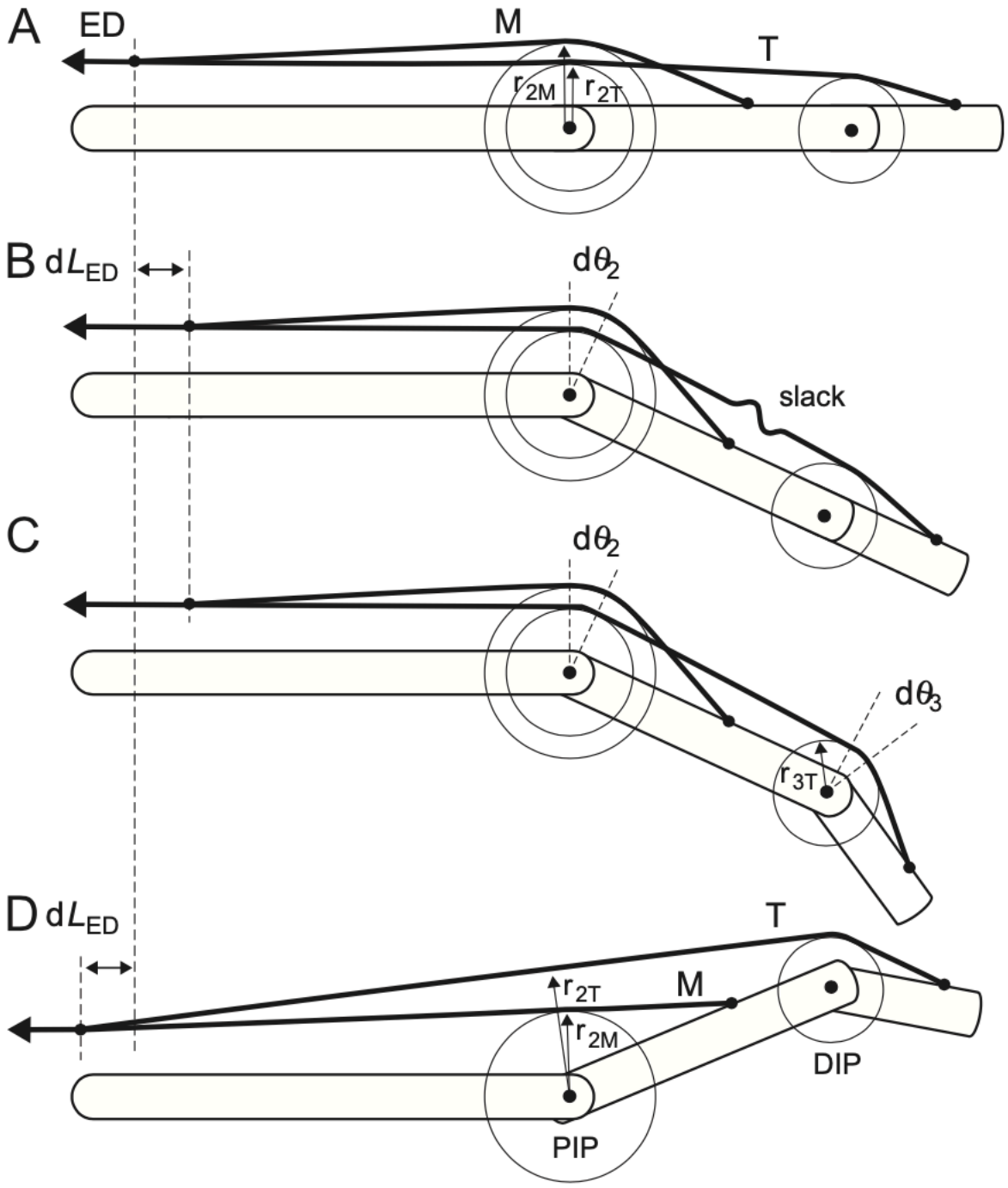


Figure 2.4: IPJ coupling visualisation [17]. If $r_{2T} < r_{2M}$ and $dL_{ED} > 0$, the PIP flexes (A to B), causing the M-string to take up a displacement m (Eq. 2.1). However, the displacements of the M-string and the T-strings should be equal, as they are driven by the same muscle (the ED). Which results in a flex of the DIP joint (C) with respect to its moment arm and the displacement (as given in Eq. 2.2). Here the DIP and PIP joints rotate in the same sense. If $r_{2T} > r_{2M}$ and $dL_{ED} < 0$, the PIP hyperextends (D), causing dorsal bowstringing. Slack now accumulates in T, causing the DIP to flex. Here the DIP and PIP joints rotate in the opposite sense.

2.2 Physical setup

With the foundations of finger anatomy and dynamics explained, the physical setup can be discussed.

The setup is shown in Fig. 2.11, it consists of servo motors, a load cell bridge, a load cell PCB, an OptiTrack capture system (Fig. 2.8) and the robotic finger. Only the ED, FP and IO are connected to a servo since the FS proved to be redundant in unloaded control [9].

2.2.1 Servo motors

The used servo motors are Dynamixel MX-106 (Fig. 2.5), its specifications are listed in Table 2.3 [8]. It is capable of current, velocity, position and voltage control.

Figure 2.5: Dynamixel MX-106 servo [8] used in the physical setup. Three are used for the ED, FP and IO.

2.2.2 Load cells

In order to have tendon force feedback, load cells (Fig. 2.6 and 2.7) are placed in between the tendons. A. Van Erum [9] designed the PCB in order to accurately read the forces applied on the tendons.

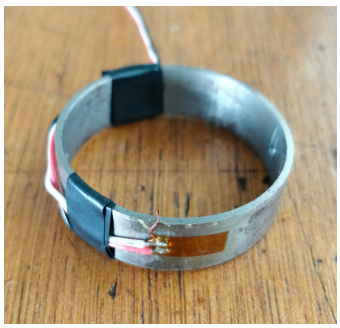


Figure 2.6: The load cells used to measure the force in the tendons. They consist of a metal ring with one strain gauge glued on the outside and one on the inside [9].

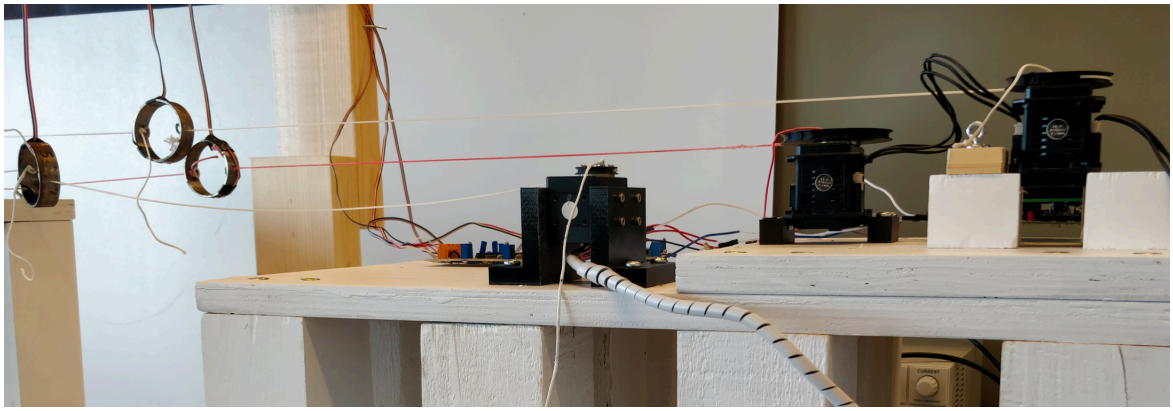


Figure 2.7: The load bridge consisting of three load cells placed in between the tendons to measure the forces. [9].

2.2.3 OptiTrack capture system

An OptiTrack capture system consisting of 6 OptiFlex 13 cameras is used to accurately capture the angles of the phalanges. Fig. 2.8 shows the portable capture system. In order for the system to measure the angles, three infrared reflectors are applied to each phalanx (see Fig. 2.9). The positions are read with the help of Motive software [20]. To capture a trajectory, each phalanx has to be defined as a rigid body by selecting its three infrared reflectors (Fig. 2.10).

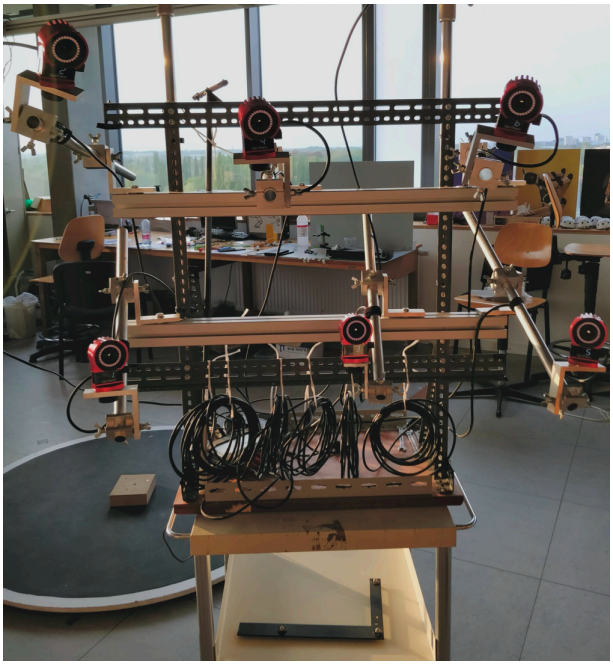


Figure 2.8: Portable OptiTrack capture system capable of accurately measuring the angles of each joint.

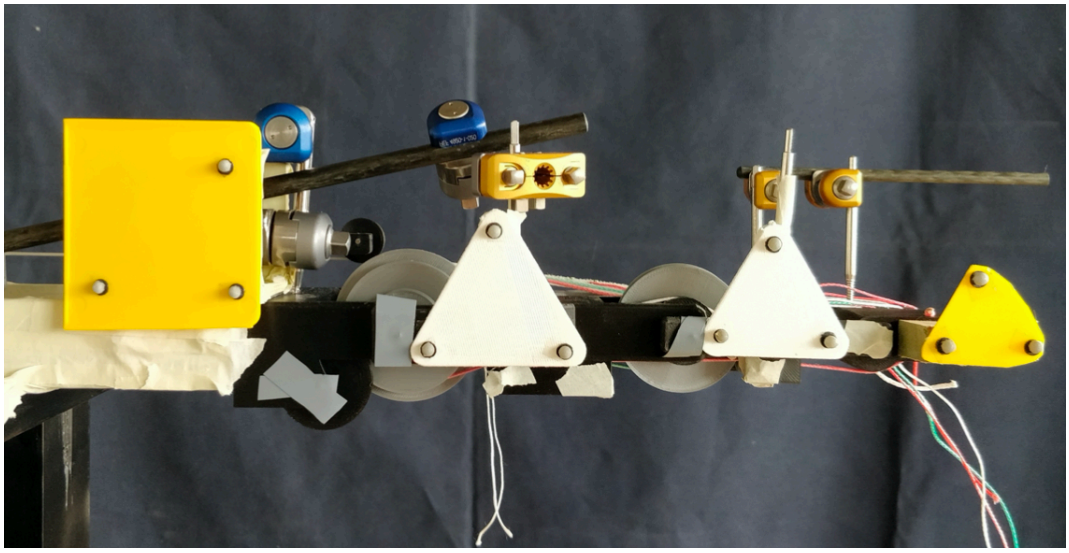


Figure 2.9: Infrared reflectors are placed on each phalanx, allowing the OptiTrack capture system to identify each phalanx.

Figure 2.10: Screenshot of the Motive capturre software. The phalanges have to be defined as rigid bodies by selecting the three infrared reflectors, it then allows to capture a trajectory [5].

Joint/Tendon	ED	FP	IO	FS
DIP	18	22	12	26
PIP	10	21	10	17
MCP		12	/	/

Table 2.1: Overview of the moment arms for each joint and tendon. / means the tendons do not impact that joint. The moment arm for the DIP joint for the ED has been left out since it is not constant and has been determined in Section 2.1. All values are in millimeter (mm).

Joint	Length (mm)
MCP	99.55
PIP	66.87
DIP	42.91

Table 2.2: The lengths of each phalanx in millimeter (mm).

2.2.4 Robotic finger

The robotic finger is 3D-printed on a scale of 2:1 [5]. Just as in the human finger, each joint has different moment arms for each tendon, these are realised by using pulleys. Their dimensions are listed in Table 2.1. The length of each phalanx is listed in Table 2.2.

Fig. 2.12 shows a close-up view. Fig. 2.13 shows the realisation of the IPJ coupling using an M-string and five T-strings. Each T-string spans over a different distance on the distal phalanx, which reproduces the changing moment arm on the DIP joint with respect to the angle of the PIP joint. Fig. 2.14 shows the locations of the tendon tracks.

Item	Value
Baudrate	8 000 [bps] - 4.5 [Mbps]
Weight	165 [g]
Dimensions (W x H x D)	40.2 x 65.1 x 46 [mm]
Gear ratio	225:1
Stall Torque	8.4 [Nm] @ 12 [V], 5.2 [A]
Radial Load	40 [N]

Table 2.3: Specifications of the Dynamixel MX-106 servo motor [8]

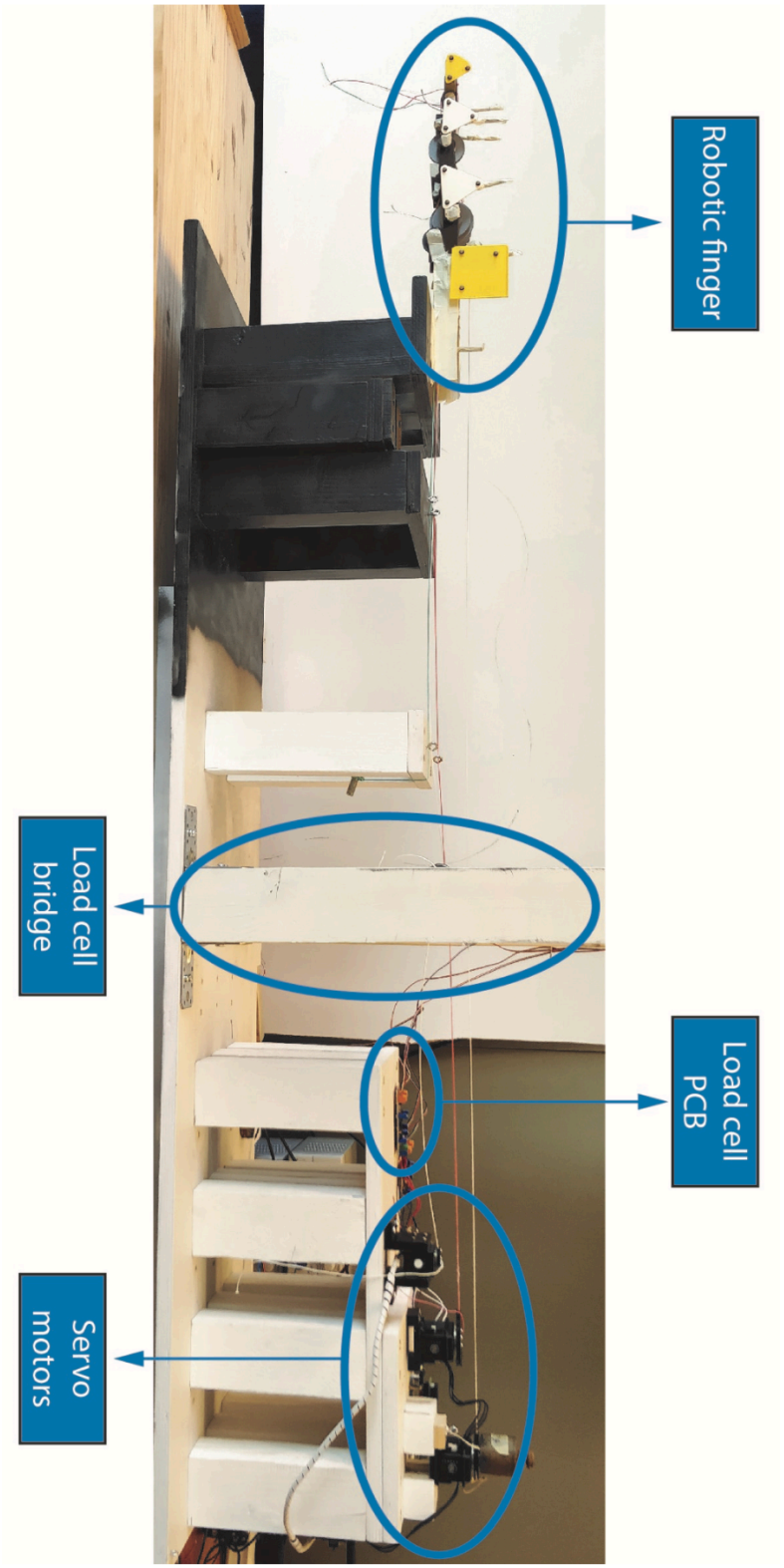


Figure 2.11: Full setup of the robotic finger

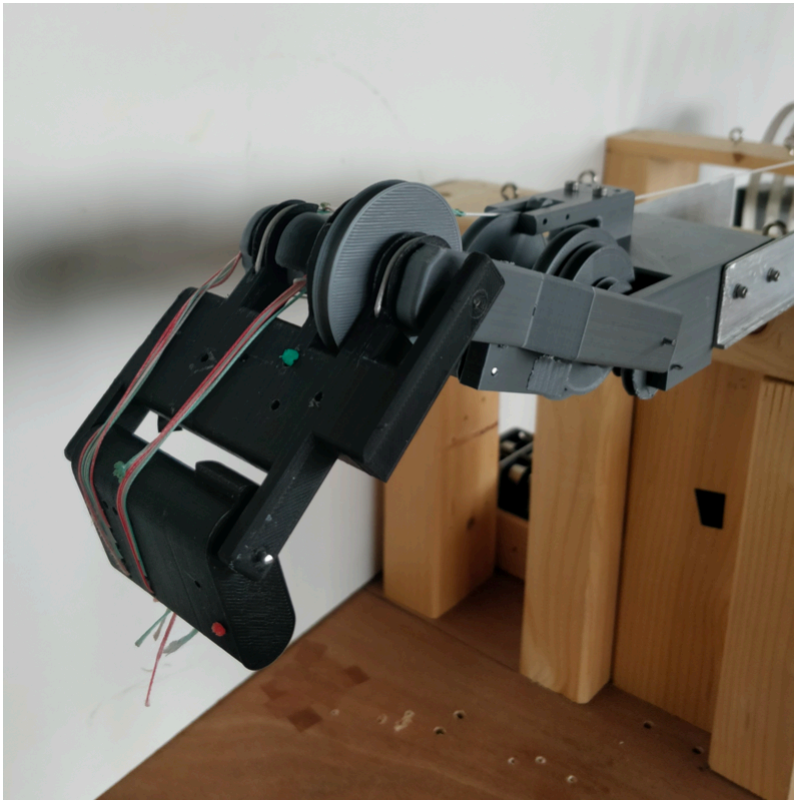


Figure 2.12: Close-up view of the robotic finger

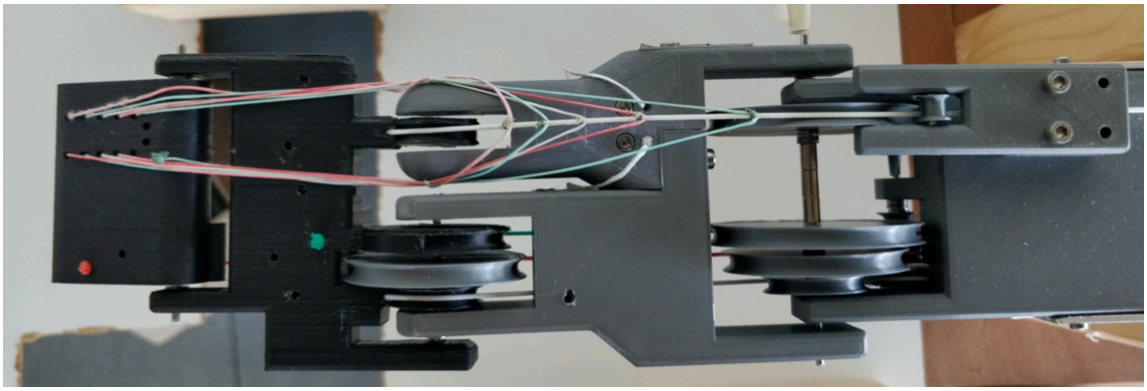


Figure 2.13: Realisation of the IPJ coupling

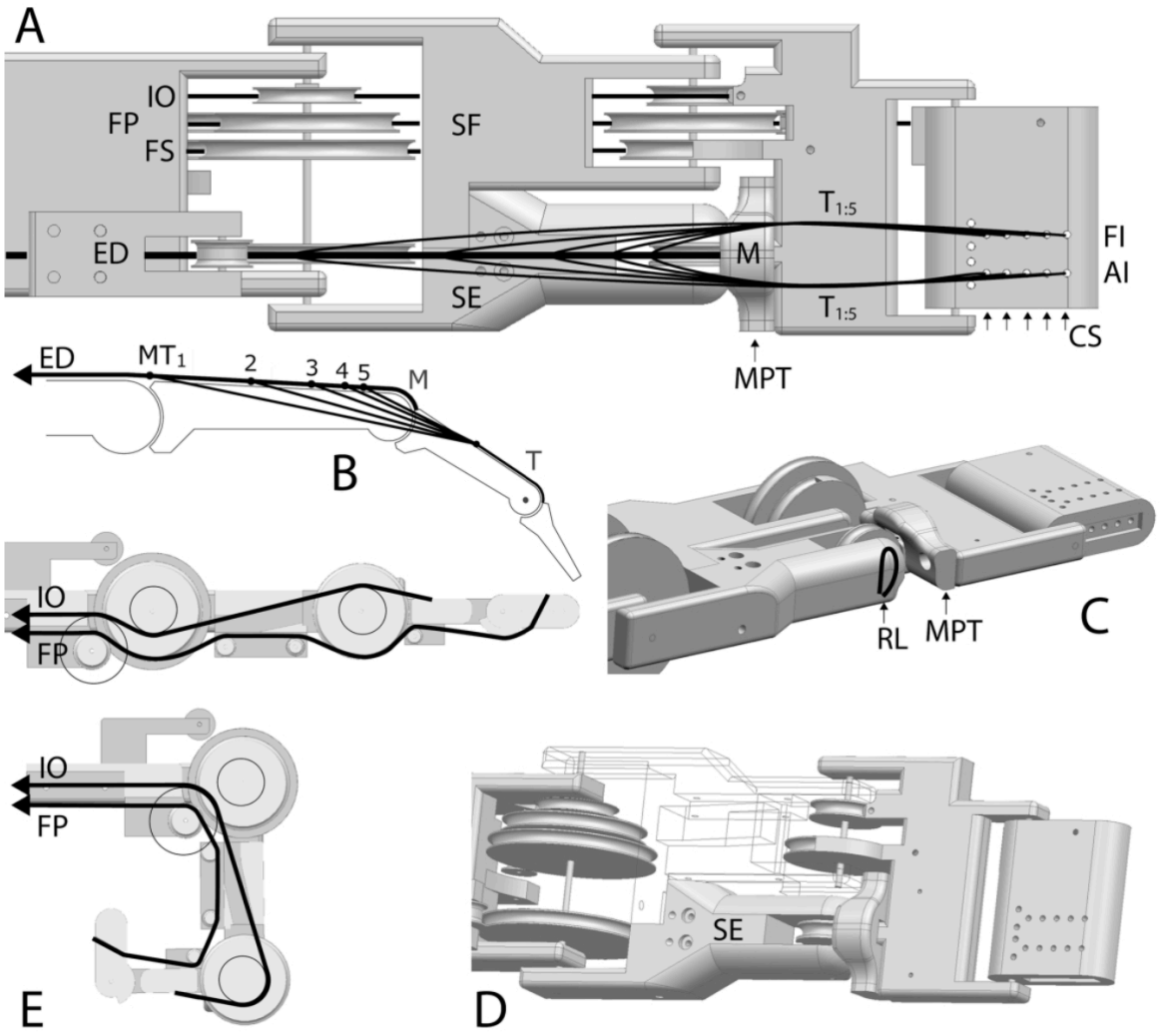


Figure 2.14: Tendon track locations in the physical robotic finger [14]

Chapter 3

Antagonistic control strategies

This chapter covers a summary of the possible antagonistic control strategies and the previous studies done on the physical setup.

3.1 Pure position control

Position control would allow the reproduction of precise trajectories. However, the motor steering cannot be accurate enough in order to compensate for any slack in the tendons while moving the finger. Some trajectories require all tendons to be taut at all times, however, sudden unforeseen forces or conflicts could produce slack which are not always manageable for the servos. The algorithm would try to mitigate this problem and thus possibly inflict destructive forces. For these reasons the pure position control strategy is not researched.

3.2 Mixed N position/M-N force control

The problem with pure position control can be solved by using a mixed N position/M-N force control strategy. Consider the case of one joint with 1 degree of freedom (DoF). This needs two antagonistic muscles to be fully controlled. With N joints, an educated guess would be to use 2N muscles. This is not the case as it can be done with N + 1 muscles.

Fig. 3.1 shows a system with 2 DoFs. There are two muscles on the right side, these control the top joint. The muscle on the left side is needed to retain the lower joint in its original position. This muscle is called the antagonist of the combined forces of the muscles at the right side. The system can be fully controlled with 3 muscles.

This concept translates to the physical setup with 3 DoF, where it can be controlled with 4 servos. With this control strategy, the ED and FP are position controlled and the IO force controlled. The FS is omitted since it is redundant in unloaded control [9].

3.2.1 Trajectory reproduction

In order to reproduce certain trajectories, a kinematic model was constructed to calculate the tendon displacements by providing the reference trajectory. These tendon displacements were fed to a

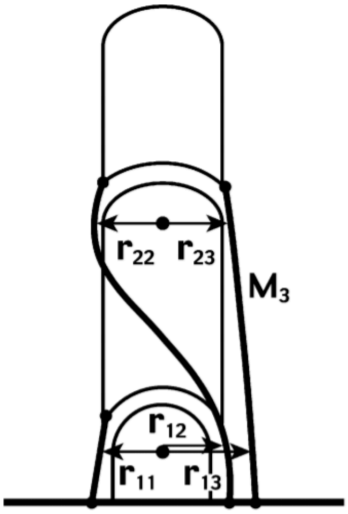


Figure 3.1: Illustration of the Mixed N position/M-N force control principle. It shows a system with 2 DoFs. The two muscles on the right side control the top joint. The left muscle is used to retain the joint in its original position, it is called the antagonist of the combined forces of the muscles at the right side. The system can be fully controlled with 3 muscles.

proportional-integral-derivative (PID) controller which regulated the rotations of the servos to approximate the displacements as close as possible.

Several reference trajectories have been tested. Fig. 3.2 show the comparison between the reference trajectories and the trajectories reproduced by the physical setup. It proved to be highly accurate for most cases. However, it did require the finetuning of different PID parameters to achieve an accurate reproduction. It also proved to not be capable of performing fast movements.

3.3 Force control

Force control is realised by changing the servo torques (thus the servo currents) as a function of time. The force changes over time generate specific trajectories. It requires training to perfect a certain trajectory since a small change in force can produce a radically different trajectory. It is also sensitive to unforeseen external forces.

3.3.1 Central Pattern Generators (CPGs)

Formally seen, Central Pattern Generators (CPGs) [4] are neural circuits in self-contained integrative nervous systems that generate repetitive patterns of motor behavior independent of any sensory input or feedback. In other words, they are biological neural networks that produce rhythmic outputs without any input.

They can be modelled by using mathematical artificial neurons and combining them together to form a neural network. The network is defined by specific parameters which allow it to generate all sorts of trajectories.

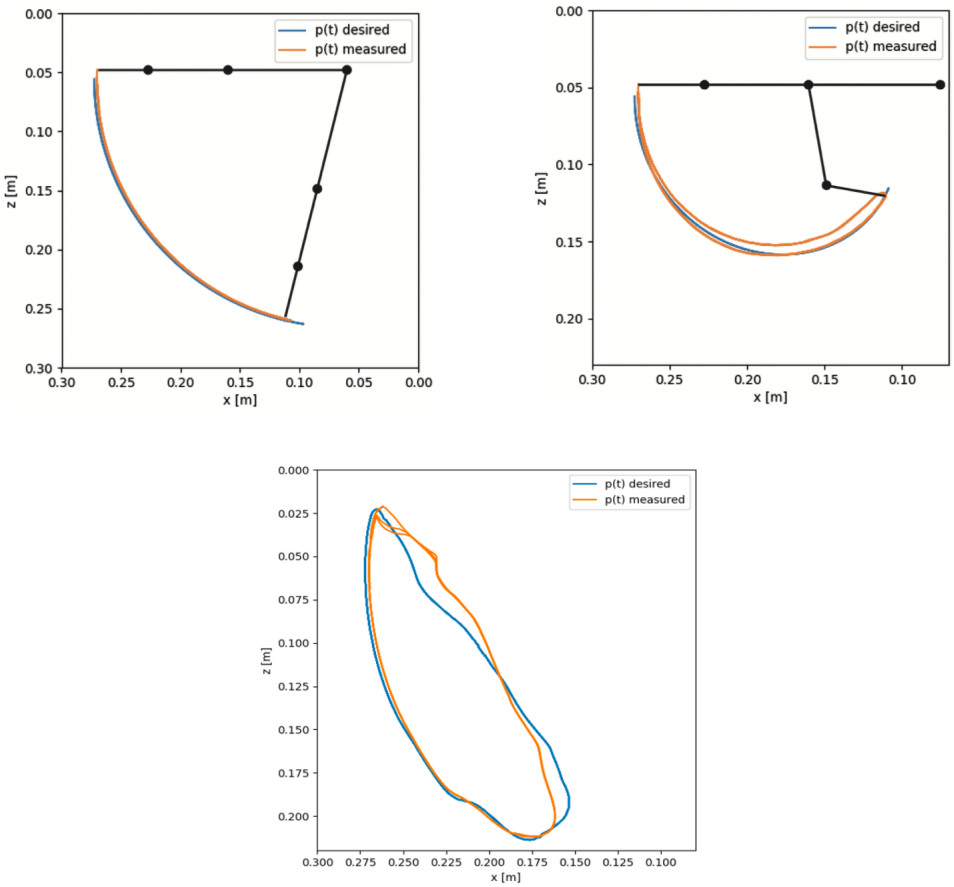


Figure 3.2: Comparison of a reference and approximated trajectory by using the mixed N position/M-N force control with the kinematic model and PID position control. The approximated trajectories (flexion with extended PIP, flexion with extended MCP and a complex trajectory) show close resemblance to their reference trajectories.

One CPG [24] has been tested on the physical setup which is inspired by the Hopf Oscillator [21]. It consists of two neurons and allows a custom shape, frequency and phase for each output of each neuron. It is given by:

$$\begin{aligned} \dot{I_n} &= \alpha(\mu - r_n^2)I_n - \omega_n J_n \\ \dot{J_n} &= \beta(\mu - r_n^2)J_n + \omega_n I_n + k\sum_m \Delta_{mn}, \end{aligned} \tag{3.1}$$

where Δ_{mn}, ω_n and r_n are respectively given by:

$$\begin{aligned} \Delta_{mn} &= J_m \cos(\theta_{mn}) + I_m \sin(\theta_{mn}), \\ \omega_n &= \frac{\omega_{\text{stance}}}{e^{-bI_n}+1} + \frac{\omega_{\text{swing}}}{e^{bI_n}+1}, \\ r_n &= \sqrt{I_n^2 + J_n^2}, \end{aligned} \tag{3.2}$$

The parameters in Eq. 3.1 and Eq. 3.2 are:

- α and β: two positive constants controlling the speed of convergence
- $\sqrt{\mu}$: the amplitude
- ω_{stance} and ω_{swing}: respectively the stance frequency (positive output of CPG) and the swing frequency (negative output of CPG)
- k: a coupling paramater
- θ_{mn}: phase between output m and n
- b: constant to acquire different frequencies during swing and stance

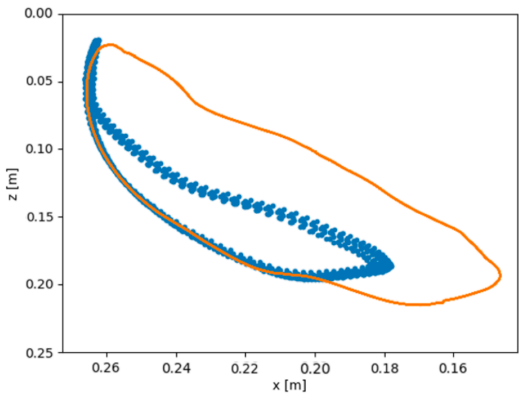
3.3.2 Approximating a trajectory by using evolutionary strategies

Covariance Matrix Adaptation Evolution Strategy (MA-ES) [11] is an evolutionary optimisation algorithm designed to optimise a set of parameters, it tries to minimise a certain loss function between the reference and approximated trajectory.

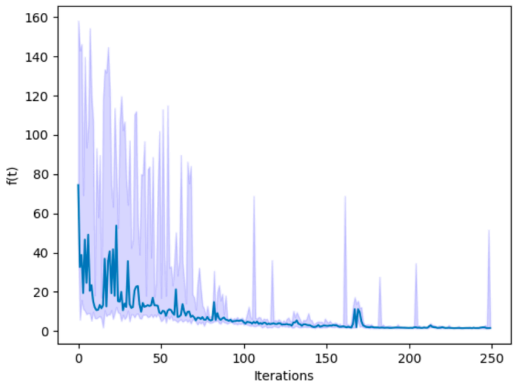
This algorithm was used to train the Hopf-based CPG. The reference and the approximated trajectory can be seen in Fig. 3.3a. It does not resemble the reference, thus clearly indicating that there is still a lot of room for improvement.

The downside of this method is the sheer number of iterations needed in order to converge into an acceptable state. Fig. 3.3b shows that the algorithm needs 250 iterations to converge. This is acceptable for a couple of trajectories, but repeating it for hundreds of times will result in significant wear.

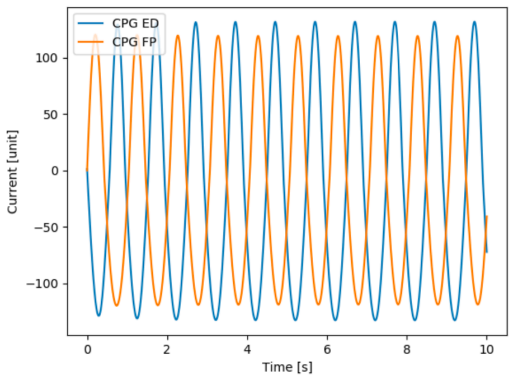
The outputs generated by the CPG are displayed in Fig. 3.3c. The amplitudes are fixed at all times which means it is not advanced enough to reproduce complex trajectories as sudden force changes are not possible.



(a) Comparison between the reference (orange) and approximated (blue) trajectory for force control with the Hopf-based CPG trained by the CMA-ES algorithm. The trajectories do not match, thus clearly indicating that there is still a lot of room for improvement.



(b) The CMA-ES loss function defined between the trajectory of the physical setup and the reference in function of the number of iterations. Around 250 iterations are needed in order to let the CMA-ES algorithm converge. This is acceptable for a couple of tryouts, but not for hundreds of times as this will result in physical wear.



(c) The output of the Hopf-based CPG after CMA-ES optimisation trying to reproduce the reference trajectory. The amplitudes remain constant. It is not flexible enough to reproduce the reference.

Figure 3.3: Overview of the results of the CMA-ES optimisation strategy.

3.4 Conclusion

The above mentioned control strategies each have their benefits and shortcomings.

The mixed N position/M-N force control strategy with the kinematic model and PID-controller showed high accuracy but lacked a universal controller and the possibility of reproducing fast trajectories.

The force control strategy with the Hopf-based CPG and CMA-ES algorithm showed a gap in its range of possible trajectories. It only allows two tendons to be actuated and shows no support for sudden force changes. These shortcomings identify the need for a more advanced CPG with force control to allow more complex trajectories, sudden force changes and fast movements. This advanced CPG is discussed in Chapter 7. In order to do efficient force control, the whole system should be described by a dynamic model, which is the topic of the next chapter. This will allow the calculation of the necessary forces for reproducing a reference trajectory.

Chapter 4

Modeling the physical setup as a dynamic model

A kinematic model allows to compute the position and orientation of the end-effector relative to a fixed point as a function of the joint variables. It does not allow to calculate which forces should be applied to reproduce a certain trajectory.

On the other hand, a dynamic model is defined by the forces itself. When given a set of input forces over time, the trajectory can be calculated as a function over time. Since there is a need for a more advanced force controlled strategy, its presence is crucial to allow repetitive simulation and training as the physical system does not allow tens of thousands of iterations in a short amount of time.

This chapter focuses on the design and implementation of a dynamic model of the physical robotic finger.

4.1 Lagrangian mechanics

There are many possible ways to model a physical system such as classical Newtonian mechanics, Hamiltonian mechanics and Lagrangian mechanics. Each has its own advantages and shortcomings. However, Lagrangian mechanics is chosen to be the preferred way of modeling the robotic finger since it is easily extendable and simple to understand [10].

Lagrangian mechanics introduce a new way of looking at a system of bodies. First the Euler-Lagrange equations are constructed and solved to find the equations of motion which are of the well known form $F = ma$ as defined in the classical Newtonian mechanics. Solving these equations of motion provides the trajectory of the system of bodies.

To construct the Euler-Lagrange equations, a term representing the dynamics of the system is necessary. It is called the Lagrangian (L) which is a mathematical function of the generalized coordinates of a system, their time derivatives and the time itself. It is defined as the difference between the kinetic energy (T) and the potential energy (V):

$$L = T - V. \tag{4.1}$$

The Euler-Lagrange equations are then defined as:

$$\frac{d}{dt}(\frac{\partial L}{\partial \dot{q_i}}) - \frac{\partial L}{\partial q_i} = F_i, \tag{4.2}$$

where L is the Lagrangian as defined in Eq. 4.1, q_i the generalized coordinates of body i, and F_i the acting forces on body i. By solving the Euler-Lagrange equations (Eq. 4.2), the equations of motions are obtained which can be used to calculate the trajectories each body will travel by defining the acting forces (F_i).

4.1.1 SymPy

Since the robotic finger is quite complex and the manual calculation of the equations of motion would be rather tedious and time-consuming, the SymPy Python package [19] is used. It allows the symbolic calculation of Lagrangian mechanics by using built-in functions.

4.2 Finger model

4.2.1 Representation and defining variables

The robotic finger is represented as shown in Fig. 4.1, the position of each phalanx is determined by its absolute angle θ_i, length l_i and connected joint(s) (MCP, DIP and PIP). It also shows the relative angles α_i and the joint torques τ_i.

The first step is to define which variables are dynamic and which are static. In Eq. 4.2 the generalised coordinates q_i are defined as the absolute angles of each phalanx (θ_i), where i ranges from 1 to 3 to respectively map on each phalanx: proximal, middle and distal. These angles and their velocities (derivatives with respect to time), along with the force on the tendons connected to the FP, IO, FS, ED are dynamic.

The other values which contain crucial information are the lengths, masses, gravitational constant and the friction coefficients. These are all static.

The torques (τ_i) are defined on each joint i, they are calculated by using the force of the tendons and are thus not explicitly defined as variables.

The syntax on how the actual implementation in Python is done can be found in Appendix B. The other necessary variables to construct the Lagrangian are now calculated.

Inertias

Each phalanx is represented as a beam. The inertia of each phalanx is then defined by:

$$I_i = \frac{m_i l_i^2}{12} \tag{4.3}$$

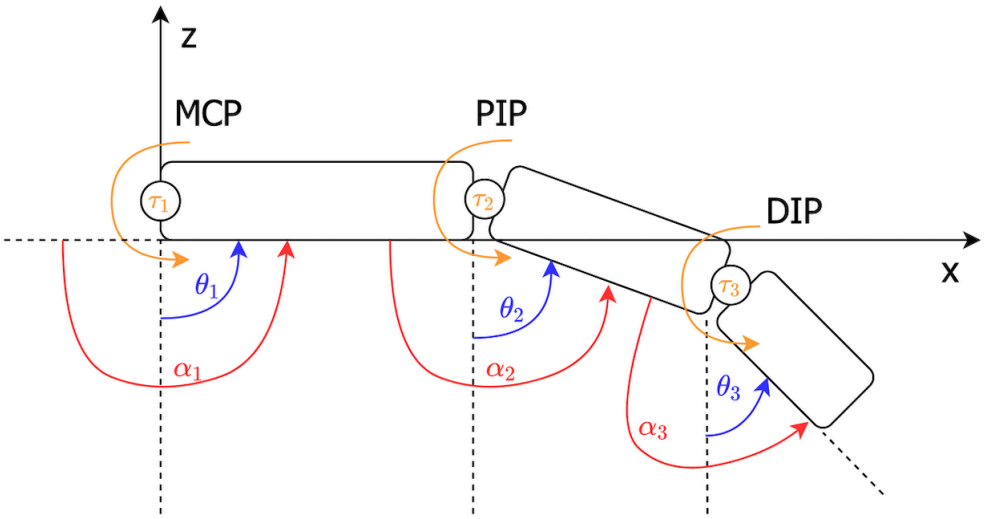


Figure 4.1: Representation of the finger with its absolute angles (θ_i), relative angles (α_i), torques (τ_i) and coordinate system.

Phalanx coordinates

The coordinates of the proximal phalanx:

$$\begin{aligned} x_1 &= l_1 \sin(\theta_1), \\ z_1 &= -l_1 \cos(\theta_1), \end{aligned} \tag{4.4}$$

for the middle phalanx:

$$\begin{aligned} x_2 &= x_1 + l_2 \sin(\theta_2), \\ z_2 &= z_1 - l_2 \cos(\theta_2), \end{aligned} \tag{4.5}$$

and for the distal phalanx:

$$\begin{aligned} x_3 &= x_2 + l_3 \sin(\theta_3), \\ z_3 &= z_2 - l_3 \cos(\theta_3). \end{aligned} \tag{4.6}$$

Center of mass

The center of mass for each phalanx is simplified to the center of each beam. The positions of the center of masses then become:

$$
\begin{aligned}
x_{c1} &= \frac{l_1}{2}\sin(\theta_1), \\
z_{c1} &= -\frac{l_1}{2}\cos(\theta_1), \\
\\
x_{c2} &= x_1 + \frac{l_2}{2}\sin(\theta_2), \\
z_{c2} &= z_1 - \frac{l_2}{2}\cos(\theta_2), \\
\\
x_{c3} &= x_2 + \frac{l_3}{2}\sin(\theta_3), \\
z_{c3} &= z_2 - \frac{l_3}{2}\cos(\theta_3).
\end{aligned}
\tag{4.7}
$$

Their velocities, which are their derivatives with respect to time are also necessary: $\dot{x_{c1}}, \dot{z_{c1}}, \dot{x_{c2}}, \dot{z_{c2}}, \dot{x_{c3}}, \dot{z_{c3}}$

Relative angles

The relative angles of each joint (α_i) and their derivatives with respect to time ($\dot{\alpha_i}$) are the last part necessary before calculating the Lagrangian (L):

$$
\begin{aligned}
\alpha_1 &= \frac{\pi}{2} + \theta_1, \\
\alpha_2 &= \pi - (\theta_1 - \theta_2), \\
\alpha_3 &= \pi + (\theta_2 - \theta_3),
\end{aligned}
\tag{4.8}
$$

4.2.2 Lagrangian

With the previous defined dynamic and static variables, the Lagrangian (L) as given in Eq. 4.1 can now be constructed.

The total potential energy is given as the sum of the potential energy of each phalanx:

$$
V = \sum_{i=1}^{3} m_i g z_{ci} \tag{4.9}
$$

The total kinetic energy is given as the sum of the kinetic energy of each phalanx:

$$
T = \sum_{i=1}^{3} \left(\frac{m_i(\dot{x_{ci}}^2 + \dot{z_{ci}}^2)}{2} + \frac{\dot{\theta_i}^2 I_i}{2}\right) \tag{4.10}
$$

Eq. 4.10 and 4.9 can now be filled in Eq. 4.1 to find the Lagrangian (L):

$$L = \frac{I_1 \left(\frac{d}{dt}\theta_1(t)\right)^2}{2} + \frac{I_2 \left(\frac{d}{dt}\theta_2(t)\right)^2}{2} + \frac{I_3 \left(\frac{d}{dt}\theta_3(t)\right)^2}{2} + \frac{g l_1 m_1 \cos\left(\theta_1(t)\right)}{2} + g m_2 \left(l_1 \cos\left(\theta_1(t)\right) + 0.5 l_2 \cos\left(\theta_2(t)\right)\right) + g m_3 \left(l_1 \cos\left(\theta_1(t)\right) + l_2 \cos\left(\theta_2(t)\right) + 0.5 l_3 \cos\left(\theta_3(t)\right)\right) + 0.125 l_1^2 m_1 \left(\frac{d}{dt}\theta_1(t)\right)^2 + \frac{m_2 \left(1.0 l_1^2 \left(\frac{d}{dt}\theta_1(t)\right)^2 + 1.0 l_1 l_2 \cos\left(\theta_1(t) - \theta_2(t)\right) \frac{d}{dt}\theta_1(t) \frac{d}{dt}\theta_2(t) + 0.25 l_2^2 \left(\frac{d}{dt}\theta_2(t)\right)^2\right)}{2} + m_3 \left(\frac{1.0 l_1^2 \left(\frac{d}{dt}\theta_1(t)\right)^2 + 2.0 l_1 l_2 \cos\left(\theta_1(t) - \theta_2(t)\right) \frac{d}{dt}\theta_1(t) \frac{d}{dt}\theta_2(t) + 1.0 l_1 l_3 \cos\left(\theta_1(t) - \theta_3(t)\right)}{2}\right) + m_3 \left(\frac{\frac{d}{dt}\theta_1(t) \frac{d}{dt}\theta_3(t) + 1.0 l_2^2 \left(\frac{d}{dt}\theta_2(t)\right)^2 + 1.0 l_2 l_3 \cos\left(\theta_2(t) - \theta_3(t)\right) \frac{d}{dt}\theta_2(t) \frac{d}{dt}\theta_3(t) + 0.25 l_3^2 \left(\frac{d}{dt}\theta_3(t)\right)^2}{2}\right) \tag{4.11}$$

4.2.3 Joint friction

The joints on the physical setup have a certain rotational friction. Modelling this is crucial to find an accurate model. Joint friction can be modelled as:

$$\tau_{fr} = -c_{fr}\dot{\alpha_i}, \tag{4.12}$$

where $\dot{\alpha_i}$ is the velocity of the relative angle of joint i, and c_{fr} the friction coefficient. The higher it is, the higher the friction and vice-versa.

4.2.4 Ligaments

The ligaments, which prevent the joints from over straining, can be modelled as dampers that activate once a joint exceeds a certain angle boundary. Although this will stop the joint from exceeding its limit any further, it is not possible to apply a torque to move it back into its allowed range as the damper will stay active once the joint has exceeded its range. To solve this, an extra condition is necessary stating that opposite torques are allowed.

A damper that activates at a certain value can be defined as a piece-wise function:

$$\text{ligament}(\alpha_i) = \begin{cases} -\dot{\alpha_i} c_l & \text{if } \alpha_i \leq \alpha_{i_{\text{min}}} \wedge 0 < \tau_i, \\ -\dot{\alpha_i} c_l & \text{if } \alpha_{i_{\text{max}}} \leq \alpha_i \wedge \tau_i < 0, \\ 0 & \text{otherwise.} \end{cases} \tag{4.13}$$

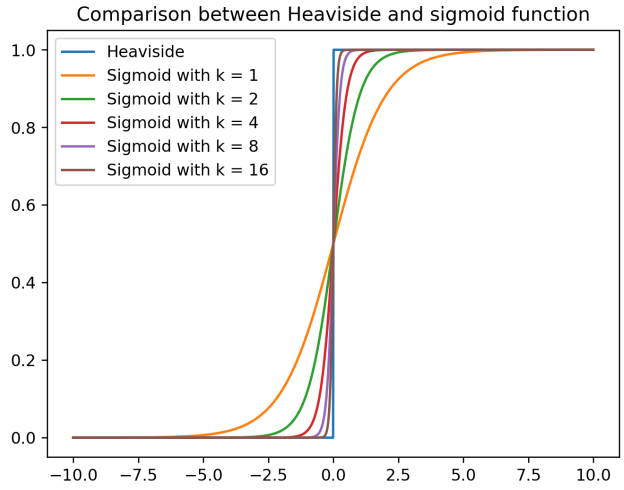


Figure 4.2: Comparison between the sigmoid function and the Heaviside function. The sigmoid function with $k = 16$ is a sufficient replacement for the Heaviside function.

where c_l is the coefficient of the ligament, this is an arbitrary value that can be chosen to have a desired "stretch" on the ligament, i.e. if an abrupt stop is preferred it should be high, if it should allow some tolerance it can be lower.

Eq. 4.13 can be translated to use Heaviside to have a one line function. However, since the Heaviside is not a continuous function, it cannot be used in the model. Therefore the Heaviside function is approximated using a sigmoid function:

$$\text{Heaviside}(\alpha_i) \approx \sigma(k, \alpha_i) = \frac{1}{1 + e^{-k\alpha_i}}, \tag{4.14}$$

where k is a constant indicating the sharpness of the transition. Fig. 4.2 shows a comparison between the sigmoid function with varying values for k and the Heaviside function. The sigmoid function with $k = 16$ is a sufficient replacement for the Heaviside function. The one line ligament function then becomes:

$$\text{ligament}(\alpha_i) = -\sigma(k, -\tau_i)\sigma(k, \alpha_{i_{\min}} - \alpha_i)c_l\dot{\alpha_i} - \sigma(k, \tau_i)\sigma(k, \alpha_i - \alpha_{i_{\max}})c_l\dot{\alpha_i} \tag{4.15}$$

The relative angle boundaries for each joint are defined in Table 4.1.

Joint i	$\alpha_{i_{\text{min}}}$ (°)	$\alpha_{i_{\text{max}}}$ (°)
MCP	95	225
PIP	60	220
DIP	80	185

Table 4.1: Relative angle boundaries of each joint (α_i)

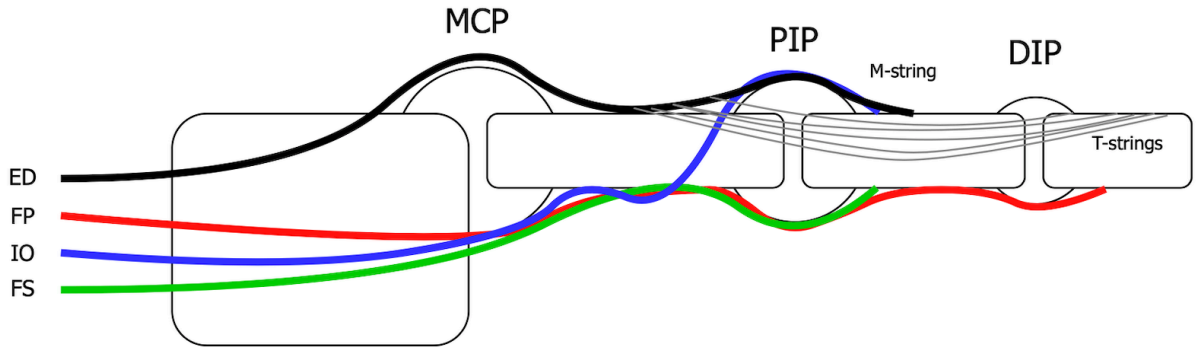


Figure 4.3: Finger representation with the tendon tracks. For the MCP joint, the FS, IO and FP pass through the palmar and the ED through the dorsal. For the PIP joint, the FS and FP pass through the palmar and the IO and ED (the M-string) through the dorsal. For the DIP joint, the FP passes through the palmar and the ED (the T-strings) through the dorsal. This figure is not on scale.

4.2.5 Tendons

To solve the Euler-Lagrange equations (Eq. 4.2), the acting forces have to be defined. These are the forces applied on the tendons by the muscles, but they cannot be used as is, as there are only rotational movements. That means the forces will have to be converted to the corresponding torques on each joint. Fig. 4.3 shows the tendons along with their tracks, each tendon has its own unique moment arm on each joint.

The total torque on each joint is defined by the sum of the torques generated by the tendons on that joint.

MCP joint

The FS, IO and FP pass through the palmar and the ED through the dorsal:

$$\tau_{\text{MCP}} = \tau_{\text{FS}_{\text{MCP}}} + \tau_{\text{IO}_{\text{MCP}}} + \tau_{\text{FP}_{\text{MCP}}} - \tau_{\text{ED}_{\text{MCP}}}, \tag{4.16}$$

which after filling in the moment arms becomes:

$$\tau_{\text{MCP}} = r_{\text{FS}_{\text{MCP}}} F_{\text{FS}} + r_{\text{IO}_{\text{MCP}}} F_{\text{IO}} + r_{\text{FP}_{\text{MCP}}} F_{\text{FP}} - r_{\text{ED}_{\text{MCP}}} F_{\text{ED}}. \tag{4.17}$$

PIP joint

Unlike in the MCP joint, the IO passes through the dorsal:

$$\begin{aligned}\tau_{\text{PIP}} &= \tau_{\text{FS}_{\text{PIP}}} - \tau_{\text{IO}_{\text{PIP}}} + \tau_{\text{FP}_{\text{PIP}}} - \tau_{\text{ED}_{\text{PIP}}} \\ &= r_{\text{FS}_{\text{PIP}}} F_{\text{FS}} - r_{\text{IO}_{\text{PIP}}} F_{\text{IO}} + r_{\text{FP}_{\text{PIP}}} F_{\text{FP}} - r_{\text{ED}_{\text{PIP}}} F_{\text{ED}}\end{aligned} \tag{4.18}$$

DIP joint

The DIP joint is only influenced by the FP and ED:

$$\begin{aligned}\tau_{\text{DIP}} &= \tau_{\text{FP}_{\text{DIP}}} - \tau_{\text{ED}_{\text{DIP}}} \\ &= r_{\text{FP}_{\text{DIP}}} F_{\text{FP}} - \tau_{\text{ED}_{\text{DIP}}},\end{aligned} \tag{4.19}$$

however, the torque generated by the ED on the DIP joint (the T-strings) is not as straightforward, since it has a changing moment arm depending on the relative angle of the PIP joint. It can be simplified by normalising the PIP angle range to a range of 0 and 1. This normalised range is then multiplied by the torque generated by the ED on the DIP joint. If e.g. the PIP angle is 60°, the torque on the DIP joint generated by the ED will be 0. If it is halfway (140°) it will be half of the torque and if it is 220 ° the full torque will be applied:

$$\tau_{\text{ED}_{\text{DIP}}} = F_{\text{ED}} \frac{\alpha_{\text{PIP}} - \alpha_{\text{PIP}_{\text{min}}}}{\alpha_{\text{PIP}_{\text{max}}} - \alpha_{\text{PIP}_{\text{min}}}} l_{\text{dp}}, \tag{4.20}$$

where l_{dp} is the distance on which the T-strings are attached on the proximal phalanx.

4.2.6 Forming the Euler-Lagrange equations and the equations of motion

The Euler-Lagrangian equations (Eq. 4.2) can now be formed by passing in the Lagrangian (L), the friction, the ligaments and the torque equations. The solution to these equations is a mass and force matrix which can be used to construct the equations of motion of the well known form in classical Newtonian mechanics:

$$\begin{aligned} & F = MA \\ \iff & A = M^{-1}F \end{aligned} \tag{4.21}$$

where M is the mass matrix (Fig. 4.4), F the force matrix (Fig. 4.5) and A the acceleration matrix which defines the trajectory. The next chapter will focus on how to solve and simulate these equations.

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & I_1 + 0.25l_1^2 m_1 + l_1^2 m_2 + l_1^2 m_3 & l_1 l_2 (0.5m_2 + m_3)\cos(\theta_1(t) - \theta_2(t)) & 0.5l_1 l_3 m_3 \cos(\theta_1(t) - \theta_3(t)) \\ 0 & 0 & 0 & l_1 l_2 (0.5m_2 + m_3)\cos(\theta_1(t) - \theta_2(t)) & I_2 + 0.25l_2^2 m_2 + l_2^2 m_3 & 0.5l_2 l_3 m_3 \cos(\theta_2(t) - \theta_3(t)) \\ 0 & 0 & 0 & 0.5l_1 l_3 m_3 \cos(\theta_1(t) - \theta_3(t)) & 0.5l_2 l_3 m_3 \cos(\theta_2(t) - \theta_3(t)) & I_3 + 0.25l_3^2 m_3 \end{bmatrix}$$

Figure 4.4: The mass matrix generated by SymPy.

$$
\begin{bmatrix}
\frac{d}{dt}\theta_1(t) \\
\frac{d}{dt}\theta_2(t) \\
\frac{d}{dt}\theta_3(t) \\
-0.5gl_1m_1\sin(\theta_1(t)) - 1.0gl_1m_2\sin(\theta_1(t)) - 1.0gl_1m_3\sin(\theta_1(t)) - 0.5l_1l_2m_2\sin(\theta_1(t)-\theta_2(t))\left(\frac{d}{dt}\theta_2(t)\right)^2 - 1.0l_1l_2m_3\sin \\
(\theta_1(t)-\theta_2(t))\left(\frac{d}{dt}\theta_2(t)\right)^2 - 0.5l_1l_3m_3\sin(\theta_1(t)-\theta_3(t))\left(\frac{d}{dt}\theta_3(t)\right)^2 + 0.018\,\mathrm{F_{ed}}(t) - 0.022\,\mathrm{F_{fp}}(t) - 0.026\,\mathrm{F_{fs}}(t) - 0.012\,\mathrm{F_{io}}(t) \\
-25.0\theta(-\theta_1(t)+0.0277777777777778\pi)\,\theta\left(-0.018\,\mathrm{F_{ed}}(t)+0.022\,\mathrm{F_{fp}}(t)+0.026\,\mathrm{F_{fs}}(t)+0.012\,\mathrm{F_{io}}(t)\right)\frac{d}{dt}\theta_1(t) \\
-25.0\theta(\theta_1(t)-0.75\pi)\,\theta\left(0.018\,\mathrm{F_{ed}}(t)-0.022\,\mathrm{F_{fp}}(t)-0.026\,\mathrm{F_{fs}}(t)-0.012\,\mathrm{F_{io}}(t)\right)\frac{d}{dt}\theta_1(t) \\
-25.0\theta(-\theta_1(t)+\theta_2(t)-0.222222222222222\pi)\,\theta\left(0.01\,\mathrm{F_{ed}}(t)-0.021\,\mathrm{F_{fp}}(t)-0.017\,\mathrm{F_{fs}}(t)+0.01\,\mathrm{F_{io}}(t)\right)\frac{d}{dt}\theta_1(t) \\
+25.0\theta(-\theta_1(t)+\theta_2(t)-0.222222222222222\pi)\,\theta\left(0.01\,\mathrm{F_{ed}}(t)-0.021\,\mathrm{F_{fp}}(t)-0.017\,\mathrm{F_{fs}}(t)+0.01\,\mathrm{F_{io}}(t)\right)\frac{d}{dt}\theta_2(t) \\
-25.0\theta(\theta_1(t)-\theta_2(t)-0.666666666666667\pi)\,\theta\left(-0.01\,\mathrm{F_{ed}}(t)+0.021\,\mathrm{F_{fp}}(t)+0.017\,\mathrm{F_{fs}}(t)-0.01\,\mathrm{F_{io}}(t)\right)\frac{d}{dt}\theta_1(t) \\
+25.0\theta(\theta_1(t)-\theta_2(t)-0.666666666666667\pi)\,\theta\left(-0.01\,\mathrm{F_{ed}}(t)+0.021\,\mathrm{F_{fp}}(t)+0.017\,\mathrm{F_{fs}}(t)-0.01\,\mathrm{F_{io}}(t)\right)\frac{d}{dt}\theta_2(t) - 0.2\frac{d}{dt}\theta_1(t) + 0.1\frac{d}{dt}\theta_2(t) \\
-0.5gl_2m_2\sin(\theta_2(t)) - 1.0gl_2m_3\sin(\theta_2(t)) + 0.5l_1l_2m_2\sin(\theta_1(t)-\theta_2(t))\left(\frac{d}{dt}\theta_1(t)\right)^2 + 1.0l_1l_2m_3\sin(\theta_1(t)-\theta_2(t))\left(\frac{d}{dt}\theta_1(t)\right)^2 - 0.5l_2l_3m_3\sin \\
(\theta_2(t)-\theta_3(t))\left(\frac{d}{dt}\theta_3(t)\right)^2 + 0.01\,\mathrm{F_{ed}}(t) - 0.021\,\mathrm{F_{fp}}(t) - 0.017\,\mathrm{F_{fs}}(t) + 0.01\,\mathrm{F_{io}}(t) \\
+25.0\theta(-\theta_1(t)+\theta_2(t)-0.222222222222222\pi)\,\theta\left(0.01\,\mathrm{F_{ed}}(t)-0.021\,\mathrm{F_{fp}}(t)-0.017\,\mathrm{F_{fs}}(t)+0.01\,\mathrm{F_{io}}(t)\right)\frac{d}{dt}\theta_1(t) \\
-25.0\theta(-\theta_1(t)+\theta_2(t)-0.222222222222222\pi)\,\theta\left(0.01\,\mathrm{F_{ed}}(t)-0.021\,\mathrm{F_{fp}}(t)-0.017\,\mathrm{F_{fs}}(t)+0.01\,\mathrm{F_{io}}(t)\right)\frac{d}{dt}\theta_2(t) \\
+25.0\theta(\theta_1(t)-\theta_2(t)-0.666666666666667\pi)\,\theta\left(-0.01\,\mathrm{F_{ed}}(t)+0.021\,\mathrm{F_{fp}}(t)+0.017\,\mathrm{F_{fs}}(t)-0.01\,\mathrm{F_{io}}(t)\right)\frac{d}{dt}\theta_1(t) \\
-25.0\theta(\theta_1(t)-\theta_2(t)-0.666666666666667\pi)\,\theta\left(-0.01\,\mathrm{F_{ed}}(t)+0.021\,\mathrm{F_{fp}}(t)+0.017\,\mathrm{F_{fs}}(t)-0.01\,\mathrm{F_{io}}(t)\right)\frac{d}{dt}\theta_2(t) \\
-25.0\theta(-\theta_2(t)+\theta_3(t)-0.0277777777777779\pi)\,\theta\left(-\frac{0.008689275\,\mathrm{F_{ed}}(t)\theta_1(t)}{\pi}+\frac{0.008689275\,\mathrm{F_{ed}}(t)\theta_2(t)}{\pi}+0.00579285\,\mathrm{F_{ed}}(t)-0.012\,\mathrm{F_{fp}}(t)\right)\frac{d}{dt}\theta_2(t) \\
+25.0\theta(-\theta_2(t)+\theta_3(t)-0.0277777777777779\pi)\,\theta\left(-\frac{0.008689275\,\mathrm{F_{ed}}(t)\theta_1(t)}{\pi}+\frac{0.008689275\,\mathrm{F_{ed}}(t)\theta_2(t)}{\pi}+0.00579285\,\mathrm{F_{ed}}(t)-0.012\,\mathrm{F_{fp}}(t)\right)\frac{d}{dt}\theta_3(t) \\
-25.0\theta(\theta_2(t)-\theta_3(t)-0.555555555555556\pi)\,\theta\left(\frac{0.008689275\,\mathrm{F_{ed}}(t)\theta_1(t)}{\pi}-\frac{0.008689275\,\mathrm{F_{ed}}(t)\theta_2(t)}{\pi}-0.00579285\,\mathrm{F_{ed}}(t)+0.012\,\mathrm{F_{fp}}(t)\right)\frac{d}{dt}\theta_2(t) \\
+25.0\theta(\theta_2(t)-\theta_3(t)-0.555555555555556\pi)\,\theta\left(\frac{0.008689275\,\mathrm{F_{ed}}(t)\theta_1(t)}{\pi}-\frac{0.008689275\,\mathrm{F_{ed}}(t)\theta_2(t)}{\pi}-0.00579285\,\mathrm{F_{ed}}(t)+0.012\,\mathrm{F_{fp}}(t)\right)\frac{d}{dt}\theta_3(t) \\
+0.1\frac{d}{dt}\theta_1(t) - 0.15\frac{d}{dt}\theta_2(t) + 0.05\frac{d}{dt}\theta_3(t) \\
-0.5gl_3m_3\sin(\theta_3(t)) + 0.5l_1l_3m_3\sin(\theta_1(t)-\theta_3(t))\left(\frac{d}{dt}\theta_1(t)\right)^2 + 0.5l_2l_3m_3\sin(\theta_2(t)-\theta_3(t))\left(\frac{d}{dt}\theta_2(t)\right)^2 - \frac{0.008689275\,\mathrm{F_{ed}}(t)\theta_1(t)}{\pi} \\
+\frac{0.008689275\,\mathrm{F_{ed}}(t)\theta_2(t)}{\pi} + 0.00579285\,\mathrm{F_{ed}}(t) - 0.012\,\mathrm{F_{fp}}(t) \\
+25.0\theta(-\theta_2(t)+\theta_3(t)-0.0277777777777779\pi)\,\theta\left(-\frac{0.008689275\,\mathrm{F_{ed}}(t)\theta_1(t)}{\pi}+\frac{0.008689275\,\mathrm{F_{ed}}(t)\theta_2(t)}{\pi}+0.00579285\,\mathrm{F_{ed}}(t)-0.012\,\mathrm{F_{fp}}(t)\right)\frac{d}{dt}\theta_2(t) \\
-25.0\theta(-\theta_2(t)+\theta_3(t)-0.0277777777777779\pi)\,\theta\left(-\frac{0.008689275\,\mathrm{F_{ed}}(t)\theta_1(t)}{\pi}+\frac{0.008689275\,\mathrm{F_{ed}}(t)\theta_2(t)}{\pi}+0.00579285\,\mathrm{F_{ed}}(t)-0.012\,\mathrm{F_{fp}}(t)\right)\frac{d}{dt}\theta_3(t) \\
+25.0\theta(\theta_2(t)-\theta_3(t)-0.555555555555556\pi)\,\theta\left(\frac{0.008689275\,\mathrm{F_{ed}}(t)\theta_1(t)}{\pi}-\frac{0.008689275\,\mathrm{F_{ed}}(t)\theta_2(t)}{\pi}-0.00579285\,\mathrm{F_{ed}}(t)+0.012\,\mathrm{F_{fp}}(t)\right)\frac{d}{dt}\theta_2(t) \\
-25.0\theta(\theta_2(t)-\theta_3(t)-0.555555555555556\pi)\,\theta\left(\frac{0.008689275\,\mathrm{F_{ed}}(t)\theta_1(t)}{\pi}-\frac{0.008689275\,\mathrm{F_{ed}}(t)\theta_2(t)}{\pi}-0.00579285\,\mathrm{F_{ed}}(t)+0.012\,\mathrm{F_{fp}}(t)\right)\frac{d}{dt}\theta_3(t) \\
+0.05\frac{d}{dt}\theta_2(t) - 0.05\frac{d}{dt}\theta_3(t)
\end{bmatrix}
$$

Figure 4.5: The force matrix generated by SymPy.

Chapter 5

Simulation and its accuracy

This chapter will focus on the simulation of the dynamic model and its accuracy compared to the physical setup. The equations of motion which were achieved in Chapter 4 are a set of ordinary differential equations (ODEs). These can be solved to find a trajectory where a certain range of forces on the tendons have been applied.

5.1 Implementation

The Python package SciPy [22] has a function for solving a set of ODEs called *odeint* which takes in a function representing the ODE, a vector of time instants and the initial positions. Although it is very user-friendly and efficiently implemented it is not suited to use any machine learning techniques which will be discussed in the next chapter, but it is sufficient enough to illustrate the simulation.

5.1.1 Formulating and solving the ODE

As mentioned in Chapter 4, the equations of motion take in the static variables and an initial position for the dynamic variables (comprised of the angles and angle velocities of each phalanx). It is then passed to the *odeint* function which returns a matrix A containing the angles of each joint and their velocities for each time instant in the provided interval. The full implementation is explained in Appendix B.

5.1.2 Animation

The matrix A contains all the necessary information to create animations which are crucial to visualise the trajectory of each phalanx. The angles are used to calculate the Cartesian coordinates of each phalanx which are then plotted on graphs for each time instant to be sequenced, resulting in an animation. Fig. 5.1 shows snapshots of an animated simulation of the simulated finger. Appendix B contains instructions on how to make custom animations.

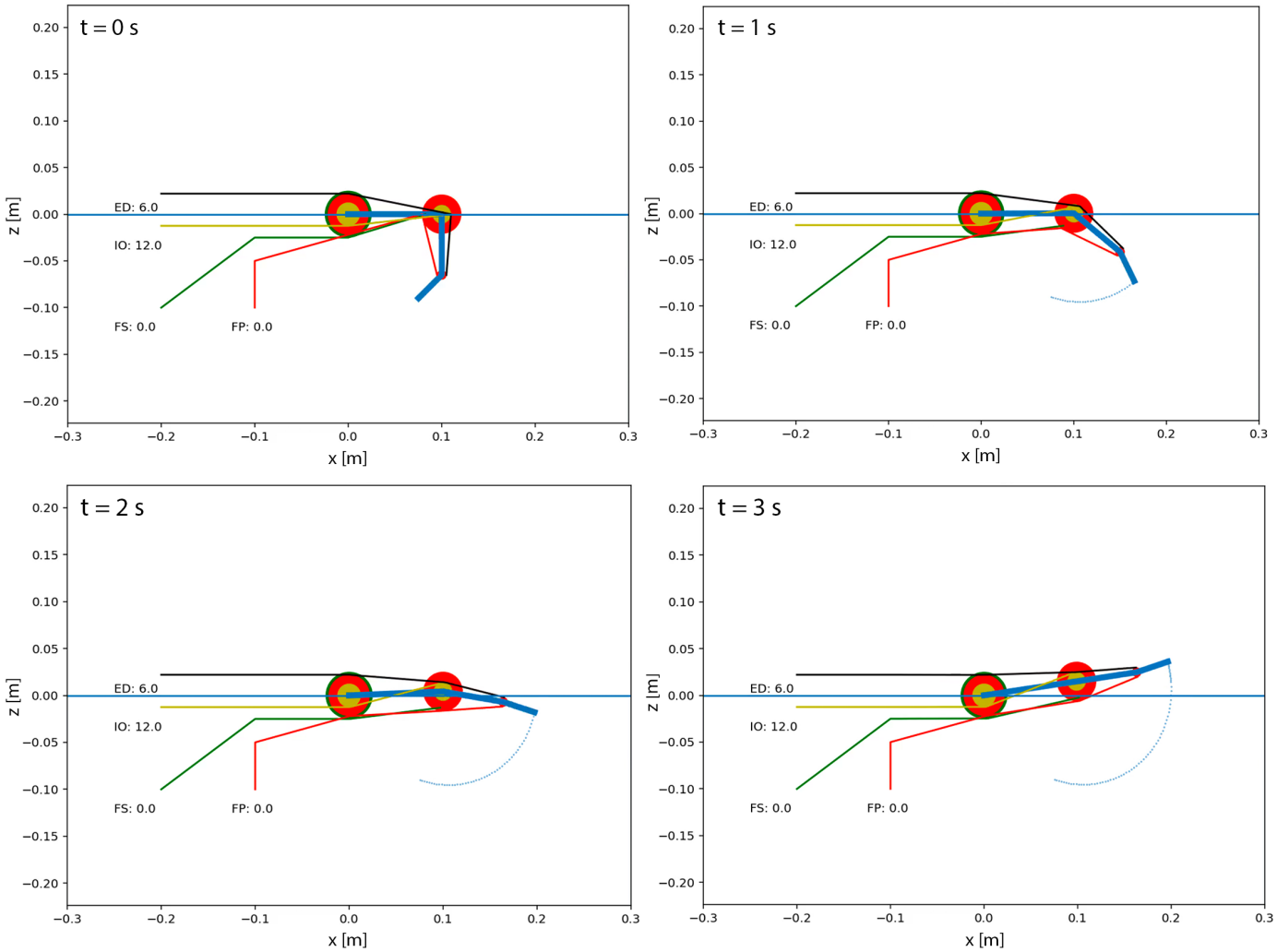


Figure 5.1: Snapshots of an animated simulation with a duration of 3 seconds of the finger with a constant force of 12 N on the IO and 6 N on the ED with an initial angle of $\alpha_{PIP} = 0$ and $\alpha_{DIP} = -\frac{\pi}{4}$ with no initial accelerations. The upper left snapshot is taken at $t = 0s$, the finger starts in a flexed state. The ED pulls with a force of 6 N causing the finger to extend. The MCP itself stays mostly stationary due to the greater force of the IO which counteracts the moment arm of the ED on the MCP. The PIP extends due both to the ED and IO. The PIP extends due to the ED. The other figures are respectively from upper right to bottom left to bottom right snapshots of the simulation at $t = 1$, $t = 2$ and the end state $t = 3$.

5.2 Accuracy

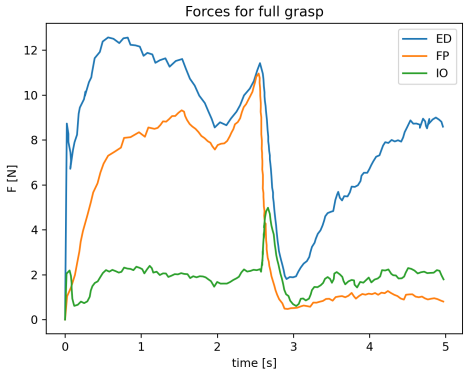
The accuracy between the simulated model and the physical setup is preferably as great as possible. The simulation can however never be completely accurate as some factors are not taken into consideration as they are believed to only have a minor impact on the overall accuracy of the model. A non-exhaustive list of unconsidered factors:

- Possible varying friction coefficients on each joint
- Measurement errors in the load bridge
- Measurement errors in the OptiTrack capture system
- Slip on the tendons
- Slip on the servomotors
- More accurate phalanx inertia
- More accurate location of the center of gravity
- Air resistance
- Tendon inertia

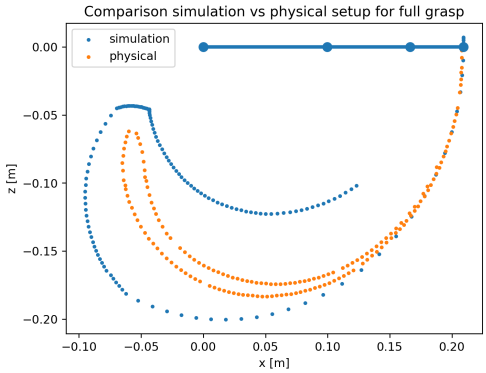
To measure the accuracy, the mixed position controlled method was used to approximate a certain provided trajectory using a PID-controller as done by A. Van Erum [9]. The reference trajectory is first converted into tendon displacements by using the kinematic model and then fed to the servocontroller. The forces on the tendons are measured by the load cells. These forces are then fed to the simulator resulting in an approximated simulated end-effector trajectory.

The accuracy of the IPJ-coupling mentioned in Chapter 2 is also discussed.

The finger in the graphs is represented by three blue dots and lines. The most left being the MCP joint, the middle the PIP joint and the most right the DIP joint. The blue scatter represents the simulated end-effector trajectory and the orange scatter the physical end-effector trajectory. Each simulation was done with a friction coefficient of 0.1.



(a) Forces for the grasp trajectory. Only the ED, FP and IO are used for this trajectory as it is not a loaded control. The IO is kept at 2N by the force PID-controller in the physical setup. The ED varies from 8N to 12N and then a sudden drop to 2N after which it goes back up to 8N over the course of two seconds. The FP roughly follows the same path except for a maximum of 8N and once it drops it stays around 1N. The sudden drop in ED and FP is to allow the finger to fully flex. The reason for the ED to be the only active force at $t = 3$ is because the finger has to fully extend again.



(b) Comparison of the simulation and physical setup for the full grasp trajectory. The finger is represented by three blue dots and lines. The most left being the MCP joint, the middle the PIP joint and the most right the PIP joint. The blue scatter represents the simulated trajectory. The orange scatter the physical trajectory.
The reference trajectory has the same overall end-effector velocity. While flexing, the velocity of the simulation trajectory is too high, resulting in a fast swing and thus preventing the PIP from fully bending. Once the force on the ED and FP drops, the finger remains stationary at a semi-flexed state. Once force on the ED rises again, the finger starts flexing. The PIP joint is however not extending enough, resulting in an extension where the PIP stays fully bent. This is because the IO does not go back to 2N but instead remains around 1N.

Figure 5.2: Overview of the accuracy of the full grasp trajectory.

5.2.1 Full grasp

This trajectory represents a full grasp of the finger. Fig. 5.2a shows the measured forces which were measured by the load cells. These forces are fed to the simulation. The trajectories are scattered as dots on the figures. If many subsequent dots are close to each other, it means that the velocity of the end-effector is greater than when the subsequent dots are more spread.

Fig. 5.2b shows the trajectories of the physical setup and of the simulation. It shows serious deviations but the overall movement is globally the same. The deviations can be explained due to factors which haven't been considered in the model.

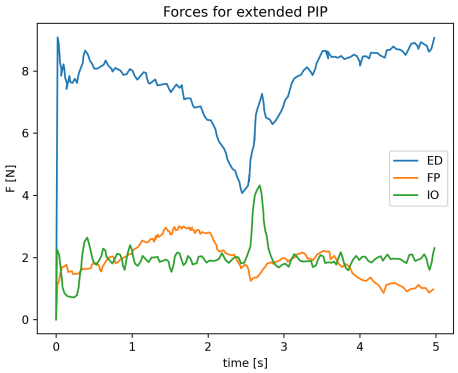
5.2.2 Extended PIP

The next trajectory is a full flexion of the MCP joint while keeping the PIP joint fully extended. The measured forces and the comparison between the trajectories can respectively be seen in Fig. 5.3a and Fig. 5.3b.

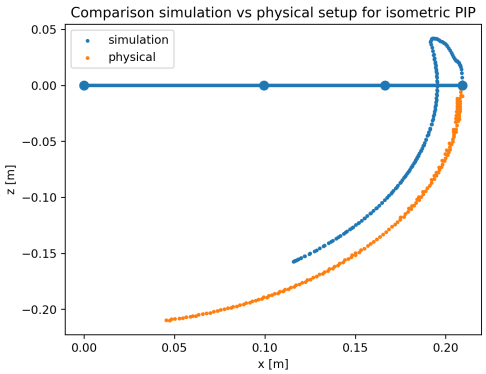
The trajectories differ but show close resemblance. The reason why the end-effector first goes up and then down is because the PIP bends over the dorsal side and reaches its ligament boundary before bending the MCP joint to the palmar side.

The PIP joint itself is kept extended only by muscle action and is otherwise unconstrained (except for its ligaments), so this means only a slight error in the friction coefficient or measured forces can show a deviant trajectory like this.

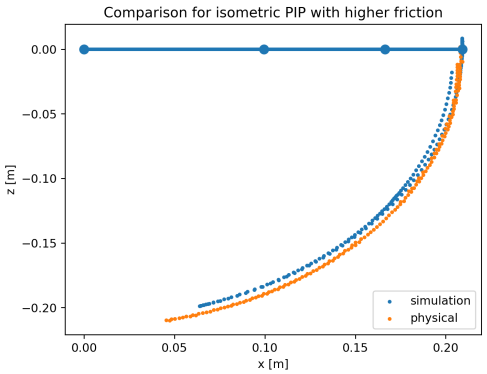
This slight deviation can be solved by increasing the friction coefficient on the PIP joint to 0.5, thus preventing the PIP joint from over extending. It now closely resembles the physical trajectory (Fig. 5.3c).



(a) Forces for the extended PIP trajectory which is a full flexion of the MCP joint while keeping the PIP joint fully extended. The IO is also kept at 2N to serve as a antagonistic force for the ED and FP. The ED starts high to ensure the full extension of the PIP, it gradually declines to allow the MCP to bend. Halfway it increases again to start extending the MCP back to its original position.



(b) Comparison of the simulation and physical setup for the extended PIP trajectory which is a full flexion of the MCP joint while keeping the PIP joint fully extended. The trajectories differ but show close resemblance. The reason why the end-effector first goes up and then down is because the PIP bends over the dorsal side and reaches its ligament boundary before bending the MCP joint. The velocities of the end-effector for the physical and simulated trajectory are almost equal.



(c) Comparison of the simulation and physical setup for the extended PIP trajectory with a higher friction on the PIP joint. The problem with the overextending of the PIP joint can be solved by using a higher friction coefficient on the PIP joint. It shows an almost equal reproduction of the trajectory of the physical setup.

Figure 5.3: Overview of the accuracy of the extended PIP trajectory.

5.2.3 Extended MCP

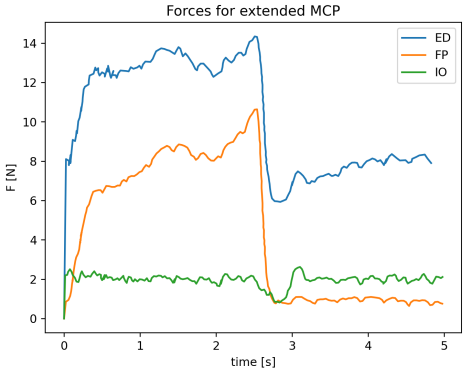
The next trajectory is a full flexion of the PIP joint while keeping the MCP joint fully extended. The measured forces and the comparison between the trajectories can respectively be seen in Fig. 5.4a and Fig. 5.4b.

The trajectories differ greatly in distance, but show close resemblance in the actual shape. The force generated by the FP is too high for the simulated MCP joint since it immediately starts bending to the palmar side. The velocity of the simulated end-effector trajectory is similar to the physical setup.

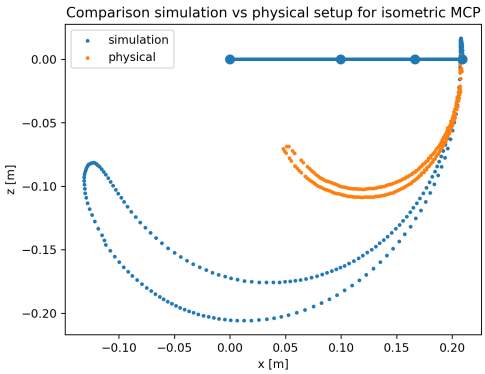
This can be approximately solved by increasing the friction of the MCP joint to prevent it from bending. A friction coefficient of 0.5 is not enough, 1.0 is the absolute minimum to prevent the MCP from bending. This indicates that the forces measured by the physical setup have to be decreased in order to prevent the MCP from bending.

As Fig. 5.4c indicates, with a higher friction on the MCP joint, the trajectories resemble more closely while flexing. The velocity is higher than the physical velocity resulting in a swing and thus slightly extending the PIP joint.

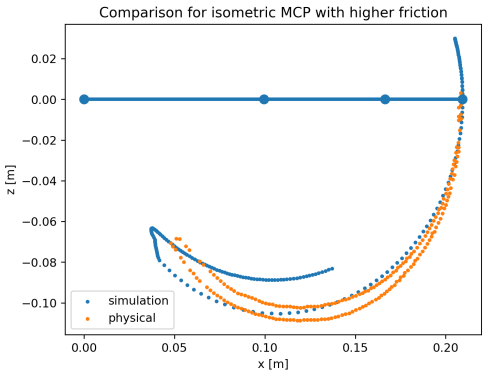
The extension is however not as accurate as its velocity is much lower compared to the physical trajectory, preventing the finger from returning to its original position. This indicates that the friction on the MCP joint is too high while extending and should be lowered.



(a) Forces for the extended MCP trajectory which is a full flexion of the PIP joint while keeping the MCP joint fully extended. For the flexion, averagely seen, the ED is kept at 13N and the FP at 9N. The reason the force on the ED is higher is to ensure the MCP stays stationary. Once the PIP joint is fully flexed, the FP drops to 0 as it impacts the bending of the PIP joint to the dorsal side. The ED can then lower to an average of 8N to ensure a smooth extension.

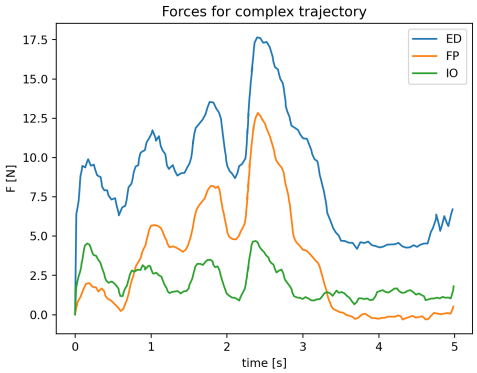


(b) Comparison of the simulation and physical setup for the full flexion but with extended MCP trajectory. The trajectories differ greatly in distance, but show close resemblance in the actual shape. The force generated by FP is too high for the simulated MCP joint since it immediately starts bending to the palmar side. The velocity of the simulated end-effector trajectory is similar to the physical setup.

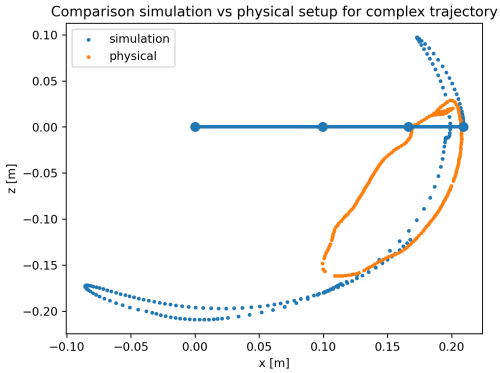


(c) Comparison of the simulation and physical setup for the full flexion but with extended MCP trajectory with a higher friction on the MCP joint. The problem of the MCP bending to the palmar side can be prevented by increasing the friction on the MCP joint. The flexion shows a much closer resemblance to the physical setup, the extension however is much slower thus preventing the finger from going back to its original full extended state. It shows that the problem does not lie in the friction but in other not considered factors.

Figure 5.4: Overview of the accuracy of the extended MCP trajectory.



(a) Forces for the complex trajectory. The ED, FP, IO all go higher and lower with the same frequency until the PIP has fully flexed. The FP drops to 0, IO remains at 2N and the ED lowers to 5N to ensure a smooth full extension.



(b) Comparison of the simulation and physical setup for the complex trajectory. The trajectories do not resemble at all. Fine-tuning the friction did not result in any improvements on the accuracy of the simulator. It indicates that complex trajectories are not accurate and should thus be avoided. The accuracy can be improved by taking the factors mentioned in section 5.2 into consideration.

Figure 5.5: Overview of the accuracy of the complex trajectory.

5.2.4 Complex trajectory

The last trajectory is an arbitrary complex trajectory with multiple joint movements, the end-effector traverses an oval-like shape. The measured forces and the comparison between the trajectories can respectively be seen in Fig. 5.5a and Fig. 5.5b. It shows a significantly different trajectory.

Fine-tuning the frictions did not result in a better simulated reproduction of the physical trajectory. The deviations are thus most likely due to the non-exhaustive list of external factors (mentioned in section 5.2) which haven't been taken into consideration. It indicates that complex trajectories are not accurate and should thus be avoided.

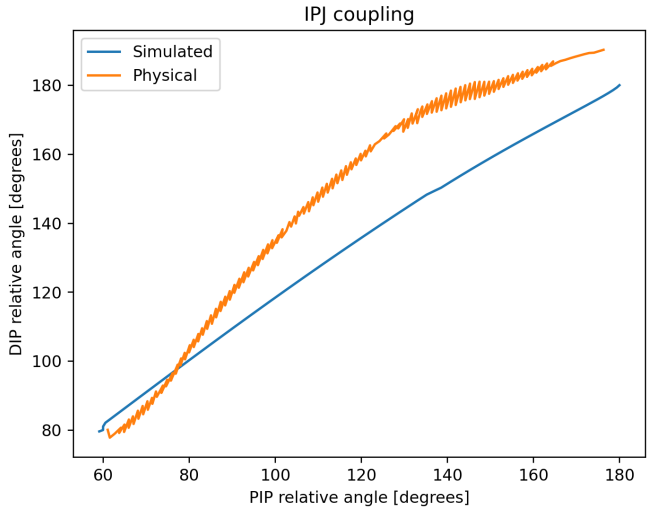


Figure 5.6: Comparison between the IPJ-coupling of the physical setup and the simulator. The finger is fully flexed and extended. The PIP and DIP angles are measured by the OptiTrack capture system. There is a noticeable difference in distance, but the overall shape is roughly the same. The simulated IPJ-coupling is more linear, because it is modelled that way. The bending of the PIP results into a linear bending of the DIP. In the physical model it is not entirely linear but more concave.

5.2.5 IPJ-coupling

The IPJ-coupling is an important factor in the physical setup as it is the key element in making the finger as anatomically correct as possible. To compare the accuracy of the IPJ-coupling, the physical finger is fully flexed and fully extended. The OptiTrack capture system measured the angles of the PIP and DIP joint. Fig. 5.6 shows the IPJ-coupling measured by the capture system and the IPJ-coupling of the simulator. There is a noticeable difference in distance, but the overall shape is roughly the same. The simulated IPJ-coupling is more linear, because it is modelled that way 4.20. The bending of the PIP results into a linear bending of the DIP. In the physical model it is not entirely linear but more concave. This is sufficient for the simulator since adding any more complexity will result in performance issues.

	Full grasp	Extended PIP	Extended MCP	Complex
Global friction coefficients	0.1903	0.1858	0.1853	0.1864
After fine-tuning friction coefficients	/	0.0934	0.1021	/

Table 5.1: Overview of the root mean squared error (RMSE) values between the trajectories done on the physical setup and the trajectory produced by passing the measured forces into the simulator. Changing the friction coefficients results in a much lower loss and thus a better resemblance of the reference. There is however no global configuration of friction coefficients to result in a RMSE decrease for all of the trajectories.

5.2.6 Discussion

It is not straightforward to design a simulator that accurately reproduces any arbitrary trajectory done by the physical setup. Most of the time, it can be tweaked to resemble it by changing the friction coefficients. However, these are not globally suited for every possible trajectory, but they do show that the simulation can approximately replicate the physical setup with the provided forces. Table 5.1 shows the root mean squared error (RMSE) of the extended PIP and extended MCP trajectory before and after fine-tuning. The RMSE is a value indicating the difference between two trajectories. It should be as low as possible.

More accurate readings and consideration of other factors mentioned in section 5.2 will definitely improve the simulation accuracy. These were not added to the model due to a strict time frame on this thesis and due to the possibility of creating extra performance impacts on the training of the self-learning trajectories which are discussed in the Chapter 7. A strategy to possibly increase the accuracy by optimising the immeasurable static parameters like the friction coefficients and the center of gravity is discussed in the next chapter.

Chapter 6

Using differentiable programming to optimise the accuracy of the simulator

As mentioned in Chapter 5, the simulation did not exactly match the physical setup which is due to the discussed unconsidered external factors. However, for some trajectories, the trajectory errors could be minimised by manually fine-tuning the friction coefficients. There is a much more efficient way of doing this with what is called differentiable programming.

6.1 Differentiable programming

Differentiable programming is a method to achieve a certain optimum by using differential optimisation algorithms like gradient descent. It requires that the applied model is differentiable, which is the case for the dynamic model of the finger.

6.1.1 Gradient descent

Gradient descent is a parameter optimisation technique used in many machine learning related projects.

Loss function

The first step is to define a loss function that describes the relation between the pattern to be approximated (called the ground truth) and the approximated pattern.

For example, if the goal is to minimise the difference between a reference and approximated trajectory, the root mean square error (RMSE) loss function could be used:

$$RMSE = \sqrt{\frac{1}{n}\Sigma_{i=1}^{n}(T_1 - T_2)^2}, \tag{6.1}$$

where n is the number of samples, T_1, T_2 respectively the reference and approximated trajectory.

Initial value

The next step is to define an initial value for the parameters that require optimisation. This can be an educated guess or a random initialisation.

Deepest descent

Lastly a step is taken into the deepest descent for each parameter by computing the gradient. The fraction of the step is called the learning rate. This goes on for a certain number of n iterations until a desired minimum is reached.

There are several pitfalls with this method like local minima. These can be prevented by picking better initial values for the parameters or using an optimized dynamic learning rate.

6.1.2 Choice of automatic differentiation computer library

Gradient descent requires the calculation of the gradients, which is a numerically intensive process. There are several Python packages that provide efficient methods for calculating gradients like Autograd [1]. Until recently it used to be the fastest implementation available. Since 2019 it was superseded by JAX [3]. JAX combines Accelerated Linear Algebra (XLA) [25] and Autograd. XLA is a domain-specific compiler for linear algebra that can accelerate TensorFlow models [25]. When a TensorFlow program is run, all of the operations are executed individually by the TensorFlow executor. XLA provides an alternative mode of running models: it compiles the TensorFlow graph into a sequence of computation kernels generated specifically for the given model. Because these kernels are unique to the model, they can exploit model-specific information for optimisation, resulting in a significant increase in execution speed.

Due to its high efficiency and execution speed, JAX is the preferred method for calculating gradients. It also allows the use of Just-In-Time compilation (JIT) and a Graphics Processing Unit (GPU). The GPU functionality is not used as the optimisation algorithm does not benefit from the parallelisation the GPU offers. The first time a JAX function is executed, it has to be compiled. This can take some time, but the overall achieved speedup for the next iterations compensates for this.

Since JAX requires purely functional code, the object-oriented nature of Python can not be used. The Python implementation and its pitfalls are discussed in Appendix B.

6.2 Optimisation strategy

Due to the limited time frame on this thesis, the strategy for optimising the accuracy of the model is only discussed and not actually implemented.

The dynamic model has many static parameters, most are perfectly measurable, like the length of each phalanx. Other parameters like the friction coefficients, moment arms, center of gravity, phalanx inertias, IPJ-coupling fraction etc. are all much harder to determine as they are not perfectly measurable. Differentiable programming is an efficient way to optimise these static variables.

6.2.1 Training data

The first step is to define the training data, which consists of many trajectories that have been executed on the physical setup along with their measured forces. The trajectories discussed in Chapter

5 are a good starting point, but more complex trajectories are required in order to find an acceptable optimum.

6.2.2 Loss function

The next step is to define the loss function. There are many different loss functions with comparable performance. The Root Mean Square Error (RMSE) is a good choice. The more important question is on which data of the trajectories this loss function has to be applied.

This depends on the desired result. If the fingertip trajectory is the most important, the coordinates of the end-effector should be used. If the overall movement of each phalanx is more important, the angles and velocities of each phalanx should be used. Since overall accuracy is preferred, the latter is chosen. It is defined as:

$$f_{\text{loss}}(A_{\text{ref}}, P) = \sqrt{\frac{1}{n}\Sigma_{i=1}^{n}\Big(A_{\text{ref}} - \text{sim}(D, F, P)\Big)^2}, \tag{6.2}$$

where A_{ref} is the angle and velocity matrix of the reference trajectory and sim the simulation function which takes in the dynamic parameters D, the acting forces F and a vector P containing the parameters to be optimised. sim returns a matrix containing the angles and velocities of the approximated trajectory.

6.2.3 Gradient descent

With the loss function and training data, gradient descent can now be performed. With the help of JAX, the gradient of the loss function is calculated. Each parameter to be optimised is subtracted by its partial derivative found in the gradient:

$$P_{i+1} = P_i - r \cdot \nabla(f_{\text{loss}}, P_i), \tag{6.3}$$

where r is the learning rate, P_i the parameter to be optimised and P_{i+1} the optimised parameter. This goes on for n number of steps until a desired value for the loss function has been achieved.

6.2.4 Optimisation diagram

Fig. 6.1 shows a visualisation of the optimisation algorithm to improve the accuracy of the simulator. In the first iteration the initial static parameters to be optimised are passed into the simulation along with the other static variables and the dynamic variables. The simulator outputs an approximate trajectory, which is passed into the loss function along with the reference trajectory (training data). The gradient is taken from the output of the loss function. The partial derivatives in the gradient are now subtracted from the initial parameters (p_{init}) to find the more optimal parameters (p_{opt}). This goes on for n iterations where for each iteration, the newly found parameters (p_{opt}) are used for the next iteration.

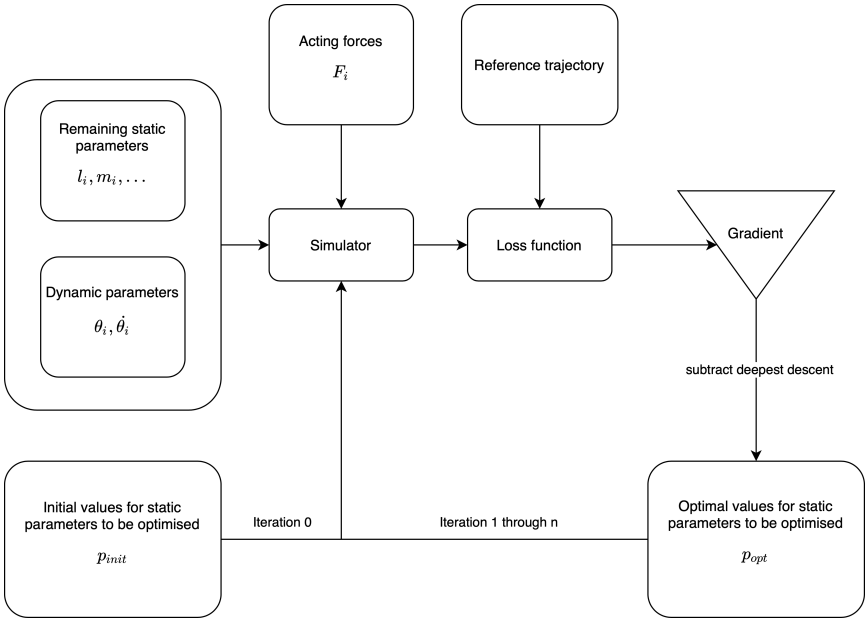


Figure 6.1: Diagram of the optimisation algorithm to improve the accuracy of the simulator. In the first iteration, the initial static parameters to be optimised are passed into the simulation along with the other static variables and the dynamic variables. The simulator outputs an approximate trajectory, which is passed into the loss function along with the reference trajectory (training data). The gradient is taken from the output of the loss function. The values in the gradient are now subtracted from the initial parameters (p_{init}) to find the more optimal parameters (p_{opt}). This goes on for n iterations where for each iteration, the newly found parameters (p_{opt}) are used for the next iteration.

6.2.5 Discussion

The necessary learning rate and number of steps is hard to predict without implementing the actual strategy. The question remains if the algorithm will converge into a minimum and be able to provide parameters to have a more accurate dynamic model. If not, the discussed external factors in Chapter 5 will have to be considered in order to increase the accuracy. This is seen as future work.

Chapter 7

Using differentiable programming to reproduce trajectories

The most interesting use case for the simulator is that it allows the use of differentiable programming to efficiently reach the optimal forces to reproduce a desired trajectory.

To refer to the piano playing problem mentioned in Chapter 1. Say a musician would like to know if he could play the piano again. A trajectory representing a keystroke could be fed to the simulator with the individual tendon coupling of the patient. The simulator can then decide if the movement is possible. If so, which tendons are required and what are their applied forces?

To optimise the necessary forces, the same technique discussed in Chapter 6 (gradient descent) is used.

7.1 Advanced Central Pattern Generator

As mentioned in Chapter 3, a more advanced oscillator than the Hopf-based CPG [9] is necessary to allow more complex trajectories. Recent work on robotic locomotion has showed that recurrent neural networks (RNNs) can behave as CPGs [7] [23]. General neural networks are not suited since they do not offer support for internal states (memory).

7.1.1 Continuous Time Recurrent Neural Network (CTRNN)

There are many different kinds of RNNs with their own purpose. The most important factor is that it should be capable of producing output along a temporal sequence and can have temporal dynamic behaviour to mitigate sudden force changes. A Continuous Time Recurrent Neural Network (CTRNN) meets these criteria. To have an output for each of the four muscles, four neurons are needed.

A CTRNN is given by [6]:

$$\tau_i \dot{y_i} = -y_i + (\sum_{j=1}^{N} w_{ji} \sigma(y_j + \theta_j)) + I_i \tag{7.1}$$

where y is the state of each neuron, τ is a time constant, w is the weight of an incoming connection, σ is the sigmoid activation function, θ is the bias term or firing threshold of the node and I is an external input which is not used since no external input is registered.

The 4-neuron-CTRNN can thus be represented by four 4-size vectors comprising of its current state, biases, gains, time constants and a 4x4 matrix comprised of the weights.

7.2 CTRNN parameter optimisation

The purpose is to train the parameters defining the CTRNN to find the necessary forces to reproduce a certain reference trajectory. The parameters of the CTRNN to optimise are its state, biases, gains, time constants and a matrix comprised of its weights.

To have a clear overview on the process, Fig. 7.1 illustrates the concept. Notice the similarities with the diagram of accuracy optimisation strategy discussed in Chapter 6.

The first step is to define the initial parameters of the CTRNN, this can be all zeros, all ones, a random initialisation or an educated guess.

Next the initial values are passed to the CTRNN which runs over a certain time interval to produce the acting forces on the tendons. The result is a matrix containing the four forces for each tendons with respect to time. These forces are then passed to the simulator along with the static and dynamic parameters defining the model. The output is a matrix containing the angles and the angle velocities for each joint with respect to time.

The approximated and reference trajectory are then passed into the loss function which returns a value indicating the similarity between them. It should be as low as possible.

The gradient is taken, and for each of the parameters its value is updated by taking a step in the steepest descent. In other words, a fraction of their derivatives are subtracted from their current value to find a more optimal value resulting in a smaller loss. These updated values are now the new parameters for the CTRNN and the whole process starts over.

This goes on until n iterations have been completed or the desired minimal loss is achieved.

7.2.1 Loss functions

The choice of loss function is crucial in order to easily find the minimum. It should be as smooth as possible to avoid any performance issues. The loss function can be defined on the coordinates of the end-effector, the coordinates of each phalanx, their accelerations, the joints angles and so forth.

The loss function defined on the joint angles and their accelerations is a good measure as it contains all the necessary information. All other information, such as the coordinates of each phalanx, can be derived from the angles and their velocities, which means they fully describe the movement. It is chosen to be the default loss function unless mentioned otherwise.

Another used loss function is based on the end-effector coordinates. This means the phalanx movements are not defined and allow multiple ways to reach a same goal. It also simplifies the creation of the reference trajectory as no angles are needed, just the actual end-effector trajectory.

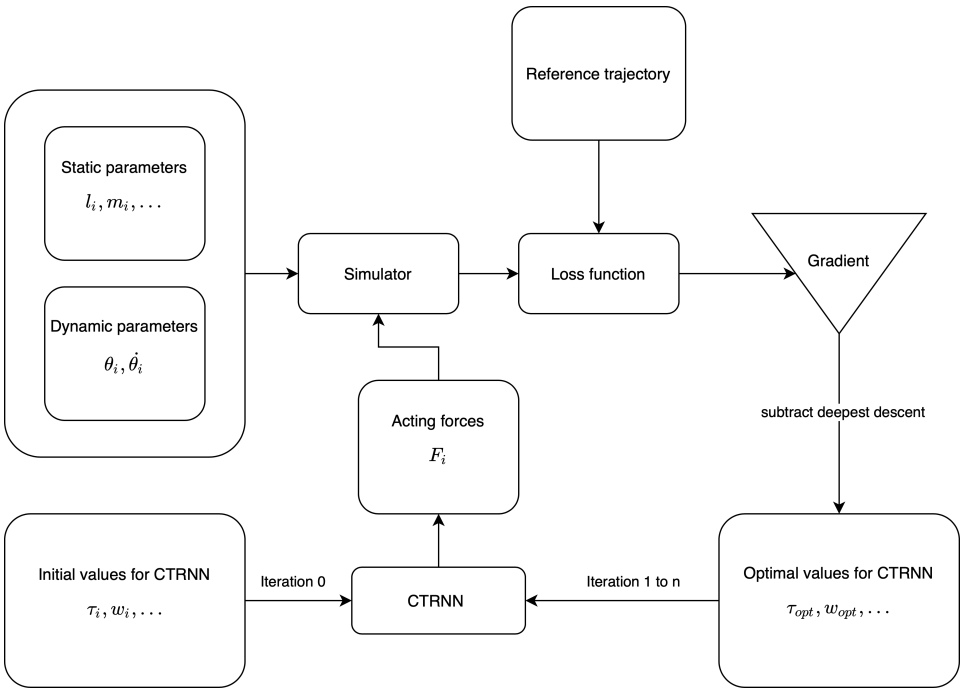


Figure 7.1: Diagram showing the optimisation algorithm to train the CTRNN parameters. The purpose of training these parameters is to let the CTRNN generate the necessary forces to reproduce a certain reference trajectory. The initial values for the CTRNN parameters are chosen and are passed to the CTRNN, which produces a matrix of approximated forces. These force are passed to the simulator along with the static and dynamic variables defining the model. The simulator outputs an approximated trajectory which is passed into the loss function along with the reference trajectory to return a number indicating the similarity between both trajectories. The gradient is taken, and each CTRNN parameter is updated by subtracting a fraction of its corresponding derivative. These updated are then fed into the CTRNN again to begin the whole process again. This is repeated for n iterations or until a desired loss is achieved.

7.3 Optimisation of the computational performance

As mentioned in subsection 6.1.2, a JAX function has to compile the first time it is executed. After this compilation, the next calls of the same function do not have to be compiled anymore and benefit from a significant speedup. The addition of tendons and ligaments has a heavy impact on the computational performance of this compilation. To illustrate this, three models are considered:

- Model A: dynamic model with tendons and with ligaments
- Model B: dynamic model with tendons, but without ligaments
- Model C: dynamic model without tendons and without ligaments

To clarify, model C only takes in the torques for each joint instead of the four forces of each muscle. Table 7.1 shows the comparison of the different models trained on the same trajectory. The differences are significant and show that the tendons and mostly the ligaments have a huge impact on the performance. Model B shows at least a double amount of time necessary. Model C is considered non-compilable as the compilation was kept on for 96 hours with no result. Replacing the Heaviside function with a sigmoid function as mentioned in Chapter 4 does not result in a compilable model C. Significant optimisation is required in order to be make it usable.

Model	Compilation time	Time per iteration
A	1 minute	30 seconds
B	2 to 10 minutes	1 to 5 minutes
C	Unknown	Unknown

Table 7.1: Overview of the average compilation times and time per iteration for each model (A, B and C). Model A with no tendons and ligaments is by far the fastest. Adding tendons (model B) results in at least a double amount of time needed. Adding ligaments (model C) results in a non-compilable model. The compilation was kept on for 96 hours with no results.

The first step is to numerically optimise the gradient descent algorithm to reduce the number of steps needed for the loss to converge. SciPy [22] has a function called *minimise*, which minimises a certain objective with a chosen algorithm like gradient descent. Using this optimisation also showed a significant reduction in the time needed per iteration for model A en B. Table 7.2 shows the time needed per iteration. However, model C still remains non-compilable.

Model	Time per iteration
A	10 seconds
B	30 seconds
C	Unknown

Table 7.2: Overview of the average time per iteration for each model (A, B and C). Model A with no tendons and ligaments is again by far the fastest, the *minimise* function significantly sped up the time needed per iteration from 30 seconds to 10 seconds. Adding tendons (model B) results again in at least a double amount of time needed, but it also shows a significant gain in speed as it went from 1 to 5 minutes to 30 seconds. Adding ligaments (model C) again results in a non-compilable model.

More time was invested in the optimisation of model C as this is the most interesting model. Since the problem undoubtedly lies with the ligaments, other approximations of the Heaviside function have

been tested:

$$\text{Heaviside}(x) \approx \frac{1}{2} + \frac{1}{2}\tanh(kx), \tag{7.2}$$

where k indicates the sharpness of the transition. Another approximation:

$$\text{Heaviside}(x) \approx \frac{x + |x|}{2x} \tag{7.3}$$

None of these approximations enabled the compilation of model A. So the problem could only lie in the representation of the ligaments. As mentioned in Chapter 4, they are modeled as a damping that activates once a certain joint angle boundary is exceeded (Eq. 4.13). Another option to model ligaments is to use a counter-torque. The idea is to apply a slightly higher torque in the opposite direction of the current joint which is about to exceed its angle limit. The improved ligament function then becomes:

$$\text{ligament}(\alpha_i) = \begin{cases} -\tau_i c_l & \text{if } \alpha_i \leq \alpha_{i_{\text{min}}} \wedge 0 < \tau_i, \\ -\tau_i c_l & \text{if } \alpha_{i_{\text{max}}} \leq \alpha_i \wedge \tau_i < 0, \\ 0 & \text{otherwise.} \end{cases} \tag{7.4}$$

where c_l is a constant indicating the magnitude of the counter-torque. Just as Eq. 4.13, it can also be transformed into a one-line function:

$$\text{ligament}(\alpha_i) = -\sigma(k, \alpha_{i_{\text{min}}} - \alpha_i) \cdot c_l \cdot \tau_i - \sigma(k, \alpha_i - \alpha_{i_{\text{max}}}) \cdot c_l \cdot \tau_i \tag{7.5}$$

With the implementation of the counter-torque instead of the damper, the program now compiles smoothly with almost equal performance to model B, indicating that the ligaments do not cause a performance impact.

7.4 Training unloaded trajectories on the simulator

With the significantly optimised dynamic model, the actual performance of the CTRNN and the optimisation algorithm can be discussed.

7.4.1 Strategy

Several simple and complex trajectories are defined and used as reference trajectories. Most of the time, two loss functions are tested: the loss function based on the coordinates of the end-effector and the loss function based on the angles and angle velocities of the joints.

To clarify, the only information the optimisation algorithm receives is either the end-effector coordinates or angle and angle velocities (depending on the loss function) and the time span. No other data is provided, the purpose of the algorithm is to determine the necessary forces to reproduce the given reference trajectory.

The CTRNN has full control over the four muscles, even though the FS proved to be redundant in unloaded control by Leijnse et al. If the CTRNN decides to use it, it proves that the FS can also be used to reproduce unloaded trajectories. If not, it shows the redundancy in the unloaded control for the respective trajectories.

For simplicity and lack of repetitiveness the optimisation done with angles/angle velocities loss function will be called $\text{opt}_{\text{angles}}$ and the optimisation done with the end-effector loss function $\text{opt}_{\text{end-eff}}$.

The performance is tested on:

- A full grasp trajectory
- A sinusoidal trajectory
- A perfect circle trajectory
- A trajectory with sudden force changes

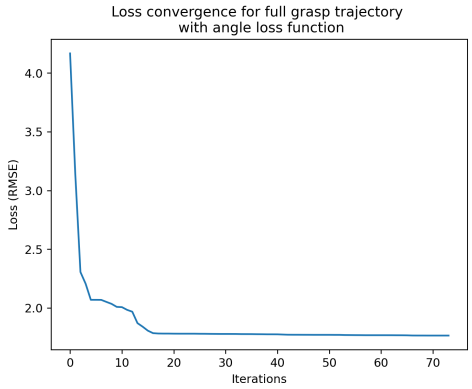
7.4.2 Full grasp trajectory

The first trajectory is a full grasp, which is a full flexion of the MCP, PIP and DIP joint. Fig. 7.3a shows the reference trajectory of the end-effector to be approximated. Fig. 7.2a and Fig. 7.2b respectively show the loss convergence of the $\text{opt}_{\text{angles}}$ and $\text{opt}_{\text{end-eff}}$. Fig. 7.3b and Fig. 7.3c respectively show the convergence of the approximated end-effector trajectory $\text{opt}_{\text{angles}}$ and $\text{opt}_{\text{end-eff}}$. These figures show a gradient bar on the right from white to black. Each end-effector trajectory for a certain iteration has a specific color in this gradient, e.g. the first iteration is almost white and the last iteration is almost black. It illustrates the way the optimisation approaches the reference.

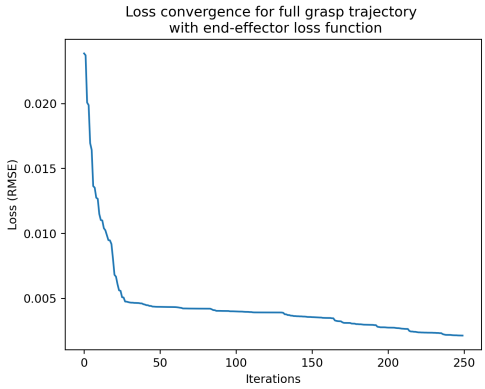
$\text{opt}_{\text{angles}}$ never fully covers the reference, unlike $\text{opt}_{\text{end-eff}}$, which almost fully covers the reference after 25 iterations with a low loss of 0.005. $\text{opt}_{\text{angles}}$ stops after 74 iterations due to precision errors notified by the SciPy minimise function.

Fig. 7.4a shows the ideal forces for each muscle to reproduce the reference trajectory. Only the FP is used, since this is the only muscle necessary for a full flexion. Fig. 7.4b and Fig. 7.4c respectively show the approximated forces generated by the optimisation algorithm for $\text{opt}_{\text{angles}}$ and $\text{opt}_{\text{end-eff}}$. These are different, which shows that there are multiple force configurations in order to reproduce the full grasp trajectory.

The benchmarks for the algorithm's performance are shown in Table 7.3. The angle/velocity loss functions takes slightly longer than the end-effector loss function.

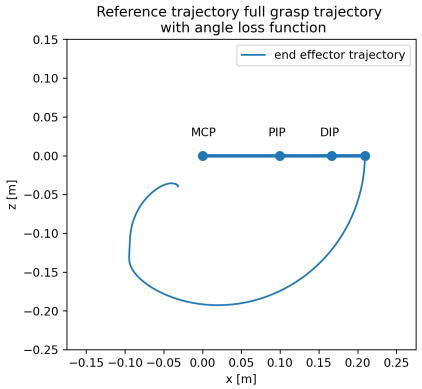


(a) Loss convergence for the full grasp trajectory for the angle/angle velocities loss function. The first three iterations are the most crucial as they result in a halving of the loss. After 15 iterations, the loss fully converges. After 74 iterations, precision errors start to occur. More iterations did not result in a lower loss.

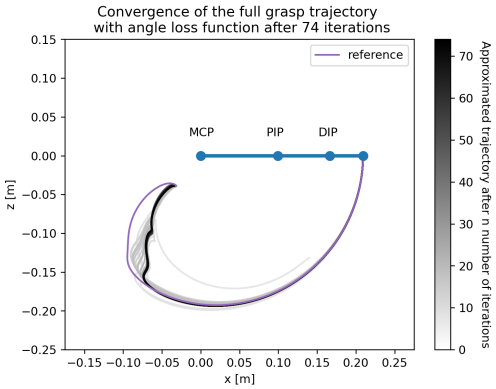


(b) Loss convergence for the full grasp trajectory for the end-effector loss function. The first 25 iterations are crucial as they keep resulting in a steep descent. Afterwards, the loss starts converging. No precision errors occured.

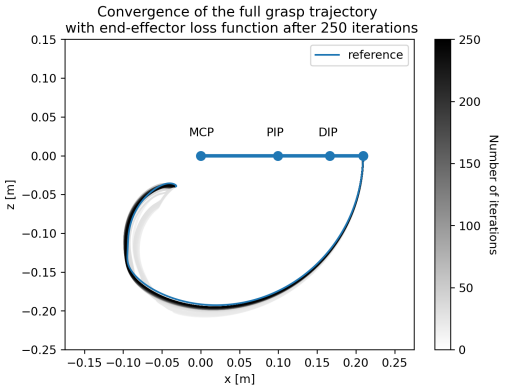
Figure 7.2: Overview of the loss convergence of the full grasp trajectory approximation with angles/angle velocity and end-effector loss functions.



(a) Reference full grasp trajectory to be approximated by the optimisation algorithm.

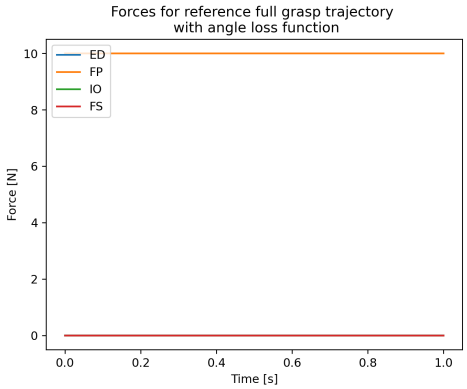


(b) Approximated full grasp trajectory convergence for the end-effector loss function. The approximated trajectory never fully resembles the reference. The optimisation stopped after 74 iterations due to precision errors. This figure shows a gradient bar on the right from white to black. Each end-effector trajectory for a certain iteration has a specific color in this gradient, e.g. the first iteration is almost white and the last iteration is almost black. It illustrates the way the optimisation approaches the reference.

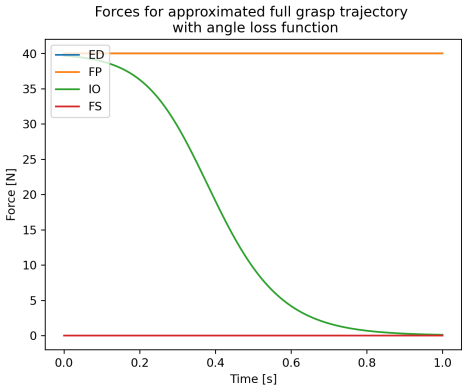


(c) Approximated full grasp trajectory convergence for the end-effector loss function. After 25 iterations, the approximated trajectory almost perfectly resembles the reference.

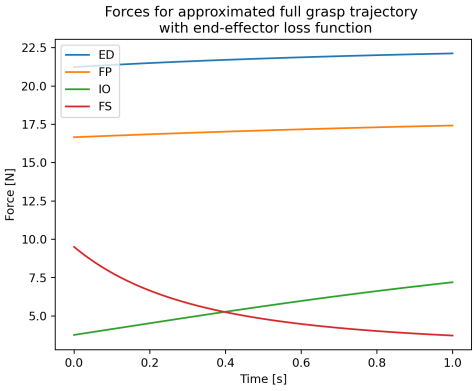
Figure 7.3: Overview of the trajectory convergence of the full grasp trajectory approximation with angles/angle velocity and end-effector loss functions.



(a) Ideal forces for the full grasp trajectory. Only the FP is used since this is the only muscle necessary for a full grasp.



(b) Approximated forces generated by the optimisation algorithm for the full grasp trajectory. The optimisation with the angles/angle velocities loss function abruptly stopped due to precision loss errors, so these forces do not fully reproduce the reference. The force applied by the FP is far too high compared to the ideal force of the FP, it tries to compensate by involving the IO, but this results in a deviant trajectory.



(c) Approximated forces generated by the optimisation algorithm for the full grasp trajectory. The reference was already closely reproduced after 25 iterations. However, mostly the ED and the FP are used to reproduce instead of only FP as in the ideal case. The force applied by the FP is too high, thus force is applied on the ED to compensate this. This is an example of another set of possible forces to reproduce the same trajectory.

Figure 7.4: Overview of the approximated forces for the full grasp trajectory approximation with angles/angle velocity and end-effector loss functions.

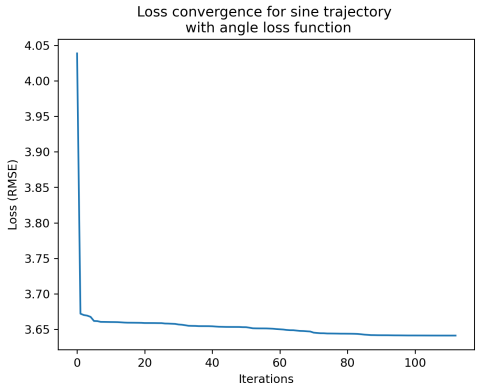
7.4.3 Sine trajectory

The next trajectory is to test the CTRNN's performance on reproducing oscillations. A sinusoidal pattern is applied on each of the muscles, resulting in a oscillating end-effector trajectory as shown in Fig. 7.6a. Fig. 7.5 shows the loss convergence of both $\text{opt}_{\text{angles}}$ and $\text{opt}_{\text{end-eff}}$. Fig. 7.6b and Fig. 7.6c respectively show the convergence of the approximated end-effector trajectory $\text{opt}_{\text{angles}}$ and $\text{opt}_{\text{end-eff}}$.

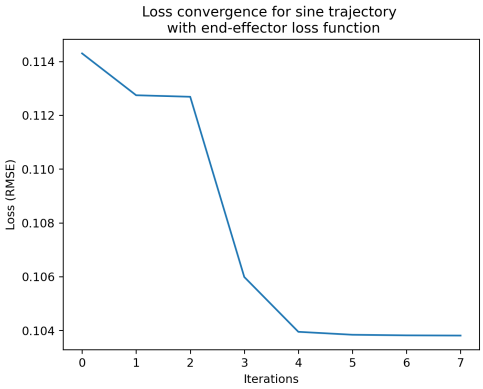
Both loss functions resulted in precision errors. Lowering the sampling rate did not have any effect. For this reason, another loss function defined on all the phalanx positions was used, but it showed the same result.

Fig. 7.7a shows the ideal forces to reproduce the reference, it shows three oscillations on the ED, FP, and IO. As can be seen in Fig. 7.7b and Fig. 7.7c, the CTRNN is incapable of producing oscillations for this trajectory.

The benchmarks for the algorithm's performance are shown in Table 7.3. The angles/angle velocity loss function is significantly faster than the end-effector loss function.

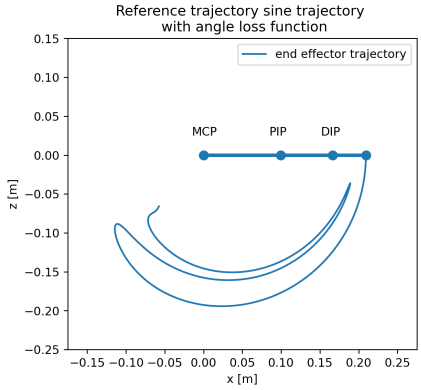


(a) Loss convergence for the sine trajectory for the angle/angle velocities loss function. The first two iterations result in a drop from 4.05 to 3.66. Further iterations show an immediate convergence of the loss. After 114 iterations, precision errors start to occur. More iterations did not result in a lower loss.

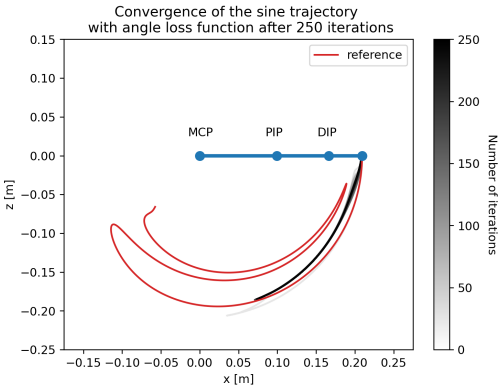


(b) Loss convergence for the sine trajectory for the end-effector loss function. The first 4 iterations result in a drop from 0.114 to 0.104. Further iterations show an immediate convergence of the loss and precision errors start to occur. More iterations did also not result in a lower loss.

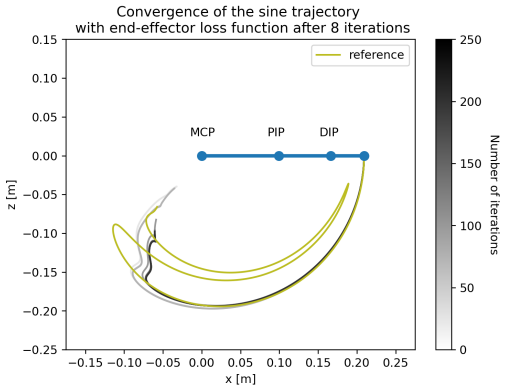
Figure 7.5: Overview of the loss convergence of the sine trajectory approximation with angles/angle velocity and end-effector loss functions.



(a) Reference sine trajectory to be approximated by the optimisation algorithm.

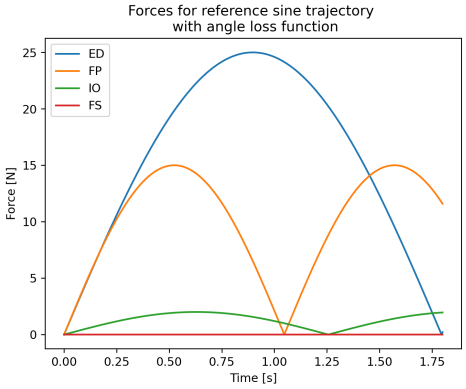


(b) Approximated sine trajectory convergence for the end-effector loss function. The approximated trajectory never fully resembles the reference. The optimisation stopped after 114 iterations due to precision errors. The finger flexes with an extended PIP and DIP and suddenly stops. There is no oscillatory behaviour. This figure shows a gradient bar on the right from white to black. Each end-effector trajectory for a certain iteration has a specific color in this gradient, e.g. the first iteration is almost white and the last iteration is almost black. It illustrates the way the optimisation approaches the reference.

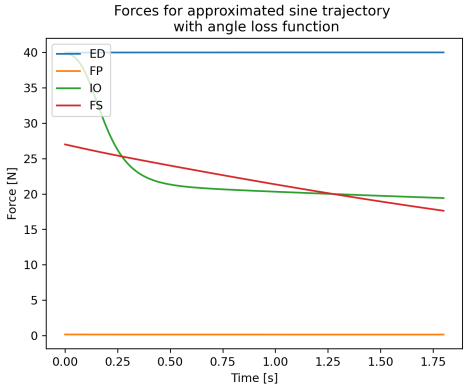


(c) Approximated sine trajectory convergence for the end-effector loss function. The approximated trajectory never fully covers the reference. The optimisation stopped after only 8 iterations due to precision errors. The finger flexes with an extended PIP and DIP and does a full swing, after which a very small oscillatory behaviour occurs.

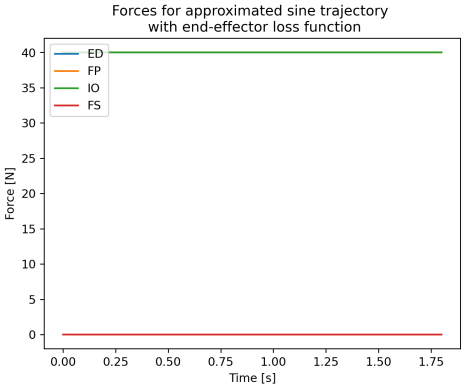
Figure 7.6: Overview of the trajectory convergence of the sine trajectory approximation with angles/angle velocity and end-effector loss functions.



(a) Ideal forces for the sine trajectory. Each muscle has a different phase and amplitude.



(b) Approximated forces generated by the optimisation algorithm for the sine trajectory with the angles/angle velocites loss function.



(c) Approximated forces generated by the optimisation algorithm for the sine trajectory with the end-effector loss function.

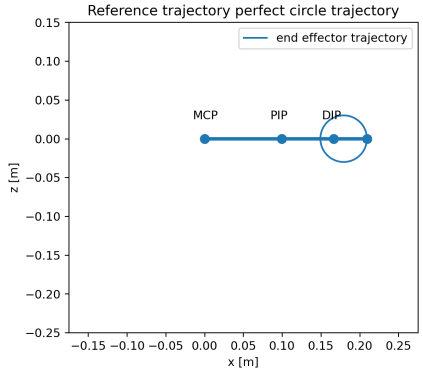
Figure 7.7: Overview of the approximated forces for the sine trajectory approximation with angles/angle velocity and end-effector loss functions. The approximated forces generated by the optimisation algorithm for the sine trajectory for both loss functions had precision errors. Neither of these force configurations reproduce the reference as they do not show the necessary oscillatory behaviour. This indicates that the CTRNN might not be capable of oscillations.

7.4.4 Perfect circle trajectory

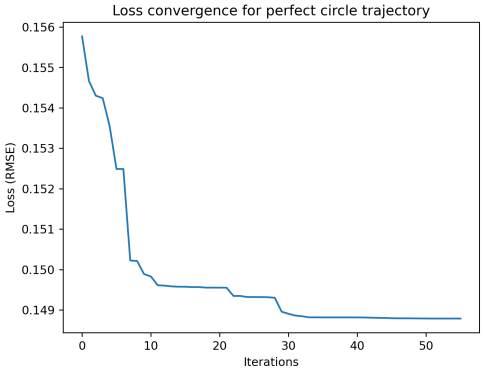
To test the CTRNN's oscillatory performance even more, a perfect circle trajectory is used as the reference. This trajectory can only be applied on the end-effector loss function since the angle/angle velocities for a perfect circle trajectory are hard to define by hand. Fig. 7.8a shows the reference trajectory, Fig. 7.8c the convergence of the approximated trajectory. Fig. 7.8b shows the convergence of the loss function and Fig. 7.8d the generated forces.

In order to reproduce a perfect circle, oscillatory forces are required. The approximation and the forces generated by the CTRNN shows no oscillatory behaviour. The loss functions also has a very minor descent. This, once again, brings doubt to the oscillatory performance of the CTRNN. It has not shown any oscillations and can thus be declared as not being a sufficient candidate for these types of trajectories.

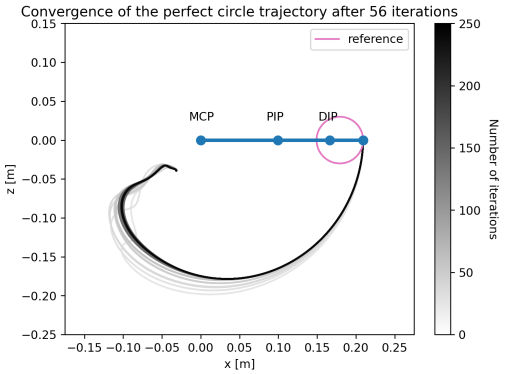
The benchmarks for the algorithm's performance are shown in Table 7.3. Doing an iteration takes much longer than the previously discussed trajectories. It shows that the perfect circle trajectory is very complex and a numerically intensive process.



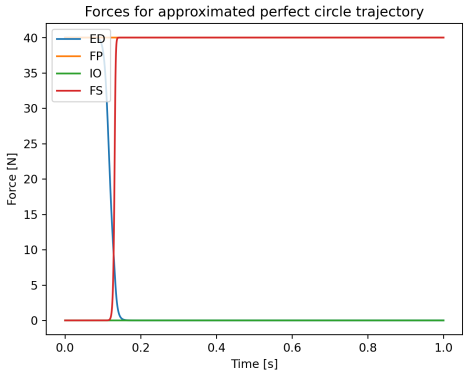
(a) Reference perfect circle trajectory to be approximated by the optimisation algorithm.



(b) Loss convergence for the perfect circle trajectory with the end-effector loss function. The loss converges after 30 iterations. After 56 iterations, precision errors start to occur.



(c) Approximated perfect circle trajectory convergence for the angles/angle velocities loss function. The approximated trajectory never fully covers the reference. The optimisation stops after 56 iterations due to precision errors. The finger does not follow the shape of a circle irregardless of its radius. This figure shows a gradient bar on the right from white to black. Each end-effector trajectory for a certain iteration has a specific color in this gradient, e.g. the first iteration is almost white and the last iteration is almost black. It illustrates the way the optimisation approaches the reference.



(d) Approximated forces generated by the optimisation algorithm for the perfect circle trajectory with the end-effector loss function. In order to reproduce a perfect circle, oscillations are necessary. The CTRNN once again fails to show any oscillatory behaviour.

Figure 7.8: Overview of the trajectory convergence of the perfect circle trajectory approximation with the end-effector loss function.

7.4.5 Sudden force change

The last trajectory is to test the CTRNN on its performance of quickly changing forces. A problem with the Hopf-based oscillator (Chapter 3) is that it could not handle sudden non-periodic force changes. Fig. 7.10a shows the reference trajectory that requires sudden force changes. The finger first flexes with an extended PIP and then suddenly overextends. Fig. 7.9a and Fig. 7.9b show the loss convergence for both loss functions. $\text{opt}_{\text{angles}}$ halts after 161 iterations and $\text{opt}_{\text{end-eff}}$ after 22 due to precision errors.

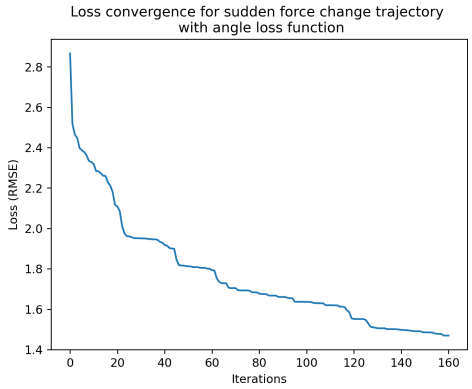
Fig. 7.10b and Fig. 7.10c show the convergence of the end-effector trajectory for each of the loss functions. $\text{opt}_{\text{angles}}$ closely resembles the reference after 60 iterations. The only difference is the flexed angle of the MCP which is slightly smaller than the reference. $\text{opt}_{\text{end-eff}}$ does not resemble the reference, it instead only does a full grasp.

Fig. 7.11a shows the ideal forces in order to reproduce the reference. Each muscle has sudden force changes. Fig. 7.11b shows the approximated forces for $\text{opt}_{\text{angles}}$. Surprisingly the ED is kept at a constant force, which is why the FS is actively used to mitigate these changes, even though it is redundant in unloaded control. The IO compensates the sudden force changes by linearly decreasing its applied force. This results in a different configuration of forces for the same end-effector trajectory as the reference.

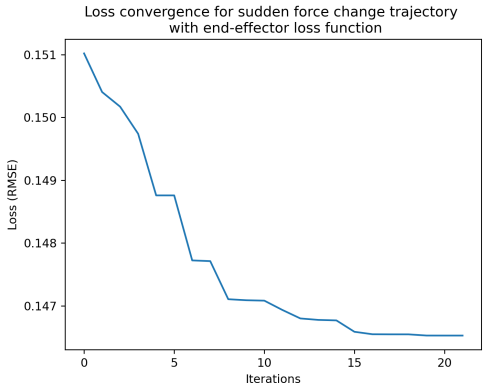
Fig. 7.11c shows the forces generated by $\text{opt}_{\text{end-eff}}$. These are not representative as they do only perform a full grasp. The forces had no chance to converge due to the precision errors at iteration 22.

Table 7.3 shows the benchmarks for this trajectory. The average time per iteration is comparable to the sine trajectory. This is most likely due to the complex nature of these trajectories. The end-effector loss function performs poorly just as with the sine trajectory.

This experiment shows that the CTRNN is indeed capable of handling sudden force changes.

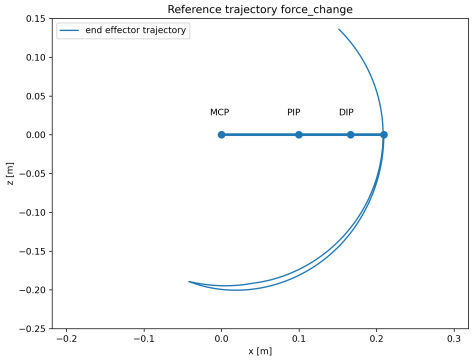


(a) Loss convergence for the sudden force change trajectory for the angle/angle velocities loss function. The first few iterations cause a significant drop, the rest of the iterations decrease the loss with an approximately equal amount. After 60 iterations it starts to converge and after 161 iterations, the loss fully converges and precision errors start to occur. More iterations did not result in a lower loss.

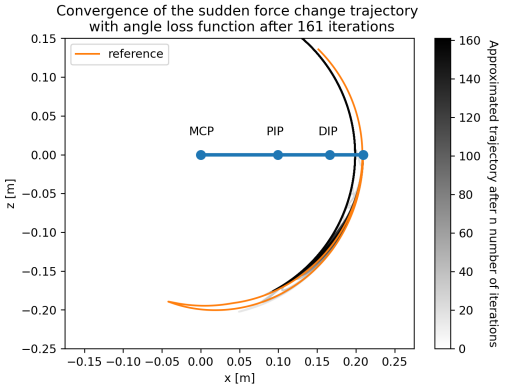


(b) Loss convergence for the sudden force change trajectory for the end-effector loss function. After only 10 iterations the loss starts converging. At 22 iterations precision errors start to occur.

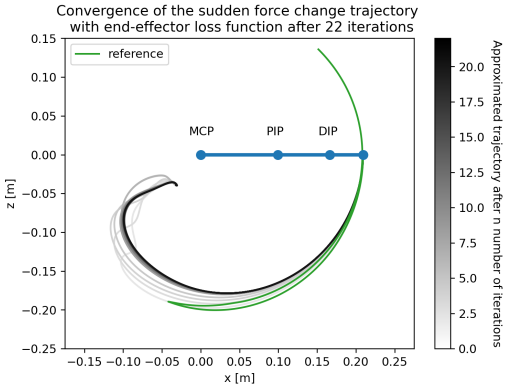
Figure 7.9: Overview of the loss convergence of the sudden force change trajectory approximation with angles/angle velocity and end-effector loss functions.



(a) Reference sudden force change trajectory to be approximated by the optimisation algorithm. The finger flexes with an extended PIP and then suddenly overextends.

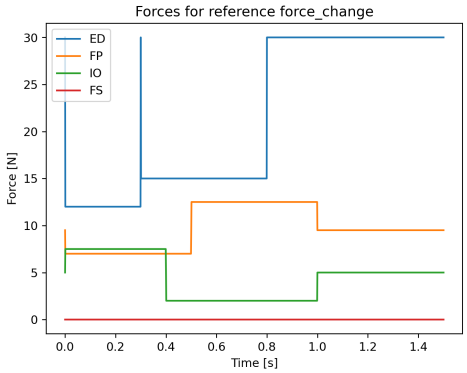


(b) Approximated sudden force change trajectory convergence for the angles/angle velocities loss function. The approximated trajectory greatly resembles the reference after 60 iterations as was expected by inspecting the loss convergence. However, it never fully resembles the reference as the MCP does not bend enough to the dorsal side, but the overal shape is the same. This figure shows a gradient bar on the right from white to black. Each end-effector trajectory for a certain iteration has a specific color in this gradient, e.g. the first iteration is almost white and the last iteration is almost black. It illustrates the way the optimisation approaches the reference.

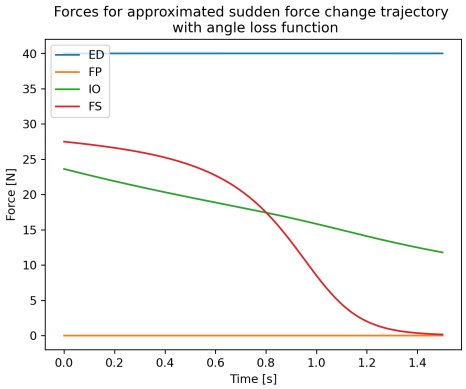


(c) Approximated sudden force change trajectory convergence for the end-effector loss function. The approximated trajectory never even comes close to the reference, instead it does a full grasp. The optimisation stopped after only 22 iterations due to precision errors.

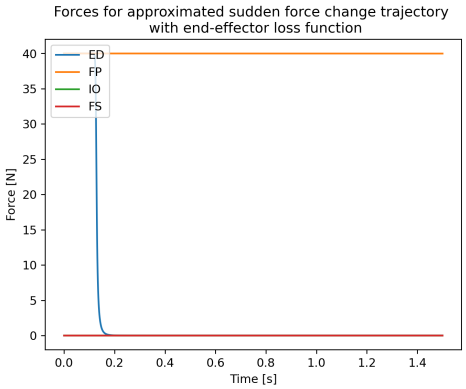
Figure 7.10: Overview of the trajectory convergence of the sudden force change trajectory approximation with angles/angle velocity and end-effector loss functions.



(a) Ideal forces for the sudden force change trajectory. Each muscle has several sudden force changes.



(b) Approximated forces generated by the optimisation algorithm for the sudden force trajectory with the angles/angle velocities loss function. Surprisingly the ED is kept at a constant force, which is why the FS is actively used to mitigate these changes, even though it is redundant in unloaded control. The IO compensates the sudden force changes by linearly decreasing its applied force. This results in a different configuration of forces for the same end-effector trajectory as the reference.



(c) Approximated forces generated by the optimisation algorithm for the sudden force change trajectory with the end-effector loss function. These forces are not representative as the trajectory does not resemble the reference. This is due the precision errors which already occurred at 22 iterations.

Figure 7.11: Overview of the approximated forces for the sine trajectory approximation with angles/angle velocity and end-effector loss functions.

7.4.6 Conclusion

The choice of loss functions has a significant effect on the accuracy of the approximated trajectory. For the sudden force change trajectory, the angles/angle velocity loss function performed much better than the end-effector loss function. But on the other hand, $\text{opt}_{\text{end-eff}}$ was more accurate when reproducing the full grasp. The CTRNN also showed that, most of the time, another configuration of forces exists to reproduce the same end-effector trajectory. More research should be done in order to prevent the precision loss errors. But despite these errors, unloaded control trajectories are certainly possible to be approximated by the CTRNN, as long as there are no oscillations.

The numerical performance for each trajectory differed greatly with the complexity of the reference trajectory. The full grasp is very simple and results in a small amount of time need for each iteration. The perfect circle, sine and sudden force change are much more complex and thus results in a greater amount of time needed for each iteration. Another interesting result is that the more complex a trajectory is, the sooner precision errors start to occur.

Trajectory	# iterations without errors		Average time per iteration		Total time	
Loss function	$\text{opt}_{\text{end-eff}}$	$\text{opt}_{\text{angles}}$	$\text{opt}_{\text{end-eff}}$	$\text{opt}_{\text{angles}}$	$\text{opt}_{\text{end-eff}}$	$\text{opt}_{\text{angles}}$
Full grasp	74	no errors	8.21 s	7.22 s	10 min	30 min
Sine	114	7	22 s	38 s	41 m	5 min
Perfect circle	57	/	4 min	/	3 h 47 min	/
Force change	160	22	16 s	35 s	45 min	13 min

Table 7.3: Overview of the benchmarks for each unloaded trajectory. If the complexity increases, the average time needed per iteration will also increase.

7.5 Training loaded trajectories on the simulator

Since the optimisation algorithm proved to be very promising, a much more interesting and usable kind of trajectories can be explored, e.g. the pressing of a piano key. Stroking a piano key requires force in order to fully press it into its final position. This changes the whole control strategy since the trajectory is now loaded. As mentioned before, the IO proved to be redundant in unloaded finger control [9]. If the IO is active during these loaded trajectories, it would support this theory.

7.5.1 Modeling the piano key

In order to simulate a piano key, it should be added to the dynamic model. A piano key can be modelled as a spring that extends once the key is pressed. According to Hooke's law, the force exerted by a spring is given by:

$$F = c_{key}\Delta z, \tag{7.6}$$

where c_{key} is the spring coefficient of the piano key and Δz the amount of distance the piano key has been pressed.

This spring should only activate once the end-effector reaches its surface. If the key is pressed, a torque is generated on each of the joints according to the length of their connected phalanges, i.e. the torque generated on the MCP joint is equal to the force exerted by the piano key spring multiplied by the sum of the length of each phalanx. More formally, the torque on the MCP joint when hitting the piano key is given by:

$$\tau_{\text{MCP}_{\text{key}}}(x_{\text{e}}, z_{\text{e}}) = \begin{cases} c_{\text{key}}\Delta z(l_{\text{pp}} + l_{\text{mp}} + l_{\text{dp}}) & \text{if } z_{\text{e}} \leq z_{\text{key}} \wedge x_{\text{e}} > x_{\text{key}}, \\ 0 & \text{otherwise.} \end{cases} \tag{7.7}$$

The torque on the PIP:

$$\tau_{PIP_{key}}(x_{\text{e}}, z_{\text{e}}) = \begin{cases} c_{\text{key}}\Delta z(l_{\text{mp}} + l_{\text{dp}}) & \text{if } z_{\text{e}} \leq z_{\text{key}} \wedge x_{\text{e}} > x_{\text{key}}, \\ 0 & \text{otherwise.} \end{cases} \tag{7.8}$$

and the torque on the DIP:

$$\tau_{\text{DIP}_{\text{key}}}(x_{\text{e}}, z_{\text{e}}) = \begin{cases} c_{\text{key}}\Delta z l_{\text{dp}} & \text{if } z_{\text{e}} \leq z_{\text{key}} \wedge x_{\text{e}} > x_{\text{key}}, \\ 0 & \text{otherwise.} \end{cases} \tag{7.9}$$

where $(x_{\text{e}}, z_{\text{e}})$ are the coordinates of the end-effector, c_{key} the spring coefficient of the piano key, $l_{\text{pp}}, l_{\text{mp}}, l_{\text{dp}}$ respectively the length of the proximal, middle and distal phalanx and $(x_{\text{key}}, z_{\text{key}})$ the coordinates of the start point of the piano key.

In this experiment, the piano key is placed at $(x, z) = (0.075, -0.075)$ in meters. Its height and stroke distance is 7 cm. The force exerted by the piano spring is chosen to be 10N when it is fully pressed.

7.5.2 Training

Now that the piano key has been modelled, its effects on the tendons can be studied. The purpose of this experiment is determine which forces are necessary in order to fully press the piano key. To achieve this, a trajectory shown in Fig. 7.12a representing a full stroke of the piano key is used as the reference. This reference was constructed by pressing the piano key without any spring force. This means that this trajectory is not entirely representative on how the finger reacts when pressing a loaded piano key. It is only used to represent the trajectory the finger must do in order to fully press the key.

Only the angles/angle velocities loss function has been used for this experiment. Fig. 7.12b shows the loss convergence, after only 10 iterations the loss already reaches a very low value, indicating that it already closely resembles the reference.

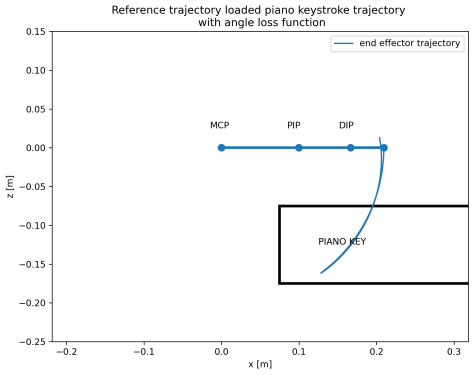
Fig. 7.12c shows the convergence of the approximated trajectory. As was concluded from the loss convergence, the trajectory already shows close resemblance after only 10 iterations. The algorithm continued for 250 iterations without any precision losses.

Fig. 7.12d shows the forces generated by the CTRNN after 250 iterations. The most interesting thing is that the FS is actively used when the piano key is pressed. When it is released, it is not used anymore. This experiment fully supports the theory of Leijnse et al. which stated that the FS muscle is necessary in loaded control.

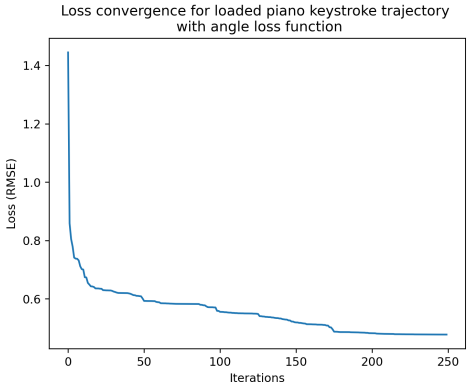
The algorithm needs an average of 26 seconds per iteration. The whole process (250 iterations) took 1 hour and 51 minutes. This is comparable to the other complex unloaded trajectories like the sudden force change, which indicates that loaded control does not have an impact on the performance of the optimisation algorithm.

To illustrate the resemblance between the approximation and the reference, Fig. 7.13 shows snapshots of the animation which can be found in Appendix A. The snapshots show that the approximation almost has the exact same velocity as the reference. The only difference is that the MCP bends over to the dorsal side when pressing the piano key. It does not affect the strike velocity of the piano key, as both end-effector trajectories and are almost equal.

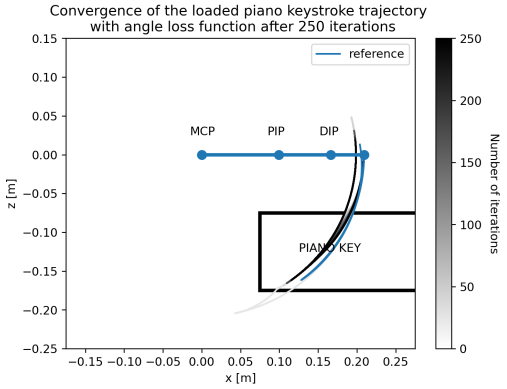
This is a very promising result as the CTRNN proves to be capable of generating the necessary forces to do a loaded trajectory with an unloaded trajectory as a reference.



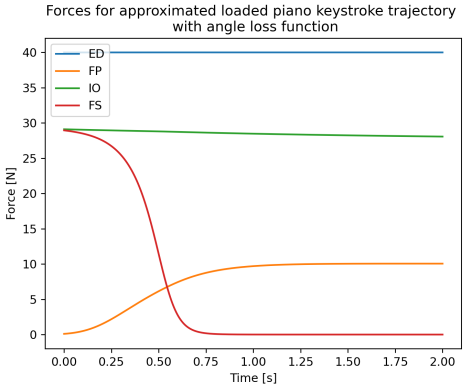
(a) Reference piano key stroke trajectory to be approximated by the optimisation algorithm. The finger fully presses the piano key and then returns to its original position.



(b) Loss convergence for the key stroke trajectory with the angle/angle velocities loss function. After 10 iterations the loss decrease rate drops significantly at an already low loss. The other iterations add small decreases to the loss. After 175 iterations, the loss almost fully converges, but no precision errors occurred.

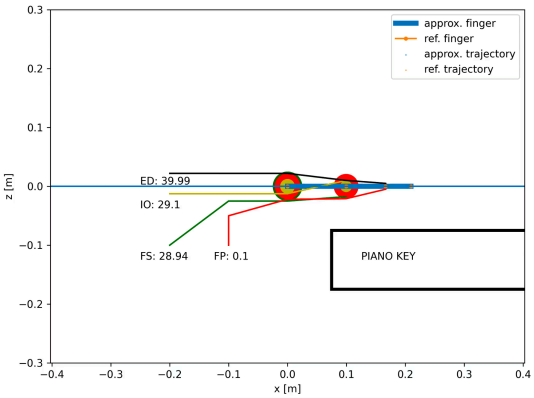


(c) Approximated piano key stroke trajectory convergence for the angles/angle velocities loss function. The approximated trajectory greatly resembles the reference after only 10 iterations as was expected by inspecting the loss convergence. However, it never perfectly resembles the reference as the MCP bends over to the dorsal side when the piano key is pressed. But the overal shape is the same. This figure shows a gradient bar on the right from white to black. Each end-effector trajectory for a certain iteration has a specific color in this gradient, e.g. the first iteration is almost white and the last iteration is almost black. It illustrates the way the optimisation approaches the reference.

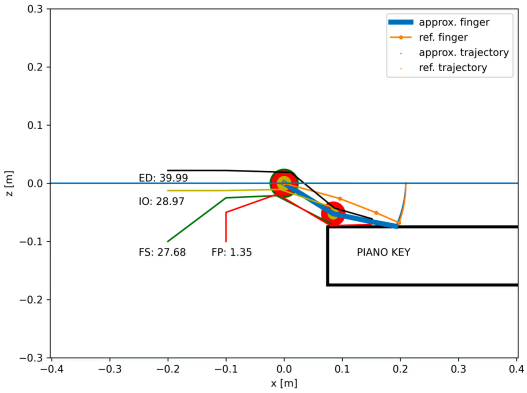


(d) Approximated forces for the piano key stroke trajectory convergence for the angle/angle velocities loss function. The most interesting thing is that the FS is actively used when the piano key is pressed. When it is released, it is not used anymore. This experiment fully supports the theory of Leijnse et al. which stated that the FS muscle is necessary in loaded control.

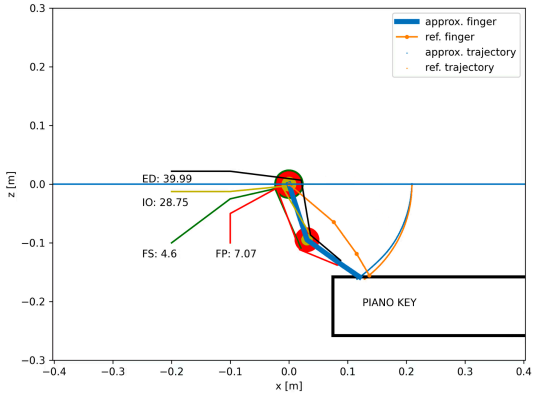
Figure 7.12: Overview of the loaded piano key trajectory.



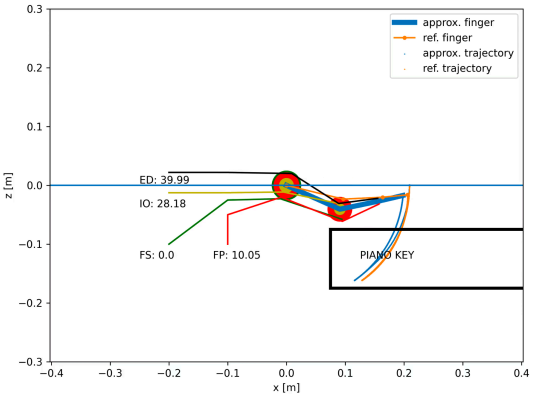
(a) Both fingers start in the same position.



(b) Both fingers bend until they touch the piano key. The PIP joint of approximated finger already starts bending to the dorsal side due to the force applied by the piano key. Both end-effectors are almost in the same position, indicating that they have the same velocity.



(c) Both fingers fully press the piano key. The PIP joint of the approximated finger is still bending to the dorsal side. Both end-effectors are again in almost the same position.



(d) Both fingers return to their original position. The PIP joint of the approximated finger is still bending to the dorsal side. Which means there is still room for improvement. But the overall end-effector coordinates show the same shape and velocities which is the desired result.

Figure 7.13: Snapshots of an animation illustrating the comparison between the reference and approximated trajectory fully pressing a loaded piano key. The approximated finger is displayed in the blue with the visualisation of the moment arms and tendons. The reference finger is displayed in orange.

Chapter 8

Conclusion

The goal of this thesis was to create an accurate simulator of a human finger and explore its capabilities of reproducing unloaded and loaded reference trajectories.

The previous research done on the physical setup highlighted ”a mixed position and force control strategy” and a ”force controlled CMA-ES based optimisation strategy”. The latter did not prove to be very promising as it was incapable of learning which forces have to be applied to the tendons in order to reproduce the complex trajectories. Only two out of the four muscles were actuated and the used force central pattern generator (CPG) showed no sign of dynamic force flexibility. It raised the question on how a purely force control strategy would perform when using a more advanced CPG.

For these reasons a Continuous Time Recurrent Neural Network (CTRNN) was chosen as a more advanced CPG. The CTRNN consists of 4 neurons, each of which is assigned to a muscle. Each neuron generates an output (the force on its tendon) based on certain parameters describing its state. In order to train this neural network to reproduce the forces necessary for reproducing a certain trajectory, an optimisation algorithm is required to find these specific parameters. But in order to do that, a model describing the energies and forces of the physical setup model has to be defined. This is exactly what a dynamic model is. It was constructed with the help of Lagrangian mechanics by describing the system by its dynamic (angles, velocities etc.) and static variables (weight, lengths etc.).

The accuracy of this dynamic model was compared to the physical setup. It showed to be fairly accurate, as the overall shapes of all the studied trajectories showed to be roughly the same. There was always room for improvement, e.g. by tweaking the friction coefficients which resulted in better matching velocities. There was, however, no single configuration of friction coefficients that improved the global accuracy of the simulator for each trajectory. This means that the other static variables that are immeasurable, such as the center of gravity and inertia of each body, are not entirely correct. A strategy to optimise these static variables was proposed and discussed, but not implemented as there was a strict time limit on this thesis due to the COVID-19 outbreak. Applying this strategy should definitely improve the accuracy of the simulator. Other discussed causes for deviant trajectories are the external factors such as the tendon inertia, air resistance etc. which are not included. Including these will also improve the accuracy.

This simulator optimisation strategy proposed an optimisation algorithm called gradient descent. It requires the model to be differentiable, which the dynamic model conveniently is. This algorithm is also used in the training of the CTRNN to approximate reference trajectories. It proved to be very efficient in calculating the minima of certain loss functions. Two loss functions were discussed. A loss

function defined on the positions of the end-effector, which allows flexible movement as the other positions are not defined, thus allowing the possibility for many different movements of each joint but still following the same end-effector trajectory. And a more strict loss function based on the angles and angle velocities of each joint, this loss function fully describes the movement and does not allow as much flexibility. The latter loss function proved overall to be the fastest and most accurate. But for some trajectories, the optimal minimum could not be reached as precision errors started to occur. Neither changing the sampling rate or changing the loss function helped solve these errors. This indicates that the CTRNN is not flexible enough to generate the forces for certain trajectories. For example, a perfect circle trajectory was chosen to be the reference. Neither the angles/angle velocities loss function nor the end-effector loss function was able to converge without precision errors. The same happened for a sinusoidal trajectory. These results imply that the CTRNN is not capable of generating complex oscillations. It did, however, proved to be very capable of generating other complex trajectories with sudden force changes.

Another very important topic is the loaded control of the finger. Out of the four tendons actuating the finger, the FS is proved to be redundant in unloaded control. This theory was tested by training the CTRNN on a loaded trajectory which represented the stroking of a piano key. The results are very promising, as the FS is fully active when the piano key is pressed and inactive when releasing. The CTRNN was already capable of fully pressing the key after 25 iterations.

8.1 Future work

This thesis was mostly an exploration of the capabilities of a simulated anthropomorphic finger and a CTRNN. The simulator is not perfect, therefore more research has to be done in order to increase the accuracy of the simulator by applying the proposed optimisation strategy in Chapter 6. There is also still much room for improvement for the CTRNN as it proved to be incapable of performing complex oscillations. A neural network with multiple layers could be the answer.

Once this is done, the dynamic model can be used to verify if a patient suffering from musician's focal dystonia can fully press a piano key by modelling the individual tendon couplings of the patient for a single finger.

The dynamic model should also be extended to incorporate multiple fingers and the full movement of an MCP joint, as only dorsal and palmar bending is considered in the current model.

Bibliography

[1] *Autograd*. https://github.com/HIPS/autograd. Accessed: 2020-05-25.

[2] Oscar Avilés Sánchez et al. "Simulation Model of an Anthropomorphic Hand". In: *International Journal of Applied Engineering Research* 11 (Dec. 2016), pp. 11114–11120.

[3] James Bradbury et al. *JAX: composable transformations of Python+NumPy programs*. Version 0.1.55. 2018. URL: http://github.com/google/jax.

[4] D. Bucher. "Central Pattern Generators". In: *Encyclopedia of Neuroscience*. Ed. by Larry R. Squire. Oxford: Academic Press, 2009, pp. 691–700. ISBN: 978-0-08-045046-9. DOI: https://doi.org/10.1016/B978-008045046-9.01944-6. URL: http://www.sciencedirect.com/science/article/pii/B9780080450469019446.

[5] Olivier Callewaert. "Ontwerp en bouw van een antropomorfe robotvinger". MA thesis. Universiteit Gent, 2017-2018.

[6] Ángel Campo and José Santos. "Evolution of adaptive center-crossing continuous time recurrent neural networks for biped robot control". In: (Jan. 2010).

[7] B. Daya, M. Chahine, and R. Awad. "A Recurrent Neural Network Model for Lamprey-Like Robot Movement". In: *2013 5th International Conference and Computational Intelligence and Communication Networks*. 2013, pp. 316–321.

[8] *Dynamixel MX-106T*. http://emanual.robotis.com/docs/en/dxl/mx/mx-106/. Accessed: 2020-05-25.

[9] Arne Van Erum. "Bio-inspired control of anthropomorphic robotic fingers". MA thesis. University of Ghent, 2018-2019.

[10] Matthew Guthrie and Gerd Wagner. "Demystifying the Lagrangian of classical mechanics". In: (July 2019).

[11] Nikolaus Hansen and Anne Auger. "CMA-ES: Evolution Strategies and Covariance Matrix Adaptation". In: *Proceedings of the 13th Annual Conference Companion on Genetic and Evolutionary Computation*. GECCO '11. Dublin, Ireland: Association for Computing Machinery, 2011, pp. 991–1010. ISBN: 9781450306904. DOI: 10.1145/2001858.2002123. URL: https://doi.org/10.1145/2001858.2002123.

[12] PhD J. N. Leijnse. "Developing an anthropomorphic robot finger: controllability conditions of humanoid tendon configurations, interphalangeal joint coupling mechanism, biomechanical swanneck pathology". In: (2019).

[13] Abbruzzese G. Front Hum Neurosci. Konczak J. "Focal dystonia in musicians: linking motor symptoms to somatosensory dysfunction". In: *Journal of Pharmaceutical Sciences* ().

[14] J. Leijnse et al. "Developing an anthropomorphic robot finger. Part 2: Design and functional evaluation of a prototype with multi- string interphalangeal joints coupling mechanism". In: *Transactions on Mechatronics* (Oct. 2019).

[15] J.N.A.L. Leijnse. "Why the lumbrical muscle should not be bigger - a force model of the lumbrical in the unloaded human finger". In: *Journal of Biomechanics* 30 (1997), pp. 1107–1114.

[16] J.N.A.L. Leijnse and J.J. Kalker. "A two-dimensional kinematic model of the lumbrical in the human finger". In: *Journal of Biomechanics* 28 (1995), pp. 237–249.

[17] Joris Leijnse and C Spoor. "Reverse engineering finger extensor apparatus morphology from measured coupled interphalangeal joint angle trajectories - a generic 2D kinematic model". In: *Journal of biomechanics* 45 (Nov. 2011), pp. 569–78. DOI: 10.1016/j.jbiomech.2011.11.002.

[18] Jonathan Maw, Kai Yuen Wong, and Patrick Gillespie. "Hand anatomy". In: *British Journal of Hospital Medicine* 77 (Mar. 2016), pp. C34–C40. DOI: 10.12968/hmed.2016.77.3.C34.

[19] Aaron Meurer et al. "SymPy: symbolic computing in Python". In: *PeerJ Computer Science* 3 (Jan. 2017), e103. ISSN: 2376-5992. DOI: 10.7717/peerj-cs.103. URL: https://doi.org/10.7717/peerj-cs.103.

[20] *Motive OptiTrack*. https://optitrack.com/products/motive/. Accessed: 2020-05-25.

[21] Ludovic Righetti and A J Ijspeert. "Pattern generators with sensory feedback for the control of quadruped locomotion". English (US). In: *IEEE International Conference on Robotics and Automation. ICRA 2008*. 2008, pp. 819–824.

[22] Pauli Virtanen et al. "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python". In: *Nature Methods* 17 (2020), pp. 261–272. DOI: https://doi.org/10.1038/s41592-019-0686-2.

[23] Gang Wang, Xi Chen, and Shi-Kai Han. "Central pattern generator and feedforward neural network-based self-adaptive gait control for a crab-like robot locomoting on complex terrain under two reflex mechanisms". In: *International Journal of Advanced Robotic Systems* 14.4 (2017), p. 1729881417723440. DOI: 10.1177/1729881417723440. eprint: https://doi.org/10.1177/1729881417723440. URL: https://doi.org/10.1177/1729881417723440.

[24] Xingming Wu et al. "CPGs with Continuous Adjustment of Phase Difference for Locomotion Control". In: *International Journal of Advanced Robotic Systems* 10.6 (2013), p. 269. DOI: 10.5772/56490. eprint: https://doi.org/10.5772/56490. URL: https://doi.org/10.5772/56490.

[25] *XLA: Optimizing Compiler for Machine Learning*. https://www.tensorflow.org/xla. Accessed: 2020-05-25.

Appendices

Appendix A

Reproducing the results

This Appendix contains the instructions to reproduce the results and to view the animations discussed in this thesis. Multiple code snippets have been provided in the form of a Jupyter notebook. These are linked to Google CoLab which can be run in any browser without any installation. The animations are placed on GitHub and can be downloaded from there. The results can also be locally generated by executing the corresponding Python file. The used libraries can be installed by passing the 'requirements.txt' to pip.

The GitHub repository is located at:

`https://github.com/killianstorm/simulated-anthropomorphic-finger`

A.1 Comparison between the physical setup and the simulator.

A.1.1 Locally

Execute `finger_model/analysis/comparison/comparison.py` to locally reproduce the results.

A.1.2 Notebook

All the results for the four comparison trajectories discussed in Chapter 5 can be calculated by executing the notebook at:

`https://colab.research.google.com/github/killianstorm/simulated-anthropomorphic-finger/blob/master/finger_model/notebooks/comparison_physical_setup/comparison_physical_setup.ipynb`

A.1.3 Animations

Full grasp comparison:

`https://github.com/killianstorm/simulated-anthropomorphic-finger/blob/master/animations/comparison/comparison_grasp.gif`

Extended PIP comparison:

`https://github.com/killianstorm/simulated-anthropomorphic-finger/blob/master/animations/comparison/comparison_extendedPIP.gif`

Extended MCP comparison:

`https://github.com/killianstorm/simulated-anthropomorphic-finger/blob/master/animations/comparison/comparison_extendedMCP.gif`

Complex comparison:

`https://github.com/killianstorm/simulated-anthropomorphic-finger/blob/master/animations/comparison/comparison_complex.gif`

A.2 Reproducing trajectories with differentiable programming

All the results for the approximations of the four trajectories discussed in Chapter 7 can be calculated by executing their corresponding notebooks.

A.2.1 Full grasp

Locally:

Run files in `finger_model/analysis/learning/tendon_model/unloaded/full_grasp/`.

Notebook:

`https://colab.research.google.com/github/killianstorm/simulated-anthropomorphic-finger/blob/master/finger_model/notebooks/unloaded/learning_grasp.ipynb`

Animation with angles/angle velocity loss function:

`https://github.com/killianstorm/simulated-anthropomorphic-finger/blob/master/animations/learning/unloaded/fullgrasp_angles_loss.gif`

Animation with end-effector loss function:

`https://github.com/killianstorm/simulated-anthropomorphic-finger/blob/master/animations/learning/unloaded/fullgrasp_endeffector_loss.gif`

A.2.2 Sine

Locally:

Run files in `finger_model/analysis/learning/tendon_model/unloaded/sine/`.

Notebook:

`https://colab.research.google.com/github/killianstorm/simulated-anthropomorphic-finger/blob/master/finger_model/notebooks/unloaded/learning_sine.ipynb`

Animation with angles/angle velocity loss function:

`https://github.com/killianstorm/simulated-anthropomorphic-finger/blob/master/animations/learning/unloaded/sine_angles_loss.gif`

Animation with end-effector loss function:

https://github.com/killianstorm/simulated-anthropomorphic-finger/blob/master/animations/learning/unloaded/sine_endeffector_loss.gif

A.2.3 Perfect circle

Locally:

Run files in `finger_model/analysis/learning/tendon_model/unloaded/circle/`.

Notebook:

https://colab.research.google.com/github/killianstorm/simulated-anthropomorphic-finger/blob/master/finger_model/notebooks/unloaded/learning_circle.ipynb

Animation with end-effector loss function:

https://github.com/killianstorm/simulated-anthropomorphic-finger/blob/master/animations/learning/unloaded/perfectcircle_endeffector_loss.gif

A.2.4 Sudden force change

Locally:

Run files in `finger_model/analysis/learning/tendon_model/unloaded/sudden_force_change/`.

Notebook:

https://colab.research.google.com/github/killianstorm/simulated-anthropomorphic-finger/blob/pianokey/finger_model/notebooks/loaded/learning_pianokey.ipynb

Animation with angles/angle velocity loss function:

https://github.com/killianstorm/simulated-anthropomorphic-finger/blob/master/animations/learning/unloaded/suddenforcechange_angles_loss.gif

Animation with end-effector loss function:

https://github.com/killianstorm/simulated-anthropomorphic-finger/blob/master/animations/learning/unloaded/suddenforcechange_endeffector.gif

A.2.5 Piano key stroke

Locally:

Run files in `finger_model/analysis/learning/tendon_model/loaded/keystroke/`.

Notebook:

https://colab.research.google.com/github/killianstorm/simulated-anthropomorphic-finger/blob/pianokey/finger_model/notebooks/loaded/learning_pianokey.ipynb

Animation with angles/angle velocity loss function:

https://github.com/killianstorm/simulated-anthropomorphic-finger/blob/master/animations/learning/loaded/keystroke_angles_loss.gif

Appendix B

Source code

This Appendix covers details of the source code and several pitfalls of the JAX library. The full source code can be found on GitHub: `https://github.com/killianstorm/simulated-anthropomorphic-finger.git`

B.1 Code usage

This section explains how to use the code in order to simulate and approximate custom trajectories and create their animations. Appendix A contains ready-to-use code snippets to reproduce the results discussed in the previous Chapters. These instructions are for custom trajectories.

B.1.1 File tree

The file structure is as follows:

```
.
├── requirements.txt
├── README.md
├── finger_model
│   ├── tools
│   │   └── plots.py
│   ├── analysis
│   │   ├── learning
│   │   │   └── …
│   │   └── comparison
│   │       └── …
│   ├── simulation
│   │   ├── loss_functions.py
│   │   ├── dynamic_model.py
│   │   ├── simulator.py
│   │   └── optimizer.py
│   └── notebooks
└── animations
```

where *loss_functions.py* contains the definition of the loss functions, *dynamic_model.py* the implementation of the dynamic model, *simulator.py* the implementation of the simulator and *optimizer.py* the implementation of the optimisation algorithm.

B.1.2 Trajectory simulation

In order to simulate a trajectory, several options are available which are listed in Table B.1. Each has its own function to be called with different parameters.

These are all defined in `finger_model/simulation/simulator.py`.

Function	Purpose	Parameters
simulate_sine	applies sinusoidal forces on the tendons	a dictionary containing the interval, the amplitudes and the phases
simulate_constant	applies constant forces on the tendons	a dictionary containing the interval and the constant forces for each tendon
simulate_predefined	applies predefined lists of forces on the tendons	a dictionary containing the interval and the lists containing the forces for each tendon at each time instant
simulate_ctrnn	applies the forces generated by a CTRNN on the tendons	a dictionary containing the interval and the parameters defining the CTRNN

Table B.1: Overview of the simulation functions and their parameters defined in `finger_model/simulation/simulator.py`.

The following shows a toy problem for a simple trajectory simulation with the CTRNN:

```
from simulation.optimizer import *
from simulation.loss_functions import *

# Interval.
tmax, dt = 3., 0.04
interval = num.arange(0, tmax + dt, dt)

p_predefined = {
    'interval': interval,
    'F_fs': np.ones(interval.shape[0]),
    'F_io': np.ones(interval.shape[0]),
    'F_fp': np.ones(interval.shape[0]),
    'F_ed': np.ones(interval.shape[0]),
}

reference = simulate_predefined(p_predefined)
```

reference contains all the information of the trajectory of the finger, e.g. absolute angles, their velocities etc. This code snippet will create a simulation where a force of 1N is applied on each tendon during three seconds.

B.1.3 Trajectory learning

In order to learn a trajectory. A reference trajectory must first be defined, this can be done by using the provided simulation functions or by defining a custom trajectory by hand. An example:

```
from finger_model.analysis.learning.gradient_descent import *

# Interval.
tmax, dt = 1., 0.001
interval = num.arange(0, tmax + dt, dt)

fp = []
for i in interval:
    fp.append(10.)

p_predefined = {
    'interval': interval,
    'F_fs': np.zeros(interval.shape[0]),
    'F_io': np.zeros(interval.shape[0]),
    'F_fp': np.array(fp),
    'F_ed': np.zeros(interval.shape[0]),
}

name = "fullgrasp"
reference = simulate_predefined(p_predefined)
plots.animate(reference, dt, name, tendons=True)

loss_function = loss_angles

# Learn to reproduce trajectory using gradient descent.
learn_gradient_descent(reference, interval, 250, loss_function=loss_function, name=name)
```

This is the code snippet used for generating the full grasp trajectory. *learn_gradient_descent* takes in the reference, the interval, the number of iterations, the loss function and a name for the simulation. It will also generate all the graphs and an animation.

B.1.4 Creating an animation

An animation can easily be created by calling the *plots.animate* function. It takes in the simulated trajectory, the time between each sample and a name.

```
plots.animate(simulated_trajectory, dt, name, GIF=False)
```

There is an option to output to a GIF file instead of a MP4 file. It also supports two trajectories, e.g. to compare two trajectories next to each other:

```
plots.animate(reference_trajectory, dt, name, approximate_trajectory, GIF=False)
```

B.1.5 Optional model parameters

Switching between a loaded and unloaded trajectory can be done by setting the *ENABLE_PIANO_KEY* boolean to respectively True for loaded and False for unloaded. There is also an option to disable the ligaments by changing the *ENABLE_LIGAMENTS* boolean. Both of these booleans are located in `finger_model/simulation/dynamic_model.py`.

B.2 Implementation

This section highlights the implementation of the dynamic model, the simulation and the optimisation algorithm.

B.2.1 JAX

JAX requires the use of pure functional code, which means that the object-oriented nature of Python must be avoided. The numerical Python package Numpy is therefore also not usable. Luckily JAX provides a functional port of Numpy. This involves certain changes in the code, for example:

Normally, updating a element in a Numpy array is as easy as:

```
import numpy as np
a = np.array([1, 2, 3])
a[0] = 2
# a is now [2, 2, 3]
```

This assignment will not work in JAX. Instead, JAX provides an *index_update* function that takes an array, the index and the new value on that index and returns the array with the updated index:

```
import jax.numpy as np
from jax.ops import index_update
a = np.array([1, 2, 3])
a = index_update(a, 0, 2)
# a is now [2, 2, 3]
```

This is just an example of the many changes involved with using JAX. More pitfalls can be found on their website [3].

B.2.2 Dynamic model

As mentioned before, the dynamic model is constructed with the help of SymPy [19], which is a Python package for symbolic mathematics. The first step is to define the dynamic and static variables describing the model. SymPy offers the support for dynamic variables with the function *dynamicsymbols*, static variables are declared with the function *symbols*.

The syntax in Python is given as:

```
## Dynamic variables

# Absolute angles
theta1 = dynamicsymbols('theta1')
theta2 = dynamicsymbols('theta2')
theta3 = dynamicsymbols('theta3')

# Absolute angle velocities
thetad1 = dynamicsymbols('theta1', 1)
thetad2 = dynamicsymbols('theta2', 1)
thetad3 = dynamicsymbols('theta3', 1)

# Force on each muscle.
F_fs, F_io, F_fp, F_ed = dynamicsymbols('F_fs,␣F_io,␣F_fp,␣F_ed')


## Static variables
l1, m1, l2, m2, I2, l3, m3, g, c_fr = symbols('l1,␣m1,␣l2,␣m2,␣I2,␣l3,␣m3,␣g,␣c_fr')
```

With these static and dynamic variables, the other necessary variables can be calculated.

The coordinates of each phalanx:

```
# Proximal phalanx coordinates
x1 = l1 * sin(theta1)
```

```
z1 = -l1 * cos(theta1)

# Middle phalanx coordinates
x2 = x1 + l2 * sin(theta2)
z2 = z1 - l2 * cos(theta2)

# Distal phalanx coordinates
x3 = x2 + l3 * sin(theta3)
z3 = z2 - l3 * cos(theta3)
```

The coordinates of the center of gravity of each phalanx:

```
# Proximal phalanx center of gravity coordinates
xc1 = (l1 / 2.) * sin(theta1)
zc1 = - (l1 / 2.) * cos(theta1)

# Middle phalanx center of gravity coordinates
xc2 = x1 + (l2 / 2.) * sin(theta2)
zc2 = z1 - (l2 / 2.) * cos(theta2)

# Distal phalanx center of gravity coordinates
xc3 = x2 + (l3 / 2.) * sin(theta3)
zc3 = z2 - (l3 / 2.) * cos(theta3)
```

The velocities of each center of gravity:

```
# Proximal phalanx center of gravity velocities
xc1d = diff(xc1, t)
zc1d = diff(zc1, t)

# Middle phalanx center of gravity velocities
xc2d = diff(xc2, t)
zc2d = diff(zc2, t)

# Distal phalanx center of gravity velocities
xc3d = diff(xc3, t)
zc3d = diff(zc3, t)
```

And lastly, the relative angles (α_i):

```
# Relative angle MCP joint.
alpha1 = Rational(1, 2) * pi + theta1
alpha1d = diff(alpha1, t)

# Relative angle PIP joint.
alpha2 = pi - (theta1 - theta2)
alpha2d = diff(alpha2, t)

# Relative angle DIP joint.
alpha3 = pi - (theta2 - theta3)
alpha3d = diff(alpha3, t)
```

With these dynamic and static variables, the Lagrangian can be constructed:

```
# Potential energy.
V = m1 * g * zc1 + m2 * g * zc2 + m3 * g * zc3

# Kinetic energy.
T1 = Rational(1, 2) * m1 * (xc1d ** 2 + zc1d ** 2) + Rational(1, 2) * (thetad1 ** 2) * I1
T2 = Rational(1, 2) * m2 * (xc2d ** 2 + zc2d ** 2) + Rational(1, 2) * (thetad2 ** 2) * I2
T3 = Rational(1, 2) * m3 * (xc3d ** 2 + zc3d ** 2) + Rational(1, 2) * (thetad3 ** 2) * I3
```

```
# Lagrangian.
L = (T1 + T2 + T3) - V
```

B.2.3 Joints

The next step is to model the movements between each phalanx. The joints can be defined by using a ReferenceFrame for the phalanges. The phalanges rotate in the 2D plane with an absolute angle velocity. The relative angle velocity is also necessary to model the friction:

```
# The main reference frame.
N = ReferenceFrame('N')

# Proximal phalanx absolute angular movement.
r_pp = ReferenceFrame('rigid_pp')
r_pp.set_ang_vel(N, thetad1 * N.z)

# Middle phalanx absolute angular movement.
r_mp = ReferenceFrame('rigid_mp')
r_mp.set_ang_vel(N, thetad2 * N.z)

# Distal phalanx absolute angular movement.
r_dp = ReferenceFrame('rigid_dp')
r_dp.set_ang_vel(N, thetad3 * N.z)

# PIP joint relative angular movement.
j_pp_mp = ReferenceFrame('joint_pp_mp')
j_pp_mp.set_ang_vel(N, alpha2d * N.z)

# DIP joint relative angular movement.
j_mp_dp = ReferenceFrame('joint_mp_dp')
j_mp_dp.set_ang_vel(N, alpha3d * N.z)
```

The next step is to model the torques generated by the tendons:

```
# RADII is a dictionary containing the moment arms

# DIP moment caused by FP.
M_FP_DIP = - F_fp * RADII[J_DIP][T_FP]

# DIP moment caused by ED.
M_ED_DIP = - F_ed * ((alpha2 - pip_angle_bounds[0]) / (pip_angle_bounds[1] - pip_angle_bounds
    [0])) * lengths[2]

# PIP Moments caused by tendons FS, IO, FP and ED.
M_FS_PIP = - F_fs * RADII[J_PIP][T_FS]
M_IO_PIP = - F_io * RADII[J_PIP][T_IO]
M_FP_PIP = - F_fp * RADII[J_PIP][T_FP]
M_ED_PIP = - F_ed * RADII[J_PIP][T_ED]

# MCP Moments caused by tendons FS, IO, FP and ED
M_FS_MCP = - F_fs * RADII[J_MCP][T_FS]
M_IO_MCP = - F_io * RADII[J_MCP][T_IO]
M_FP_MCP = - F_fp * RADII[J_MCP][T_FP]
M_ED_MCP = - F_ed * RADII[J_MCP][T_ED]

# DIP torque.
tau3 =   M_FP_DIP \
       - M_ED_DIP

# PIP torque.
```

```
tau2 =   M_FS_PIP \
       - M_IO_PIP \
       + M_FP_PIP \
       - M_ED_PIP

# MCP torque.
tau1 =   M_FS_MCP \
       + M_IO_MCP \
       + M_FP_MCP \
       - M_ED_MCP
```

The ligaments which are modeled as a counter-torque that activates once a joint reaches its angle boundary:

```
def sigmoid_heaviside(x):
    """
    Sigmoid approximation of the Heaviside function.
    """
    return 1. / (1 + exp(- 16 * x))

def d2r(d):
    """
    Degrees to radians
    """
    return d * (pi / 180.)

# Joint angle bounds.
mcp_angle_bounds = (d2r(95), d2r(225))
pip_angle_bounds = (d2r(60), d2r(220))
dip_angle_bounds = (d2r(80), d2r(185))

counter_torque = 4.

def ligament(torque, relative_angle, angle_bounds):
    return - sigmoid_heaviside(-torque) * sigmoid_heaviside(angle_bounds[0] - relative_angle)
         * torque * counter_torque - sigmoid_heaviside(torque) * sigmoid_heaviside(
        relative_angle - angle_bounds[1]) * torque * counter_torque

ligament_MCP = ligament(tau1, alpha1, mcp_angle_bounds)
ligament_PIP = ligament(tau2, alpha2, pip_angle_bounds)
ligament_DIP = ligament(tau3, alpha3, dip_angle_bounds)
```

The piano key, which should only be activated if the trajectory is loaded. The force generated by the piano key on the finger when it is pressed is modeled as a torque on each joint that enables once the fingertip is in a certain region.

```
if ENABLE_PIANO_KEY:
    def load(length):
        return sigmoid_heaviside(x3 - pianokey_coordinates[0]) * \
               sigmoid_heaviside(-(z3 - pianokey_coordinates[1])) * \
               length * piano_force * abs(z3 - pianokey_coordinates[1])

    load_DIP = load(lengths[2])
    load_PIP = load(lengths[1] + lengths[2])
    load_MCP = load(lengths[0] + lengths[1] + lengths[2])
```

The torques on each joint generated by the tendons, the ligaments and the friction are then placed in a force matrix:

```
# Friction coefficients.
c_fr_mcp = 0.1
```

```
c_fr_pip = 0.1
c_fr_dip = 0.1

# Force list with ligaments, friction and optional piano key load.
FL = [(r_pp, (tau1 + ligament_MCP + load_MCP) * N.z),  # Tendon torques
      (r_mp, (tau2 + ligament_PIP + load_PIP) * N.z),
      (r_dp, (tau3 + ligament_DIP + load_DIP) * N.z),

      (r_pp,    (-c_fr_mcp * alpha1d) * N.z),  # Joint frictions
      (j_pp_mp, (-c_fr_pip * alpha2d) * N.z),
      (j_mp_dp, (-c_fr_dip * alpha3d) * N.z)
      ]
```

To form the Euler-Lagrangian equations (Eq. 4.2), the force list and the reference frame (N) are passed into the *LagrangesMethod* function of the SymPy package along with the absolute angles and the Lagrangian L.

```
# Construction of Euler-Lagrange equations.
LM = LagrangesMethod(L, [theta1, theta2, theta3], forcelist=FL, frame=N)

# Calculate equations of motion.
equations = LM.form_lagranges_equations()
```

This returns a mass and force matrix representing the equations of motion. These are however still symbolically represented. JAX is numerical package, which means the matrices will have to be 'lambdified'. Lambdifying allows the translation of symbolic to numeric representation. A function called 'lambdify' is provided by SymPy. It takes in the symbolic representation and a dictionary with information on how to translate the symbolic expressions.

```
# List of static parameters describing the finger and its environment.
parameters = [
            l1, m1, I1,
            l2, m2, I2,
            l3, m3, I3,
            F_fs, F_io, F_fp, F_ed,
            g, c_fr]

# List of unknown dynamic variables.
y = [theta1, theta2, theta3,
     thetad1, thetad2, thetad3]

# The unknown variables have to substituted by Dummy objects to mark them as unknown.
unknowns = [Dummy() for _ in y]
unknown_dict = dict(zip(y, unknowns))

# The Dummy objects are then substituted in the mass and force matrix.
mm = LM.mass_matrix_full.subs(unknown_dict)
fm = LM.forcing_full.subs(unknown_dict)

# Mapping of sympy functions to jax numpy functions to enable lambdifying.
mapping = {'sin': np.sin, 'cos': np.cos, 'tanh': np.tanh, 'pi': np.pi, 'array': np.array, '
    ImmutableDenseMatrix': np.array, 'exp': np.exp}

# Lambdify matrices.
mass_matrix = lambdify([unknowns] + parameters, mm, mapping)
forcing_matrix = lambdify([unknowns] + parameters, fm, mapping)
```

The mass and force matrix are now in their numerical form ready to be used by numpy.

B.2.4 Simulation

The equations of motion are a set of Ordinary Differential Equations (ODEs). These can now be simulated with the help of an ODE solver.

Formulating and solving the ODE

JAX has a function for solving a set of ODE called *odeint* which takes in a function representing the ODE, a vector of time instants and the initial positions for the dynamic variables. As mentioned in Chapter 4, the equations of motion take in the static variables and an initial position (comprised of the angles and angle velocities of each phalanx). The *ODE* is defined as:

```
def ode(y, time, forces):
    '''
    Representation of the ODE.

    Parameters:
        y (vector): initial positions.
        time (vector): time instants
        forces (vector): applied forces on tendons
    '''

    # Formulate parameters.
    params = [y,
              lengths[0], masses[0], inertias[0],
              lengths[1], masses[1], inertias[1],
              lengths[2], masses[2], inertias[2],
              *forces,
              9.8, 0.5]

    # Solve equations of motion on current time instant.
    return np.linalg.pinv(MM(*params)) @ FM(*params)
```

The result is a matrix containing the angles of each joint and their velocities for each time instant in the provided interval. This matrix contains all the necessary information to create animations.

Continuous Time Recurrent Neural Network (CTRNN)

The forces which have to be passed to the ODE are generated by the Continuous Time Recurrent Neural Network (CTRNN). A simulation function called 'simulate_ctrnn_oscillator' takes in the parameters defining the CTRNN and a time interval. It calculates the forces generated by the CTRNN by passing in the parameters. The output is then passed to the 'odeint' function.

```
@jit
def simulate_ctrnn(p):
    '''
    Simulates the finger with a continuous time recurrent neural network (CTRNN).
    parameters: dict containing interval, taus, biases, states, gains and weights
    '''

    # 4 neurons.
    size = 4

    # Max force of 40N.
    max_val = 40.

    dt = p['interval'][1] - p['interval'][0]
```

```
    # Set up network.
    taus = p[RNN_TAUS]
    biases = p[RNN_BIASES]
    weights = np.array(p[RNN_WEIGHTS]).reshape(size, size)
    states = p[RNN_STATES]
    gains = p[RNN_GAINS]
    out = jax.nn.sigmoid(states)

    # Step function for network.
    def step(current_state, index):
        _output, _state = current_state
        external_inputs = np.zeros(size)  # No external inputs.
        total_inputs = external_inputs + np.dot(weights, _output)
        _state += dt * (1. / taus) * (total_inputs - _state)
        _out = jax.nn.sigmoid(gains * (_state + biases))
        return (_out, _state), _out

    # Simulate network to get output.
    _, outputs = jax.lax.scan(step, (out, states), p['interval'])

    # Scale output according to max values of force or torque.
    outputs = np.multiply(max_val, outputs)

    @jit
    def ode(y, time, _dt, _torques):

        # Calculate which index for current time instant.
        _index = time / _dt
        _out = _torques[_index.astype(int)]

        _params = [y,
                   lengths[0], masses[0], inertias[0],
                   lengths[1], masses[1], inertias[1],
                   lengths[2], masses[2], inertias[2],
                   *_out,
                   9.8, 0.5]
        return np.linalg.inv(MM(*_params)) @ FM(*_params)

    # Solve with CTRNN outputs on interval.
    history = odeint_jax(ode, initial_positions, p['interval'], dt, outputs)

    results = calculate_positions(history)
    results['torques'] = outputs

    return results
```

calculate_positions is a supportive function that translates the angles and their velocities into usable Cartesian coordinates of each phalanx. Notice the optimisation of the execution of the CTRNN:

```
# Step function for network.
def step(current_state, index):
    _output, _state = current_state
    external_inputs = np.zeros(size)  # No external inputs.
    total_inputs = external_inputs + np.dot(weights, _output)
    _state += dt * (1. / taus) * (total_inputs - _state)
    _out = jax.nn.sigmoid(gains * (_state + biases))
    return (_out, _state), _out

# Simulate network to get output.
_, outputs = jax.lax.scan(step, (out, states), p['interval'])
```

jax.lax.scan is an optimised loop function, similar to the for-loop of Python. It is however much more efficient as it prevents loop unrolling, and thus preventing significant performance issues. It takes in a step function describing the things to be done each step, a state and an interval to iterate. The *step* function iterates over the time interval and generates the output of the CTRNN. The outputs are saved in *_out*.

B.2.5 Optimisation algorithm

With the implementation of the dynamic model and the simulator, the optimisation algorithm can now be implemented. The first thing to do is define the loss function. This is either the angles/angle velocities or the end-effector loss function. The loss function is defined as a wrapper that encapsulates the simulator and the actual calculation of the root mean square error (RMSE):

```
@jit
def _loss_wrapper(p):
    """
    Loss function wrapper which simulates the trajectory and calculates the loss.
    """
    _reference = p['reference']
    _interval = p['interval']
    _simulated = simulate_ctrnn(p)
    return loss(_reference, _simulated)
```

where the argument p contains the interval, the reference and the initial parameters defining the continuous time recurrent neural network (CTRNN).

This function is then transformed passed into the *value_and_grad* JAX function. This function is also JIT compiled.

```
    grad_function = jit(value_and_grad(lambda gradient_params, static_params: _loss_wrapper
        ({**gradient_params, **static_params})))
```

The reason the loss wrapper has been passed as a lambda is to allow the choice of which parameters should be optimised. These are located in a dictionary called *gradient_params*. *static_params* contains information like the reference and the interval.

As mentioned in Chapter 7, the minimisation of the loss function is handled by the SciPy *minimize* function. It requires an objective and a number of iterations. It also offers support for callbacks.

The objective is defined as:

```
def objective(params):
    p = array_to_dict(params)
    return _loss_wrapper({**p, **static_params})

objective_with_grad = jit(value_and_grad(objective))
```

The reason for the extra wrapping is because the *minimize* function only accepts an array of parameters to be optimised. This wrapping allows a dictionary to be used.

The minimisation is then done as follows:

```
result = minimize(
    objective_with_grad,
    dict_to_array(grad_params),
    jac=True,
    method='CG',
    options={
```

```
        'maxiter': iterations,
        'disp': True
    },
    callback=iteration_callback)
```

where *iteration_callback* is a callback helper function used for indicating the progress.

The result is an array containing the optimised parameters for a minimised loss function. These can be passed into the *simulate_ctrnn* function to visualise the approximation.

FACULTY OF ENGINEERING AND ARCHITECTURE

Design and control of a simulated anthropomorphic robotic finger using differentiable programming

Killian Storm
Student number: 01500259

Supervisors: Prof. dr. ir. Francis wyffels, Prof. dr. Joris Nikolaas Leijnse
Counsellor: Rembert Daems

Master's dissertation submitted in order to obtain the academic degree of
Master of Science in Computer Science Engineering

Academic year 2019-2020

GHENT UNIVERSITY