MASTER THESIS

ONLINE OPTIMIZATION OF EMG USING A HYBRID MODEL APPROACH

Thijs van Aalten

FACULTY OF ENGINEERING TECHNOLOGY
DEPARTMENT OF BIOMECHANICAL ENGINEERING

EXAMINATION COMMITTEE
G.V. Durandau
M. Sartori
D. Dresscher
H. Van der Kooij

DOCUMENT NUMBER
BE - 776

UNIVERSITY OF TWENTE.

04-01-2021

Abstract

Two state of the art methods for neuromusculoskeletal modeling are inverse dynamics based modeling and electromyography (EMG) driven modeling. These methods can be combined into a hybrid model to benefit from the strengths of both methods. A real-time hybrid neuromusculoskeletal model was developed that enables real-time measurement of ankle joint's EMG signals that account for realistic joint torques. Simulated annealing is used to optimize for excitations that resemble measured EMG and produce joint torque close to joint torque measured by inverse dynamics. Human kinematic data, ground reaction force and EMG were measured and used to test the model's ability to calculate optimized excitations in real-time. It is shown that real-time calculated optimized excitations show large correlation with EMG signals that were optimized in an offline environment. Joint torques resulting from optimized excitations show large correlation with joint torques measured by inverse dynamics. This real-time hybrid neuromusculoskeletal model can potentially be used in a clinical environment to obtain online measurements of neuromuscular data and can be used to drive a wearable robotic device.

Preface

The completion of this master thesis is the finish mark of my 3.5-year long period at the University of Twente (as a student). Being the big climax of the master, the thesis formed a major part of my adventure in Enschede. Starting this adventure in September 2017, I would never have thought that it would end sitting in a 16 m^2 student room for 9 months.

One does not finish his master thesis all by himself. Therefore, I want to thank everyone that helped or supported me while finishing my master thesis. Special thanks go to the following people.

First of all, I want to thank my daily supervisor Guillaume Durandau for always being ready to help me if I had a question or if I got stuck with my code. Although I never dared to try, I think I could ring his doorbell in the middle of the night with a question and he would still let me inside and look at the code with me.

Secondly, I want to thank Annabel for reminding me that a compliment is a compliment, and preventing me from finding a way to turn it around into a negative remark.

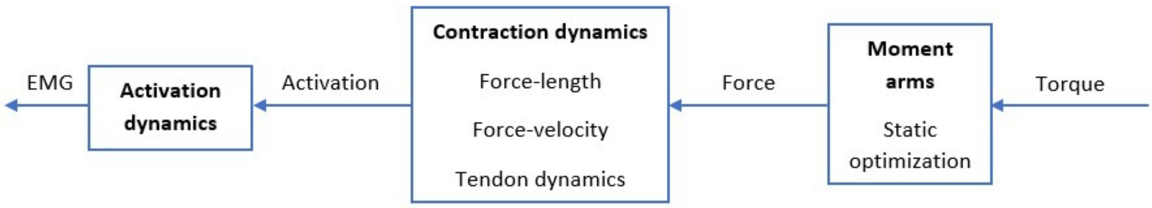
Finally, I want to thank my roommates from Huize BosHut for helping me find my *chill* when I had lost it again. I know it must be hard for students to have a roommate who wants to go to bed in time because he is busy with his thesis, but you (almost) always understood that choice.

Contents

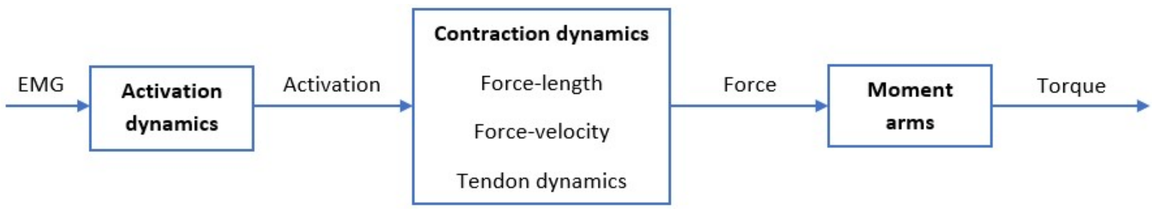
1 Introduction **1**
2 Software implementation **5**
2.1 Hybrid model 5
2.2 XML file 6
2.3 Optimization plugin 6
2.4 Bug fixes 8
2.5 Pagmo 10
3 Experimental analysis **11**
3.1 Data collection 11
3.2 DOF choice 11
3.3 Algorithms 12
3.4 Validation 12
4 Results **13**
5 Discussion **18**
Appendix **21**
A XML file 21
B Erase unused DOFs 21
C Optimization plugin 22
D User-defined problem 35

1 Introduction

Neuromusculoskeletal (NMS) modeling gives insight in how the human body generates motion. The brain sends neural signals to the muscle-tendon units (MTUs) by firing alpha motor neurons. Innervated MTUs generate forces acting on the bones, resulting in joint moments that move body segments and make interaction with the environment possible. Nowadays, treatments for mobility impairments are often selected based on clinical experience of the clinician rather than objective prediction of post-treatment function (Fregly et al., 2012). NMS modeling gives understanding of the NMS mechanisms involved in the human body, enabling clinicians to see what muscle causes an abnormal gait pattern and to predict the outcome of a treatment. It can also be used to observe load patterns during sports activities, giving insight in the cause of an injury. Direct measurement of internal properties of the NMS system is often not possible, so in order to fully understand what is happening in the human body in case of a pathology or injury, it is important to have NMS models that give insight in certain properties of the NMS system. Current state-of-the-art NMS systems are inverse dynamics (ID) based models and electromyography (EMG) driven models.



(a) Schematic of ID-based models. Torque is calculated from joint kinematics and ground reaction force using the equations of motion. Then, static optimization is done to overcome the muscle load sharing problem. Resulting muscle forces can be fed into contraction dynamics to calculate muscle activation. Using activation dynamics, muscle excitations can be calculated.



(b) Schematic of EMG-based NMS models. Raw EMG is measured and, high-pass filtered, full-wave rectified, low-pass filtered and normalized. This is used as input to the activation dynamics describing the delay caused by calcium release from the sarcoplasmic reticulum. Contraction dynamics uses a Hill-type muscle model to calculate muscle force, which is translated into joint torque using the moment arms.

Figure 1: Schematics of ID-based models (a) and EMG-based models (b).

ID based models (figure 1a) use measured joint kinematics and ground reaction force (GRF) as input of the model to calculate joint torques from the equations of motion (Erdemir et al., 2007). Joint torques and moment arms are then used to calculate muscle forces. In general, the number of muscles is larger than the number of joints. This muscular load sharing problem can be solved by static optimization, which requires an assumption of the strategy used by the brain to coordinate MTU activation. Sum of cubed muscle stresses or sum of squared muscle forces are commonly used as minimization criteria, but other criteria can also be used depending on the task. The assumption on muscle recruitment strategy determines the calculated muscle force, so ID-based models are unable to account for differences in recruitment strategy due to pathology or environmental conditions such as a slippery or uneven surface. ID-based models usually don't make use of a calibrated muscle model for the translation from muscle force to EMG. Therefore, they are unable to account for differences in subject-specific muscle properties when estimating EMG.

EMG is an indirect measure of the excitation of muscles. Therefore, measured EMG can be used as input of a forward dynamics model to model human movement mechanics while accounting for differences in muscle recruitment strategy. These EMG-driven models (figure 1b) are based on the Hill-type muscle model (Hill, 1938). EMG measurements are used as input to the MTUs to calculate muscle forces and joint moments. To be able to calculate these forces and moments for different individuals, the model must have knowledge of subject-specific parameters such as muscle's optimal fiber length, tendon slack length, maximum isometric force and many others. These parameters are calculated with a calibration of the model. Experimental joint moments calculated with ID are used as validation of the predicted joint moments by the EMG-driven model during calibration. However, when performing tasks that were not done during calibration, the predicted joint moment does not always represent the experimental joint moment. This is because of limitations of surface EMG measurements such as cross-talk, movement artefacts, the inability to access deep muscles and the weak association between EMG amplitude and motor unit action potentials (Farina and Negro, 2012). Furthermore, NMS modeling using EMG involves model imperfections due to simplifications and errors in model parameters.

To benefit from both EMG-driven model's ability to account for differences in individual's muscle recruitment strategy and static optimization based method's ability to account for the right joint moments, Sartori et al. (2014) developed a hybrid NMS model (figure 2), combining EMG-driven modeling with static optimization methods. The model adjusts measured EMG to track experimental joint moments. Its static optimization component makes sure that (1) experimental joint torques are tracked, (2) EMG measurements are minimally adjusted and (3) squared excitations are low. While EMG-driven models can only measure EMG of superficial muscles, the hybrid model also enables EMG measurements of deep muscles. Furthermore, current EMG-driven methods involve uncertainties in EMG measurements caused by cross-talk, movement artefacts and the weak association between EMG amplitude and motor unit action potentials. Therefore, muscle forces and joint torques predictions may contain errors. The hybrid model counteracts these errors because it ensures tracking of experimental joint torques. Therefore, the hybrid model combines the benefits from both EMG-driven and ID-based models.

NMS models are elaborate models requiring a lot of computations. Therefore, most NMS models are used offline. Certain methods can be used to make the model faster, so it can be used in real-time. Real-time models are models that can do computations fast enough to provide useful information on the current movement and can be used for applications

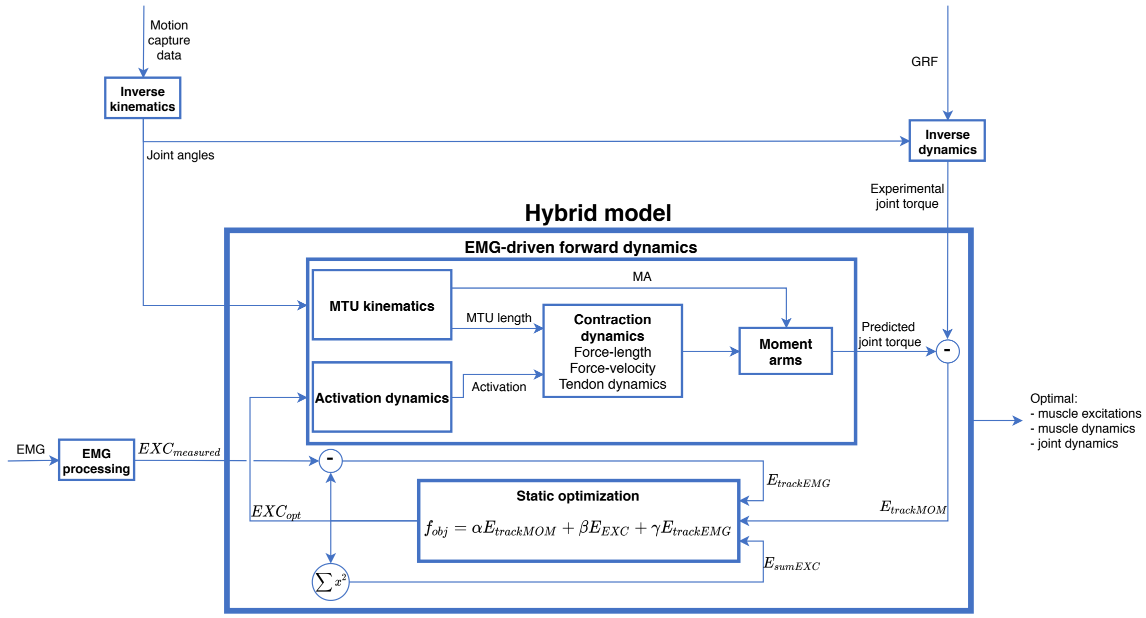


Figure 2: Working mechanism of the hybrid model. EMG-driven forward dynamics is used to calculate predicted joint torque from measured excitations and kinematics. ID and kinematics are used to calculate experimental joint torque. The hybrid model does a static optimization where it optimizes for (1) predicted joint torques close to experimental joint torques, (2) sum of squared excitations and (3) excitations close to measured excitations. Adapted from "Hybrid neuromusculoskeletal modeling to best track joint moments using a balance between muscle excitations derived from electromyograms and optimization," by M. Sartori, D. Farina and D. G. Lloyd, 2014, *Journal of Biomechanics, 47(15), p. 3615.*

such as biofeedback (Pizzolato et al., 2017) or wearable robotic devices (Sartori et al., 2018; Durandau et al., 2019). Wearable robotic devices are usually controlled using excitations, so the real-time hybrid model could generate adjusted excitations to control the device. Currently, state of the art NMS models are used to calculate joint loads (Kinney et al., 2013), joint stiffness (Sartori et al., 2015) or muscle stiffness in an offline environment. This induces a delay in obtaining clinical data. A real-time NMS model is important in a clinical environment, because it enables immediate insight into clinical data such as joint loads. The real-time hybrid model can help to increase precision of the estimation of muscle stiffness and joint loads compared to an open-loop NMS model or an ID-based model.

Durandau et al. (2018) adapted the offline EMG-driven model from Sartori et al. (2012) to work in real-time by changing the OpenSim inverse kinematics (IK) algorithm to a multithreaded algorithm and simultaneously running multiple IK computations on different threads. The model also used one B-spline function per MTU for faster calculation of the muscle-tendon length and moment arm. Another possible method to improve computation time is to gather kinematics data by using inertial measurement units or joint sensors instead of optical markers that involve time-consuming IK computations.

The goals of this study are (1) to adapt the hybrid model from Sartori et al. (2014) to work in real-time, (2) validate if the algorithm meets real-time requirements and (3) validate the output of the model using experimental data.

2 Software implementation

The online optimization plugin is written in C++ as an extension to the model written by Durandau et al. (2018). This model performs real-time open-loop musculoskeletal modeling as in figure 1b. It uses activation dynamics to calculate activation from measured excitations. The OpenSim Inverse Kinematics algorithm (Delp et al., 2007; Seth et al., 2018) calculates joint angles from experimental markers. Musculotendon lengths (LMT) and moment arms (MA) are then calculated from joint angles using multidimensional cubic Bspline functions. Then, using the force-length and force-velocity relationships and tendon dynamics it calculates muscle force from muscle activations and LMT. Predicted joint torque is calculated from muscle force and MA.

The open-loop model consists of one core class with the NMS model containing its properties and data. The NMS model's properties include subject-specific parameters such as muscle's optimal fiber length, tendon slack length, maximum isometric force and many others and are set during the initialization. The NMS model's data are received from plugins during execution of the software. The EMG plugin receives time and EMG data and processes measured EMG signals to obtain excitations. The IK-ID plugin receives angle data and computes LMT and MA from that. It also receives GRF data in order to compute ID torque.

Data can be measured in real-time or read from file. The former method is used when doing experiments. Data are measured and processed by their plugins and are immediately sent to the NMS model. The latter method can be used to test the real-time model without the need to do an experiment each time you test the data. Data can be prerecorded and saved to a file. When running the software, data is read from this file frame by frame as if the measurement is done in real-time. During this study, data was always read from file. This holds for the software analysis phase described in this section as well as for the experimental analysis described in section 3.

2.1 Hybrid model

The hybrid model (Sartori et al., 2014) uses ID joint torque to adjust muscle excitations in order to produce realistic joint torques. The objective function optimized by the optimization algorithm can be seen in equation 1.

$$f_{obj} = \alpha E_{trackMOM} + \beta E_{EXC} + \gamma E_{trackEMG} \tag{1}$$

The objective function minimizes three terms. The first term contains $E_{trackMOM}$, the sum of squared differences between predicted and ID joint torque, and ensures that predicted excitations produce realistic joint torques. The second term contains E_{EXC}, the sum of squared excitations, and ensures that muscle excitations remain low. The third term contains $E_{trackEMG}$, the sum of absolute differences between measured and adjusted excitations, and ensures that adjusted excitations remain close to measured excitations. In this study, $\alpha = 1$, $\beta = 15$ and $\gamma = 500$. The hybrid model enables measurement of excitations that account for realistic joint torques. Furthermore, excitations of deep muscles can be obtained.

2.2 XML file

To pass certain parameters onto the plugin, an Extensible Markup Language (XML) file was created in which these parameters can be given as input to the model. This file should be specified in the command line when running the software. The software contains an XML Schema Definition (XSD) file that describes how elements should be described in the XML file to be readable by the software. From this XSD file, the XML file in appendix A was created. It contains an element for the hybrid model, which contains weighting parameters α, β and γ from equation 1. Furthermore, tracked and predicted muscles can be specified. Tracked muscles are the muscles that were measured and should be optimized. Predicted muscles are the (deep) muscles that were not measured and should be optimized.

Many muscles are biarticular. This biarticularity of muscles increases the complexity of NMS modeling and thus increases computation time of the hybrid model. Therefore, the possibility to choose which DOFs to optimize was added in this study. This can be specified in the *DOFsOptimized* element. For this purpose the XSD file was changed, as well as the files containing classes that read data from the XML file, compare the data to the XSD file and pass the data to the optimization plugin.

2.3 Optimization plugin

Initialization

The online optimization plugin (figure 3) is written in C++ as an extension to the open-loop model written by Durandau et al. (2018). During start-up, the optimization plugin is initialized. In this initialization, the NMS model core class is set up in the plugin. This includes reading the muscles and DOFs from a file, including their calibrated parameters. Then, the XML file parameters are loaded to the plugin in order to obtain the weighting parameters of equation 1 and which muscles and DOFs to include in the optimization. Then, the loggers are initialized which output certain signals (such as EMG, LMT and torque) to their output files. Finally, the DOFs that are not optimized are erased from the NMS model. For this, a function was written that erases those DOFs from the NMS model. This function can be seen in appendix B.

Loop

During the loop, the open-loop model transmits time, EMG, ID torque, MA and LMT data to the optimization plugin. Time and EMG data are received from the EMG plugin at one frequency. ID torque, MA and LMT data are received from the IK-ID plugin at another frequency. When data is received from the IK-ID plugin, one optimization instance is started before waiting until new LMT, MA and ID data are transmitted. Then, the next optimization instance is started. Within one optimization instance ID torque, time, EMG, LMT and MA data are set to the NMS model from the optimization plugin before starting the actual optimization algorithm, which will be explained in section 2.5.

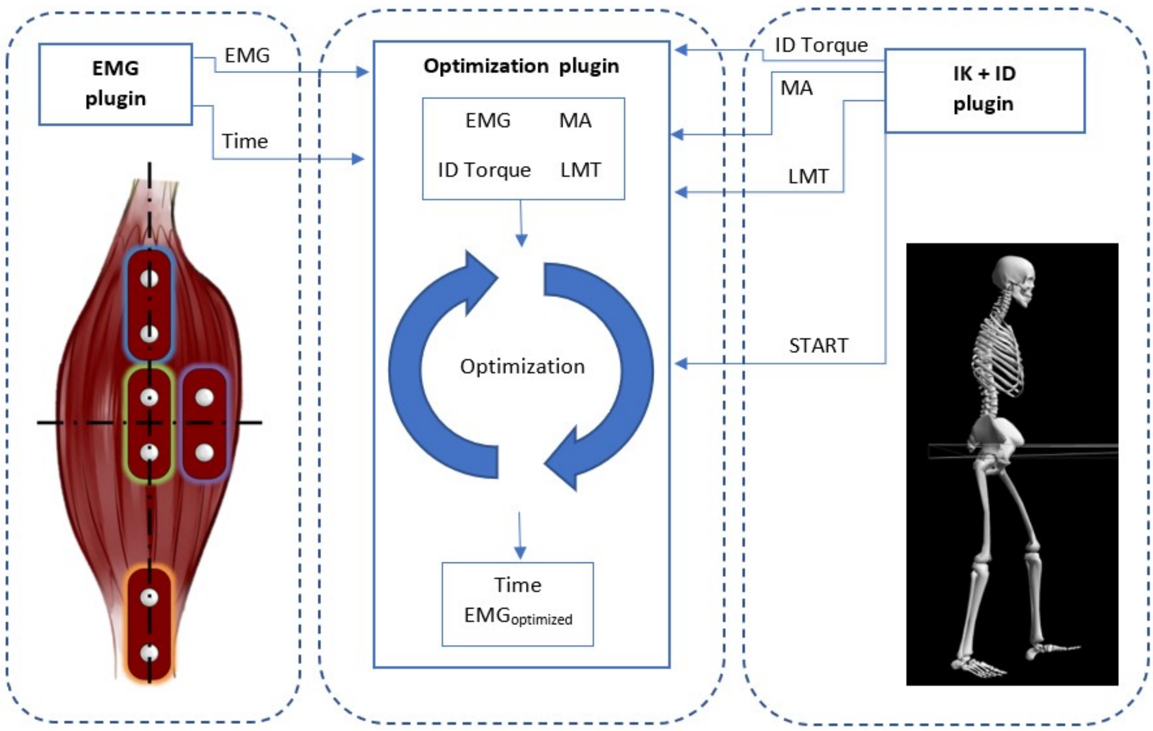


Figure 3: Schematics of the plugin working mechanism. The EMG plugin receives EMG data and transmits these to the optimization plugin. The IK-ID plugin receives IK and ID data and transmits these to the optimization plugin. The EMG and IK-ID plugins transmit data on different frequencies. When data from the IK-ID plugin are transmitted to the optimization plugin, one optimization instance is started. This optimization instance executes the hybrid model block diagram from figure 2 The next optimization instance is started if the previous optimization instance is finished, only if new data are received from the IK-ID.

Finally, data are sent to the loggers to be saved to the output files. Complete code of the optimization plugin can be found in appendix appendix C.

Offline optimization

For some applications, it may be useful to do the optimization in an offline way. For example, if you want to compare the real-time optimization to a slower offline optimization. Running the slow optimization would currently lead to the discard of many input data values, because the next optimization instance is started only when the previous optimization instance is finished. This would lead to a loss of data. To enable offline optimization in the current framework, a buffer was created, which remembers all input data points. Also, the plugin is not shut down after execution of the open-loop model to enable the plugin to finish the optimization of all data points. This buffer can be used by hardcoding a few changes in the software. These changes are highlighted in appendix C using the comment 'Comment out for buffer'.

2.4 Bug fixes

Some bugs in the open-loop model were exposed when running the optimization plugin. The bugs that needed to be fixed are described in this subsection.

Read data from file

As described in the introduction of this section, data can be measured in real-time or read from file. This can be different for the EMG and IK-ID plugin. For both plugins, it can be specified in the XML file of the open-loop model which real-time measurement plugin to use. When a read-from-file plugin must be used, this should be specified in the command line when running the application, including the directory in which these data files can be found. For the IK-ID plugin there also was a choice between reading only the IK angle data or both the IK angle and the ID data.

The IK-ID plugin where both angle and ID data were read from file included the header file from the IK-ID plugin where only the angle data were read from file, and therefore building the application failed. After including the right header file the application built successfully.

The read-from-file IK-ID plugin automatically loaded the plugin where only the angle data were read from file, and therefore the ID data could not be read from file. This was changed in such a way that if an ID file was present, IK and ID data are read from file and if no ID file was present, only IK data are read from file.

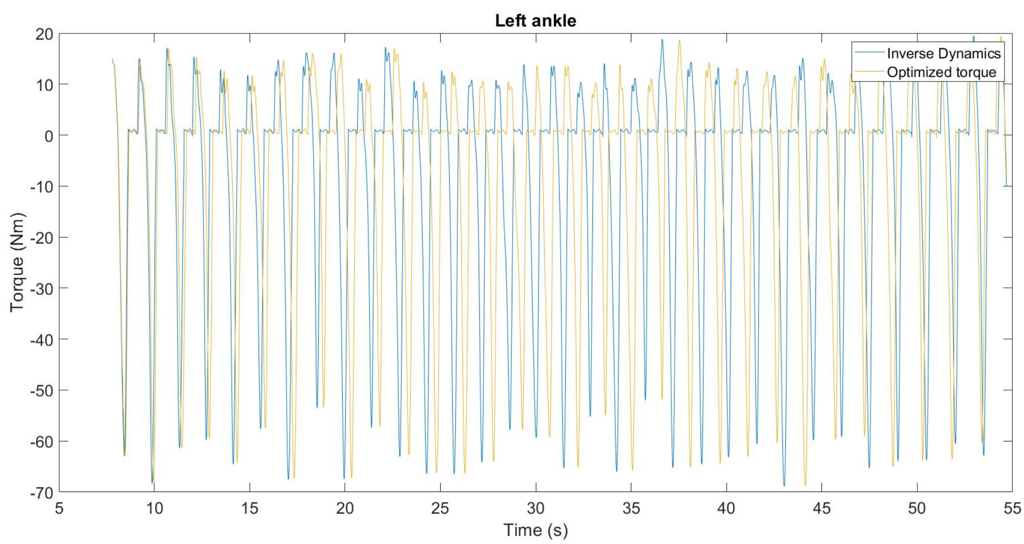


Figure 4: It can be seen that the optimized torque develops a delay compared to the ID torque during execution. This is weird, because the ID torque is used to optimize the torque. Therefore, the period of the signals should be similar.

ID torque

When running the application, the ID torque sent to the plugin by the IK-ID plugin remains doesn't change during the execution of the application. It is found that the variable which holds the ID torque was declared twice in the IK-ID plugin, of which one declaration was used in an if-statement. Therefore, these declarations were regarded as different variables. The second declaration was updated with the datum that was read from file (inside the if-statement), while the first declaration was sent to the optimization plugin (outside the if-statement). The only exception was for the first time frame. Therefore, the ID torque sent to the plugin by the IK-ID plugin remained constant with the first ID torque value. When the second statement was deleted, the ID torque sent to the plugin did change its value during execution.

However, it can be seen from figure 4 that the optimized torque develops a delay compared to the ID torque during execution. This is weird, because the optimized torque is optimized (among others) with respect to the ID torque, so although errors may exist, the period of the signal should be similar. It was found that the ID torque values that are read from the ID torque file are put into a buffer, out of which the IK-ID plugin takes ID torque data to use in the software. The data is put into the buffer at the back and it is received from the buffer at the front. For the hybrid plugin this buffer is problematic, so it was changed such that the ID torque is both put into and received from the back of the buffer. Figure 5 shows that the delay disappeared.

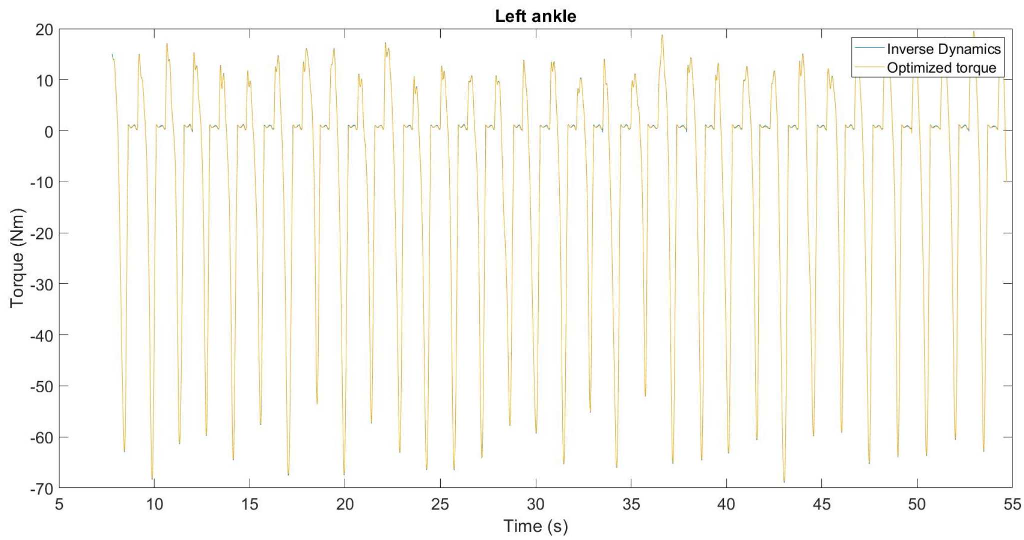


Figure 5: After changing the way in which torque data is read from the buffer, the delay issue is fixed.

2.5 Pagmo

Pagmo was used for the implementation of the optimization (Biscani and Izzo, 2020). Pagmo is an open-source library which can be used for parallel optimization. It supports various algorithms that can be used to solve optimization problems.

Linking pagmo

To use the pagmo library in the optimization plugin, it was installed and linked to the optimization plugin library. First of all, the pagmo library source code (version 2.15.0[1]) was downloaded from https://github.com/esa/pagmo2/releases. The pagmo library requires three other libraries: Boost, Intel TBB and Eigen3. Boost is already used by the plugin. The TBB library (version 2020.3) was downloaded from https://github.com/oneapi-src/oneTBB/releases/tag/v2020.3 and the Eigen3 library was downloaded from (version 3.3.8) from http://eigen.tuxfamily.org/index.php?title=Main_Page#Download and installed. Then, the following paths were added to the PATH environment variable (to enable the pagmo library to find the TBB library):

- C:\PATH\TO\TBBFOLDER\tbb\lib\intel64\vc14
- C:\PATH\TO\TBBFOLDER\tbb\bin\intel64\vc14

Also, the TBB and Eigen3 directories were set in the CMakeLists file from pagmo. Then, pagmo was built and installed.

In the CMakeLists file from the optimisation plugin, the pagmo library was searched for using the *find_package* statement. Then, the pagmo library was linked to the optimization plugin.

Pagmo problem

A user-defined problem (UDP) was created which could be solved by various algorithms provided by the pagmo library. The code of the UDP can be found in appendix D. A UDP is a class that contains at least the *fitness(EMG)* and *get_bounds()* functions. The former evaluates the objective function (equation 1) for a given set of EMGs and returns the objective function value. The latter is used to provide the bounds for the the EMGs. In this study, the UDP also contains the subject core class to which data is set before evaluating the objective function as described in section 2.3. The actual optimization is done in the *evalfp()* function. First, torques are received from (and calculated by) the *staticComputation_* object, which is a class that is used to remember all variables that are necessary to evaluate the objective function, such as the EMG values before and after adjustment and predicted torque values. Then, the first term of the objective function is calculated. After that, initial and adjusted EMG values of the muscles of which EMG is measured are received from the *staticComputation_* object to calculate the third term of the objective function. Finally, the second term of the objective function is evaluated and the objective function value is calculated.

[1]From 2.16.0 onwards, pagmo requires a compiler which is able to understand at least C++17. However, XSD offers no support for C++17, so using C++17 causes build errors with the XSD. Until XSD has a new release with support for C++17, version 2.15.0 of pagmo should be used.

3 Experimental analysis

Experimental data were used to analyse the ability of the optimization plugin to optimize in real-time and validate the output of the model. In this section, it is described how experimental data was collected, which DOFs were considered for optimization and which algorithms were compared to each other. It is concluded with a description of the procedure that is used to validate the output.

3.1 Data collection

GRF data were recorded from one subject while walking with 1.8 km/h on a split-belt treadmill (Motek Forcelink, the Netherlands) with a sampling frequency of 2 Khz. Kinematics were captured with a motion capture system (Qualisys, Sweden) with a sampling frequency of 128 Hz.

Subject's EMGs were measured using an EMG amplifier (Delsys Bagnoli System, USA) with a sampling frequency of 2 Khz. Raw EMG signals were first high-pass filtered with a cut-off frequency of 20 Hz. Then, they were full-wave rectified followed by a low-pass filter with a cut-off frequency of 6 Hz. They were finally normalized with the EMG values during maximum voluntary contraction, which were collected offline before the measurement. Resulting muscle excitations were used as input to the model, as well as torque calculated by ID.

In section section 2.3 it was described that a new optimization instance is started when new LMT, MA and ID data are received by the plugin. These data are received at 64 Hz[2], so the goal is to get a computation time of $\frac{1}{64} = 0.015625$ seconds. Subject-specific parameters from the EMG-driven forward dynamics block were determined by a calibration of the model. During this calibration, only the EMG-driven forward dynamics block was used to calculate predicted joint torque, which was compared to the ID joint torque in order to determine the subject-specific parameters.

3.2 DOF choice

In this study, it was chosen to only optimize the ankle joint in order to decrease computation time. This limits the complexity of the problem and therefore the computation time. The muscles spanning the ankle joint are soleus (SO), gastrocnemius medialis (GM), gastrocnemius lateralis (GL) and tibialis anterior (TA).

[2]This should be 128 Hz, but there is a bug in the IK-ID plugin. Each time one ID torque value should be read, it actually reads two values and discards the first one. Therefore, the information is written to the plugin at 64 Hz instead of 128 Hz. This was discovered when writing the report, so it could not be changed anymore.

3.3 Algorithms

Three algorithms were tested on their ability to meet real-time requirements. The first algorithm is simulated annealing (Corana et al., 1987). This algorithm is reliable in finding the global optimum (as opposed to a local optimum) because of its ability to make uphill moves. However, it is a costly algorithm and its sequential nature makes parallelization of single objective function evaluations impossible. The other algorithms are PSO_{gen} (Poli et al., 2007) and CMAES (Hansen, 2006). These are population-based algorithms, so single objective function evaluations can be parallelized, decreasing computation time. Several tests were done for the PSO_{gen} and CMAES algorithms with different values for population size and number of generations.

Muscle excitation ranges from 0 to 1. However, it costs valuable computation time to search the whole field, while excitations don't change instantaneously. Information from the previous excitation value can be used to set the boundaries for searching the current excitation value. From EMG measurements it was checked that the maximum change within 15.625 ms was 0.22. Therefore, the boundaries for the optimization problem were in the range of:

$$(EMG_{past} - 0.22) \leq EMG_{current} \leq (EMG_{past} + 0.22)$$

Optimization of CMAES and PSO_{gen} involves a large population of individuals whose initial values are randomly assigned within the boundaries. The simulated annealing optimization involves only one individual, whose value was initialized with EMG_{past}.

3.4 Validation

The ability from each algorithm to work in real-time was assessed using the mean value of the objective function and the mean computation time including their standard deviations.

Results from the online optimization were compared to the results from an offline optimization using simulated annealing. This algorithm used different parameters that made it slow, but likely to find the global optimum. Therefore we can assume that the global optimum of the optimization problem was found. Predicted joint torque and adjusted excitations were validated using Pearson's r and mean absolute error.

4 Results

Figure 6 shows the computation time density distribution for each algorithm including their 95% confidence interval, mean and median computation time values. This distribution was obtained by fitting a Kernel density estimation to the computation time data.

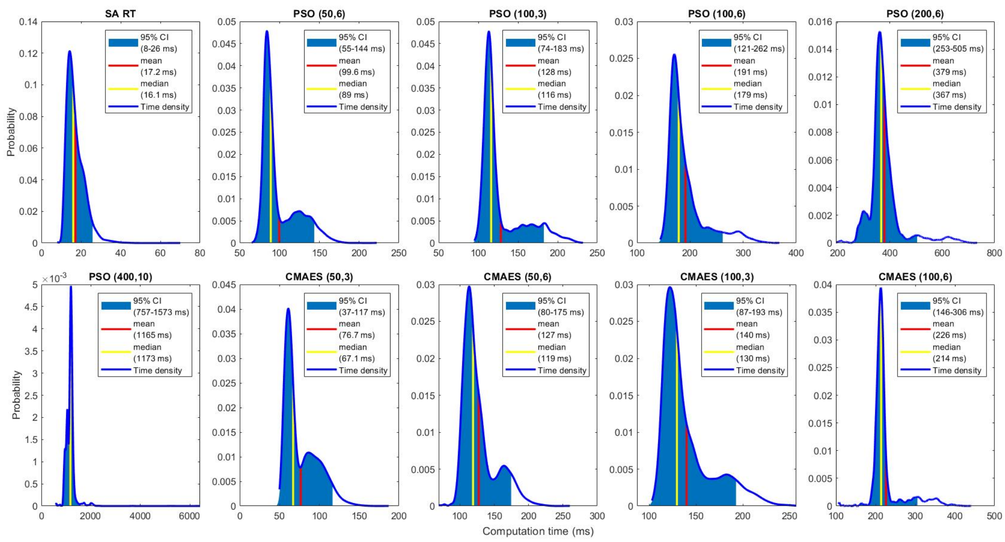


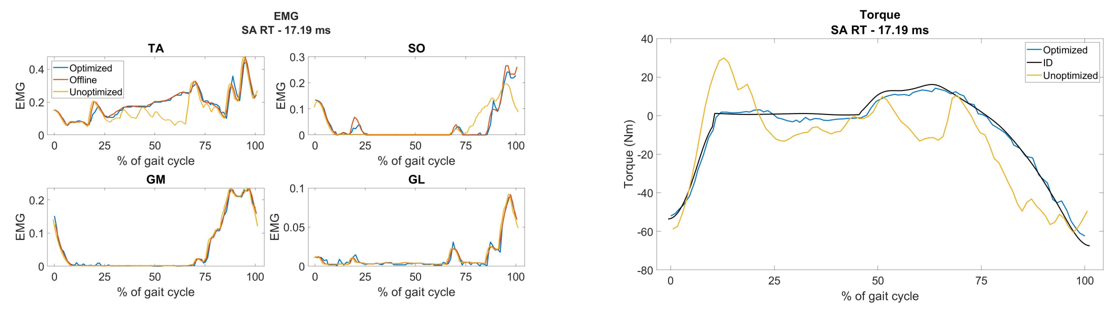
Figure 6: For each algorithm, computation time density is shown, including their 95% confidence interval (blue), mean (red) and median (yellow). Probability density distribution was obtained by fitting a Kernel distribution to the computation time data.

With the SA algorithm, mean objective function value calculated by equation 1 was 48.13 ± 32.05. The mean computation time was 17.19 ± 4.53 ms. The PSO_{gen} algorithm obtained a better mean objective function value of 46.31 ± 30.61 when using 400 individuals and 10 generations. Mean computation time for this algorithm was 1165 ± 208.24 ms. When using 50 individuals and 6 generations, a mean computation time of 99.60 ± 22.86 ms was obtained, resulting in a mean objective function value of 55.03 ± 31.14. The CMAES algorithm resulted in mean objective function value of 47.14 ± 30.78 when using 100 individuals and 6 generations, which had a mean computation time of 226.21 ± 40.71. Faster results were obtained with 50 individuals and 6 generations with a mean computation time of 127.35 ± 24.23 ms. Mean objective function value for this algorithm was 51.42 ± 30.78. All results are presented in table 1.

Table 1: The mean objective function values f and time (including standard deviations) for the three algorithms with several parameters. Population size is the number of individuals used in the population. Simulated annealing is not a population-based algorithm, so it contains 1 individual. PSO_{gen} and CMAES are population-based algorithms, so their performance and computation time depend on the population size. Also the number of generations gen is varied for those algorithms.

Algorithm	Pop size	Gen	f	Time (ms)
$SA_{offline}$	1	-	43.66 ± 31.03	861.49 ± 73.46
SA_{RT}	1	-	48.13 ± 32.05	17.19 ± 4.53
PSO_{gen}	400	10	46.31 ± 30.61	1165 ± 208.24
PSO_{gen}	200	6	49.53 ± 30.41	378.88 ± 64.47
PSO_{gen}	100	6	51.49 ± 30.50	191.39 ± 36.02
PSO_{gen}	100	3	65.86 ± 33.69	128.34 ± 27.64
PSO_{gen}	50	6	55.03 ± 31.14	99.60 ± 22.86
CMAES	100	6	47.14 ± 30.78	226.21 ± 40.71
CMAES	100	3	54.82 ± 30.91	140.00 ± 27.29
CMAES	50	6	51.42 ± 30.78	127.35 ± 24.23
CMAES	50	3	65.91 ± 34.68	76.68 ± 20.36

Figure 7 shows plots of EMG and torque for the real-time EMG model using SA. Optimized EMG using the real-time algorithm is compared to optimized EMG using the offline SA algorithm and unoptimized EMG for TA, SO, GM and GL muscles. Although validation was done using data from multiple gait cycles, only one gait cycle is shown in figures 7 to 9 for clarity. Begin and end of the gait cycle were determined by the low peak in torque calculated by ID. It can be seen that excitation and torque values are close to those values for the offline SA algorithm.



(a) EMG plot for the real-time SA algorithm (blue) compared to optimized EMG using the offline SA algorithm (red) and unoptimized EMG (orange) for each muscle.

(b) Torque plot for the real-time SA algorithm (blue) compared to ID torque (black) and torque calculated from unoptimized EMG (orange).

Figure 7: EMG (a) and torque (b) plots for the real-time SA algorithm.

Figure 8 shows the same plots of EMG and torque for the CMAES algorithms. It can be seen that for slow CMAES algorithms excitation and torque values are close to those values for the offline SA algorithm. However, for faster CMAES algorithms, the number of times that the global minimum can not be found increases.

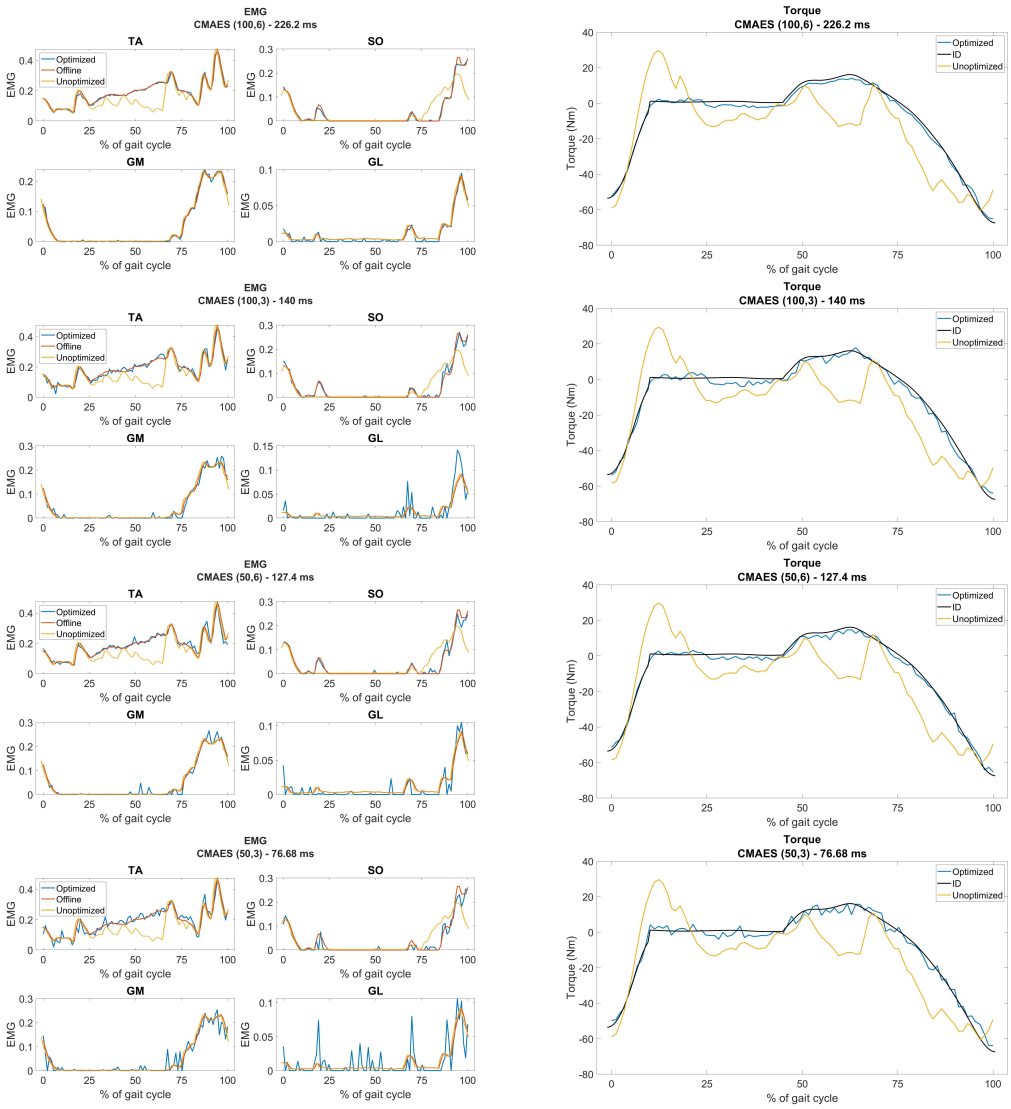


Figure 8: Plots of EMG (left) and torque (right) for the CMAES algorithm with different parameters (rows). The values between brackets represent population size and number of generations, respectively. Begin and end of the gait cycle were determined by the low peak in torque calculated by ID. Optimized EMG (blue) is compared to optimized EMG using the offline SA algorithm (red) and unoptimized (measured) EMG (orange) for tibialis anterior (TA), soleus (SO), gastrocnemius medialis (GM) and gastrocnemius lateralis (GL) muscles. Optimized torque (blue) is compared to ID torque (black) and torque calculated from unoptimized (measured) EMG (orange).

The same plots for the PSO$_{gen}$ algorithm are shown in figure figure 9. As with the CMAES algorithm, excitation and torque values are close to those values for the offline SA algorithm for slow PSO algorithms, but with faster PSO algorithms the similarity decreases.

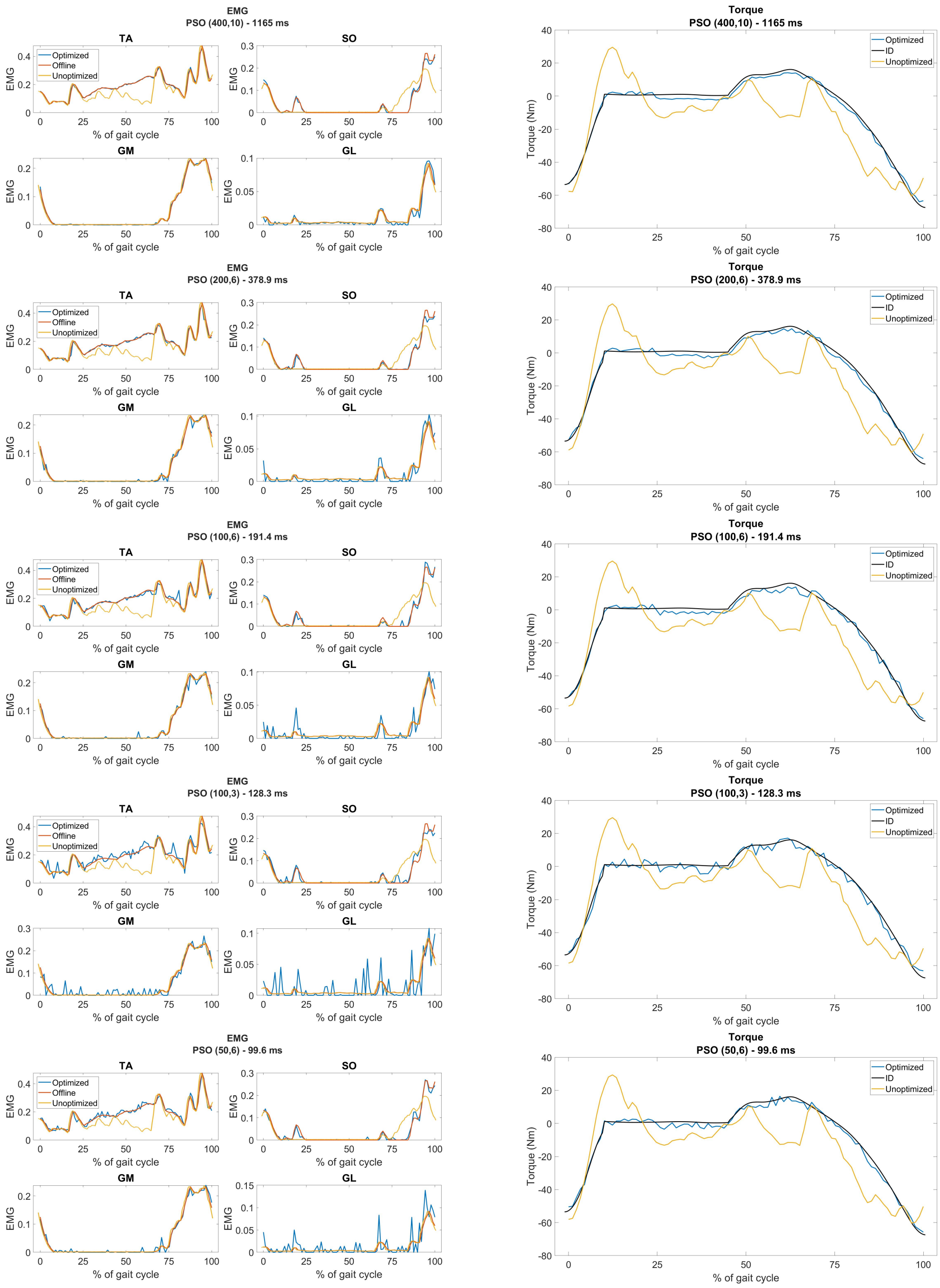


Figure 9: Plots of EMG (left) and torque (right) for the PSO$_{gen}$ algorithm with different parameters (rows). The values between brackets represent population size and number of generations, respectively. Begin and end of the gait cycle were determined by the low peak in torque calculated by ID. Optimized EMG (blue) is compared to optimized EMG using the offline SA algorithm (red) and unoptimized (measured) EMG (orange) for tibialis anterior (TA), soleus (SO), gastrocnemius medialis (GM) and gastrocnemius lateralis (GL) muscles. Optimized torque (blue) is compared to ID torque (black) and torque calculated from unoptimized (measured) EMG (orange).

Pearson's r and mean absolute error for each algorithm compared to the offline SA algorithm are shown in table 2.

Table 2: Pearson's correlation value r and mean absolute error (MAE) and standard deviation for each algorithm (Alg). Population size P and number of generations G are specified for the PSO_{gen} and CMAES algorithms. r and MAE are shown for each muscle as well as the mean value for the muscles and the value for the torque.

Alg		**r**	**MAE**	**Alg**		**r**	**MAE**
SA_{RT}	TA	0.98	0.01 ± 0.01	PSO_{gen}	TA	1.00	0.004 ± 0.01
	SO	0.98	0.005 ± 0.01		SO	1.00	0.002 ± 0.004
	GM	1.00	0.003 ± 0.004	P = 400	GM	1.00	0.002 ± 0.003
	GL	0.98	0.002 ± 0.003	G = 10	GL	0.98	0.002 ± 0.003
	Mean muscles	0.98	0.005 ± 0.009		Mean muscles	0.99	0.003 ± 0.004
	Torque	1.00	1.95 Nm ± 1.22		Torque	1.00	1.93 Nm ± 0.88
CMAES	TA	0.99	0.01 ± 0.01	PSO_{gen}	TA	0.98	0.01 ± 0.01
	SO	1.00	0.003 ± 0.005		SO	0.99	0.004 ± 0.01
P = 100	GM	1.00	0.003 ± 0.004	P = 200	GM	1.00	0.004 ± 0.005
G = 6	GL	0.98	0.003 ± 0.003	G = 6	GL	0.94	0.004 ± 0.005
	Mean muscles	0.99	0.004 ± 0.01		Mean muscles	0.98	0.005 ± 0.01
	Torque	1.00	1.96 Nm ± 0.97		Torque	1.00	1.93 Nm ± 1.03
CMAES	TA	0.97	0.01 ± 0.01	PSO_{gen}	TA	0.98	0.01 ± 0.01
	SO	0.99	0.005 ± 0.01		SO	0.99	0.004 ± 0.01
P = 100	GM	0.99	0.01 ± 0.01	P = 100	GM	0.99	0.005 ± 0.01
G = 3	GL	0.85	0.01 ± 0.01	G = 6	GL	0.89	0.005 ± 0.01
	Mean muscles	0.95	0.01 ± 0.01		Mean muscles	0.97	0.01 ± 0.01
	Torque	1.00	2.13 Nm ± 1.39		Torque	1.00	1.97 Nm ± 1.16
CMAES	TA	0.98	0.01 ± 0.01	PSO_{gen}	TA	0.93	0.02 ± 0.02
	SO	0.99	0.004 ± 0.01		SO	0.98	0.01 ± 0.01
P = 50	GM	0.99	0.005 ± 0.01	P = 100	GM	0.97	0.01 ± 0.02
G = 6	GL	0.91	0.01 ± 0.01	G = 3	GL	0.66	0.01 ± 0.02
	Mean muscles	0.97	0.01 ± 0.01		Mean muscles	0.88	0.01 ± 0.02
	Torque	1.00	2.01 Nm ± 1.21		Torque	0.99	2.22 Nm ± 1.62
CMAES	TA	0.93	0.02 ± 0.02	PSO_{gen}	TA	0.97	0.01 ± 0.01
	SO	0.97	0.01 ± 0.01		SO	0.99	0.005 ± 0.01
P = 50	GM	0.96	0.01 ± 0.02	P = 50	GM	0.99	0.01 ± 0.01
G = 3	GL	0.62	0.01 ± 0.02	G = 6	GL	0.84	0.01 ± 0.01
	Mean muscles	0.87	0.01 ± 0.02		Mean muscles	0.94	0.01 ± 0.01
	Torque	0.99	2.34 Nm ± 1.69		Torque	1.00	2.09 Nm ± 1.33

5 Discussion

A real-time hybrid NMS model was developed that overcomes torque errors in open-loop EMG-based NMS modeling due to limitations in EMG measurements (such as cross-talk, movement artefacts and the weak association between EMG amplitude and motor unit action potentials) and model imperfections. The hybrid model from Sartori et al. (2014) is adapted such that it also works in real-time. It is implemented as an extension to the real-time open-loop NMS model from Durandau et al. (2018). This model works in real-time with a frequency of 64 Hz and therefore enables online measurement of muscle excitations, forces and joint torques, which can be used in experiments for biofeedback or in a clinical environment to obtain insight in impairments of a patient's NMS system.

Several optimizations were done in order to compare SA to population-based algorithm PSO_{gen}. It can be seen from table 1 and table 2 that SA with real-time parameters performs similar to PSO_{gen} and CMAES even with population sizes of 100 individuals and 6 generations, where computation time is too large to work in real-time. This means that it will not benefit from multithreading, because even when PSO_{gen} or CMAES are multithreaded to work in real-time, they will not or hardly perform better than SA.

For this study, it was chosen to use the input frequency of the IK and ID plugins as a goal for the optimization frequency. A higher frequency can be obtained when the optimization is performed each time a new EMG datum is obtained. For this to work, computation time of the optimization must be decreased. With SA, the only possibility is to change the parameters, which will decrease the precision of the optimization. With PSO_{gen} and CMAES algorithms, similar precision can be maintained with a multithreaded environment. Therefore, PSO_{gen} and CMAES have the ability to provide similar precision at a higher frequency.

This study focused on the ankle joint only. When optimizing for multiple joints, computation time increases. Calculation of muscle activations, fibre kinematics, muscle forces and joint torques can be multithreaded to counteract this. For one joint with four muscles as in this study, this will not make a big difference, but for modeling the lower extremities during walking this will have a large effect.

Joint loads (Kinney et al., 2013) and joint stiffness (Sartori et al., 2015) are currently measured offline in a clinical environment. The real-time hybrid NMS model that was developed in this study can potentially be used to obtain online measurements of these human movement data, or of other data such as EMG, muscle stiffness or joint torque.

The hybrid optimization of EMG measurements enables online measurement of EMG signals that produce realistic joint torques as calculated by ID. This real-time adjusted EMG signal can be used to drive an exoskeleton, which would not be possible with offline EMG measurements.

Bibliography

Biscani, F. and Izzo, D. (2020). A parallel global multiobjective framework for optimization: pagmo. *Journal of Open Source Software*, 5(53):2338. Available from: https://doi.org/10.21105/joss.02338, doi:10.21105/joss.02338.

Corana, A., Marchesi, M., Martini, C., and Ridella, S. (1987). Minimizing multimodal functions of continuous variables with the "simulated annealing" algorithm—corrigenda for this article is available here. *ACM Trans. Math. Softw.*, 13(3):262–280. Available from: https://doi.org/10.1145/29380.29864, doi:10.1145/29380.29864.

Delp, S., Anderson, F., Arnold, A., Loan, P., Habib, A., John, C., Guendelman, E., and Thelen, D. (2007). Opensim: Open-source software to create and analyze dynamic simulations of movement. *Biomedical Engineering, IEEE Transactions on*, 54:1940 – 1950. doi:10.1109/TBME.2007.901024.

Durandau, G., Farina, D., and Asín-Prieto, G. (2019). Voluntary control of wearable robotic exoskeletons by patients with paresis via neuromechanical modeling. *J NeuroEngineering Rehabil*, 16(91).

Durandau, G., Farina, D., and Sartori, M. (2018). Robust real-time musculoskeletal modeling driven by electromyograms. *IEEE Transactions on Biomedical Engineering*, 65(3):556–564.

Erdemir, A., McLean, S., Herzog, W., and van den Bogert, A. J. (2007). Model-based estimation of muscle forces exerted during movements. *Clinical Biomechanics*, 22(2):131 – 154. Available from: http://www.sciencedirect.com/science/article/pii/S0268003306001835, doi:https://doi.org/10.1016/j.clinbiomech.2006.09.005.

Farina, D. and Negro, F. (2012). Accessing the neural drive to muscle and translation to neurorehabilitation technologies. *IEEE Reviews in Biomedical Engineering*, 5:3–14.

Fregly, B., Boninger, M., and Reinkensmeyer, D. (2012). Personalized neuromusculoskeletal modeling to improve treatment of mobility impairments: A perspective from european research sites. *Journal of neuroengineering and rehabilitation*, 9:18. doi:10.1186/1743-0003-9-18.

Hansen, N. (2006). *The CMA Evolution Strategy: A Comparing Review*, Pp. 75–102. Berlin, Heidelberg: Springer Berlin Heidelberg. Available from: https://doi.org/10.1007/3-540-32494-1_4, doi:10.1007/3-540-32494-1_4.

Hill, A. V. (1938). The heat of shortening and the dynamic constants of muscle. *Proceedings of the Royal Society of London. Series B - Biological Sciences*, 126(843):136–195. Available from: https://royalsocietypublishing.org/doi/abs/10.1098/rspb.1938.0050, doi:10.1098/rspb.1938.0050.

Kinney, A. L., Besier, T. F., D'Lima, D. D., and Fregly, B. J. (2013). Update on Grand Challenge Competition to Predict in Vivo Knee Loads. *Journal of Biomechanical Engineering*, 135(2). 021012. Available from: https://doi.org/10.1115/1.4023255, doi:10.1115/1.4023255.

Pizzolato, C., Reggiani, M., Saxby, D., Ceseracciu, E., Modenese, L., and Lloyd, D. (2017). Biofeedback for gait retraining based on real-time estimation of tibiofemoral joint contact forces. *IEEE Transactions on Neural Systems and Rehabilitation Engineering*, PP:1–1. doi:10.1109/TNSRE.2017.2683488.

Poli, R., Kennedy, J., and Blackwell, T. (2007). Particle swarm optimization: An overview. *Swarm Intelligence*, 1. doi:10.1007/s11721-007-0002-0.

Sartori, M., Durandau, G., Dosen, S., and Farina, D. (2018). Robust simultaneous myoelectric control of multiple degrees of freedom in wrist-hand prostheses by real-time neuromusculoskeletal modeling. *Journal of Neural Engineering*, 15(6). doi:10.1088/1741-2552/aae26b.

Sartori, M., Farina, D., and Lloyd, D. G. (2014). Hybrid neuromusculoskeletal modeling to best track joint moments using a balance between muscle excitations derived from electromyograms and optimization. *Journal of Biomechanics*, 47(15):3613 – 3621. Available from: http://www.sciencedirect.com/science/article/pii/S0021929014005284, doi:https://doi.org/10.1016/j.jbiomech.2014.10.009.

Sartori, M., Maculan, M., Pizzolato, C., Reggiani, M., and Farina, D. (2015). Modeling and simulating the neuromuscular mechanisms regulating ankle and knee joint stiffness during human locomotion. *Journal of Neurophysiology*, 114(4):2509–2527. PMID: 26245321. Available from: https://doi.org/10.1152/jn.00989.2014, doi:10.1152/jn.00989.2014.

Sartori, M., Reggiani, M., Farina, D., and Lloyd, D. G. (2012). Emg-driven forward-dynamic estimation of muscle force and joint moment about multiple degrees of freedom in the human lower extremity. *PLOS ONE*, 7(12):1–11. Available from: https://doi.org/10.1371/journal.pone.0052618, doi:10.1371/journal.pone.0052618.

Seth, A., Hicks, J. L., Uchida, T. K., Habib, A., Dembia, C. L., Dunne, J. J., Ong, C. F., DeMers, M. S., Rajagopal, A., Millard, M., Hamner, S. R., Arnold, E. M., Yong, J. R., Lakshmikanth, S. K., Sherman, M. A., Ku, J. P., and Delp, S. L. (2018). Opensim: Simulating musculoskeletal dynamics and neuromuscular control to study human and animal movement. *PLOS Computational Biology*, 14(7):1–20. Available from: https://doi.org/10.1371/journal.pcbi.1006223, doi:10.1371/journal.pcbi.1006223.

Appendix

A. XML file

```
<?xml version="1.0"?>
<optimization xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:noNamespaceSchemaLocation="../../XSD/executionOptimization.xsd">

        <Hybrid>
                <alpha>1</alpha>
                <beta>15</beta>
                <gamma>500</gamma>
                <trackedMuscles>med_gas_l lat_gas_l soleus_l tib_ant_l</trackedMuscles>
                <predictedMuscles></predictedMuscles>
                <DOFsOptimized>ankle_angle_l</DOFsOptimized>
        </Hybrid>

</optimization>
```

B. Erase unused DOFs

```
template <typename Activation, typename Tendon, CurveMode::Mode mode>
void NMSmodel<Activation, Tendon, mode>::eraseUnusedDofs(const std::vector<std::string>& dofNamesUsed, std::vector<unsigned int>& dofsNotErased) {
  auto dofNameIt = dofNames_.begin();     // Iterator through the DOF names
  auto dofIt = dofs_.begin();             // Iterator through the DOFs
  dofsNotErased.clear();                  // dofsNotErased remembers which DOFs of the model are not erased. These can be used for using the right MA and ID torque data
  unsigned int i{ 0 };
  while (dofNameIt != dofNames_.end())    // loop through all DOFs in the model
  {
    if (std::find(dofNamesUsed.begin(), dofNamesUsed.end(), *dofNameIt) == dofNamesUsed.end())    // if the DOF is not used: erase the DOF
    {
      dofIt = dofs_.erase(dofIt);
      dofNameIt = dofNames_.erase(dofNameIt);
    }
    else                          // else: increase iterators
    {
      ++dofNameIt;
      ++dofIt;
      dofsNotErased.push_back(i);        // now we know which DOFs are not erased, so we know which MAs to use
    }
```

```
        i++;
    }
}
```

C. Optimization plugin

Header files that are included were hidden for readability.

OptimizationHybridPlugin.h

```
using namespace Hybrid;
using namespace pagmo;

#ifdef WIN32
template <typename NMSmodelT>
class __declspec(dllexport) OptimizationHybridPlugin : public
    OptimizationPlugin<NMSmodelT>
#endif
#ifdef UNIX
    template <typename NMSmodelT>
class OptimizationHybridPlugin : public OptimizationPlugin<NMSmodelT>
#endif
{
public:
    OptimizationHybridPlugin();
    ~OptimizationHybridPlugin();
    void init(NMSmodelT& model, const std::string& executionXMLFileName,
        const std::string& configurationFileName);
    void newData();

    std::vector<double> getDofTorque();
    std::vector<double> getShapeFactor();
    std::vector<double> getTendonSlackLengths();
    std::vector<double> getOptimalFiberLengths();
    std::vector<double> getGroupMusclesBasedOnStrengthCoefficients();

    void setLmt(const std::vector<double>& lmt);
    void setMA(const std::vector<std::vector<double> >& ma);
    void setMuscleForce(const std::vector<double>& muscleForce);
    void setDOFTorque(const std::vector<double>& dofTorque); // not used
    void setExternalDOFTorque(const std::vector<double>& externalDOFTorque);
    void setActivations(const std::vector<double>& activations);
    void setFibreLengths(const std::vector<double>& fibreLengths);
    void setFibreVelocities(const std::vector<double>& fibreVelocities);
    void setPennationAngle(const std::vector<double>& pennationAngle);
    void setTendonLength(const std::vector<double>& tendonLength);
    void setTime(const double& time);
    void setEmgs(const std::vector<double>& Emgs);
    //void setBuffer(const double& time, const std::vector<double>& Emgs,
        const std::vector<double>& externalDOFTorque, const std::vector<
        double>& lmt, const std::vector<std::vector<double> >& ma);
    void setJointAngle(const std::vector<double>& jointAngle) {}
```

```

    void start();
    void stop();
    void setDirectories(const std::string& outDirectory, const std::string&
        inDirectory = std::string());
    void setVerbose(const int& verbose);
    void setRecord(const bool& record);

protected:

    void optimization();

    void setupSubject(NMSmodelT& mySubject, string configurationFile);

    NMSmodelT* model_;
    ErrorMinimizerAnnealing<NMSmodelT>* torqueErrorMinimizer_;
    std::vector<std::string> dofName_;
    std::vector<std::string> muscleName_;

    std::vector<double> shapeFactor_;
    std::vector<double> tendonSlackLengths_;
    std::vector<double> optimalFiberLengths_;
    std::vector<double> groupMusclesBasedOnStrengthCoefficients_;

    std::vector<std::vector<double> > ma_;
    std::vector<double> lmt_;
    std::vector<double> muscleForce_;
    std::vector<double> dofTorque_; // not used
    std::vector<double> externalDOFTorque_;
    std::vector<double> activations_;
    std::vector<double> fibreLengths_;
    std::vector<double> fibreVelocities_;
    std::vector<double> pennationAngle_;
    std::vector<double> tendonLength_;
    std::vector<double> Emgs_;
    double time_;
    std::list<double> timeBuffer_;
    std::list<std::vector<double>> EmgsBuffer_;
    std::list<std::vector<double>> externalDOFTorqueBuffer_;
    std::list<std::vector<double>> lmtBuffer_;
    std::list<std::vector<std::vector<double>>> maBuffer_;
    bool bufferEmpty_;
    std::vector<unsigned int> dofsUsed_;
    double totalTime_;
    double totalFitness_;

    std::string outDirectory_;
    bool record_;
    OpenSimFileLogger<NMSmodelT>* logger_;

    boost::thread* thread_;
    boost::mutex DataMutex_;
```

```
    boost::mutex newDataMutex_;
    bool newData_;
    boost::condition_variable conditionNewData_;
    bool threadStop_;

    // Pagmo
    PagmoProblem<NMSmodelT> UDP_;
    std::vector<unsigned> muscleIndexWithEMGtoTrack_;
    std::vector<unsigned> muscleIndexWithEMGtoPredict_;
    std::vector<unsigned> muscleIndexWithEMGtoOptimize_;
    unsigned noParameters_;
    boost::mutex pagmoMutex_;

    // Computation time logger
    std::ofstream computationTimeLogger_;
};
#endif
```

OptimizationHybridPlugin.cpp

```
#include "OptimizationHybridPlugin.h"

template <typename NMSmodelT>
OptimizationHybridPlugin<NMSmodelT>::OptimizationHybridPlugin() :
    threadStop_(true), record_(false), newData_(false), bufferEmpty_(false),
     totalTime_(0), totalFitness_(0)
{
}

template <typename NMSmodelT>
OptimizationHybridPlugin<NMSmodelT>::~OptimizationHybridPlugin()
{
}

template <typename NMSmodelT>
void OptimizationHybridPlugin<NMSmodelT>::init(NMSmodelT& model, const std
    ::string& executionXMLFileName, const std::string& configurationFileName
    )
{
    model_ = new NMSmodelT();
    setupSubject(*model_, configurationFileName);

    model_->getMuscleNames(muscleName_);

    ExecutionXmlReader execution(executionXMLFileName);
    ExecutionOptimizationXmlReader executionOptimization(execution.
        getOptimizationFile());

    dofName_ = executionOptimization.getHybridDOFsOptimized();

    if (record_)
    {
```

```
        std::string directoriOptimization = outDirectory_ + "/Optimization/";
        logger_ = new OpenSimFileLogger<NMSmodelT>(*model_, directoriOptimization);
        logger_->addLog(Logger::ShapeFactor, muscleName_);
        logger_->addLog(Logger::TendonSlackLengths, muscleName_);
        logger_->addLog(Logger::OptimalFiberLengths, muscleName_);
        logger_->addLog(Logger::GroupMusclesBasedOnStrengthCoefficients, muscleName_);
        logger_->addLog(Logger::Emgs, muscleName_);
        logger_->addLog(Logger::Activations, muscleName_);
        logger_->addLog(Logger::FibreLengths, muscleName_);
        logger_->addLog(Logger::FibreVelocities, muscleName_);
        logger_->addLog(Logger::MuscleForces, muscleName_);
        logger_->addLog(Logger::LMT, muscleName_);
        logger_->addLog(Logger::Torques, dofName_);
        std::vector<std::string> muscle{ "muscle1","muscle2","muscle3","muscle4" };
        for (std::vector<std::string>::const_iterator it = dofName_.begin(); it != dofName_.end(); it++)
        {
            logger_->addMa(*it, muscle);
        }
        computationTimeLogger_.open(directoriOptimization + "ComputationTime.sto"); // opens the file
        computationTimeLogger_ << "Time\t" << "Computation time\t" << "Fitness" << std::endl;
        if (!computationTimeLogger_)
        {
            std::cerr << "Error: computation time logger file could not be opened" << std::endl;
            exit(1);
        }
    }

    model_->eraseUnusedDofs(dofName_, dofsUsed_);

    UDP_.setModel(model_);
    UDP_.setWeightings(executionOptimization.getHybridWeightings());
    UDP_.setPerformanceCriterion(executionOptimization.getPerformanceCriterion());

    model_->getMusclesIndexFromMusclesList(muscleIndexWithEMGtoTrack_, executionOptimization.getHybridMuscleWithEMG());
    model_->getMusclesIndexFromMusclesList(muscleIndexWithEMGtoPredict_, executionOptimization.getHybridMuscleWithEMGToPredict());
    muscleIndexWithEMGtoOptimize_.assign(muscleIndexWithEMGtoTrack_.begin(), muscleIndexWithEMGtoTrack_.end());
    muscleIndexWithEMGtoOptimize_.insert(muscleIndexWithEMGtoOptimize_.end(), muscleIndexWithEMGtoPredict_.begin(), muscleIndexWithEMGtoPredict_.end());
    noParameters_ = muscleIndexWithEMGtoOptimize_.size();
```

```
    UDP_.setMusclesNamesWithEmgToTrack(executionOptimization.
        getHybridMuscleWithEMG());
    UDP_.setMusclesNamesWithEmgToPredict(executionOptimization.
        getHybridMuscleWithEMGToPredict());
    UDP_.setNoParameters(noParameters_);
    UDP_.setParameters(); // should be after setModel(),
        setMusclesNamesWithEmgToTrack() and setMusclesNamesWithEmgToPredict()

    std::vector<double> lowerBounds;
    std::vector<double> upperBounds;
    for (int i = 0; i < noParameters_; i++)
    {
        lowerBounds.push_back(0.0);
        upperBounds.push_back(1.0);
    }
    UDP_.set_bounds(lowerBounds, upperBounds); // initial boundaries are
        between 0 and 1
}

template <typename NMSmodelT>
void OptimizationHybridPlugin<NMSmodelT>::setupSubject(NMSmodelT& mySubject
    , string configurationFile)
{
    SetupDataStructure<NMSmodelT, Curve<CurveMode::Online> > setupData(
        configurationFile);
    setupData.createCurves();
    setupData.createMuscles(mySubject);
    setupData.createDoFs(mySubject);
    setupData.createMusclesNamesOnChannel(mySubject);
}

template <typename NMSmodelT>
std::vector<double> OptimizationHybridPlugin<NMSmodelT>::getDofTorque()
{
    boost::mutex::scoped_lock lock(DataMutex_);
    return dofTorque_;
}

template <typename NMSmodelT>
std::vector<double> OptimizationHybridPlugin<NMSmodelT>::getShapeFactor()
{
    boost::mutex::scoped_lock lock(DataMutex_);
    return shapeFactor_;
}

template <typename NMSmodelT>
std::vector<double> OptimizationHybridPlugin<NMSmodelT>::
    getTendonSlackLengths()
{
    boost::mutex::scoped_lock lock(DataMutex_);
    return tendonSlackLengths_;
}
```

```

template <typename NMSmodelT>
std::vector<double> OptimizationHybridPlugin<NMSmodelT>::
    getOptimalFiberLengths()
{
    boost::mutex::scoped_lock lock(DataMutex_);
    return optimalFiberLengths_;
}

template <typename NMSmodelT>
std::vector<double> OptimizationHybridPlugin<NMSmodelT>::
    getGroupMusclesBasedOnStrengthCoefficients()
{
    boost::mutex::scoped_lock lock(DataMutex_);
    return groupMusclesBasedOnStrengthCoefficients_;
}

template <typename NMSmodelT>
void OptimizationHybridPlugin<NMSmodelT>::setMuscleForce(const std::vector<
    double>& muscleForce)
{
    boost::mutex::scoped_lock lock(DataMutex_);
    muscleForce_ = muscleForce;
}

template <typename NMSmodelT>
void OptimizationHybridPlugin<NMSmodelT>::setDOFTorque(const std::vector<
    double>& dofTorque)
{
    boost::mutex::scoped_lock lock(DataMutex_);
    dofTorque_ = dofTorque;
    //newData_ = true;
}

template <typename NMSmodelT>
void OptimizationHybridPlugin<NMSmodelT>::setExternalDOFTorque(const std::::
    vector<double>& externalDOFTorque)
{
    boost::mutex::scoped_lock lock(DataMutex_);
    externalDOFTorque_ = externalDOFTorque; // Comment out for buffer
    //externalDOFTorqueBuffer_.push_back(externalDOFTorque);
}

template <typename NMSmodelT>
void OptimizationHybridPlugin<NMSmodelT>::setActivations(const std::vector<
    double>& activations)
{
    boost::mutex::scoped_lock lock(DataMutex_);
    activations_ = activations;
}

template <typename NMSmodelT>
```

```
void OptimizationHybridPlugin<NMSmodelT>::setFibreLengths(const std::vector
    <double>& fibreLengths)
{
    boost::mutex::scoped_lock lock(DataMutex_);
    fibreLengths_ = fibreLengths;
}

template <typename NMSmodelT>
void OptimizationHybridPlugin<NMSmodelT>::setFibreVelocities(const std::
    vector<double>& fibreVelocities)
{
    boost::mutex::scoped_lock lock(DataMutex_);
    fibreVelocities_ = fibreVelocities;
}

template <typename NMSmodelT>
void OptimizationHybridPlugin<NMSmodelT>::setPennationAngle(const std::
    vector<double>& pennationAngle)
{
    boost::mutex::scoped_lock lock(DataMutex_);
    pennationAngle_ = pennationAngle;
}

template <typename NMSmodelT>
void OptimizationHybridPlugin<NMSmodelT>::setTendonLength(const std::vector
    <double>& tendonLength)
{
    boost::mutex::scoped_lock lock(DataMutex_);
    tendonLength_ = tendonLength;
}

template <typename NMSmodelT>
void OptimizationHybridPlugin<NMSmodelT>::setTime(const double& time)
{
    boost::mutex::scoped_lock lock(DataMutex_);
    time_ = time;  // Comment out for buffer
    //timeBuffer_.push_back(time);
}

template <typename NMSmodelT>
void OptimizationHybridPlugin<NMSmodelT>::setEmgs(const std::vector<double
    >& Emgs)
{
    boost::mutex::scoped_lock lock(DataMutex_);
    Emgs_ = Emgs; // Comment out for buffer
    //EmgsBuffer_.push_back(Emgs);
}

template <typename NMSmodelT>
bool OptimizationHybridPlugin<NMSmodelT>::getFromBuffer(double& time, std::
    vector<double>& Emgs, std::vector<double>& externalDOFTorque, std::
    vector<double>& lmt, std::vector<std::vector<double> >& ma)
```

```
{
    boost::mutex::scoped_lock lock(DataMutex_);
    time = timeBuffer_.front();
    timeBuffer_.pop_front();
    Emgs = EmgsBuffer_.front();
    EmgsBuffer_.pop_front();
    externalDOFTorque = externalDOFTorqueBuffer_.front();
    externalDOFTorqueBuffer_.pop_front();
    lmt = lmtBuffer_.front();
    lmtBuffer_.pop_front();
    ma = maBuffer_.front();
    maBuffer_.pop_front();

    return EmgsBuffer_.empty();
}

template <typename NMSmodelT>
void OptimizationHybridPlugin<NMSmodelT>::setLmt(const std::vector<double>& lmt)
{
    boost::mutex::scoped_lock lock(DataMutex_);
    lmt_ = lmt; // Comment out for buffer
    //lmtBuffer_.push_back(lmt);
}

template <typename NMSmodelT>
void OptimizationHybridPlugin<NMSmodelT>::setMA(const std::vector<std::vector<double> >& ma)
{
    boost::mutex::scoped_lock lock(DataMutex_);
    ma_ = ma; // Comment out for buffer
    //maBuffer_.push_back(ma);
}

template <typename NMSmodelT>
void OptimizationHybridPlugin<NMSmodelT>::start()
{
    thread_ = new boost::thread(boost::bind(&OptimizationHybridPlugin<NMSmodelT>::optimization, this));
}

template <typename NMSmodelT>
void OptimizationHybridPlugin<NMSmodelT>::stop()
{
    threadStop_ = false;
    {
        boost::mutex::scoped_lock lock(newDataMutex_);
        newData_ = true;
        conditionNewData_.notify_one();
    } // Comment out for buffer
    thread_->join();
    if (record_)
```

```
    {
        logger_->stop();
        delete logger_;
    }
    delete thread_;
    delete model_;
}

template <typename NMSmodelT>
void OptimizationHybridPlugin<NMSmodelT>::setDirectories(const std::string& outDirectory, const std::string& inDirectory)
{
    outDirectory_ = outDirectory;
}

template <typename NMSmodelT>
void OptimizationHybridPlugin<NMSmodelT>::setVerbose(const int& verbose)
{
}

template <typename NMSmodelT>
void OptimizationHybridPlugin<NMSmodelT>::setRecord(const bool& record)
{
    record_ = record;
}

template <typename NMSmodelT>
void OptimizationHybridPlugin<NMSmodelT>::newData()
{
    boost::mutex::scoped_lock lock(newDataMutex_);
    newData_ = true;
    conditionNewData_.notify_one();
}

template <typename NMSmodelT>
void OptimizationHybridPlugin<NMSmodelT>::optimization()
{
    int pass{ 0 };
    while (threadStop_)// not real-time: (!bufferEmpty_ || (pass < 2999)) // when working in real-time this should be: while(threadStop_)
    {
        ++pass;

        {
            boost::mutex::scoped_lock lock(newDataMutex_);
            while (!newData_) conditionNewData_.wait(lock);
            newData_ = false; // Comment out for buffer
        }

        //bufferEmpty_ = getFromBuffer(time_, Emgs_, externalDOFTorque_, lmt_, ma_); // get data from one time frame

```

```
    auto start = std::chrono::steady_clock::now();

    DataMutex_.lock();

    unsigned int i = 0;
    for (auto it = dofsUsed_.begin(); it != dofsUsed_.end(); ++it) // set external torques for the DOFs that are optimized
    {
        UDP_.setSingleExternalTorque(externalDOFTorque_.at(*it), dofName_.at(i));
        i++;
    }

    model_->setTime(time_);
    model_->setEmgs(Emgs_);
    model_->setMuscleTendonLengths(lmt_);

    unsigned int j = 0;
    std::vector<vector<double>> ma_log; // for logging the moment arms
    for (auto it = dofsUsed_.begin(); it != dofsUsed_.end(); ++it) // set moment arms for the DOFs that are optimized
    {
        model_->setMomentArms(ma_.at(*it), j);
        j++;
        ma_log.push_back(ma_.at(*it));
    }

    UDP_.setStaticComputation();
    DataMutex_.unlock();

    vector<double> pastEMGs, initialGuess, lowerBounds, upperBounds;
    initialGuess.resize(noParameters_);
    lowerBounds.resize(noParameters_);
    upperBounds.resize(noParameters_);
    model_->getPastEmgs(pastEMGs);
    unsigned indexCt = 0;
    double range = .22;
    for (unsigned i = 0; i < muscleIndexWithEMGtoTrack_.size(); ++i, ++indexCt)
    {
        initialGuess.at(indexCt) = pastEMGs.at(muscleIndexWithEMGtoTrack_.at(i));
        lowerBounds.at(indexCt) = initialGuess.at(indexCt) - range;
        upperBounds.at(indexCt) = initialGuess.at(indexCt) + range;
        if (lowerBounds.at(indexCt) < 0.)
            lowerBounds.at(indexCt) = 0.;
        if (upperBounds.at(indexCt) > 1.)
            upperBounds.at(indexCt) = 1.;
    }
    for (unsigned i = 0; i < muscleIndexWithEMGtoPredict_.size(); ++i, ++indexCt)
    {
```

```
            initialGuess.at(indexCt) = pastEMGs.at(
                muscleIndexWithEMGtoPredict_.at(i));
            lowerBounds.at(indexCt) = initialGuess.at(indexCt) - range;
            upperBounds.at(indexCt) = initialGuess.at(indexCt) + range;
            if (lowerBounds.at(indexCt) < 0.)
                lowerBounds.at(indexCt) = 0.;
            if (upperBounds.at(indexCt) > 1.)
                upperBounds.at(indexCt) = 1.;
        }

        UDP_.set_bounds(lowerBounds, upperBounds);

        problem prob{ UDP_ };

        //auto UDA = pagmo::pso_gen(6u);
        //UDA.set_bfe(pagmo::bfe());
        //algorithm algo{UDA};

        algorithm algo{ pagmo::simulated_annealing(20.,.1,3u,1u,5u,range) };

        std::vector<double> best_f;

        population pop{prob,0}; // problem, pop_size(0), (seed)

        pop.push_back(initialGuess); // have measured EMGs as initial guess

        pop = algo.evolve(pop);

        std::vector<double> x = pop.champion_x();

        best_f = UDP_.fitness(x); // to make sure that the champion x is set
            to the model and used to compute torque (for logging)

        if (record_)
        {
            logger_->log(Logger::ShapeFactor, time_);
            logger_->log(Logger::TendonSlackLengths, time_);
            logger_->log(Logger::OptimalFiberLengths, time_);
            logger_->log(Logger::GroupMusclesBasedOnStrengthCoefficients,
                time_);
            logger_->log(Logger::Emgs, time_);
            logger_->log(Logger::Activations, time_);
            logger_->log(Logger::FibreLengths, time_);
            logger_->log(Logger::FibreVelocities, time_);
            logger_->log(Logger::MuscleForces, time_);
            logger_->log(Logger::LMT, time_);
            logger_->log(Logger::Torques, time_);
            logger_->logMa(dofName_, time_, ma_log);
            //computationTimeLogger_ << time_ << " \t" << std::chrono::
                duration <double, std::milli>(diffMinimizer).count() << " \t"
                << best_f.at(0) << std::endl;
        }
```

```
        /*std::cout << "ma:\n";
        for (unsigned int i_dof = 0; i_dof < ma_.size(); ++i_dof)
        {
            std::cout << "DOF " << i_dof << ": ";
            for (unsigned int i_ma = 0; i_ma < ma_.at(i_dof).size(); ++i_ma)
                std::cout << ma_.at(i_dof).at(i_ma) << " ";
            std::cout << "\n";
        }*/
        std::vector<double> emgs;
        std::vector<double> torques;
        std::vector<double> torques1;
        //std::vector<double> torques2;
        DataMutex_.lock();
        model_->getActivations(activations_);
        model_->getFiberLengths(fibreLengths_);
        model_->getFiberVelocities(fibreVelocities_);
        model_->getPennationAngle(pennationAngle_);
        model_->getTendonLength(tendonLength_);
        model_->getEmgs(emgs);
        //model_->getTorques(torques); // torques from minimizer
        //model_->updateState_HYBRID();
        //model_->getTorques(torques1); // torques after updating state
        //model_->setEmgs(emgs);
        //model_->updateState_HYBRID();
        //model_->getTorques(torques2); // torques after setting emg and updating state
        DataMutex_.unlock();

        auto end = std::chrono::steady_clock::now();
        auto diff = end - start;
        std::cout << std::chrono::duration <double, std::milli>(diff).count() << std::endl; // << " ms"
        totalTime_ += std::chrono::duration <double, std::milli>(diff).count();
        totalFitness_ += best_f.at(0);
    }
    std::cout << "Mean time: " << totalTime_ / pass << " ms" << std::endl;
    std::cout << "Mean fitness: " << totalFitness_ / pass << std::endl;
    std::cout << "Optimizations done: " << pass << std::endl;
    computationTimeLogger_.close();
}

#ifdef UNIX
extern "C" {
    OptimizationPlugin<NMSmodel<ExponentialActivationRT, StiffTendon<Curve<CurveMode::Online> >, CurveMode::Online> >* createEAS()
    {
        return new OptimizationHybridPlugin<NMSmodel<ExponentialActivationRT, StiffTendon<Curve<CurveMode::Online> >, CurveMode::Online> >;
    }

```

```
    OptimizationPlugin<NMSmodel<ExponentialActivationRT, ElasticTendon_BiSec
        <Curve<CurveMode::Online> >, CurveMode::Online> >* createEAEB()
    {
        return new OptimizationHybridPlugin<NMSmodel<ExponentialActivationRT,
            ElasticTendon_BiSec<Curve<CurveMode::Online> >, CurveMode::Online
            > >;
    }

    OptimizationPlugin<NMSmodel<ExponentialActivationRT, ElasticTendon<Curve
        <CurveMode::Online> >, CurveMode::Online> >* createEAE()
    {
        return new OptimizationHybridPlugin<NMSmodel<ExponentialActivationRT,
            ElasticTendon<Curve<CurveMode::Online> >, CurveMode::Online> >;
    }
}

extern "C" {
    void destroyEAS(OptimizationPlugin<NMSmodel<ExponentialActivationRT,
        StiffTendon<Curve<CurveMode::Online> >, CurveMode::Online> >* p)
    {
        delete p;
    }

    void destroyEAEB(OptimizationPlugin<NMSmodel<ExponentialActivationRT,
        ElasticTendon_BiSec<Curve<CurveMode::Online> >, CurveMode::Online> >*
         p)
    {
        delete p;
    }

    void destroyEAE(OptimizationPlugin<NMSmodel<ExponentialActivationRT,
        ElasticTendon<Curve<CurveMode::Online> >, CurveMode::Online> >* p)
    {
        delete p;
    }
}

#endif

#ifdef WIN32 // __declspec (dllexport) id important for dynamic loading
extern "C" {
    __declspec (dllexport) OptimizationPlugin<NMSmodel<
        ExponentialActivationRT, StiffTendon<Curve<CurveMode::Online> >,
        CurveMode::Online> >* __cdecl createEAS()
    {
        return new OptimizationHybridPlugin<NMSmodel<ExponentialActivationRT,
            StiffTendon<Curve<CurveMode::Online> >, CurveMode::Online> >;
    }

    __declspec (dllexport) OptimizationPlugin<NMSmodel<
        ExponentialActivationRT, ElasticTendon_BiSec<Curve<CurveMode::Online>
         >, CurveMode::Online> >* __cdecl createEAEB()
```

```
    {
        return new OptimizationHybridPlugin<NMSmodel<ExponentialActivationRT,
             ElasticTendon_BiSec<Curve<CurveMode::Online> >, CurveMode::Online
           > >;
    }

    __declspec (dllexport) OptimizationPlugin<NMSmodel<
        ExponentialActivationRT, ElasticTendon<Curve<CurveMode::Online> >,
        CurveMode::Online> >* __cdecl createEAE()
    {
        return new OptimizationHybridPlugin<NMSmodel<ExponentialActivationRT,
             ElasticTendon<Curve<CurveMode::Online> >, CurveMode::Online> >;
    }
}

extern "C" {
    __declspec (dllexport) void __cdecl destroyEAS(OptimizationPlugin<
        NMSmodel<ExponentialActivationRT, StiffTendon<Curve<CurveMode::Online
        > >, CurveMode::Online> >* p)
    {
        delete p;
    }

    __declspec (dllexport) void __cdecl destroyEAEB(OptimizationPlugin<
        NMSmodel<ExponentialActivationRT, ElasticTendon_BiSec<Curve<CurveMode
        ::Online> >, CurveMode::Online> >* p)
    {
        delete p;
    }

    __declspec (dllexport) void __cdecl destroyEAE(OptimizationPlugin<
        NMSmodel<ExponentialActivationRT, ElasticTendon<Curve<CurveMode::
        Online> >, CurveMode::Online> >* p)
    {
        delete p;
    }
}
#endif

template class OptimizationHybridPlugin<NMSmodel<ExponentialActivationRT,
    StiffTendon<Curve<CurveMode::Online> >, CurveMode::Online> >;
template class OptimizationHybridPlugin<NMSmodel<ExponentialActivationRT,
    ElasticTendon_BiSec<Curve<CurveMode::Online> >, CurveMode::Online> >;
template class OptimizationHybridPlugin<NMSmodel<ExponentialActivationRT,
    ElasticTendon<Curve<CurveMode::Online> >, CurveMode::Online> >;
```

D. User-defined problem

The code files (.h and .cpp) for the Pagmo problem class are included in this appendix. The "set functions (init)" functions are called only once at the initialization of the plugin. The "set functions (loop)" functions are called for each time instance. At the same time subject data (such as EMG, torque, lmt, ma) are set by the plugin. Then, the optimization is started. This optimization iteratively calls fitness() to evaluate the objective function.

PagmoProblem.h

```
template<typename NMSmodelT>
class PagmoProblem
{
public:
    PagmoProblem();
    ~PagmoProblem();

    // Mandatory functions (see pagmo user-defined problem documentation):
    std::vector<double> fitness(const std::vector<double>& dv) const; // dv: decision vector (values of EMG)
    std::pair<std::vector<double>, std::vector<double>> get_bounds() const;

    // Set functions (init)
    void setModel(NMSmodelT* subject);
    void setWeightings(HybridWeightings hybridParameters) { hybridParameters_ = hybridParameters; }
    void setPerformanceCriterion(const std::string performanceCriterion) { performanceCriterion_ = performanceCriterion; }
    void setMusclesNamesWithEmgToTrack(const std::vector<std::string>& musclesNamesWithEmgToTrack);
    void setMusclesNamesWithEmgToPredict(const std::vector<std::string>& musclesNamesWithEmgToPredict);
    void setNoParameters(unsigned noParameters) { noParameters_ = noParameters; }
    void setParameters();

    // Set functions (loop)
    void setSingleExternalTorque(double externalTorque, const std::string& whichDof);
    void setStaticComputation();
    void set_bounds(std::vector<double> lowerBounds, std::vector<double> upperBounds);

    // Objective function evaluation
    double evalfp() const;

private:
    NMSmodelT* subject_;
    StaticComputation<NMSmodelT, StaticComputationMode::Default<NMSmodelT>>* staticComputation_{ nullptr };

    std::vector<std::string> subjectDofNames_;
    std::pair<std::vector<double>, std::vector<double>> bounds_;
    HybridWeightings hybridParameters_;
    std::string performanceCriterion_;

    std::vector<std::string> musclesNamesWithEmgToTrack_;
    std::vector<std::string> musclesNamesWithEmgToPredict_;
```

```
    std::vector<unsigned> muscleIndexWithEMGtoTrack_;
    std::vector<unsigned> muscleIndexWithEMGtoPredict_;
    std::vector<unsigned> muscleIndexWithEMGtoOptimize_;
    unsigned noParameters_;

    std::vector<double> externalTorques_;
};
```

PagmoProblem.cpp

```
template<typename NMSmodelT>
PagmoProblem<NMSmodelT>::PagmoProblem()
{
}

template<typename NMSmodelT>
std::vector<double> PagmoProblem<NMSmodelT>::fitness(const std::vector<
    double>& dv) const
{
    std::vector<double> emgValues;
    subject_->getEmgs(emgValues);
    for (unsigned i = 0; i < muscleIndexWithEMGtoOptimize_.size(); ++i)
        emgValues.at(muscleIndexWithEMGtoOptimize_.at(i)) = dv.at(i);
    subject_->setEmgs(emgValues);
    double fp = evalfp();

    return { fp };
}

template<typename NMSmodelT>
std::pair<std::vector<double>, std::vector<double>> PagmoProblem<NMSmodelT
    >::get_bounds() const
{
    return bounds_;
}

template<typename NMSmodelT>
void PagmoProblem<NMSmodelT>::setModel(NMSmodelT* subject)
{
    subject_ = subject;
    subject_->getDoFNames(subjectDofNames_);
    externalTorques_.resize(subjectDofNames_.size());
    for (auto it = subjectDofNames_.begin(); it != subjectDofNames_.end();
        ++it)
    {
        std::cout << *it << std::endl;
    }
}

template<typename NMSmodelT>
void PagmoProblem<NMSmodelT>::setMusclesNamesWithEmgToTrack(const std::
    vector<std::string>& musclesNamesWithEmgToTrack)
```

```
{
    musclesNamesWithEmgToTrack_.assign(musclesNamesWithEmgToTrack.begin(), musclesNamesWithEmgToTrack.end());
}

template<typename NMSmodelT>
void PagmoProblem<NMSmodelT>::setMusclesNamesWithEmgToPredict(const std::vector<std::string>& musclesNamesWithEmgToPredict)
{
    musclesNamesWithEmgToPredict_.assign(musclesNamesWithEmgToPredict.begin(), musclesNamesWithEmgToPredict.end());
}

template<typename NMSmodelT>
void PagmoProblem<NMSmodelT>::setParameters()
{
    subject_->getMusclesIndexFromMusclesList(muscleIndexWithEMGtoTrack_, musclesNamesWithEmgToTrack_);
    subject_->getMusclesIndexFromMusclesList(muscleIndexWithEMGtoPredict_, musclesNamesWithEmgToPredict_);

    //concatenate muscleIndexWithEMGtoTrack_ and muscleIndexWithEMGtoPredict_
    muscleIndexWithEMGtoOptimize_.assign(muscleIndexWithEMGtoTrack_.begin(), muscleIndexWithEMGtoTrack_.end());
    muscleIndexWithEMGtoOptimize_.insert(muscleIndexWithEMGtoOptimize_.end(), muscleIndexWithEMGtoPredict_.begin(), muscleIndexWithEMGtoPredict_.end());
}

template<typename NMSmodelT>
void PagmoProblem<NMSmodelT>::set_bounds(std::vector<double> lowerBounds, std::vector<double> upperBounds)
{
    bounds_ = { lowerBounds, upperBounds };
}

template<typename NMSmodelT>
void PagmoProblem<NMSmodelT>::setSingleExternalTorque(double externalTorque, const std::string& whichDof)
{
    vector<string>::const_iterator it = subjectDofNames_.begin();
    while (*it != whichDof && it != subjectDofNames_.end())
        ++it;
    if (*it == whichDof) {
        unsigned pos = std::distance<vector<string>::const_iterator>(subjectDofNames_.begin(), it);
        externalTorques_.at(pos) = externalTorque;
    }
    else {
        std::cout << "ErrorMinimizer::setSingleExternalTorque ERROR\n" << whichDof << " not found in the subject\n";
```

```
        exit(EXIT_FAILURE);
    }
}

template<typename NMSmodelT>
void PagmoProblem<NMSmodelT>::setStaticComputation()
{
    if (staticComputation_) // should not be necessary if staticComputation is initialized as a null pointer
        delete staticComputation_;

    staticComputation_ = new StaticComputation<NMSmodelT, StaticComputationMode::Default<NMSmodelT>>(*subject_, musclesNamesWithEmgToTrack_, musclesNamesWithEmgToPredict_);
}

template<typename NMSmodelT>
double PagmoProblem<NMSmodelT>::evalfp() const
{
    // 1) calculate first term of objective function: Least Squares Fitting
    double torqueLeastSquaresFitting = .0;
    vector<double> torques;
    staticComputation_->getTorques(torques);

    for (unsigned d = 0; d < torques.size(); ++d)
    {
        torqueLeastSquaresFitting += fabs(externalTorques_.at(d) - torques.at(d)) * fabs(externalTorques_.at(d) - torques.at(d));
    }

    //2) calculate the second term of objective function: track experimental EMGs
    double emgTracking = .0;
    vector<double> experimentalEMGs, adjustedEMGs;
    staticComputation_->getInitialValuesOfTrackedEMGs(experimentalEMGs); // emg value for the tracked muscles before the emg adjustment
    staticComputation_->getAdjustedValuesOfTrackedEMGs(adjustedEMGs);


    for (unsigned e = 0; e < adjustedEMGs.size(); ++e)
        emgTracking += fabs(experimentalEMGs.at(e) - adjustedEMGs.at(e));

    //3) calculate the performance criterion term of the objective function
    double performanceCriterion = .0;
    vector<double> currentEMGs;
    staticComputation_->getCurrentEMGs(currentEMGs);
    for (unsigned i = 0; i < currentEMGs.size(); ++i) {
        performanceCriterion += currentEMGs.at(i) * currentEMGs.at(i);
    }

    double fp;
```

```
    fp = hybridParameters_.alpha * torqueLeastSquaresFitting +
        hybridParameters_.gamma * emgTracking + hybridParameters_.beta *
        performanceCriterion;

    return fp;
}
```