

Wearable Sensors for Indirect Assessment of Ground Reaction Forces and Moments in Human Movement Analysis

**Master thesis submitted in order to be awarded the Master's
Degree in Biomedical Engineering**

Student: Hanne Willems

Promotor: prof. dr. ir. Tom Verstraten

Supervisor: dr. ir. Kevin Langlois
dr. ir. Joris De Winter

Academic year: 2024–2025

Statement of Confidentiality

Important: this master's dissertation contains confidential information and/or confidential research results proprietary to Ghent University, VUB or third parties. It is strictly forbidden to publish, cite or make public in any way this master's dissertation or any part thereof without the express written permission of Ghent University or VUB. Under no circumstance may this master's dissertation be communicated to or put at the disposal of third parties. Photocopying or duplicating it in any other way is strictly prohibited. Disregarding the confidential nature of this master's dissertation may cause irremediable damage to Ghent University or VUB. The stipulations mentioned above are in force until the embargo date 10/01/2027.

Acknowledgement

This thesis is the final sprint after five years of study, with many trials and errors along the way. There were many frustrating moments when the results were not as expected, but there were also moments of great happiness when things did work out. Fortunately, I always had the help and support of the people around me.

Firstly, I would like to thank my supervisor, Kevin Langlois, for his guidance and enthusiasm. He provided me with a workspace amidst the office chaos and gave me the opportunity to find my own way during the work on this thesis. Whenever I had questions, he always had a lot of ideas. I would also like to thank my second supervisor, Joris De Winter, for his technical advice. His realism taught me to accept that, at some point, you have to put your work aside.

I would also like to thank Professor Tom Verstraten, my promotor, for his guidance in structuring this thesis, for his insightful feedback and for all the opportunities he offered me.

I would also like to thank Amine for his help with the data collection, all the participants who took part in it, Evandro for explaining quaternions to me, Robbe for answering my questions about AI, Menthy for showing me how to use OpenSim, and Preben and Silvia for proofreading my thesis. Additionally, I would like to thank everyone else who gave me ideas, which I cannot name here.

Finally, I would like to thank my wonderful family for all their support over the past five years. Thanks to them, I was able to begin this journey, and now, with this final sprint, I can almost reach the finish line.

The author gives permission to make this master dissertation available for consultation and to copy parts of this master dissertation for personal use. In all cases of other use, the copyright terms have to be respected, in particular with regard to the obligation to state explicitly the source when quoting results from this master's dissertation.

Hanne Willems, 02/06/2025

This master's dissertation is part of an exam. Any comments formulated by the assessment committee during the oral presentation of the master's dissertation are not included in this text.

Abstract

The quantification of biomechanical loads is crucial to gain insights in keeping the balance between enhancing training performance, and preventing injuries. The Ground Reaction Force (GRF) is an important variable to assess the biomechanical loading. Measuring the GRF is restricted to a gait laboratory, however, it would be more valuable outside the lab in the runners' own environment. The development of sensorised running pants with five integrated inertial measurement units makes it possible to measure accelerations and orientations of the body's segments. This thesis investigates wearable-sensor based methods to estimate the Ground Reaction Force in the field, using the data of the sensorised running pants.

Firstly, a literature study was conducted to identify the existing GRF estimation models. Based on this information, it was decided to further develop two models: the link-segment model and the mass-spring damper model. The link-segment model is based on a division of the body into segments to which Newton's second law is applied. The mass-spring-damper model represents the body as a series of masses, springs and dampers. Then, steps were taken to make the models as accurate as possible by further refining the different components, both with and without artificial intelligence.

The results demonstrate the potential of the link-segment model, as it can estimate the vertical GRF of running at a constant speed (8 km/h, 10 km/h and 12 km/h) and jumping with a relative root mean squared error (rRMSE) of approximately 7 %. In particular for running, the peak error and the impulse error were both approximately 7 %. Capturing the impact peak of running with the model was challenging, but this issue was solved by combining the link-segment model with a two-mass model. The estimation of the Ground Reaction Moment (GRM) and the Centre-of-Pressure (CoP) was less successful: the GRM around the vertical axis, with the ankle joint axis as the point of application, showed an error rate of $16.50 \pm 6.11\%$. The optimal results were achieved with weighting factors calculated using machine learning rather than anthropometric data. However, their practical implementation is not clear. Furthermore, it is important to note that the GRF estimation is improved by increasing the number of segments. The pelvis provides a rough estimation of the active peak, while the other segments provide the smaller features, such as the impact peak. The cut-off frequency for the low-pass filter of the acceleration of each segment will need to be selected individually for each segment, and is dependent on the task.

The results demonstrate that the mass-spring damper model is less suitable for estimating the Ground Reaction Force. Despite the small variability in parameter values among the participants, and the almost perfect alignment of the modelled accelerations and measured accelerations, the model was unable to accurately estimate the GRF.

In conclusion, running pants equipped with inertial measurement units offer the possibility to monitor training loads in the field based on the estimated GRF. The link-segment model seems to be the best method to use, under the assumption of a good optimisation of the weight factors, cut-off frequency of the low-pass filter and the number of segments.

Keywords: training load monitoring, inertial measurement unit, Ground Reaction Force, Ground Reaction Moment, link-segment model, mass-spring-damper model

Extended Abstract

Introduction

The quantification of biomechanical loads on the body is crucial to gain insights in keeping the balance between improving training performance, and minimising the injury risk [1]. The Ground Reaction Force (GRF) is an important variable to assess the biomechanical load. Measuring the GRF is restricted to a gait laboratory. However, it would be more valuable outside the lab within the runners' natural environment. The development of running pants with five inertial measurement units (IMUs) makes it possible to measure accelerations and orientations of the body segments. This study investigates IMU-based methods to estimate the Ground Reaction Force (GRF) outside the lab [2], using the data of the sensorised running pants. Two models are developed: a link-segment model combined with a two-body model [3] [4], and a mass-spring-damper model [5] [6]. The aim is that the final model can estimate the GRF for running and jumping with a relative root mean square error below 10 % in the vertical direction.

Methods

Link-segment model

Newton's second law states that acceleration is proportional to the force. Consequently, the GRF can be estimated from the acceleration of the body segments, as shown in equation 2, for the medio-lateral, anterior-posterior and vertical direction. Since the IMUs measure continuously and do not detect when the leg is in the flight phase, the vertical accelerations ($\mathbf{a}_{i,v}$) were corrected by stating that they should be larger than the gravitational acceleration ($g = 9.81m/s^2$):

$$\mathbf{a}_{i,v} = 0 \quad \text{if } \mathbf{a}_{i,v} \leq -g \quad (1)$$

After the calculation (see equation 2), the eGRF is normalised for body weight ($BW = m \cdot g$). This model applies to both running and jumping, and can estimate the GRF in the medio-lateral, anterior-posterior and vertical directions.

$$eGRF = m \cdot \mathbf{G} + \sum_{i=1}^N w_i \cdot m \cdot \mathbf{A}_i \quad (2)$$

where m : the body mass of the participant

$G = [0, 0, g]$: the gravitational acceleration in the medio-lateral (0), anterior-posterior (0) and vertical ($g = 9.81m/s^2$) direction

i : annotates the segment number and ranges from 1 to N , with N the total number of segments. The number of segments is depend on the number of sensors. In this case, five segments are defined, based on the data of the IMUs on the pelvis, left shank, left thigh, right shank and right thigh.

w : the weighting factor that represents the fraction of body mass corresponding to the segment where the sensor or marker is attached to. In this case, the weighting factors of De Leva for men are used [7]

$A_i = [Ax_i, Ay_i, Az_i]$: the measured acceleration of each body segment, with x the medio-lateral direction, y the anterior-posterior direction, and z is the vertical direction

Two-body model

The fundamental hypothesis of the two-body model is that the total vertical Ground Reaction Force is composed of two overlapping bell-shaped impulses, as described by Clark [4]. The first impulse (J_1) is due to the vertical deceleration of the shank and the foot (m_1) during initial contact with the running surface. Therefore, the impact peak of the GRF waveform originates from this impulse. The second impulse (J_2) is due to the concurrent vertical accelerations of the rest of the body (m_2) after surface impact. The two-body model can only be used to estimate the GRF in the vertical direction, and is only valid for running.

$$eGRF_z(t) = F_1(t) + F_2(t) \quad (3)$$

$$= F_{1,avg} \left[1 + \cos \left(\frac{t - \Delta t_1}{\Delta t_1} \pi \right) \right] + F_{2,avg} \left[1 + \cos \left(\frac{t - \frac{t_c}{2}}{\frac{t_c}{2}} \pi \right) \right] \quad (4)$$

where m_1 : mass of the shank and the foot

Δt_1 : time interval between initial contact and the vertical velocity of m_1 slowing to zero

$F_{1,avg}$: the average force during the total time interval ($2\Delta t_1$) of impulse J_1

t_c : period of time the foot is in contact with the ground (stance phase), i.e. from initial contact to toe-off

When the link-segment model is combined with the two-body model to estimate the GRF in the vertical direction, the first force (F_1) is calculated using the vertical GRF of the link-segment model, and the second force (F_2) is calculated using the two-body model. The combination of both models ensures that the impact peak can be approached, and that the deviation on the active peak remains small. The GRF in the medio-lateral and anterior-posterior direction was estimated based on the link-segment model, because the two-body model is not defined in that direction.

Mass-spring-damper model

The human body behaves in a spring-like manner during bouncing gaits such as running and hopping. Therefore, a mass-spring-damper model describes the body as a series of masses that are connected with springs and dampers. The vertical GRF acting on the MSD-model is then calculated as

$$eGRF_z = \frac{(m_1 + m_2)\omega_2}{1 + \lambda}(\omega_2 \mathbf{p}_2 - 2\zeta \mathbf{v}_2) \quad (5)$$

The velocities of the upper and lower masses ($\mathbf{v}_1$ and $\mathbf{v}_2$) are determined by integrating the accelerations of the IMUs. Their positions ($\mathbf{p}_1$ and $\mathbf{p}_2$) can be calculated in a similar manner by integrating the accelerations twice. The natural frequencies of the springs (ω_1, ω_2), the mass ratio (λ) and the damping coefficient (ζ) are still unknown in these equations. With the prior knowledge of the weighting factors from the link-segment model, m_1, m_2 and therefore λ can already be estimated. The other parameters are calculated with a purpose-written algorithm, based on finding the optimal value with two first-order differential equations. The mass-spring-damper model can only be used to estimate the GRF in the vertical direction, and is only valid for running.

Model Validation

The models are validated by comparing the estimated GRF with the measured GRF of an instrumented treadmill. Eight healthy athletes participated in the data collection for this study, of which seven recordings were ultimately usable. The participants of the processed recordings were all men, 27 ± 5 years old, 1.77 ± 0.09 m tall, and weigh 70 ± 6 kg. All participants signed a written informed consent beforehand.

The relative root mean square error (rRMSE) provides a general idea of how correct the fitting is between the estimated and the measured GRF, and is defined as:

$$rRMSE = \frac{\sqrt{\frac{\sum_{n=1}^N (mGRF_n - eGRF_n)^2}{N}}}{\max(mGRF) - \min(mGRF)} \cdot 100 \quad (6)$$

Results

The number of analysed steps is 1337 at 8 km/h, 1630 at 10 km/h, and 1615 at 12 km/h. For jumping, 29 countermovement jumps were analysed.

Link-segment model, combined with two-mass model

Figure 1 illustrates the mean eGRF with its standard deviation, using a link-segment model in combination with a two-body model (blue). The measured GRF is shown for comparison (orange). The visual analysis indicates that the combination of the two models should be effective. More quantitatively, the mean rRMSE with standard deviation is 8.45 ± 2.98 % at 8

km/h, 10.58 ± 5.38 % at 10 km/h and, and 17.13 ± 12.53 % at 12 km/h for the vertical GRF. For the medio-lateral GRF, the errors are 25.06 ± 13.50 % at 8 km/h, 27.38 ± 14.05 % at 10 km/h, and 31.45 ± 24.13 % at 12 km/h. For the anterior-posterior GRF, the errors are 20.60 ± 11.54 %, 22.04 ± 15.81 %, 29.51 ± 35.07 %, at 8 km/h, 10 km/h and 12 km/h respectively.

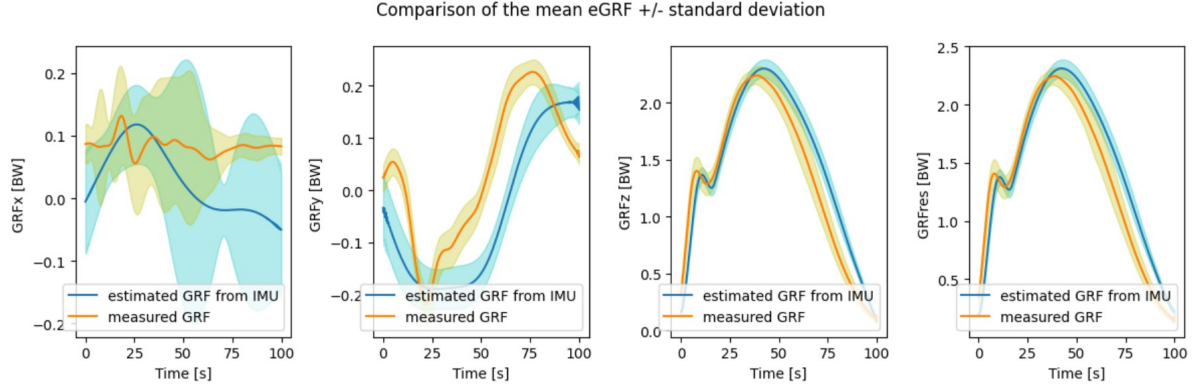


Figure 1: Mean estimated GRF with standard deviation from a link-segment model combined with a two-body model (blue), and the mean measured GRF with standard deviation (orange) as a reference.

Despite the fact that a jumping movement is less repetitive than a running movement, the measured GRF and the estimated GRF visually match well (see figure 2). The rRMSE of the medio-lateral GRF is 27.34 ± 12.91 %, 12.48 ± 4.40 % for the anterior-posterior GRF, and 6.11 ± 4.42 % for the vertical GRF.

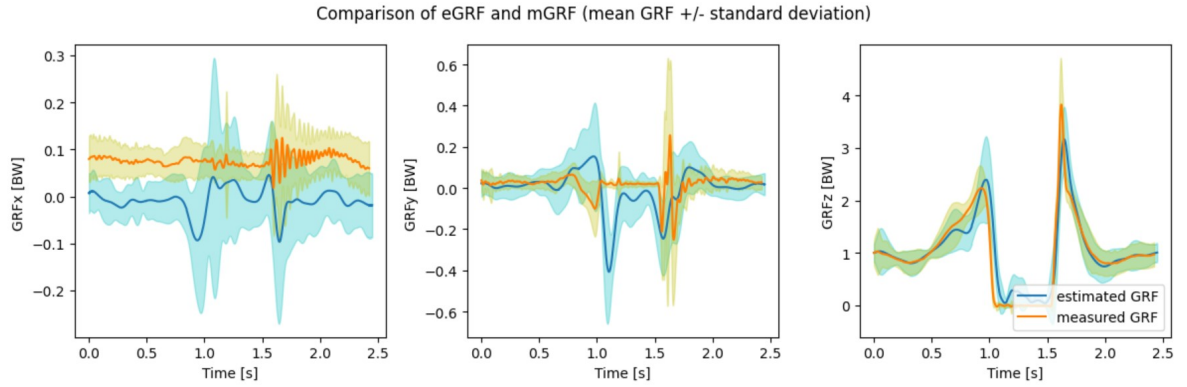


Figure 2: Mean and standard deviation of the estimated (blue) and measured Ground Reaction Force (orange) during jumping for all participants.

Mass-spring damper model

Although the error remained relatively constant during the calculation of the parameters, the mass-spring-damper model did not produce the expected results. Figure 3 shows a representative example of the estimated Ground Reaction Force of the mass-spring-damper model.

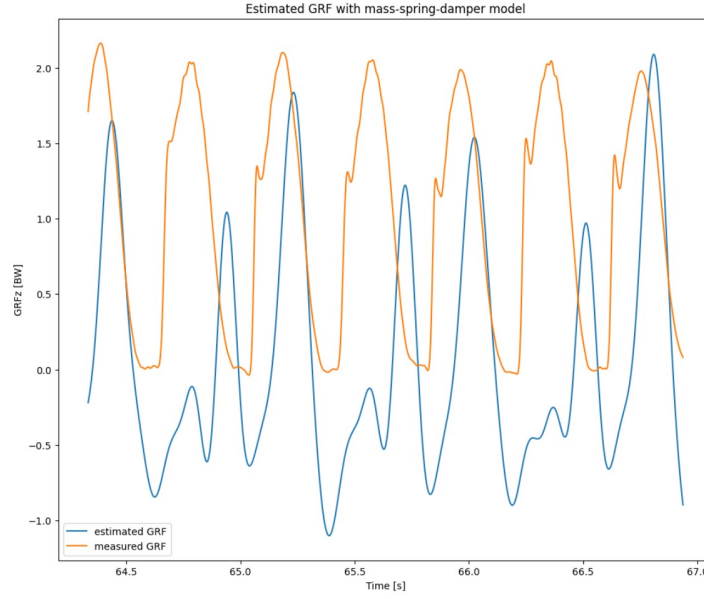


Figure 3: Representative example of the estimated vertical GRF from the mass-spring damper model (blue), with the measured vertical GRF as a reference (orange).

Discussion

Discussion on link-segment model combined with two-body mode

The error of the link-segment model increases as the speed increases, and is therefore dependent on the task. The model can be further improved by adding segments to the model, optimising the weighting factors, and choosing a better cut-off frequency. The optimal results are achieved with weighting factors calculated using machine learning rather than anthropometric data. However, their practical implementation is not clear. Furthermore, it is important to note that the GRF estimation is improved by increasing the number of segments. The pelvis provides a rough estimation of the active peak, while the other segments provide smaller components, such as the impact peak. The cut-off frequency for the low-pass filter of the acceleration of each segment will need to be selected individually for each segment, and is dependent on the task. The higher the intensity of the task, the higher the required cut-off frequency [8].

Discussion on mass-spring damper model

Although most papers assume a constant natural frequency in a mass-spring-damper model, varying ω_1 allows for a quasi-perfect approximation of the acceleration of the upper mass (a_1) with the model, while a constant ω_1 does not achieve the desired results [6][9]. This idea is confirmed by the more theoretical literature: the leg is known to be stiffer during landing than take-off [5], while the muscle-tendon units produce more work during the push-off phase [10][11]. The average value of this varying ω_1 is approximately the same as the constant value.

Calculating parameters ω_2 and ζ for the approximation of the acceleration of the lower mass (a_2) was a more complex process. This is also the bottleneck of the mass-spring damper model. Firstly, it should be noted that the results of the estimation algorithm for ω_2 and ζ are very dependent on the chosen initial values. After conducting a series of trials, it was determined that $\omega_{2,mean}$ was approximately equal to 18, and ζ_{mean} fell within the range of -4.8 to 0.5 across all participants. The optimal parameters appeared to be fairly consistent across all participants, and these values are in line with those of previous studies [6]. However, the results were not as expected.

The results also demonstrated that ω_2 is primarily responsible for the active peak, while ζ is more associated with the impact peak. If the value of ω_2 increases, the active peak and the impact peak both rise as well. Increasing the damping ratio led to a reduction in the impact peak, and decreased the time from initial contact to the impact peak [5] [12]. However, it is important to note that the model is very sensitive to small changes in the parameters. Furthermore, the parameters are challenging to estimate due to the absence of a physical interpretation.

Conclusion

This study is a first step towards measuring Ground Reaction Forces in the field, with the aim of improving injury prevention by better monitoring biomechanical body loads. The link-segment model has the potential to achieve this goal with sufficient accuracy. The mean rRMSE of the vertical estimated GRF with standard deviation is $8.45 \pm 2.98 \%$, $10.58 \pm 5.38 \%$ and $17.13 \pm 12.53 \%$ for running at 8 km/h, 10 km/h and 12 km/h, respectively, and $6.11 \pm 4.42 \%$ for jumping. The mass-spring damper model was not able to deliver accurate predictions of GRF waveforms. The primary bottleneck of this model is the estimation of the natural frequency of the lower spring (ω_2) and the damping coefficient (ζ).

Contents

Statement of Confidentiality	1
Acknowledgement	2
Abstract	4
Extended Abstract	6
List of Figures	15
List of Tables	20
1 Introduction	23
1.1 Objectives	24
1.2 Outline Thesis	25
2 Medical background	27
2.1 Definitions	27
2.2 Training load monitoring	28
2.3 Ground Reaction Force	29
2.3.1 Measuring and estimating the Ground Reaction Force	31
2.3.1.1 Laboratory methods	31
2.3.1.2 Ambulatory methods	31
2.3.2 Characteristics of the Ground Reaction Force	33
2.3.2.1 GRF in walking	33
2.3.2.2 GRF in running	35
2.3.2.3 GRF in jumping	38
3 State-of-the-art	40
3.1 Model Validation	40
3.2 Wearable-sensor based Ground Reaction Force estimation models	42
3.2.1 Dynamics	43
3.2.1.1 Inverse dynamics	43
3.2.1.1.1 Link-segment model	44
3.2.1.1.1.1 Number of segments	45
3.2.1.1.1.2 Model performance	48
3.2.1.1.2 Mass-spring-damper model	52
3.2.1.1.2.1 One-body model	53
3.2.1.1.2.2 Two-body model	55
3.2.1.1.2.3 Multi-body model	59
3.2.1.2 Forward dynamic approach	60
3.2.2 Deep learning	60

3.2.3	Hybrid Method	63
3.2.4	Conclusion	64
4	Methods	68
4.1	Experimental Design and Data Processing	68
4.2	Anthropometry	70
4.3	Step detection	71
4.3.1	Step detector 1: arbitrary threshold on the vertical Ground Reaction Force	71
4.3.2	Step detector 2: quaternions	71
4.3.3	Step detector 3: velocity of the pelvis	71
4.3.4	Jump detection	71
4.4	Model Engineering	72
4.4.1	Link-segment model	72
4.4.1.1	Ground Reaction Force	72
4.4.1.2	Translated accelerations	74
4.4.1.3	Weighting factors	74
4.4.1.3.1	Anthropometric measurements	74
4.4.1.3.2	Machine learning	75
4.4.1.3.2.1	Linear Regression	75
4.4.1.3.2.2	Bayesian Ridge Regression	76
4.4.1.3.2.3	Stochastic Gradient Descent	76
4.4.1.4	Number of Components	77
4.4.1.5	Principal component analysis	78
4.4.1.6	Filter cut-off frequency analysis	79
4.4.1.7	Ground Reaction Moment	79
4.4.1.8	Centre-of-pressure	80
4.4.2	Two-body model	81
4.4.3	Mass-spring-damper model	83
5	Results	87
5.1	Data analysis	87
5.2	Step detection	90
5.2.1	Step detector 1: arbitrary threshold on the vertical Ground Reaction Force	90
5.2.2	Step detector 2: quaternions	91
5.2.3	Step detector 3: velocity of the pelvis	92
5.2.4	Analysis of the detected steps	93
5.2.5	Jump detection	93
5.3	Link-segment model	94
5.3.1	Translated accelerations	95
5.3.2	Model performance	96

5.3.2.1	Ground Reaction Force	96
5.3.2.1.1	Weighting factors	101
5.3.2.1.2	Number of segments	105
5.3.2.1.3	Principal component analysis	107
5.3.2.1.4	Filter cut-off frequency analysis	113
5.3.2.2	Ground Reaction Moment	116
5.3.2.3	Centre-of-pressure	119
5.4	Two-body model	120
5.5	Combination of link-segment model and two-body model	122
5.6	Mass-spring damper model	126
6	Discussion	129
6.1	General remarks	129
6.2	Remarks on the model development	129
6.3	Remarks on the model validation	130
7	Conclusion	131
8	Societal reflection	132
8.1	Sustainable Development Goal 3: Good Health and Well-Being	132
8.2	Sustainable Development Goal 9: Industry, Innovation and Infrastructure . . .	132
8.3	Sustainable Development Goal 12: Responsible Consumption and Production .	132
8.4	Sustainable Development Goal 17: Partnerships for the Goals	133
	References	134
A	Appendix: Usage of AI	138
B	Appendix: Code of Model Development	139

List of Figures

1	Mean estimated GRF with standard deviation from a link-segment model combined with a two-body model (blue), and the mean measured GRF with standard deviation (orange) as a reference.	9
2	Mean and standard deviation of the estimated (blue) and measured Ground Reaction Force (orange) during jumping for all participants.	9
3	Representative example of the estimated vertical GRF from the mass-spring damper model (blue), with the measured vertical GRF as a reference (orange).	10
4	Sensorised running pants, with one IMU attached on the pelvis, one IMU on each thigh and one IMU on each shank.	24
5	The components of movement analysis.	27
6	Anatomical terms of location, labelled on the human body.	28
7	Load monitoring framework, demonstrating the duality of the physiological and biomechanical load-response pathway.	29
8	Schematic representation of the Ground Reaction Force.	30
9	Illustration of a force plate and optical motion capture.	31
10	Schematic representation of an IMU. An IMU is composed of an accelerometer, a gyroscope and a magnetometer.	32
11	Schematic representation of one gait cycle, which is defined by two consecutive foot contact events on the same leg.	34
12	Recording of the Ground Reaction Force during one gait cycle.	35
13	Schematic representation of one running cycle.	36
14	The transition from walking (blue) to running (red) is visible in the recordings of the Ground Reaction Force.	37
15	GRF during running.	38
16	Schematic representation of the Ground Reaction Force of a countermovement jump	40
17	Overview of methods to estimate the GRF.	43
18	Link-segment model.	44
19	Best combinations of segments across all the tasks for each given number of segments.	46
20	Root mean square errors (RMSE) for estimated resultant GRF curves for each task. Bars represent the percentage of trials (primary y-axis) within the very low (<1 N kg ⁻¹), low (1-2 N kg ⁻¹), moderate (2-3 N kg ⁻¹), high (3-4 N kg ⁻¹) or very high (>4 N kg ⁻¹) error boundaries, and black dots represent the mean errors (secondary y-axis), for each given number of segments.	47
21	Measurement set-up of the experiment of D'Andrea (2019), that investigated the Ground Reaction Force of alternate lateral jumps.	51
22	Schematic representation of the mass-spring-damper model during running. It consists of a mass, a spring and a damper to represent the mass and mechanical properties of the human body.	53

23	Passive single-body model as a schematic representation of (a) 1D motion and (b) 2D motion.	54
24	An illustration of the human body represented as a two-body MSD-model. The MSD-model consists of a lower mass-spring-damper element (m_2 , k_2 , c), that represents the support leg of the human body, and an upper mass-spring element (m_1 , k_1), that represents the rest of the human body.	55
25	Representative examples of the trunk accelerometry fitting, GRF prediction, and the median RMSE across running speeds, of a single stride from multiple subjects. Figure (A) displays the fitting of the upper mass acceleration to the trunk accelerometry signal across running speeds, and figure (B) displays the measured and predicted GRF for the same trials. The RMSE for the trunk accelerometry fitting and GRF predictions are written in brackets for the individual examples. Figures (C) and (D) display the average RMSE median, and 25th and 75th interquartile range for the trunk accelerometry fitting and GRF prediction, respectively within and across the individual running speeds. . . .	56
26	Typical examples of measured (black continuous line) and modelled (red dotted line) Ground Reaction Force (GRF) profiles including the root mean square error (RMSE) between both curves. (Verheul, 2021)	58
27	Test predictions from 10 repetitions of training. (a) r^2 coefficient between prediction and data; (b) corresponding RMSE. Results obtained using: (i) arbitrary options, without appending signal length; (ii) arbitrary options, with signal length appended; and (iii) optimised options, with signal length appended. Central horizontal lines show the median and boxes show the interquartile range. Whiskers show up to 1.5 times the interquartile range; any values outside this are plotted as individual points, which show the r^2 or RMSE value for individual outlying impact predictions.	63
28	Schematic representation of the local frame of the IMU and the global frame of the lab.	69
29	Block diagram of the implementation of the link-segment model.	73
30	Schematic representation of the two-body model, that composes the Ground Reaction Force of two overlapping bell-shaped impulses.	82
31	Schematic representation of the passive two-body mass-spring-damper model, as described by Derrick [12].	84
32	Representative example of the comparison between the accelerations of the IMUs (blue) and the markers (orange). The top row are the accelerations in the medio-lateral direction, the middle row in the anterior-posterior direction and the bottom row in the vertical direction. The columns show the body segments, from left to right: right thigh, left shank, left thigh, right shank and pelvis.	88

33	Representative example of the comparison between the angular velocities of the IMUs (blue) and the markers (orange). The top row are the angular velocities in the medio-lateral direction, the middle row in the anterior-posterior direction and the bottom row in the vertical direction. The columns show the body segments, from left to right: right thigh, left shank, left thigh, right shank and pelvis.	89
34	Representative example of step detector 1, which used an arbitrary threshold value on the Ground Reaction Force to determine initial contact and toe-off. For each step, initial contact is indicated with an orange cross, and toe-off with a green cross.	90
35	Representative example of step detector 2, which used the quaternions of the IMU on the right shank to determine initial contact and toe-off. For each step, initial contact is indicated with an orange cross, and toe-off with a green cross.	91
36	Representative example of step detector 3, which used the peak downward velocity of the pelvis to determine initial contact and toe-off. For each step, initial contact is indicated with an orange cross, and toe-off with a green cross.	92
37	Representative example of the jump detector. The start of the jump is indicated with an orange cross, and the end with a green cross.	94
38	Representative example of the comparison between the original accelerations of the IMUs (blue) and the translated accelerations (orange). The top row are the accelerations in the medio-lateral direction, the middle row in the anterior-posterior direction and the bottom row in the vertical direction. The columns show the body segments, from left to right: right thigh, left shank, left thigh, right shank and pelvis.	95
39	Representative example of the comparison between the estimated GRF from the original accelerations of the IMUs (blue), the estimated GRF from the translated accelerations (green) and the measured GRF (orange). The top row is the GRF in the medio-lateral direction, the middle row in the anterior-posterior direction and the bottom row in the vertical direction.	96
40	Mean and standard deviation of the estimated (blue) and measured Ground Reaction Force (orange) during running for all participants.	97
41	Mean and standard deviation of the estimated (blue) and measured Ground Reaction Force (orange) during jumping for all participants.	98
42	Violin plot of the loading rate error, impulse error and peak error of the estimated Ground Reaction Force of the link-segment model, during running for all participants.	99
43	Violin plot of the relative root mean squared error of the estimated Ground Reaction Force of the link-segment model, during running for all participants.	100

44	Overview of the calculated weighting factors per body segment (column), per assignment and participant (row). The weighting factors calculated using machine learning are LSE (Linear Regression), BR (Bayesian Ridge Regression), SGD (Stochastic Gradient Descent) with the conditions of only symmetric weighting factors (SGD symm), non-symmetric weighting factors (SGD nosymm), and fitting based on the peaks (SGD peaks). For the weighting factors based on anthropometric measurements, there are MdL (De Leva for men), FdL (De Leva for women), Winter (W), and AM (method based on approximate volume of the body segment).	102
45	Correlation matrix of the accelerations of all participants for running.	105
46	Correlation matrix of the accelerations of all participants for jumping.	106
47	Explained variance of principal component analysis for all participants during running.	108
48	Representative example of the individual principal components (left) and the reconstructed Ground Reaction Force (right), from the first three principal components for running. (Second part on figure 49).	109
49	Representative example of the individual principal components (left) and the reconstructed Ground Reaction Force (right), from principal components three until five for running. (First part on figure 48).	110
50	Explained variance of principal component analysis for all participants during jumping.	111
51	Representative example of the individual principal components (left) and the reconstructed Ground Reaction Force (right), from the first three principal components for jumping. (Second part on figure 52).	112
52	Representative example of the individual principal components (left) and the reconstructed Ground Reaction Force (right), from the fourth principal component for jumping. (First part on figure 51).	113
53	Representative examples of the frequency spectra of the vertical accelerations of the pelvis, thighs and shanks.	113
54	Representative examples of the measured vertical Ground Reaction Force (orange) and estimated vertical GRF (blue), when the cut-off frequency of the IMU filtering is 2 Hz, 4 Hz and 6 Hz.	115
55	Representative examples of the measured vertical Ground Reaction Force (orange) and estimated vertical GRF (blue), when the cut-off frequency of the IMU filtering is 8 Hz, 10 Hz and 12 Hz.	115
56	Representative examples of the measured vertical Ground Reaction Force (orange) and estimated vertical GRF (blue), when the cut-off frequency of the IMU filtering is 12 Hz, 14 Hz and 16 Hz.	115
57	Mean and standard deviation of the estimated (blue) and measured Ground Reaction Moment (orange) during running for all participants.	116
58	Violin plot of the relative root mean squared error between the estimated and measured Ground Reaction Moment, during running for all participants. . . .	118

59	Representative example of the estimated centre-of-pressure (blue) and measured centre-of-pressure (orange) during running.	119
60	Representative example of the GRF estimation (blue) during running from the two-body model, and the measured GRF (orange) as reference.	120
61	Violin plot of the loading rate error, impulse error and peak error of the estimated Ground Reaction Force from the two-body model, during running for all participants.	121
62	Violin plot of the relative root mean squared error of the estimated Ground Reaction Force from the two-body model, during running for all participants.	122
63	Mean estimated GRF with standard deviation from a link-segment model combined with a two-body model (blue), and the mean measured GRF with standard deviation (orange) as a reference.	123
64	Violin plot of the relative loading rate error, relative impulse error and relative peak error on the estimated GRF for running of the link-segment model combined with the two-body model.	124
65	Violin plot of the rRMSE (vertical direction) on the estimated GRF of the link-segment model combined with the two-body model.	125
66	Approximation of the acceleration of the upper mass with the mass-spring-damper model, if the natural frequency of the upper mass (ω_1) is constant (left) or variable over time (right).	126
67	Representative example of the course of ω_2 and ζ over the recording, as the running speed increased from 8 km/h to 12 km/h.	128
68	Representative example of the estimated GRF from the mass-spring damper model (blue), with the measured GRF as a reference (orange).	129

List of Tables

1	List of abbreviations	22
2	Comparison between Ground Reaction Forces, estimated from seventeen IMUs for multiple walking speeds, and force plate measurements. (Karatsidis, 2017).	49
3	Root Mean Square Error (RMSE), Relative RMSE (rRMSE), Pearson's correlation coefficient and peak error of the vertical GRF, estimated from three segmental accelerations for running at a constant speed. Values are means $\pm$ one standard deviation. (Scheltinga, 2023).	50
4	Root Mean Square Error (RMSE) and Relative RMSE of the medio-lateral and vertical GRF estimated from eight segmental accelerations, for alternate lateral bounding. (D'Andrea, 2019).	51
5	Root Mean Square Error (RMSE) and Relative RMSE (rRMSE) of the resultant GRF (tri-axial GRF) estimated from fifteen segmental accelerations, for several dynamic tasks. Values are means $\pm$ one standard deviation. (Verheul, 2019).	52
6	Mean $\pm$ standard deviations for root mean square errors (RMSE) and relative root mean square errors (rRMSE) of the modelled GRF profiles for different tasks. (Verheul, 2021).	59
7	Model performance for the direct model in the different directions. (Scheltinga, 2023).	62
8	Model performance for the hybrid model in the different directions. (Scheltinga, 2023).	64
9	Overview of dynamic model accuracies. In case of multiple speeds and movements the average accuracy is reported.	66
10	Overview of data-driven model accuracies.	67
11	Reference body dimensions for men and women, as reported in the study by Paolo de Leva.	70
12	Adjusted weighting factors, which are based on the reported values in the tables of De Leva [7] and Winter [13].	74
13	Weight factors for the analysis of the segments' importance	77
14	Reference values of inertia tensor from literature [39].	80
15	Overview of Pearson Correlation Coefficients to determine the similarity between the accelerations measured by the IMUs and derived from reflective markers.	88
16	Overview of Pearson Correlation Coefficients to determine the similarity between the angular velocities measured by the IMUs and derived from reflective markers.	89
17	Overview of the detected steps for step detector 1, step detector 3 and the steps from the control algorithm.	93
18	Performance of jump detector	94

19	Performance of the link-segment model to estimate the Ground Reaction Force for running and jumping. The mean and standard deviation were calculated across the seven participants.	101
20	Performance of each weighting factor on estimating the loading rate and the impulse of the Ground Reaction Force. The best weighting factor is indicated in blue.	103
21	Performance of each weighting factor on estimating the peak and the general shape of the Ground Reaction Force (rRMSE). The best weighting factor is indicated in blue.	104
22	Best combinations of segments with the lowest mean root mean square errors (RMSE) and relative root mean square error (rRMSE) across participants (6) and tasks (running and jumping).	107
23	Explained variance of principal components for running.	108
24	Explained variance of principal components for jumping.	111
25	Root Mean Square Error (RMSE) change (%) of the estimated GRF across different frequencies.	114
26	Overview of the results of the link-segment model to estimate the Ground Reaction Moment.	119
27	Summary of the results of the two-body model to estimate the Ground Reaction Force.	122
28	Summary of the results of the link-segment model combined with the two-body model to estimate the Ground Reaction Force.	126

List of abbreviations

The following table describes the various abbreviations used throughout the thesis.

Abbreviation	Definition
2MM	Two-body model
CoM	Centre-of-Mass
CoP	Centre-of-Pressure
eGRF	estimated Ground Reaction Force
eGRM	estimated Ground Reaction Moment
FdL	weighting factors for women tabulated by Paolo de Leva [7]
GRF	Ground Reaction Force
GRM	Ground Reaction Moment
IMU	Inertial Measurement Unit
LSM	Link-segment model
MdL	weighting factors for men tabulated by Paolo de Leva [7]
mGRF	measured Ground Reaction Force
MGRM	measured Ground Reaction Moment
MSD	Mass-spring-damper model
pGRF	principal ground reaction force
RMSE	Root Mean Square Error
rRMSE	relative Root Mean Square Error
SGD	Stochastic gradient descent
W	weighting factors tabulated by David Winter [13]

Table 1: List of abbreviations

1 Introduction

Saturday 12th August 2017. It is the final of the 4x100m World Championships at the London Olympic Stadium in the United Kingdom. Usain Bolt, currently the fastest man in the whole world, is the last runner for Jamaica's relays team. He starts in third position, a few steps behind the two athletes in front of him. The crowd watches with bated breath to see what Bolt will do. He approaches his opponents, but 20 meters before the finish line, his race goes totally wrong... Bolt grasps his left hamstring, while his leg can no longer support any weight. After refusing help, he drags himself back onto his feet and crosses the finish line despite being in visible pain. Later, it turned out that he had suffered a hamstring injury during the race [14].

What if sports injuries could be prevented? Most sports injuries are not caused by contact, but are the consequence of a too low or a too high training load [15]. Monitoring and understanding how the biomechanical load on the body causes the muscles, tendons and bones to change is a key factor in discovering the mechanisms behind injuries [16]. The Ground Reaction Force (GRF) provides a general indication of the biomechanical load on the musculoskeletal system. It is defined as the force that the ground exerts on a body in contact with it [17]. Current methods for measuring the Ground Reaction Force are constrained to laboratory instrumentation, are labour-intensive, costly, and require highly-trained specialist knowledge [2]. The recent development of low-cost wearable sensors offers the opportunity to evaluate the biomechanical load on the body outside the laboratory environment, for example during training. With the kinematic data of inertial measurement units (IMUs) as input, a physical or data-driven model can make an estimation of the Ground Reaction Force [18].

In this study, five IMUs are integrated into sensorised running pants to make an estimation of the GRF with a self-designed model. The sensorised running pants (figure 4) are developed at the BruBotics Research Group of the Vrije Universiteit Brussel [16]. The prototype contains five inertial measurement units with nine degrees-of-freedom (BHI260AP, Bosh, Germany), which are placed at the pelvis, thighs and shanks. The ultimate goal is to prevent injuries by monitoring the training load in the field with the GRF estimation from data of the running pants.

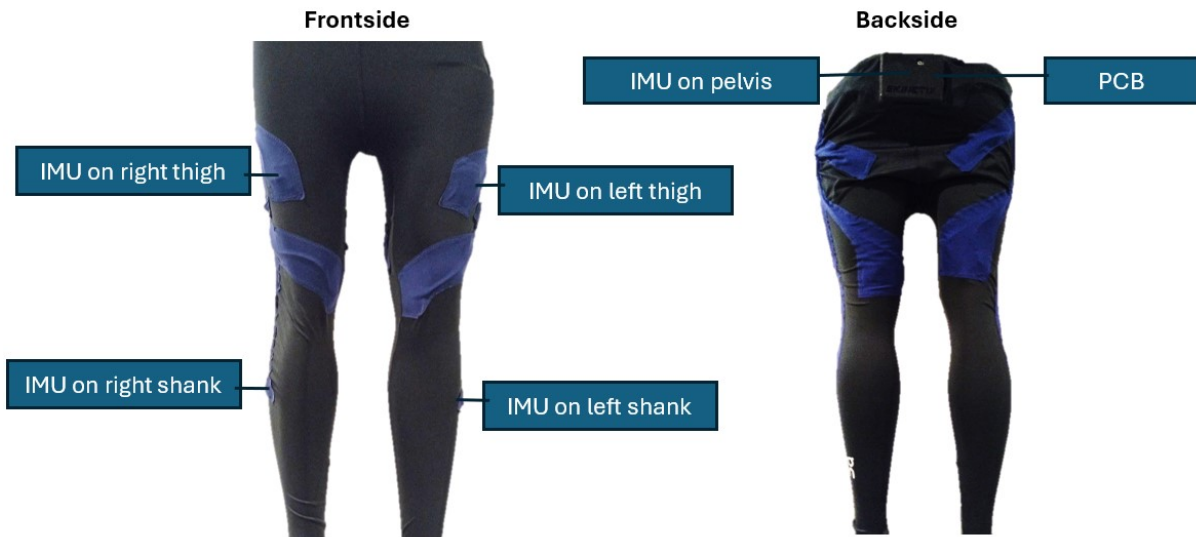


Figure 4: Sensorised running pants, with one IMU attached on the pelvis, one IMU on each thigh and one IMU on each shank.

1.1 Objectives

This thesis investigates wearable-sensor based methods to estimate the Ground Reaction Force in the field and is structured according to three objectives. The first objective is to synthesise the literature on IMU-based GRF estimation models. The second objective is to develop a new GRF estimation model that uses the kinematic data from five IMUs integrated into sensorised running pants. Based on the literature study, two models with two different approaches will be developed.

- **Model 1: link-segment model, in combination with a two-body model**
In a link-segment model, the human body is represented as a chain of rigid body segments that are linked by joints. By applying the second law of Newton to each individual body segment, the Ground Reaction Force is estimated. This model can be used for almost all movements, and can estimate the ground reaction force in the medio-lateral, anterior-posterior and vertical direction. The combination with a two-body model ensures that the impact peak of the vertical GRF is better estimated during running. In addition, this model can be used to calculate the ground reaction moment.
- **Model 2: mass-spring-damper model**
The human body behaves in a spring-like manner during bouncing gaits such as running and hopping. Therefore, a mass-spring-damper model describes the body as a series of masses that are connected with springs and dampers. This model can only estimate the GRF in the vertical direction.

The models are validated by comparing the estimated GRF to the measured GRF of an instrumented treadmill. Such a treadmill is equipped with two force plates and is the gold standard method for measuring forces, but it is restricted to laboratory environments. After the valida-

tion, the best model will be implemented. The chosen model will be further refined. Since the literature study showed that increasing the number of segments, for example adding an IMU on the trunk or foot, and optimising the filter settings of the IMUs and the weighting factors of the body segments can further increase the performance of the model, the influence of these adjustments on the developed model will be investigated [8] [17].

Because the accuracy of the proposed model should meet the expectations for the use of sensorised running pants in an injury prevention test, the third objective is to obtain a final model with a relative root mean square error (rRMSE) below 10 % for the following sagittal motions: running at a constant speed of 8 km/h, 10 km/h and 12 km/h, and jumping. This value is chosen because meaningful performance or injury related differences in loading characteristics can be as small as 3 to 10 percent of the GRF [8].

1.2 Outline Thesis

Chapter 1: Medical Background and State-of-the-art

Firstly, the medical background about the monitoring of training loads is discussed. One of the parameters is the Ground Reaction Force. Therefore, the characteristic shape of the Ground Reaction Force for running and jumping is discussed in more detail. Finally, the current state of Ground Reaction Force estimation models is explored and the various studies are compared with each other.

Chapter 2: Methods

In this chapter, two models are developed to estimate the Ground Reaction Force, with the accelerations measured by inertial measurement units on sensorised running pants as the main inputs. The Ground Reaction Moment (GRM) and the Centre-of-Pressure (CoP) are also estimated. With the information from the state-of-the-art as inspiration, two models have been selected for further investigation: a link-segment model combined with a two-body model and a mass-spring damper model.

Chapter 3: Results

This chapter provides a discussion on the accuracy of the wearable-sensor based estimation models that have been created. The primary metric is the relative root mean square error, which calculates the average error between the estimated GRF of the model and the measured GRF obtained with an instrumented treadmill. The evaluation of the estimation for the GRM and the CoP is performed in an analogous manner. For the GRF of running, there are several specific metrics that are additionally defined: the relative loading rate error, the relative impulse error and the relative peak error. Finally, an attempt is made to further optimise the models by investigating the influence of the number of segments and the filter cut-off frequency.

Chapter 4: Discussion

This chapter discusses the limitations of this research, and provides a basis for further improvements.

Chapter 5: Conclusion

Finally, a general conclusion is drawn on which wearable-based GRF estimation model is the most accurate, and which adjustments are the most interesting.

2 Medical background

This chapter provides the required medical background for a better understanding of this thesis. First, some frequently occurring terms are defined. Then it is explained how the training load can be monitored, with the emphasis on the Ground Reaction Force. In particular, the definition of the Ground Reaction Force, measuring methods and characteristic patterns for running and jumping movements are described.

2.1 Definitions

Movement analysis is the assessment of a person's motion, and includes the examination of kinetics and kinematics (see figure 5). The difference between kinetics and kinematics lies in the different perspective used to analyse the same motion. Kinematics describes the motion of body segments without considering their masses or causal forces. It is quantified with either linear variables (linear position, linear velocity and linear acceleration) or angular variables (angular position or orientation, angular velocity and angular acceleration). Kinetics focusses on the relation between the motion and its causes, by quantifying the forces (linear cause) and moments (angular cause) applied to the segments and joints of the human body [19].

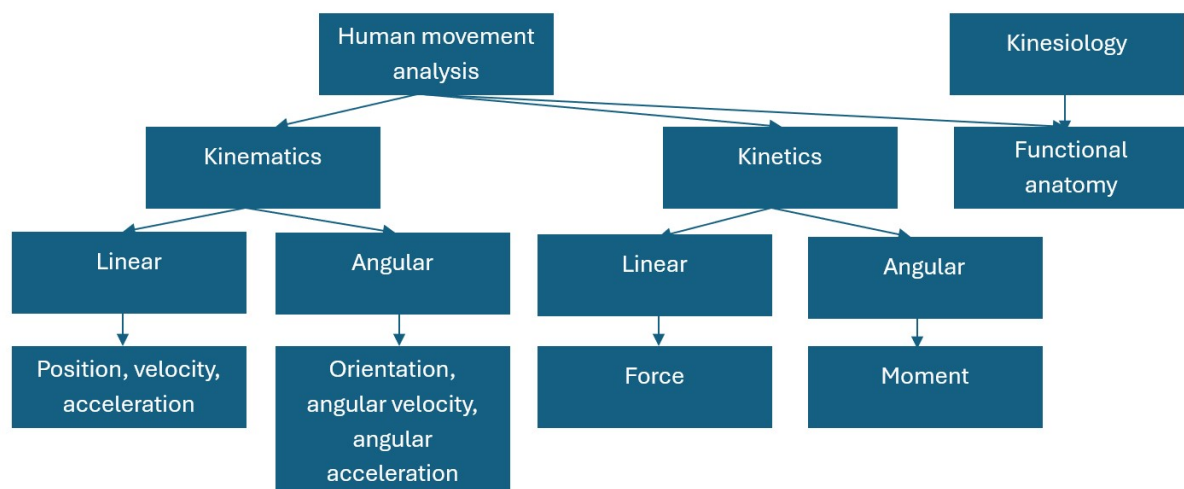


Figure 5: The components of movement analysis.

In anatomical terminology, the anterior and posterior directions (y) refer to the front and back of the body, respectively. The medio-lateral direction (x) refers to the midline of the body, and the vertical direction (z) is along the vertical axis of the body [11]. The directions are illustrated in figure 6.

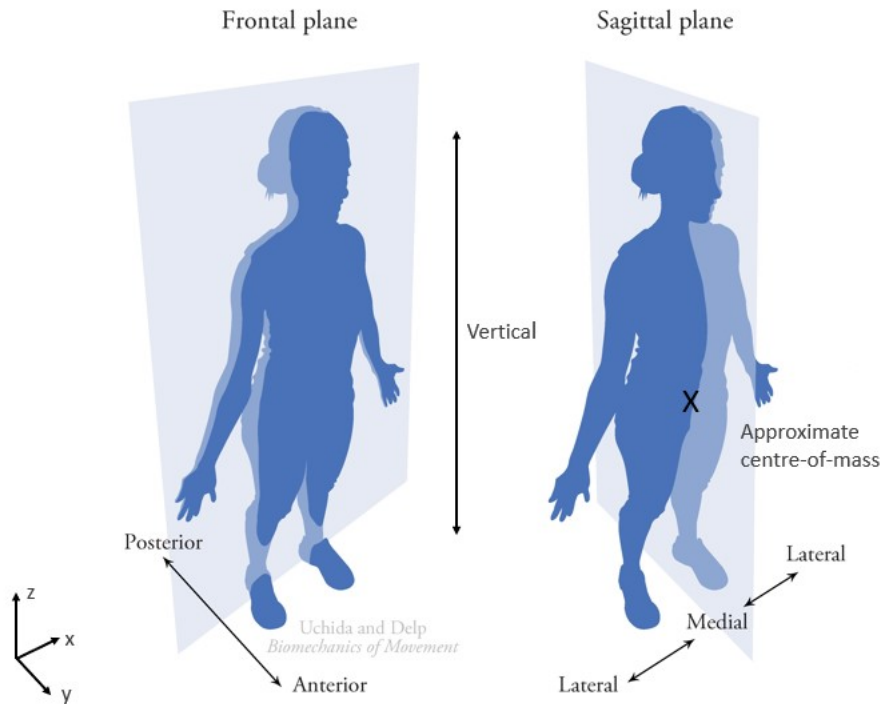


Figure 6: Anatomical terms of location, labelled on the human body.

2.2 Training load monitoring

Training load monitoring is essential in sports to enhance performance and reduce the risk of injury. When athletes are challenged to an appropriate level during training, their bodies experience external and internal loads. External load refers to the physical work performed, while internal load refers to the body's response to that work. However, excessive loading can lead to fatigue and injuries, particularly overuse injuries in the lower limbs [20]. In order to manage this balance, training load is commonly assessed through two distinct but complementary pathways: the physiological and the biomechanical load-response pathway. An overview of the training load monitoring framework is shown in figure 7.

- Physiological load-response pathway

The physiological pathway focuses on metabolic and cardio-respiratory adaptations, with common internal measures including oxygen uptake ($\dot{V}O_2$), heart rate, and blood lactate, and external measures such as speed and distance.

- Biomechanical load-response pathway

The biomechanical pathway focuses on how muscles, tendons, ligaments, bones, and cartilage adapt to mechanical stress. Soft tissue is compressed by absorbing braking forces due to the surface impact, and extended by generating propulsive forces to push off against the ground. When this cycle of compressive and extensive stresses on the tissue continues for a long time, it will adapt by becoming stronger or weaker in some places. It is challenging to measure the internal biomechanical load. In most cases, this

is done using invasive techniques or complex musculoskeletal modelling. The external biomechanical load is often assessed using the Ground Reaction Force (GRF), a well-established indicator of whole-body loading [21] [1]. Because this research focuses on the Ground Reaction Force, it will be explained in more detail in the next section.

The physiological and biomechanical load-response pathway offer unique insights into how athletes adapt to training, and monitoring them provides a more complete understanding of an athlete's performance potential and injury risk. To illustrate both pathways, one could make the analogy with with Usain Bolt's sprinting. Physiological adaptation refers to the body's energy metabolism, where the main focus is on the efficient uptake of oxygen and the consecutive release of energy. The biomechanical adaptation involves maintaining the mechanical properties of the musculoskeletal system [2].

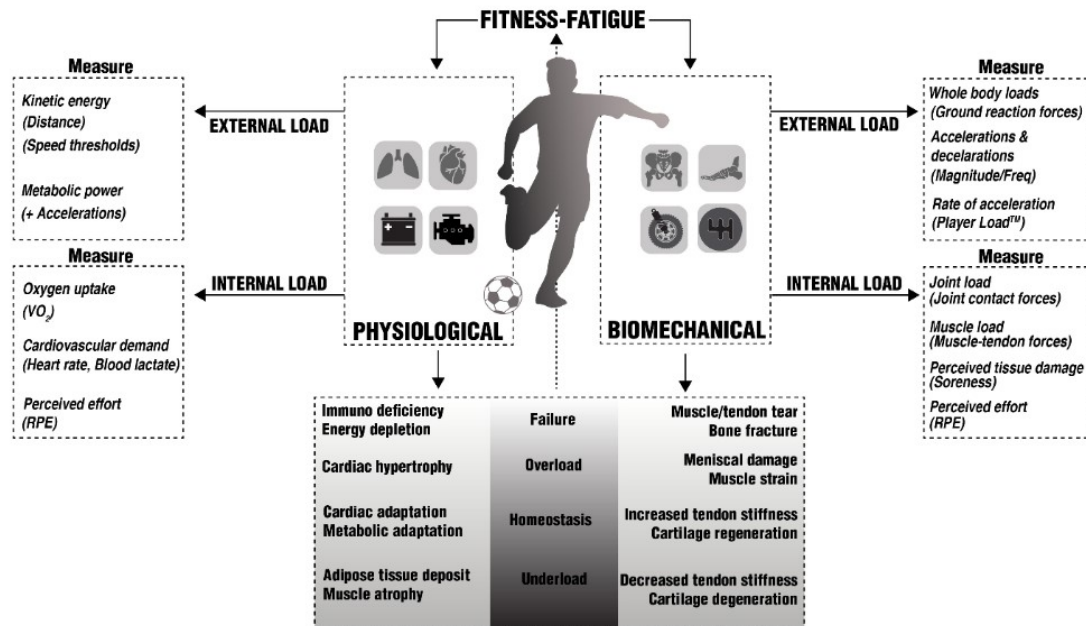


Figure 7: Load monitoring framework, demonstrating the duality of the physiological and biomechanical load-response pathway.

2.3 Ground Reaction Force

According to the definition, the Ground Reaction Force (GRF) is the force exerted by the ground on a body in contact with it. Because the Ground Reaction Force is a three-dimensional force, it can be represented as a three-dimensional vector with a direction and a magnitude. The vector is typically decomposed in one component for each dimension: a vertical component, an anterior-posterior component, and a medio-lateral component. In the context of training load monitoring, the focus has predominantly been on the vertical component of the Ground Reaction Force [17]. In addition to the direction and the magnitude, each vector has a point of application which is defined as the point at which the force exerts its effect. For the Ground

Reaction Force, this point is called the Centre-of-Pressure (CoP), and it is located on the foot sole [13].

A schematic representation of the Ground Reaction Force is shown in figure 8. When Usain Bolt is sprinting, his foot exerts a contact force on the ground to propel himself forwards. The third law of Newton states that for every action, there is an equal and opposite reaction. Consequently, at the same time when Bolt is pushing with his foot on the ground, an equal and opposite Ground Reaction Force is exerted by the ground on his body.

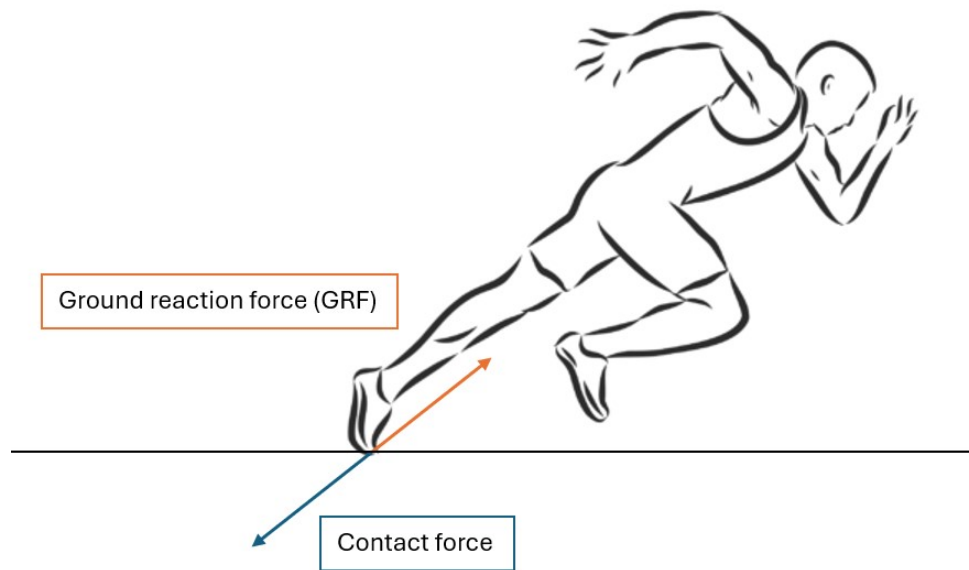


Figure 8: Schematic representation of the Ground Reaction Force.

The main interaction between a subject and the environment is when the subject's feet are in contact with the ground. In that case, the Ground Reaction Force is interpreted as a generic indicator of the cumulative external loading experienced by the whole musculoskeletal system during physical activity. Therefore, assessing the joint loads requires the assessment of Ground Reaction Forces and Moments [19]. Monitoring an athlete's GRF during running can help to better understand the relationship between the external forces experienced by the body and the tissue adaptations as a consequence of this forces, ultimately helping to prevent musculoskeletal injury. However, researchers and practitioners should be very cautious to extrapolate the effect to the internal load component of biomechanical load. The GRF may even be a poor predictor of the loads experienced at a structural level [8].

2.3.1 Measuring and estimating the Ground Reaction Force

2.3.1.1 Laboratory methods

Traditionally, the Ground Reaction Force (GRF) has been measured in a (gait) laboratory. It is commonly equipped with optical motion capture systems and force plates, which measure the kinematics and kinetics respectively. A force plate is a rigid, flat platform with piezoelectric crystals or strain gauges that convert small platform displacements into voltages, which are proportional to the force. Although force plates are a bit like a regular spring-based bathroom scale, there are major differences in precision and speed. Because the Ground Reaction Forces can change rapidly, especially at the beginning and ending of foot contact, the data are collected at a rate of thousands of measurements per second. Furthermore, the force is reported along three orthogonal axes, rather than just one as with a bathroom scale [11]. A pressure sensor is similar to a force sensor, but measures the pressure, a scalar value, instead of the force, a vector. Both sensors can be integrated into the floor or a treadmill to measure GRF during movements. They are highly accurate, with reported sub-Newton accuracy [13]. The gold standard in the medical world is currently a force plate in the floor. A force plate is often used in combination with markers, also referred to as optical motion capture, to obtain kinematic data in addition to kinetic data. An illustration of both measurement techniques is shown in figure 9. However, these methods have certain limitations. They are hindered by their high cost, lack of portability, and inability to capture data in natural settings [11].

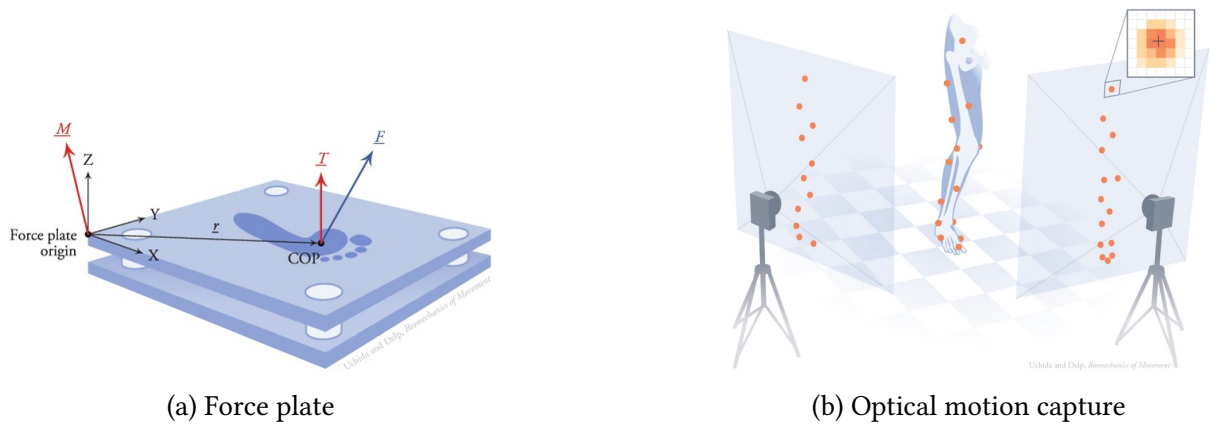


Figure 9: Illustration of a force plate and optical motion capture.

2.3.1.2 Ambulatory methods

Various approaches have been explored to estimate the GRF from wearable devices in the field to overcome these obstacles.

A first, intuitive method is by using instrumented insoles. These insoles are typically worn in or under the shoe and provide a summation of the pressure that the foot exerts on the ground. Although instrumented insoles can estimate GRF for running and jumping fairly well, their limited accuracy for high-intensity movements, high cost, technical issues and practical difficulties like movement restrictions, added mass in the shoe and discomfort, leaves the feasibility

of using insoles for monitoring GRF on a large-scale in the field currently still questionable [2].

The second method uses inertial measurement units [17]. With the development of novel data processing approaches and deep learning in recent years, the inertial measurement unit (IMU) is one of the most popular and promising wearable sensors to monitor biomechanical loading during sports. A schematic representation of an IMU is shown in figure 10. It is typically composed of an accelerometer, a gyroscope, and a magnetometer [18].

- An accelerometer is a motion sensor that measures linear acceleration, which is the rate of change in velocity of an object relative to a local inertial reference frame [18].
- Angular rate sensors, more commonly known as gyroscopes, indicate the angular velocity. The angular velocity is the rate of rotation about an axis [18].
- A magnetometer is used to detect and measure the strength and direction of the Earth's magnetic field. As such, the magnetic heading of the IMU is described, which is the three-dimensional orientation of the IMU with respect to the magnetic north. This information allows to correct for gyroscopic drift and to improve the stability in determining orientation changes [18] [22].

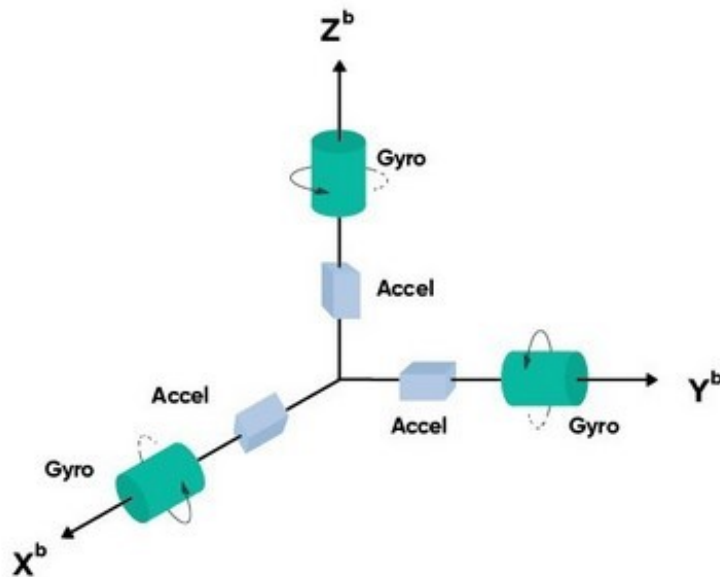


Figure 10: Schematic representation of an IMU. An IMU is composed of an accelerometer, a gyroscope and a magnetometer.

With the information of these three components, IMUs are able to directly measure its orientation, and the linear acceleration and the angular velocity of the segment where it is attached to, for example the shank of a human body [11].

Some advantages of IMUs are their low weight, cost effectiveness, convenience of use, and accessibility for the general population when compared with a laboratory setup [18].

A disadvantage is that the components of an IMU are highly sensitive. The most prominent source of error is due to the reliance on the magnetometer. Typically, the determination of the IMU's orientation assumes that the magnetic field within the environment of use is homogeneous. However, this assumption is shown to be invalid for several indoor environments, for example in the presence of ferromagnetic material which can cause electromagnetic interference. A secondary source of error is that the accuracy of the accelerometer and gyroscope is sensitive to temperature, due to physical expansion or contraction of microscopic components [18]. Therefore, the calibration of the IMU is critical to ensure that the output from the sensors is accurate and repeatable within the specified operating conditions. The sensor should output the same result, or very close to it, each time the IMU is subjected to the same inertial conditions at the same temperature.

Inertial measurement units cannot measure the Ground Reaction Force directly, as do the force plate and pressure sensors. Instead, the Ground Reaction Force is estimated by a model with as input the variables that the IMUs measure, and as output an approximation of the Ground Reaction Force. The accuracy of the IMUs and the model is therefore of great importance to obtain a reliable result that is similar to the measured Ground Reaction Force. As an indication of the required accuracy, meaningful performance or injury related differences in loading characteristics can be as small as 3 to 10 percent of the GRF [8]. Despite the difficulty of first creating a model, it is certainly interesting to use wearable sensors. They are needed to go beyond the limits of traditional lab-based methods, in order to monitor training load when it is happening in the field, not just when it is being tracked in a lab [2].

2.3.2 Characteristics of the Ground Reaction Force

Typical human movements have characteristic Ground Reaction Force patterns that are quite different from each other. This thesis focuses on the Ground Reaction Force during running and jumping. It is important to note that all these activities take place in the sagittal plane. Consequently, the vertical GRF has the highest magnitude, while the medio-lateral and anterior-posterior GRF are smaller.

2.3.2.1 GRF in walking

To clarify the GRF pattern of running, it is best to start with walking for better understanding. Walking is a cyclic pattern of movement, where one walking gait cycle is defined by two consecutive foot contact events on the same leg. Each leg has a stance phase and a swing phase, which are further divided into initial contact, loading response, midstance, terminal stance, preswing, initial swing, mid swing and terminal swing. These phases are demarcated by different gait events: toe contact, foot flat, heel off, toe off and peak knee flexion, as shown in figure 11. The stance phase occupies 60% of the gait cycle, and during this phase the foot is in contact with the ground. The swing phase occupies 40% of the gait cycle, with the foot

off the ground and the body weight supported by the other leg. Because the stance phase is longer than the swing phase, both feet are in contact with the ground at the same time for 20% of one gait cycle. This is called double-support. The remaining time is in the single support phase, when only one foot is in contact with the ground.

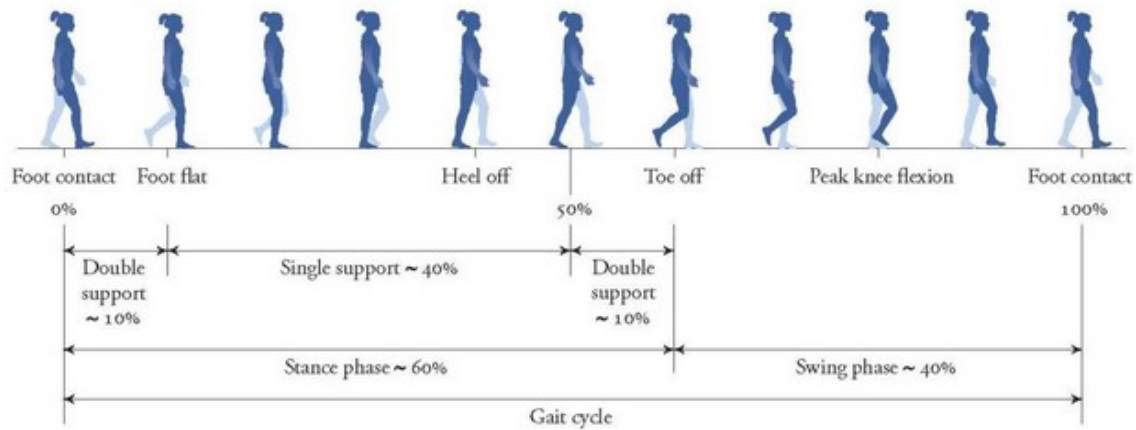


Figure 11: Schematic representation of one gait cycle, which is defined by two consecutive foot contact events on the same leg.

Recordings of the Ground Reaction Force during walking show several features, as seen in figure 12). The vertical component of the Ground Reaction Force rises rapidly after foot contact and reaches a force greater than body weight at about 10% of the gait cycle. The vertical force falls below body weight at midstance and then rises again above body weight during pushing off. The vertical force falls to zero afterwards, because after toe-off, the foot is in the swing phase and therefore no longer in contact with the ground. On average, the total Ground Reaction Force is equal to one time the body weight.

The anterior-posterior Ground Reaction Force is directed towards the back of the body, decelerating or 'braking' the body during the first half of stance. Afterwards, it is directed towards the front of the body until the swing phase. As a result, the body is accelerated or 'propelled' forward [11].

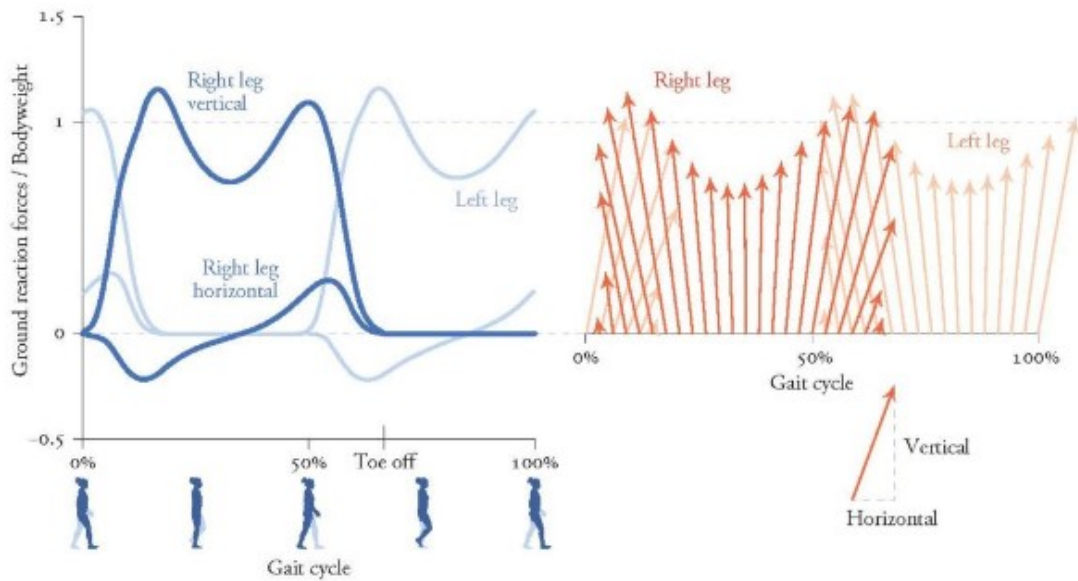


Figure 12: Recording of the Ground Reaction Force during one gait cycle.

The within-subject variability of the Ground Reaction Force is quite low: the coefficient of variation for the anterior-posterior direction is 20 % and for the vertical direction 7 %. The vertical Ground Reaction Force is dominated by gravitational forces, but the muscularly generated accelerations in both vertical and anterior-posterior directions show a consistent net pattern. The variability of the Ground Reaction Force increases with increasing walking speed. It is suggested that there is "an optimum speed" at 5.5 - 5.8 km/h for the neuromuscular locomotor system at which variability is minimum, but only for the propulsion control mechanism [23].

2.3.2.2 GRF in running

As a person walks faster and faster, the duration of the stance phase decreases until the stance phase corresponding to each leg will eventually span less than half the gait cycle, giving rise to a flight phase. During this flight phase, both feet leave the ground. Figure 13 depicts one running cycle. Similar to walking, a running cycle is divided into a stance phase and a swing phase. The stance phase begins at foot contact, ends at toe-off, and accounts for about 30 % to 45 % of the gait cycle in running, although it may account for 25 % or less during high-speed sprinting.

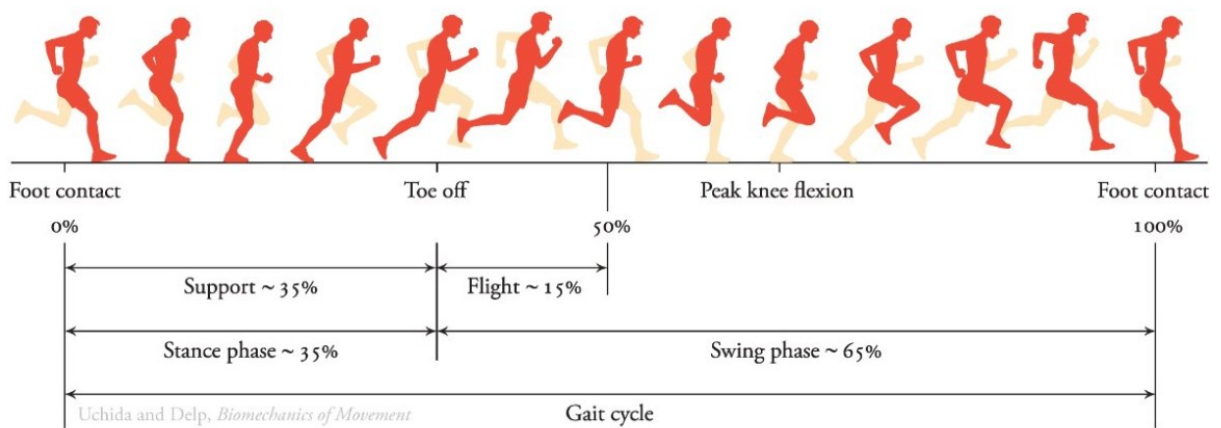


Figure 13: Schematic representation of one running cycle.

What drives the transition between gaits? When moving at a particular speed, humans typically select the mode of locomotion with the lowest energy cost. Consequently, they transition from walking to running around the speed at which running becomes more economical. The transition from walking to running is also visible in the recordings of the Ground Reaction Force, as shown in figure 14. The increasing speed results in a shorter support phase duration, which leads to a change in the shape of the Ground Reaction Force curve (figure 15). The vertical peak GRF will increase, whereas the anterior-posterior peak forces decrease. The higher the speed, the higher the Ground Reaction Force will be and the more impact the body has to absorb. At each foot landing, the body has to absorb an impact of about three the body weight. The impact forces in running are therefore much higher than in walking, explaining why more injuries occur during fast running than walking [11].

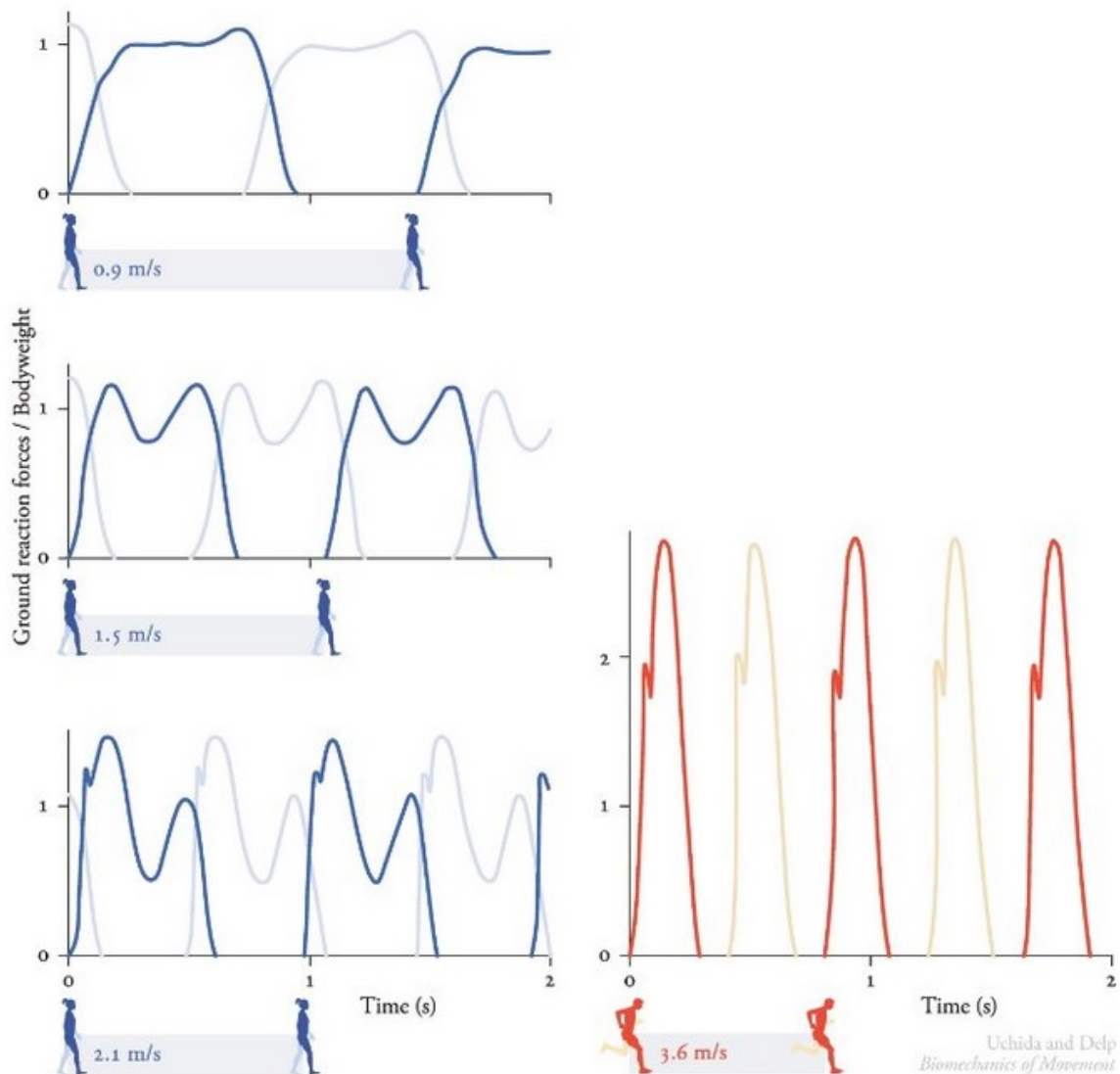


Figure 14: The transition from walking (blue) to running (red) is visible in the recordings of the Ground Reaction Force.

When the vertical Ground Reaction Force recording is investigated in more detail, as shown in figure 14 (red graph on the right) and figure 15), one can conclude that the vertical component of the GRF includes two peaks. The first peak is called the impact peak, and the second peak is the active peak. Both are important load measures, related to overuse injuries in running. The higher the load, the higher the risk of injuries. The impact peak is a high-frequency component of the GRF, that results from the heel that hits the ground. It occurs at about 15 to 20 percent of the gait cycle. The interpretation of the impact peak is mainly related to tibial stress fractures and plantar fasciitis [24].

The active peak or peak GRF is a low-frequency component and it is influenced by neuromuscular feedback. It is the maximal force the body experiences at the moment the foot bears the entire body, and it is expressed in body weight. Typically, the vertical Ground Reaction Force peaks at approximately two or three times the bodyweight when running at a moderate speed.

The peak GRF is mainly related to calf, quadriceps, ankle, and knee injuries [25]. Whether or not both force peaks are visible in the recording of the vertical GRF depends on the foot strike pattern and the footwear. For example, if the runner strikes the ground on the forefoot, the vertical GRF raises more gradually and it may be more difficult to identify the impact peak. It has been argued that this style of running is more 'natural' and protects against impact-related injuries, but subsequent research has cast doubt on this conclusion [17] [25].

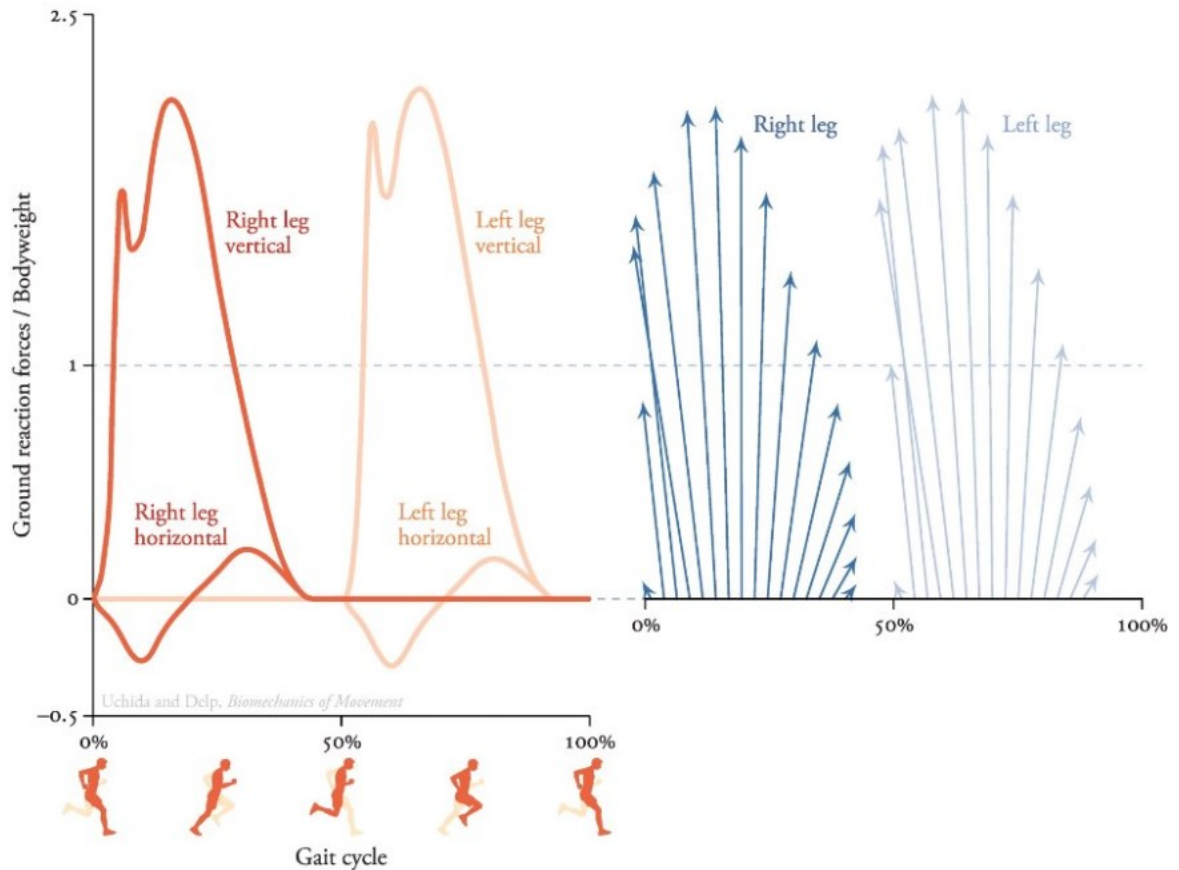


Figure 15: GRF during running.

Research into the variability of the Ground Reaction Force shows that a 'movement stereotype' for running exists. Differences between experienced and inexperienced runners are small for the shock absorption variables, which are the magnitude of the vertical force impact peak, and the force load rate leading to this peak, but marked for the running style variables. Inexperienced runners demonstrated greater braking and propulsion impulses, and greater variability in these two variables, indicating that they are using a less economical action.

2.3.2.3 GRF in jumping

Figure 16 shows a schematic representation of the countermovement jump. First, a person descends into a jump. During this phase (a), he bends his knees, swings his arms and prepares

the jump by lowering the centre-of-mass. From a sports science perspective, the velocity of descent prior to the jump is a key metric.

In the second part of the countermovement (b), the person has to decelerate the downward movement in order to reach a short moment of pause at the deepest position of the jump.

The third phase (c-f) begins with the person at the bottom of the jump, as he begins raising upwards towards take-off. He then accelerates further until his feet leave the ground and there are no more Ground Reaction Forces measurable.

During the flight phase (g), the subject can't impact the velocity of his centre-of-mass any further. The height of the jump has been predetermined by the build up of speed before and during take-off. The only force that is now acting upon the subject is the gravity that is pulling him back down.

Finally, the subject lands (h) on the ground with exactly the same velocity during landing as during take-off. GRF jumping

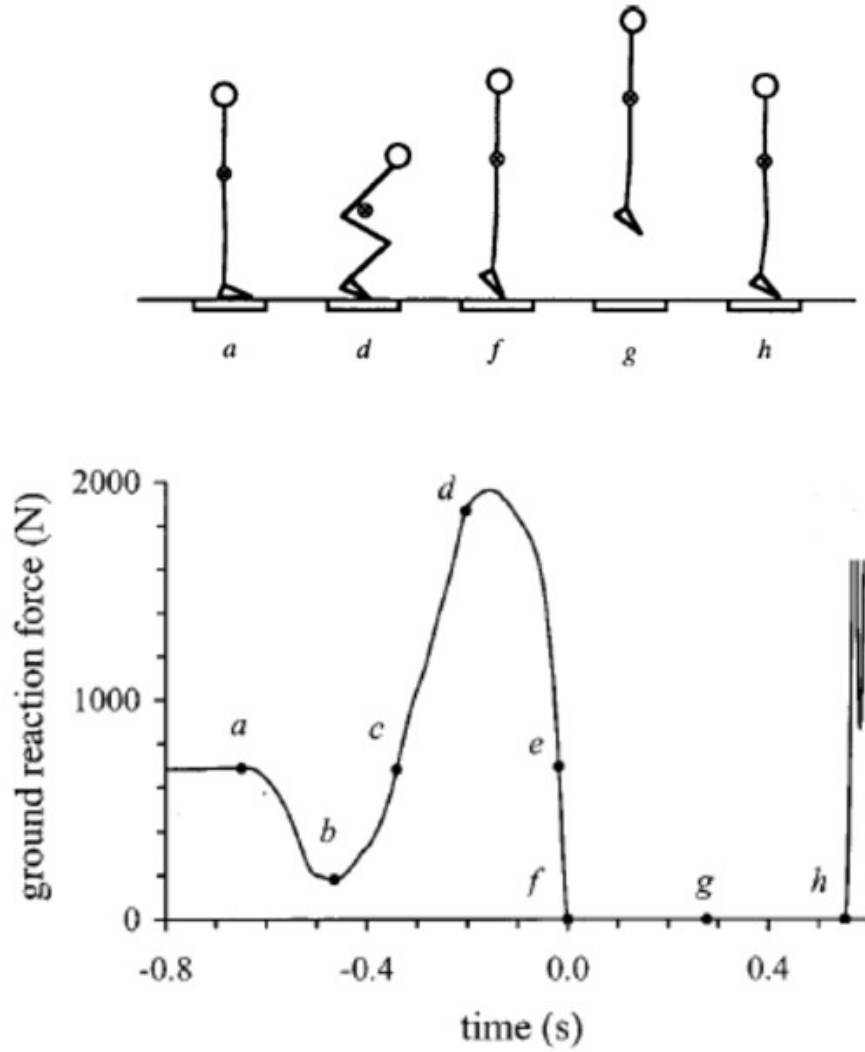


Figure 16: Schematic representation of the Ground Reaction Force of a countermovement jump

3 State-of-the-art

This chapter synthesises the literature on IMU-based GRF estimation models, as described in the first objective of this thesis. First, the metrics to evaluate the different models are discussed. Next, the GRF estimation methods are synthesised and discussed in more detail.

3.1 Model Validation

By determining the accuracy of each model, the most suitable model can be selected. Various metrics exist to compare the performance of different models in terms of accuracy. Some metrics provide a general idea of how correct the fitting is, for example the (relative) root mean square error and the pearson correlation coefficient. Other metrics are more related to specific features of the GRF during running, such as the peak error, loading rate error and impulse error. A difficulty is that not all metrics are reported in studies, which makes it difficult to

compare different models. In addition, the definition of the variables can differ between papers, which makes the comparison even more difficult. To avoid confusion, the metrics are defined below as they will be used in this thesis.

The first metric is the root mean square error (RMSE) between the measured GRF (mGRF) and the estimated GRF (eGRF). It is defined as the root of the quadratic difference between both variables. A value of zero would indicate a perfect fit, which is almost never achieved in practice. The RMSE is expressed in body weight (BW) or Newton/kilogram (N/kg), depending on the study. Because the RMSE quantifies the average magnitude of errors in the same units as the target variable, it allows to easily understand how far predictions deviate from actual values. Typically, the estimated Ground Reaction Force of the model and the measured Ground Reaction Force are compared over the stance phase. Some studies also include the flight phase in the comparison, but this phase is easy to predict as the GRF equals zero Newton, and it results in a 'better' performance than the actual one [17]

$$RMSE = \sqrt{\frac{\sum_{n=1}^N (mGRF_n - eGRF_n)^2}{N}} \quad (7)$$

where: N : sample size

$mGRF_n, eGRF_n$: the individual sample points, indexed with n

The RMSE depends on the scale of the values of the measured GRF waveform. Normalising the RMSE facilitates the comparison between the GRF estimation in all three directions, that have different scales. Although there is no consistent definition of normalisation in the literature, a common choice is dividing the RMSE by the range of the measured data, expressed as the maximum value minus the minimum value. The normalised RMSE is commonly referred to as the relative RMSE (rRMSE). The unit of the rRMSE is percent.

$$rRMSE = \frac{RMSE}{\max(mGRF) - \min(mGRF)} * 100 \quad (8)$$

In addition to the RMSE and rRMSE as a general metric of the accuracy, the Pearson correlation coefficient is used in some studies. It is a statistical metric that expresses the linear correlation between two sets of data. Because it is essentially a normalised measurement of the covariance (see equation 9, it has always a value between -1 and 1. A Pearson correlation coefficient equal to 1 corresponds to a perfect fit. A value of zero implies that there is no linear dependency between the measurement and the estimation.

$$\rho = \frac{\sum_{n=1}^N (eGRF_n - \overline{eGRF})(mGRF_n - \overline{mGRF})}{\sqrt{\sum_{n=1}^N eGRF_n^2 - N\overline{eGRF}^2} \sqrt{\sum_{n=1}^N mGRF_n^2 - N\overline{mGRF}^2}} [-] \quad (9)$$

where: $\overline{mGRF} = \frac{1}{N} \sum_{n=1}^N mGRF_n$: the sample mean (analogous for $\overline{eGRF}$)

Next, more running-related metrics are discussed. Specifically for the application of injury prevention, it is important that the features defining the injury risk are correctly estimated.

A first measure is the absolute peak error. It is defined as the absolute value of the difference between the estimated and measured active peak Ground Reaction Force in the vertical direction (equation 10). To allow for comparison of different GRF magnitudes, a relative peak error is defined in equation 11[26].

$$\text{absolute peak error} = |eGRF_{active\ peak} - mGRF_{active\ peak}| \quad (10)$$

$$\text{relative peak error} = \frac{|eGRF_{active\ peak} - mGRF_{active\ peak}|}{mGRF_{active\ peak}} \cdot 100 \quad (11)$$

A second running-related metric is the loading rate error [22]. The loading rate is the average GRF gradient from initial contact to the impact peak. Again, an absolute and a relative loading rate error are defined, according to equations 12 and 13. For the practical implementation, the average gradient during the first 30% of the stance phase was taken when no impact peak was detectable.

$$\text{absolute loading rate error} = |eGRF_{loading\ rate} - mGRF_{loading\ rate}| \quad (12)$$

$$\text{relative loading rate error} = \frac{|eGRF_{loading\ rate} - mGRF_{loading\ rate}|}{mGRF_{loading\ rate}} \cdot 100 \quad (13)$$

A final metric is the impulse error [22]. The impulse is the area under the GRF curve. Analogous to the other metrics, the absolute impulse error and the relative impulse error are defined as follows:

$$\text{absolute impulse error} = |eGRF_{impulse} - mGRF_{impulse}| \quad (14)$$

$$\text{relative loading rate error} = \frac{|eGRF_{impulse} - mGRF_{impulse}|}{mGRF_{impulse}} \cdot 100 \quad (15)$$

3.2 Wearable-sensor based Ground Reaction Force estimation models

Various models have been proposed to estimate the Ground Reaction Force in the literature, because different movements involve distinct biomechanical processes and therefore different models. An overview of the three most common methods to develop a GRF estimation model is shown in figure 17. The model development is categorised into three main methods: the dynamic method, the data-driven method, and the hybrid method. Depending on the implementation, these three methods are subdivided into more specific models.

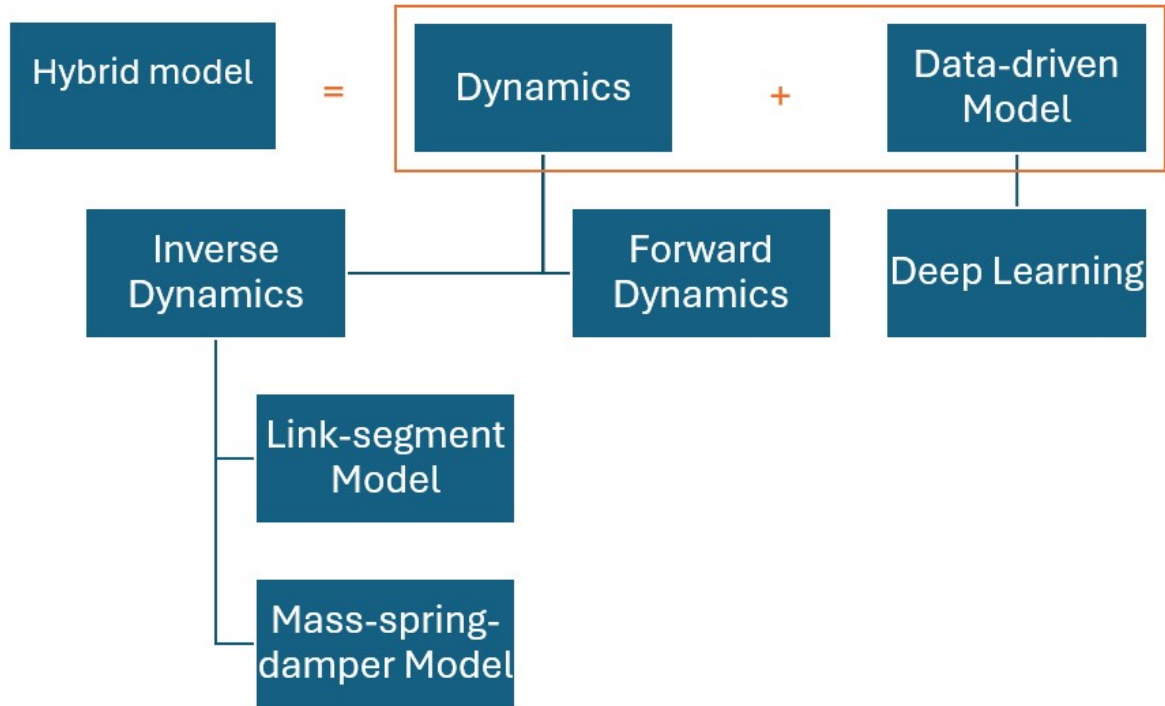


Figure 17: Overview of methods to estimate the GRF.

3.2.1 Dynamics

The first method to estimate the Ground Reaction Force during running is called 'Dynamics'. Dynamics uses Newton's laws of motion to explore the forces acting on the body. Depending on the way of applying Newton's laws, it is further subdivided into 'Inverse Dynamics' and 'Forward Dynamics'. The inverse dynamics approach is based on the assumption that the movement is known. The forces and moments required to bring about that movement, are calculated. In the direct dynamics approach, movements are calculated by integrating the equations of motion, usually based on a Lagrangian formulation. This is only possible when the joint moments of force are known, typically via optimisation techniques [27]. The big advantage of the dynamic approach is that the results are fully explainable by physics.

3.2.1.1 Inverse dynamics

In biomechanics, inverse dynamics is a technique used to calculate the internal forces and torques within a musculoskeletal system, such as muscles and ligaments, based on measured movement and external forces like Ground Reaction Forces. The Ground Reaction Force can be computed via a biomechanical model from the kinematics. Typically, the kinematics are described by IMU-measurements or reflective markers. The two most used biomechanical models are a link-segment model and a mass-spring-damper model.

3.2.1.1.1 Link-segment model

In a link segment model, the body is broken down into multiple rigid segments with their own masses and accelerations, as shown in figure 18. The segments are linked with joints. Some link-segment models represent only a few body segments, for instance the feet and the lower legs, while other link-segment models include all body segments. The model is applicable to any movement, from walking to running and jumping. It proved to be a valuable tool in the study of human gait and dynamic balance [28][13].

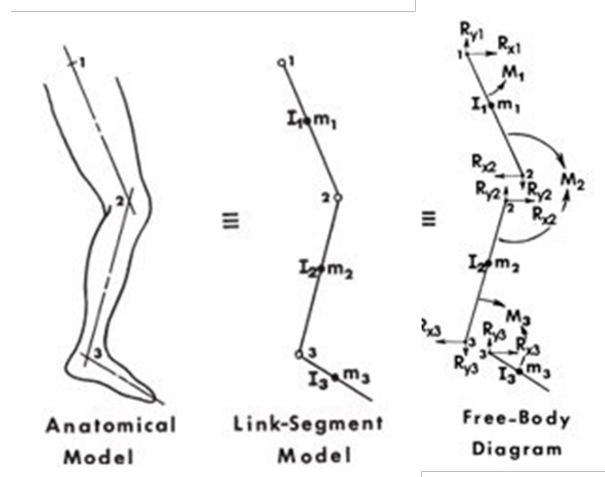


Figure 18: Link-segment model.

The link-segment model makes the following assumptions [13]:

1. Each segment has a fixed mass, which is located as a point mass at the centre of mass (CoM) of the segment. If only the vertical direction (1D) is considered in the model as a simplification of the full 3D-space, the centre of mass is often called the centre of gravity (CoG).
2. The location of the CoM of each segment remains fixed during the movement.
3. The joints are modelled as ball-and-socket joints, also called spheroid or hinge joints. A ball-and-socket joint enables the joint to move in many directions, by fitting the end of one rounded bone, forming the ball, into the end of the other bone within the joint containing a cup-like depression, forming the socket.
4. The mass moment of inertia of each segment about its rotational axis, in this case the corresponding joint axis, is constant during the movement.
5. The length of each segment remains constant during the movement. In other words, the distance between the ball-and-socket joints is fixed.

The calculation of the Ground Reaction Force is based on Newton's second law, which states that at any instant of time, the net force on a body is equal to the rate at which the body's

momentum is changing with time. Newton's second law is better known in the form of the equations of the balance of force and momentum:

- Balance of forces: $\sum F = m * a$
- Balance of momentum: $\sum M = I * \alpha$

where: m : mass of the body segment

a : (linear) acceleration of the centre-of-mass of the body segment

I : mass moment of inertia of the body segment about its centre-of-mass

α : angular acceleration of the segment

Using Newton's second law, the mass of the segments should be optimised because the mass distribution per corresponding acceleration does have an influence on the estimated Ground Reaction Force. The mass of the body segments is typically obtained as a percentage of the total body weight. The percentages or weight factors can be found in statistical tables, according to the person's height, weight, and sometimes sex. In more recent studies, the weight factors from the tables are further optimised with machine learning, where a loss function minimises the error between the mGRF and the eGRF by adjusting the weight factors [29].

3.2.1.1.1.1 Number of segments

The relationship between the measured segmental accelerations and the GRF seems to be affected by the location of the IMU, but it remains uncertain which segmental acceleration would better relate to the whole-body mechanical loading.

A study of Germanotta et al. (2021) compared the accuracy of two methods to estimate the body's centre of mass during six postural tasks. Fifteen healthy subjects took part in the study-CoMIMU. The first method collected the data by a single sensor on the pelvis, while the second method used seven sensors on the pelvis and lower limbs. The accuracy of the two methods was compared via the RMSE, using a marker-based system as a reference. Their findings showed that the RMSE of the method with one IMU was lower in tasks with little or no oscillations, while the method with seven IMUs outperformed in tasks with greater displacement of the CoM. Therefore, they conclude that a model with a high number of segments should be preferred to estimate the kinematic parameters. It should be noticed that this insight can be extrapolated to the kinetic parameters like the GRF, based on Newton's second law.

A study of Verheul et al. [8] aimed to assess the validity of estimating Ground Reaction Forces (GRF) from segmental accelerations for various running activities, more specifically running at a constant low, moderate or high speed, accelerating and decelerating, and 90° cutting. The segmental accelerations were recorded with a three-dimensional motion capture system, and the GRF profiles were estimated as the sum of the product of all fifteen segmental masses and accelerations, or a reduced number of segments. The estimated GRF was compared with the measured GRF of a single force platform. Afterwards, principal component analysis (PCA) was applied to reduce the amount of redundant information and extract the key segments of the biomechanical model.

When the GRF was estimated from a single segment, the trunk acceleration was the best predictor across tasks despite the high mean RMSE [1]. Moreover, the trunk was part of all combinations of segments, and therefore the main contributor to the GRF estimation. Intuitively, this result makes sense because the centre-of-mass of the human body is approximately at the level of the belly button. However, the PCA analysis showed that the trunk's large contribution to the GRF is primarily due to its large mass rather than the characteristics of its acceleration. The results of the PCA analysis are shown in figure 19, with in descending order of importance of the contribution, the trunk is followed by the thighs, head, shanks, arms, pelvis and feet. Although arm movements are not the primary contributors to the GRF, these segments do account for a considerable part of the overall GRF, which highlights that all segments should be considered when assessing whole-body loading [8].

#	Segments in the combination	RMSE (N·kg ⁻¹)	
		Mean	SD
1	Trunk	6.76	±3.62
2	Trunk + thigh	5.91	±3.17
3	Trunk + thighs	4.54	±2.48
4	Trunk + thighs + pelvis	4.36	±2.47
5	Trunk + thighs + pelvis + head	4.00	±1.94
6	Trunk + thighs + pelvis + shanks	3.76	±1.81
7	Trunk + thighs + shanks + head + upper arm	3.61	±1.66
8	Trunk + thighs + shanks + head + upper arm + forearm	3.49	±1.73
9	Trunk + thighs + shanks + head + upper arms + forearm	3.42	±1.75
10	Trunk + thighs + shanks + head + upper arms + forearms	3.37	±1.74
11	Trunk + thighs + shanks + head + upper arms + forearms + hand	3.31	±1.73
12	Trunk + thighs + shanks + head + upper arms + forearms + hand + foot	3.28	±1.72
13	Trunk + thighs + shanks + head + upper arms + forearms + hand + feet	3.26	±1.71
14	Trunk + thighs + shanks + head + upper arms + forearms + hands + feet	3.26	±1.71
15	Trunk + thighs + shanks + head + upper arms + forearms + hands + feet + pelvis	3.26	±1.72

Best combinations of segments (i.e. with the lowest mean root mean square errors (RMSE) across subjects, tasks and trials) for each number of segments. If only one of two segments was included in a combination (e.g. thigh or foot rather than thighs or feet), this was the segment on the side of the support leg. SD = standard deviation.

Figure 19: Best combinations of segments across all the tasks for each given number of segments.

It may not be feasible in training and competition environments to measure the accelerations of all body segments. Therefore, a well-considered choice of the measured segments should be made, so that the difference between the estimated GRF and the measured GRF is minimal. In order to achieve estimated GRF errors that were moderate or better (<3 N/kg) for at least 50 % of the combinations and trials in the study [30], a minimum of two and three segments was required for low- and moderate-speed running respectively. However, eight segments (90° cuts) and eleven segments (accelerations) were necessary for more dynamic tasks. Furthermore, for

the high-intensity tasks like decelerations and high-speed running, the majority of trials and combinations resulted in very high errors, regardless of the number of segments used. The influence of the number of segments on the accuracy is summarised in figure 20.

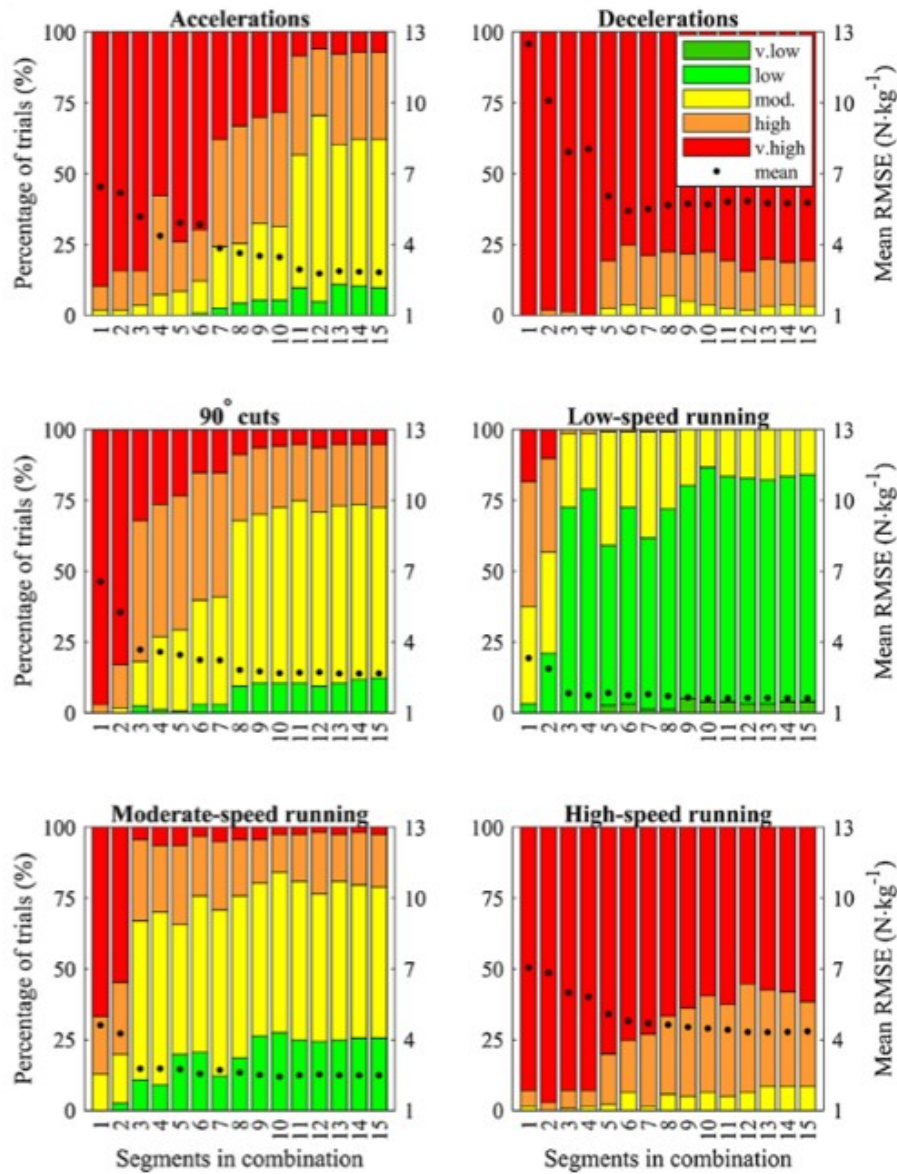


Figure 20: Root mean square errors (RMSE) for estimated resultant GRF curves for each task. Bars represent the percentage of trials (primary y-axis) within the very low (<1 N kg⁻¹), low (1-2 N kg⁻¹), moderate (2-3 N kg⁻¹), high (3-4 N kg⁻¹) or very high (>4 N kg⁻¹) error boundaries, and black dots represent the mean errors (secondary y-axis), for each given number of segments.

These results are confirmed in the study of Scheltinga (2022), where they recommend at least ten sensors. In addition, they determined the optimal sensor configuration for running tasks at a constant speed if only three sensors are used, from an initial configuration of sensors on the sternum, pelvis, upper legs, tibias and feet. The setup with sensors on both tibia and the pelvis showed the best results, with an average RMSE of 0.179 BW, a peak error of 3.6% and a

Pearson's correlation coefficient of 0.98Previous3IMU

In conclusion, an IMU only measures the acceleration of the segment it is attached to, and it is inadequate to measure the acceleration of the whole body due to the complex coordination of multiple body segments during movementsRelationshipWholeBodyExternalLoadAndAccelerometry There might be the potential of using the accelerations of body segments to estimate the GRF, but reducing the number of segments to a more feasible number for a practical setting substantially increases the GRF error. The results of both studies highlight that segmental contributions to the GRF estimation are movement dependent, and this insight could explain why link-segment models with one or a few acceleration signals cannot lead to accurate GRF estimates across different tasks. For example, one particular segment might be suitable to estimate GRF profiles for sprinting, while the same segment is less suitable to describe the GRF for decelerated running.

3.2.1.1.1.2 Model performance

The performance of the inverse dynamic approach with a link-segment model has been tested by several researchers for different motions, because the model is rather simple and Newton's equations of motion are applicable to any motion.

One of the first link-segment models to estimate GRF during walking was proposed in a study by Karatsidis et al. (2016) [3]. Seventeen IMUs were mounted on a tight-fitting Lycra suit, and the following segments were defined: head, sternum, pelvis, upper legs, lower legs, feet, shoulders, upper arms, forearms and hands. Eleven participants walked at a self-selected normal speed, a fast speed and a slow speed, where their fast and slow speeds were at least 20% higher or lower than their mean normal speed respectively. The method was evaluated for the three walking speeds, by comparing the predicted GRF with the measurement of a force plate. The results are summarised in table 2. They demonstrate that the estimation of the GRF during walking, using only kinematic information from IMUs, is achievable with a strong correlation compared to the force plate measurements. Additionally, the results were comparable to studies using reflective marker input.

	Slow speed	Normal speed	Fast speed
	Pearson's correlation coefficient [-]		
A-P GRF	0.955	0.965	0.950
M-L GRF	0.853	0.862	0.821
vGRF	0.990	0.992	0.986
	Root Mean Square Error [N/BW]		
A-P GRF	0.036 $\pm$ 0.012	0.034 $\pm$ 0.007	0.047 $\pm$ 0.011
M-L GRF	0.018 $\pm$ 0.005	0.017 $\pm$ 0.003	0.022 $\pm$ 0.004
vGRF	0.075 $\pm$ 0.039	0.063 $\pm$ 0.035	0.090 $\pm$ 0.040
	Relative Root Mean Square Error [%]		
A-P GRF	10.4 $\pm$ 3.2	9.4 $\pm$ 2.5	10.9 $\pm$ 3.1
M-L GRF	13.8 $\pm$ 3.3	13.1 $\pm$ 2.8	14.6 $\pm$ 3.1
vGRF	6.3 $\pm$ 3.3	5.3 $\pm$ 3.1	6.9 $\pm$ 3.0

Table 2: Comparison between Ground Reaction Forces, estimated from seventeen IMUs for multiple walking speeds, and force plate measurements. (Karatsidis, 2017).

The success of Karatsidis' model encouraged other researchers to extend his approach to other movements, like running and jumping. In 2018, Raper et al. investigated the reliability and validity of one tibial accelerometer to estimate the peak vertical Ground Reaction Force from the vertical tibial accelerations during running, by comparing the results to the GRF measured by force plates. The data were obtained from ten professional athletes that ran at three constant, self-selected speeds: slow, medium and fast. His study showed that the model based on one IMU on the tibia underestimated the peak vertical force value by about 400 N. This difference was not the same for all participants, and consequently, cannot simply be added to the peak GRF. A possible explanation is that the peak vertical tibial acceleration occurs about 25 ms after first foot contact, while the peak vertical Ground Reaction Forces are suggested to occur between 50 and 150 ms after first foot contact, depending on the point of contact. It is therefore unlikely that the peak vertical Ground Reaction Force and the peak vertical tibial acceleration occur at the same time. In conclusion, one IMU on the tibia does not measure vertical GRF as compared to the standard force plates. When comparing the reliability of the model and the force plate to obtain the same peak GRF value for the same athlete running at the same speed in repeated trials, both showed excellent reliability. The intraclass correlation coefficient (ICC), a descriptive statistic that describes how strongly samples in the same group resemble each other, was calculated for this comparison. The peak Ground Reaction Force from the model had an ICC of 0.877 (95% CI: 0.825-0.917). As expected, also the force plate showed an excellent reliability with an ICC of 0.974 (95% CI: 0.964-0.982). He suggested that the reliability could be further increased by averaging the peak GRF over multiple strides.

Nevertheless, Raper's model is limited to the peak vertical Ground Reaction Force. Scheltinga et al. (2023) did create a link-segment model that can estimate the full waveform of the vertical Ground Reaction Force during running, based on 3 IMUs attached to both tibia and pelvis.

The model was validated with 12 participants, all were athletes running with a heel-strike. This last condition applies because the data were also used for another study. It is not unimportant, because as mentioned in section 2.3, foot placement has an influence on the waveform of the Ground Reaction Force. Typically, a heel-strike runner will have a higher impact peak than a runner with a flat foot. The participants ran on an instrumented treadmill in random order at 10 km/h (2.8 m/s), 12 km/h (3.3 m/s) and 14 km/h (3.9 m/s) at a preferred, a high, and a low stride frequency. They imposed different stride frequencies to make the data more diverse. The high and low stride frequencies were determined by respectively increasing or decreasing the preferred stride frequency with 10%. An overview of the model's performance is given in table 3. On average over the three running speeds, the model had an RMSE of 0.18 ± 0.03 BW and corresponding rRMSE of 6.6 ± 1.2 %. The peak error was 3.61 ± 1.80 %. The Pearson correlation coefficients between the estimated and measured values were excellent, with a value of 0.98 ± 0.01 . However, a large spread was observed between the subjects. An additional visual inspection of the GRF waveforms revealed that most errors in the estimated GRF waveform are directly after the impact peak.

	RMSE [BW]	rRMSE [%]	Pearson's correlation coefficient [-]	Peak error [%]
vGRF	0.18 ± 0.03	6.6 ± 1.2	0.98 ± 0.01	3.61 ± 1.80

Table 3: Root Mean Square Error (RMSE), Relative RMSE (rRMSE), Pearson's correlation coefficient and peak error of the vertical GRF, estimated from three segmental accelerations for running at a constant speed. Values are means $\pm$ one standard deviation. (Scheltinga, 2023).

In addition to running, jumping is another interesting movement to investigate the corresponding Ground Reaction Force for the purpose of injury prevention. A study by D'Andrea et al. (2019) examined Ground Reaction Force in football players during unilateral bounding. Twenty-five male participants performed alternate lateral jumps of increasing distance (1.2m, 1.4m, 1.6m, 1.8m and 1.9 m), with their arms flexed and close to their torso. The movement is shown in figure 21. The vertical and medio-lateral GRF were estimated using eight IMUs, attached to the feet, shanks, thighs, pelvis and trunk. The results are listed in table 4. The RMSE values indicate that the vertical GRF was estimated with higher precision than the medio-lateral GRF. Because the RMSE increases from the first to third stance phase, the model's performance to estimate the vertical GRF is related to the jumping distance. This trend is not present in the estimates of the medio-lateral GRF.

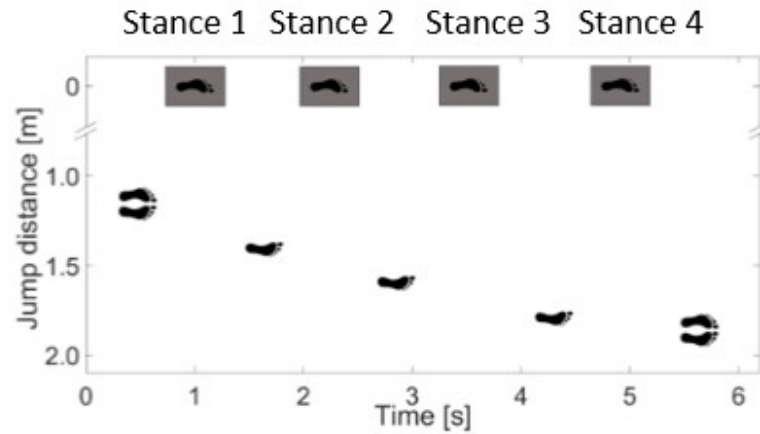


Figure 21: Measurement set-up of the experiment of D'Andrea (2019), that investigated the Ground Reaction Force of alternate lateral jumps.

	Stance 1	Stance 2	Stance 3	Stance 4
	Root Mean Square Error [N]			
M-L GRF	55.93	65.66	65.22	57.35
vGRF	43.42	52.17	76.21	71.59
	Relative Root Mean Square Error [%]			
M-L GRF	11.98	12.75	12.10	10.59
vGRF	4.45	5.23	7.51	7.10

Table 4: Root Mean Square Error (RMSE) and Relative RMSE of the medio-lateral and vertical GRF estimated from eight segmental accelerations, for alternate lateral bounding. (D'Andrea, 2019).

Finally, the running and jumping movements from the previous studies are investigated in a broad study by Verheul et al. (2019). The resultant GRF was estimated for a variety of dynamic and high-intensity tasks that are regularly performed during running-based sports: acceleration from stand-still to maximal sprinting speed, deceleration from maximal sprinting to immediate standstill, 90° cutting, which is a sharp change of direction, and constant speed running at a low (2 m/s - 3 m/s, or 7.2 km/h - 10.8 km/h), moderate (4 m/s - 5 m/s, or 14.4 km/h - 18 km/h) and high speed (> 6 m/s or 21.6 km/h). Fifteen team-sport athletes participated in the study, and full-body kinematic data were collected using a seventy-six retro-reflective marker set attached to anatomical landmarks of the body. The tri-axial GRF profile was estimated for each task as the sum of the product of fifteen segmental masses and accelerations, or a reduced number of segments, for each of the three directions.

	RMSE [N/kg]	rRMSE [%]
Acceleration	2.82 ± 0.7	8.4 ± 1.4
Deceleration	5.77 ± 1.8	6.1 ± 8.8
90° cuts	2.67 ± 0.7	3.3 ± 4.1
Constant speed running: slow speed (2-3 m/s)	1.62 ± 0.4	1.8 ± 2.0
Constant speed running: moderate speed (4-5 m/s)	2.48 ± 0.6	3.1 ± 5.7
Constant speed running: high speed (> 6 m/s)	4.35 ± 1.3	6.4 ± 7.6

Table 5: Root Mean Square Error (RMSE) and Relative RMSE (rRMSE) of the resultant GRF (tri-axial GRF) estimated from fifteen segmental accelerations, for several dynamic tasks. Values are means $\pm$ one standard deviation. (Verheul, 2019).

As shown in table 5, the accuracy of the estimated GRF profile varied across tasks. For lower intensity movements, the accuracy was high. As an example, the RMSE was low for running at low speed. The accuracy was moderate for accelerations, 90° cuts and running at moderate speed. For decelerations and high-speed running, the RMSE was very high. They concluded that a direct mechanical approach cannot provide valid estimates of the GRF from segmental accelerations across dynamic and high-intensity running tasks.

The accuracy of Verheul’s model with fifteen segments is similar or higher than the one of Scheltinga’s model with three segments for the constant speed running tasks. Although it contains fewer segments, the performance is not much worse. The model of Scheltinga achieves an rRMSE of 6.6 ± 1.2 %, while Verheul obtains 3.1 ± 5.7 % for moderate speed running and 6.4 ± 7.6 % for high speed running. Furthermore, both models show the trend that the higher the movement speed, the worse the model performance. A possible explanation for the less deteriorated performance than expected is that the weight factors and cut-off frequencies were obtained via a model optimisation. In addition, Verheul estimated the resultant GRF, while Scheltinga only estimated the vertical component. Nevertheless, Verheul concludes that acceleration signals and derived training load measures from one or several segments are unlikely to accurately represent biomechanical loads. All other studies considered the use of wearable sensors to evaluate athlete kinetics in the field as a ‘promising’ method. Therefore, the question whether the estimated value of the GRF is sufficiently accurate to determine the biomechanical load remains unclear across studies, and it is difficult to make a general conclusion.

3.2.1.1.2 Mass-spring-damper model

Another approach for modelling human movement is the mass-spring-damper (MSD) model, because the GRF during running is absorbed and returned by the body in a spring-like manner. The human body is modelled as a composition of one or more masses, that are connected to each other via springs and dampers. The masses represent the inertial properties of the

corresponding segments of the body, where the segments include hard and soft tissues, like bones, tendons, muscles and ligaments. The connecting springs and dampers are the mechanical properties of the hard and soft tissues. The springs represent the stiffness properties, while the dampers represent the damping properties [10]. A general benefit of the mass-spring-damper model is that the estimated model parameters could provide additional information for injury and performance evaluation. A disadvantage is that due to the elastic nature of the model, it is only suitable to study the response of the human body during hopping and running.

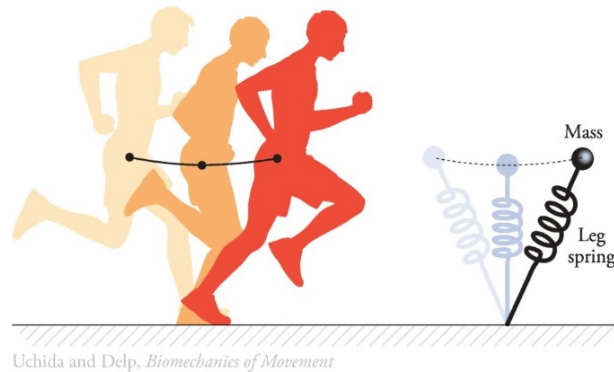


Figure 22: Schematic representation of the mass-spring-damper model during running. It consists of a mass, a spring and a damper to represent the mass and mechanical properties of the human body.

Several mass-spring-damper models have been developed over the years. They are divided into two categories, more specifically the active and passive models. The majority of the mass-spring-damper models belong to the category of passive models. In this case, the stiffness and damping properties are assumed to remain constant. Because they do not include the effects of active elements, such as muscles, the response of the model is the same regardless of the environment or the type of footwear. Active models, in contrary, do adapt the mechanical properties of the parameters to external loads. However, little research has been done on these models and therefore, they will not be discussed further.

Based on the number of masses, the passive models are further categorised as either single-body models, also called one-body models, or multi-body models.

3.2.1.1.2.1 One-body model

The single-body model was originally proposed by Blickhan. It is built by attaching one mass to a weightless spring and an optional damper. There are two types of single-body mass-spring-damper models. The first type is a one-dimensional model, that represents hopping when there is no forward motion. The second type is a two-dimensional model, used to study forward running in one plane. Both models are shown in figure 23.

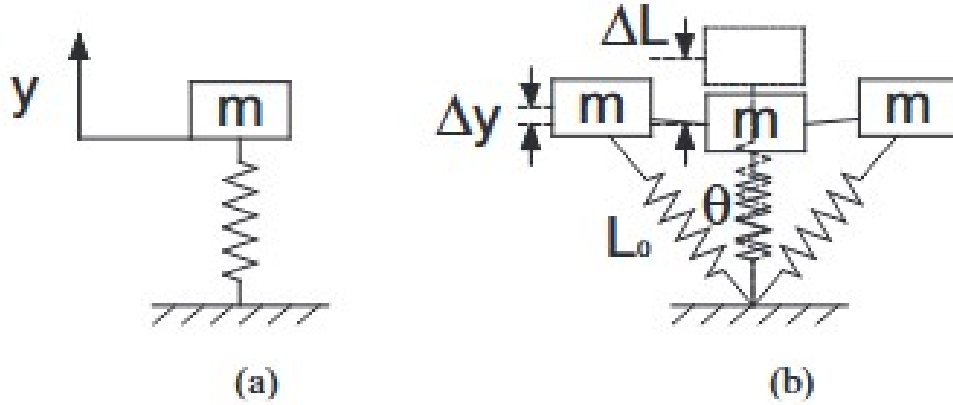


Figure 23: Passive single-body model as a schematic representation of (a) 1D motion and (b) 2D motion.

For the one-dimensional model, the equation of motion is given by:

$$m\ddot{y} + k_{ver}y = mg \quad (16)$$

where: k_{ver} : vertical stiffness, calculated as $k_{leg} = \frac{F_{max}}{\Delta y_{max}}$
 F_{max} : maximum vertical force
 Δy_{max} : maximum spring deformation

For the two-dimensional model, the equation of motion is written as follows:

$$m\ddot{y} + k_{leg}y = mg \quad (17)$$

where: k_{leg} : leg stiffness, calculated as $k_{leg} = \frac{F_{max}}{\Delta L}$
 F_{max} : maximum vertical force
 ΔL : vertical displacement of the mass, calculated as $\Delta L = \Delta y + L_0(1 - \cos \theta)$
 L_0 : standing leg length
 θ : angle, calculated as $\sin^{-1}\left(\frac{\dot{x}t_c}{2L_0}\right)$

Because of their simplicity, single-body models are the most widely used mass-spring-damper models. They have multiple applications, but the most important purpose is the estimation of the Ground Reaction Force during hopping and running. Additionally, single-body models are used to study the energy cost of running, the effects of running and hopping speed on the dynamics of the human body, and the determination of the stiffness of the lower-extremities.

However, the simplicity of the single-body models raises questions about their accuracy in predicting the characteristics of running. By comparing the model's predictions to the ex-

perimental data, it is concluded that one-body models predict the vertical active peak GRF reasonably well in normal conditions. Unfortunately, the initial high-frequency impact peak typically observed in the GRF waveform during running, which is speculated to be linked with negative tissue adaptations and risk of injury, cannot be estimated with this oversimplified model.

3.2.1.1.2.2 Two-body model

Because the human body is a complex system with many masses, a multi-body mass-spring-damper model has been proposed to achieve a more detailed description of human locomotion. In this case, several masses are connected to each other and to the ground via linear springs and dampers. The simplest example of a multi-body model is a two-body model. It is made of two masses, which represent the upper and lower body.

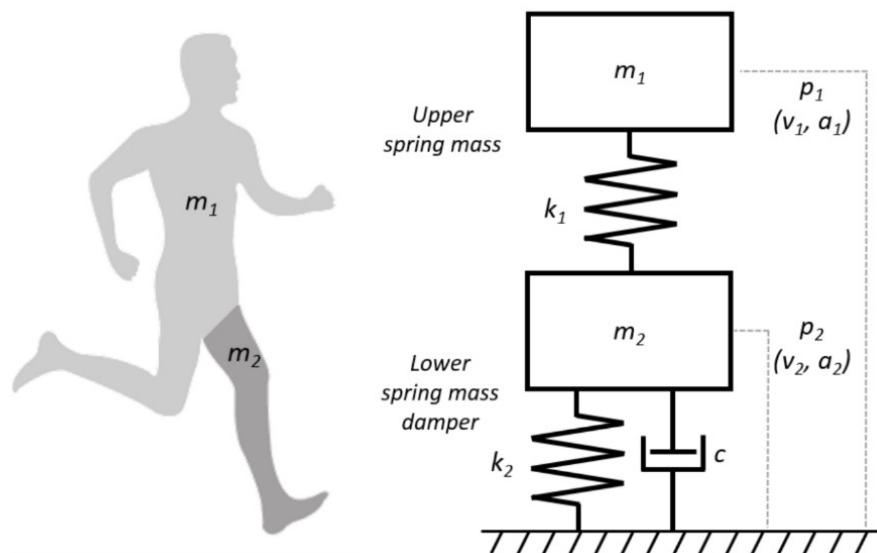


Figure 24: An illustration of the human body represented as a two-body MSD-model. The MSD-model consists of a lower mass-spring-damper element (m_2 , k_2 , c), that represents the support leg of the human body, and an upper mass-spring element (m_1 , k_1), that represents the rest of the human body.

Derrick et al. (2000) were the first to create a two-body MSD model, for the application of running with different stride lengths. However, it remained a mathematical model that was not used in practical experiments. A study of Nedergaard et al. (2018) did put this two-body mass-spring-damper model into practice, as shown on figure 24. Based on the hypothesis that the acceleration of the upper mass in the MSD-model primarily represents the acceleration of the trunk segment, they tried to estimate from one IMU on the trunk the required MSD-model parameters in order to predict the vertical GRF during running. For this study, twenty male athletes ran at a constant speed between 2 m/s (7.2 km/h) and 5 m/s (18 km/h). They reported that despite the good match between the measured trunk accelerometry

and the upper mass acceleration of the MSD-model (median RMSE: 0.16 g - 0.22 g), the vertical GRF estimation accuracy was poor (median RMSE between 6.68 and 12.77 N/kg). These results are illustrated in figure 25. The errors increased with the running speed.

A possible explanation for the low accuracy is provided by the analysis of the segments in the link-segment model, as explained in section 3.2.1.1.1) ValiditySegmentModel. The large contribution of the trunk to the GRF in a link-segment model could be primarily due to its large mass, rather than the characteristics of its acceleration. Consequently, the value of only using trunk accelerometry in a MSD-model to predict accurate GRF waveforms is probably limited.

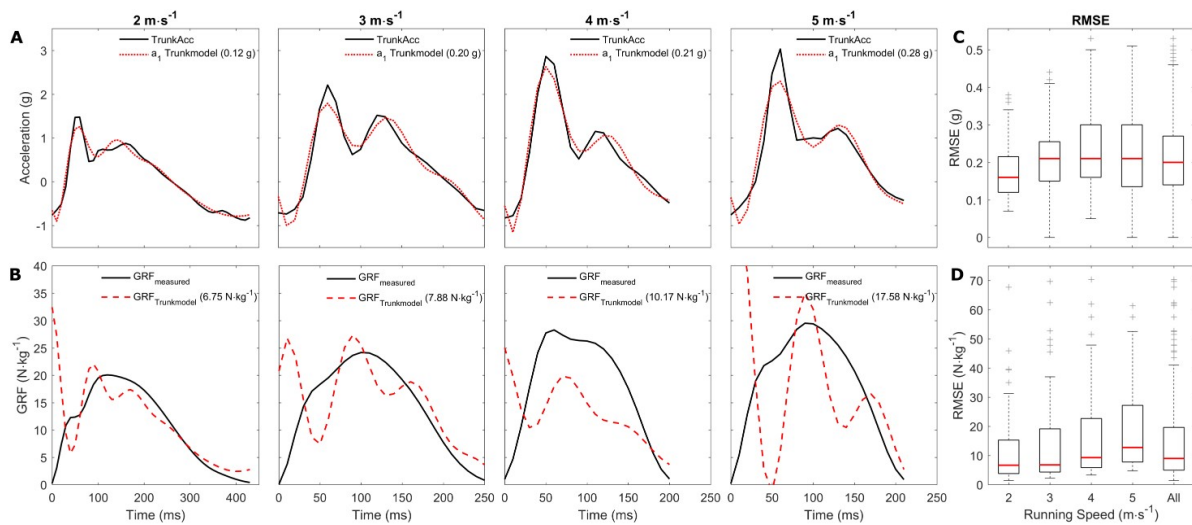


Figure 25: Representative examples of the trunk accelerometry fitting, GRF prediction, and the median RMSE across running speeds, of a single stride from multiple subjects. Figure (A) displays the fitting of the upper mass acceleration to the trunk accelerometry signal across running speeds, and figure (B) displays the measured and predicted GRF for the same trials. The RMSE for the trunk accelerometry fitting and GRF predictions are written in brackets for the individual examples. Figures (C) and (D) display the average RMSE median, and 25th and 75th interquartile range for the trunk accelerometry fitting and GRF prediction, respectively within and across the individual running speeds.

In collaboration with Nedergaard, a study of Verheul et al. (2021) aimed to examine the feasibility of a two mass-spring-damper model, based on eight model parameters, to reproduce the resultant Ground Reaction Force for high-intensity running tasks. Additionally, the study aimed to verify whether the required model parameters were physically meaningful. Fifteen athletes participated, and they performed rapid accelerations and decelerations, constant speed running and maximal sprints. The results are displayed in figure 26 and table 6. The model parameters and equations were solved using a purpose-written optimisation script, starting from the accelerations of the two masses as input. The estimated GRF was then compared to the measured GRF of a force plate. They concluded that the GRF profiles could be reproduced with low to very low errors across tasks, but more subtle loading character-

istics (for example impact peak and loading rate) were modelled less accurately. Moreover, the required model parameters varied strongly between trials and they had minimal physical meaning. These results show that although a two mass-spring-damper model can be used to reproduce overall GRFs for high-intensity running tasks, the application for predicting GRFs in the field and consequently understanding the biomechanical loads of training in greater detail is likely limited.

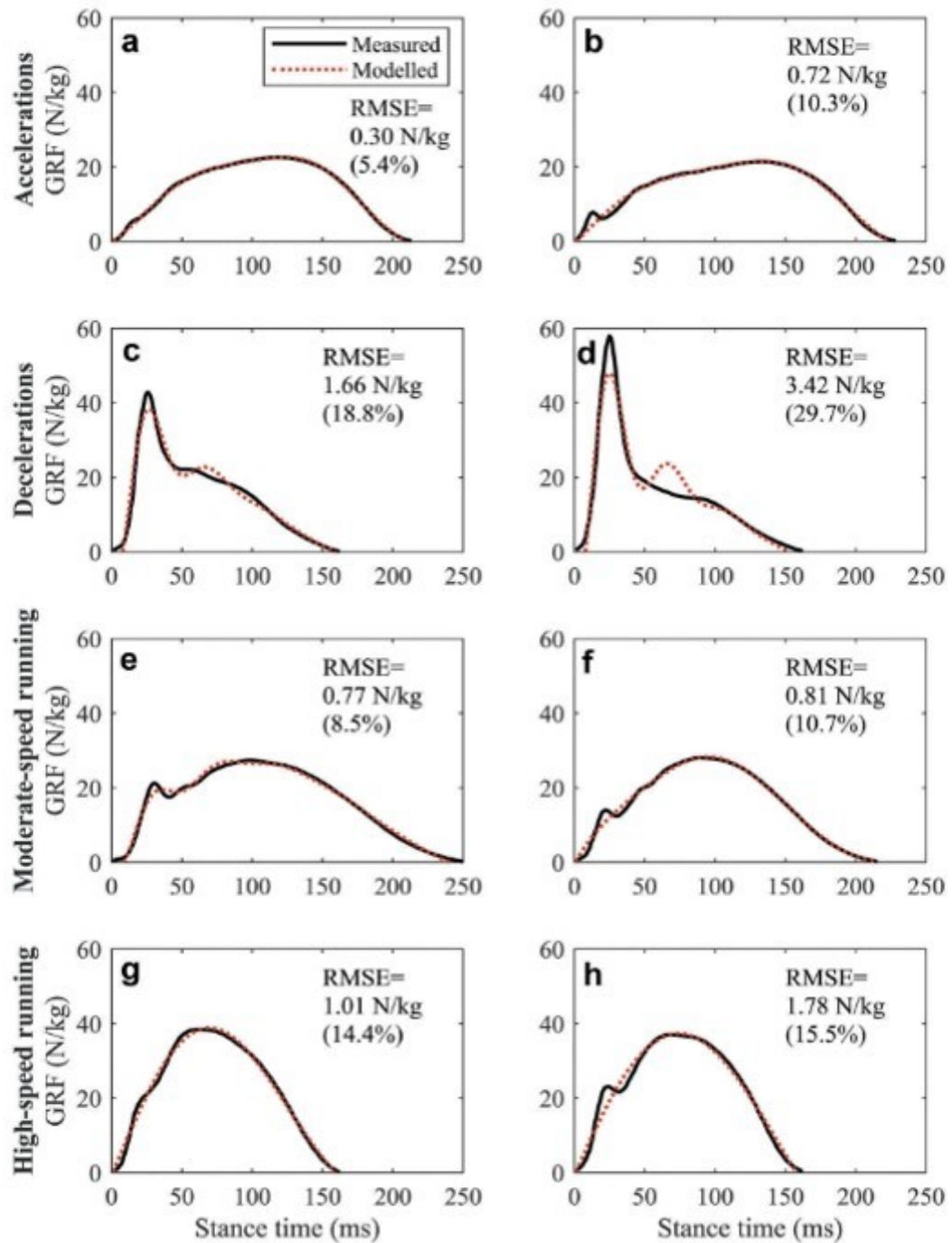


Figure 26: Typical examples of measured (black continuous line) and modelled (red dotted line) Ground Reaction Force (GRF) profiles including the root mean square error (RMSE) between both curves. (Verheul, 2021)

	RMSE [N/kg]	rRMSE [%]
Acceleration	0.69 ± 0.47	9.9 ± 6.4
Deceleration	2.48 ± 1.17	33.9 ± 28.3
Constant speed running: slow speed (2-3 m/s)	0.48 ± 0.22	7.6 ± 5.8
Constant speed running: moderate speed (4-5 m/s)	0.78 ± 0.25	9.4 ± 3.9
Constant speed running: high speed (> 6 m/s)	1.21 ± 0.56	13.6 ± 7.1

Table 6: Mean $\pm$ standard deviations for root mean square errors (RMSE) and relative root mean square errors (rRMSE) of the modelled GRF profiles for different tasks. (Verheul, 2021).

3.2.1.1.2.3 Multi-body model

For a multi-body system, the equations of motion are written in a general form as follows

$$M\ddot{X} + C\dot{X} + KX = W \quad (18)$$

where: M: the mass matrix

C: the matrix of damping coefficients

K: the stiffness matrix

X: displacement vector

$\dot{X}$: velocity vector

$\ddot{X}$: acceleration vectors

W: column vector that contains the weight of the masses included in the model. In some cases, t

A three-body model was developed by Kim et alMass-spring-damperModelOverview Two masses represent the foot and shank, and the other mass represents the rest of the body. The masses are connected together via linear springs and dampers. Additionally, the shoe-ground interaction is modelled using a spring and a damper.

The four-body model of Liu and Nigg (LN model) is probably the most used multi-body MSD model of the human body during hopping and running. The model contains four masses, five springs, and four dampers. Two important novelties distinguish this model. First, the upper- and lower-body masses have been divided into rigid and wobbling masses. The wobbling masses represent all non-rigid parts of the body, for example muscles, skin and blood vessels. Secondly, the shoe-ground-interaction is modelled with a non-linear functionMass-spring-damperModelOverview

It can be concluded that the complexity of the MSD-model increases rapidly as the number of masses increases. It is therefore difficult to implement these mathematical multi-body MSD models in practice with wearable sensors.

3.2.1.2 Forward dynamic approach

The majority of kinetic analyses of human movement are performed based on inverse dynamic models. As explained earlier, this approach takes the kinematic measures to estimate for example the Ground Reaction Force. It is the inverse of what really happens. Human movement begins with an electrical signal from a motor nerve to the muscles, resulting in varying levels of recruitment of agonist and antagonist muscles. The net effect of all these muscle forces acting at each joint, is the generation of a time-varying moment on the joint, which in turn accelerates or decelerates the adjacent body segments. The wearable sensors finally record the accelerations. The constraints of forward solution models are considerable compared to inverse solutions. For a forward solution, all body segments that are capable of moving must be modelled before start. If any segment has the wrong mass or a joint has a missing or unrealistic constraint, the model will start to generate displacement errors, and these errors will accumulate over time.

a computational model tries to refine the calculation of the forces with the measured kinematic data. However,

Despite the complexity of this approach, a study by Haraguchi (2024) developed a forward dynamic model that predicts the GRF in walking based on six IMUs, mounted on the trunk, lower arm, pelvis, thigh, shank and foot. Therefore, their algorithm used forward dynamics calculation to make a estimation of the kinematics. A hybrid cost function optimised the estimated forces by minimising the error between the IMU recording data and the estimated kinematics motion of the forward dynamic model. The accuracy of their model was tested in an experiment with eighteen participants, and three conditions for walking speed: normal, fast and slow. The normal speed was set to the preferred walking speed of each participant, and the fast and the slow speeds were set to approximately +20% and -20%, of the normal walking speed. In the three walking speeds, a strong correlation was observed for the anterior-posterior GRF ($\rho = 0.709-0.796$, RMSE = 6.34-7.42 %BW, rRMSE = 14.4-16.5 %) and the vertical GRF ($\rho = 0.866-0.898$, RMSE = 20.5-24.1 %BW, rRMSE = 17.6-18.3 %). Only a moderate correlation was observed for the lateral GRF ($\rho = 0.342-0.504$, RMSE = 3.03-3.99 %BW, rRMSE = 26.8-29.4 %). However, the accuracy achieved is generally much lower than the one obtained with an inverse dynamic approach. In addition, the iterative optimisation takes hours and makes it difficult to use this method in practice.

3.2.2 Deep learning

Although the predictions of the GRF-estimation models with the dynamic approach are promising, their application is either restricted to a limited range of movements or not yet feasible in the field. Furthermore, to perform an inverse dynamics approach for modelling tissue specific loads, all directions of the GRF are needed as input, while the Ground Reaction Force is mostly only modelled in the vertical direction. As an example, the anterior-posterior GRF plays an important role in the push-off phase [17].

To overcome the limitations of models based on the dynamic approach, a data-driven method

to predict the GRF from wearable sensors can be exploited. In this case, a combination of multi-layer perceptrons (MLPs), also called deep learning or a neural network, will make a prediction of the Ground Reaction Force based on the data of the IMUs. The rapid development of artificial intelligence has made this approach more popular in recent years. The computing power of artificial intelligence has become so strong, that one sensor is even sufficient to estimate the GRF accurately. The most important disadvantage of artificial intelligence is that the estimation is primarily based on data inputs, rather than biomechanical reasoning. Another difficulty is that the neural network requires data from many observations before it can make a prediction with sufficient accuracy, and without the risk of overfitting.

Data-driven models can be either generic or personalised. It is evident that personalised machine learning models will result in better model performance than generic models. Nevertheless, creating personal models is time consuming and not always possible. Creating generic models is more challenging, as they should be able to handle between-subject variability in the data. The advantage is that these models can be used “out of the box”: once a generic model is created, no specific model training is needed to estimate the GRF for new subjects.

Most studies focus on developing generic models, and this was also the case in a study by Scheltinga et al. (2023). They placed three IMUs on the pelvis and tibias, and these sensors provided the input for an artificial neural network (ANN) to estimate the three components of the GRF. They built an ensemble deep learning model, where different subjects were used for training and validation for each ensemble-member. Twelve participants with a heel-strike gait pattern participated. They ran nine trials at three different velocities (10 km/h, 12 km/h and 14 km/h) and three stride frequencies (preferred stride frequency and preferred stride frequency $\pm 10\%$). The predicted GRF was compared to the measured GRF of an instrumented treadmill. The results are listed in table 7.

Constant speed running (low: 10 km/h, moderate: 12 km/h, fast: 14 km/h)	
	Pearson's correlation coefficient (r) [-]
A-P GRF	0.91 ± 0.09
M-L GRF	0.57 ± 0.21
vGRF	0.97 ± 0.02
	Root Mean Square Error (RMSE) [BW]
A-P GRF	0.08 ± 0.03
M-L GRF	0.05 ± 0.01
vGRF	0.19 ± 0.04
	Relative Root Mean Square Error (rRMSE) [%]
A-P GRF	7.8 ± 3.1
M-L GRF	10.9 ± 2.5
vGRF	7.3 ± 1.8
	Peak error [%]
A-P GRF	-
M-L GRF	-
vGRF	4.21 ± 1.36

Table 7: Model performance for the direct model in the different directions. (Scheltinga, 2023).

The model predicts the vertical GRF (rRMSE: 7.3 ± 1.8 %) most accurately, compared to the anterior-posterior GRF (rRMSE: 7.8 ± 3.1 %) and medio-lateral GRF (rRMSE: 10.9 ± 2.5 %). A possible explanation is that larger forces are predicted better by the model. As a possible improvement, it is suggested that, since there are multiple ways to ensemble a model, there are also other ways to combine the ensemble-members. In the current study, the average between all members was taken, but for example, the weights of different members can be altered based on the model performance during training.

A study of Pogson (2020) used a neural network method to predict the GRF magnitude from the trunk acceleration magnitude for a variety of different running tasks across fifteen individuals: straight overground accelerated, decelerated and constant-speed running trials, ranging between 2 m/s (7.2 km/h) and 8 m/s (28.8 km/h) with 1 m/s increments. The magnitude of the GRF was predicted with an average r^2 of around 0.9 for the time series of each task. The method therefore offers a promising approach to estimate GRF in the field (see figure 27). The study has not considered the importance of sensor location on the body, but the results may be improved by placing the accelerometer elsewhere on the body. Since the impact peak is likely related to lower limb accelerations during landing [25], the addition of signals from the lower limbs might improve the prediction.

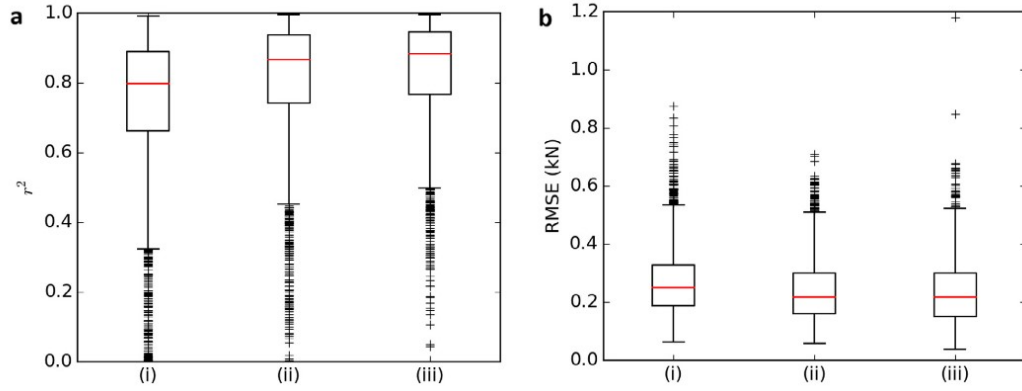


Figure 27: Test predictions from 10 repetitions of training. (a) r^2 coefficient between prediction and data; (b) corresponding RMSE. Results obtained using: (i) arbitrary options, without appending signal length; (ii) arbitrary options, with signal length appended; and (iii) optimised options, with signal length appended. Central horizontal lines show the median and boxes show the interquartile range. Whiskers show up to 1.5 times the interquartile range; any values outside this are plotted as individual points, which show the r^2 or RMSE value for individual outlying impact predictions.

When comparing the studies of Pogson ($r^2 \approx 0.9$) and Scheltinga ($r^2 = \rho^2 = 0.94$), Scheltinga's model is slightly better. However, the former only had data from one sensor, which was attached to the trunk, while the latter used 3 sensors placed on the pelvis and tibia. To counteract the lower variation in the data, the construction of Pogson's deep learning network was therefore much more complex. However, it shows that a well-considered choice of the architecture of the deep learning network can make a difference in achieving good accuracy.

Furthermore, Pogson's deep learning network and Nedergaard's mass-spring-damper model [9] used both a single IMU on the trunk. If both methods are compared for the estimation of the vertical GRF, the data-driven approach clearly outperforms the dynamic approach.

3.2.3 Hybrid Method

Because dynamics-based and deep learning techniques have both been used to estimate the Ground Reaction Force, a hybrid method can be applied. It integrates the traditional physical laws with the advanced deep learning to make a prediction of the Ground Reaction Force. As such, the explainable biomechanical foundations of the dynamic approach and the computational accuracy of the data-driven method are combined and can lead to improved model performance.

A study of Scheltinga et al. (2023) used data from three IMUs, placed on the pelvis and lower legs, as input for an artificial neural network (ANN) to estimate the GRF. The estimated vertical GRF from a physical model was used as an additional input3IMU The data was collected from 12 heel-strike runners, who ran at three different velocities (10 km/h, 12 km/h and 14 km/h) and three stride frequencies (preferred and preferred $\pm 10\%$) on an instrumented treadmill.

The results are summarised in table 8.

Constant speed running (low: 10 km/h, moderate: 12 km/h, fast: 14 km/h)	
	Pearson's correlation coefficient (r) [-]
A-P GRF	0.91 ± 0.10
M-L GRF	0.58 ± 0.22
vGRF	0.97 ± 0.01
	Root Mean Square Error (RMSE) [BW]
A-P GRF	0.07 ± 0.03
M-L GRF	0.05 ± 0.01
vGRF	0.18 ± 0.04
	Relative Root Mean Square Error (rRMSE) [%]
A-P GRF	7.8 ± 3.2
M-L GRF	10.8 ± 2.5
vGRF	6.8 ± 1.7
	Peak error [%]
A-P GRF	-
M-L GRF	-
vGRF	4.09 ± 1.35

Table 8: Model performance for the hybrid model in the different directions. (Scheltinga, 2023).

Since this study investigated a link-segment model, a deep learning model and hybrid model, it is interesting to compare the performance of the three different approaches. The performance of the hybrid model (rRMSE: 6.8 ± 1.7 %) outperformed the deep learning model in the vertical direction (rRMSE: 7.3 ± 1.8 %), and was comparable with the link-segment model (rRMSE: 6.6 ± 1.2 %). Additionally, the model performance was related to the magnitude of the estimated forces, meaning that lower forces, for example the GRF in the medio-lateral direction, had higher errors. The findings suggest that, using a hybrid model, it is possible to estimate the GRF in the anterior-posterior and vertical direction using data from 3 IMUs on the pelvis and one on each tibia, but it is not possible to precisely estimate the medio-lateral forces (Pearson's correlation coefficient: 0.58). A hybrid model is therefore more interesting than a link-segment model if the aim is to estimate the GRF also in the anterior-posterior direction, instead of only in the vertical direction.

3.2.4 Conclusion

An overview of the results of all mentioned studies is shown in tables 9 and 10. Besides the efforts to make these models as accurate as possible, numerous questions remain unanswered. For all approaches, there is no consensus about the amount and position of the sensors on

the body, although sensors located on the trunk seem to provide more relevant information for modelling the first peak, and those on the lower limbs appear to provide more relevant information on modelling the second peak and the loading rate of the GRF profile in runningSensitivity

A final remark is that a simple model is almost always preferable to a more complicated version, because the latter provides similar utility but it is more difficult to understandBiomechanicsOfMovement After all, the explainability of the model is extremely important in a clinical setting, where every decision must have a theoretical basis.

Reference	Number of sensors	Position	Movement	Estimated variable	Model	Accuracy
(Karatsidis, 2017) [3]	17	whole body	walking	A-P GRF M-L GRF vGRF	link-segment model	rRMSE: A-P GRF = 10.2 ± 2.9 %, M-L GRF = 13.8 ± 3.1 %, vGRF = 6.2 ± 3.1 %
(D'Andrea, 2019) [31]	1	tibia	alternate lateral bounding	link-segment model	M-L GRF, vGRF	rRMSE: M-L GRF = 6.1 %, vGRF = 11.9 %
(Verheul, 2019) [8]	15	whole body	running	link-segment model	Tri-axial GRF	rRMSE: resGRF = 4.8 ± 8.3
(Scheltinga, 2022) [29]	3	pelvis, thigh	running	link-segment model	vGRF	rRMSE: vGRF = 6.6 ± 1.2
(Nedergaard, 2018) [9]	2	trunk, tibia	running	Tri-axial GRF	two mass-spring-damper model	$r^2 = 0.9$
(Verheul, 2021) [6]	2	trunk, tibia	running	Tri-axial GRF	two mass-spring-damper model	rRMSE: Tri-axial GRF = 17 ± 19.1
(Haraguchi, 2024) [32]	6	trunk, pelvis, right lower arm, right thigh, right shank, right foot	walking	A-P GRF, M-L GRF, vGRF	forward dynamic model	rRMSE: A-P GRF = 16.4 %, M-L GRF = 29.1 %, vGRF = 18.2 %

Table 9: Overview of dynamic model accuracies. In case of multiple speeds and movements the average accuracy is reported.

Reference	Number of sensors	Position	Movement	Estimated variable	Model	Accuracy
(Scheltinga, 2023) [17]	3	trunk, tibia	running	A-P GRF, M-L GRF, vGRF	deep learning	rRMSE: A-P GRF = 7.8 ± 3.1 , M-L GRF = 10.9 ± 2.5 , vGRF = 7.3 ± 1.8
(Scheltinga, 2023) [17]	3	trunk, tibia	running	A-P GRF, M-L GRF, vGRF	hybrid learning: link-segment model + deep learning (3520 strides)	rRMSE: A-P GRF = 7.8 ± 3.2 , M-L GRF = 10.8 ± 2.5 , vGRF = 6.6 ± 1.2 $r^2 = 0.94$
(Pogson, 2020) [33]	1	trunk	running	Tri-axial GRF	deep learning	$r^2 = 0.9$

Table 10: Overview of data-driven model accuracies.

4 Methods

In this chapter, various biomechanical models that can estimate the Ground Reaction Force and moment are designed and implemented. The IMUs in the sensorised running pants provide the necessary data inputs. The models are then validated by comparing the estimated GRF with the measured GRF of an instrumented treadmill.

4.1 Experimental Design and Data Processing

Eight healthy athletes participated in the data collection for this study, of which seven recordings were ultimately usable. One recording was lost because the treadmill had failed. The participants of the processed recordings are all men, 27 ± 5 years old, 1.77 ± 0.09 m tall, and weigh 70 ± 6 kg. All participants signed a written informed consent beforehand.

The participants were recorded while running on a dual-belt treadmill (X-Mill Dual Belt Instrumented Treadmill, ForceLink B.V, Culemborg, The Netherlands). They ran one trial at three different speeds: 8 km/h, 10 km/h, 12 km/h for 60 seconds in increasing order. The sampling frequency of the treadmill is 960 Hz. The recorded GRF data will be used as a reference for the GRF estimation. In order to make more accurate comparisons between the different GRF magnitudes of the participants, the GRF data is normalised for body weight, by dividing the GRF by the mass of the participant multiplied with the gravitational acceleration ($g = 9.81\text{m/s}^2$). Subsequently, the GRF data is filtered using a second-order, low-pass Butterworth filter with a cut-off frequency of 30 Hz and zero-phase shift. This cut-off frequency was selected as it was the most commonly cited value in the literature [17].

Because most of the existing research uses optical motion capture to apply GRF prediction methods to kinematic data, sixteen marker trajectories were measured synchronously with the treadmill data using the software Vicon Nexus (Vicon Motion Systems Ltd, UK). The reflective markers were attached to anatomical landmarks of the lower limbs, according to the Lower Body Plug-in Gait Model. The marker trajectories were sampled by the cameras at a frequency of 120 Hz. For markers that were not detected for a short period of time, the trajectory was approximated by a linear curve. The kinematic data was used to build a seven-segment (pelvis, thighs, shanks and feet) six-degree-of-freedom model of the lower limbs in Vicon Nexus [34]. The trajectories of the markers were derived once to obtain the markers' speeds, and twice to obtain the accelerations. Subsequently, these variables were filtered with a zero-phase third order low-pass Butterworth filter with a cut-off frequency of 6.5 Hz, and upsampled to the same length as the measured GRF. It was decided to upsample the signals to prevent information loss. A disadvantage is that the values are interpolated and therefore, there is a risk of creating incorrect points.

At the same time, the participant was wearing the sensorised running pants with five integrated IMUs, as previously described in section 1. On top of the pelvis sensor, an additional belt was placed to prevent movement artefacts due to sagging or flapping running pants. The data were obtained at a sampling frequency of 100 Hz. Before the start of the measurement,

the IMUs were calibrated according to the manufacturer's instructions. The IMUs measure the linear accelerations in multiples of the gravitational acceleration (g). In order to work in SI units, the values are first expressed in m/s^2 by multiplying with the gravitational acceleration. An important note is that each IMU measures the accelerations according to its own orientation, as schematically depicted in figure 28. The IMU's orientation is expressed in quaternions, and these are obtained by fusing the signals of the accelerometer, gyroscope and magnetometer. Based on the quaternions, a rotation matrix corresponding to the orientation of the IMU is constructed. By multiplying the accelerations and the angular velocities of the IMUs with the inverse of the rotation matrix, the accelerations and angular velocities are expressed in the same frame as the treadmill and the markers. Finally, gravity is subtracted from the rotated vertical accelerations in order to obtain the sensor free acceleration.

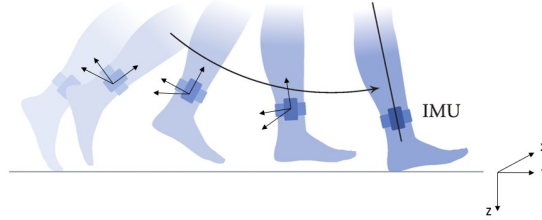


Figure 28: Schematic representation of the local frame of the IMU and the global frame of the lab.

The signals from the treadmill and the markers are recorded together by the Vicon system, but the IMUs in the sensorised running pants record independently. Therefore, the signals from the two systems must be synchronised. Each trial started with three jumps, serving as a rough reference point to synchronise the IMU data and the Vicon data. The data were trimmed so that they started from the reference point. The end point was selected 60 seconds after the reference point. Next, the cross-correlation between the vertical acceleration of the IMU on the pelvis and the measured GRF was calculated to refine the rough starting point provided by the jumps. A range of lag values between -0.5 seconds and 0.5 seconds in steps of 0.01 seconds, the sampling frequency, was added to the reference point of the jumps and subtracted from the end. The lag value corresponding to the maximum in the cross-correlation was used to optimise the synchronisation of the treadmill, markers and IMU signals. Finally, the recording was filtered with a third-order zero-phase shift low-pass Butterworth filter with cut-off frequency of 6.5 Hz, and upsampled to the same length as the corresponding measured GRF.

The kinematic and kinetic data of the IMUs, markers and treadmill were exported to Python 3.11.4 for further processing and analysis. The entire code of this research is available in appendix B.

To increase the readability of the calculations in this thesis, vectors and matrices are written in bold lowercase letters and bold capital letters respectively.

4.2 Anthropometry

The body dimensions and body mass of the subject are crucial in any model to calculate the Ground Reaction Force. The first, and most logical way, is to measure the requested dimensions on the subject. Typically, anatomical landmarks are used, such as the left anterior superior iliac spine or the flexion-extension axis of the knee. However, these measurements require good anatomical knowledge and are also very time-consuming. Consequently, this was not a feasible option.

Since the required dimensions were not available from all participants, a second method was developed based on body length. The body is easy to measure and therefore, the developed models can be used quickly. The method starts from the standard body dimensions for men and women, as reported in the study by De Leva (1996) [7]. The dimensions were adapted to the placement of the sensors in the sensorised running pants, as shown in table 11. For example, it has been decided that the pelvis sensor represents the movement of the upper body, because it is most close to the body's CoM. The reference lengths of De Leva are then personalised to the subject by rescaling them based on the ratio of the subject's body height to the standard body height. The tables do not mention the width (medio-lateral direction) or depth (anterior-posterior direction) of the body segments. However, these could be calculated based on the data from the male participants of this study.

Segment	Definition		Width [m]		Depth [m]		Longitudinal length [m]	
	Starting point	End point	Men	Women	Men	Women	Men	Women
Pelvis ($\approx$ upper body)	Vertex	Mid-hip	0.2739	-	0.1442	-	0.7352	0.7295
Thigh	Hip joint centre	Knee joint centre	0.1176	-	0.1216	-	0.4222	0.3685
Shank and foot	Knee joint centre	Sole of the foot	0.1064	-	0.1307	-	0.5836	0.4323
Total	Vertex	Sole of the foot	-	-	-	-	1.741	1.735

Table 11: Reference body dimensions for men and women, as reported in the study by Paolo de Leva.

The software Vicon Nexus can also estimate the body lengths, based on the position of the markers. When comparing the body dimensions, with the reported values of Vicon as a reference, the error for leg length is on average 13cm and for knee width 1.2cm. The leg length was

calculated from the rescaled values of the table as the sum of the thigh length and the shank length, and the knee length as the average of the width of the thigh and the shank.

4.3 Step detection

In order to evaluate the accuracy of the model per running step, a step detector was first created. Three possibilities for a step detection algorithm were investigated: an arbitrary threshold on the Ground Reaction Force, the values of the quaternions, and the acceleration of the pelvis.

4.3.1 Step detector 1: arbitrary threshold on the vertical Ground Reaction Force

Force platforms detect a foot contact if the vertical component of the Ground Reaction Force crosses a set threshold, typically a value between 10 N to 50 N [35]. In this application, stance phase is detected when the threshold exceeds 15 N longer than 0.05 s. The step detection algorithm is followed by an additional control algorithm, that checks if each initial contact is followed by toe-off within a time period not longer than the time between two initial contacts.

Since this method relies entirely on the value of the vertical GRF, incorrect estimation of the GRF will result in incorrect detection of the steps. Therefore, it was decided that it should only be used to divide the mGRF into steps. For this reason, step detectors 2 and 3 were developed.

4.3.2 Step detector 2: quaternions

The method of using the quaternions to detect a step was proposed by the manufacturer of the IMUs. When the scalar part of a quaternion measured by the IMU exceeds a certain threshold value, in this case 10^4 , a step is detected.

4.3.3 Step detector 3: velocity of the pelvis

The peak downward velocity of the pelvis (PDVP) is proposed and validated as a kinematic method to detect initial contact and toe-off in running regardless of the running strike pattern, according to studies of Benson and Reenalda [35] [36]. In this method, initial contact and toe-off are detected when the velocity of the pelvis reaches a minimum. The step detection algorithm is followed by an additional control algorithm, that checks if each initial contact is followed by toe-off within a time period not longer than the time between two initial contacts.

4.3.4 Jump detection

Similarly to the step detector for running, an algorithm was created to segment the jumps in the time recording. Again, a distinction is made between the measured Ground Reaction Force and the accelerations of the inertial measurement units, but the principle remains the same

and is similar to that of step detector 1. When the mGRF is lower than 15 N for a specified time period, for example 0.1s, the person is jumping. For the IMUs, this corresponds to the condition where the vertical acceleration of the pelvis is less than -9.81 m/s^2 . Once the start and end of the jump have been identified, 1 s is subtracted from the start and 1 s is added to the end. These values were chosen arbitrarily based on a visual analysis of the recording. Finally an addition control algorithm verified the jump's physical reliability, by assuming a maximum duration for the flight phase of 2 seconds.

4.4 Model Engineering

Based on the literature study on the state-of-the-art of GRF estimation models, it is chosen to develop a link-segment model and a mass-spring-damper model. Both models will have the data of the IMUs in the sensorised running pants as inputs.

4.4.1 Link-segment model

Figure 29 provides an overview of the steps involved in calculating the Ground Reaction Force using the link-segment model. The first two steps were discussed in section 4.1, while the subsequent sections cover the final two steps.

4.4.1.1 Ground Reaction Force

Newton's second law states that acceleration is proportional to the force. Consequently, the GRF can be estimated from the acceleration of the body segments, as shown in equation 19, for the medio-lateral, anterior-posterior and vertical direction. Since the IMUs measure continuously and do not detect when the leg is in the flight phase, the vertical accelerations were corrected by stating that $a_{i,v} = 0$ if $a_{i,v} \leq -g$ with $g = 9.81 \text{ m/s}^2$. After the calculation, the eGRF is normalised for body weight (BW) by dividing with $m \cdot G$.

$$eGRF = m \cdot G + \sum_{i=1}^N w_i \cdot m \cdot A_i \quad \text{with } A_i \geq 0 \quad (19)$$

where m : the body mass of the participant

$G = [0, 0, g]$: the gravitational acceleration in the medio-lateral (0), anterior-posterior (0) and vertical ($g = 9.81 \text{ m/s}^2$) direction

i : annotates the segment number and goes from 1 to N , with N the total number of segments. The number of segments is depend on the number of sensors or markers. In this case, five segments are defined: the pelvis, left shank, left thigh, right shank and right thigh.

w : the weighting factor that represents the fraction of body mass corresponding to the segment where the sensor or marker is attached to.

$A_i = [Ax_i, Ay_i, Az_i]$: the measured acceleration of each body segment, with x the medio-lateral direction, y the anterior-posterior direction, and z is the vertical direction

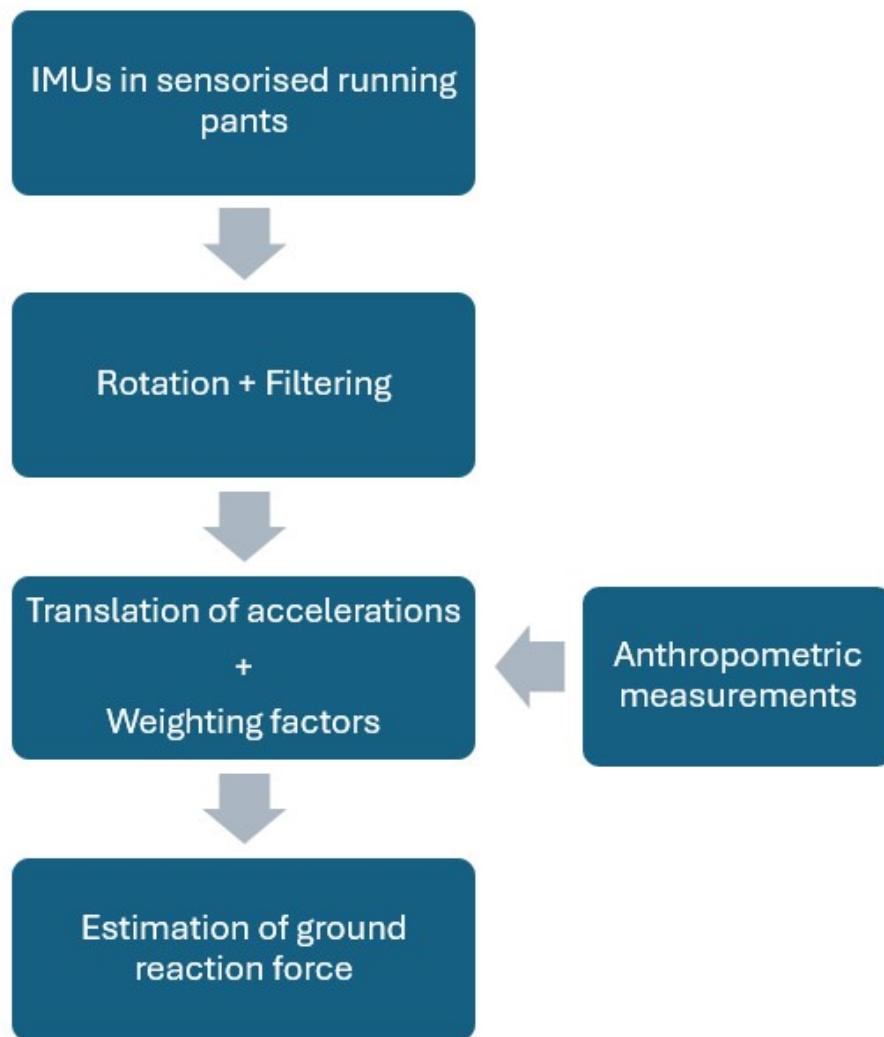


Figure 29: Block diagram of the implementation of the link-segment model.

4.4.1.2 Translated accelerations

It should be noted that the link-segment model assumes that the accelerations are measured in the CoM of the segment. As it is physically not possible to place the IMUs there, the accelerations are being translated according to equation 20.

$$\mathbf{a}_i = \mathbf{a}_{i,o} + \dot{\omega} \times \mathbf{r}_{oi} + \omega_i \times (\omega_i \times \mathbf{r}_{oi}) \quad (20)$$

where ω : the angular velocity of the body segment

r : the distance between the IMU and the segment's CoM

In this equation, ω is the angular velocity and r is the distance between the IMU and the CoM.

4.4.1.3 Weighting factors

The weighting factors define the mass of each body segment as a percentage of the total body mass. Consequently, the weighting factors should be positive and their sum should be equal to one, because as such, the body segments will sum to the total body mass. They are defined either from anthropometric measurements or obtained by machine learning.

4.4.1.3.1 Anthropometric measurements

In case of anthropometric measurements, general weighting factors can be found in the literature or a more personal approach can be followed. General weighting factors are often taken from the tables of de Leva [7] or Winter [13][20]. The weighting factors have been adjusted to the configuration of the sensorised running pants, so that the weighting factor of the shank includes the mass of the shank and foot, and the weighting factor of the pelvis includes the mass of the head, trunk, and arms. Table 12 contains the adjusted weighting factors that are used in the model.

	Pelvis	Thigh	Shank
De Leva men (MdL)	0.6028	0.1416	0.057
De Leva women (FdL)	0.5824	0.1478	0.061
Winter (W)	0.678	0.1	0.061

Table 12: Adjusted weighting factors, which are based on the reported values in the tables of De Leva [7] and Winter [13].

However, it should be noted that each body is different and these general values are not necessarily applicable to everyone. Therefore, an attempt was made to define more personal weight factors. The obtained formula is stated in equation 21. The method is based on the estimation of the volume of the thigh and the shank as a cylinder. To calculate the mass of the body segment, its volume must be multiplied by its density. The adjusted body dimensions from table 11 were used for the radius and height of the cylinder, and the density was taken from

Winter [13]. These density values are 1090 kg/m^3 for the shank and 1050 kg/m^3 for the thigh. The mass of the pelvis was calculated by subtracting the mass of both femurs and both lower legs from the total body mass. Finally, the estimated mass of the body segment was divided by the total body mass to obtain the weighting factor. A potential drawback of this method is that it shifts the previous problem to the density, which is again a generalisation.

$$w_i = \frac{m_i}{M} \quad (21)$$

$$= \frac{V_i \rho_i}{M} \quad (22)$$

$$= \frac{r_i^2 \pi h \rho_i}{M} \quad (23)$$

where V : the volume of body segment i

r_i : the radius of the body segment i , which corresponds to half its width

h_i : the longitudinal length of body segment i

ρ_i : the density of body segment i

M : the total body mass

4.4.1.3.2 Machine learning

A second approach to determine the weighting factors is by using machine learning. The machine learning methods will calculate the weighting factors for which the error between the estimated and the measured GRF is minimal, in order to find the best fitting weight factors. The error is typically expressed using the root mean squared error (equation 24). Three methods were tested: Linear Regression, Bayesian Ridge Regression, and a self-written Stochastic Gradient Descent algorithm.

$$J(w) = \frac{1}{P} \sum_{j=1}^P (eGRFv_j(w) - mGRFv_j)^2 \quad (24)$$

where P : the number of data points

$eGRF_j$: the GRF estimation at time point j , which depends on the weighting factors w

$mGRF_j$: the measured GRF at time point j

4.4.1.3.2.1 Linear Regression

Linear regression assumes that the predicted variable is a linear combination of other variables. With a small reformulation (equation 25), the link-segment model is a linear combination of the accelerations of each body segment with the weighting factors as coefficients. The model was implemented in such a way that only positive coefficients were allowed, in order to ensure that the weighting factors remained as realistic as possible.

$$\mathbf{mGRF} \cdot \mathbf{m} \cdot \mathbf{g} - \mathbf{m} \cdot \mathbf{G} \approx \sum_{i=1}^N W_i \cdot \mathbf{m} \cdot \mathbf{A}_i \quad (25)$$

4.4.1.3.2.2 Bayesian Ridge Regression

Bayesian Ridge Regression provides a probabilistic interpretation of the linear regression model. It can include regularisation parameters in the estimation procedure, in order to prevent over-fitting.

4.4.1.3.2.3 Stochastic Gradient Descent

Stochastic gradient descent is an optimisation algorithm that allows to find a local minimum of a loss function (see equation 24). Note that stochastic gradient descent finds a local minimum of the loss function, which may not be the global minimum. Therefore, it is important to make an accurate initial guess for the weight factors. The minimum of the loss function is found by updating the weighting factors in the direction of the negative gradient. The updates will be repeated until a minimum is reached. The learning rate α is added to control the size of the updates, and it needs to be selected carefully. If it is set too low, the algorithm may converge slowly; if it is set too high, it may 'jump' over the minimum and diverge. The formula to calculate the weighting factors is then given by:

$$\mathbf{w}_{(i+1)} = \mathbf{w}_{(i)} - \alpha \frac{\partial J(\mathbf{w})}{\partial \mathbf{w}} \quad (26)$$

where $\mathbf{w}_{(i)}$: the i -th iteration of the algorithm to optimise weight factors $\mathbf{w}$

α : the learning rate

$\frac{\partial J(\mathbf{w})}{\partial \mathbf{w}}$: the partial derivative of the loss function to each weighting factor

In formula 26, the derivative of the loss function with respect to each weighting factor is still unknown. Specifically for the link-segment model, this becomes:

$$J(\mathbf{w}) = \sqrt{\frac{1}{P} \sum_{j=1}^P (eGRF_{v,j} - mGRF_{v,j})^2} \quad (27)$$

$$\Leftrightarrow J(\mathbf{w})^2 = \frac{1}{P} \sum_{j=1}^P (eGRF_{v,j} - mGRF_{v,j})^2 \quad (28)$$

$$\Leftrightarrow J(\mathbf{w})^2 = \frac{1}{P} \sum_{j=1}^P \left(\sum_{i=1}^N (w_i m A_{v,ij}) - mGRF_{v,j} \right)^2 \quad (29)$$

The derivative of the loss function with respect to each weighting factor is then given by:

$$\frac{\partial J(w)}{\partial w} = 0 \quad (30)$$

$$\Leftrightarrow \frac{\partial}{\partial w} \frac{1}{P} \sum_{j=1}^P \left(\sum_{i=1}^N w_i m A_{v,ij} - m GRF_{v,j} \right)^2 = 0 \quad (31)$$

$$\Leftrightarrow \frac{1}{P} \sum_{j=1}^P 2 \left(\sum_{i=1}^N w_i m A_{v,ij} - m GRF_{v,j} \right) (m A_{v,ij} - m A_{v,Nj}) = 0 \quad \text{if } i \neq N \quad (32)$$

where the condition is met that the weighting factors should sum to one, by stating that

$$w_N = 1 - \sum_{i=1}^{N-1} w_i \quad (33)$$

The self-written algorithm of the stochastic gradient descent allows to impose additional conditions on the weighting factors. The method was implemented once such that the weighting factors for the left and right legs are symmetric, and once such that they are asymmetric. Furthermore, it was noted that it is important that the peaks of the estimated GRF and the measured GRF agree well. Therefore, the loss function was adjusted so that the error between the peaks is minimised instead of the entire curve.

An additional safety measure has been implemented whereby, when the algorithm diverges (i.e. when the loss function increases if the weighting factors are updated), the calculation stops immediately.

4.4.1.4 Number of Components

To further investigate the most contributing body segments, the link-segment model was calculated with a varying number of segments and corresponding weighting factors. The weighting factors used are listed in table 13. They are based on David Winter's table, and they continue to reflect the anatomy of the body so that the total sum remains one. This prevents a particular acceleration from having too much of an effect due to its large mass. If necessary, the weighting factors were adjusted using linear interpolation.

	Pelvis	Left thigh	Right thigh	Left shank	Right shank
1 body segment	1	-	-	-	-
	-	0.5	0.5	-	-
	-	-	-	0.5	0.5
2 body segments	0.678	0.161	0.161	-	-
	0.678	-	-	0.161	0.161
	-	0.3413	0.3413	0.1587	0.1587
3 body segments	0.678	0.1	0.1	0.061	0.061

Table 13: Weight factors for the analysis of the segments' importance

4.4.1.5 Principal component analysis

The aim of the principal component analysis is to identify generalised accelerations of the body segments and to identify their contribution to the Ground Reaction Force for running and jumping. The methodology was adopted from the research conducted by Federolf [37] and Verheul [30]. For completeness, a detailed description is provided below.

All five segmental accelerations in the medio-lateral (x), anterior-posterior (y) and vertical (z) direction are rearranged in an acceleration vector $\mathbf{a}$ for every time point t :

$$\mathbf{a}(t) = [\mathbf{a}_{x1}(t), \mathbf{a}_{y1}(t), \mathbf{a}_{z1}(t), \mathbf{a}_{x2}(t), \dots, \mathbf{a}_{z5}(t)] \quad (34)$$

The acceleration vectors were normalised and scaled, in order to assure that every participant equally contributes to the variance of the total acceleration. To do this, the average acceleration vector $\bar{\mathbf{a}} = [\bar{\mathbf{a}}_{x1}, \bar{\mathbf{a}}_{y1}, \dots, \bar{\mathbf{a}}_{z1}]$ for each participant is first calculated and subtracted from the acceleration vector:

$$\mathbf{a}'(t) = [\mathbf{a}_{x1}(t) - \bar{\mathbf{a}}_{x1}, \mathbf{a}_{y1}(t) - \bar{\mathbf{a}}_{y1}, \mathbf{a}_{z1}(t) - \bar{\mathbf{a}}_{z1}, \mathbf{a}_{x2}(t) - \bar{\mathbf{a}}_{x2}, \dots, \mathbf{a}_{z5}(t) - \bar{\mathbf{a}}_{z5}] \quad (35)$$

Secondly, the adjusted acceleration vector was divided by the mean Euclidean norm $\bar{d} = \overline{\|\mathbf{a}'(t)\|_2}$ of all acceleration vectors for each participant:

$$\mathbf{a}''(t) = \frac{1}{\bar{d}} \cdot \mathbf{a}'(t) \quad (36)$$

Finally, each acceleration vector was normalised for the relative segmental masses by multiplying them with a weighting vector $\mathbf{w}$, which contains mass ratios of each segment relative to the total body mass (equation 37). In this case the weighting factors were based on the tabulated values of Winter [13], with $w_{thigh} = 0.1$, $w_{shank} = 0.061$ and $w_{pelvis} = 0.678$.

$$\mathbf{a}'''(t) = \mathbf{w} \cdot \mathbf{a}''(t) \quad (37)$$

Thus, the normalised acceleration vectors are defined as follows:

$$\mathbf{a}'''(t) = \frac{1}{\bar{d}} \cdot [w_1(\mathbf{a}_{x1}(t) - \bar{\mathbf{a}}_{x1}), w_1(\mathbf{a}_{y1}(t) - \bar{\mathbf{a}}_{y1}), w_1(\mathbf{a}_{z1}(t) - \bar{\mathbf{a}}_{z1}), w_2(\mathbf{a}_{x2}(t) - \bar{\mathbf{a}}_{x2}), \dots, w_5(\mathbf{a}_{z5}(t) - \bar{\mathbf{a}}_{z5})] \quad (38)$$

The normalised acceleration vectors of the seven participants were combined by vertically stacking them into the acceleration matrix $\mathbf{A}$. The principal component analysis (PCA) is performed on this acceleration matrix $\mathbf{A}$. The PCA has three results: a matrix $\mathbf{EV}$ consisting of the orthogonal eigenvectors $\mathbf{ev}_k$ or principal components, a matrix consisting of the associated eigenvalues λ_k , and a time evolution matrix or score matrix $\mathbf{C}(t)$. Each eigenvector $\mathbf{ev}_k$

indicates the largest acceleration variability for all segments. The eigenvalues λ_k quantify the amount of variability described by each eigenvector $\mathbf{ev}_k$. The score matrix $\mathbf{C}$ describes how the original segmental accelerations evolve along the principal accelerations. It is calculated by projecting each normalised and scaled acceleration $\mathbf{a}'''(t)$ onto each eigenvector $\mathbf{ev}_k$:

$$\mathbf{C}(t) = \mathbf{A}(t) \cdot \mathbf{E}\mathbf{V} \quad (39)$$

The principal accelerations $\mathbf{PA}_k$ were reconstructed from each eigenvector $\mathbf{ev}_k$, in order to investigate how they contribute to the GRF. The principal acceleration $\mathbf{PA}_{1k}$ was also calculated as the sum of the first K reconstructions, in order to examine the number of eigenvectors required to accurately describe the GRF waveform. The principal accelerations were expressed in the original vector space by retracing the normalisation steps:

$$PA_k = \bar{\mathbf{a}} + \bar{\mathbf{d}} \cdot \mathbf{w}^{-1} \cdot \mathbf{C}_k(t) \cdot \mathbf{ev}_k \quad (40)$$

$$PA_{1k} = \bar{\mathbf{a}} + \bar{\mathbf{d}} \cdot \mathbf{w}^{-1} \cdot \left(\sum_{k=1}^K \mathbf{C}_k(t) \cdot \mathbf{ev}_k \right) \quad (41)$$

Because the reconstructed principal accelerations are consistent with Newton's laws, the principal segmental acceleration vectors can be used to calculate the GRF, called the principal GRF (pGRF). The pGRF was calculated according to the link-segment model (see equation 19):

$$\mathbf{pGRF} = m \cdot \mathbf{G} + \sum_{i=1}^N w_i \cdot m \cdot \mathbf{PA}_k \quad (42)$$

The accuracy of the principal GRF was evaluated as the root mean squared error (RMSE) with respect to the measured GRF.

4.4.1.6 Filter cut-off frequency analysis

It should be noted that the filtering settings of the IMU accelerations influence the estimated Ground Reaction Force [29][17][3]. A trade-off must be made between retaining the high-frequency components of the GRF, for example the impact peak, and avoiding to include too much noise. Therefore, it is essential to optimise the filtering settings in order to achieve the highest possible accuracy. The aim of this sensitivity analysis is to investigate what filter cut-off frequency and order of the low-pass filter lead to the most accurate resultant GRF estimates. In this analysis, the same filter is applied to all IMUs.

4.4.1.7 Ground Reaction Moment

The Ground Reaction Moment was calculated about the projection of the ankle joint on the ground. Two possible equations were initially implemented. The first equation (equation 43) is

taken from the literature review [3], and is based on Euler's equation of rigid body dynamics. The second equation 44 is derived from the formula of angular momentum, in which it was used that the time derivative of the angular moment equals the applied moment [38]. From visual inspection, it followed that equation 44 was most accurate in the medio-lateral direction, and equation 43 was most correct in the anterior-posterior and vertical directions. Subsequently, the GRM is normalised by dividing with body weight times body height.

$$GRM = - \left(\sum_{i=1}^N [\mathbf{J}_i \dot{\omega}_i + \omega_i \times (\mathbf{J}_i \omega_i)] - \sum_{i=1}^N \sum_{j=1}^{K_i} (\mathbf{r}_{ij} \times \mathbf{F}_{ij}) \right) \quad (43)$$

$$GRM = \sum_{i=1}^N \frac{\partial}{\partial t} (\mathbf{J}_i \omega_i + \mathbf{r}_i \times m_i \mathbf{v}_i) \quad (44)$$

where K_i : the number of endpoints in each segment (joints and external contact points like the ground)

ω_i and $\dot{\omega}_i$: the angular velocity and angular acceleration of each segment

$\mathbf{J}_i$: the inertia tensor around the CoM of each segment

$\mathbf{r}_{ij}$: the position vectors between the segment's CoM and the end points of each segment

$\mathbf{F}_{ij}$ is the resultant force in the end points of each segment.

For the inertia tensor, reference values from the literature provided the most reliable results [39]. These values are listed in table 14. It was also attempted to calculate the inertia of each segment by approximating the body segments as a cylinder. This method was found to be effective for the thigh and shank, however, the inertia of the pelvis was overestimated by a factor of 10 for each direction.

	$I_{xx} [kg \cdot m^2]$	$I_{yy} [kg \cdot m^2]$	$I_{zz} [kg \cdot m^2]$
Pelvis	0.23432	0.39353	0.33894
Thigh	0.19994	0.19978	0.04134
Shank	0.0371	0.0385	0.00646

Table 14: Reference values of inertia tensor from literature [39].

4.4.1.8 Centre-of-pressure

The centre of pressure can be computed from the Ground Reaction Force and Ground Reaction Moment using the following equation[13] [11]:

$$\mathbf{M} = \mathbf{r} \times \mathbf{F} + \mathbf{T} \Leftrightarrow \begin{cases} M_x = r_y \cdot F_z - r_z \cdot F_y + T_x \\ M_y = r_z \cdot F_x - r_x \cdot F_z + T_y \\ M_z = r_x \cdot F_y - r_y \cdot F_x + T_z \end{cases} \quad (45)$$

where **M**: the moment at the force plate origin, in this case the Ground Reaction Moment **GRM**
r: the vector from the origin to the centre of pressure. Assume that the surface is flat,
so $r_z = 0$
F: the Ground Reaction Force
T: the moment required to produce an equivalent system when the force **F** is applied
at the centre of pressure. Assume that friction can only generate a moment about the
vertical axis, so $T_y = T_z = 0$

If the following substitutions are made: $(r_x, r_y, r_z) = (COP_x, COP_y, COP_z)$, $(M_x, M_y, M_z) = (GRM_x, GRM_y, GRM_z)$ and $(F_x, F_y, F_z) = (GRF_x, GRF_y, GRF_z)$, and if the equations are solved to COP_x and COP_y , then

$$\begin{cases} CoP_x = -\frac{GRM_y}{GRF_z} \\ CoP_y = \frac{GRM_x}{GRF_y} \\ CoP_z = 0 \end{cases} \quad (46)$$

Notice that the equations are not valid when the vertical component of the Ground Reaction Force (GRF_z) approached zero. Therefore, the estimated centre of pressure is unreliable near initial contact and toe-off.

The calculation of the centre-of pressure is important because, besides the GRF and GRM, it is an unknown for inverse dynamics analysis. An inverse dynamic analysis computes the net forces and moments at each joint, by solving a cascade of free body diagrams of each joint, beginning at the feet. These values are mainly used in clinical practice to perform gait analyses, instead of accelerations of segments.

4.4.2 Two-body model

A schematic representation of the two-body model is shown in figure 30. Note that the two-body model is only valid for the GRF component in the vertical direction during running. The fundamental hypothesis of the two-body model is that the total vertical Ground Reaction Force is composed of two overlapping bell-shaped impulses, as described by Clark [4].

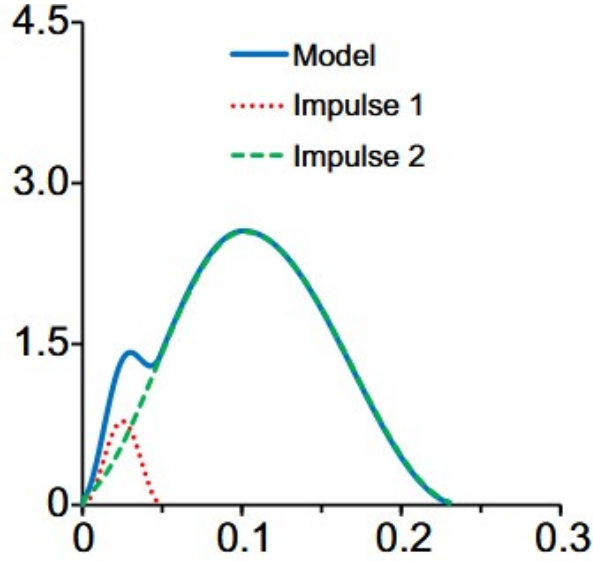


Figure 30: Schematic representation of the two-body model, that composes the Ground Reaction Force of two overlapping bell-shaped impulses.

The first impulse (J_1) is due to the vertical deceleration of the shank and the foot (m_1) during initial contact with the running surface. Therefore, the impact peak of the GRF waveform originates from this impulse.

$$J_1 = F_{1,avg}(2\Delta t_1) = (m_1 \frac{\Delta v_1}{\Delta t_1} + m_1 g)(2\Delta t_1) \quad (47)$$

where m_1 : mass of the shank and the foot

Δt_1 : time interval between initial contact and the vertical velocity of m_1 slowing to zero

Δv_1 : change in vertical velocity of m_1 during Δt_1

$F_{1,avg}$: the average force during the total time interval ($2\Delta t_1$) of impulse J_1

In the initial paper of Clark [4], initial contact is detected when the position of the foot marker reaches a minimum. The sensorised running pants do not have an IMU on the foot, so the shank IMU was used instead. The condition for detecting initial contact when the marker's position reaches a minimum is mathematically equal to the IMU's velocity crossing zero. Because there is a difference between the speed of the foot and the shank, a correction factor was added. With the speed of the recorded markers on the foot and shank in mind, after trying a few values, a factor of 2.2 for Δv_1 and a factor of 0.3 for Δt_1 provided the best results.

The second impulse (J_2) is due to the concurrent vertical accelerations of the rest of the body (m_2) after surface impact. It is determined from the total ground reaction impulse J_T and impulse J_1 :

$$J_T = F_{T,avg} \cdot t_c \quad (48)$$

$$= m \cdot g \cdot \frac{t_c + t_a}{t_c} \cdot t_c \quad (49)$$

$$J_2 = F_{2,avg} \cdot t_c \quad (50)$$

$$= J_T - J_1 \quad (51)$$

$$(52)$$

where t_c : period of time the foot is in contact with the ground (stance phase), i.e. from initial contact to toe-off

t_a : time period the foot is in the air (flight phase), i.e. from toe-off to the next initial contact

The average force $F_{2,avg}$ of the rest of the body (m_2) during the contact interval t_c is then:

$$F_{2,avg} = \frac{J_2}{t_c} \quad (53)$$

The Ground Reaction Force [26] is a summation of the two bell-shaped force curves obtained from the impulses J_1 and J_2 :

$$eGRF_z(t) = F_1(t) + F_2(t) \quad (54)$$

$$= F_{1,avg} \left[1 + \cos \left(\frac{t - \Delta t_1}{\Delta t_1} \pi \right) \right] + F_{2,avg} \left[1 + \cos \left(\frac{t - \frac{t_c}{2}}{\frac{t_c}{2}} \pi \right) \right] \quad (55)$$

In equation 54, the force waveform $F_1(t)$ is valid in the time interval $0 \leq t \leq 2 \Delta t_1$ for each step, and is equal to zero elsewhere. Analogously, the force waveform $F_2(t)$ is valid in the interval $0 \leq t \leq t_c$, and zero elsewhere.

4.4.3 Mass-spring-damper model

An alternative to the link-segment model and the two-body model to describe the dynamics of the human body during the stance phase of running, is a passive two-body mass-spring-damper model (MSD). The other mass-spring damper models discussed in the state-of-the-art were only theoretical models or not applicable with the IMUs in the running pants. A schematic representation of the model is shown in figure 31:

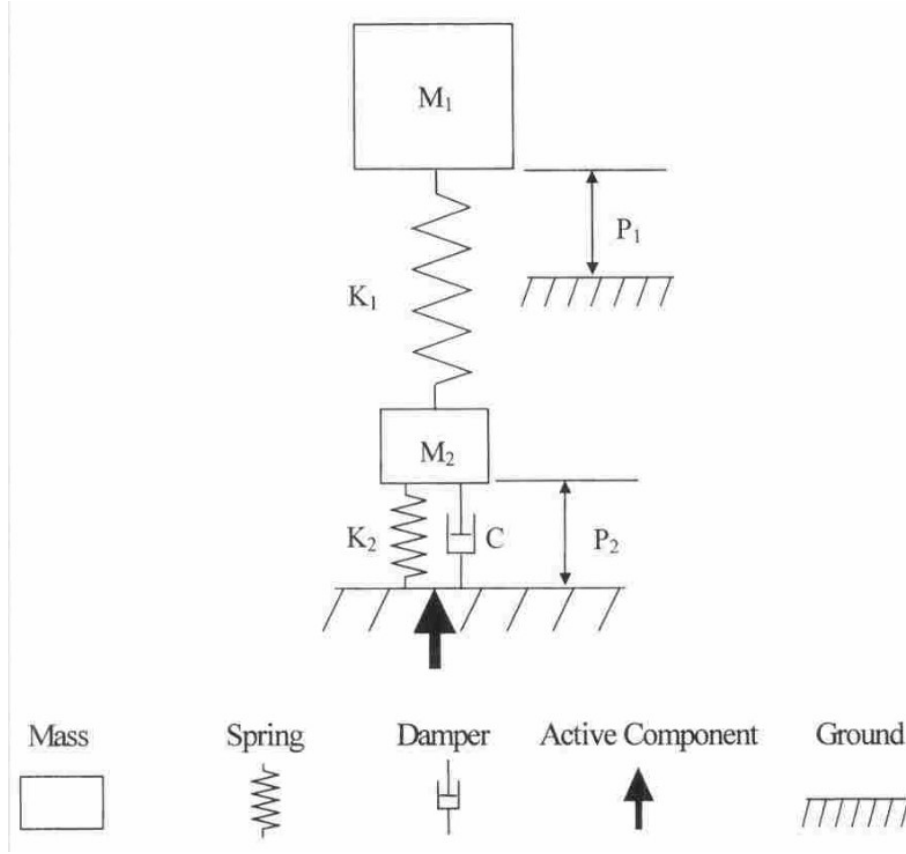


Figure 31: Schematic representation of the passive two-body mass-spring-damper model, as described by Derrick [12].

The equations for the vertical motion of the original MSD model are [12]:

$$a_1 = \frac{k_1}{m_1}(p_1 - p_2) + g \quad (56)$$

$$a_2 = -\frac{k_2}{m_2}p_2 - \frac{k_1}{m_2}(p_2 - p_1) - \frac{c}{m_2}v_2 + g \quad (57)$$

$$\text{GRFz} = k_2p_2 - cv_2 \quad (58)$$

where m_2 : the point mass of the lower element, representing the support leg
 m_1 : the point mass of the upper element, representing the remainder of the body
 p_1, v_1, a_1 : the vertical position, velocity and acceleration of m_1
 p_2, v_2, a_2 : the vertical position, velocity and acceleration of m_2
 k_1 : the linear spring constant of the upper spring
 k_2, c : the linear spring constant and damping coefficient of the lower spring and damper, respectively
 g : acceleration due to gravity (-9.81 m/s^2)

Because other studies did not provide a consistent formulation of the equations, it was decided to derive the equations of the original model in terms of natural parameters as described in

other studies [9] [6]. To do this, the original equations are rewritten by making the following substitutions:

$$\omega_1^2 = \frac{k_1}{m_1} \quad (59)$$

$$\omega_2^2 = \frac{k_2}{m_2} \quad (60)$$

$$\zeta = \frac{c}{2\sqrt{k_2 m_2}} \quad (61)$$

$$\lambda = \frac{m_1}{m_2} \quad (62)$$

$$(63)$$

where ω_1, ω_2 : natural frequency of the upper and lower spring, describing their stiffness

ζ : damping ratio of the damper

λ : mass ratio of the upper mass relative to the lower mass

The original model can then be rewritten as:

$$\mathbf{a}_1 = -\omega_1^2(\mathbf{p}_1 - \mathbf{p}_2) + g \quad (64)$$

$$\mathbf{a}_2 = -\omega_2^2 \mathbf{p}_2 + \omega_1^2 \lambda (\mathbf{p}_1 - \mathbf{p}_2) - 2\zeta \omega_2 \mathbf{v}_2 + g \quad (65)$$

The vertical GRF acting on the MSD-model is then calculated as

$$eGRF_z = \frac{(m_1 + m_2)\omega_2}{1 + \lambda}(\omega_2 \mathbf{p}_2 - 2\zeta \mathbf{v}_2) \quad (66)$$

The velocities of the upper and lower masses ($\mathbf{v}_1$ and $\mathbf{v}_2$) are determined by integrating the accelerations of the IMUs. Their positions ($\mathbf{p}_1$ and $\mathbf{p}_2$) can be calculated in a similar manner by integrating the accelerations twice. The parameters $\omega_1, \omega_2, \lambda$ and ζ in these equations are still unknown. With the prior knowledge of the weighting factors from the link-segment model, m_1, m_2 and therefore λ can already be estimated. With the weighting factors reported by Winter is m_2 equal to 1.61 % of the total body mass, and m_1 is equal 8.38 % of the total body mass. It follows that λ has a value of 5.21. Derrick's original model uses λ equal to 2, and Verheul's study recommends a value of 3 as theoretically reasonable and physically meaningful. Although the value of 5.21 is slightly higher, it does not seem potentially erroneous.

The other parameters are calculated by optimising the fitting of the model's accelerations (a_1 and a_2) to the measured accelerations of the IMUs. The acceleration of the support leg (a_1 is fitted to the average of the accelerations measured by the IMUs on the left thigh and the left shank. The acceleration of the remainder of the body (a_1) is fitted to the acceleration of the IMU on the pelvis, since this IMU is located closest to the CoM of the body.

The value of ω_1 is derived directly from the equation for a_1 :

$$\omega_1 = \begin{cases} \sqrt{-\frac{a_1 - g}{p_1 - p_2}} & \text{if } -\frac{a_1 - g}{p_1 - p_2} \geq 0 \\ 0 & \text{if } -\frac{a_1 - g}{p_1 - p_2} < 0 \end{cases} \quad (67)$$

Initially, the obtained array of values for ω_1 was averaged, but the estimation was found to work better when ω_1 varied over time (see results).

The calculation of parameters ω_2 and ζ from acceleration a_2 is less straightforward, because two unknown parameters must be determined from only one known variable. Since a varying parameter over time worked best for ω_1 , it was chosen to follow this approach for ω_2 and ζ as well. Therefore, two additional equations were formulated:

$$\frac{\partial a_2}{\partial \zeta} = -2\omega_2 \mathbf{v}_2 \quad (68)$$

$$\frac{\partial a_2}{\partial \omega_2} = -2\omega_2 \mathbf{p}_2 - 2\zeta \mathbf{v}_2 \quad (69)$$

These two first-order differential equations were solved with a self-written algorithm. In order to work, the algorithm needs initial values for ω_2 and ζ . These were chosen as 19 and 0.31, respectively, because these values were also used in the original model. The initial values for ω_2 and ζ are set at the starting time point ($t = 0s$). Then, based on the previous time step, a prediction of ω_2 and ζ is made for the next time step with the explicit forward Euler method.

$$\mathbf{y}_{i+1} = \mathbf{y}_i + h\mathbf{f}(\mathbf{x}_i, \mathbf{y}_i) \quad (70)$$

In this formula, $\mathbf{y}_i (\in \mathbb{R}^{N \times 2})$ contains the estimates of ω_2 and ζ at the previous time points i , $\mathbf{y}_{i+1}$ is the estimate at the next time point for the N time points in the time interval, and $\mathbf{f}(\mathbf{x}_i, \mathbf{y}_i)$ is given by equations 69

This prediction was improved ten times with the second-order Adams-Moulton method. Then it is shifted further to the next time step, and so on until the parameters are estimated at all N time steps.

The second-order Adams-Moulton method was used to improve this prediction tenfold. The process is then repeated for the next time point, until the parameters are estimated at all N time points.

$$\mathbf{y}_{i+1} = \mathbf{y}_i + \frac{h}{2} \mathbf{f}(\mathbf{x}_i, \mathbf{y}_i) + \mathbf{f}(\mathbf{x}_{i+1}, \mathbf{y}_{i+1}) \quad (71)$$

It was also tried to implement that the algorithm would only proceed to the subsequent time point if the error fell below a predefined threshold. However, this approach introduced the potential risk of algorithm failure if the threshold was not met.

5 Results

5.1 Data analysis

A first question is whether any anomalous results are due to the sensor accuracy or the model. To determine the impact of the sensors, the similarity of the accelerations and angular velocities measured by the IMUs and derived from the reflective markers are compared. A visual comparison of the accelerations derived from the markers and those measured by the IMUs in the running pants reveals that the measurements are similar. A representative example is shown in Figure 32. Quantitatively, the Pearson Correlation Coefficient (ρ) of both signals is 0.40 ± 0.26 in the medio-lateral direction, 0.81 ± 0.91 in the anterior-posterior direction, and 0.84 ± 0.13 in the vertical direction. The correlation remained approximately the same for all running speeds and jumping. It can be concluded that possible inaccuracies of the estimated GRF and the measured GRF are not due to bad measurements of the sensors for the anterior-posterior and vertical directions. In the medio-lateral direction it could be the case, but this direction is less important for the GRF of sagittal movements. However, for the GRM this acceleration in the medio-lateral direction is also important.

When looking at the accelerations in the vertical direction of each body segment, which are most important for the model, the accelerations measured at the pelvis are most similar ($\rho = 0.99 \pm 0.05$). This result was expected, as this marker and IMU are closest to the CoM of the body, where the accelerations do not show extreme changes and values. The pelvis is followed by the left thigh and the right thigh, respectively with correlation coefficients of $\rho = 0.92 \pm 0.07$ and $\rho = 0.90 \pm 0.09$. The most deviating are the vertical accelerations of the shanks, with $\rho = 0.70 \pm 0.24$ for the left shank and $\rho = 0.70 \pm 0.28$ for the right shank. The legs experience the highest accelerations and acceleration changes shortly after ground contact, because they endure most of the surface impact. The Pearson Correlation Coefficients are summarised in table 15.

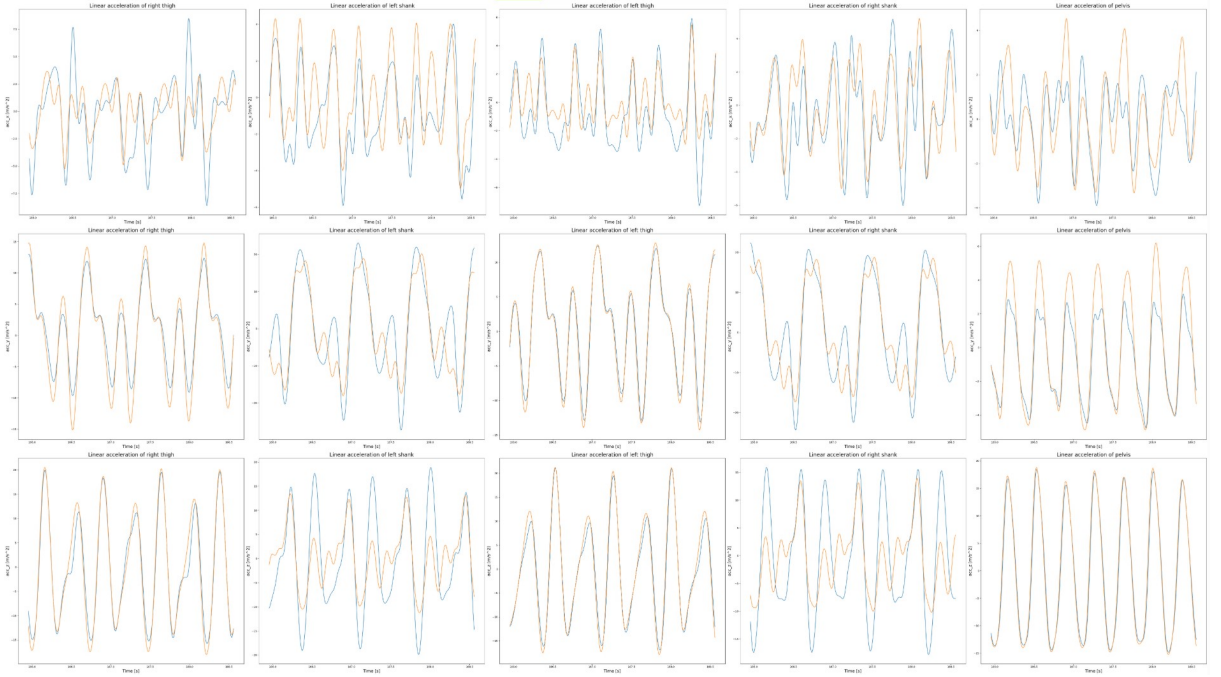


Figure 32: Representative example of the comparison between the accelerations of the IMUs (blue) and the markers (orange). The top row are the accelerations in the medio-lateral direction, the middle row in the anterior-posterior direction and the bottom row in the vertical direction. The columns show the body segments, from left to right: right thigh, left shank, left thigh, right shank and pelvis.

	Medio-lateral	Anterior-posterior	Vertical				
			Left Thigh	Right Thigh	Left Shank	Right Shank	Pelvis
Pearson Correlation Coefficient [%]	0.40 ± 0.26	0.81 ± 0.91	0.84 ± 0.13	0.92 ± 0.07	0.70 ± 0.24	0.70 ± 0.28	0.99 ± 0.05

Table 15: Overview of Pearson Correlation Coefficients to determine the similarity between the accelerations measured by the IMUs and derived from reflective markers.

Besides the linear accelerations, the angular velocities are an important component of the model. The angular velocity describes how quickly an object rotates around an axis of rotation. The angular velocity is a direct output from the IMUs, but not from the markers. The marker trajectories were loaded into the software OpenSim, where the angular velocities can be requested as an output. A representative example of the comparison between the angular velocities measured by the IMUs and derived from the markers is shown in figure 33. The Pearson Correlation Coefficient of both signals is 95.6 ± 4.36 for the angular velocity of the rotation around the medio-lateral axis, 0.49 ± 0.23 for the rotation around the anterior-posterior axis and 0.34 ± 0.26 for the rotation around the vertical axis. When the angular velocities of the body segments around the vertical direction are examined in more detail, the pelvis is on

average the most accurate ($\rho = 0.63$), followed by the shanks (left: $\rho = 0.53$ and right: $\rho = 0.37$), and finally the thighs (left: $\rho = -0.13$ and right: $\rho = 0.29$). The Pearson Correlation Coefficients are summarised in table 16.

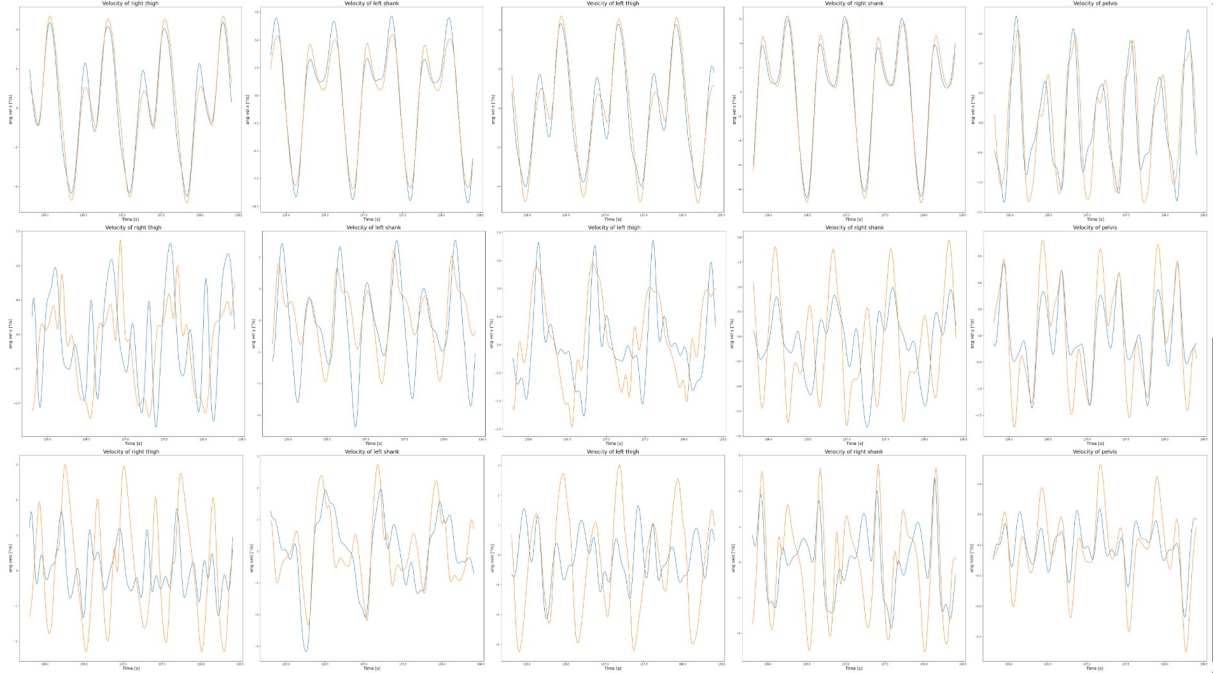


Figure 33: Representative example of the comparison between the angular velocities of the IMUs (blue) and the markers (orange). The top row are the angular velocities in the medio-lateral direction, the middle row in the anterior-posterior direction and the bottom row in the vertical direction. The columns show the body segments, from left to right: right thigh, left shank, left thigh, right shank and pelvis.

	Medio-lateral	Anterior-posterior	Vertical				
			Left Thigh	Right Thigh	Left Shank	Right Shank	Pelvis
Pearson Correlation Coefficient [%]	95.6 ± 4.36	0.49 ± 0.23	- 0.13	0.29	0.53	0.37	0.63

Table 16: Overview of Pearson Correlation Coefficients to determine the similarity between the angular velocities measured by the IMUs and derived from reflective markers.

This analysis suggests that the signals from the IMUs and markers are quite similar, and therefore models from the literature that involve markers can also be applied with IMUs. Possible unexpected deviations in the model will be related to errors in the measured angular velocities of the IMUs rather than the linear accelerations.

5.2 Step detection

5.2.1 Step detector 1: arbitrary threshold on the vertical Ground Reaction Force

Figure 34 shows a representative example of the algorithm. Initial contact and toe-off are indicated as expected, although the algorithm is susceptible for the chosen threshold value. The higher the threshold value, the later initial contact is detected. In contrary, if the threshold is chosen too low, initial contact could be detected already during flight phase.

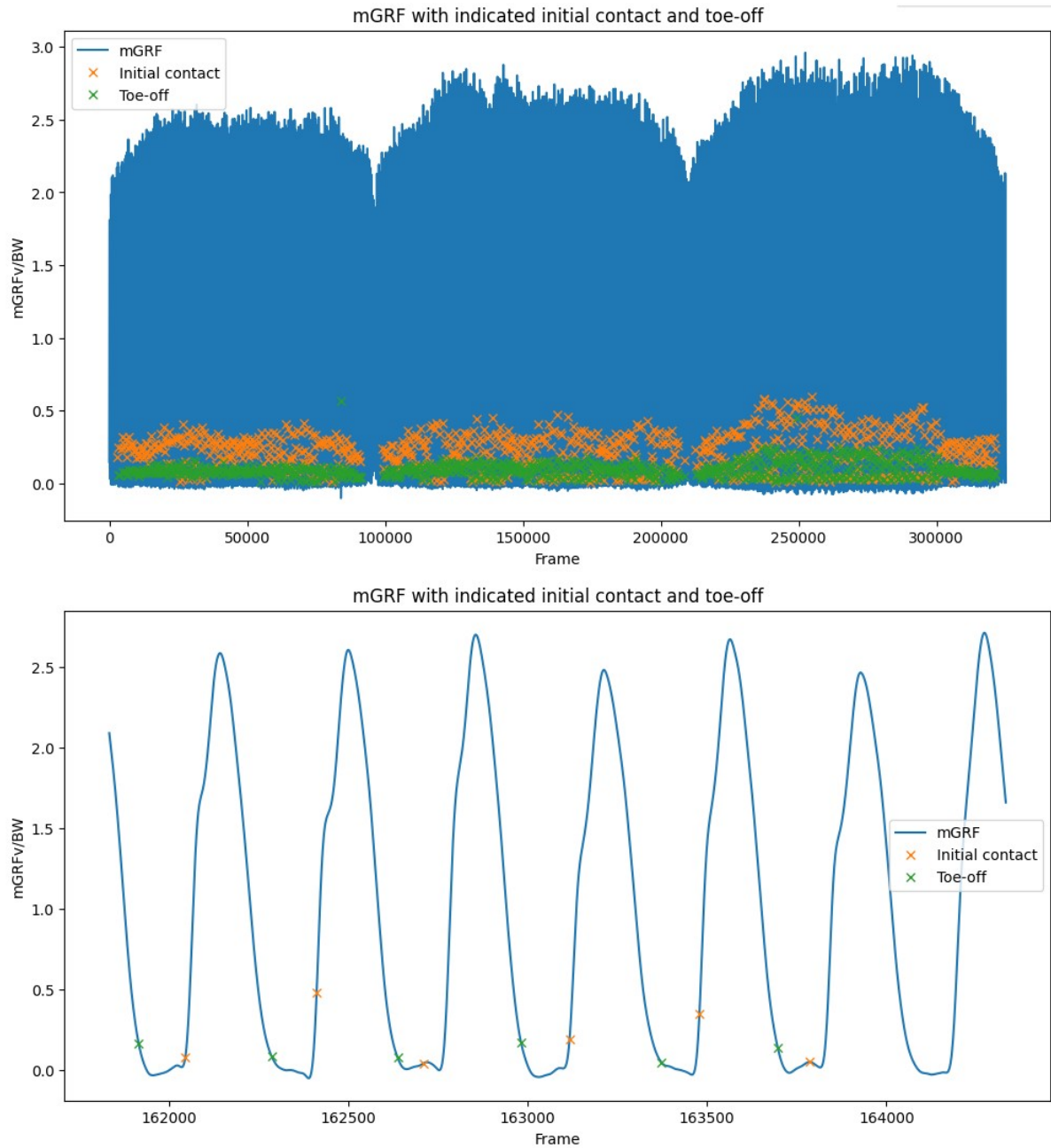


Figure 34: Representative example of step detector 1, which used an arbitrary threshold value on the Ground Reaction Force to determine initial contact and toe-off. For each step, initial contact is indicated with an orange cross, and toe-off with a green cross.

5.2.2 Step detector 2: quaternions

The results of the algorithm did not fully meet the expectations, as shown in figure 35. The algorithm detects the moment when the vertical Ground Reaction Force reaches its active peak, rather than initial contact and toe-off. An unexpected advantage of this algorithm is that it can distinguish whether a step has been taken with the left or right leg. This is because the threshold value is only reached by the body segment to which the IMU is attached.

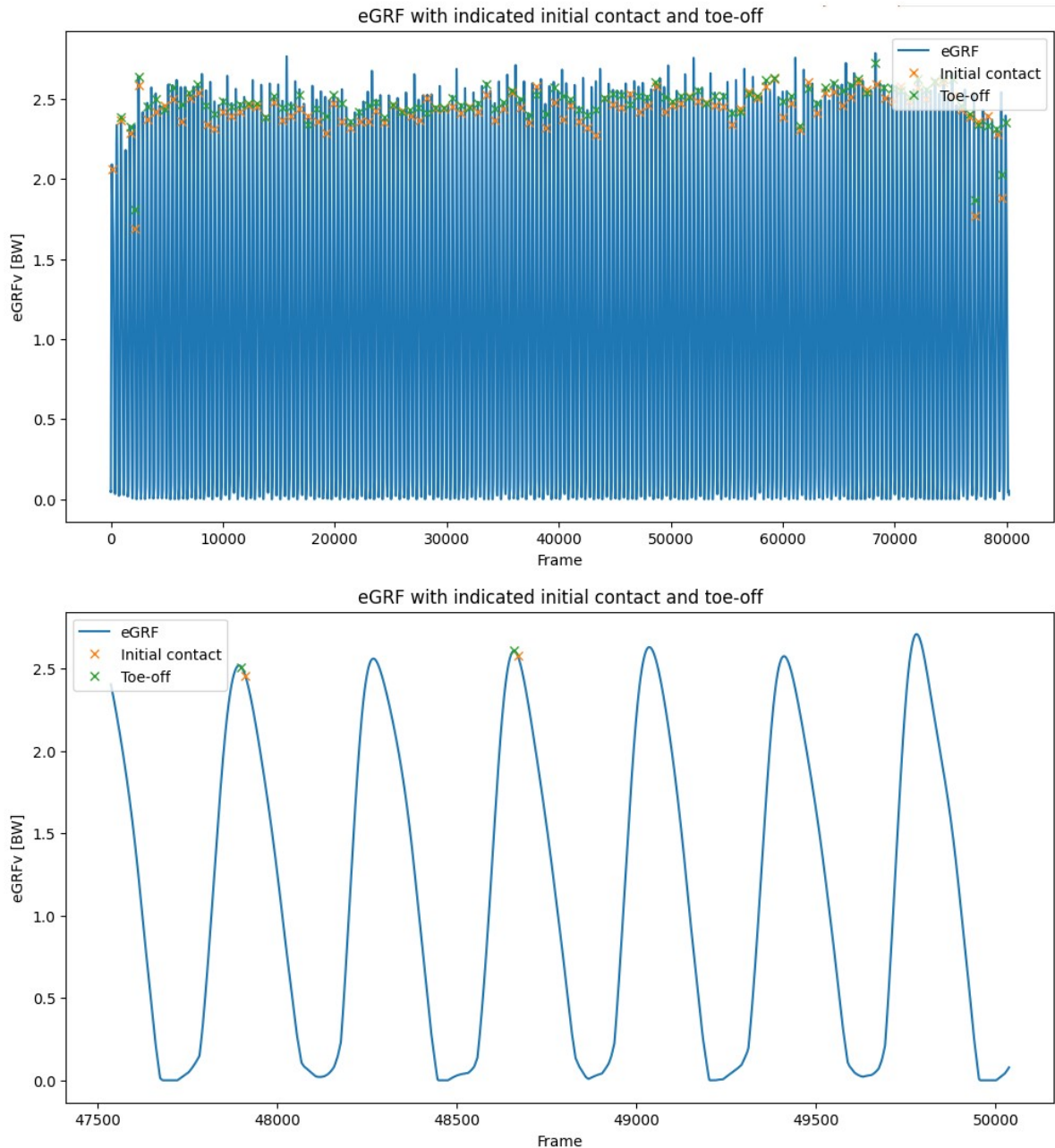


Figure 35: Representative example of step detector 2, which used the quaternions of the IMU on the right shank to determine initial contact and toe-off. For each step, initial contact is indicated with an orange cross, and toe-off with a green cross.

5.2.3 Step detector 3: velocity of the pelvis

A representative example of the results is shown in figure 36. Initial contact and toe-off are detected well. When step detector 3 is compared with the steps of the mGRF from the threshold-based step detector, initial contact is detected earlier and toe-off occurs around the same time for the same step in the recording. When the eGRF and mGRF steps are later compared one-to-one, this time difference causes an additional error.

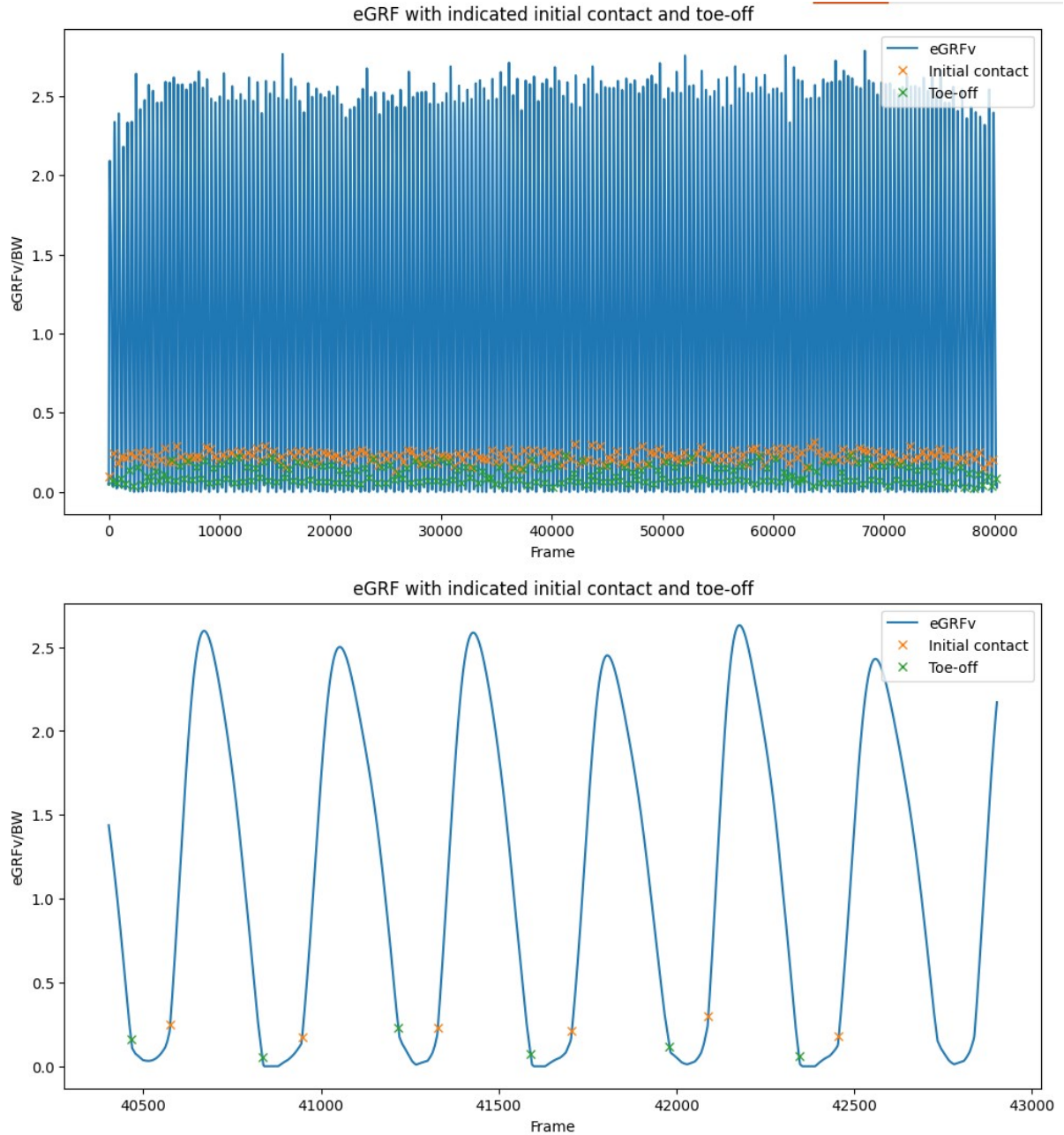


Figure 36: Representative example of step detector 3, which used the peak downward velocity of the pelvis to determine initial contact and toe-off. For each step, initial contact is indicated with an orange cross, and toe-off with a green cross.

5.2.4 Analysis of the detected steps

Based on the output of the first step detector, the mGRF recordings contained 1337 steps at 8 km/h, 1630 steps at 10 km/h and 1767 steps at 12 km/h. Altogether, this makes a total of 4734 steps. The third-step detector was used to analyse the eGRF, showing that 1472 steps had been taken at 8 km/h, 1796 at 10 km/h, and 1732 at 12 km/h, making a total of 5000 steps. An overview of the detected steps, both per participant as per running task, is given in table 17.

As a manual count of the actual number of steps would be very labour-intensive, it was not possible to determine the performance of both algorithms, for example with a confusion table. Because of the higher number of steps from step detector 3, and assuming that the threshold-based step detector of the mGRF is the most accurate, it can be concluded that the step detector for the eGRF is a bit oversensitive. The results are roughly similar for all participants, suggesting that both step detectors are independent of the participant's running style like rearfoot strike or forefoot strike. Running speed also seems to have no effect on the performance.

To ensure that the same steps were compared between the eGRF and the mGRF, an additional algorithm was introduced to check that the time interval between the corresponding initial contacts of the mGRF and the eGRF was not too long. After this check, the number of steps analysed was 1337 at 8 km/h, 1630 at 10 km/h, and 1615 at 12 km/h. In most cases, the number of steps checked is equal to the minimum of the mGRF and eGRF step counts. However, for the trial of person 7 at 12 km/h, the difference is significantly higher. This is probably due to poor synchronisation between the IMU and the treadmill signals.

	8 km/h			10 km/h			12 km/h		
	IMU	mGRF	check	IMU	mGRF	check	IMU	mGRF	check
Participant 1	155	155	155	166	164	164	-	-	-
Participant 2	169	167	167	170	168	168	176	173	170
Participant 3	223	188	188	288	268	268	388	360	356
Participant 4	262	232	232	313	289	289	331	314	310
Participant 5	211	205	205	315	311	311	301	306	299
Participant 6	194	174	174	269	209	209	327	297	295
Participant 7	258	216	216	275	221	221	209	317	185
Total	1472	1337	1337	1796	1630	1630	1732	1767	1615

Table 17: Overview of the detected steps for step detector 1, step detector 3 and the steps from the control algorithm.

5.2.5 Jump detection

Figure 37 shows a representative example of the algorithm's results. A total of 29 jumps were detected based on the IMUs and 30 jumps based on the mGRF. An overview is provided in table 18. The algorithm recognised 74 % of the jumps based on accelerations and 70 % based

on mGRF. The first jump was mostly not detected because of the required margin of 1 s at the beginning of the recording, and the same applied to the last 1 s of the recording for the last jump. Therefore, the percentage of detected jumps may be slightly higher in reality.

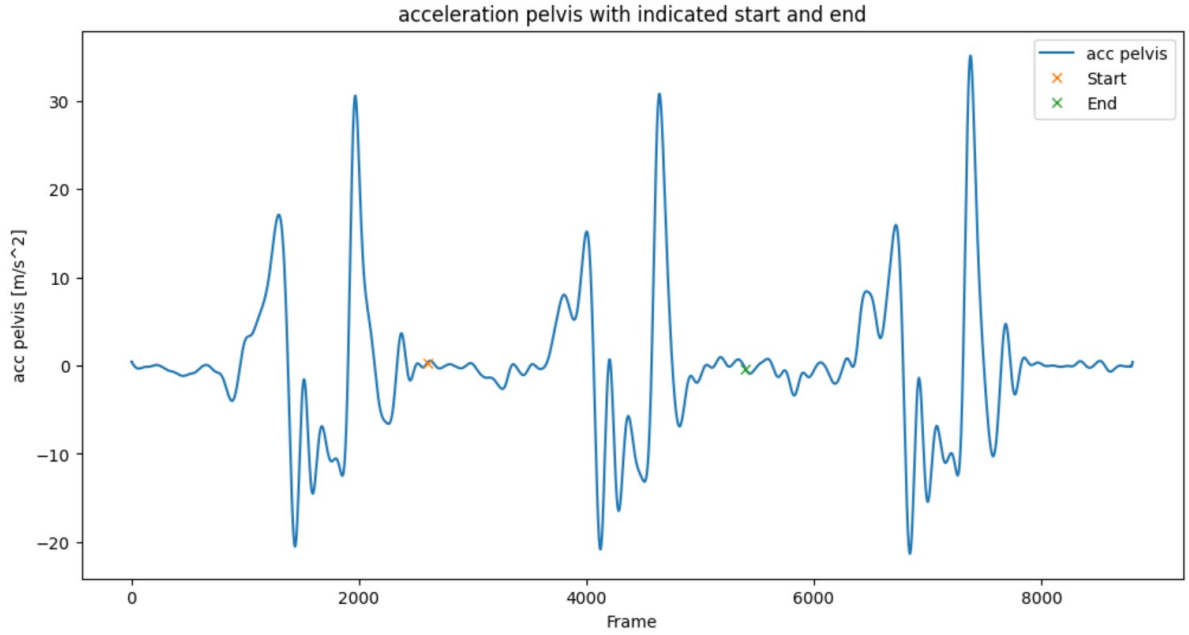


Figure 37: Representative example of the jump detector. The start of the jump is indicated with an orange cross, and the end with a green cross.

	IMU	mGRF	expected
Participant 1	0	0	0
Participant 2	1	1	3
Participant 3	9	7	10
Participant 4	8	7	10
Participant 5	1	1	3
Participant 6	7	8	10
Participant 7	8	8	10
In total	34	32	46

Table 18: Performance of jump detector

5.3 Link-segment model

As outlined in the definition of the link-segment model, the accelerations and weighting factors are crucial for achieving an accurate approximation. The following paragraphs will provide a more in-depth discussion of these components with the aim of further improving them.

5.3.1 Translated accelerations

The effect of the translation on the accelerations is shown in figure 38: at some points in time the acceleration increases, and at other points it decreases. When the Pearson Correlation Coefficient is used to assess the similarity between the original and displaced accelerations, the medio-lateral direction is modified the most ($\rho = 0.94 \pm 0.11$). Next comes the anterior-posterior direction ($\rho = 0.97 \pm 0.03$), and finally the vertical direction ($\rho = 0.97 \pm 0.02$). At the level of body segments, the shanks change the most (left: $\rho = 0.87 \pm 0.07$, right: $\rho = 0.99 \pm 0.01$), followed by the pelvis (0.99 ± 0.01), and finally the thighs (left: $\rho = 1$, right: $\rho = 1$). It is unclear whether this adjustment has truly corrected the accelerations, as the accelerations in the CoM of the segment are not known. An attempt was made to determine these with OpenSim, but unfortunately this was unsuccessful, most likely due to the manner in which the software calculates accelerations.

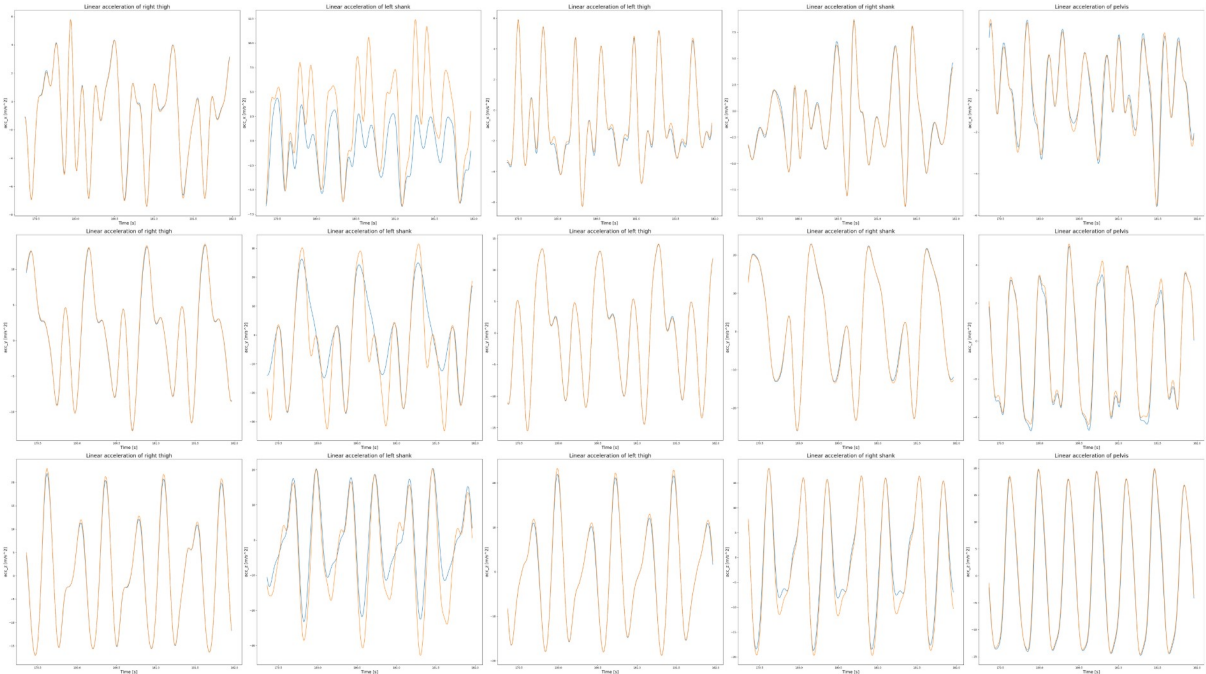


Figure 38: Representative example of the comparison between the original accelerations of the IMUs (blue) and the translated accelerations (orange). The top row are the accelerations in the medio-lateral direction, the middle row in the anterior-posterior direction and the bottom row in the vertical direction. The columns show the body segments, from left to right: right thigh, left shank, left thigh, right shank and pelvis.

Therefore, it is hard to conclude whether translating the accelerations improved the GRF estimation. The translation increased the GRF in the vertical direction, which was beneficial for some participants but not for others. Furthermore, it improved and deteriorated the GRF in the medio-lateral and antero-posterior direction, depending on the time point. In general, the translation had an effect of $\pm 2\%$ on the rRMSE of the GRF estimation. A representative example of the approximate GRF with original and translated accelerations is given in figure 39. Due to the ambiguity of the result, it was decided to continue the model development with

the original accelerations.

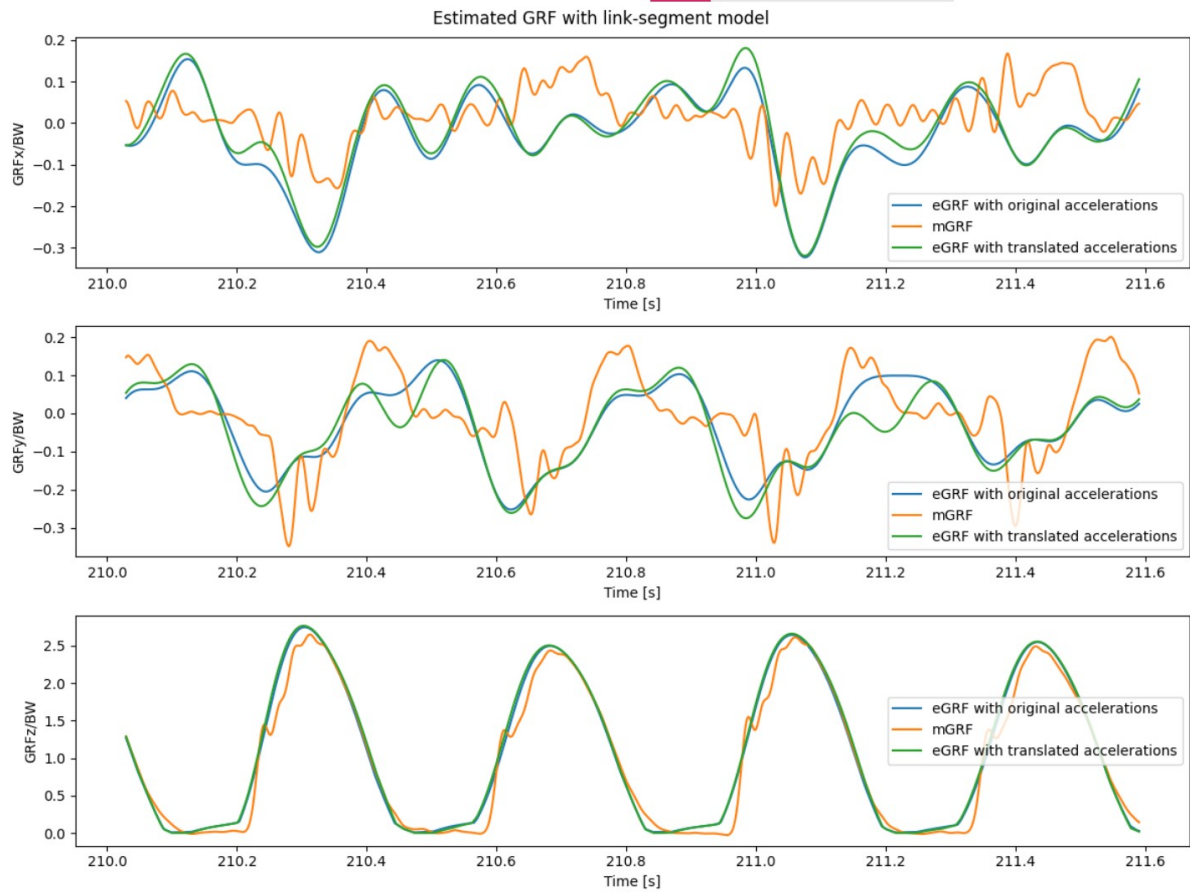


Figure 39: Representative example of the comparison between the estimated GRF from the original accelerations of the IMUs (blue), the estimated GRF from the translated accelerations (green) and the measured GRF (orange). The top row is the GRF in the medio-lateral direction, the middle row in the anterior-posterior direction and the bottom row in the vertical direction.

5.3.2 Model performance

The main performance metric to evaluate the link-segment model is the relative root mean squared error (rRMSE) between the mGRF and eGRF during the stance phase, and analogously for the GRM and CoP. For the GRF during running, the relative loading rate error, the relative impulse error and the relative peak error are also used. All metrics were calculated as described in section 3.1.

5.3.2.1 Ground Reaction Force

The mean GRF and one standard deviation of the estimated GRF (blue) and measured GRF (orange) during running and jumping are depicted in figure 40 and figure 41 respectively. The average was taken across all running speeds and participants.

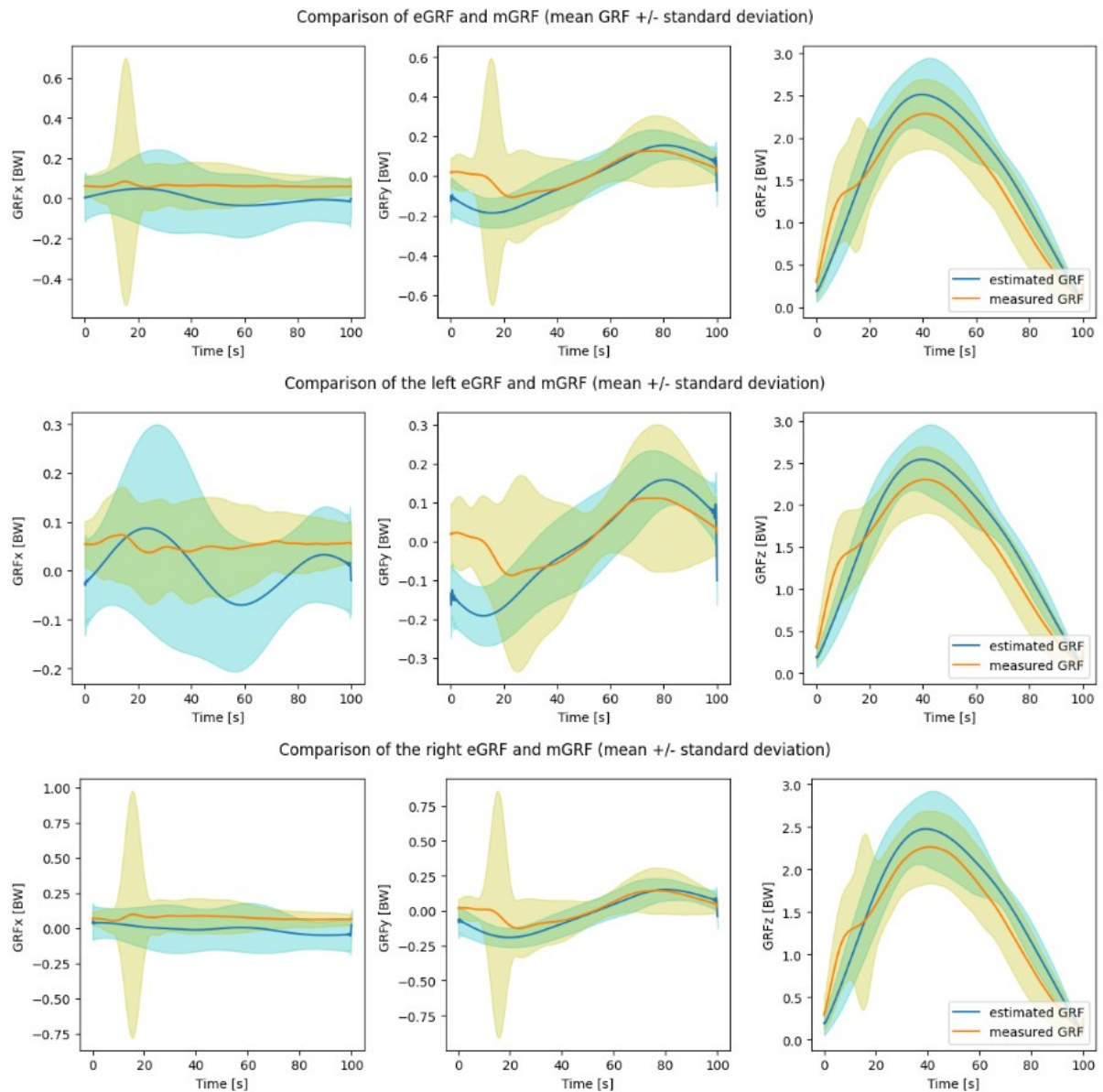


Figure 40: Mean and standard deviation of the estimated (blue) and measured Ground Reaction Force (orange) during running for all participants.

For running, it was observed that some participants exhibited a discrepancy in leg strength. For instance, the mGRF for the left leg was larger than the mGRF for the right leg. Therefore, the decision was taken to report the GRF for both legs separately. However, when the mean of all participants is taken, this effect is less visible.

The results are in line with the expectations, but there is an unexpected peak in the standard deviation of the medio-lateral mGRF and the anterior-posterior mGRF of the right leg. This peak is not usually present, and it is therefore suspected that this error is due to resampling.

Finally, it should be noted that the eGRF does not show an impact peak in the vertical direction. However, this is an important component in the context of injury prevention. This could be

due to a too low cut-off frequency for the low-pass filter of the signals. Therefore, it was decided to do a filter cut-off frequency analysis (see section 4.4.1.6).

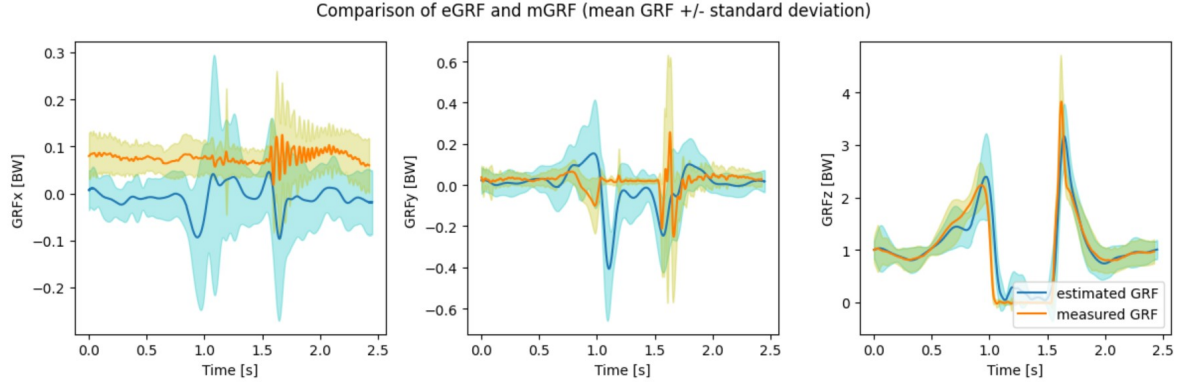


Figure 41: Mean and standard deviation of the estimated (blue) and measured Ground Reaction Force (orange) during jumping for all participants.

Despite the fact that a jumping movement is less repetitive than a running movement, the measured GRF and the estimated GRF visually match well. Especially in the anterior-posterior direction and in the vertical direction the similarity is great. In the medio-lateral direction, the measured GRF shows an offset of approximately 0.1 times the body weight. This could be a calibration error of the instrumented treadmill, implying that this offset is also present in the measured Ground Reaction Force for running. The prominent oscillations in the medio-lateral mGRF and anterior-posterior mGRF around 1.5 seconds occur when the participant lands after the jump. It is more difficult to capture these with the IMUs, due to the lower cut-off frequencies applied ($f_{c_{treadmill}} = 30Hz$ and $f_{c_{IMU}} = 6.5Hz$).

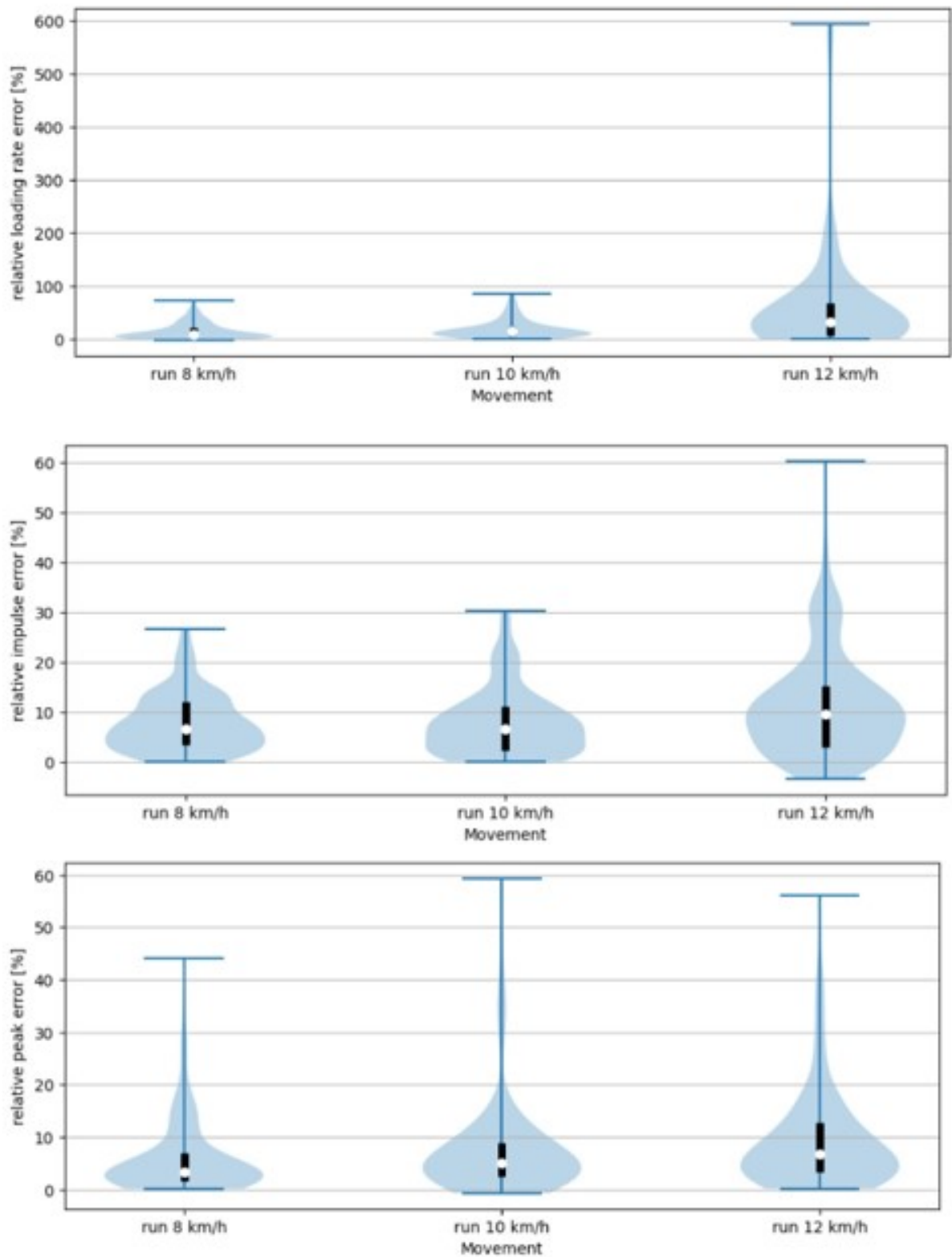


Figure 42: Violin plot of the loading rate error, impulse error and peak error of the estimated Ground Reaction Force of the link-segment model, during running for all participants.

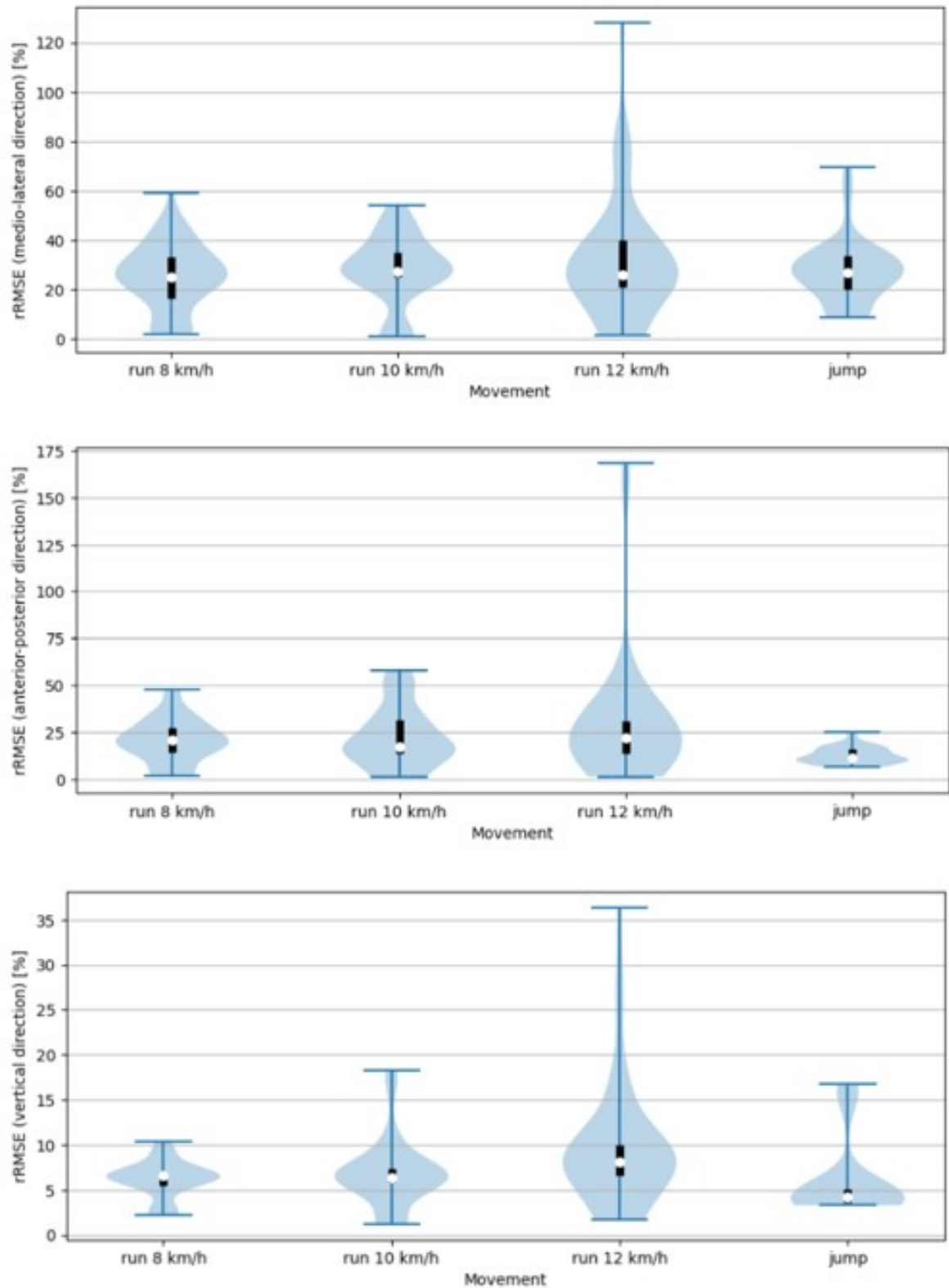


Figure 43: Violin plot of the relative root mean squared error of the estimated Ground Reaction Force of the link-segment model, during running for all participants.

The visual analysis can be made more quantitative by applying the metrics from section 3.1. The results are presented using violin plots (figures 42 and 43) [40]. Violin plots are similar to

boxplots, but the probability density (blue shaded area) of the data points is added to have a clearer indication of the spread of the data. The median is represented by a white sphere, the black box runs from the first quartile to the third quartile, and the blue bars extend from the minimum to the maximum. Generally, the higher the speed, the larger the error. The relative and absolute loading rate error are equal to $32.69 \pm 68.71\%$ and $12110 \pm 24620 \text{ N/s}$ over all speeds, which is quite high. However, this high value is mainly due to the poor performance at 12 km/h (8 km/h: $14.91 \pm 15.50\%$ or N/s , 10 km/h: $18.93 \pm 17.71\%$, 12 km/h: $64.21 \pm 110.05\%$). The relative impulse error and the relative peak error are acceptable, with $8.99 \pm 8.20\%$ (absolute: $33 \pm 27 \text{ N} \cdot \text{s}$) and 7.93 ± 9.27 (absolute: $96 \pm 55 \text{ N}$) respectively. The rRMSE for the GRF in the vertical direction was smallest, with a value of $7.18 \pm 4.87\%$ (absolute: $124 \pm 137 \text{ N}$), followed by the anterior-posterior direction (relative: $21.15 \pm 21.08\%$, absolute: $103 \pm 102 \text{ N}$) and the medio-lateral direction ($27.81 \pm 16.95\%$, absolute: $89 \pm 53 \text{ N}$). This result was also expected based on the accelerations, where the IMU accelerations were most similar to the accelerations of Vicon in the vertical direction, followed by the accelerations in the anterior-posterior direction and medio-lateral direction. All results are summarised in table 19. When these results are compared with those of the studies on link-segment models from the literature study, the results are approximately the same.

	8 km/h	10 km/h	12 km/h	Jump	All
Relative loading rate error [%]	14.91 ± 15.50	18.93 ± 17.71	64.21 ± 110.05	-	32.69 ± 68.71
Relative impulse error [%]	7.84 ± 5.87	7.78 ± 6.67	11.37 ± 10.68	-	8.99 ± 8.20
Relative peak error [%]	6.26 ± 7.39	8.02 ± 10.04	9.51 ± 9.86	-	7.93 ± 9.27
rRMSE (medio-lateral direction) [%]	25.06 ± 13.50	27.38 ± 14.05	31.45 ± 24.13	27.34 ± 12.91	27.81 ± 16.95
rRMSE (anterior-posterior direction) [%]	20.60 ± 11.54	22.04 ± 15.81	29.51 ± 35.07	12.48 ± 4.40	21.15 ± 21.08
rRMSE (vertical direction) [%]	6.22 ± 2.00	6.78 ± 3.86	9.64 ± 6.94	6.11 ± 4.42	7.18 ± 4.87

Table 19: Performance of the link-segment model to estimate the Ground Reaction Force for running and jumping. The mean and standard deviation were calculated across the seven participants.

5.3.2.1.1 Weighting factors

An overview of the calculated weighting factors per body segment (column), per assignment and participant (row), is shown in figure 44.

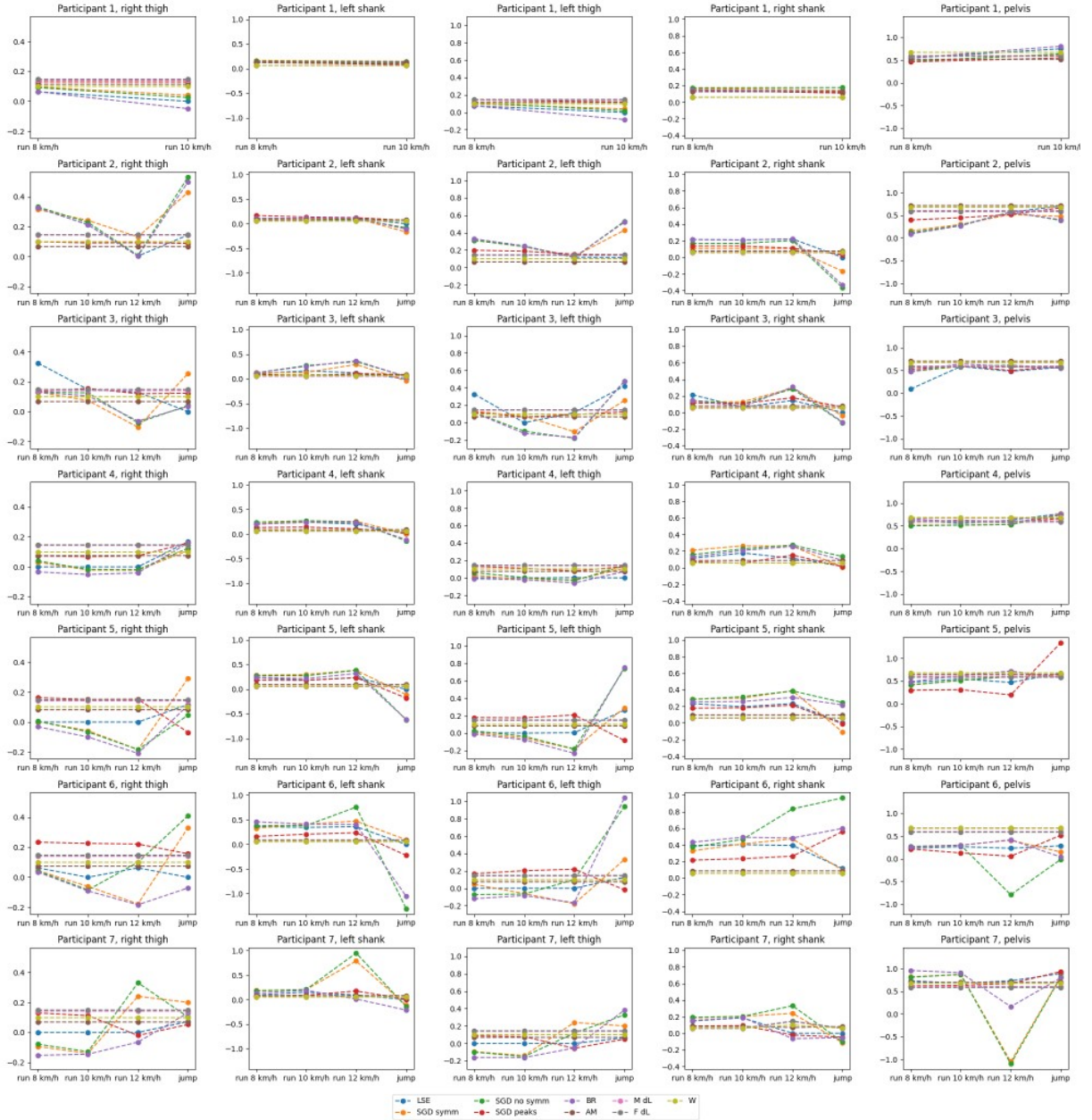


Figure 44: Overview of the calculated weighting factors per body segment (column), per assignment and participant (row). The weighting factors calculated using machine learning are LSE (Linear Regression), BR (Bayesian Ridge Regression), SGD (Stochastic Gradient Descent) with the conditions of only symmetric weighting factors (SGD symm), non-symmetric weighting factors (SGD nosymm), and fitting based on the peaks (SGD peaks). For the weighting factors based on anthropometric measurements, there are MdL (De Leva for men), FdL (De Leva for women), W (Winter), and AM (method based on approximate volume of the body segment).

In general, the values of the weighting factors per participant are all similar to each other, regardless of whether they are calculated based on anthropometric calculations or machine learning.

When the Linear Regression algorithm was applied on the IMU accelerations and the measured GRF, the coefficients were positive or zero. A zero coefficient would mean that these accelerations were irrelevant. This sometimes occurred with the acceleration of the thigh or shank, but never with the acceleration of the pelvis. One explanation for this could be that the accelerations deviated too much from the expected measured GRF pattern. Additionally, the sum of the coefficients was close to one, showing that summing the IMU accelerations with appropriate weighting factors is an effective way to assess the whole-body load represented by the Ground Reaction Force.

Due to the Bayesian framework, the weighting factors found with the Bayesian Ridge Regression method are slightly different to the ones found with Linear Regression. However, Bayesian Ridge Regression is more robust to ill-posed problems. An unexpected result was that some of the weighting factors were negative, which would physically imply a negative mass of the body segment. Interestingly, these negative weighting factors corresponded to the zero coefficients of Linear Regression.

The weighting factors calculated with the self-written Stochastic Gradient Descent algorithm are consistent with those calculated with the Bayesian Ridge Regression algorithm. With this method, certain weighting factors also became negative.

The assignment seems to have little influence on the weighting factors from machine learning, which is beneficial as it ensures the consistency of the weighting factors. From an anatomical perspective, it is logical that the mass of a body segment remains the same when the participant performs a different movement. There is a slight trend that at higher speeds the accelerations of the thighs become less meaningful, and those of the shanks become more important.

	Relative loading rate error [%]			Relative Impulse error [%]		
	8 km/h	10 km/h	12 km/h	8 km/h	10 km/h	12 km/h
LSE	9.35 ± 8.68	13.53 ± 6.01	26.32 ± 10.31	5.36 ± 2.32	5.93 ± 1.77	7.65 ± 0.82
SGD symm	9.31 ± 7.40	15.13 ± 6.61	41.89 ± 33.64	5.64 ± 2.25	6.65 ± 2.01	10.96 ± 4.17
SGD no symm	9.41 ± 7.51	14.85 ± 6.66	44.66 ± 32.23	5.51 ± 2.09	6.36 ± 1.94	18.65 ± 15.10
SGD peaks	8.75 ± 6.63	13.53 ± 6.93	31.71 ± 9.61	6.14 ± 2.73	5.71 ± 1.81	7.71 ± 1.37
BR	9.75 ± 8.39	14.73 ± 6.94	38.80 ± 26.58	5.32 ± 2.34	6.50 ± 1.82	13.59 ± 8.98
AM	21.07 ± 18.09	23.53 ± 22.53	39.09 ± 20.81	9.23 ± 6.18	9.28 ± 6.19	10.43 ± 6.28
MdL	17.10 ± 15.75	21.61 ± 19.48	37.08 ± 17.87	9.18 ± 5.51	9.32 ± 5.99	10.46 ± 6.15
FdL	16.50 ± 14.23	20.63 ± 18.27	35.82 ± 17.01	8.67 ± 5.59	9.11 ± 5.77	10.28 ± 5.90
W	21.41 ± 18.42	24.43 ± 23.04	40.45 ± 20.78	9.44 ± 6.19	9.71 ± 6.58	7.47 ± 1.51

Table 20: Performance of each weighting factor on estimating the loading rate and the impulse of the Ground Reaction Force. The best weighting factor is indicated in blue.

	Relative peak error [%]			Relative RMSE (vertical direction) [%]		
	8 km/h	10 km/h	12 km/h	8 km/h	10 km/h	12 km/h
LSE	2.85 ± 0.93	5.15 ± 3.50	5.47 ± 2.55	5.51 ± 1.49	6.21 ± 2.46	7.77 ± 3.31
SGD symm	2.73 ± 0.68	6.16 ± 4.38	10.77 ± 5.79	5.62 ± 1.55	6.30 ± 2.48	10.25 ± 7.16
SGD no symm	2.94 ± 1.15	6.64 ± 3.96	11.53 ± 6.04	5.56 ± 1.53	6.24 ± 2.44	14.64 ± 12.35
SGD peaks	2.75 ± 1.01	2.67 ± 0.83	5.65 ± 2.16	5.85 ± 1.70	6.75 ± 2.82	8.52 ± 4.02
BR	2.91 ± 0.99	6.75 ± 4.03	9.78 ± 5.59	5.47 ± 1.48	6.14 ± 2.44	11.06 ± 9.24
AM	9.31 ± 8.25	8.70 ± 11.45	9.84 ± 11.19	6.64 ± 2.17	7.97 ± 4.70	8.59 ± 4.55
MdL	8.63 ± 7.00	9.03 ± 10.63	9.95 ± 10.71	6.59 ± 2.07	7.88 ± 4.35	8.63 ± 4.43
FdL	8.14 ± 6.59	8.59 ± 10.20	9.54 ± 10.28	6.51 ± 2.01	7.84 ± 4.21	8.57 ± 4.34
W	11.61 ± 11.25	11.81 ± 17.10	13.04 ± 16.31	6.81 ± 2.17	8.13 ± 4.80	8.76 ± 4.70

Table 21: Performance of each weighting factor on estimating the peak and the general shape of the Ground Reaction Force (rRMSE). The best weighting factor is indicated in blue.

Tables 20 and 21 show the performance of the estimated Ground Reaction Force based on each weighting factor, for the loading rate error, the impulse error, the peak error and the relative root mean square error. The mean and the standard deviation were taken over all participants per assignment.

The weighting factors based on anthropometric measurements perform slightly worse than the weighting factors obtained with machine learning. Only for the loading rate error there is a big difference between both approaches. In that case, the difference in weighting factors can be small, but the effect on the estimated Ground Reaction Force can be large. An additional remark is that the errors increase as speed increases.

The self-written Stochastic Gradient Descent method that optimizes the fitting based on the peaks appears to do its job well. For the relative peak error, these weighting factors are the best or second-best option. However, a good fit on the peaks does not mean that the other errors are also minimised.

It is therefore difficult to decide what the 'ideal' weighting factors are. The weighting factors based on machine learning are generally the most accurate, but they have a practical drawback. They cannot be calculated if the measured GRF is not known, which is almost always the case. They could be useful if, for example, there is a database with participant data. Another possibility is that the participant first performs a short test on an instrumented treadmill when using the sensorised running and then uses them in the field.

Therefore, the results based on weighting factors from machine learning can be interpreted in this comparison as the 'most accurate' GRF estimation possible with the given accelerations. When only weighting factors based on anthropometric measurements are considered, De Leva's weighting factors based on the body dimensions of women seem to perform best, despite the fact that all participants were men.

As a final note, it is important to consider whether the most effective weighting factors will

also be the most anatomically correct.

5.3.2.1.2 Number of segments

The aim of this section is to determine which body segments contribute the most to a correct estimation of the GRF. The correlation matrix provides an initial indication of the most important segments. The correlation matrix shows the Pearson Correlation Coefficient between the vertical acceleration of each body segment measured by the IMUs, and the GRF of the left leg (left mGRFv), the right leg (right mGRFv) and the sum of both components (mGRFv).

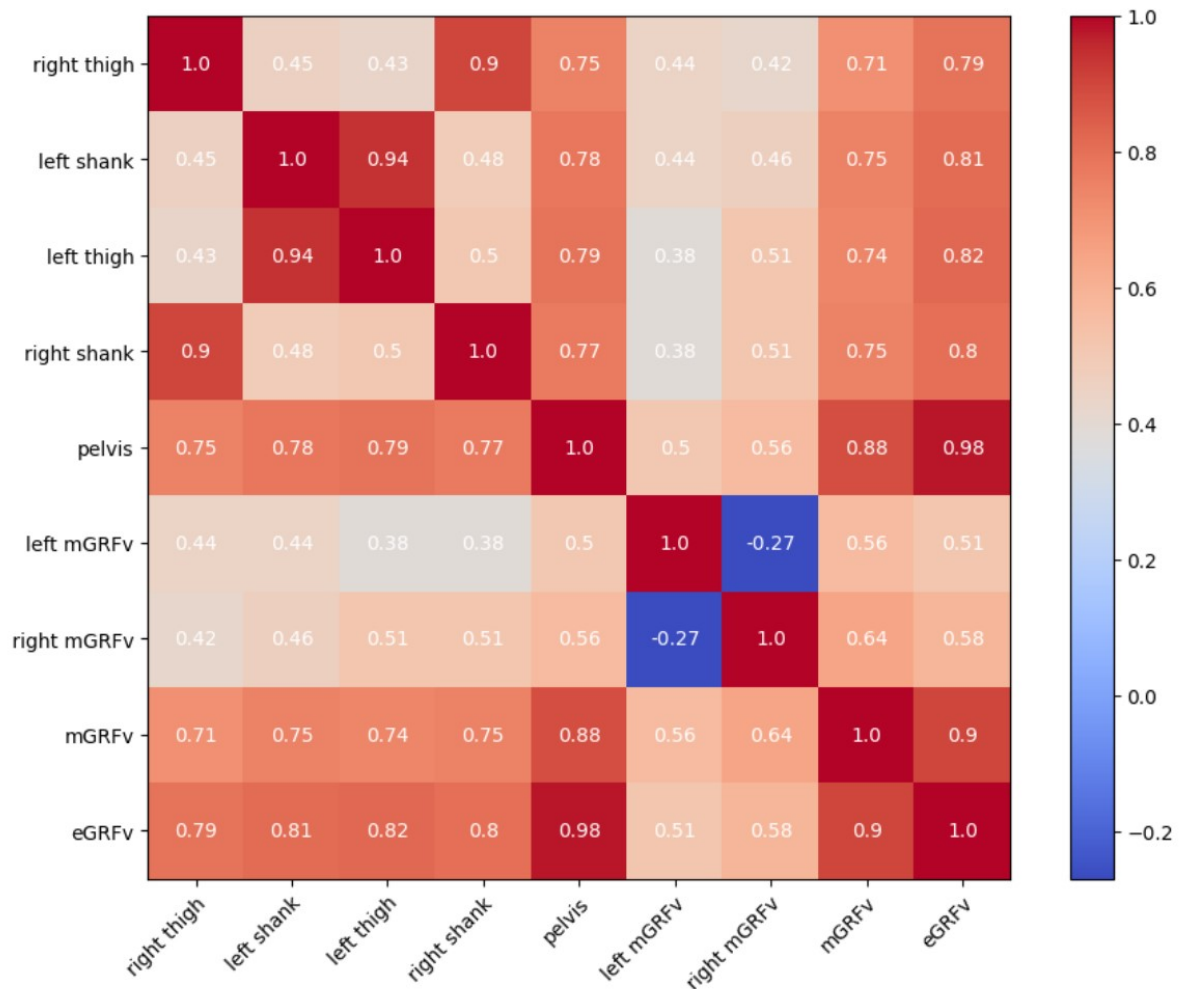


Figure 45: Correlation matrix of the accelerations of all participants for running.

As can be seen in Figure 45, the acceleration of the pelvis most closely resembles the mGRFv (0.88), followed by the shanks (left: 0.75, right: 0.75) and thighs (left: 0.74, right: 0.71). However, the pelvis is much more similar to the eGRFv (0.98), suggesting that its importance in the eGRFv may be slightly overestimated. A more unexpected result is that for the left mGRFv, the left side of the body is not necessarily the most important, with the pelvis, left shank and right thigh being the main components. The same is true for the right mGRFv, where the main components are the pelvis, right shank and left thigh. This may be because the participants

were not running with both feet perfectly on each treadmill belt.

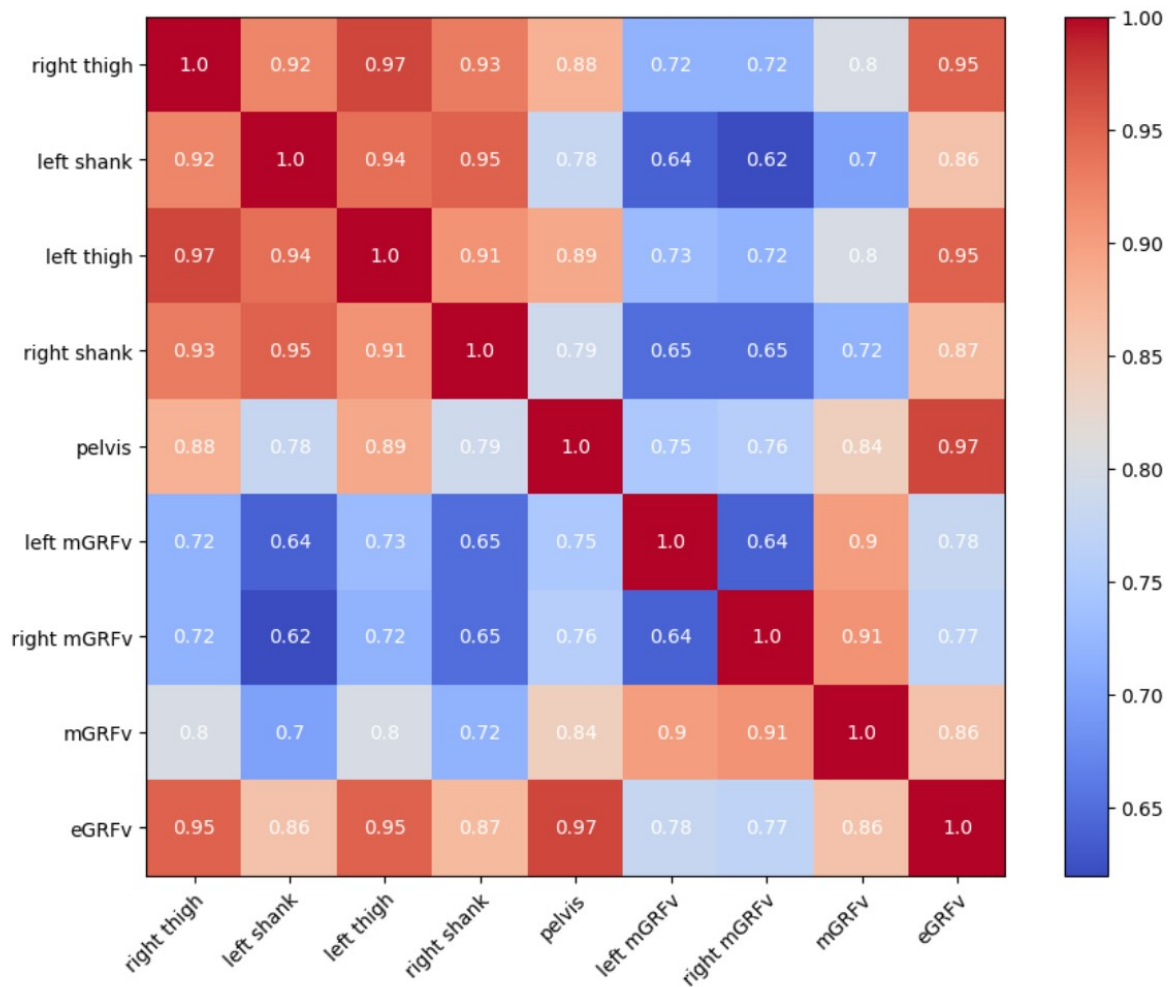


Figure 46: Correlation matrix of the accelerations of all participants for jumping.

Figure 46 shows the correlation matrix for jumping. As in running, the acceleration of the pelvis is most similar (0.84). This is logical, since the IMU on the pelvis is closest to the centre-of-mass of the body. The thighs (left: 0.8; right: 0.8) are next most similar, followed by the shanks (left: 0.7; right: 0.72). Therefore, the difference in contribution between the thighs and shanks is clearer for jumping than for running. Once again, the acceleration of the pelvis is more dominant in the eGRFv (0.97), which confirms the probably too high weighting factor.

When the results are examined in more detail by varying the number of segments in the model, it is noticeable that for the jumping task the pelvis and the thighs are the most important. Adding the shank accelerations had no added value. For the running task, the answer is more nuanced. At a pace of 8 km/h the pelvis and the thighs were also the most important, but from 10 km/h there was a transition. At that pace the thighs and the shanks became equally important to be the second component, and from 12 km/h the two most important segments were the pelvis and the shanks. Therefore, the higher the speed, the more important the

accelerations of the shanks became. This can probably be explained by the fact that the impact peak becomes more prominent at higher speeds, and that this information lies in the shank accelerations. This is confirmed in Liang’s review: of the 14 papers examined, 10 studies added a shank IMU to a pelvic IMU for higher accuracy, while 7 studies used the combination of thigh and shank [18]. An overview is given in table 22.

#	Segments in the combination	RMSE [BW]		rRMSE [%]	
		Mean	SD	Mean	SD
1	Pelvis	0.339	± 0.194	8.71	± 6.81
2	Pelvis + thighs	0.298	± 0.165	7.44	± 5.24
3	Pelvis + thighs + shanks	0.288	± 0.165	7.13	± 5.00

Table 22: Best combinations of segments with the lowest mean root mean square errors (RMSE) and relative root mean square error (rRMSE) across participants (6) and tasks (running and jumping).

This analysis was repeated with the accelerations of the Vicon markers, and the same result followed. Since a marker was also placed on the foot, it could be determined whether an IMU on the foot would be an added value to determine the GRF. However, it turned out not to be the case. The accelerations of the foot show many oscillations due to the impact with the ground, which worsens the estimation. In the study by Verheul that was cited in the literature study 3.2.1.1.1.1, the estimation of the GRF did improve by adding the feet accelerations. This result might also be achieved by selecting more appropriate weighting factors or filter frequency of the low-pass filter.

5.3.2.1.3 Principal component analysis

The plot of the explained variance per principal component shows a clear elbow at five components for running (see figure 47 and table 23). This means that the GRF is primarily described by the first five principal components. Together, they explain 98.75 % of all segmental acceleration variability across participants and running speeds.

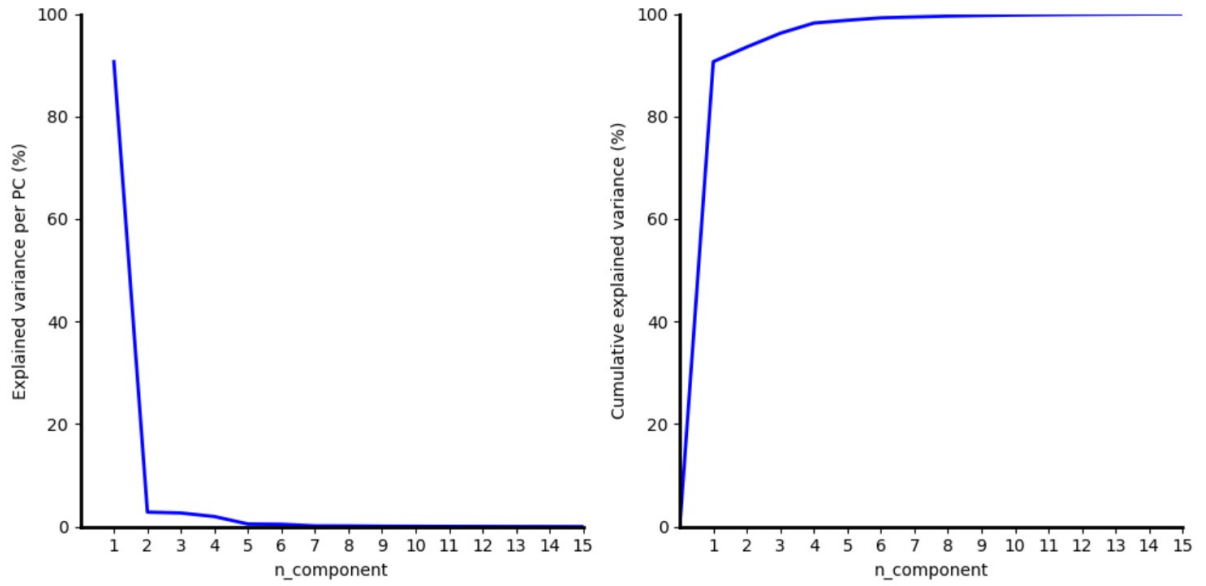


Figure 47: Explained variance of principal component analysis for all participants during running.

Principal components	1	2	3	4	5
Explained variance [%]	90.68	2.86	2.68	1.99	0.54
Cumulative explained variance [%]	90.68	93.54	96.22	98.21	98.75

Table 23: Explained variance of principal components for running.

The GRF was then reconstructed based on the principal components, and is called the principal GRF (pGRF). A visual representation of the reconstruction is shown in figures 48 and 49. The first principal component ($K=1$) only provides a rough estimation of the GRF curve. Further details are added from the second component, primarily in the medio-lateral and anterior-posterior directions. When the results are compared with the accelerations of the IMUs, it can be deduced that the first principal component is similar to the trunk acceleration. Given that the body centre-of-mass is located in the trunk, accelerations of the trunk multiplied with the body mass should provide a rough estimation of the active peak of the GRF. The smaller oscillations observed on top of the first principal component may originate from other segments, like the thigh and shank. As most impact is absorbed by the legs shortly after ground contact, they should explain the impact peak [17]. Unfortunately, even on the pGRF with five components, no impact peak can be seen. Potential explanations for this observation are that the impact is filtered out due to a too low cut-off frequency; or the impact peak is not present in these signals and additional segments are needed; or the impact peak is not detected by the PCA.

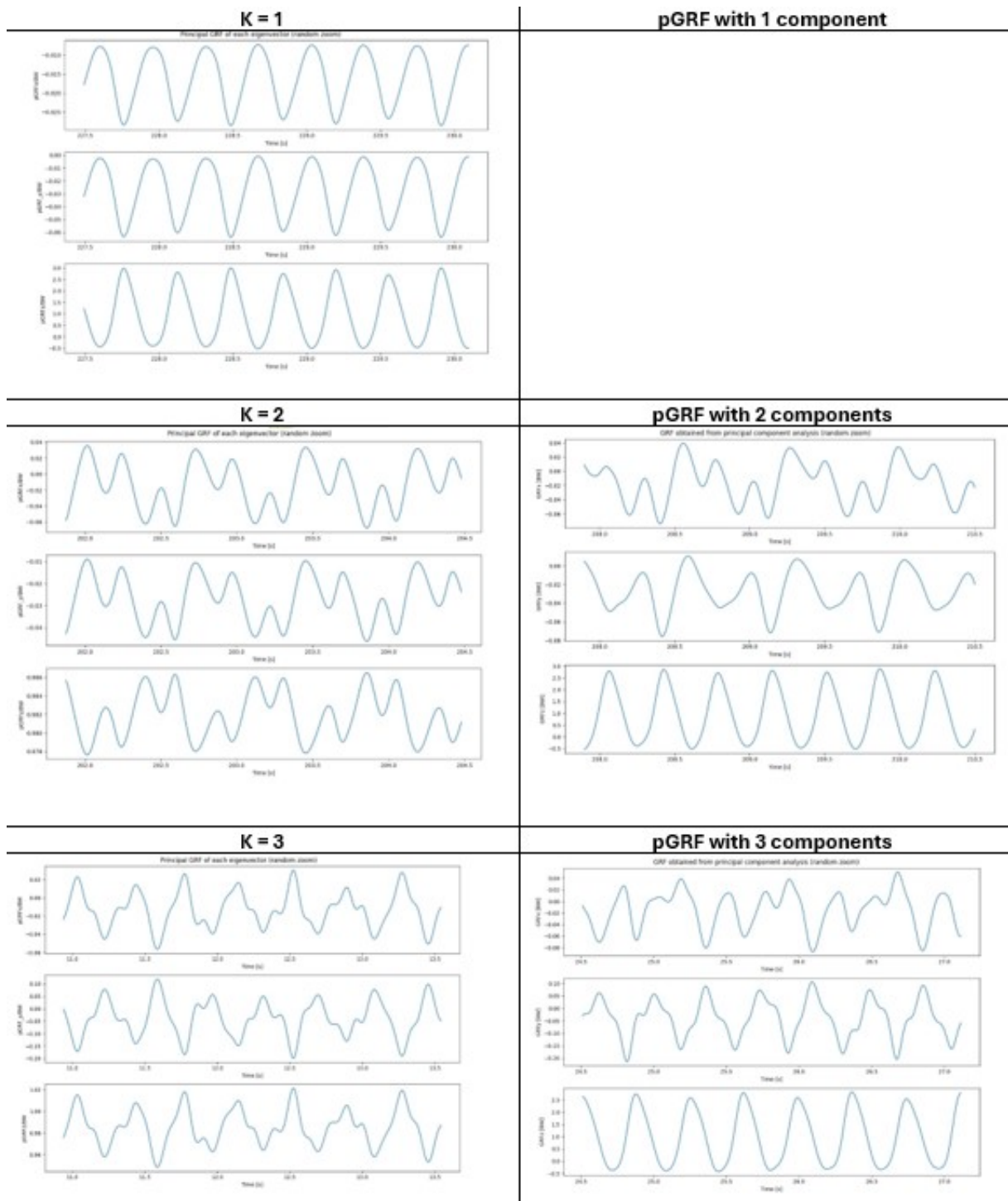


Figure 48: Representative example of the individual principal components (left) and the reconstructed Ground Reaction Force (right), from the first three principal components for running. (Second part on figure 49).

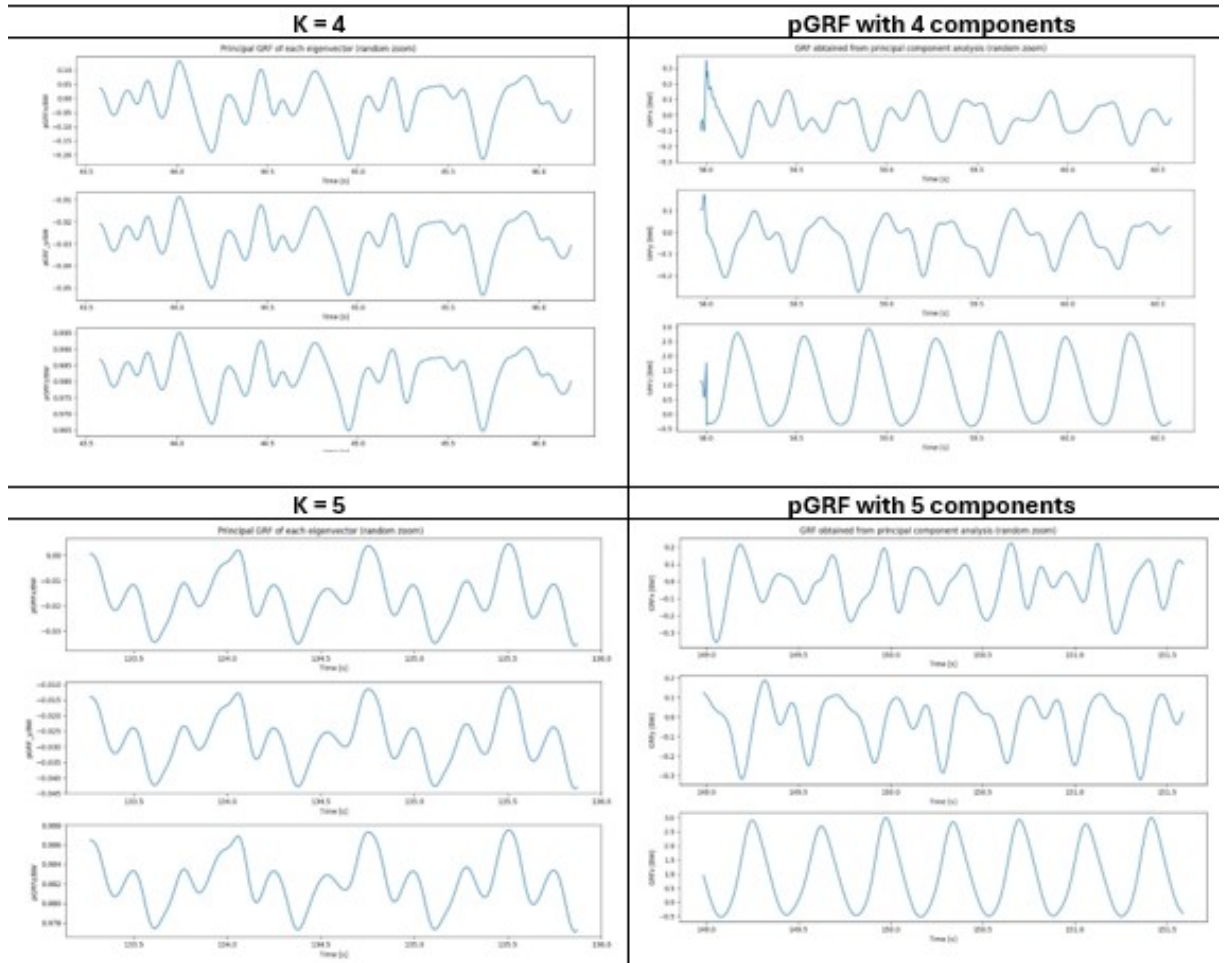


Figure 49: Representative example of the individual principal components (left) and the reconstructed Ground Reaction Force (right), from principal components three until five for running. (First part on figure 48).

When the same analysis is performed for jumping, it follows that the GRF is mainly described by the first four principal components. Taken together, these components account for 99.17% of the observed variation among the participants. The curve of the (cumulative) explained variance of the principal components is shown in figure 50. It should be noted that the choice for the number of principal components is subjective, and other possibilities exist. For example, it is also possible to consider only two components. Table 24 provides a quantitative summary of the results.

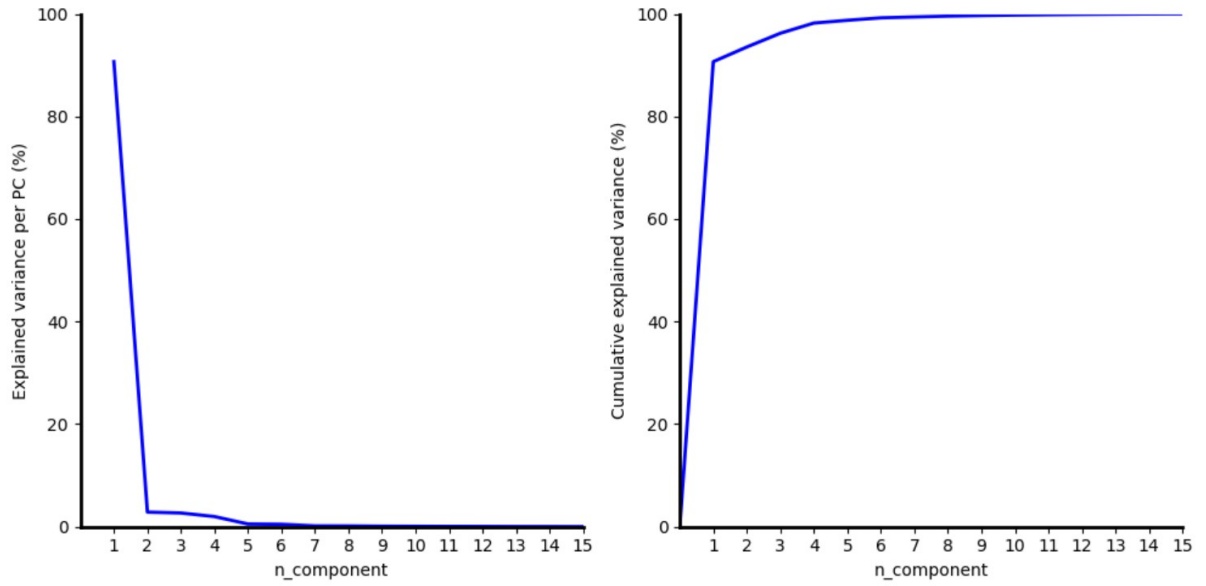


Figure 50: Explained variance of principal component analysis for all participants during jumping.

Principal components	1	2	3	4
Explained variance [%]	90.06	6.69	1.81	0.61
Cumulative explained variance [%]	90.06	96.75	98.56	99.17

Table 24: Explained variance of principal components for jumping.

When the principal components for jumping are added one by one to reconstruct the principal Ground Reaction Force, the same progression is visible as for running (see figures 51 and 52). The first principal component ($K=1$) provides an initial approximation of the shape. However, the flight phase is not yet completely correct, and the countermovement is also not yet visible. It is only possible to observe the finer details starting from the third component ($K=3$). Again, the parallel can be drawn with the analysis of the number of segments, where the overall shape of the GRF is determined by the pelvis sensor close to the CoM, and with the sensors on the thighs providing the finer details.

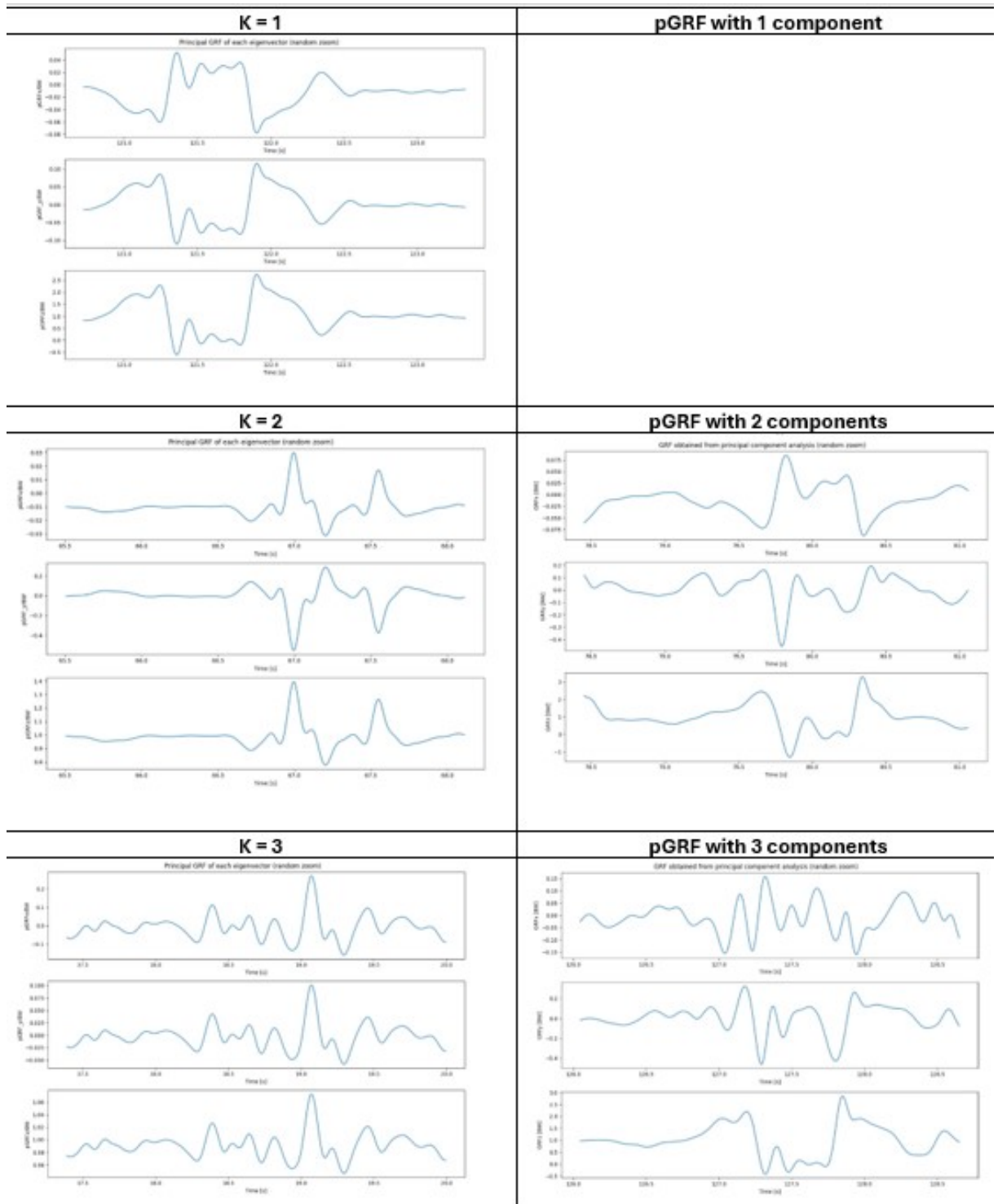


Figure 51: Representative example of the individual principal components (left) and the reconstructed Ground Reaction Force (right), from the first three principal components for jumping. (Second part on figure 52).

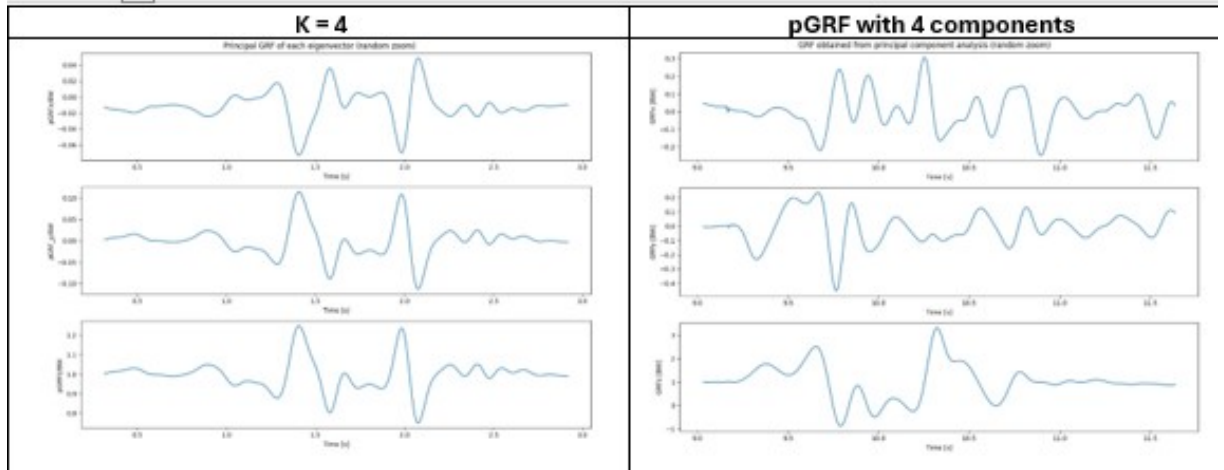
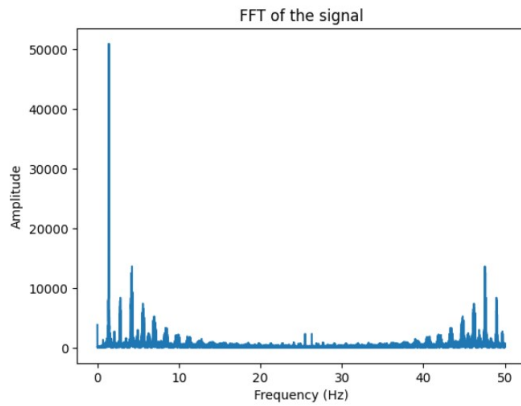


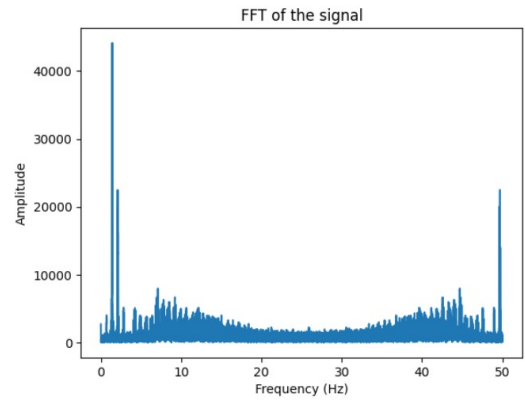
Figure 52: Representative example of the individual principal components (left) and the reconstructed Ground Reaction Force (right), from the fourth principal component for jumping. (First part on figure 51).

5.3.2.1.4 Filter cut-off frequency analysis

Based on the frequency spectrum with the Discrete Fourier Transform sampling frequencies of the vertical accelerations of the IMUs (see figure 53), it was decided to vary the frequencies in the frequency analysis between 2 Hz and 16 Hz.



(a) Representative example of the frequency spectrum of the vertical acceleration of the pelvis.



(b) Representative example of the frequency spectrum of the vertical acceleration of the thighs and shanks.

Figure 53: Representative examples of the frequency spectra of the vertical accelerations of the pelvis, thighs and shanks.

To determine the effect of the filter cut-off frequency, a visual analysis (see figures 54, 55 and 56) and a quantitative analysis based on the rRMSE (see table 25) were performed. Estimated GRF errors generally decreased for lower cut-off frequencies. However, excessive smoothing of the accelerations resulted in the loss of important GRF characteristics (3 Hz and lower).

Higher frequencies include more oscillations in the estimated GRF profiles (8 Hz and higher), resulting in a higher active peak and impact peak. From 16 Hz onwards, noise dominated the GRF. In this case, a cut-off frequency of 6.5 Hz provides the best results. Although the error is lower at a cut-off frequency of 4 Hz, the cut-off frequency was set at 6.5 Hz, because the small additional error is due to impact peak characteristics. This is consistent with the results in Karatsidis' paper, where a cut-off frequency of 6 Hz is recommended for walking movements [3]. One might expect a higher cut-off frequency for running. In his study of constant-speed running at 10 km/h to 14 km/h, Scheltinga recommended a cut-off frequency of 5.6 Hz to 6.6 Hz for the pelvis and 8.2 Hz to 8.8 Hz for the tibia [17]. A maximum speed of 12 km/h, or even 14 km/h, is probably still low enough to be in the same range of cut-off frequencies as walking.

Table 25 shows the percentage changes in the root mean square error ($RMSE$) of the medio-lateral, anterior-posterior, vertical and resultant GRF, relative to the cut-off frequency of the second-order, zero-phase shift low-pass Butterworth filter. The percentage change was calculated as follows:

$$RMSE \text{ change} = \frac{|RMSE_{ref} - RMSE|}{\frac{RMSE_{ref} + RMSE}{2}} \cdot 100 \quad (72)$$

Negative values indicate a decrease in $RMSE$, while positive values indicate an increase. The selected cut-off frequency ($RMSE_{ref}$) is used as a reference for comparison.

Frequency (Hz)	2	4	6	8	10	12	14	16
Medio-Lateral GRF	-16.22	-5.13	0.0	+9.52	+18.18	+26.09	+29.79	+36.73
Anterior-Posterior GRF	-10.53	-5.13	0.0	+4.88	+13.95	+18.18	+22.22	+26.09
Vertical GRF	48.64	-7.4	0.0	+4.65	+11.24	+15.38	+19.35	+23.16
Resultant GRF	44.25	-7.06	0.0	+6.59	+10.75	+14.74	+18.56	+22.22

Table 25: Root Mean Square Error ($RMSE$) change (%) of the estimated GRF across different frequencies.

Generally, an ideal cut-off frequency is difficult to determine, and this is also illustrated in figures 54, 55, and 56. They show representative examples of the GRF when the accelerations are filtered with a cut-off frequency of 2 Hz, 4 Hz, 6 Hz, 8 Hz, 10 Hz, 12 Hz, 14 Hz, and 16 Hz. An additional division was made at 12 Hz in the figure (12 Hz (a) and 12 Hz (b)), because this frequency had a different effect on different participants. For some participants, an impact peak appeared before the active peak. In the other participants, the active peak became much larger and an additional peak appeared after it.

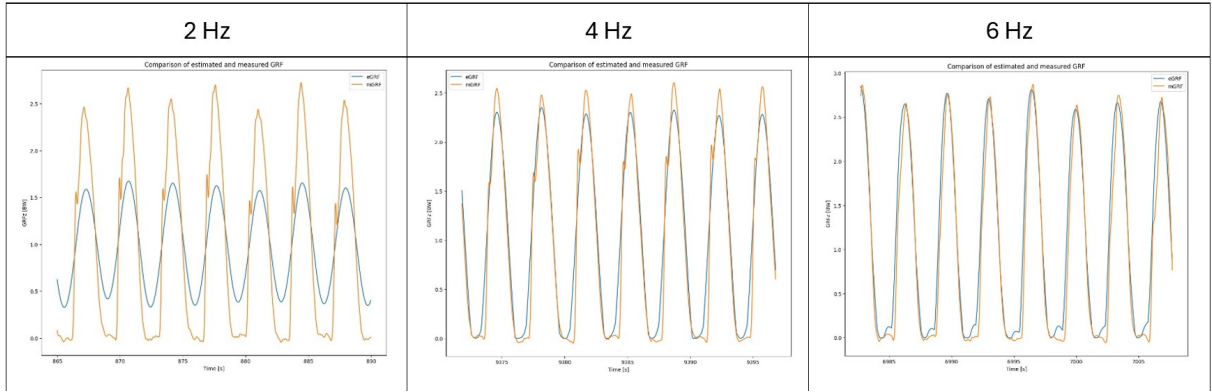


Figure 54: Representative examples of the measured vertical Ground Reaction Force (orange) and estimated vertical GRF (blue), when the cut-off frequency of the IMU filtering is 2 Hz, 4 Hz and 6 Hz.

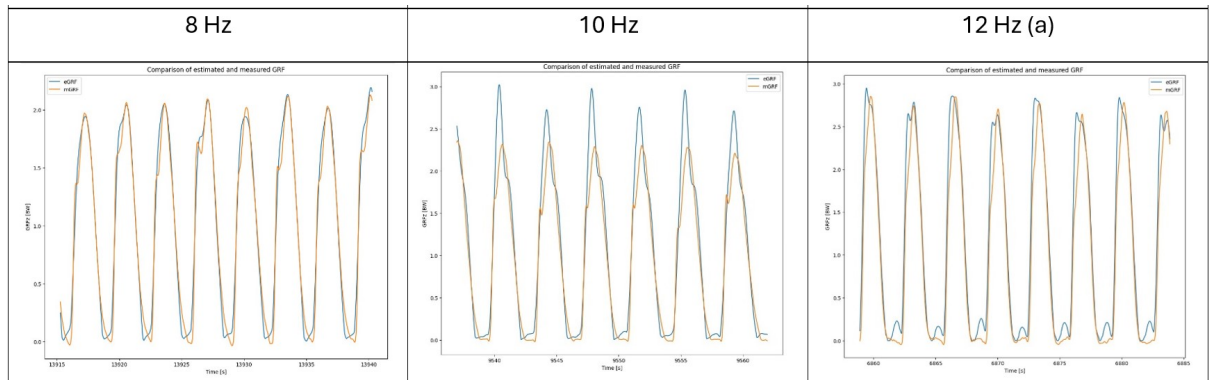


Figure 55: Representative examples of the measured vertical Ground Reaction Force (orange) and estimated vertical GRF (blue), when the cut-off frequency of the IMU filtering is 8 Hz, 10 Hz and 12 Hz.

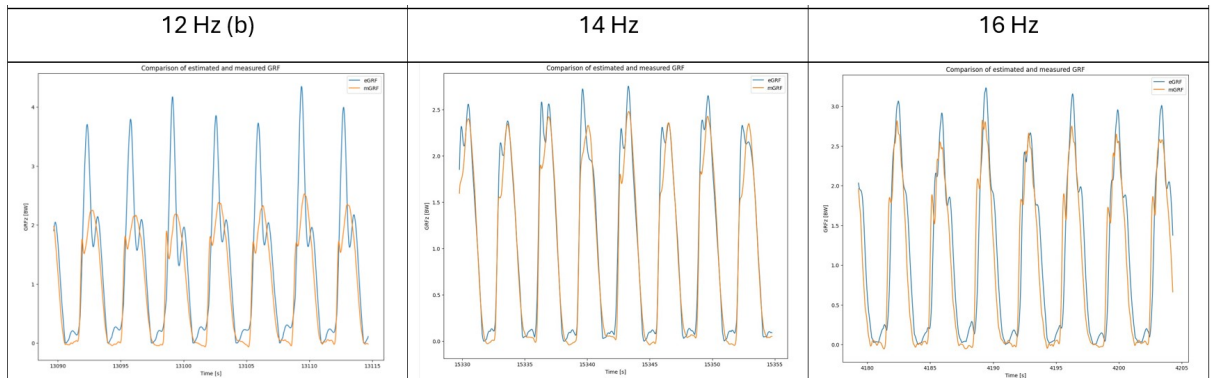


Figure 56: Representative examples of the measured vertical Ground Reaction Force (orange) and estimated vertical GRF (blue), when the cut-off frequency of the IMU filtering is 12 Hz, 14 Hz and 16 Hz.

It could be hypothesised that the estimated resultant Ground Reaction Force (GRF) profiles

would be more accurate if a combination of different cut-off frequencies was used for different inertial measurement units (IMUs). Due to limitations in computing power, this could not be attempted. However, it was investigated in a study by Verheul [8], which recommended a cut-off frequency of 10 Hz for the trunk marker and 15 Hz for the other markers. Additionally, the task can affect the optimal cut-off frequency. For example, constant speed running requires a lower cut-off frequency than accelerations and decelerations. However, from a practical point of view, achieving this with sensorised trousers for multiple movements is difficult.

5.3.2.2 Ground Reaction Moment

Figure 57 shows the mean GRM with standard deviation for all participants and speeds during the stance phase of running. The literature on the GRM during running was limited, making it difficult to conclude whether the curves are in line with the expectations. For the GRM around the medio-lateral axis (transverse plane), the shape of the eGRM and the mGRM seems the same, but an offset is visible. The mGRM around the anterior-posterior axis (sagittal plane) appears to be an average of two opposing curves based on the standard deviation, suggesting that the axis may have changed between participants. The problem could have been solved by multiplying these mirrored curves with a factor (-1). For the GRM around the vertical axis (frontal plane), the two curves agree quite well. Furthermore, the large standard deviation is particularly notable for both the estimation and the measurement of the GRM.

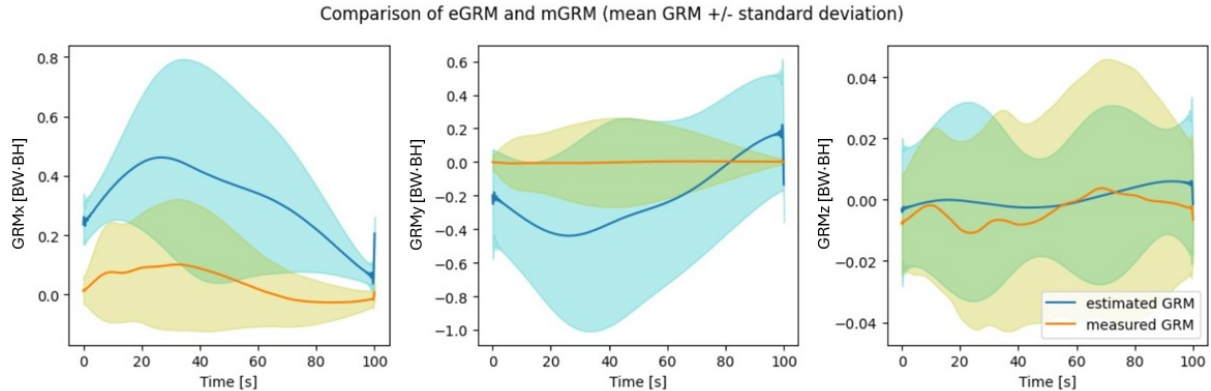


Figure 57: Mean and standard deviation of the estimated (blue) and measured Ground Reaction Moment (orange) during running for all participants.

This visual analysis was made quantitative by calculating the relative root mean square error between the mGRM and the eGRM. Violin plots were used to provide an overview of the distribution of the metrics across all participants (see figure 58), and table 26 shows a summary of the mean rRMSE and its standard deviation. Unlike the GRF, the error does not increase with higher running speeds, but remains almost constant. This could be explained by the fact that the GRM calculation relies more on the values of the angular velocities than of the linear accelerations. The error is significantly higher in comparison to the GRF. However, when the angular velocities of OpenSim and the IMUs are compared, a similar trend is observed.

When the results are compared with those of a study by Karatsidis [3], this implementation of the model has a slightly higher rRMSE value for the GRM around the medio-lateral axis ($44.90 \pm 25.21\%$, compared to $21.60 \pm 4.20\%$) and a much higher value for the anterior-posterior axis ($75.86 \pm 30.29\%$, compared to $16.10 \pm 3.20\%$). The GRM estimation around the vertical axis ($16.50 \pm 6.11\%$) is much better than that of Karatsidis (30.60 ± 8.00). The large difference in error of the GRM estimation around the anterior-posterior axis is probably due to the suspected reversed axis of the measured GRM.

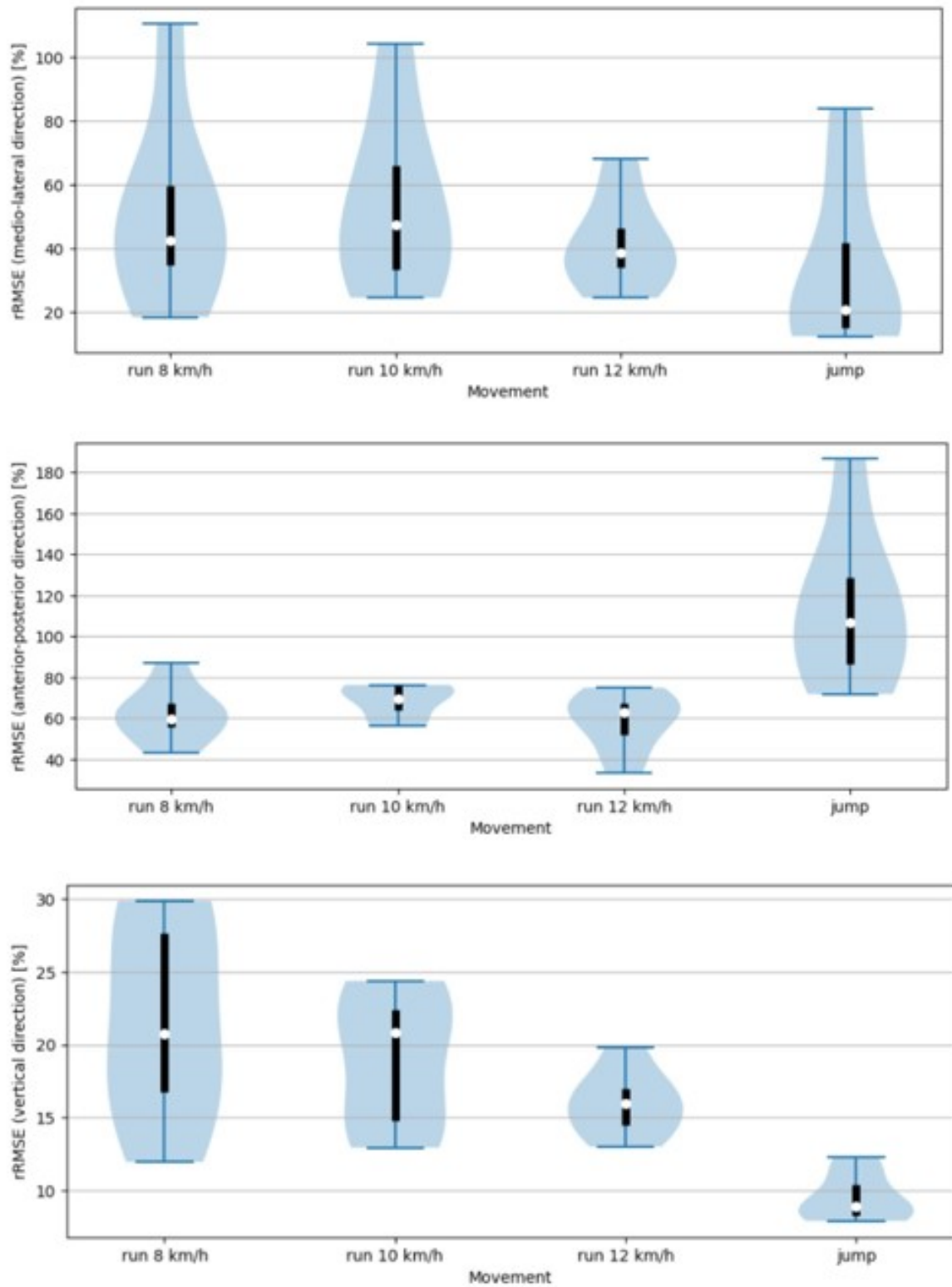


Figure 58: Violin plot of the relative root mean squared error between the estimated and measured Ground Reaction Moment, during running for all participants.

	8 km/h	10 km/h	12 km/h	Jump	All
rRMSE (medio-lateral direction)	51.48 ± 27.78	53.56 ± 25.96	41.64 ± 13.63	32.93 ± 25.28	44.90 ± 25.21
rRMSE (anterior-posterior direction)	62.26 ± 12.82	68.83 ± 7.66	58.63 ± 13.17	113.73 ± 36.17	75.86 ± 30.29
rRMSE (vertical direction)	21.64 ± 6.46	18.93 ± 4.36	15.95 ± 2.10	9.49 ± 1.54	16.50 ± 6.11

Table 26: Overview of the results of the link-segment model to estimate the Ground Reaction Moment.

5.3.2.3 Centre-of-pressure

Unfortunately, the deviation in the Ground Reaction Moment was too large to make a reliable estimate of the centre-of-pressure. To be complete, a representative example is shown in figure 59.

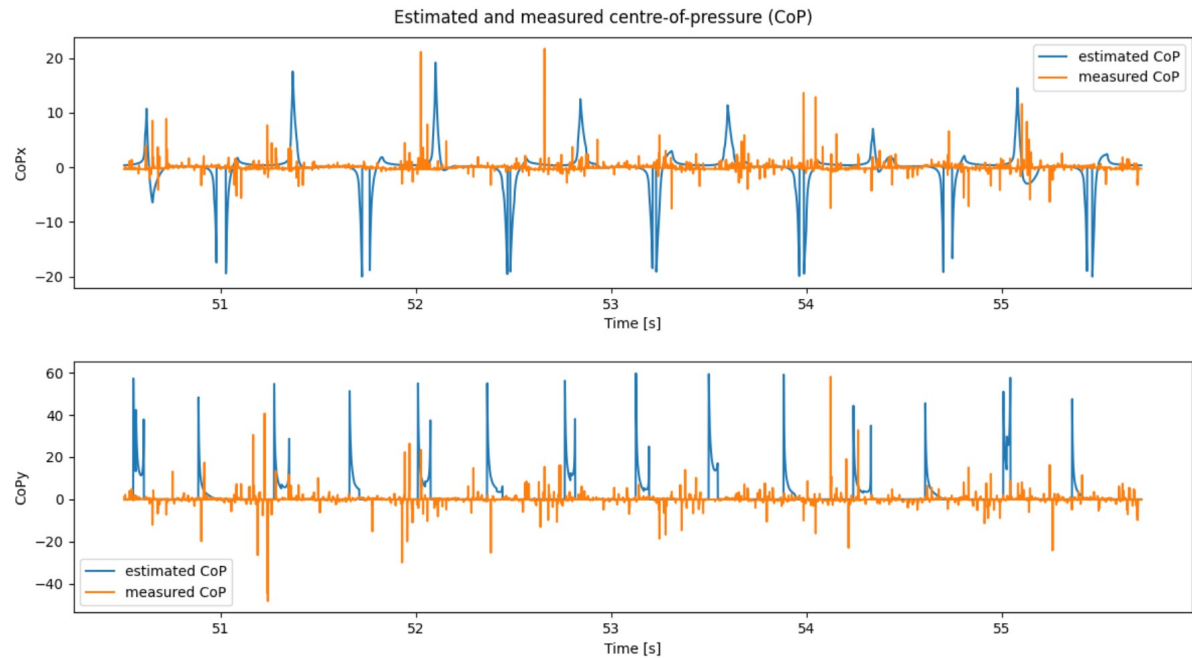


Figure 59: Representative example of the estimated centre-of-pressure (blue) and measured centre-of-pressure (orange) during running.

5.4 Two-body model

Figure 60 shows a representative example of the estimated GRF (blue), based on the two-body model, with the measured GRF (orange) indicated as a reference. An interesting advantage of this model is its ability to estimate the impact peak. However, the less accurate active peak is a disadvantage. This model can also only be used to calculate the vertical GRF for running.

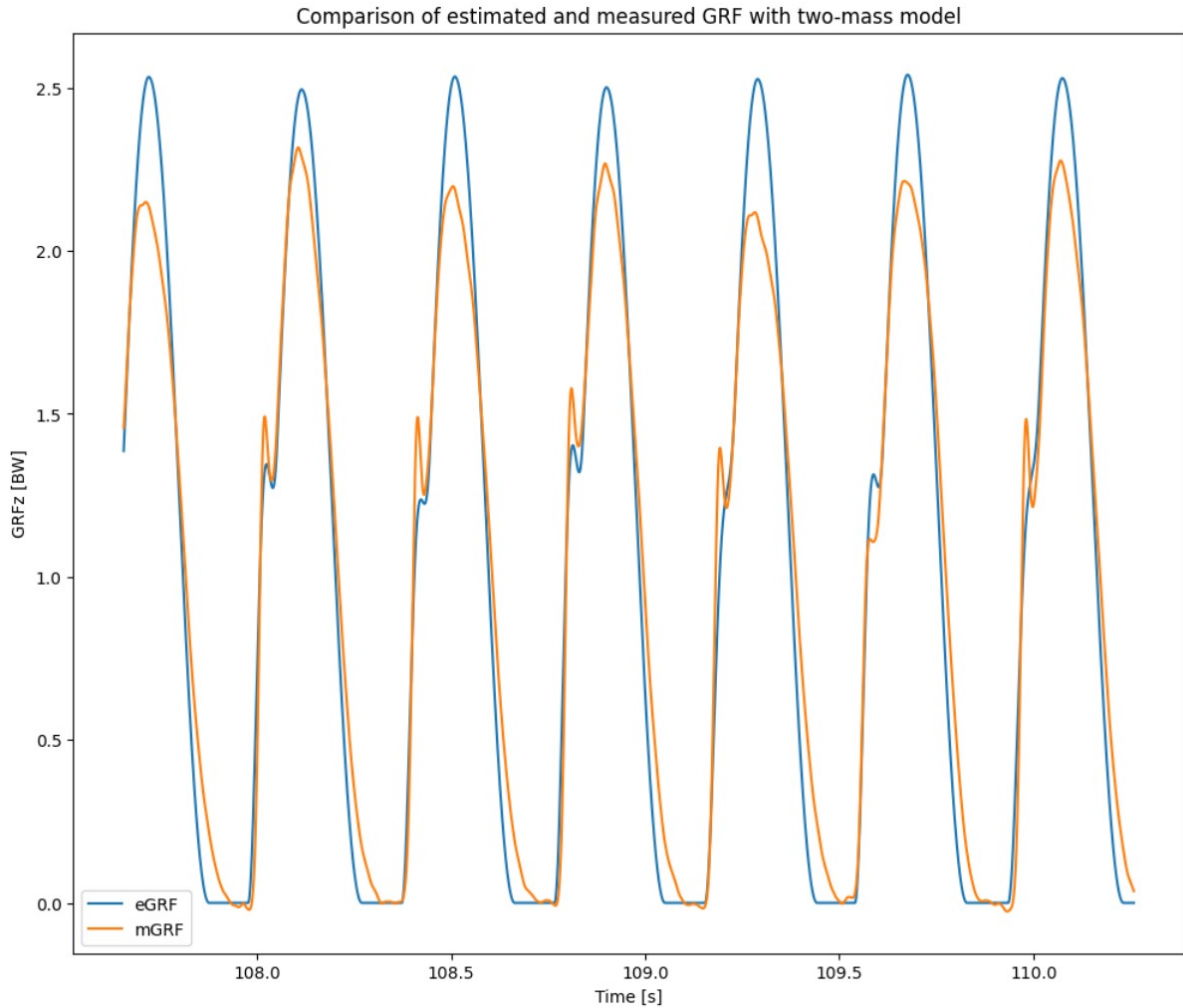


Figure 60: Representative example of the GRF estimation (blue) during running from the two-body model, and the measured GRF (orange) as reference.

The successful addition of the impact peak is also demonstrated by the metrics (see table 27 and the violin plots in figures 61 and 62). The loading rate error and the relative impulse error are smaller than for the link-segment model, due to the accurate estimation of the impact peak. The value of the loading rate is $5.59 \pm 19.33\%$ and $32.69 \pm 68.71\%$ for the two-body model and link-segment model, respectively. The relative impulse error is $6.92 \pm 5.03\%$, compared to $8.99 \pm 8.20\%$. However, the relative peak error and the rRMSE have increased due to the higher error at the active peak of the GRF, namely $15.78 \pm 14.13\%$ compared to $7.93 \pm 9.27\%$ for the relative peak error, and $13.91 \pm 5.17\%$ compared to 7.18 ± 4.87 for the rRMSE. As with the link-segment model, the error of the two-body model increases with velocity.

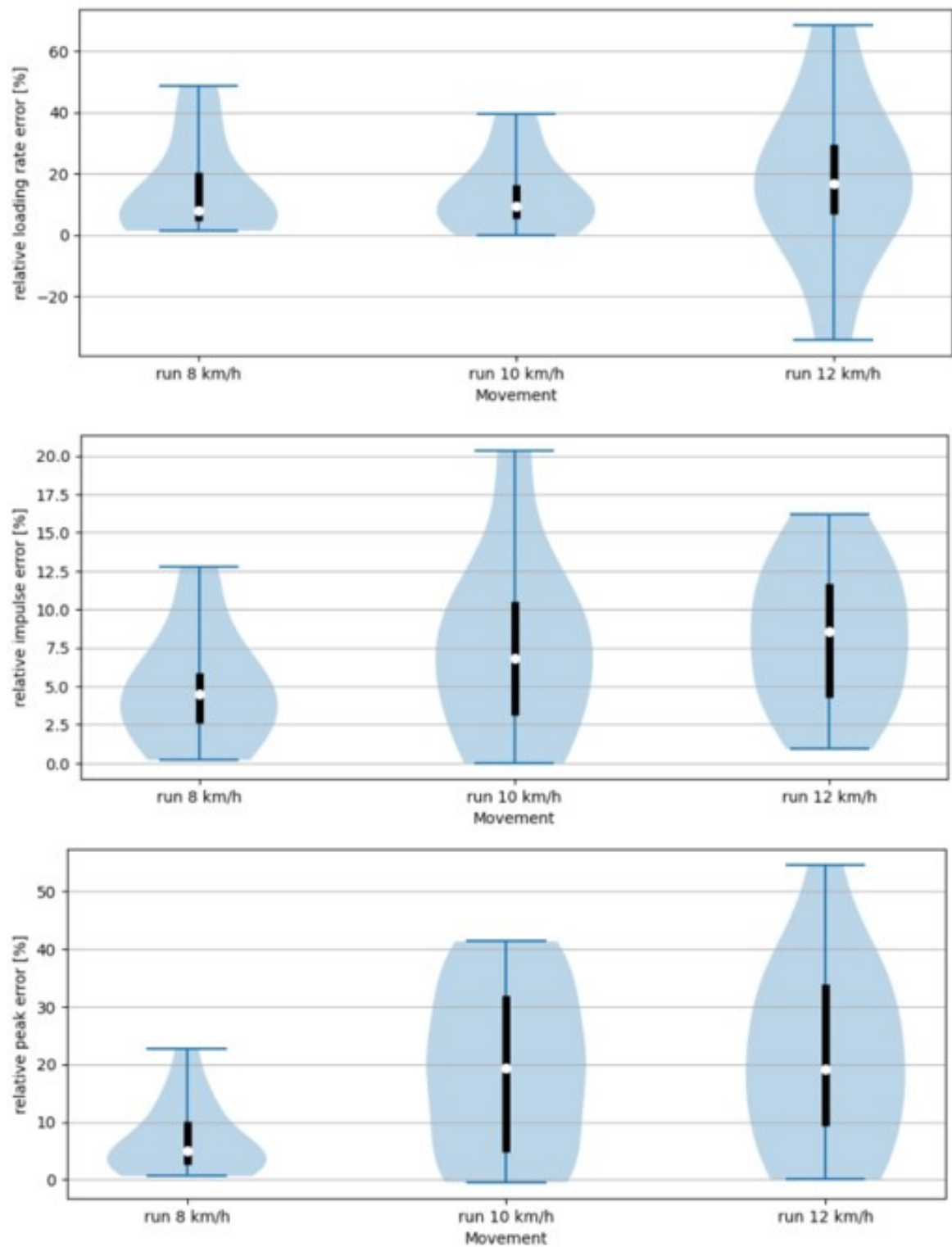


Figure 61: Violin plot of the loading rate error, impulse error and peak error of the estimated Ground Reaction Force from the two-body model, during running for all participants.

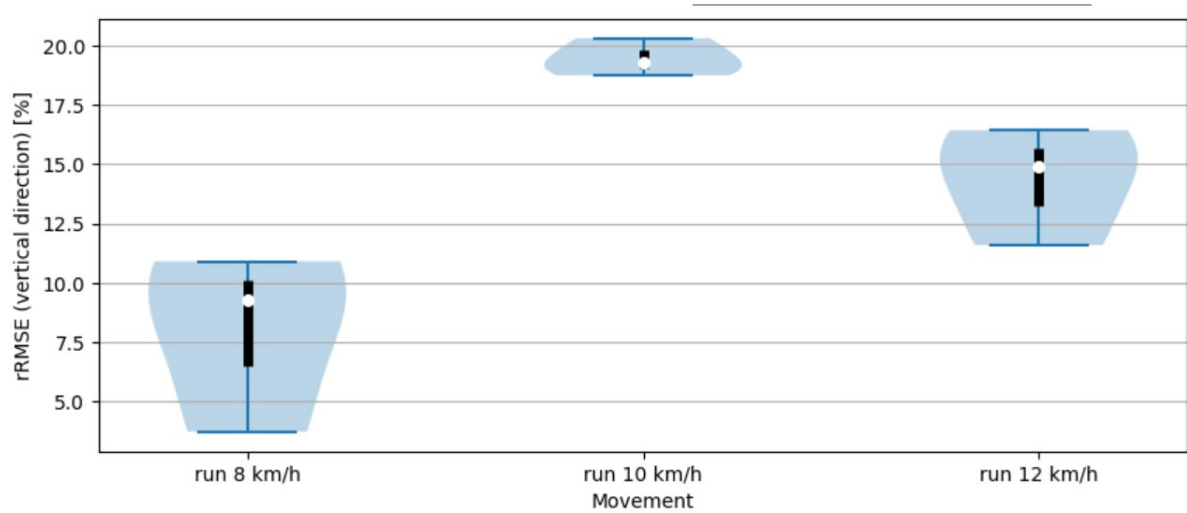


Figure 62: Violin plot of the relative root mean squared error of the estimated Ground Reaction Force from the two-body model, during running for all participants.

	8 km/h	10 km/h	12 km/h	All
Relative loading rate error [%]	13.67 ± 13.91	12.64 ± 11.17	18.05 ± 26.84	15.59 ± 19.33
Relative impulse error [%]	4.50 ± 3.44	7.54 ± 5.47	8.37 ± 4.62	6.92 ± 5.03
Relative peak error [%]	6.98 ± 6.26	19.29 ± 14.10	21.09 ± 15.60	15.78 ± 14.13
rRMSE (vertical direction) [%]	9.31 ± 3.53	17.37 ± 3.66	14.31 ± 2.02	13.91 ± 5.17

Table 27: Summary of the results of the two-body model to estimate the Ground Reaction Force.

Both the link-segment model and the two-body model have their own set of advantages and disadvantages. Consequently, the decision was taken to combine the link-segment model with the two-body model in one GRF estimation model. In that case, the first impulse (J_1) is given by the two-body model and the second impulse (J_2) by the link-segment model. As such, the two-body model should provide a reliable estimation of the impact peak, while the link-segment model should provide an accurate assessment of the active peak.

5.5 Combination of link-segment model and two-body model

Figure 63 illustrates the mean vertical eGRF with its standard deviation, using a link-segment model in combination with a two-body model (blue). The measured GRF is shown for comparison (orange). The GRF in the medio-lateral and anterior-posterior direction was estimated based on the link-segment model, because the two-body model is not defined in that direction. In the vertical direction, the combination of both models ensures that the impact peak can be

approached, and that the deviation on the active peak remains small. Therefore, the visual analysis indicates that the combination of the two models should be effective.

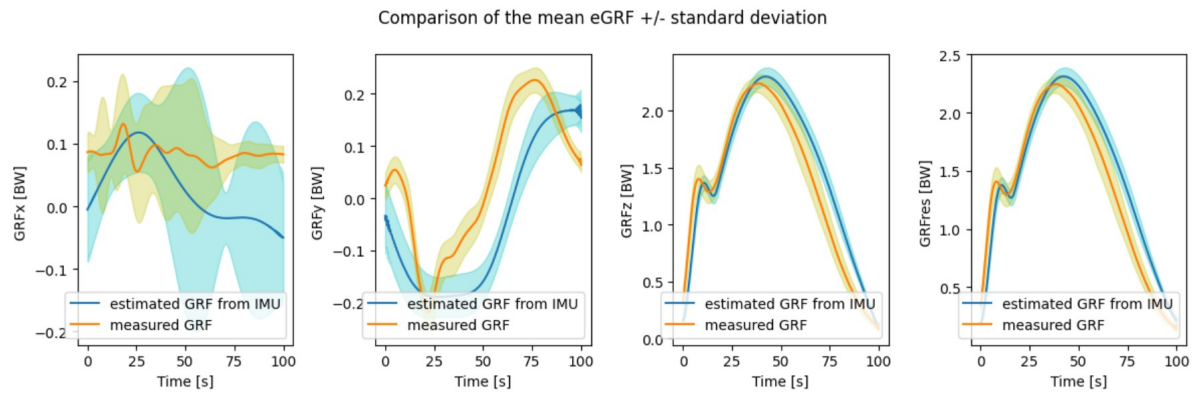


Figure 63: Mean estimated GRF with standard deviation from a link-segment model combined with a two-body model (blue), and the mean measured GRF with standard deviation (orange) as a reference.

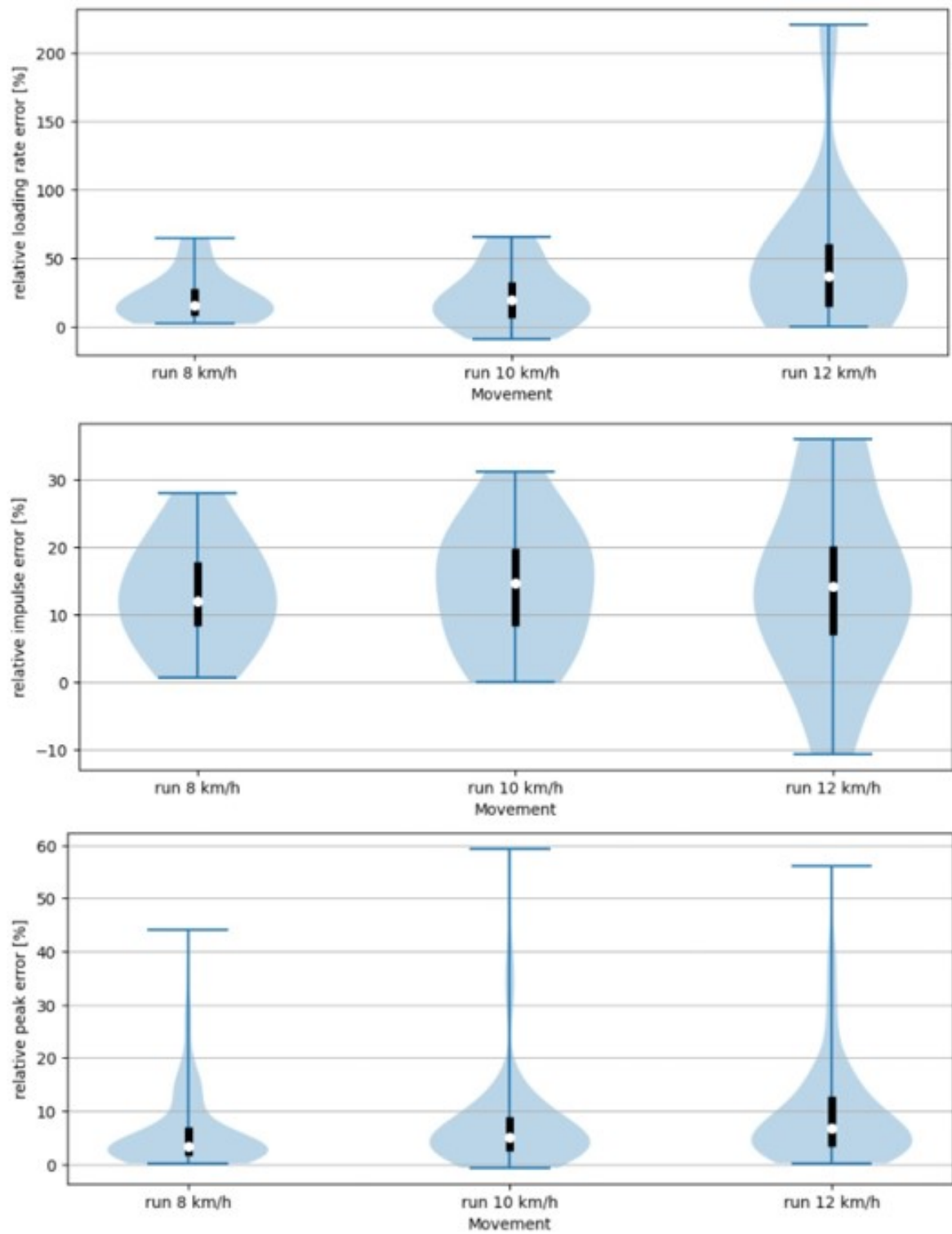


Figure 64: Violin plot of the relative loading rate error, relative impulse error and relative peak error on the estimated GRF for running of the link-segment model combined with the two-body model.

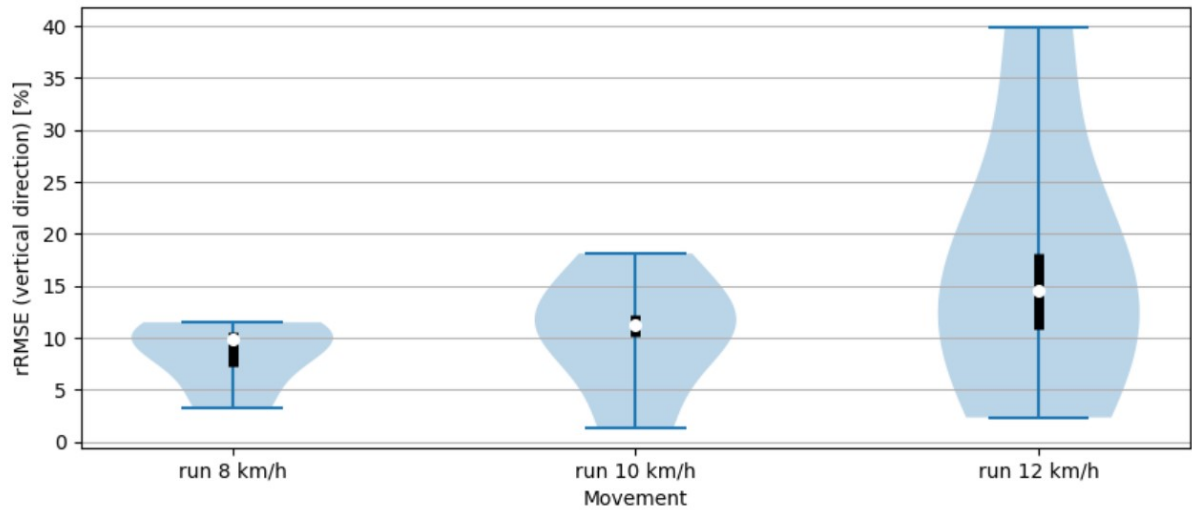


Figure 65: Violin plot of the rRMSE (vertical direction) on the estimated GRF of the link-segment model combined with the two-body model.

This combination of two models in one model can be seen in the results, as they are between the results of the two models. The distribution of the various metrics is illustrated in the violin plots on figures 64 and 65. The means and standard deviations are listed in table 28. The relative loading rate error has an average value of $30.85 \pm 39.03\%$, while for the link-segment model it was $32.69 \pm 68.71\%$ and for the two-body model $15.59 \pm 19.33\%$. This error is higher than expected, but it is probably due to impulse J_2 of the link-segment model that also influences the loading rate. The relative impulse error is found to be $13.94 \pm 10.32\%$, in comparison to $8.99 \pm 8.20\%$ for the link-segment model and $6.92 \pm 5.03\%$ for the two-mass model. The relative peak error is consistent with that of the link-segment model, with a value of $7.93 \pm 9.27\%$. Finally, the rRMSE results were on average $12.06 \pm 8.86\%$, where it was $7.18 \pm 4.87\%$ for the link-segment model and $13.91 \pm 5.17\%$ for the two-body model. The slightly higher rRMSE can be explained by adding the impact peak, but the advantages outweigh the disadvantages. The rRMSE in the medio-lateral and anterior-posterior directions remain the same as for the link-segment model, because the same model is used. Again, it should be noticed the errors increase as the velocity goes up.

	8 km/h	10 km/h	12 km/h	All
Relative loading rate error [%]	18.83 ± 17.18	19.18 ± 20.38	48.78 ± 57.39	30.85 ± 39.03
Relative impulse error [%]	11.43 ± 7.73	12.65 ± 9.06	14.56 ± 13.15	13.94 ± 10.32
Relative peak error [%]	6.26 ± 7.39	8.02 ± 10.04	9.51 ± 9.86	7.93 ± 9.27
rRMSE (vertical direction) [%]	8.45 ± 2.98	10.58 ± 5.38	17.13 ± 12.53	12.06 ± 8.86

Table 28: Summary of the results of the link-segment model combined with the two-body model to estimate the Ground Reaction Force.

5.6 Mass-spring damper model

Although most papers assume a constant natural frequency in a mass-spring-damper model, varying ω_1 allows for a quasi-perfect approximation of the acceleration of the upper mass (a_1) with the model, while a constant ω_1 does not achieve the desired results [6][9]. This idea is confirmed by the more theoretical literature: the leg is known to be stiffer during landing than take-off [5], while the muscle-tendon units produce more work during the push-off phase [10][11]. The average value of this varying ω_1 is approximately the same as the constant value. These results are illustrated by figure 66.

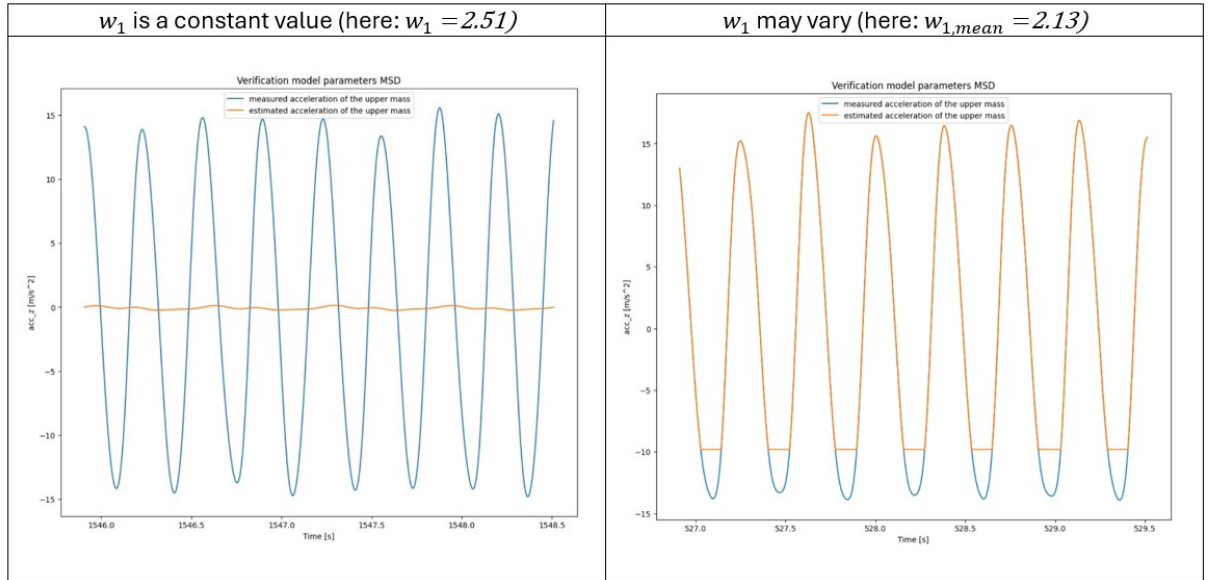


Figure 66: Approximation of the acceleration of the upper mass with the mass-spring-damper model, if the natural frequency of the upper mass (ω_1) is constant (left) or variable over time (right).

Calculating parameters ω_2 and ζ for the approximation of the acceleration of the lower mass

(a_2) was a more complex process. This is also the bottleneck of the mass-spring damper model. Firstly, it should be noted that the results of the estimation algorithm for ω_2 and ζ are very dependent on the chosen initial values. After conducting a series of trials, it was determined that $w_{2,mean}$ was approximately equal to 18, and ζ_{mean} fell within the range of -4.8 to 0.5 across all participants (see figure 67). The optimal parameters appeared to be fairly consistent across all participants, and these values are in line with those of previous studies [6].

The results also demonstrated that ω_2 is primarily responsible for the active peak, while ζ is more associated with the impact peak. If the value of ω_2 increases, the active peak and the impact peak both rise as well. Increasing the damping ratio led to a reduction in the impact peak and the time from impact peak and the time from initial contact to impact peak [5]. However, it is important to note that the model is very sensitive to small changes in these parameters. Furthermore, these parameters are challenging to estimate due to the absence of a physical interpretation.

Finally, it should be noted that the integration constant of the positions is not known. These were calculated as the distance from the ground to the IMU, using the anthropometric measurement data.

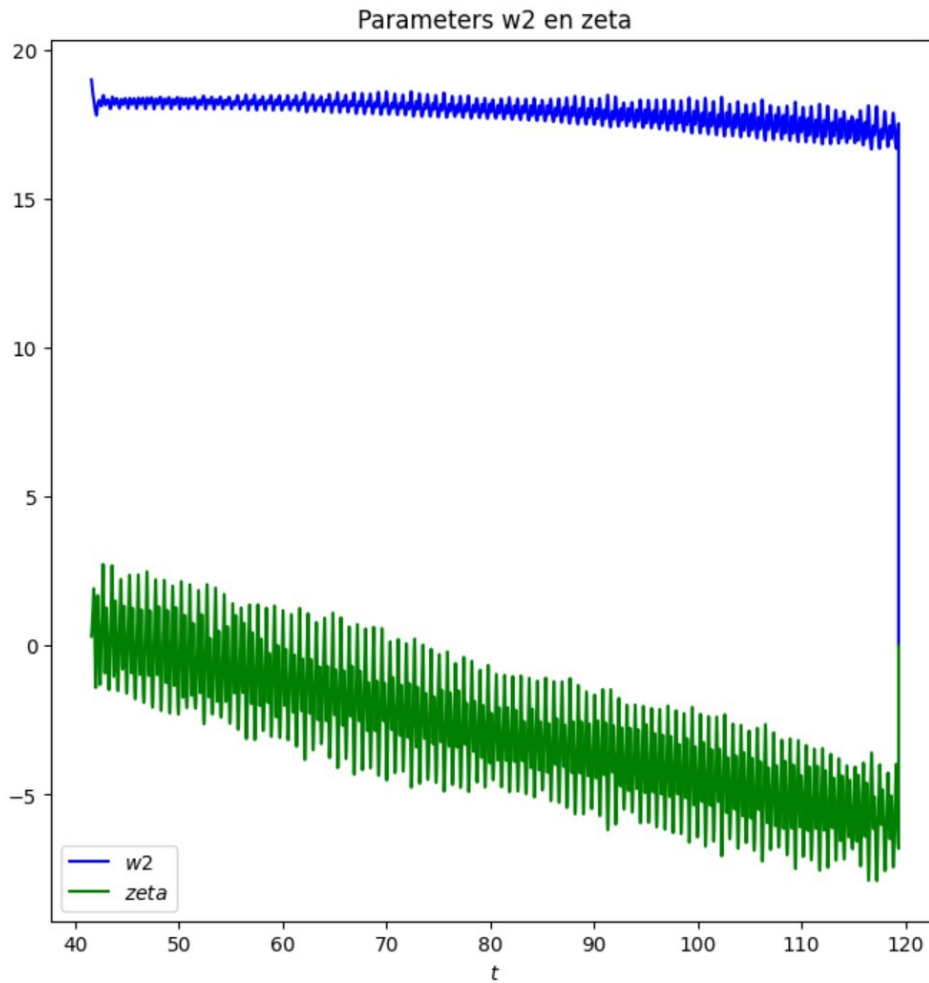


Figure 67: Representative example of the course of ω_2 and ζ over the recording, as the running speed increased from 8 km/h to 12 km/h.

Despite the error remaining relatively constant during the calculation, the mass-spring-damper model did not produce the expected results. Figure 68 shows a representative example of the estimated Ground Reaction Force of the mass-spring-damper model. Although the shape of the estimated GRF is similar to the measured GRF during stance phase, the timing is not correct. An attempt was made to refine the algorithm by stating that the estimation of ω_2 and ζ continued until the error was below a chosen threshold. However, this approach caused convergence problems. Based on this visual assessment it is concluded that the eGRF and mGRF were too different. Therefore, no metrics were calculated.

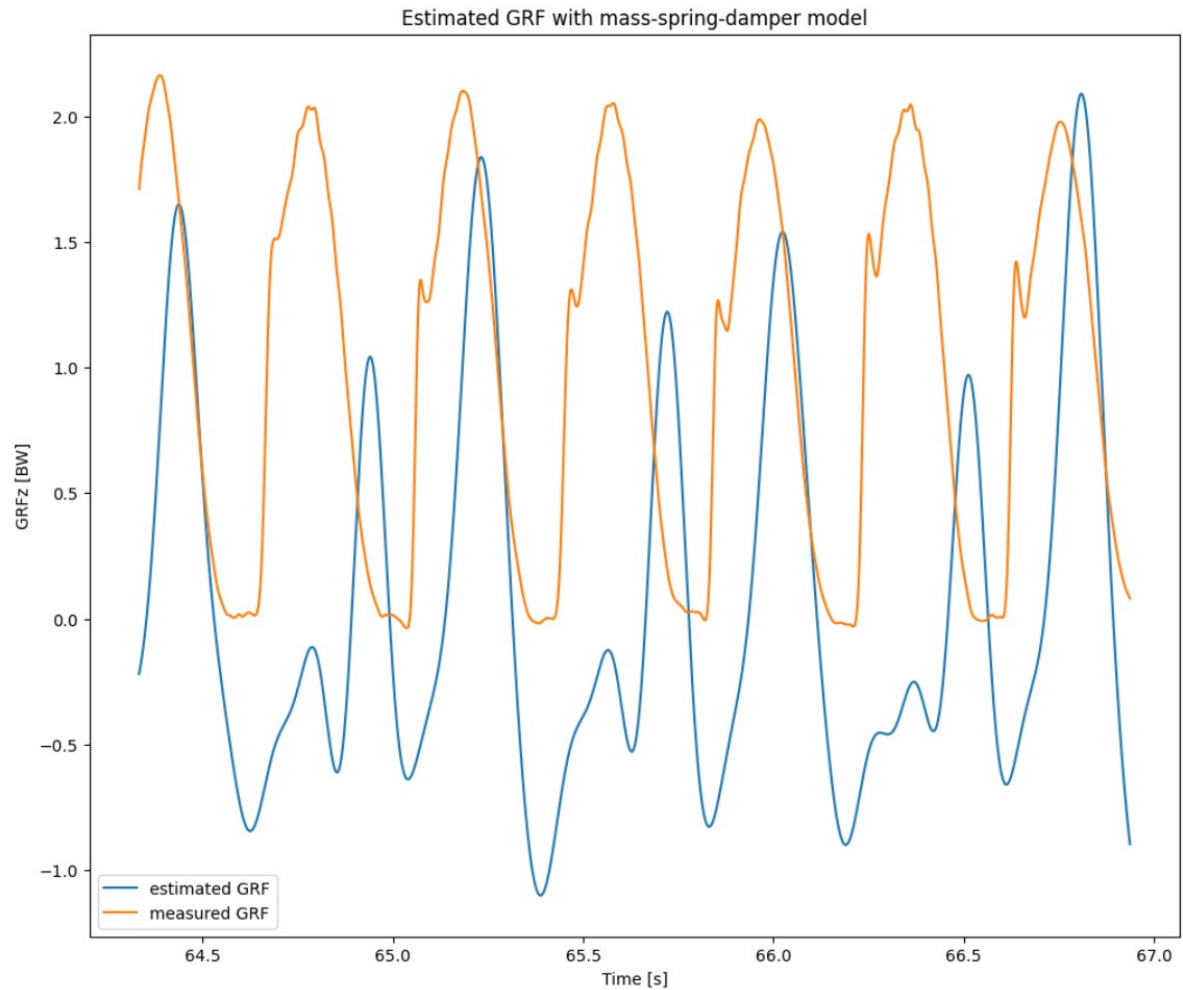


Figure 68: Representative example of the estimated GRF from the mass-spring damper model (blue), with the measured GRF as a reference (orange).

6 Discussion

6.1 General remarks

For this study, the GRF patterns for running and jumping tasks of 7 participants were examined. The results are comparable to those of similar studies [17] [8] [3]. However, the participants were only men, and therefore it was not possible to know whether the results would be different for women. For example, women have a different body shape to men, resulting in a lower centre-of-mass [41].

6.2 Remarks on the model development

To the best of the writer's knowledge, this is the first study to have created a mass-spring-damper model and link-segment model for the same IMU data to estimate the GRF. A limitation of the models is that estimated GRF errors could be due to measurement and methodological

inaccuracies. For example, the application of a step detector on the GRF recording caused an additional distortion in the calculation of metrics. To minimise the methodological error for further research, the full code is shared in Appendix B.

It should be noted that the number of wearable sensors is limited to five IMUs due to the design of the running pants. This limits the number of segments in a link-segment model, and therefore the accuracy of the prediction of the GRF. However, there is still potential for optimisation through the refinement of weighting factors and the cut-off frequencies. This thesis only explores a preliminary attempt to optimise them.

The mass-spring damper model could be further improved by estimating the parameters via other methods, for example by using deep learning. In previous studies, this approach was successful for low-intensity movements [42].

Another potential improvement is to combine models, for example via weighting factors that make a model more reliable depending on the movement. As outlined in the study by Scheltinga [17], the physical models could also be combined successfully with a neural network.

6.3 Remarks on the model validation

Besides running and jumping, it would also be interesting to look at more complex movements such as sprinting, accelerations, decelerations and side-cuts. On the instrumented treadmill that was used for the model validation, these movements can't be done for practical reasons. The utilisation of a force plate would be a viable alternative for further experiments. Previous studies have shown that the accuracy of the model will likely deteriorate [8].

Although the current research was performed on an instrumented treadmill in a lab, the GRF estimation models are not limited to a laboratory setting. Before this method can be implemented in practice, the potential impact of outdoor factors should be investigated. For example, the impact of the running surface, incline, and running shoes should be examined. Validation of the model outdoors is very challenging, but could be performed with pressure insoles.

Finally, how accurate should a model be to be clinically relevant? Despite thorough research, it was not possible to find a specific value. An rRMSE lower than 10 % was taken as an objective, because meaningful performance or injury related differences in loading characteristics can be as small as 3 to 10 percent of the GRF [8]. However, it should be noted that the rRMSE does not provide all information about the performance of the model. For instance, the error can be large at the peak of the GRF, or the loading rate can be wrong. To address this need for clinical relevant metrics, additional metrics were calculated such as the peak error and the loading rate error. However, it results in a large number of metrics, and not all metrics are reported in all studies. In some cases, the metrics may also be calculated differently. Therefore, a discussion is required about current metrics in literature to create a uniform framework to evaluate wearable-sensor based GRF estimation models. One potential question to consider is whether the asymmetry between both legs should be taken into account. Would it be advisable

to report an average of both legs, or should the legs be reported separately? The unit of the metric is also important: a relative metric in percentages allows for easier comparison between movements with a different GRF, but the original unit and magnitude of the absolute metric are lost. The question of whether an ideal metric exists is a complex one.

A final remark, while extensive research has been conducted on estimating Ground Reaction Forces and Moments, there is less written about their interpretation. This complicates the identification of the important features of the Ground Reaction Force and moment estimation.

7 Conclusion

The primary objective of this thesis was to estimate the Ground Reaction Force for running and jumping with a relative root mean square error of less than 10 %. Two models were tested, with the data of IMUs on sensorised running pants as input: a link-segment model combined with a two-body model, and a mass-spring-damper model.

It can be concluded that this objective has been achieved for the GRF estimation with a link-segment model, in combination with a two-body model. The mean rRMSE with standard deviation of the vertical GRF is 8.45 ± 2.98 %, 10.58 ± 5.38 % and 17.13 ± 12.53 % for running at 8 km/h, 10 km/h and 12 km/h, respectively. Consequently, the error increases as the speed increases, and is therefore dependent on the task. The rRMSE for jumping is 6.11 ± 4.42 % in the vertical direction. Additionally, the estimation of the GRM around the vertical axis for running had an error of 21.64 ± 6.46 % at 8 km/h, 18.93 ± 4.36 % at 10 km/h, 15.95 ± 2.10 % at 12 km/h, and 9.49 ± 1.54 % for jumping. The model can be further improved by adding segments to the model, optimising the weighting factors, and choosing a better cut-off frequency. The optimal results are achieved with weighting factors calculated using machine learning rather than anthropometric data. However, their practical implementation is not clear. Furthermore, it is important to note that the GRF estimation is improved by increasing the number of segments. The pelvis provides a rough estimation of the active peak, while the other segments provide the smaller features, such as the impact peak. The cut-off frequency for the low-pass filter of the acceleration of each segment will need to be selected individually for each segment, and is dependent on the task. The higher the intensity of the task, the higher the required frequency [8].

Unfortunately, the mass-spring damper model was not able to deliver accurate predictions of GRF waveforms. The primary bottleneck of this model is the estimation of the parameters.

In summary, this study is a first step towards measuring Ground Reaction Forces and moments in the field, with the aim of improving injury prevention and rehabilitation by better monitoring biomechanical loads on the body. The link-segment model has the potential to achieve this goal with sufficient accuracy.

8 Societal reflection

This appendix reflects on the master's thesis from a societal perspective, using the Sustainable Development Goals of the United Nations as a guideline [43]. The aim of this societal report is to discuss the social, economic and political impact of the research.

8.1 Sustainable Development Goal 3: Good Health and Well-Being

Doing sports is important for a healthy lifestyle, but according to Dutch research, one in ten sporters suffered an injury in the last three months [44]. By utilising the sensorised running pants, the training load can be more effectively monitored. For example, the ground reaction force based on the GRF estimation model can determine the load on the tibia after heavy a running training [24]. Consequently, the running pants have the potential to reduce the risk of injuries, thereby enabling sporters to engage in physical activity for a longer period. In addition, the running pants can help in the rehabilitation after an injury, for example to detect a difference in strength between both legs. Initially, the product may be for a niche audience, such as top athletes and physiotherapists, but in the long term it may become accessible to a wider audience. Maybe in a few years everyone will be training with these sensorised running pants?

8.2 Sustainable Development Goal 9: Industry, Innovation and Infrastructure

The process of transforming an innovative idea into a final product is a long and arduous journey. Firstly, it is essential to conduct thorough research on the design of the sensorised running pants. The next step is to test them, and this requires the involvement of an ethical committee. Then, the pants must be produced and marketed. In addition, investors who are interested in providing financial support for the development are sought. All of these steps require infrastructure and an environment that stimulates innovation.

8.3 Sustainable Development Goal 12: Responsible Consumption and Production

Once the design phase has been completed, the next stage is the production of the sensorised running pants. Extra attention was paid to ensure a cost-effective and sustainable solution. In order to avoid the necessity of purchasing new running pants after each use, it is important that they are washable without causing damage to the sensors.

During the test phase, efforts were made to ensure that the sometimes expensive materials were used responsibly. For instance, to test the washability, an outdated sensor was initially tested to prevent the newer sensors from being damaged.

8.4 Sustainable Development Goal 17: Partnerships for the Goals

A project of this complexity demands a diverse range of expertise. In order to create a visually appealing running pants, seamstresses are required. Furthermore, the integration of sensors must be carried out by engineers. Administrative and financial tasks must best completed, while programmers are needed to ensure clear reporting. Finally, physiotherapists are required to extract relevant information from the data. And it is likely that there are many people who have been overlooked... In conclusion, it is essential for the successful completion of this project that all parties work together.

References

- [1] J. Vanrenterghem, N. Nedergaard, M. Robinson, and B. Drust, "Training load monitoring in team sports: A novel framework separating physiological and biomechanical load-adaptation pathways," *Sports Medicine*, vol. 47, pp. 1–8, 11/2017. doi: [10.1007/s40279-017-0714-2](https://doi.org/10.1007/s40279-017-0714-2).
- [2] J. Verheul, N. J. Nedergaard, J. Vanrenterghem, and M. A. Robinson, "Measuring biomechanical loads in team sports – from lab to field," *Science and Medicine in Football*, vol. 4, no. 3, pp. 246–252, 07/2020. doi: [10.1080/24733938.2019.1709654](https://doi.org/10.1080/24733938.2019.1709654).
- [3] A. Karatsidis, G. Bellusci, H. Schepers, M. D. Zee, M. Andersen, and P. Veltink, "Estimation of ground reaction forces and moments during gait using only inertial motion capture," *Sensors*, vol. 17, no. 1, 12/2016. doi: [10.3390/s17010075](https://doi.org/10.3390/s17010075).
- [4] K. P. Clark, L. J. Ryan, and P. G. Weyand, "A general relationship links gait mechanics and running ground reaction forces," *Journal of Experimental Biology*, vol. 220, no. 2, pp. 247–258, 01/2017. doi: [10.1242/jeb.138057](https://doi.org/10.1242/jeb.138057).
- [5] R. Blickhan, "The spring-mass model for running and hopping," *Journal of Biomechanics*, vol. 22, no. 11, pp. 1217–1227, 01/1989. doi: [10.1016/0021-9290\(89\)90224-8](https://doi.org/10.1016/0021-9290(89)90224-8).
- [6] J. Verheul, N. J. Nedergaard, M. Robinson, *et al.*, "Biomechanical loading during running: Can a two mass-spring-damper model be used to evaluate ground reaction forces for high-intensity tasks?" *Sports Biomechanics*, vol. 20, no. 5, pp. 571–582, 07/2021. doi: [10.1080/14763141.2019.1584238](https://doi.org/10.1080/14763141.2019.1584238).
- [7] P. D. Leva, "Adjustments to zatsiorsky-seluyanov's segment inertia parameters," *Journal of Biomechanics*, vol. 29, no. 9, pp. 1223–1230, 09/1997. doi: [10.1016/0021-9290\(95\)00178-6](https://doi.org/10.1016/0021-9290(95)00178-6).
- [8] J. Verheul, W. Gregson, P. Lisboa, J. Vanrenterghem, and M. A. Robinson, "Whole-body biomechanical load in running-based sports: The validity of estimating ground reaction forces from segmental accelerations," *Journal of Science and Medicine in Sport*, vol. 22, no. 6, pp. 716–722, 06/2019. doi: [10.1016/j.jsams.2018.12.007](https://doi.org/10.1016/j.jsams.2018.12.007).
- [9] N. J. Nedergaard, J. Verheul, B. Drust, *et al.*, "The feasibility of predicting ground reaction forces during running from a trunk accelerometry driven mass-spring-damper model," *PeerJ*, vol. 6, 12/2018. doi: [10.7717/peerj.6105](https://doi.org/10.7717/peerj.6105).
- [10] N. Nikooyan and A. Zadpoor, "Mass-spring-damper modelling of the human body to study running and hopping – an overview," *Proceedings of the Institution of Mechanical Engineers, Part H: Journal of Engineering in Medicine*, vol. 225, no. 12, pp. 1121–1135, 12/2011. doi: [10.1177/0954411911424210](https://doi.org/10.1177/0954411911424210).
- [11] T. K. Uchida and S. L. Delp, *Biomechanics of Movement - The Science of Sports, Robotics, and Rehabilitation*. SD Books, 2021.

- [12] T. R. Derrick, G. E. Caldwell, and J. Hamill, "Modeling the stiffness characteristics of the human body while running with various stride lengths," *Journal of Applied Biomechanics*, vol. 16, no. 1, pp. 36–51, 2000. DOI: [10.1123/jab.16.1.36](https://doi.org/10.1123/jab.16.1.36).
- [13] D. A. Winter, *Biomechanics and Motor Control of Human Movement*. Wiley, 2009. DOI: [10.1002/9780470549148](https://doi.org/10.1002/9780470549148).
- [14] Independent. "Usain bolt explains how he got injury during world championships relay and defends decision to continue after rio." (2017), [Online]. Available: <https://www.independent.co.uk/sport/general/athletics/usain-bolt-injury-relay-why-how-world-athletics-championships-2017-video-will-he-race-again-rio-2016-a7892241.html>.
- [15] T. J. Gabbett, "Debunking the myths about training load, injury and performance: Empirical evidence, hot topics and recommendations for practitioners," *British Journal of Sports Medicine*, 2018. DOI: [10.1136/bjsports-2018-099784](https://doi.org/10.1136/bjsports-2018-099784).
- [16] H. Toremans-De Weert, "Benchmarking sensorised garments for emg signal monitoring: Enhancing recovery strategies," M.S. thesis, Vrije Universiteit Brussel, 2024.
- [17] B. L. Scheltinga, J. N. Kok, J. H. Buurke, and J. Reenalda, "Estimating 3d ground reaction forces in running using three inertial measurement units," *Frontiers in Sports and Active Living*, vol. 5, 05/2023. DOI: [10.3389/fspor.2023.1176466](https://doi.org/10.3389/fspor.2023.1176466).
- [18] W. Liang, F. Wang, A. Fan, W. Zhao, W. Yao, and P. Yang, "Extended application of inertial measurement units in biomechanics: From activity recognition to force estimation," *Sensors*, vol. 23, no. 9, 2023. DOI: [10.3390/s23094229](https://doi.org/10.3390/s23094229).
- [19] A. Karatsidis, "Kinetic gait analysis using inertial motion capture: New tools for knee osteoarthritis," Ph.D. dissertation, University of Twente, Enschede, The Netherlands, 2018. DOI: [10.3990/1.9789036546027](https://doi.org/10.3990/1.9789036546027).
- [20] A. Ancillao, S. Tedesco, J. Barton, and B. O'Flynn, "Indirect measurement of ground reaction forces and moments by means of wearable inertial sensors: A systematic review," *Sensors*, vol. 18, no. 8, 08/2018. DOI: [10.3390/s18082564](https://doi.org/10.3390/s18082564).
- [21] A. D. Vigotsky, K. E. Zelik, J. Lake, and R. N. Hinrichs, "Mechanical misconceptions: Have we lost the "mechanics" in "sports biomechanics"?" *Journal of Biomechanics*, vol. 93, pp. 1–5, 08/2019. DOI: [10.1016/j.jbiomech.2019.07.005](https://doi.org/10.1016/j.jbiomech.2019.07.005).
- [22] S. Krishnakumar, B. van Beijnum, C. Baten, P. Veltink, and J. Buurke, "Estimation of kinetics using imus to monitor and aid in clinical decision-making during acl rehabilitation: A systematic review," *Sensors*, vol. 24, no. 7, 01/2024. DOI: [10.3390/s24072163](https://doi.org/10.3390/s24072163).
- [23] K. Masani, M. Kouzaki, and T. Fukunaga, "Variability of ground reaction forces during treadmill walking," *J Appl Physiol*, vol. 92, no. 5, pp. 1885–1890, 05/2002. DOI: [10.1152/japplphysiol.00969.2000](https://doi.org/10.1152/japplphysiol.00969.2000).
- [24] OnTracx. "The science behind ontracx." (12/2024), [Online]. Available: <https://www.ontracx.com/science>.

- [25] A. Zadpoor and N. Nikooyan, "The effects of lower-extremity muscle fatigue on the vertical ground reaction force: A meta-analysis," *Proceedings of the Institution of Mechanical Engineers Part H Journal of Engineering in Medicine*, vol. 226, pp. 579–588, 08/2012. DOI: [10.1177/0954411912447021](https://doi.org/10.1177/0954411912447021).
- [26] L. J. R. A. B. Udofa and P. G. Weyand, "Impact forces during running: Loaded questions, sensible outcomes," *IEEE 13th International Conference on Wearable and Implantable Body Sensor Networks (BSN)*, pp. 371–376, 06/2016. DOI: [10.1109/BSN.2016.7516290](https://doi.org/10.1109/BSN.2016.7516290).
- [27] B. (F. J. M.) Koopman, "Dynamics of human movement," *THC*, vol. 18, no. 4-5, pp. 371–385, 11/2010. DOI: [10.3233/THC-2010-0599](https://doi.org/10.3233/THC-2010-0599).
- [28] M. Germanotta, I. Miletì, I. Conforti, Z. D. Prete, I. Aprile, and E. Palermo, "Estimation of human center of mass position through the inertial sensors-based methods in postural tasks: An accuracy evaluation," *JSensors*, vol. 21, no. 2, 01/2021. DOI: [10.3390/s21020601](https://doi.org/10.3390/s21020601).
- [29] B. L. Scheltinga, H. Usta, J. Reenalda, and J. H. Buurke, "Estimating vertical ground reaction force during running with 3 inertial measurement units," *Journal of Biomedical Engineering and Biosciences (JBEB)*, vol. 9, no. 1, pp. 31–38, 11/2022. DOI: [10.11159/jbeb.2022.006](https://doi.org/10.11159/jbeb.2022.006).
- [30] J. Verheul, J. Warmenhoven, P. Lisboa, W. Gregson, J. Vanrenterghem, and M. A. Robinson, "Identifying generalised segmental acceleration patterns that contribute to ground reaction force features across different running tasks," *Journal of Science and Medicine in Sport*, vol. 22, no. 12, pp. 1355–1360, 12/2019. DOI: [doi:10.1016/j.jsams.2019.07.006](https://doi.org/10.1016/j.jsams.2019.07.006).
- [31] F. d'Andrea, B. Heller, D. James, H. Koerger, and M. Dunn, "Ground reaction force estimation in football using inertial measurement units during alternate lateral bounding," *Footwear Science*, vol. 11, no. sup1, s77–s78, 06/2019. DOI: [10.1080/19424280.2019.1606087](https://doi.org/10.1080/19424280.2019.1606087).
- [32] N. Haraguchi and K. Hase, "Prediction of ground reaction forces and moments and joint kinematics and kinetics by inertial measurement units using 3d forward dynamics model," *JBSE*, vol. 19, no. 1, 2024. DOI: [10.1299/jbse.23-00130](https://doi.org/10.1299/jbse.23-00130).
- [33] M. Pogson, J. Verheul, M. A. Robinson, J. Vanrenterghem, and P. Lisboa, "A neural network method to predict task- and step-specific ground reaction force magnitudes from trunk accelerations during running activities," *Medical Engineering I& Physics*, 2020. DOI: [10.1016/j.medengphy.2020.02.002](https://doi.org/10.1016/j.medengphy.2020.02.002).
- [34] V. M. S. Limited. "Plug-in gait reference guide." (2016), [Online]. Available: <https://help.vicon.com/download/attachments/11378719/Plug-in%20Gait%20Reference%20Guide.pdf>.

- [35] J. Reenalda, M. A. Zandbergen, J. H. D. Harbers, M. R. Paquette, and C. E. Milner, "Detection of foot contact in treadmill running with inertial and optical measurement systems," *Journal of Biomechanics*, vol. 121, 05/2021. DOI: [doi:10.1016/j.jbiomech.2021.110419](https://doi.org/10.1016/j.jbiomech.2021.110419).
- [36] L. C. Benson, C. A. Clermont, R. Watari, T. Exley, and R. Ferber, "Automated accelerometer-based gait event detection during multiple running conditions," *Sensors (Basel)*, vol. 19, no. 7, 03/2019. DOI: [10.3390/s19071483](https://doi.org/10.3390/s19071483).
- [37] P. A. Federolf, "A novel approach to study human posture control: "principal movements" obtained from a principal component analysis of kinematic marker data," *Journal of Biomechanics*, vol. 49, no. 3, pp. 364–370, 02/2016. DOI: [10.1016/j.jbiomech.2015.12.030](https://doi.org/10.1016/j.jbiomech.2015.12.030).
- [38] H. Herr and M. Popovic, "Angular momentum in human walking," *Journal of Experimental Biology*, vol. 211, no. 4, pp. 476–481, 02/2008. DOI: [10.1242/jeb.008573](https://doi.org/10.1242/jeb.008573).
- [39] D. D. G. Nikolova and M. Tsveov, "New results for the mass-inertial parameters of human body based on 3d mathematical modelling," *Journal of Theoretical and Applied Mechanics*, vol. 50, 11/2020. DOI: [0.7546/JTAM.50.20.04.05](https://doi.org/0.7546/JTAM.50.20.04.05).
- [40] K. L. et al., "Investigating the effects of strapping pressure on human-robot interface dynamics using a soft robotic cuff," *IEEE Trans. Med. Robot. Bionics*, vol. 3, no. 1, pp. 146–155, 2021. DOI: [10.1109/TMRB.2020.3042255](https://doi.org/10.1109/TMRB.2020.3042255).
- [41] L. K. Smith, J. L. Lelas, and D. C. Kerrigan, "Gender differences in pelvic motions and center of mass displacement during walking: Stereotypes quantified," *Journal of Women's Health & Gender-Based Medicine*, vol. 11, no. 5, pp. 453–458, 2002. DOI: [10.1089/15246090260137626](https://doi.org/10.1089/15246090260137626).
- [42] N. J. Nedergaard, M. A. Robinson, E. Eusterwiemann, B. Drust, P. J. Lisboa, and J. Vanrenterghem, "The relationship between whole-body external loading and body-worn accelerometry during team-sport movements," *Int J Sports Physiol Perform*, vol. 12, no. 1, 01/2017. DOI: [10.1123/ijssp.2015-0712](https://doi.org/10.1123/ijssp.2015-0712).
- [43] U. Nations. "Sustainable development goals." (2015), [Online]. Available: <https://sdgs.un.org/goals>.
- [44] R. voor Volksgezondheid en Milieu Nederland. "Sportblessures." (2024), [Online]. Available: <https://www.sportenbewegenincijfers.nl/kernindicatoren/blessurerisico>.

A Appendix: Usage of AI

With the exception of the algorithms mentioned in this thesis, no artificial intelligence was used.

B Appendix: Code of Model Development

Model development: Estimation of the ground reaction force and moment

0. Load packages, constants and data of participant

```
import pandas as pd
import numpy as np
import scipy.signal
import matplotlib.pyplot as plt
import copy
from sklearn.metrics import mean_squared_error
from scipy.stats import pearsonr
import random
from scipy.spatial.transform import Rotation
import numpy.linalg
import scipy.integrate
import sklearn.linear_model
```

```
# gravitational acceleration
g = 9.81 # m/s^2
G = np.array([0, 0, g])
```

```
from google.colab import drive      # find the datasets in your Google Drive
drive.mount('/content/drive', force_remount=True)

# Participant 0
# Running dataset: Teams -> Files -> Session_341
info = '/content/drive/My Drive/Colab Notebooks/Thesis/Participant 0/metadata.txt'
IMU = '/content/drive/My Drive/Colab Notebooks/Thesis/Participant 0/IMU_Running.csv'
Vicon = '/content/drive/My Drive/Colab Notebooks/Thesis/Participant 0/Vicon_Running.csv'
# subtable "joints" starts from i=587431, and "trajectories" starts from 734299
EMG = '/content/drive/My Drive/Colab Notebooks/Thesis/Participant 0/EMG_Running.csv'
# EMG dataset is not complete
OpenSimPos = '/content/drive/My Drive/Colab Notebooks/Thesis/Participant 0/Positions_OpenSim_GlobalFrame.cs
OpenSimAccGF = '/content/drive/My Drive/Colab Notebooks/Thesis/Participant 0/Accelerations_OpenSim_GlobalFr
OpenSimAccBF = '/content/drive/My Drive/Colab Notebooks/Thesis/Participant 0/Accelerations_OpenSim_BodyFram
OpenSimVelGF = '/content/drive/My Drive/Colab Notebooks/Thesis/Participant 0/AngularVelocities_OpenSim_Glob
OpenSimVelBF = '/content/drive/My Drive/Colab Notebooks/Thesis/Participant 0/AngularVelocities_OpenSim_Body

# Participant 1
info1 = '/content/drive/My Drive/Colab Notebooks/Thesis/Participant 1/metadata1.txt'
IMU1 = '/content/drive/My Drive/Colab Notebooks/Thesis/Participant 1/IMU_Running1.csv'
Vicon1 = '/content/drive/My Drive/Colab Notebooks/Thesis/Participant 1/Vicon_Running1.csv'

# Participant 2
info2 = '/content/drive/My Drive/Colab Notebooks/Thesis/Participant 2/metadata2.txt'
IMU2 = '/content/drive/My Drive/Colab Notebooks/Thesis/Participant 2/IMU_Running2.csv'
Vicon2 = '/content/drive/My Drive/Colab Notebooks/Thesis/Participant 2/Vicon_Running2.csv'

# Participant 3
# info3 = '/content/drive/My Drive/Colab Notebooks/Thesis/Participant 3/metadata.docx'
IMU3_8 = '/content/drive/My Drive/Colab Notebooks/Thesis/Participant 3/IMU8.csv'
IMU3_10 = '/content/drive/My Drive/Colab Notebooks/Thesis/Participant 3/IMU10.csv'
IMU3_12 = '/content/drive/My Drive/Colab Notebooks/Thesis/Participant 3/IMU12.csv'
IMU3_jump = '/content/drive/My Drive/Colab Notebooks/Thesis/Participant 3/IMUjump.csv'
Vicon3_8 = '/content/drive/My Drive/Colab Notebooks/Thesis/Participant 3/Run 8Km.csv'
Vicon3_10 = '/content/drive/My Drive/Colab Notebooks/Thesis/Participant 3/Run 10Km.csv'
```

```

Vicon3_12 = '/content/drive/My Drive/Colab Notebooks/Thesis/Participant 3/Run 12Km.csv'
Vicon3_jump = '/content/drive/My Drive/Colab Notebooks/Thesis/Participant 3/Jump.csv'

# Participant 4
info4 = '/content/drive/My Drive/Colab Notebooks/Thesis/Participant 4/metadata.txt'
IMU4_8 = '/content/drive/My Drive/Colab Notebooks/Thesis/Participant 4/IMU8.csv'
IMU4_10 = '/content/drive/My Drive/Colab Notebooks/Thesis/Participant 4/IMU10.csv'
IMU4_12 = '/content/drive/My Drive/Colab Notebooks/Thesis/Participant 4/IMU12.csv'
IMU4_jump = '/content/drive/My Drive/Colab Notebooks/Thesis/Participant 4/IMUjump.csv'
Vicon4_8 = '/content/drive/My Drive/Colab Notebooks/Thesis/Participant 4/8Km.csv'
Vicon4_10 = '/content/drive/My Drive/Colab Notebooks/Thesis/Participant 4/10Km.csv'
Vicon4_12 = '/content/drive/My Drive/Colab Notebooks/Thesis/Participant 4/12Km.csv'
Vicon4_jump = '/content/drive/My Drive/Colab Notebooks/Thesis/Participant 4/MaxJump.csv'

# Participant 5
info5 = '/content/drive/My Drive/Colab Notebooks/Thesis/Participant 5/metadata.txt'
IMU5_8 = '/content/drive/My Drive/Colab Notebooks/Thesis/Participant 5/IMU8.csv'
IMU5_10 = '/content/drive/My Drive/Colab Notebooks/Thesis/Participant 5/IMU10.csv'
IMU5_12 = '/content/drive/My Drive/Colab Notebooks/Thesis/Participant 5/IMU12.csv'
IMU5_jump = '/content/drive/My Drive/Colab Notebooks/Thesis/Participant 5/IMUjump.csv'
Vicon5_8 = '/content/drive/My Drive/Colab Notebooks/Thesis/Participant 5/8Km.csv'
Vicon5_10 = '/content/drive/My Drive/Colab Notebooks/Thesis/Participant 5/10Km.csv'
Vicon5_12 = '/content/drive/My Drive/Colab Notebooks/Thesis/Participant 5/12Km.csv'
Vicon5_jump = '/content/drive/My Drive/Colab Notebooks/Thesis/Participant 5/Max Jump.csv'

# Participant 6
info6 = '/content/drive/My Drive/Colab Notebooks/Thesis/Participant 6/metadata.txt'
IMU6_8 = '/content/drive/My Drive/Colab Notebooks/Thesis/Participant 6/IMU8.csv'
IMU6_10 = '/content/drive/My Drive/Colab Notebooks/Thesis/Participant 6/IMU10.csv'
IMU6_12 = '/content/drive/My Drive/Colab Notebooks/Thesis/Participant 6/IMU12.csv'
IMU6_jump = '/content/drive/My Drive/Colab Notebooks/Thesis/Participant 6/IMUjump.csv'
Vicon6_8 = '/content/drive/My Drive/Colab Notebooks/Thesis/Participant 6/8Km.csv'
Vicon6_10 = '/content/drive/My Drive/Colab Notebooks/Thesis/Participant 6/10Km.csv'
Vicon6_12 = '/content/drive/My Drive/Colab Notebooks/Thesis/Participant 6/12Km.csv'
Vicon6_jump = '/content/drive/My Drive/Colab Notebooks/Thesis/Participant 6/Jump.csv'

# Participant 7
info7 = '/content/drive/My Drive/Colab Notebooks/Thesis/Participant 7/metadata.txt'
IMU7_8 = '/content/drive/My Drive/Colab Notebooks/Thesis/Participant 7/IMU8.csv'
IMU7_10 = '/content/drive/My Drive/Colab Notebooks/Thesis/Participant 7/IMU10.csv'
IMU7_12 = '/content/drive/My Drive/Colab Notebooks/Thesis/Participant 7/IMU12.csv'
IMU7_jump = '/content/drive/My Drive/Colab Notebooks/Thesis/Participant 7/IMUjump.csv'
Vicon7_8 = '/content/drive/My Drive/Colab Notebooks/Thesis/Participant 7/8Km.csv'
Vicon7_10 = '/content/drive/My Drive/Colab Notebooks/Thesis/Participant 7/10Km.csv'
Vicon7_12 = '/content/drive/My Drive/Colab Notebooks/Thesis/Participant 7/12Km.csv'
Vicon7_jump = '/content/drive/My Drive/Colab Notebooks/Thesis/Participant 7/Jump.csv'

```

 Mounted at /content/drive

```

def Plot(data, x_axis, x_label, y_label, nr_frames, title, names=None):

    """
    The function makes a plot of 1 variabele.
    - data: variable that you want to plot (array, with in each column a different variable), for example the
    - x_axis: variable on the x_axis, for example time
    - x_label: name of the x_axis
    - y_label: names of the y_axes, corresponding to the data in the columns
    - nr_frames: length of the plot with a close-up
    - title: title of the plot
    """

    # GRF data
    # large plot
    if len(data.shape) == 2:
        fig, axs = plt.subplots(data.shape[1], 1, figsize=(12,12)) # adjust axes

```

```

fig.suptitle(title)
random_nr = random.randint(0, data.shape[0]-nr_frames)

for i in range(data.shape[1]):
    axs[i].plot(x_axis, data[:, i])
    axs[i].set_xlabel(x_label)
    axs[i].set_ylabel(y_label[i])

fig.tight_layout()
plt.show()

# small plot
fig, axs = plt.subplots(data.shape[1], 1, figsize=(12,12)) # adjust axes
fig.suptitle(title + ' (random zoom)')

for i in range(data.shape[1]):
    axs[i].plot(x_axis[random_nr:random_nr+nr_frames], data[:, i][random_nr:random_nr+nr_frames])
    axs[i].set_xlabel(x_label)
    axs[i].set_ylabel(y_label[i])

fig.tight_layout()
plt.show()

# acceleration data
# large plot
if len(data.shape) == 3:
    for i in range(data.shape[2]):
        fig, axs = plt.subplots(1, data.shape[0], figsize=(64,12)) # adjust axes
        for j in range(data.shape[0]):
            axs[j].plot(x_axis, data[j][:, i])
            axs[j].set_xlabel(x_label)
            axs[j].set_ylabel(y_label[i])
            axs[j].set_title(names[j])
        fig.tight_layout()
        plt.show()

# small plot
random_nr = random.randint(0, data.shape[1]-nr_frames)
for i in range(data.shape[2]):
    fig, axs = plt.subplots(1, data.shape[0], figsize=(64,12)) # adjust axes
    for j in range(data.shape[0]):
        axs[j].plot(x_axis[random_nr:random_nr+nr_frames], data[j][:, i][random_nr:random_nr+nr_frames])
        axs[j].set_xlabel(x_label)
        axs[j].set_ylabel(y_label[i])
        axs[j].set_title(names[j])
    fig.tight_layout()
    plt.show()

```

```

def Plot2(data1, data2, data_label, x_axis, x_label, y_label, nr_frames, title, names=None):

```

```

    """

```

```

    The function compares 2 variables in a plot.

```

- data1: variable 1 that you want to plot (array, with in each column a different variable), for example
- data2: variable 2 that you want to plot (array, with in each column a different variable), for example
- data_label: label the data in the legend
- x_axis: variable on the x_axis, for example time
- x_label: name of the x-axis
- y_label: names of the y-axes, corresponding to the data in the columns
- nr_frames: length of the plot with a close-up
- title: title of the plot

```

    """

```

```

# acceleration data

```

```

if len(data1.shape) == 3:

```

```

    random_nr1 = random.randint(0, 4)

```

```

random_nr2 = random.randint(0, data1.shape[1]-nr_frames)

# large plot
fig, axs = plt.subplots(data1.shape[2], 1, figsize=(12,10)) # adjust axes
fig.suptitle(names[random_nr1] + ' before and after filtering')

for i in range(data1.shape[2]):
    axs[i].plot(x_axis, data1[random_nr1][:, i], label=data_label[0])
    axs[i].plot(x_axis, data2[random_nr1][:, i], label=data_label[1])
    axs[i].set_xlabel(x_label)
    axs[i].set_ylabel(y_label[i])
    axs[i].legend()

fig.tight_layout()
plt.show()

# small plot
fig, axs = plt.subplots(data1.shape[2], 1, figsize=(12,10)) # adjust axes
fig.suptitle(names[random_nr1] + ' before and after filtering')

for i in range(data1.shape[2]):
    axs[i].plot(x_axis[random_nr2:random_nr2+nr_frames], data1[random_nr1][:, i][random_nr2:random_
    axs[i].plot(x_axis[random_nr2:random_nr2+nr_frames], data2[random_nr1][:, i][random_nr2:random_
    axs[i].set_xlabel(x_label)
    axs[i].set_ylabel(y_label[i])
    axs[i].legend()

fig.tight_layout()
plt.show()

# GRF data
elif len(data1.shape) == 2:
    random_nr = random.randint(0, data1.shape[0]-nr_frames)

    # large plot
    fig, axs = plt.subplots(data1.shape[1], 1, figsize=(12,10)) # adjust axes
    fig.suptitle(title)

    for i in range(data1.shape[1]):
        axs[i].plot(x_axis, data1[:, i], label=data_label[0])
        axs[i].plot(x_axis, data2[:, i], label=data_label[1])
        axs[i].set_xlabel(x_label)
        axs[i].set_ylabel(y_label[i])
        axs[i].legend()

    fig.tight_layout()
    plt.show()

    # small plot
    fig, axs = plt.subplots(data1.shape[1], 1, figsize=(12,10)) # adjust axes
    fig.suptitle(title)

    for i in range(data1.shape[1]):
        axs[i].plot(x_axis[random_nr:random_nr+nr_frames], data1[:, i][random_nr:random_nr+nr_frames],
        axs[i].plot(x_axis[random_nr:random_nr+nr_frames], data2[:, i][random_nr:random_nr+nr_frames],
        axs[i].set_xlabel(x_label)
        axs[i].set_ylabel(y_label[i])
        axs[i].legend()

    fig.tight_layout()
    plt.show()

# one component of GRF
elif len(data1.shape)==1:
    random_nr = random.randint(0, data1.shape[0]-nr_frames)

```

```

# large plot
plt.figure(figsize=(12,10))
plt.plot(x_axis, data1, label=data_label[0])
plt.plot(x_axis, data2, label=data_label[1])
plt.xlabel(x_label)
plt.ylabel(y_label)
plt.title(title)
plt.legend()
plt.show()

# small plot
plt.figure(figsize=(12,10))
plt.plot(x_axis[random_nr:random_nr+nr_frames], data1[random_nr:random_nr+nr_frames], label=data_la
plt.plot(x_axis[random_nr:random_nr+nr_frames], data2[random_nr:random_nr+nr_frames], label=data_la
plt.xlabel(x_label)
plt.ylabel(y_label)
plt.title(title)
plt.legend()
plt.show()

```

```

def Plot3(data1, data2, data_label, x_axis, x_label, y_label, nr_frames, title, names=None):

    """
    The function compares 2 variables in a plot, and each component is shown.
    - data1: variable 1 that you want to plot (array, with in each column a different variable), for exampl
    - data2: variable 2 that you want to plot (array, with in each column a different variable), for exampl
    - data_label: label the data in the legend
    - x_axis: variable on the x_axis, for example time
    - x_label: name of the x-axis
    - y_label: names of the y-axes, corresponding to the data in the columns
    - nr_frames: length of the plot with a close-up
    - title: title of the plot
    """

    # large plot
    for i in range(data1.shape[2]):
        fig, axs = plt.subplots(1, data1.shape[0], figsize=(64,12)) # adjust axes
        for j in range(data1.shape[0]):
            axs[j].plot(x_axis, data1[j][:, i])
            axs[j].plot(x_axis, data2[:, i])
            axs[j].set_xlabel(x_label)
            axs[j].set_ylabel(y_label[i])
            axs[j].set_title(names[j])
        fig.tight_layout()
        plt.show()

    # small plot
    random_nr = random.randint(0, data1.shape[1]-nr_frames)
    for i in range(data1.shape[2]):
        fig, axs = plt.subplots(1, data1.shape[0], figsize=(64,12)) # adjust axes
        for j in range(data1.shape[0]):
            axs[j].plot(x_axis[random_nr:random_nr+nr_frames], data1[j][:, i][random_nr:random_nr+nr_frames]
            axs[j].plot(x_axis[random_nr:random_nr+nr_frames], data2[:, i][random_nr:random_nr+nr_frames])
            axs[j].set_xlabel(x_label)
            axs[j].set_ylabel(y_label[i])
            axs[j].set_title(names[j])
        fig.tight_layout()
        plt.show()

```

```

def Plot4(data1, data2, data3, data_label, x_axis1, x_axis2, x_axis3, x_label, y_label, nr_frames, title, n

    """
    The function compares 3 variables in a plot.

```

```

- data1: variable 1 that you want to plot (array, with in each column a different variable), for exampl
- data2: values of indicated points on variable 1 that you want to plot (array, with in each column a d
- data3: values indicated points on variable 1 that you want to plot (array, with in each column a diff
- data_label: label the data in the legend
- x_axis1: variable on the x_axis for data1, for example time
- x_axis2: indices of indicated points
- x_axis3: indices of indicated points
- x_label: name of the x-axis
- y_label: names of the y-axes, corresponding to the data in the columns
- nr_frames: length of the plot with a close-up
- title: title of the plot
"""

```

```

random_nr = random.randint(0, data1.shape[0]-nr_frames)

```

```

# large plot
plt.figure(figsize=(12,6))
plt.plot(x_axis1, data1, label=data_label[0])
# plt.vlines(x = x_axis2, ymin=0, ymax=np.max(data1), colors = 'orange', label = 'IC')
# plt.vlines(x = x_axis3, ymin=0, ymax=np.max(data1), colors = 'green', label = 'TO')
plt.plot(x_axis2, data2, "x", label=data_label[1])
plt.plot(x_axis3, data3, "x", label=data_label[2])
plt.xlabel(x_label)
plt.ylabel(y_label)
plt.title(title)
plt.legend()
plt.show()

# small plot
plt.figure(figsize=(12,6))
plt.plot(x_axis1[random_nr:random_nr+nr_frames], data1[random_nr:random_nr+nr_frames], label=data_label
I_x_axis2_sel = []
I_x_axis3_sel = []
for i,e in enumerate(x_axis2):
    if e in np.arange(random_nr, random_nr+nr_frames, 1):
        I_x_axis2_sel.append(i)
for i,e in enumerate(x_axis3):
    if e in np.arange(random_nr, random_nr+nr_frames, 1):
        I_x_axis3_sel.append(i)
# plt.vlines(x = I_x_axis2_sel, ymin=0, ymax=np.max(data1), colors = 'orange', label = 'IC')
# plt.vlines(x = I_x_axis3_sel, ymin=0, ymax=np.max(data1), colors = 'green', label = 'TO')
plt.plot(x_axis2[I_x_axis2_sel[0]:I_x_axis2_sel[-1]], data2[I_x_axis2_sel[0]:I_x_axis2_sel[-1]], "x", 1
plt.plot(x_axis3[I_x_axis3_sel[0]:I_x_axis3_sel[-1]], data3[I_x_axis3_sel[0]:I_x_axis3_sel[-1]], "x", 1
plt.xlabel(x_label)
plt.ylabel(y_label)
plt.title(title)
plt.legend()
plt.figure(figsize=(12,12))
plt.show()

```

```

def Plot5(data1, data2, data3, data4, data_label, x_axis1, x_axis2, x_axis3, x_axis4, x_label, y_label, nr_

"""
The function compares 3 variables in a plot.
- data1: variable 1 that you want to plot (array, with in each column a different variable), for exampl
- data2: variable 2 that you want to plot (array, with in each column a different variable), for exampl
- data3: variable 3 that you want to plot (array, with in each column a different variable), for exampl
- data3: variable 3 that you want to plot (array, with in each column a different variable), for exampl
- data_label: label the data in the legend
- x_axis1: variable on the x_axis for data1, for example time
- x_axis2: variable on the x_axis for data2, for example time
- x_axis3: variable on the x_axis for data3, for example time
- x_axis4: variable on the x_axis for data3, for example time
- x_label: name of the x-axis
- y_label: names of the y-axes, corresponding to the data in the columns

```

```

- nr_frames: length of the plot with a close-up
- title: title of the plot
"""

random_nr = random.randint(0, data1.shape[0]-nr_frames)

# large plot
plt.figure(figsize=(12,6))
plt.plot(x_axis1, data1, label=data_label[0])
plt.plot(x_axis2, data2, "x", label=data_label[1]) # "x" is the symbol that you want to use
plt.plot(x_axis3, data3, "x", label=data_label[2])
plt.plot(x_axis4, data4, label=data_label[3])
plt.xlabel(x_label)
plt.ylabel(y_label)
plt.title(title)
plt.legend()
plt.show()

# small plot
plt.figure(figsize=(12,6))
plt.plot(x_axis1[random_nr:random_nr+nr_frames], data1[random_nr:random_nr+nr_frames], label=data_label)
I_x_axis2_sel = []
I_x_axis3_sel = []
for i,e in enumerate(x_axis2):
    if e in np.arange(random_nr, random_nr+nr_frames, 1):
        I_x_axis2_sel.append(i)
for i,e in enumerate(x_axis3):
    if e in np.arange(random_nr, random_nr+nr_frames, 1):
        I_x_axis3_sel.append(i)
plt.plot(x_axis2[I_x_axis2_sel[0]:I_x_axis2_sel[-1]], data2[I_x_axis2_sel[0]:I_x_axis2_sel[-1]], "x", 1)
plt.plot(x_axis3[I_x_axis3_sel[0]:I_x_axis3_sel[-1]], data3[I_x_axis3_sel[0]:I_x_axis3_sel[-1]], "x", 1)
plt.plot(x_axis4[random_nr:random_nr+nr_frames], data4[random_nr:random_nr+nr_frames], label=data_label)
plt.xlabel(x_label)
plt.ylabel(y_label)
plt.title(title)
plt.legend()
plt.figure(figsize=(12,12))
plt.show()

```

```

def PlotPeaks(data1, peaks1, data2, peaks2, data_label, x_axis1, x_axis2, x_label, y_label, nr_frames, titl

"""
The function compares 2 variables in a plot, and the peaks of each variable are indicated.
- data1: variable 1 that you want to plot (array, with in each column a different variable), for exampl
- peaks1: peaks of variable 1
- data2: variable 2 that you want to plot (array, with in each column a different variable), for exampl
- peaks2: peaks of variable 2
- data_label: label the data in the legend
- x_axis1: variable on the x_axis for data1, for example time
- x_axis2: variable on the x_axis for data2, for example time
- x_label: name of the x-axis
- y_label: names of the y-axes, corresponding to the data in the columns
- nr_frames: length of the plot with a close-up
- title: title of the plot
"""

random_nr = random.randint(0, data1.shape[0]-nr_frames)

# large plot
plt.figure(figsize=(12,6))
plt.plot(x_axis1, data1, label=data_label[0]) # plot in this order to be consistent in the colours
plt.plot(x_axis2, data2, label=data_label[2])
plt.plot(peaks1, data1[peaks1], "x", label=data_label[1])
plt.plot(peaks2, data2[peaks2], "x", label=data_label[3])
plt.xlabel(x_label)

```

```

plt.ylabel(y_label)
plt.title(title)
plt.legend()
plt.show()

# small plot
plt.figure(figsize=(12,6))
I_peaks1_sel = []
I_peaks2_sel = []
for i,e in enumerate(peaks1):
    if e in np.arange(random_nr, random_nr+nr_frames, 1):
        I_peaks1_sel.append(i)
for i,e in enumerate(peaks2):
    if e in np.arange(random_nr, random_nr+nr_frames, 1):
        I_peaks2_sel.append(i)
plt.plot(x_axis1[random_nr:random_nr+nr_frames], data1[random_nr:random_nr+nr_frames], label=data_label)
plt.plot(x_axis2[random_nr:random_nr+nr_frames], data2[random_nr:random_nr+nr_frames], label=data_label)
plt.plot(peaks1[I_peaks1_sel[0]:I_peaks1_sel[-1]], data1[peaks1[I_peaks1_sel[0]:I_peaks1_sel[-1]]], "x")
plt.plot(peaks2[I_peaks2_sel[0]:I_peaks2_sel[-1]], data2[peaks2[I_peaks2_sel[0]:I_peaks2_sel[-1]]], "x")
plt.xlabel(x_label)
plt.ylabel(y_label)
plt.title(title)
plt.legend()
plt.figure(figsize=(12,12))
plt.show()

```

```

def Plot6(data1, data2, data_label, x_axis, x_label, y_label, nr_frames, title, names=None):

    """
    The function compares 2 variables in a plot, and each component is shown.
    - data1: variable 1 that you want to plot (array, with in each column a different variable), for exampl
    - data2: variable 2 that you want to plot (array, with in each column a different variable), for exampl
    - data_label: label the data in the legend
    - x_axis: variable on the x_axis, for example time
    - x_label: name of the x-axis
    - y_label: names of the y-axes, corresponding to the data in the columns
    - nr_frames: length of the plot with a close-up
    - title: title of the plot
    """

    # large plot
    for i in range(data1.shape[2]):
        fig, axs = plt.subplots(1, data1.shape[0], figsize=(64,12)) # adjust axes
        for j in range(data1.shape[0]):
            axs[j].plot(x_axis, data1[j][:, i])
            axs[j].plot(x_axis, data2[j][:, i])
            axs[j].set_xlabel(x_label, fontsize=16)
            axs[j].set_ylabel(y_label[i], fontsize=16)
            axs[j].set_title(names[j], fontsize=20)
        fig.tight_layout()
        plt.show()

    # small plot
    random_nr = random.randint(0, data1.shape[1]-nr_frames)
    for i in range(data1.shape[2]):
        fig, axs = plt.subplots(1, data1.shape[0], figsize=(64,12)) # adjust axes
        for j in range(data1.shape[0]):
            axs[j].plot(x_axis[random_nr:random_nr+nr_frames], data1[j][:, i][random_nr:random_nr+nr_frames])
            axs[j].plot(x_axis[random_nr:random_nr+nr_frames], data2[j][:, i][random_nr:random_nr+nr_frames])
            axs[j].set_xlabel(x_label, fontsize=16)
            axs[j].set_ylabel(y_label[i], fontsize=16)
            axs[j].set_title(names[j], fontsize=20)
        fig.tight_layout()
        plt.show()

```

```

def Plot7(data1, data2, data3, data_label, x_axis, x_label, y_label, nr_frames, title, names=None):

    """
    The function compares 3 variables in a plot.
    - data1: variable 1 that you want to plot (array, with in each column a different variable), for exampl
    - data2: variable 2 that you want to plot (array, with in each column a different variable), for exampl
    - data2: variable 2 that you want to plot (array, with in each column a different variable), for exampl
    - data_label: label the data in the legend
    - x_axis: variable on the x_axis, for example time
    - x_label: name of the x-axis
    - y_label: names of the y-axes, corresponding to the data in the columns
    - nr_frames: length of the plot with a close-up
    - title: title of the plot
    """

    # acceleration data
    if len(data1.shape) == 3:
        random_nr1 = random.randint(0, 4)
        random_nr2 = random.randint(0, data1.shape[1]-nr_frames)

        # large plot
        fig, axs = plt.subplots(data1.shape[2], 1, figsize=(12,12)) # adjust axes
        fig.suptitle(names[random_nr1] + ' before and after filtering')

        for i in range(data1.shape[2]):
            axs[i].plot(x_axis, data1[random_nr1][:, i], label=data_label[0])
            axs[i].plot(x_axis, data2[random_nr1][:, i], label=data_label[1])
            axs[i].plot(x_axis, data3[random_nr1][:, i], label=data_label[2])
            axs[i].set_xlabel(x_label)
            axs[i].set_ylabel(y_label[i])
            axs[i].legend()

        fig.tight_layout()
        plt.show()

        # small plot
        fig, axs = plt.subplots(data1.shape[2], 1, figsize=(12,12)) # adjust axes
        fig.suptitle(names[random_nr1])

        for i in range(data1.shape[2]):
            axs[i].plot(x_axis[random_nr2:random_nr2+nr_frames], data1[random_nr1][:, i][random_nr2:random_
            axs[i].plot(x_axis[random_nr2:random_nr2+nr_frames], data2[random_nr1][:, i][random_nr2:random_
            axs[i].plot(x_axis[random_nr2:random_nr2+nr_frames], data3[random_nr1][:, i][random_nr2:random_
            axs[i].set_xlabel(x_label)
            axs[i].set_ylabel(y_label[i])
            axs[i].legend()

        fig.tight_layout()
        plt.show()

    # GRF data
    elif len(data1.shape) == 2:
        random_nr = random.randint(0, data1.shape[0]-nr_frames)

        # large plot
        fig, axs = plt.subplots(data1.shape[1], 1, figsize=(12,12)) # adjust axes
        fig.suptitle(title)

        for i in range(data1.shape[1]):
            axs[i].plot(x_axis, data1[:, i], label=data_label[0])
            axs[i].plot(x_axis, data2[:, i], label=data_label[1])
            axs[i].plot(x_axis, data3[:, i], label=data_label[2])
            axs[i].set_xlabel(x_label)
            axs[i].set_ylabel(y_label[i])
            axs[i].legend()

```

```

fig.tight_layout()
plt.show()

# small plot
fig, axs = plt.subplots(data1.shape[1], 1, figsize=(12,12)) # adjust axes
fig.suptitle(title)

for i in range(data1.shape[1]):
    axs[i].plot(x_axis[random_nr:random_nr+nr_frames], data1[:, i][random_nr:random_nr+nr_frames],
    axs[i].plot(x_axis[random_nr:random_nr+nr_frames], data2[:, i][random_nr:random_nr+nr_frames],
    axs[i].plot(x_axis[random_nr:random_nr+nr_frames], data3[:, i][random_nr:random_nr+nr_frames],
    axs[i].set_xlabel(x_label)
    axs[i].set_ylabel(y_label[i])
    axs[i].legend()

fig.tight_layout()
plt.show()

# one component of GRF
elif len(data1.shape)==1:
    random_nr = random.randint(0, data1.shape[0]-nr_frames)

    # large plot
    plt.figure(figsize=(12,6))
    plt.plot(x_axis, data1, label=data_label[0])
    plt.plot(x_axis, data2, label=data_label[1])
    plt.plot(x_axis, data3, label=data_label[2])
    plt.xlabel(x_label)
    plt.ylabel(y_label)
    plt.title(title)
    plt.legend()
    plt.show()

    # small plot
    plt.figure(figsize=(12,6))
    plt.plot(x_axis[random_nr:random_nr+nr_frames], data1[random_nr:random_nr+nr_frames], label=data_la
    plt.plot(x_axis[random_nr:random_nr+nr_frames], data2[random_nr:random_nr+nr_frames], label=data_la
    plt.plot(x_axis[random_nr:random_nr+nr_frames], data3[random_nr:random_nr+nr_frames], label=data_la
    plt.xlabel(x_label)
    plt.ylabel(y_label)
    plt.title(title)
    plt.legend()
    plt.figure(figsize=(12,12))
    plt.show()

```

```

def ViolinPlot(data, x_labels, y_label):

    fig, axs = plt.subplots(nrows=1, ncols=1, figsize=(10, 4))

    axs.violinplot(data, showextrema=True, showmedians=False)
    axs.yaxis.grid(True)
    axs.set_xticks([y + 1 for y in range(len(data))], labels=x_labels)
    axs.set_xlabel('Movement')
    axs.set_ylabel(y_label)

    quartile1, medians, quartile3 = np.percentile(data, [25, 50, 75], axis=1)
    inds = np.arange(1, len(medians) + 1)
    axs.scatter(inds, medians, marker='o', color='white', s=30, zorder=3)
    axs.vlines(inds, quartile1, quartile3, color='k', linestyle='-', lw=5)
    plt.show()

```

```

# Standard body for men, according to De Leva

```

```

l_MdL = 1741 /1000 # from mm to m
m_MdL = 73

# pelvis: from head to hip bone
l_p_MdL = (203.3 + 531.9) /1000    # head + whole trunk

# thigh
l_t_MdL = 422.2 /1000

# shank (without foot): from knee joint to ankle
l_s_MdL = 440.3 /1000

# foot: from footsole to ankle joint
l_f_MdL = l_MdL - l_p_MdL - l_t_MdL - l_s_MdL
d_f_MdL = 258.1 / 1000

# shank + foot
l_sf_MdL = l_s_MdL + l_f_MdL

# summary matrix
L_MdL = [l_t_MdL, l_sf_MdL, l_p_MdL, l_MdL]
L_MdL_Vicon = [l_t_MdL, l_s_MdL, l_p_MdL, l_f_MdL, l_MdL]

```

```

# Standard body for women, according to De Leva

l_FdL = 1735 /1000 # from mm to m
m_FdL = 61.9

# pelvis: from head to hip bone
l_p_FdL = (200.2 + 529.3) /1000

# thigh
l_t_FdL = 368.5 /1000

# shank (without foot): from knee joint to ankle
l_s_FdL = 438.6 /1000

# foot: from footsole to ankle joint
l_f_FdL = l_FdL - l_p_FdL - l_t_FdL - l_s_FdL
d_f_FdL = 228.3 / 1000

# shank + foot
l_sf_FdL = l_s_FdL + l_f_FdL

# summary matrix
L_FdL = [l_t_FdL, l_sf_FdL, l_p_FdL, l_FdL]
L_FdL_Vicon = [l_t_FdL, l_s_FdL, l_p_FdL, l_f_FdL, l_FdL]

```

```

def PrintInfo(info_file):
    print('Info from', info_file, ':')
    with open(info_file, 'r') as file:
        # Read the content of the file
        file_content = file.read()
        # Print the content
        print(file_content)

```

```

PrintInfo(info)

# info of the participant
M0 = 77 # kg # see Teams: metadata.txt
l0 = 181/100 # from cm to m

ratio_l = l0/l_MdL

```

➡ Info from /content/drive/My Drive/Colab Notebooks/Thesis/Participant 0/metadata.txt :

date: 01-11-2024

time: 17:19:00

Offset (s): 477

Subject height (cm): 181

Subject weight (kg): 77

Activities:

- Walking & Running (Speeds from 1 to 10km/h)

start (s): 44

end (s): 477

Notes:

- Lazarus pants

```
PrintInfo(info1)
```

```
M1 = 63
```

```
# whole body
```

```
l1 = 177 /100 # from cm to m
```

```
# anthropometric measurements (with other participant (aka my brother)): length (longitudinal=z), width (tr
# values are approximately the same for left and right side of the body for this participant
# BUT I assume not always the case, for example after surgery?
```

```
# thigh: from hip bone to knee
```

```
l_t = 52.5 /100
```

```
w_t_upper = 18.5 /100
```

```
w_t_lower = 10.75 /100
```

```
w_t = (w_t_upper + w_t_lower)/2 # mean of measurement at top of the thigh (just below hip) and bottom of th
```

```
d_t_upper = 18.5 /100
```

```
d_t_lower = 11.75 /100
```

```
d_t = (d_t_upper + d_t_lower)/2
```

```
# shank (with foot): from knee joint to footsole
```

```
l_sf = 48 /100
```

```
w_sf_upper = 11.5 /100
```

```
w_sf_lower = 6 /100
```

```
w_sf = (w_sf_upper + w_sf_lower)/2 # mean of measurement at top of the shank (just below knee) and bottom o
```

```
d_sf_upper = 13.5 /100
```

```
d_sf_lower = 8 /100
```

```
d_sf = (d_sf_upper + d_sf_lower)/2
```

```
# foot: from footsole to ankle joint
```

```
l_f = 9.5 /100
```

```
w_f_upper = 9.5 /100
```

```
w_f_lower = 4.5 /100
```

```
w_f = (w_f_upper + w_f_lower)/2 # mean of measurement at toes and ankle
```

```
d_f_upper = 25 /100
```

```
d_f_lower = 20 /100
```

```
d_f = (d_f_upper + d_f_lower)/2 # mean of measurement at small toe and big toe
```

```
# shank (without foot): from knee joint to ankle
```

```
l_s = l_sf - l_f
```

```
w_s_upper = w_sf_upper
```

```
w_s_lower = w_sf_lower
```

```
w_s = (w_s_upper + w_s_lower)/2 # mean of measurement at top of the shank (just below knee) and bottom of t
```

```
d_s_upper = d_sf_upper
```

```
d_s_lower = d_sf_lower
```

```
d_s = (d_s_upper + d_s_lower)/2
```

```
# pelvis: from head to hip bone
```

```
l_p = l1 - l_t - l_sf
```

```
w_p = 28.5 /100
```

```
d_p = 15 /100
```

```
AM1 = np.array([[w_t, d_t, l_t], [w_sf, d_sf, l_sf], [w_p, d_p, l_p]])
AM1_Vicon = np.array([[w_t, d_t, l_t], [w_s, d_s, l_s], [w_p, d_p, l_p], [w_f, d_f, l_f]])

ratio_l1 = l1 / l_MdL
```

➡ Info from /content/drive/My Drive/Colab Notebooks/Thesis/Participant 1/metadata1.txt :

```
date: 20 february 2025
time: approximately 11:30:00
Offset (s):
Subject height (cm): 176
Subject weight (kg): 63
Activities:
- Running: 8 km/h for 1 min
  start (s): 4:35
  end (s): 5:35
- Running: 10 km/h for 1 min
  start (s): 2:00
  end (s): 3:00
- Running: 12 km/h for 1 min
  start (s): 3:18
  end (s): 4:18
```

Notes:

- During the recording, IMU2 stopped for 1s. Therefore, I advice to remove this part of the dat
- A video recording of this measurement is available on the computer in the lab (C: Users/Publi
- Participant has run on socks.
- Force plate until index 335861. Problems with data?
- Trajectories from index 419843

Anthropometric measurements:

- LegLength left (mm): 930
- KneeWidth left (mm): 105
- AnkleWidth left (mm): 75
- LegLength right (mm): 930
- KneeWidth right (mm): 105
- AnkleWidth right (mm): 75

PrintInfo(info2)

```
M2 = 66.2
l2 = 167 /100

l_ts2 = ((850 + 830)/2) /1000 # leg length
w_t2_lower = ((90 + 95.16)/2) /1000 # knee width
w_s2_lower = ((65 + 62.5)/2) /1000 # ankle width

ratio_l2 = l2 / l_MdL
```

➡ Info from /content/drive/My Drive/Colab Notebooks/Thesis/Participant 2/metadata2.txt :

```
date: 20 february 2025
time: approximately 16:00:00
Offset (s):
Subject height (cm): 167
Subject weight (kg): 66.2
Activities in Vicon time:
- Running: 8 km/h for 1 min
  start (s): 56"
  end (s): 1:56
- Running: 10 km/h for 1 min
  start (s): 2:06
  end (s): 3:06
- Running: 12 km/h for 1 min
  start (s): 3:18
  end (s): 4:18
```

Notes:

- Participant has run on socks

- Force plate data until index 278 069 of the csv-file
- Trajectories from index 347601

Anthropometric measurements (according to Vicon guidelines):

- LegLength left (mm): 850
- KneeWidth left (mm): 90
- AnkleWidth left (mm): 65
- LegLength right (mm): 830
- KneeWidth right (mm): 95.16
- AnkleWidth right (mm): 62.6

```
#PrintInfo(info3)    # Note: more information available in the bioinformation file
```

```
M3 = 64
```

```
l3 = 168 /100
```

```
ratio_l3 = l3 / l_MdL
```

```
#PrintInfo(info4)    # Note: more information available in the bioinformation file
```

```
M4 = 79.45
```

```
l4 = 187 /100
```

```
ratio_l4 = l4 / l_MdL
```

```
# PrintInfo(info5)    # Note: more information available in the bioinformation file
```

```
M5 = 75
```

```
l5 = 189 /100
```

```
ratio_l5 = l5 / l_MdL
```

```
# PrintInfo(info6)    # Note: more information available in the bioinformation file
```

```
M6 = 68
```

```
l6 = 177 /100
```

```
ratio_l6 = l6 / l_MdL
```

```
# PrintInfo(info7)    # Note: more information available in the bioinformation file
```

```
M7 = 62
```

```
l7 = 168 /100
```

```
ratio_l7 = l7 / l_MdL
```

✓ 1. Scale anthropometry

✓ 1.1 Measurement

```
# Participant 3
```

```
l_leg_left = 820 # mm
```

```
l_leg_right = 820
```

```
d_knee_left = 340/np.pi
```

```
d_knee_right = 340/np.pi
```

```
d_ankle_left = 287/np.pi
```

```
d_ankle_right = 287/np.pi
```

```
L3 = np.array([(d_knee_left + d_knee_right)/2, l_leg_left]) /1000 #[m]
```

```
# L = [diameter shank, total leg length]

# Participant 4
l_leg_left = 990
l_leg_right = 990
d_knee_left = 361/np.pi
d_knee_right = 361/np.pi
d_ankle_left = 264/np.pi
d_ankle_right = 264/np.pi
L4 = np.array([(d_knee_left + d_knee_right)/2, l_leg_left]) /1000

# Participant 5
l_leg_left = 930
l_leg_right = 930
d_knee_left = 375/np.pi
d_knee_right = 375/np.pi
d_ankle_left = 220/np.pi
d_ankle_right = 220/np.pi
L5 = np.array([(d_knee_left + d_knee_right)/2, l_leg_left]) /1000

# Participant 6
l_leg_left = 890
l_leg_right = 890
d_knee_left = 310/np.pi
d_knee_right = 310/np.pi
d_ankle_left = 220/np.pi
d_ankle_right = 220/np.pi
L6 = np.array([(d_knee_left + d_knee_right)/2, l_leg_left]) /1000

# Participant 7
l_leg_left = 870
l_leg_right = 870
d_knee_left = 348/np.pi
d_knee_right = 348/np.pi
d_ankle_left = 221/np.pi
d_ankle_right = 221/np.pi
L7 = np.array([(d_knee_left + d_knee_right)/2, l_leg_left]) /1000
```

✓ 1.2 Method 1

The mass of each segment is linearly scaled based on the total body mass and the segment mass ratio values reported by Winter. The inertial parameters are calculated by considering the segments as cylinders with uniform density. In addition, geometric scaling of each segment, where the longitudinal axis is defined as the second entry, is achieved using the following matrix:

$$S = \begin{bmatrix} \sqrt{\frac{m_s}{l_s}} & 0 & 0 \\ 0 & l_s & 0 \\ 0 & 0 & \sqrt{\frac{m_s}{l_s}} \end{bmatrix}$$

where S is the scaling matrix, l_s is the ratio between the unscaled and scaled lengths of the segment, m_s is the mass ratio of the segment.

Source: https://ris.utwente.nl/ws/portalfiles/portal/283741263/Thesis_A_Karatsidis.pdf

```
ratio_l_t1 = l_t / l_t_MdL
ratio_l_s1 = l_s / l_s_MdL
ratio_l_ts1 = (l_t+l_s) / (l_t_MdL + l_s_MdL)
ratio_l_p1 = l_p / l_p_MdL
ratio_l_f1 = l_f / l_f_MdL
ratio_l_sf1 = l_sf / l_sf_MdL
```

```

AM_ref = np.zeros_like(AM1)
AM_ref[0, :] = AM1[0, :]/ratio_l_t1
AM_ref[1, :] = AM1[1, :]/ratio_l_sf1
AM_ref[2, :] = AM1[2, :]/ratio_l_p1

AM_ref_Vicon = np.zeros_like(AM1_Vicon)
AM_ref_Vicon[0, :] = AM1_Vicon[0, :]/ratio_l_t1
AM_ref_Vicon[1, :] = AM1_Vicon[1, :]/ratio_l_s1
AM_ref_Vicon[2, :] = AM1_Vicon[2, :]/ratio_l_p1
AM_ref_Vicon[3, :] = AM1_Vicon[3, :]/ratio_l_f1

```

We start from the reference body dimensions of De Leva, for men and women separately, and multiply these dimensions with the matrix S.

```

def ScaleAM(WF, L_ref, l_part):

    if len(WF) == 5:
        l_t_ref, l_sf_ref, l_p_ref, l_ref = L_ref[0], L_ref[1], L_ref[2], L_ref[3]
        ratio_t = l_t_ref / l_ref
        ratio_sf = l_sf_ref / l_ref
        ratio_p = l_p_ref / l_ref

        WF_t, WF_s, WF_p = WF[0], WF[1], WF[4]

        ls_t = ratio_t*l_part / l_t_ref
        S_t = np.array([[np.sqrt(WF_t/ls_t), 0, 0], [0, ls_t, 0], [0, 0, np.sqrt(WF_t/ls_t)]])
        t_part = S_t * l_t_ref
        w_t, l_t, d_t = np.diag(t_part)

        ls_sf = ratio_sf*l_part / l_sf_ref
        S_sf = np.array([[np.sqrt(WF_s/ls_sf), 0, 0], [0, ls_sf, 0], [0, 0, np.sqrt(WF_s/ls_sf)]])
        sf_part = S_sf * l_sf_ref
        w_sf, l_sf, d_sf = np.diag(sf_part)

        ls_p = ratio_p*l_part / l_p_ref
        S_p = np.array([[np.sqrt(WF_p/ls_p), 0, 0], [0, ls_p, 0], [0, 0, np.sqrt(WF_p/ls_p)]])
        p_part = S_p * l_p_ref
        w_p, l_p, d_p = np.diag(p_part)

        AM = np.array([[w_t, d_t, l_t], [w_sf, d_sf, l_sf], [w_p, d_p, l_p]])

    if len(WF) == 7:
        l_t_ref, l_s_ref, l_p_ref, l_f_ref, l_ref = L_ref[0], L_ref[1], L_ref[2], L_ref[3], L_ref[4]
        ratio_t = l_t_ref / l_ref
        ratio_s = l_s_ref / l_ref
        ratio_p = l_p_ref / l_ref
        ratio_f = l_f_ref / l_ref

        WF_t, WF_s, WF_p, WF_f = WF[0], WF[1], WF[4], WF[5]

        ls_t = ratio_t*l_part / l_t_ref
        S_t = np.array([[np.sqrt(WF_t/ls_t), 0, 0], [0, ls_t, 0], [0, 0, np.sqrt(WF_t/ls_t)]])
        t_part = S_t * l_t_ref    # note: width and depth are the same
        w_t, l_t, d_t = np.diag(t_part)

        ls_s = ratio_s*l_part / l_s_ref
        S_s = np.array([[np.sqrt(WF_s/ls_s), 0, 0], [0, ls_s, 0], [0, 0, np.sqrt(WF_s/ls_s)]])
        s_part = S_s * l_s_ref
        w_s, l_s, d_s = np.diag(s_part)

        ls_p = ratio_p*l_part / l_p_ref
        S_p = np.array([[np.sqrt(WF_p/ls_p), 0, 0], [0, ls_p, 0], [0, 0, np.sqrt(WF_p/ls_p)]])

```

```

p_part = S_p * l_p_ref
w_p, l_p, d_p = np.diag(p_part)

ls_f = ratio_f*l_part / l_f_ref
S_f = np.array([[np.sqrt(WF_f/ls_f), 0, 0], [0, ls_f, 0], [0, 0, np.sqrt(WF_f/ls_f)]])
f_part = S_f * l_f_ref
w_f, l_f, d_f = np.diag(f_part)

AM = np.array([[w_t, d_t, l_t], [w_s, d_s, l_s], [w_p, d_p, l_p], [w_f, d_f, l_f]])

return AM

```

```

# weight factors for men
WF_Mthigh_dL = 14.16 /100
WF_Mshank_dL = 4.33 /100
WF_Mfoot_dL = 1.37 /100
WF_Mpelvis_dL = 1 - 2*(WF_Mthigh_dL + WF_Mshank_dL + WF_Mfoot_dL)

WF_M_dL = np.array([WF_Mthigh_dL, WF_Mshank_dL+WF_Mfoot_dL, WF_Mthigh_dL, WF_Mshank_dL+WF_Mfoot_dL, WF_Mpelvis_dL])
WF_M_dL_Vicon = np.array([WF_Mthigh_dL, WF_Mshank_dL, WF_Mthigh_dL, WF_Mshank_dL, WF_Mpelvis_dL, WF_Mfoot_dL])

```

```

# Participant 0
AM0 = ScaleAM(WF_M_dL, L_MdL, l0)
AM0_Vicon = ScaleAM(WF_M_dL_Vicon, L_MdL_Vicon, l0)

# Participant 1
AM1 = ScaleAM(WF_M_dL, L_MdL, l1)
AM1_Vicon = ScaleAM(WF_M_dL_Vicon, L_MdL_Vicon, l1)

# Participant 2
AM2 = ScaleAM(WF_M_dL, L_MdL, l2)
AM2_Vicon = ScaleAM(WF_M_dL_Vicon, L_MdL_Vicon, l2) # with this method: leg length would be 821 mm

# Participant 3
AM3 = ScaleAM(WF_M_dL, L_MdL, l3)
AM3_Vicon = ScaleAM(WF_M_dL_Vicon, L_MdL_Vicon, l3)
L3_ScaleAM = [(AM3[1,0]+AM3[1,1])/2, AM3[0,2]+AM3[1,2]]

# Participant 4
AM4 = ScaleAM(WF_M_dL, L_MdL, l4)
AM4_Vicon = ScaleAM(WF_M_dL_Vicon, L_MdL_Vicon, l4)
L4_ScaleAM = [(AM4[1,0]+AM4[1,1])/2, AM4[0,2]+AM4[1,2]]

# Participant 5
AM5 = ScaleAM(WF_M_dL, L_MdL, l5)
AM5_Vicon = ScaleAM(WF_M_dL_Vicon, L_MdL_Vicon, l5)
L5_ScaleAM = [(AM5[1,0]+AM5[1,1])/2, AM5[0,2]+AM5[1,2]]

# Participant 6
AM6 = ScaleAM(WF_M_dL, L_MdL, l6)
AM6_Vicon = ScaleAM(WF_M_dL_Vicon, L_MdL_Vicon, l6)
L6_ScaleAM = [(AM6[1,0]+AM6[1,1])/2, AM6[0,2]+AM6[1,2]]

# Participant 7
AM7 = ScaleAM(WF_M_dL, L_MdL, l7)
AM7_Vicon = ScaleAM(WF_M_dL_Vicon, L_MdL_Vicon, l7)
L7_ScaleAM = [(AM7[1,0]+AM7[1,1])/2, AM7[0,2]+AM7[1,2]]

```

✓ 1.2 Method 2

```

def ScaleAM2(AM_ref, ratio_l):
    AM = copy.deepcopy(AM_ref)

```

```

AM = ratio_l * AM_ref
return AM

```

```

# Participant 0
AM = ScaleAM2(AM_ref, ratio_l)
AM_Vicon = ScaleAM2(AM_ref_Vicon, ratio_l)

# Participant 1
AM1 = ScaleAM2(AM_ref, ratio_l1)
AM1_Vicon = ScaleAM2(AM_ref_Vicon, ratio_l1)

# Participant 2
AM2 = ScaleAM2(AM_ref, ratio_l2)
AM2_Vicon = ScaleAM2(AM_ref_Vicon, ratio_l2) # with this method: leg length would be 821 mm

# Participant 3
AM3 = ScaleAM2(AM_ref, ratio_l3)
AM3_Vicon = ScaleAM2(AM_ref_Vicon, ratio_l3)
L3_ScaleAM2 = [(AM3[1,0]+AM3[1,1])/2, AM3[0,2]+AM3[1,2]]

# Participant 4
AM4 = ScaleAM2(AM_ref, ratio_l4)
AM4_Vicon = ScaleAM2(AM_ref_Vicon, ratio_l4)
L4_ScaleAM2 = [(AM4[1,0]+AM4[1,1])/2, AM4[0,2]+AM4[1,2]]

# Participant 5
AM5 = ScaleAM2(AM_ref, ratio_l5)
AM5_Vicon = ScaleAM2(AM_ref_Vicon, ratio_l5)
L5_ScaleAM2 = [(AM5[1,0]+AM5[1,1])/2, AM5[0,2]+AM5[1,2]]

# Participant 6
AM6 = ScaleAM2(AM_ref, ratio_l6)
AM6_Vicon = ScaleAM2(AM_ref_Vicon, ratio_l6)
L6_ScaleAM2 = [(AM6[1,0]+AM6[1,1])/2, AM6[0,2]+AM6[1,2]]

# Participant 7
AM7 = ScaleAM2(AM_ref, ratio_l7)
AM7_Vicon = ScaleAM2(AM_ref_Vicon, ratio_l7)
L7_ScaleAM2 = [(AM7[1,0]+AM7[1,1])/2, AM7[0,2]+AM7[1,2]]

```

2. Load the dataset

2.1 IMU

```

# Participant 0
df_IMU = pd.read_csv(IMU)
header = df_IMU.columns.values.tolist()

# Participant 1
df_IMU1_part1 = pd.read_csv(IMU1, skiprows=[1], nrows=17318)
df_IMU1_part2 = pd.read_csv(IMU1, names=header, skiprows=17330)

# Participant 2
df_IMU2 = pd.read_csv(IMU2)

# Participant 3
df_IMU3_8 = pd.read_csv(IMU3_8)
df_IMU3_10 = pd.read_csv(IMU3_10)
df_IMU3_12 = pd.read_csv(IMU3_12)
df_IMU3_jump = pd.read_csv(IMU3_jump)

```

```
# Participant 4
df_IMU4_8 = pd.read_csv(IMU4_8)
df_IMU4_10 = pd.read_csv(IMU4_10)
df_IMU4_12 = pd.read_csv(IMU4_12)
df_IMU4_jump = pd.read_csv(IMU4_jump)
```

```
# Participant 5
df_IMU5_8 = pd.read_csv(IMU5_8)
df_IMU5_10 = pd.read_csv(IMU5_10)
df_IMU5_12 = pd.read_csv(IMU5_12)
df_IMU5_jump = pd.read_csv(IMU5_jump)
```

```
# Participant 6
df_IMU6_8 = pd.read_csv(IMU6_8)
df_IMU6_10 = pd.read_csv(IMU6_10)
df_IMU6_12 = pd.read_csv(IMU6_12)
df_IMU6_jump = pd.read_csv(IMU6_jump)
```

```
# Participant 7
df_IMU7_8 = pd.read_csv(IMU7_8)
df_IMU7_10 = pd.read_csv(IMU7_10)
df_IMU7_12 = pd.read_csv(IMU7_12)
df_IMU7_jump = pd.read_csv(IMU7_jump)
```

```
def ReadIMUs(df_IMU):
```

```
    t_IMU_initial = df_IMU["Timestamp"].values # The first value is sometimes a random number
    dt_IMU = t_IMU_initial[2] - t_IMU_initial[1] # Note: time[0] can be a random value (I don't know why)
    fs_IMU = int(1/dt_IMU)
    t_IMU = np.arange(0, len(t_IMU_initial)*dt_IMU, dt_IMU)
    # t_IMU = t_IMU_initial - t_IMU_initial[0] # Vragen!
```

```
    # quaternions
```

```
    q1 = np.array(df_IMU[[" q_x1", " q_y1", " q_z1", " q_w1"]].values) # values are in the columns
    q2 = np.array(df_IMU[[" q_x2", " q_y2", " q_z2", " q_w2"]].values)
    q3 = np.array(df_IMU[[" q_x3", " q_y3", " q_z3", " q_w3"]].values)
    q4 = np.array(df_IMU[[" q_x4", " q_y4", " q_z4", " q_w4"]].values)
    q5 = np.array(df_IMU[[" q_x5", " q_y5", " q_z5", " q_w5"]].values)
```

```
    Q = np.array([q1, q2, q3, q4, q5])
```

```
    # accelerations
```

```
    # linear accelerations are expressed in 'LSB/g' (LSB: least-squared bit)
    # Divide by 'LSB' (4096 for range of 8g) and multiply with 'g' to have 'normalised' accelerations in 'g'
    # Divide by 'g' to have accelerations in 'm/s^2'
    # For your information (see Teams Skinetix):
    # For a ±2g range, the sensitivity is 16384 LSB/g
    # For a ±4g range, the sensitivity is 8192 LSB/g
    # For a ±8g range, the sensitivity is 4096 LSB/g
    # For a ±16g range, the sensitivity is 2048 LSB/g
```

```
    # IMU 1 corresponds to "right thigh" (see Teams: Skinetix)
```

```
    a1_x = np.array(df_IMU[" acc_x1"].values)
    a1_y = np.array(df_IMU[" acc_y1"].values)
    a1_z = np.array(df_IMU[" acc_z1"].values)
    a1 = np.concatenate((np.expand_dims(a1_x, axis=1), np.expand_dims(a1_y, axis=1), np.expand_dims(a1_z, a
```

```
    # IMU 2 corresponds to "left shank"
```

```
    a2_x = np.array(df_IMU[" acc_x2"].values)
    a2_y = np.array(df_IMU[" acc_y2"].values)
    a2_z = np.array(df_IMU[" acc_z2"].values)
    a2 = np.concatenate((np.expand_dims(a2_x, axis=1), np.expand_dims(a2_y, axis=1), np.expand_dims(a2_z, a
```

```
    # IMU 3 corresponds to "left thigh"
```

```

a3_x = np.array(df_IMU[" acc_x3"].values)
a3_y = np.array(df_IMU[" acc_y3"].values)
a3_z = np.array(df_IMU[" acc_z3"].values)
a3 = np.concatenate((np.expand_dims(a3_x, axis=1), np.expand_dims(a3_y, axis=1), np.expand_dims(a3_z, a

# IMU 4 corresponds to "right shank"
a4_x = np.array(df_IMU[" acc_x4"].values)
a4_y = np.array(df_IMU[" acc_y4"].values)
a4_z = np.array(df_IMU[" acc_z4"].values)
a4 = np.concatenate((np.expand_dims(a4_x, axis=1), np.expand_dims(a4_y, axis=1), np.expand_dims(a4_z, a

# IMU 5 corresponds to "pelvis"
a5_x = np.array(df_IMU[" acc_x5"].values)
a5_y = np.array(df_IMU[" acc_y5"].values)
a5_z = np.array(df_IMU[" acc_z5"].values)
a5 = np.concatenate((np.expand_dims(a5_x, axis=1), np.expand_dims(a5_y, axis=1), np.expand_dims(a5_z, a

A = np.array([a1, a2, a3, a4, a5])*g/4096 # obtain accelerations in m/s^2?

# Angular velocities

# Angular velocities are expressed in 'LSB/(°/s)'
# Again, the values should be normalised
# Divide by 16.4 for a range of +/- 2000°/s (worst sensitivity, but adjustable if necessary)

# For your information:
# For a range of ±125°/s, the sensitivity is 262.4 LSB/°/s
# For a range of ±250°/s, the sensitivity is 131.2 LSB/°/s
# For a range of ±500°/s, the sensitivity is 65.6 LSB/°/s
# For a range of ±1000°/s, the sensitivity is 32.8 LSB/°/s
# For a range of ±2000°/s, the sensitivity is 16.4 LSB/°/s

# IMU 1 corresponds to "right thigh"
w1_x = np.array(df_IMU[" gyro_x1"].values)
w1_y = np.array(df_IMU[" gyro_y1"].values)
w1_z = np.array(df_IMU[" gyro_z1"].values)
w1 = np.concatenate((np.expand_dims(w1_x, axis=1), np.expand_dims(w1_y, axis=1), np.expand_dims(w1_z, a

# IMU 2 corresponds to "left shank"
w2_x = np.array(df_IMU[" gyro_x2"].values)
w2_y = np.array(df_IMU[" gyro_y2"].values)
w2_z = np.array(df_IMU[" gyro_z2"].values)
w2 = np.concatenate((np.expand_dims(w2_x, axis=1), np.expand_dims(w2_y, axis=1), np.expand_dims(w2_z, a

# IMU 3 corresponds to "left thigh"
w3_x = np.array(df_IMU[" gyro_x3"].values)
w3_y = np.array(df_IMU[" gyro_y3"].values)
w3_z = np.array(df_IMU[" gyro_z3"].values)
w3 = np.concatenate((np.expand_dims(w3_x, axis=1), np.expand_dims(w3_y, axis=1), np.expand_dims(w3_z, a

# IMU 4 corresponds to "right shank"
w4_x = np.array(df_IMU[" gyro_x4"].values)
w4_y = np.array(df_IMU[" gyro_y4"].values)
w4_z = np.array(df_IMU[" gyro_z4"].values)
w4 = np.concatenate((np.expand_dims(w4_x, axis=1), np.expand_dims(w4_y, axis=1), np.expand_dims(w4_z, a

# IMU 5 corresponds to "pelvis"
w5_x = np.array(df_IMU[" gyro_x5"].values)
w5_y = np.array(df_IMU[" gyro_y5"].values)
w5_z = np.array(df_IMU[" gyro_z5"].values)
w5 = np.concatenate((np.expand_dims(w5_x, axis=1), np.expand_dims(w5_y, axis=1), np.expand_dims(w5_z, a

# convert deg/s to rad/s
W = np.deg2rad(np.array([w1, w2, w3, w4, w5])/16.4)

```

```
return t_IMU, fs_IMU, Q, A, W
```

```
# Participant 0
t_IMU_initial, fs_IMU, Q_initial, A_initial, W_initial = ReadIMUs(df_IMU)

# Participant 1
t1_IMU_part1_initial, fs1_IMU_part1, Q1_part1_initial, A1_part1_initial, W1_part1_initial = ReadIMUs(df_IMU1)
t1_IMU_part2_initial, fs1_IMU_part2, Q1_part2_initial, A1_part2_initial, W1_part2_initial = ReadIMUs(df_IMU1)

# Participant 2
t2_IMU_initial, fs2_IMU, Q2_initial, A2_initial, W2_initial = ReadIMUs(df_IMU2)

# Participant 3
t3_8_IMU_initial, fs3_8_IMU, Q3_8_initial, A3_8_initial, W3_8_initial = ReadIMUs(df_IMU3_8)
t3_10_IMU_initial, fs3_10_IMU, Q3_10_initial, A3_10_initial, W3_10_initial = ReadIMUs(df_IMU3_10)
t3_12_IMU_initial, fs3_12_IMU, Q3_12_initial, A3_12_initial, W3_12_initial = ReadIMUs(df_IMU3_12)
t3_jump_IMU_initial, fs3_jump_IMU, Q3_jump_initial, A3_jump_initial, W3_jump_initial = ReadIMUs(df_IMU3_jump)

# Participant 4
t4_8_IMU_initial, fs4_8_IMU, Q4_8_initial, A4_8_initial, W4_8_initial = ReadIMUs(df_IMU4_8)
t4_10_IMU_initial, fs4_10_IMU, Q4_10_initial, A4_10_initial, W4_10_initial = ReadIMUs(df_IMU4_10)
t4_12_IMU_initial, fs4_12_IMU, Q4_12_initial, A4_12_initial, W4_12_initial = ReadIMUs(df_IMU4_12)
t4_jump_IMU_initial, fs4_jump_IMU, Q4_jump_initial, A4_jump_initial, W4_jump_initial = ReadIMUs(df_IMU4_jump)

# Participant 5
t5_8_IMU_initial, fs5_8_IMU, Q5_8_initial, A5_8_initial, W5_8_initial = ReadIMUs(df_IMU5_8)
t5_10_IMU_initial, fs5_10_IMU, Q5_10_initial, A5_10_initial, W5_10_initial = ReadIMUs(df_IMU5_10)
t5_12_IMU_initial, fs5_12_IMU, Q5_12_initial, A5_12_initial, W5_12_initial = ReadIMUs(df_IMU5_12)
t5_jump_IMU_initial, fs5_jump_IMU, Q5_jump_initial, A5_jump_initial, W5_jump_initial = ReadIMUs(df_IMU5_jump)

# Participant 6
t6_8_IMU_initial, fs6_8_IMU, Q6_8_initial, A6_8_initial, W6_8_initial = ReadIMUs(df_IMU6_8)
t6_10_IMU_initial, fs6_10_IMU, Q6_10_initial, A6_10_initial, W6_10_initial = ReadIMUs(df_IMU6_10)
t6_12_IMU_initial, fs6_12_IMU, Q6_12_initial, A6_12_initial, W6_12_initial = ReadIMUs(df_IMU6_12)
t6_jump_IMU_initial, fs6_jump_IMU, Q6_jump_initial, A6_jump_initial, W6_jump_initial = ReadIMUs(df_IMU6_jump)

# Participant 7
t7_8_IMU_initial, fs7_8_IMU, Q7_8_initial, A7_8_initial, W7_8_initial = ReadIMUs(df_IMU7_8)
t7_10_IMU_initial, fs7_10_IMU, Q7_10_initial, A7_10_initial, W7_10_initial = ReadIMUs(df_IMU7_10)
t7_12_IMU_initial, fs7_12_IMU, Q7_12_initial, A7_12_initial, W7_12_initial = ReadIMUs(df_IMU7_12)
t7_jump_IMU_initial, fs7_jump_IMU, Q7_jump_initial, A7_jump_initial, W7_jump_initial = ReadIMUs(df_IMU7_jump)
```

```
# Note: the graphs of the forces and accelerations show that the local frame of the IMU is not aligned with
# Solve by rotating the accelerations to the initial frame, via the quaternions
# q_w, q_x, q_y en q_z zijn "quaternions" (alternative for Euler-angles)
```

```
def RotateIMUs(Q, A, W):
# Rotate the local frame of the IMU to the reference frame of the lab, via the quaternions
```

```
    q1 = Q[0, :, :]
    q2 = Q[1, :, :]
    q3 = Q[2, :, :]
    q4 = Q[3, :, :]
    q5 = Q[4, :, :]

    rotations1 = Rotation.from_quat(q1)
    r1 = rotations1.as_matrix()
    rotations2 = Rotation.from_quat(q2)
    r2 = rotations2.as_matrix()
    rotations3 = Rotation.from_quat(q3)
    r3 = rotations3.as_matrix()
    rotations4 = Rotation.from_quat(q4)
    r4 = rotations4.as_matrix()
```

```

rotations5 = Rotation.from_quat(q5)
r5 = rotations5.as_matrix()

R = np.array([r1, r2, r3, r4, r5])

A_rot = np.zeros_like(A)
W_rot = np.zeros_like(W)

for j in range(R.shape[0]):
    for i in range(R.shape[1]):
        # Note: be careful with the calculation order, because the matrix multiplication is NOT commutative
        A_rot[j, i, :] = np.dot(A[j, i, :], np.linalg.inv(R[j, i, :]))
        A_rot[j, i, 0] *= (-1)
        A_rot[j, i, 1] *= (-1)
        A_rot[j, i, 2] -= g
        W_rot[j, i, :] = np.dot(W[j, i, :], np.linalg.inv(R[j, i, :]))

return A_rot, W_rot

```

```

# Participant 0
A_rot_initial, W_rot_initial = RotateIMUs(Q_initial, A_initial, W_initial)

# Participant 1
A1_part1_rot_initial, W1_part1_rot_initial = RotateIMUs(Q1_part1_initial, A1_part1_initial, W1_part1_initial)
A1_part2_rot_initial, W1_part2_rot_initial = RotateIMUs(Q1_part2_initial, A1_part2_initial, W1_part2_initial)

# Participant 2
A2_rot_initial, W2_rot_initial = RotateIMUs(Q2_initial, A2_initial, W2_initial)

# Participant 3
A3_8_rot_initial, W3_8_rot_initial = RotateIMUs(Q3_8_initial, A3_8_initial, W3_8_initial)
A3_10_rot_initial, W3_10_rot_initial = RotateIMUs(Q3_10_initial, A3_10_initial, W3_10_initial)
A3_12_rot_initial, W3_12_rot_initial = RotateIMUs(Q3_12_initial, A3_12_initial, W3_12_initial)
A3_jump_rot_initial, W3_jump_rot_initial = RotateIMUs(Q3_jump_initial, A3_jump_initial, W3_jump_initial)

# Participant 4
A4_8_rot_initial, W4_8_rot_initial = RotateIMUs(Q4_8_initial, A4_8_initial, W4_8_initial)
A4_10_rot_initial, W4_10_rot_initial = RotateIMUs(Q4_10_initial, A4_10_initial, W4_10_initial)
A4_12_rot_initial, W4_12_rot_initial = RotateIMUs(Q4_12_initial, A4_12_initial, W4_12_initial)
A4_jump_rot_initial, W4_jump_rot_initial = RotateIMUs(Q4_jump_initial, A4_jump_initial, W4_jump_initial)

# Participant 5
A5_8_rot_initial, W5_8_rot_initial = RotateIMUs(Q5_8_initial, A5_8_initial, W5_8_initial)
A5_10_rot_initial, W5_10_rot_initial = RotateIMUs(Q5_10_initial, A5_10_initial, W5_10_initial)
A5_12_rot_initial, W5_12_rot_initial = RotateIMUs(Q5_12_initial, A5_12_initial, W5_12_initial)
A5_jump_rot_initial, W5_jump_rot_initial = RotateIMUs(Q5_jump_initial, A5_jump_initial, W5_jump_initial)

# Participant 6
A6_8_rot_initial, W6_8_rot_initial = RotateIMUs(Q6_8_initial, A6_8_initial, W6_8_initial)
A6_10_rot_initial, W6_10_rot_initial = RotateIMUs(Q6_10_initial, A6_10_initial, W6_10_initial)
A6_12_rot_initial, W6_12_rot_initial = RotateIMUs(Q6_12_initial, A6_12_initial, W6_12_initial)
A6_jump_rot_initial, W6_jump_rot_initial = RotateIMUs(Q6_jump_initial, A6_jump_initial, W6_jump_initial)

# Participant 7
A7_8_rot_initial, W7_8_rot_initial = RotateIMUs(Q7_8_initial, A7_8_initial, W7_8_initial)
A7_10_rot_initial, W7_10_rot_initial = RotateIMUs(Q7_10_initial, A7_10_initial, W7_10_initial)
A7_12_rot_initial, W7_12_rot_initial = RotateIMUs(Q7_12_initial, A7_12_initial, W7_12_initial)
A7_jump_rot_initial, W7_jump_rot_initial = RotateIMUs(Q7_jump_initial, A7_jump_initial, W7_jump_initial)

```

```

def VelocityFromIMU(fs, A):
    V = scipy.integrate.cumulative_trapezoid(A, dx=1/fs, axis=1)
    V = scipy.signal.detrend(V, axis=1, type='linear')
    return V

```

```

# Participant 0
V_initial = VelocityFromIMU(fs_IMU, A_rot_initial)

# Participant 2
V2_initial = VelocityFromIMU(fs2_IMU, A2_rot_initial)

# Participant 3
V3_8_initial = VelocityFromIMU(fs3_8_IMU, A3_8_rot_initial)
V3_10_initial = VelocityFromIMU(fs3_10_IMU, A3_10_rot_initial)
V3_12_initial = VelocityFromIMU(fs3_12_IMU, A3_12_rot_initial)
V3_jump_initial = VelocityFromIMU(fs3_jump_IMU, A3_jump_rot_initial)

# Participant 4
V4_8_initial = VelocityFromIMU(fs4_8_IMU, A4_8_rot_initial)
V4_10_initial = VelocityFromIMU(fs4_10_IMU, A4_10_rot_initial)
V4_12_initial = VelocityFromIMU(fs4_12_IMU, A4_12_rot_initial)
V4_jump_initial = VelocityFromIMU(fs4_jump_IMU, A4_jump_rot_initial)

# Participant 5
V5_8_initial = VelocityFromIMU(fs5_8_IMU, A5_8_rot_initial)
V5_10_initial = VelocityFromIMU(fs5_10_IMU, A5_10_rot_initial)
V5_12_initial = VelocityFromIMU(fs5_12_IMU, A5_12_rot_initial)
V5_jump_initial = VelocityFromIMU(fs5_jump_IMU, A5_jump_rot_initial)

# Participant 6
V6_8_initial = VelocityFromIMU(fs6_8_IMU, A6_8_rot_initial)
V6_10_initial = VelocityFromIMU(fs6_10_IMU, A6_10_rot_initial)
V6_12_initial = VelocityFromIMU(fs6_12_IMU, A6_12_rot_initial)
V6_jump_initial = VelocityFromIMU(fs6_jump_IMU, A6_jump_rot_initial)

# Participant 7
V7_8_initial = VelocityFromIMU(fs7_8_IMU, A7_8_rot_initial)
V7_10_initial = VelocityFromIMU(fs7_10_IMU, A7_10_rot_initial)
V7_12_initial = VelocityFromIMU(fs7_12_IMU, A7_12_rot_initial)
V7_jump_initial = VelocityFromIMU(fs7_jump_IMU, A7_jump_rot_initial)

```

```

def PositionFromIMU(fs, A):
    V = scipy.integrate.cumulative_trapezoid(A, dx=1/fs, axis=1)
    X = scipy.integrate.cumulative_trapezoid(V, dx=1/fs, axis=1)
    X = scipy.signal.detrend(X, axis=1, type='linear')
    return X

```

```

# Participant 0
X_initial = PositionFromIMU(fs_IMU, A_rot_initial)

# Participant 2
X2_initial = PositionFromIMU(fs2_IMU, A2_rot_initial)

# Participant 3
X3_8_initial = PositionFromIMU(fs3_8_IMU, A3_8_rot_initial)
X3_10_initial = PositionFromIMU(fs3_10_IMU, A3_10_rot_initial)
X3_12_initial = PositionFromIMU(fs3_12_IMU, A3_12_rot_initial)
X3_jump_initial = PositionFromIMU(fs3_jump_IMU, A3_jump_rot_initial)

# Participant 4
X4_8_initial = PositionFromIMU(fs4_8_IMU, A4_8_rot_initial)
X4_10_initial = PositionFromIMU(fs4_10_IMU, A4_10_rot_initial)
X4_12_initial = PositionFromIMU(fs4_12_IMU, A4_12_rot_initial)
X4_jump_initial = PositionFromIMU(fs4_jump_IMU, A4_jump_rot_initial)

# Participant 5
X5_8_initial = PositionFromIMU(fs5_8_IMU, A5_8_rot_initial)
X5_10_initial = PositionFromIMU(fs5_10_IMU, A5_10_rot_initial)
X5_12_initial = PositionFromIMU(fs5_12_IMU, A5_12_rot_initial)

```

```
X5_jump_initial = PositionFromIMU(fs5_jump_IMU, A5_jump_rot_initial)
```

```
# Participant 6
```

```
X6_8_initial = PositionFromIMU(fs6_8_IMU, A6_8_rot_initial)
```

```
X6_10_initial = PositionFromIMU(fs6_10_IMU, A6_10_rot_initial)
```

```
X6_12_initial = PositionFromIMU(fs6_12_IMU, A6_12_rot_initial)
```

```
X6_jump_initial = PositionFromIMU(fs6_jump_IMU, A6_jump_rot_initial)
```

```
# Participant 7
```

```
X7_8_initial = PositionFromIMU(fs7_8_IMU, A7_8_rot_initial)
```

```
X7_10_initial = PositionFromIMU(fs7_10_IMU, A7_10_rot_initial)
```

```
X7_12_initial = PositionFromIMU(fs7_12_IMU, A7_12_rot_initial)
```

```
X7_jump_initial = PositionFromIMU(fs7_jump_IMU, A7_jump_rot_initial)
```

✓ 2.2 Vicon: treadmill

```
def CheckViconFile(df_Device):
```

```
    # Treadmill and markers should have recorded the same timespan, but treadmill can lack some frames
    # fill NaN values via linear interpolation (maybe better possibilities?)
    # BUT Please note that only method='linear' is supported for DataFrame/Series with a MultiIndex.
```

```
    df_Device = df_Device.interpolate(method='linear', axis=1) # 0: index, # 1: columns
```

```
    # check if the file has already the combined data of both force plates
```

```
    if 'LRFx' not in df_Device.columns:
```

```
        df_Device['LRFx'] = df_Device['LFX'] + df_Device['RFx']
```

```
    if 'LRFy' not in df_Device.columns:
```

```
        df_Device['LRFy'] = df_Device['LFy'] + df_Device['RFy']
```

```
    if 'LRFz' not in df_Device.columns:
```

```
        df_Device['LRFz'] = df_Device['LFz'] + df_Device['RFz']
```

```
    if 'LRMx' not in df_Device.columns:
```

```
        df_Device['LRMx'] = df_Device['LMx'] + df_Device['RMx']
```

```
    if 'LRMy' not in df_Device.columns:
```

```
        df_Device['LRMy'] = df_Device['LMy'] + df_Device['RMy']
```

```
    if 'LRMz' not in df_Device.columns:
```

```
        df_Device['LRMz'] = df_Device['LMz'] + df_Device['RMz']
```

```
    if 'LRCx' not in df_Device.columns:
```

```
        df_Device['LRCx'] = df_Device['LCx'] + df_Device['RCx']
```

```
    if 'LRCy' not in df_Device.columns:
```

```
        df_Device['LRCy'] = df_Device['LCy'] + df_Device['RCy']
```

```
    if 'LRCz' not in df_Device.columns:
```

```
        df_Device['LRCz'] = df_Device['LCz'] + df_Device['RCz']
```

```
    return df_Device
```

```
# Note: I changed the names in the csv-files to read them more easily
```

```
# Participant 0
```

```
df_Device = pd.read_csv(Vicon, skiprows=[0,1,2,4], nrows=587424, usecols=['Frame', 'Sub Frame', 'LFX', 'LFY
```

```
# Participant 1
```

```
# df_Device1 = pd.read_csv(Vicon1, skiprows=[0,1,2,4], nrows=335861, usecols=['Frame', 'Sub Frame', 'LFX',
```

```
# Participant 2
```

```
df_Device2 = pd.read_csv(Vicon2, skiprows=[0,1,2,4], nrows=278064, usecols=['Frame', 'Sub Frame', 'LFX', 'L
```

```
# Participant 3
```

```
df_Device3_8 = pd.read_csv(Vicon3_8, skiprows=[0,1,2,3,4], nrows=155624, names = ['Frame','Sub Frame','LFz1
```

```
df_Device3_8 = CheckViconFile(df_Device3_8)
```

```
df_Device3_10 = pd.read_csv(Vicon3_10, skiprows=[0,1,2,3,4], nrows=177999, names = ['Frame','Sub Frame','LF
```

```
df_Device3_10 = CheckViconFile(df_Device3_10)
```

```

df_Device3_12 = pd.read_csv(Vicon3_12, skiprows=[0,1,2,3,4], nrows=202572, names = ['Frame','Sub Frame','LF
df_Device3_12 = CheckViconFile(df_Device3_12)
df_Device3_jump = pd.read_csv(Vicon3_jump, skiprows=[0,1,2,3,4], nrows=37428, names = ['Frame','Sub Frame',
df_Device3_jump = CheckViconFile(df_Device3_jump)

# Participant 4
df_Device4_8 = pd.read_csv(Vicon4_8, skiprows=[0,1,2,3,4], nrows=169416, names = ['Frame','Sub Frame','LFz1
df_Device4_8 = CheckViconFile(df_Device4_8)
df_Device4_10 = pd.read_csv(Vicon4_10, skiprows=[0,1,2,3,4], nrows=185016, names = ['Frame','Sub Frame','LF
df_Device4_10 = CheckViconFile(df_Device4_10)
df_Device4_12 = pd.read_csv(Vicon4_12, skiprows=[0,1,2,3,4], nrows=190608, names = ['Frame','Sub Frame','LF
df_Device4_12 = CheckViconFile(df_Device4_12)
df_Device4_jump = pd.read_csv(Vicon4_jump, skiprows=[0,1,2,3,4], nrows=43912, names = ['Frame','Sub Frame',
df_Device4_jump = CheckViconFile(df_Device4_jump)

# Participant 5
df_Device5_8 = pd.read_csv(Vicon5_8, skiprows=[0,1,2,3,4], nrows=171672, names = ['Frame','Sub Frame','LFz1
df_Device5_8 = CheckViconFile(df_Device5_8)
df_Device5_10 = pd.read_csv(Vicon5_10, skiprows=[0,1,2,3,4], nrows=199352, names = ['Frame','Sub Frame','LF
df_Device5_10 = CheckViconFile(df_Device5_10)
df_Device5_12 = pd.read_csv(Vicon5_12, skiprows=[0,1,2,3,4], nrows=185048, names = ['Frame','Sub Frame','LF
df_Device5_12 = CheckViconFile(df_Device5_12)
df_Device5_jump = pd.read_csv(Vicon5_jump, skiprows=[0,1,2,3,4], nrows=14152, names = ['Frame','Sub Frame',
df_Device5_jump = CheckViconFile(df_Device5_jump)

# Participant 6
df_Device6_8 = pd.read_csv(Vicon6_8, skiprows=[0,1,2,3,4], nrows=142353, names = ['Frame','Sub Frame','LFz1
df_Device6_8 = CheckViconFile(df_Device6_8)
df_Device6_10 = pd.read_csv(Vicon6_10, skiprows=[0,1,2,3,4], nrows=168368, names = ['Frame','Sub Frame','LF
df_Device6_10 = CheckViconFile(df_Device6_10)
df_Device6_12 = pd.read_csv(Vicon6_12, skiprows=[0,1,2,3,4], nrows=178408, names = ['Frame','Sub Frame','LF
df_Device6_12 = CheckViconFile(df_Device6_12)
df_Device6_jump = pd.read_csv(Vicon6_jump, skiprows=[0,1,2,3,4], nrows=34408, names = ['Frame','Sub Frame',
df_Device6_jump = CheckViconFile(df_Device6_jump)

# Participant 7
df_Device7_8 = pd.read_csv(Vicon7_8, skiprows=[0,1], nrows=159545, names = ['Frame','Sub Frame','LFz1','LFz
df_Device7_8 = CheckViconFile(df_Device7_8)
df_Device7_10 = pd.read_csv(Vicon7_10, skiprows=[0,1], nrows=172849, names = ['Frame','Sub Frame','LFz1','L
df_Device7_10 = CheckViconFile(df_Device7_10)
df_Device7_12 = pd.read_csv(Vicon7_12, skiprows=[0,1,2,3,4], nrows=202808, names = ['Frame','Sub Frame','LF
df_Device7_12 = CheckViconFile(df_Device7_12)
df_Device7_jump = pd.read_csv(Vicon7_jump, skiprows=[0,1,2,3,4], nrows=42424, names = ['Frame','Sub Frame',
df_Device7_jump = CheckViconFile(df_Device7_jump)

```

Note

Het center of pressure in de z-richting light altijd op -60 mm = -6 cm

x: medio-lateral direction

y: anterior-posterior direction

z: vertical direction (multiplied with -1, because axis/connection is reversed)

def ReadTreadmill(df_Device, M, height):

fs_Vicon = 960 # Hz

t_Vicon = np.arange(0, 1/fs_Vicon*len(df_Device["Frame"]), 1/fs_Vicon)

measured GRF of the left belt of the force plate

mGRFx_left = np.array(df_Device["LFx"].values/(M*g)) # normalise the GRF with body weight

mGRFy_left = np.array(df_Device["LFy"].values/(M*g))

mGRFz_left = np.array(df_Device["LFz"].values/(M*g)*(-1))

mGRF_left = np.concatenate((np.expand_dims(mGRFx_left, axis=1), np.expand_dims(mGRFy_left, axis=1), np.

measured GRF of the right belt of the force plate

```

mGRFx_right = np.array(df_Device["RFx"].values/(M*g))
mGRFy_right = np.array(df_Device["RFy"].values/(M*g))
mGRFz_right = np.array(df_Device["RFz"].values/(M*g)*(-1))
mGRF_right = np.concatenate((np.expand_dims(mGRFx_right, axis=1), np.expand_dims(mGRFy_right, axis=1),

# measured GRF of the combined left and right belt of the force plate
mGRFx = np.array(df_Device["LRFx"].values/(M*g))
mGRFy = np.array(df_Device["LRFy"].values/(M*g))
mGRFz = np.array(df_Device["LRFz"].values/(M*g)*(-1))
mGRF = np.concatenate((np.expand_dims(mGRFx, axis=1), np.expand_dims(mGRFy, axis=1), np.expand_dims(mGR

# measured GRM of the left belt of the force plate [N * mm]
mGRMx_left = np.array(df_Device["LMx"].values/(M*g*height*1000)) # normalise by BW * BH, and set unit
mGRMy_left = np.array(df_Device["LMy"].values/(M*g*height*1000))
mGRMz_left = np.array(df_Device["LMz"].values/(M*g*height*1000)*(-1))
mGRM_left = np.concatenate((np.expand_dims(mGRMx_left, axis=1), np.expand_dims(mGRMy_left, axis=1), np.

# measured GRM of the right belt of the force plate
mGRMx_right = np.array(df_Device["RMx"].values/(M*g*height*1000))
mGRMy_right = np.array(df_Device["RMy"].values/(M*g*height*1000))
mGRMz_right = np.array(df_Device["RMz"].values/(M*g*height*1000)*(-1))
mGRM_right = np.concatenate((np.expand_dims(mGRMx_right, axis=1), np.expand_dims(mGRMy_right, axis=1),

# measured GRM of the combined left and right belt of the force plate
mGRMx = np.array(df_Device["LRMx"].values/(M*g*height*1000))
mGRMy = np.array(df_Device["LRMy"].values/(M*g*height*1000))
mGRMz = np.array(df_Device["LRMz"].values/(M*g*height*1000)*(-1))
mGRM = np.concatenate((np.expand_dims(mGRMx, axis=1), np.expand_dims(mGRMy, axis=1), np.expand_dims(mGR

# measured CoP of the left belt of the force plate [mm]
mCoPx_left = np.array(df_Device["LCx"].values/1000) # set unit to m
mCoPy_left = np.array(df_Device["LCy"].values/1000)
mCoPz_left = np.array(df_Device["LCz"].values/1000*(-1))
mCoP_left = np.concatenate((np.expand_dims(mCoPx_left, axis=1), np.expand_dims(mCoPy_left, axis=1), np.

# measured CoP of the right belt of the force plate
mCoPx_right = np.array(df_Device["RCx"].values/1000)
mCoPy_right = np.array(df_Device["RCy"].values/1000)
mCoPz_right = np.array(df_Device["RCz"].values/1000*(-1))
mCoP_right = np.concatenate((np.expand_dims(mCoPx_right, axis=1), np.expand_dims(mCoPy_right, axis=1),

# measured CoP of the combined left and right belt of the force plate
mCoPx = np.array(df_Device["LRCx"].values/1000)
mCoPy = np.array(df_Device["LRCy"].values/1000)
mCoPz = np.array(df_Device["LRCz"].values/1000*(-1))
mCoP = np.concatenate((np.expand_dims(mCoPx, axis=1), np.expand_dims(mCoPy, axis=1), np.expand_dims(mCo

return t_Vicon, fs_Vicon, mGRF_left, mGRF_right, mGRF, mGRM_left, mGRM_right, mGRM, mCoP_left, mCoP_right,

```

```

# Participant 0
t_Vicon_initial, fs_Vicon, mGRF_left_initial, mGRF_right_initial, mGRF_initial, mGRM_left_initial, mGRM_right_initial, mGRM_initial

# Participant 1
# mGRF_left_initial1, mGRF_right_initial1, mGRF_initial1, mGRM_left_initial1, mGRM_right_initial1, mGRM_initial1

# Participant 2
t_Vicon2_initial, fs_Vicon2, mGRF_left_initial2, mGRF_right_initial2, mGRF_initial2, mGRM_left_initial2, mGRM_right_initial2, mGRM_initial2

# Participant 3
t_Vicon3_8_initial, fs_Vicon3_8, mGRF_left_initial3_8, mGRF_right_initial3_8, mGRF_initial3_8, mGRM_left_initial3_8, mGRM_right_initial3_8, mGRM_initial3_8
t_Vicon3_10_initial, fs_Vicon3_10, mGRF_left_initial3_10, mGRF_right_initial3_10, mGRF_initial3_10, mGRM_left_initial3_10, mGRM_right_initial3_10, mGRM_initial3_10
t_Vicon3_12_initial, fs_Vicon3_12, mGRF_left_initial3_12, mGRF_right_initial3_12, mGRF_initial3_12, mGRM_left_initial3_12, mGRM_right_initial3_12, mGRM_initial3_12
t_Vicon3_jump_initial, fs_Vicon3_jump, mGRF_left_initial3_jump, mGRF_right_initial3_jump, mGRF_initial3_jump, mGRM_left_initial3_jump, mGRM_right_initial3_jump, mGRM_initial3_jump

# Participant 4

```

```

t_Vicon4_8_initial, fs_Vicon4_8, mGRF_left_initial4_8, mGRF_right_initial4_8, mGRF_initial4_8, mGRM_left_in
t_Vicon4_10_initial, fs_Vicon4_10, mGRF_left_initial4_10, mGRF_right_initial4_10, mGRF_initial4_10, mGRM_le
t_Vicon4_12_initial, fs_Vicon4_12, mGRF_left_initial4_12, mGRF_right_initial4_12, mGRF_initial4_12, mGRM_le
t_Vicon4_jump_initial, fs_Vicon4_jump, mGRF_left_initial4_jump, mGRF_right_initial4_jump, mGRF_initial4_jum

# Participant 5
t_Vicon5_8_initial, fs_Vicon5_8, mGRF_left_initial5_8, mGRF_right_initial5_8, mGRF_initial5_8, mGRM_left_in
t_Vicon5_10_initial, fs_Vicon5_10, mGRF_left_initial5_10, mGRF_right_initial5_10, mGRF_initial5_10, mGRM_le
t_Vicon5_12_initial, fs_Vicon5_12, mGRF_left_initial5_12, mGRF_right_initial5_12, mGRF_initial5_12, mGRM_le
t_Vicon5_jump_initial, fs_Vicon5_jump, mGRF_left_initial5_jump, mGRF_right_initial5_jump, mGRF_initial5_jum

# Participant 6
t_Vicon6_8_initial, fs_Vicon6_8, mGRF_left_initial6_8, mGRF_right_initial6_8, mGRF_initial6_8, mGRM_left_in
t_Vicon6_10_initial, fs_Vicon6_10, mGRF_left_initial6_10, mGRF_right_initial6_10, mGRF_initial6_10, mGRM_le
t_Vicon6_12_initial, fs_Vicon6_12, mGRF_left_initial6_12, mGRF_right_initial6_12, mGRF_initial6_12, mGRM_le
t_Vicon6_jump_initial, fs_Vicon6_jump, mGRF_left_initial6_jump, mGRF_right_initial6_jump, mGRF_initial6_jum

# Participant 7
t_Vicon7_8_initial, fs_Vicon7_8, mGRF_left_initial7_8, mGRF_right_initial7_8, mGRF_initial7_8, mGRM_left_in
t_Vicon7_10_initial, fs_Vicon7_10, mGRF_left_initial7_10, mGRF_right_initial7_10, mGRF_initial7_10, mGRM_le
t_Vicon7_12_initial, fs_Vicon7_12, mGRF_left_initial7_12, mGRF_right_initial7_12, mGRF_initial7_12, mGRM_le
t_Vicon7_jump_initial, fs_Vicon7_jump, mGRF_left_initial7_jump, mGRF_right_initial7_jump, mGRF_initial7_jum

```

2.3 Vicon: trajectories

```

header = ['Frame', 'Sub Frame', 'LASIx', 'LASIy', 'LASIz', 'RASIx', 'RASIy', 'RASIz', 'LPSIx', 'LPSIy', 'LP
names = ['Frame', 'Sub Frame', 'LASIx', 'LASIy', 'LASIz', 'RASIx', 'RASIy', 'RASIz', 'LPSIx', 'LPSIy', 'LPS

# Participant 0
df_Trajectories = pd.read_csv(Vicon, skiprows=734300, nrows=73433)

# Participant 1
df_Trajectories1 = pd.read_csv(Vicon1, names = header, skiprows=419842)

# Participant 2
df_Trajectories2 = pd.read_csv(Vicon2, names = header, skiprows=347602)

# Participant 3
df_Trajectories3_8 = pd.read_csv(Vicon3_8, names=header, skiprows=155635, usecols=header)
df_Trajectories3_10 = pd.read_csv(Vicon3_10, names=header, skiprows=178010, usecols=header)
df_Trajectories3_12 = pd.read_csv(Vicon3_12, names=header, skiprows=202583, usecols=header)
df_Trajectories3_jump = pd.read_csv(Vicon3_jump, names=header, skiprows=37439, usecols=header)

# Participant 4
df_Trajectories4_8 = pd.read_csv(Vicon4_8, names=header, skiprows=169427, usecols=header)
df_Trajectories4_10 = pd.read_csv(Vicon4_10, names=header, skiprows=185027, usecols=header)
df_Trajectories4_12 = pd.read_csv(Vicon4_12, names=header, skiprows=190619, usecols=header)
df_Trajectories4_jump = pd.read_csv(Vicon4_jump, names=header, skiprows=43923, usecols=header)

# Participant 5
df_Trajectories5_8 = pd.read_csv(Vicon5_8, names=header, skiprows=171683, usecols=header)
df_Trajectories5_10 = pd.read_csv(Vicon5_10, names=header, skiprows=199363, usecols=header)
df_Trajectories5_12 = pd.read_csv(Vicon5_12, names=header, skiprows=185059, usecols=header)
df_Trajectories5_jump = pd.read_csv(Vicon5_jump, names=header, skiprows=14163, usecols=header)

# Participant 6
df_Trajectories6_8 = pd.read_csv(Vicon6_8, names=header, skiprows=142364, usecols=header)
df_Trajectories6_10 = pd.read_csv(Vicon6_10, names=header, skiprows=168379, usecols=header)
df_Trajectories6_12 = pd.read_csv(Vicon6_12, names=header, skiprows=178419, usecols=header)
df_Trajectories6_jump = pd.read_csv(Vicon6_jump, names=header, skiprows=34419, usecols=header)

# Participant 7
df_Trajectories7_8 = pd.read_csv(Vicon7_8, names=header, skiprows=159552, usecols=header)

```

```
df_Trajectories7_10 = pd.read_csv(Vicon7_10, names=header, skiprows=172856, usecols=header)
df_Trajectories7_12 = pd.read_csv(Vicon7_12, names=header, skiprows=202819, usecols=header)
df_Trajectories7_jump = pd.read_csv(Vicon7_jump, names=header, skiprows=42435, usecols=header)
```

```
def VelocityFromMarker(marker, dt):
    Xx, Xy, Xz = marker[:, 0], marker[:, 1], marker[:, 2]
    Vx = np.gradient(Xx, dt, edge_order=2) # gradient works only for 1D-arrays
    Vy = np.gradient(Xy, dt, edge_order=2)
    Vz = np.gradient(Xz, dt, edge_order=2)
    return np.array([Vx, Vy, Vz]).T
```

```
def AccelerationFromMarker(marker, dt):
    Xx, Xy, Xz = marker[:, 0], marker[:, 1], marker[:, 2]
    Ax = np.gradient(np.gradient(Xx, dt, edge_order=2), dt, edge_order=2) # gradient works only for 1D-ar
    Ay = np.gradient(np.gradient(Xy, dt, edge_order=2), dt, edge_order=2)
    Az = np.gradient(np.gradient(Xz, dt, edge_order=2), dt, edge_order=2)
    return np.array([Ax, Ay, Az]).T
```

```
# Position of the markers is expressed in 'mm'
# The markers are sampled at a frequency of 120 Hz
```

```
def ReadMarkers(df_Trajectories):
```

```
    fs_markers = 120 # Hz
    t_markers = np.arange(0, 1/fs_markers*len(df_Trajectories["Frame"]), 1/fs_markers)
```

```
    # Treadmill and markers should have recorded the same timespan, but markers can lack some frames
    # fill NaN values via linear interpolation (maybe better possibilities?)
    df_Trajectories = df_Trajectories.interpolate(method='linear', axis=1)
```

```
    # Be careful: dropna removes the rows with 'nans'. This can create errors if a lot of data is missing!
    LASI = np.array(df_Trajectories[["LASIx", "LASIy", "LASIz"]].values)/1000 # left anterior superior ili
    RASI = np.array(df_Trajectories[["RASIx", "RASIy", "RASIz"]].values)/1000 # right anterior superior i
    LPSI = np.array(df_Trajectories[["LPSIx", "LPSIy", "LPSIz"]].values)/1000 # left posterior superior i
    RPSI = np.array(df_Trajectories[["RPSIx", "RPSIy", "RPSIz"]].values)/1000 # right posterior superior
    LTHI = np.array(df_Trajectories[["LTHIx", "LTHIy", "LTHIz"]].values)/1000 # left thigh -> should cor
    RTHI = np.array(df_Trajectories[["RTHIx", "RTHIy", "RTHIz"]].values)/1000 # right thigh -> should cor
    LKNE = np.array(df_Trajectories[["LKNEx", "LKNEy", "LKNEz"]].values)/1000 # left knee
    RKNE = np.array(df_Trajectories[["RKNEx", "RKNEy", "RKNEz"]].values)/1000 # right knee
    LTIB = np.array(df_Trajectories[["LTIBx", "LTIBy", "LTIBz"]].values)/1000 # left tibia -> should cor
    RTIB = np.array(df_Trajectories[["RTIBx", "RTIBy", "RTIBz"]].values)/1000 # right tibia -> should co
    LANK = np.array(df_Trajectories[["LANKx", "LANKy", "LANKz"]].values)/1000 # left ankle
    RANK = np.array(df_Trajectories[["RANKx", "RANKy", "RANKz"]].values)/1000 # right ankle
    LHEE = np.array(df_Trajectories[["LHEEx", "LHEEy", "LHEEz"]].values)/1000 # left heel
    RHEE = np.array(df_Trajectories[["RHEEx", "RHEEy", "RHEEz"]].values)/1000 # right heel
    LTOE = np.array(df_Trajectories[["LTOEx", "LTOEy", "LTOEz"]].values)/1000 # left toe
    RTOE = np.array(df_Trajectories[["RTOEx", "RTOEy", "RTOEz"]].values)/1000 # right toe
```

```
    traj_markers = np.array([LASI, RASI, LPSI, RPSI, LTHI, RTHI, LKNE, RKNE, LTIB, RTIB, LANK, RANK, LHEE,
```

```
    # Convert positions to accelerations, in order to compare IMU and Vicon
    # Compared to the IMUs, the gravitational acceleration is not included?
```

```
    dt = 1/fs_markers
```

```
    # right thigh
```

```
    x1_Vicon = RTHI
```

```
    a1_Vicon = AccelerationFromMarker(RTHI, dt) # Be careful: this function removes the rows with 'nans'!
```

```
    # left shank
```

```
    x2_Vicon = LTIB
```

```
    a2_Vicon = AccelerationFromMarker(LTIB, dt)
```

```

# left thigh
x3_Vicon = LTHI
a3_Vicon = AccelerationFromMarker(LTHI, dt)

# right shank
x4_Vicon = RTIB
a4_Vicon = AccelerationFromMarker(RTIB, dt)

# pelvis
marker_pelvis = (LPSI + RPSI)/2
x5_Vicon = marker_pelvis
a5_Vicon = AccelerationFromMarker(marker_pelvis, dt)

# left foot
marker_foot_left = (LTOE + LHEE + LANK)/3
x6_Vicon = marker_foot_left
a6_Vicon = AccelerationFromMarker(marker_foot_left, dt)

# right foot
marker_foot_right = (RTOE + RHEE + RANK)/3
x7_Vicon = marker_foot_right
a7_Vicon = AccelerationFromMarker(marker_foot_right, dt)

X_Vicon = np.array([x1_Vicon, x2_Vicon, x3_Vicon, x4_Vicon, x5_Vicon])
X_Vicon[:, :, 0] *= (-1)
X_Vicon[:, :, 1] *= (-1)
X_Vicon_all = np.array([x1_Vicon, x2_Vicon, x3_Vicon, x4_Vicon, x5_Vicon, x6_Vicon, x7_Vicon])
X_Vicon_all[:, :, 0] *= (-1)
X_Vicon_all[:, :, 1] *= (-1)

A_Vicon = np.array([a1_Vicon, a2_Vicon, a3_Vicon, a4_Vicon, a5_Vicon])
A_Vicon[:, :, 0] *= (-1)
A_Vicon[:, :, 1] *= (-1)
A_Vicon_all = np.array([a1_Vicon, a2_Vicon, a3_Vicon, a4_Vicon, a5_Vicon, a6_Vicon, a7_Vicon])
A_Vicon_all[:, :, 0] *= (-1)
A_Vicon_all[:, :, 1] *= (-1)

return t_markers, fs_markers, X_Vicon, X_Vicon_all, A_Vicon, A_Vicon_all, traj_markers

```

```

# Participant 0
t_traj_initial, fs_traj, X_Vicon_initial, X_Vicon_all_initial, A_Vicon_initial, A_Vicon_all_initial, traj_m

# Participant 1
t1_traj_initial, fs1_traj, X1_Vicon_initial, X1_Vicon_all_initial, A1_Vicon_initial, A1_Vicon_all_initial,

# Participant 2
t2_traj_initial, fs2_traj, X2_Vicon_initial, X2_Vicon_all_initial, A2_Vicon_initial, A2_Vicon_all_initial,

# Participant 3
t3_8_traj_initial, fs3_8_traj, X3_8_Vicon_initial, X3_8_Vicon_all_initial, A3_8_Vicon_initial, A3_8_Vicon_a
t3_10_traj_initial, fs3_10_traj, X3_10_Vicon_initial, X3_10_Vicon_all_initial, A3_10_Vicon_initial, A3_10_V
t3_12_traj_initial, fs3_12_traj, X3_12_Vicon_initial, X3_12_Vicon_all_initial, A3_12_Vicon_initial, A3_12_V
t3_jump_traj_initial, fs3_jump_traj, X3_jump_Vicon_initial, X3_jump_Vicon_all_initial, A3_jump_Vicon_initia

# Participant 4
t4_8_traj_initial, fs4_8_traj, X4_8_Vicon_initial, X4_8_Vicon_all_initial, A4_8_Vicon_initial, A4_8_Vicon_a
t4_10_traj_initial, fs4_10_traj, X4_10_Vicon_initial, X4_10_Vicon_all_initial, A4_10_Vicon_initial, A4_10_V
t4_12_traj_initial, fs4_12_traj, X4_12_Vicon_initial, X4_12_Vicon_all_initial, A4_12_Vicon_initial, A4_12_V
t4_jump_traj_initial, fs4_jump_traj, X4_jump_Vicon_initial, X4_jump_Vicon_all_initial, A4_jump_Vicon_initia

# Participant 5
t5_8_traj_initial, fs5_8_traj, X5_8_Vicon_initial, X5_8_Vicon_all_initial, A5_8_Vicon_initial, A5_8_Vicon_a
t5_10_traj_initial, fs5_10_traj, X5_10_Vicon_initial, X5_10_Vicon_all_initial, A5_10_Vicon_initial, A5_10_V
t5_12_traj_initial, fs5_12_traj, X5_12_Vicon_initial, X5_12_Vicon_all_initial, A5_12_Vicon_initial, A5_12_V
t5_jump_traj_initial, fs5_jump_traj, X5_jump_Vicon_initial, X5_jump_Vicon_all_initial, A5_jump_Vicon_initia

```

```
# Participant 6
t6_8_traj_initial, fs6_8_traj, X6_8_Vicon_initial, X6_8_Vicon_all_initial, A6_8_Vicon_initial, A6_8_Vicon_a
t6_10_traj_initial, fs6_10_traj, X6_10_Vicon_initial, X6_10_Vicon_all_initial, A6_10_Vicon_initial, A6_10_V
t6_12_traj_initial, fs6_12_traj, X6_12_Vicon_initial, X6_12_Vicon_all_initial, A6_12_Vicon_initial, A6_12_V
t6_jump_traj_initial, fs6_jump_traj, X6_jump_Vicon_initial, X6_jump_Vicon_all_initial, A6_jump_Vicon_initia
```

```
# Participant 7
t7_8_traj_initial, fs7_8_traj, X7_8_Vicon_initial, X7_8_Vicon_all_initial, A7_8_Vicon_initial, A7_8_Vicon_a
t7_10_traj_initial, fs7_10_traj, X7_10_Vicon_initial, X7_10_Vicon_all_initial, A7_10_Vicon_initial, A7_10_V
t7_12_traj_initial, fs7_12_traj, X7_12_Vicon_initial, X7_12_Vicon_all_initial, A7_12_Vicon_initial, A7_12_V
t7_jump_traj_initial, fs7_jump_traj, X7_jump_Vicon_initial, X7_jump_Vicon_all_initial, A7_jump_Vicon_initia
```

```
# Posing 1
```

```
def AngularVelocityFromMarker(dt_Vicon, traj_markers, AM):
    # Convert positions to velocities, in order to compare IMU and Vicon

    LASI, RASI, LPSI, RPSI, LTHI, RTHI, LKNE, RKNE, LTIB, RTIB, LANK, RANK, LHEE, RHEE, LTOE, RTOE = traj_m

    # right thigh
    v1_Vicon = VelocityFromMarker(RTHI, dt_Vicon)

    # left shank
    v2_Vicon = VelocityFromMarker(LTIB, dt_Vicon)

    # left thigh
    v3_Vicon = VelocityFromMarker(LTHI, dt_Vicon)

    # right shank
    v4_Vicon = VelocityFromMarker(RTIB, dt_Vicon)

    # pelvis
    marker_pelvis = (LPSI + RPSI)/2
    v5_Vicon = VelocityFromMarker(marker_pelvis, dt_Vicon)

    # left foot
    marker_foot_left = (LTOE + LHEE + LANK) /3
    v6_Vicon = VelocityFromMarker(marker_foot_left, dt_Vicon)

    # right foot
    marker_foot_right = (RTOE + RHEE + RANK) /3
    v7_Vicon = VelocityFromMarker(marker_foot_right, dt_Vicon)

    # Treadmill and markers should have recorded the same timespan
    V_Vicon = np.array([v1_Vicon, v2_Vicon, v3_Vicon, v4_Vicon, v5_Vicon])
    V_Vicon[:, :, 0] *= (-1)
    V_Vicon[:, :, 1] *= (-1)
    V_Vicon_all = np.array([v1_Vicon, v2_Vicon, v3_Vicon, v4_Vicon, v5_Vicon, v6_Vicon, v7_Vicon])
    V_Vicon_all[:, :, 0] *= (-1)
    V_Vicon_all[:, :, 1] *= (-1)

    # Define a matrix with the radius of the circle that the object is moving along
    # TO DO: improve by measuring the exact lengths on the pants
    # I assume now that the IMU is located in the middle of the body segment, on the lateral side

    # Based on the accelerations and for a right-handed coordinate system (see Vicon Nexus in the lab):
    # x-axis to the right: medio-lateral or width
    # y-axis backward: anterior-posterior or depth
    # z-axis downward: longitudinal or length

    # How to define different rotation axes:
    # rotating around x-axis needs distance in y- or z-direction
    # rotating around y-axis needs distance in x- or z-direction
    # rotating round z-axis needs distance in x- or y-direction
```

```

# thigh
rx_t_left = np.abs(LTHI[:, 0] - LASI[:, 0])
ry_t_left = np.abs(LTHI[:, 1] - LASI[:, 1])
rz_t_left = np.abs(LTHI[:, 2] - LASI[:, 2])
r_t_left = np.array([rx_t_left, ry_t_left, rz_t_left])

rx_t_right = np.abs(RTHI[:, 0] - RASI[:, 0])
ry_t_right = np.abs(RTHI[:, 1] - RASI[:, 1])
rz_t_right = np.abs(RTHI[:, 2] - RASI[:, 2])
r_t_right = np.array([rx_t_right, ry_t_right, rz_t_right])

# shank
rx_s_left = np.abs(LTIB[:, 0] - LKNE[:, 0]) # distance from shank marker to knee marker (in each direc
ry_s_left = np.abs(LTIB[:, 1] - LKNE[:, 1])
rz_s_left = np.abs(LTIB[:, 2] - LKNE[:, 2])
r_s_left = np.array([rx_s_left, ry_s_left, rz_s_left])

rx_s_right = np.abs(RTIB[:, 0] - RKNE[:, 0])
ry_s_right = np.abs(RTIB[:, 1] - RKNE[:, 1])
rz_s_right = np.abs(RTIB[:, 2] - RKNE[:, 2])
r_s_right = np.array([rx_s_right, ry_s_right, rz_s_right])

# pelvis
THI = (LTHI + RTHI)/2
rx_p = np.abs(THI[:, 0] - marker_pelvis[:, 0])
ry_p = np.abs(THI[:, 1] - marker_pelvis[:, 1])
rz_p = np.abs(THI[:, 2] - marker_pelvis[:, 2])
r_p = np.array([rx_p, ry_p, rz_p])

# foot # distance from foot marker to ankle marker
rx_f_left = np.abs(marker_foot_left[:, 0] - LANK[:, 0])
ry_f_left = np.abs(marker_foot_left[:, 1] - LANK[:, 1])
rz_f_left = np.abs(marker_foot_left[:, 2] - LANK[:, 2])
r_f_left = np.array([rx_f_left, ry_f_left, rz_f_left])

rx_f_right = np.abs(marker_foot_right[:, 0] - RANK[:, 0])
ry_f_right = np.abs(marker_foot_right[:, 1] - RANK[:, 1])
rz_f_right = np.abs(marker_foot_right[:, 2] - RANK[:, 2])
r_f_right = np.array([rx_f_right, ry_f_right, rz_f_right])

R_Vicon = np.array([r_t_right.T, r_s_left.T, r_t_left.T, r_s_right.T, r_p.T])
R_Vicon_all = np.array([r_t_right.T, r_s_left.T, r_t_left.T, r_s_right.T, r_p.T, r_f_left.T, r_f_right.

W_Vicon = np.zeros_like(V_Vicon)
for i in range(V_Vicon.shape[0]):
    W_Vicon[i, :, :] = np.cross(R_Vicon[i, :, :], V_Vicon[i, :, :])

W_Vicon_all = np.zeros_like(V_Vicon_all)
for i in range(V_Vicon_all.shape[0]):
    W_Vicon_all[i, :, :] = np.cross(V_Vicon_all[i, :, :], R_Vicon_all[i, :, :])

return V_Vicon, V_Vicon_all, W_Vicon, W_Vicon_all

```

```

def AngularVelocityFromMarker2(dt_Vicon, traj_markers, AM):
    # Convert positions to velocities, in order to compare IMU and Vicon

    LASI, RASI, LPSI, RPSI, LTHI, RTHI, LKNE, RKNE, LTIB, RTIB, LANK, RANK, LHEE, RHEE, LTOE, RTOE = traj_m

    # right thigh
    v1_Vicon = VelocityFromMarker(RTHI, dt_Vicon)

    # left shank
    v2_Vicon = VelocityFromMarker(LTIB, dt_Vicon)

```

```

# left thigh
v3_Vicon = VelocityFromMarker(LTHI, dt_Vicon)

# right shank
v4_Vicon = VelocityFromMarker(RTIB, dt_Vicon)

# pelvis
marker_pelvis = (LPSI + RPSI)/2
v5_Vicon = VelocityFromMarker(marker_pelvis, dt_Vicon)

# left foot
marker_foot_left = (LTOE + LHEE + LANK) /3
v6_Vicon = VelocityFromMarker(marker_foot_left, dt_Vicon)

# right foot
marker_foot_right = (RTOE + RHEE + RANK) /3
v7_Vicon = VelocityFromMarker(marker_foot_right, dt_Vicon)

# Treadmill and markers should have recorded the same timespan
V_Vicon = np.array([v1_Vicon, v2_Vicon, v3_Vicon, v4_Vicon, v5_Vicon])
V_Vicon[:, :, 0] *= (-1)
V_Vicon[:, :, 1] *= (-1)
V_Vicon_all = np.array([v1_Vicon, v2_Vicon, v3_Vicon, v4_Vicon, v5_Vicon, v6_Vicon, v7_Vicon])
V_Vicon_all[:, :, 0] *= (-1)
V_Vicon_all[:, :, 1] *= (-1)

# Define a matrix with the radius of the circle that the object is moving along
# TO DO: improve by measuring the exact lengths on the pants
# I assume now that the IMU is located in the middle of the body segment, on the lateral side

# Based on the accelerations and for a right-handed coordinate system (see Vicon Nexus in the lab):
# x_axis to the right: medio-lateral or width
# y-axis backward: anterior-posterior or depth
# z-axis downward: longitudinal or length

# How to define different rotation axes:
# rotating around x-axis needs distance in y- or z-direction
# rotating around y-axis needs distance in x- or z-direction
# rotating round z-axis needs distance in x- or y-direction

# thigh
rx_t_left = np.linalg.norm(LTHI[:, 1:2] - LKNE[:, 1:2]) # distance from thigh marker to pelvis marker
ry_t_left = np.linalg.norm(LTHI[:, 0:3:2] - LKNE[:, 0:3:2])
rz_t_left = np.linalg.norm(LTHI[:, 0:1] - LKNE[:, 0:1])

rx_t_right = np.linalg.norm(RTHI[:, 1:2] - RKNE[:, 1:2])
ry_t_right = np.linalg.norm(RTHI[:, 0:3:2] - RKNE[:, 0:3:2])
rz_t_right = np.linalg.norm(RTHI[:, 0:1] - RKNE[:, 0:1])

# rx_t_left = np.linalg.norm(LTHI[:, 1:2] - marker_pelvis[:, 1:2]) # distance from thigh marker to pel
# ry_t_left = np.linalg.norm(LTHI[:, 0:3:2] - marker_pelvis[:, 0:3:2])
# rz_t_left = np.linalg.norm(LTHI[:, 0:1] - marker_pelvis[:, 0:1])

# rx_t_right = np.linalg.norm(RTHI[:, 1:2] - marker_pelvis[:, 1:2])
# ry_t_right = np.linalg.norm(RTHI[:, 0:3:2] - marker_pelvis[:, 0:3:2])
# rz_t_right = np.linalg.norm(RTHI[:, 0:1] - marker_pelvis[:, 0:1])

# shank
rx_s_left = np.linalg.norm(LTIB[:, 1:2] - LANK[:, 1:2]) # distance from shank marker to knee marker (i
ry_s_left = np.linalg.norm(LTIB[:, 0:3:2] - LANK[:, 0:3:2])
rz_s_left = np.linalg.norm(LTIB[:, 0:1] - LANK[:, 0:1])

rx_s_right = np.linalg.norm(RTIB[:, 1:2] - RANK[:, 1:2])
ry_s_right = np.linalg.norm(RTIB[:, 0:3:2] - RANK[:, 0:3:2])
rz_s_right = np.linalg.norm(RTIB[:, 0:1] - RANK[:, 0:1])

```

```

# rx_s_left = np.linalg.norm(LTIB[:, 1:2] - LKNE[:, 1:2]) # distance from shank marker to knee marker
# ry_s_left = np.linalg.norm(LTIB[:, 0:3:2] - LKNE[:, 0:3:2])
# rz_s_left = np.linalg.norm(LTIB[:, 0:1] - LKNE[:, 0:1])

# rx_s_right = np.linalg.norm(RTIB[:, 1:2] - RKNE[:, 1:2])
# ry_s_right = np.linalg.norm(RTIB[:, 0:3:2] - RKNE[:, 0:3:2])
# rz_s_right = np.linalg.norm(RTIB[:, 0:1] - RKNE[:, 0:1])

# pelvis
THI = (LTHI + RTHI)/2
rx_p = np.linalg.norm(THI[:, 1:2] - marker_pelvis[:, 1:2])
ry_p = np.linalg.norm(THI[:, 0:3:2] - marker_pelvis[:, 0:3:2])
rz_p = np.linalg.norm(THI[:, 0:1] - marker_pelvis[:, 0:1])

# rx_p = AM[2, 1]/2
# ry_p = 1e-6
# rz_p = AM[2, 1]/2

# foot # distance from foot marker to ankle marker
rx_f_left = np.linalg.norm(marker_foot_left[:, 1:2] - LANK[:, 1:2])
ry_f_left = np.linalg.norm(marker_foot_left[:, 0:3:2] - LANK[:, 0:3:2])
rz_f_left = np.linalg.norm(marker_foot_left[:, 0:1] - LANK[:, 0:1])

rx_f_right = np.linalg.norm(marker_foot_right[:, 1:2] - RANK[:, 1:2])
ry_f_right = np.linalg.norm(marker_foot_right[:, 0:3:2] - RANK[:, 0:3:2])
rz_f_right = np.linalg.norm(marker_foot_right[:, 0:1] - RANK[:, 0:1])

R_Vicon = np.array([[rx_t_right, ry_t_right, rz_t_right], [rx_s_left, ry_s_left, rz_s_left], [rx_t_left,
R_Vicon_all = np.array([[rx_t_right, ry_t_right, rz_t_right], [rx_s_left, ry_s_left, rz_s_left], [rx_t_

W_Vicon = np.zeros_like(V_Vicon)
for i in range(V_Vicon.shape[0]):
    W_Vicon[i, :, :] = V_Vicon[i, :, :] / R_Vicon[i]

W_Vicon_all = np.zeros_like(V_Vicon_all)
for i in range(V_Vicon_all.shape[0]):
    W_Vicon_all[i, :, :] = V_Vicon_all[i, :, :] / R_Vicon_all[i]

return V_Vicon, V_Vicon_all, W_Vicon, W_Vicon_all

```

```

# Participant 0
V_Vicon_initial, V_Vicon_all_initial, W_Vicon_initial, W_Vicon_all_initial = AngularVelocityFromMarker(1/fs

# Participant 1
V1_Vicon_initial, V1_Vicon_all_initial, W1_Vicon_initial, W1_Vicon_all_initial = AngularVelocityFromMarker(

# Participant 2
V2_Vicon_initial, V2_Vicon_all_initial, W2_Vicon_initial, W2_Vicon_all_initial = AngularVelocityFromMarker(

# Participant 3
V3_8_Vicon_initial, V3_8_Vicon_all_initial, W3_8_Vicon_initial, W3_8_Vicon_all_initial = AngularVelocityFro
V3_10_Vicon_initial, V3_10_Vicon_all_initial, W3_10_Vicon_initial, W3_10_Vicon_all_initial = AngularVelocit
V3_12_Vicon_initial, V3_12_Vicon_all_initial, W3_12_Vicon_initial, W3_12_Vicon_all_initial = AngularVelocit
V3_jump_Vicon_initial, V3_jump_Vicon_all_initial, W3_jump_Vicon_initial, W3_jump_Vicon_all_initial = Angula

# Participant 4
V4_8_Vicon_initial, V4_8_Vicon_all_initial, W4_8_Vicon_initial, W4_8_Vicon_all_initial = AngularVelocityFro
V4_10_Vicon_initial, V4_10_Vicon_all_initial, W4_10_Vicon_initial, W4_10_Vicon_all_initial = AngularVelocit
V4_12_Vicon_initial, V4_12_Vicon_all_initial, W4_12_Vicon_initial, W4_12_Vicon_all_initial = AngularVelocit
V4_jump_Vicon_initial, V4_jump_Vicon_all_initial, W4_jump_Vicon_initial, W4_jump_Vicon_all_initial = Angula

# Participant 5
V5_8_Vicon_initial, V5_8_Vicon_all_initial, W5_8_Vicon_initial, W5_8_Vicon_all_initial = AngularVelocityFro
V5_10_Vicon_initial, V5_10_Vicon_all_initial, W5_10_Vicon_initial, W5_10_Vicon_all_initial = AngularVelocit

```

```
V5_12_Vicon_initial, V5_12_Vicon_all_initial, W5_12_Vicon_initial, W5_12_Vicon_all_initial = AngularVelocit
V5_jump_Vicon_initial, V5_jump_Vicon_all_initial, W5_jump_Vicon_initial, W5_jump_Vicon_all_initial = Angula
```

```
# Participant 6
```

```
V6_8_Vicon_initial, V6_8_Vicon_all_initial, W6_8_Vicon_initial, W6_8_Vicon_all_initial = AngularVelocityFro
V6_10_Vicon_initial, V6_10_Vicon_all_initial, W6_10_Vicon_initial, W6_10_Vicon_all_initial = AngularVelocit
V6_12_Vicon_initial, V6_12_Vicon_all_initial, W6_12_Vicon_initial, W6_12_Vicon_all_initial = AngularVelocit
V6_jump_Vicon_initial, V6_jump_Vicon_all_initial, W6_jump_Vicon_initial, W6_jump_Vicon_all_initial = Angula
```

```
# Participant 7
```

```
V7_8_Vicon_initial, V7_8_Vicon_all_initial, W7_8_Vicon_initial, W7_8_Vicon_all_initial = AngularVelocityFro
V7_10_Vicon_initial, V7_10_Vicon_all_initial, W7_10_Vicon_initial, W7_10_Vicon_all_initial = AngularVelocit
V7_12_Vicon_initial, V7_12_Vicon_all_initial, W7_12_Vicon_initial, W7_12_Vicon_all_initial = AngularVelocit
V7_jump_Vicon_initial, V7_jump_Vicon_all_initial, W7_jump_Vicon_initial, W7_jump_Vicon_all_initial = Angula
```

✓ 2.4 OpenSim

```
# Participant 0
```

```
df_OpenSimPos = pd.read_csv(OpenSimPos, names = ['time', 'femur_r_X', 'femur_r_Y', 'femur_r_Z', 'femur_r_Ox
```

```
df_OpenSimVelGF = pd.read_csv(OpenSimVelGF, names = ['time', 'femur_r_X', 'femur_r_Y', 'femur_r_Z', 'femur_
df_OpenSimVelBF = pd.read_csv(OpenSimVelBF, names = ['time', 'femur_r_X', 'femur_r_Y', 'femur_r_Z', 'femur_
```

```
df_OpenSimAccGF = pd.read_csv(OpenSimAccGF, names = ['time', 'pelvis_X', 'pelvis_Y', 'pelvis_Z', 'pelvis
df_OpenSimAccBF = pd.read_csv(OpenSimAccBF, names = ['time', 'pelvis_X', 'pelvis_Y', 'pelvis_Z', 'pelvis
```

```
# Let op: de volgorde van de header verandert in OpenSim, afhankelijk van hoe je de bodies ingeeft!
```

```
fs_OpenSim = int(1/(df_OpenSimPos["time"][1] - df_OpenSimPos["time"][0]))
```

```
# merk op: OpenSim heeft dezelfde sampling frequency als de markers
print(fs_OpenSim)
```

🔄 120

```
def ReadPosOpenSim(df_OpenSim, rot=False):
```

```
    t = df_OpenSim["time"] #time
    X = np.zeros((5, len(t), 3)) # positions
    R = np.zeros((5, len(t), 3)) # orientations
```

```
    X[0, :, :] = np.array(df_OpenSim[["femur_r_X", "femur_r_Y", "femur_r_Z"]].values)
    X[1, :, :] = np.array(df_OpenSim[["tibia_l_X", "tibia_l_Y", "tibia_l_Z"]].values)
    X[2, :, :] = np.array(df_OpenSim[["femur_l_X", "femur_l_Y", "femur_l_Z"]].values)
    X[3, :, :] = np.array(df_OpenSim[["tibia_r_X", "tibia_r_Y", "tibia_r_Z"]].values)
    X[4, :, :] = np.array(df_OpenSim[["pelvis_X", "pelvis_Y", "pelvis_Z"]].values)
```

```
    R[0, :, :] = np.array(df_OpenSim[["femur_r_Ox", "femur_r_Oy", "femur_r_Oz"]].values)
    R[1, :, :] = np.array(df_OpenSim[["tibia_l_Ox", "tibia_l_Oy", "tibia_l_Oz"]].values)
    R[2, :, :] = np.array(df_OpenSim[["femur_l_Ox", "femur_l_Oy", "femur_l_Oz"]].values)
    R[3, :, :] = np.array(df_OpenSim[["tibia_r_Ox", "tibia_r_Oy", "tibia_r_Oz"]].values)
    R[4, :, :] = np.array(df_OpenSim[["pelvis_Ox", "pelvis_Oy", "pelvis_Oz"]].values)
```

```
    R = np.deg2rad(R)
```

```
    if rot == True:
```

```
        X_rot = np.zeros((5, len(t), 3)) # positions
        R_rot = np.zeros((5, len(t), 3)) # orientations
```

```
        X_rot[:, :, 0] = X[:, :, 2]*(-1)
        X_rot[:, :, 1] = X[:, :, 0]*(-1)
```

```

X_rot[:, :, 2] = X[:, :, 1]

R_rot[:, :, 0] = R[:, :, 2]
R_rot[:, :, 1] = R[:, :, 0]
R_rot[:, :, 2] = R[:, :, 1]

return t, X_rot, R_rot

return t, X, R

```

```
t_OpenSim_initial, X_OpenSim_initial, R_OpenSim_initial = ReadPosOpenSim(df_OpenSimPos, rot=True)
```

```
def ReadVelOpenSim(df_OpenSim, rot=False):
```

```

t = df_OpenSim["time"] #time
V = np.zeros((5, len(t), 3)) # linear velocities
W = np.zeros((5, len(t), 3)) # angular velocities

V[0, :, :] = np.array(df_OpenSim[["femur_r_X", "femur_r_Y", "femur_r_Z"]].values)
V[1, :, :] = np.array(df_OpenSim[["tibia_l_X", "tibia_l_Y", "tibia_l_Z"]].values)
V[2, :, :] = np.array(df_OpenSim[["femur_l_X", "femur_l_Y", "femur_l_Z"]].values)
V[3, :, :] = np.array(df_OpenSim[["tibia_r_X", "tibia_r_Y", "tibia_r_Z"]].values)
V[4, :, :] = np.array(df_OpenSim[["pelvis_X", "pelvis_Y", "pelvis_Z"]].values)

W[0, :, :] = np.array(df_OpenSim[["femur_r_Ox", "femur_r_Oy", "femur_r_Oz"]].values)
W[1, :, :] = np.array(df_OpenSim[["tibia_l_Ox", "tibia_l_Oy", "tibia_l_Oz"]].values)
W[2, :, :] = np.array(df_OpenSim[["femur_l_Ox", "femur_l_Oy", "femur_l_Oz"]].values)
W[3, :, :] = np.array(df_OpenSim[["tibia_r_Ox", "tibia_r_Oy", "tibia_r_Oz"]].values)
W[4, :, :] = np.array(df_OpenSim[["pelvis_Ox", "pelvis_Oy", "pelvis_Oz"]].values)

W = np.deg2rad(W)

if rot == True:
    V_rot = np.zeros((5, len(t), 3)) # linear velocities
    W_rot = np.zeros((5, len(t), 3)) # angular velocities

    V_rot[:, :, 0] = V[:, :, 2]*(-1)
    V_rot[:, :, 1] = V[:, :, 0]*(-1)
    V_rot[:, :, 2] = V[:, :, 1]

    W_rot[:, :, 0] = W[:, :, 2]
    W_rot[:, :, 1] = W[:, :, 0]
    W_rot[:, :, 2] = W[:, :, 1]

    return t, V_rot, W_rot

return t, V, W

```

```

t_OpenSimGF_initial, V_OpenSimGF_initial, W_OpenSimGF_initial = ReadVelOpenSim(df_OpenSimVelGF, rot=True)
t_OpenSimBF_initial, V_OpenSimBF_initial, W_OpenSimBF_initial = ReadVelOpenSim(df_OpenSimVelBF, rot=True)

```

```
def ReadAccOpenSim(df_OpenSim, rot=False):
```

```

t = df_OpenSim["time"] #time
A = np.zeros((5, len(t), 3)) # (linear) accelerations
alpha = np.zeros((5, len(t), 3)) # (angular) accelerations

A[0, :, :] = np.array(df_OpenSim[["femur_r_X", "femur_r_Y", "femur_r_Z"]].values)
A[1, :, :] = np.array(df_OpenSim[["tibia_l_X", "tibia_l_Y", "tibia_l_Z"]].values)
A[2, :, :] = np.array(df_OpenSim[["femur_l_X", "femur_l_Y", "femur_l_Z"]].values)
A[3, :, :] = np.array(df_OpenSim[["tibia_r_X", "tibia_r_Y", "tibia_r_Z"]].values)
A[4, :, :] = np.array(df_OpenSim[["pelvis_X", "pelvis_Y", "pelvis_Z"]].values)

```

```
alpha[0, :, :] = np.array(df_OpenSim[["femur_r_Ox", "femur_r_Oy", "femur_r_Oz"]].values)
alpha[1, :, :] = np.array(df_OpenSim[["tibia_l_Ox", "tibia_l_Oy", "tibia_l_Oz"]].values)
alpha[2, :, :] = np.array(df_OpenSim[["femur_l_Ox", "femur_l_Oy", "femur_l_Oz"]].values)
alpha[3, :, :] = np.array(df_OpenSim[["tibia_r_Ox", "tibia_r_Oy", "tibia_r_Oz"]].values)
alpha[4, :, :] = np.array(df_OpenSim[["pelvis_Ox", "pelvis_Oy", "pelvis_Oz"]].values)
```

```
alpha = np.deg2rad(alpha)
```

```
if rot == True:
```

```
    A_rot = np.zeros((5, len(t), 3)) # (linear) accelerations
    alpha_rot = np.zeros((5, len(t), 3)) # (angular) accelerations
```

```
    A_rot[:, :, 0] = A[:, :, 2]*(-1)
    A_rot[:, :, 1] = A[:, :, 0]*(-1)
    A_rot[:, :, 2] = A[:, :, 1]
```

```
    alpha_rot[:, :, 0] = alpha[:, :, 2]
    alpha_rot[:, :, 1] = alpha[:, :, 0]
    alpha_rot[:, :, 2] = alpha[:, :, 1]
```

```
    return t, A_rot, alpha_rot
```

```
return t, A, alpha
```

```
t_OpenSimGF_initial, A_OpenSimGF_initial, alpha_OpenSimGF_initial = ReadAccOpenSim(df_OpenSimAccGF, rot=True)
t_OpenSimBF_initial, A_OpenSimBF_initial, alpha_OpenSimBF_initial = ReadAccOpenSim(df_OpenSimAccBF, rot=True)
```

✓ 3. Data cleaning: remove the time offsets from the data

```
def find_nearest(array, value):
    i = (np.abs(array - value)).argmin()
    return i
```

```
def SelectFrame(data, start_i, stop_i=None, err_i=0):

    """
    Select the time frame of interest from the data.
    - data: dataset
    - start_i: start of the time frame (in seconds)
    - stop_i: end of the time frame (in seconds). Default is the end of the dataset?
    - err_i: small correction index, that is added for a better synchronisation of two signals.
    """

    # time
    if len(data.shape)==1:
        if stop_i == None:
            stop_i = len(data)
        return data[start_i+err_i:stop_i+err_i]

    elif len(data.shape)==2:
        if stop_i == None:
            stop_i = data.shape[0]
        return data[start_i+err_i:stop_i+err_i, :]

    elif len(data.shape)==3:
        if stop_i == None:
            stop_i = data.shape[1]
        return data[:, start_i+err_i:stop_i+err_i, :]
```

```
def AlignViconAndIMU3(data1_initial, ViconStart_time, ViconEnd_time, data2_initial, IMUstart_time, IMUstop_time):
```

```

# Stel dat data1 en data2 numpy-arrays zijn met bijbehorende sampling rates
fs1 = 960 # Hz

# Adjust the arrays
if fs2 != '?':
    fs2 = fs2 # Hz

# Tijdvectoren
t1 = np.arange(len(data1_initial[:, 2])) / fs1
t2 = np.arange(len(data2_initial[0, :, 2])) / fs2

IMUstart_i = find_nearest(t2, IMUstart_time)
IMUend_i = find_nearest(t2, IMUend_time)+1 # slice goes until stop-1
ViconStart_i = find_nearest(t1, ViconStart_time)
ViconEnd_i = find_nearest(t1, ViconEnd_time)+1

# Select the frame of interest
data1 = SelectFrame(data1_initial, ViconStart_i, ViconEnd_i)
A = SelectFrame(A_initial, IMUstart_i, IMUend_i)
W = SelectFrame(W_initial, IMUstart_i, IMUend_i)
W_rot = SelectFrame(W_rot_initial, IMUstart_i, IMUend_i)
Q = SelectFrame(Q_initial, IMUstart_i, IMUend_i)
V = SelectFrame(V_initial, IMUstart_i, IMUend_i)
X = SelectFrame(X_initial, IMUstart_i, IMUend_i)
data2 = SelectFrame(data2_initial, IMUstart_i, IMUend_i)

# Kies een gemeenschappelijke samplingfrequentie, bijvoorbeeld:
fs_common = 960

# Resample beide datasets
data1 = FilterData(data1, fs1, 2, 30, 'lowpass', 0)
data1_resampled = scipy.signal.resample(data1, int(len(data1[:, 2]) * fs_common / fs1), axis=0)
data2 = FilterData(data2, fs2, 3, 6.5, 'lowpass', 1)
data2_resampled = scipy.signal.resample(data2, int(len(data2[0, :, 2]) * fs_common / fs2), axis=1)
A = FilterData(A, fs2, 3, 6.5, 'lowpass', 1)
A_resampled = scipy.signal.resample(A, int(len(data2[0, :, 2]) * fs_common / fs2), axis=1)
W = FilterData(W, fs2, 3, 6.5, 'lowpass', 1)
W_resampled = scipy.signal.resample(W, int(len(data2[0, :, 2]) * fs_common / fs2), axis=1)
W_rot = FilterData(W_rot, fs2, 3, 6.5, 'lowpass', 1)
W_rot_resampled = scipy.signal.resample(W_rot, int(len(data2[0, :, 2]) * fs_common / fs2), axis=1)
Q = FilterData(Q, fs2, 3, 6.5, 'lowpass', 1)
Q_resampled = scipy.signal.resample(Q, int(len(data2[0, :, 2]) * fs_common / fs2), axis=1)
V = FilterData(V, fs2, 3, 6.5, 'lowpass', 1)
V_resampled = scipy.signal.resample(V, int(len(data2[0, :, 2]) * fs_common / fs2), axis=1)
X = FilterData(X, fs2, 3, 6.5, 'lowpass', 1)
X_resampled = scipy.signal.resample(X, int(len(data2[0, :, 2]) * fs_common / fs2), axis=1)

# Bepaal het tijdsverschil met de cross-correlatie
data2_estimation = LinkSegmentModel(M, WF, data2_resampled)
correlation = scipy.signal.correlate(data1_resampled[:, 2], data2_estimation[:, 2], mode='full')
lags = scipy.signal.correlation_lags(len(data1_resampled[:, 2]), len(data2_resampled[0, :, 2]), m)
lag = lags[np.argmax(correlation)]

# Tijdvertraging berekenen
time_shift = lag / fs_common

if lag > 0:
    data1_sync = data1_resampled[lag:, :]
    data2_sync = data2_resampled[:, :len(data1_sync[:, 2]), :]
    A_sync = A_resampled[:, :len(data1_sync[:, 2]), :]
    W_rot_sync = W_rot_resampled[:, :len(data1_sync[:, 2]), :]
    W_sync = W_resampled[:, :len(data1_sync[:, 2]), :]
    Q_sync = Q_resampled[:, :len(data1_sync[:, 2]), :]
    V_sync = V_resampled[:, :len(data1_sync[:, 2]), :]
    X_sync = X_resampled[:, :len(data1_sync[:, 2]), :]

```

```

else:
    data2_sync = data2_resampled[:, -lag:, :]
    A_sync = A_resampled[:, -lag:, :]
    W_sync = W_resampled[:, -lag:, :]
    W_rot_sync = W_rot_resampled[:, -lag:, :]
    Q_sync = Q_resampled[:, -lag:, :]
    V_sync = V_resampled[:, -lag:, :]
    X_sync = X_resampled[:, -lag:, :]
    data1_sync = data1_resampled[:len(data2_sync[0, :, 2]), :]

r, RMSE, rRMSE = CalculateMetrics(LinkSegmentModel(M, WF, data2_sync), data1_sync, printer=True)
return data1_sync, data2_sync, A_sync, W_sync, W_rot_sync, Q_sync, V_sync, X_sync

# Detect problems with the sampling frequency
elif fs2 == '?':
    Fs2 = np.arange(98, 102, 0.1)
    correlation_best = [(0,0)]
    for fs2 in Fs2:
        print(fs2)

        # Tijdvectoren
        t1 = np.arange(0, len(data1_initial[:, 2])/fs1, 1/fs1)
        t2 = np.arange(0, len(data2_initial[0, :, 2])/fs2, 1/fs2)

        IMUstart_i = find_nearest(t2, IMUstart_time)
        IMUend_i = find_nearest(t2, IMUend_time)+1 # slice goes until stop-1
        ViconStart_i = find_nearest(t1, ViconStart_time)
        ViconEnd_i = find_nearest(t1, ViconEnd_time)+1

        # Select the frame of interest
        data1 = SelectFrame(data1_initial, ViconStart_i, ViconEnd_i)
        A = SelectFrame(A_initial, IMUstart_i, IMUend_i)
        W = SelectFrame(W_initial, IMUstart_i, IMUend_i)
        W_rot = SelectFrame(W_rot_initial, IMUstart_i, IMUend_i)
        Q = SelectFrame(Q_initial, IMUstart_i, IMUend_i)
        V = SelectFrame(V_initial, IMUstart_i, IMUend_i)
        X = SelectFrame(X_initial, IMUstart_i, IMUend_i)
        data2 = SelectFrame(data2_initial, IMUstart_i, IMUend_i)

        # Kies een gemeenschappelijke samplingfrequentie, bijvoorbeeld:
        fs_common = 960

        # Resample beide datasets
        data1 = FilterData(data1, fs1, 2, 30, 'lowpass', 0)
        data1_resampled = scipy.signal.resample(data1, int(len(data1[:, 2]) * fs_common / fs1), axis=
        data2 = FilterData(data2, fs2, 3, 6.5, 'lowpass', 1)
        data2_resampled = scipy.signal.resample(data2, int(len(data2[0, :, 2]) * fs_common / fs2), ax
        A_resampled = scipy.signal.resample(A, int(len(data2[0, :, 2]) * fs_common / fs2), axis=1)
        W_resampled = scipy.signal.resample(W, int(len(data2[0, :, 2]) * fs_common / fs2), axis=1)
        W_rot_resampled = scipy.signal.resample(W_rot, int(len(data2[0, :, 2]) * fs_common / fs2), ax
        Q_resampled = scipy.signal.resample(Q, int(len(data2[0, :, 2]) * fs_common / fs2), axis=1)
        V_resampled = scipy.signal.resample(V, int(len(data2[0, :, 2]) * fs_common / fs2), axis=1)
        X_resampled = scipy.signal.resample(X, int(len(data2[0, :, 2]) * fs_common / fs2), axis=1)

        # Bepaal het tijdsverschil met de cross-correlatie
        data2_estimation = LinkSegmentModel(M, WF, data2_resampled)
        correlation = scipy.signal.correlate(data1_resampled[:, 2], data2_estimation[:, 2], mode='ful
        if np.max(correlation) > correlation_best[-1][1]:
            correlation_best.append((fs2, np.max(correlation)))
        lags = scipy.signal.correlation_lags(len(data1_resampled[:, 2]), len(data2_resampled[0, :, 2]
        lag = lags[np.argmax(correlation)]

        # Tijdvertraging berekenen
        time_shift = lag / fs_common

```

```

    if lag > 0:
        data1_sync = data1_resampled[lag:, :]
        data2_sync = data2_resampled[:, :len(data1_sync[:, 2]), :]
        A_sync = A_resampled[:, :len(data1_sync[:, 2]), :]
        W_rot_sync = W_rot_resampled[:, :len(data1_sync[:, 2]), :]
        W_sync = W_resampled[:, :len(data1_sync[:, 2]), :]
        Q_sync = Q_resampled[:, :len(data1_sync[:, 2]), :]
        V_sync = V_resampled[:, :len(data1_sync[:, 2]), :]
        X_sync = X_resampled[:, :len(data1_sync[:, 2]), :]
    else:
        data2_sync = data2_resampled[:, -lag:, :]
        A_sync = A_resampled[:, -lag:, :]
        W_sync = W_resampled[:, -lag:, :]
        W_rot_sync = W_rot_resampled[:, -lag:, :]
        Q_sync = Q_resampled[:, -lag:, :]
        V_sync = V_resampled[:, -lag:, :]
        X_sync = X_resampled[:, -lag:, :]
        data1_sync = data1_resampled[:len(data2_sync[0, :, 2]), :]

    r, RMSE, rRMSE = CalculateMetrics(LinkSegmentModel(M, WF, data2_sync), data1_sync, printer=Tr

    return data1_sync, data2_sync, LinkSegmentModel(M, WF, data2_sync), correlation_best[-1]

```

```

# Make interactive plots (but doesn't work in Colab)

# from google.colab import output
# output.enable_custom_widget_manager()

# fig = FigureWidgetResampler(go.Figure())
# fig.add_trace(go.Scattergl(name='Az pelvis', showlegend=True), hf_x=t4_8_IMU_initial, hf_y=np.ascontiguousarray(Az_pelvis))
# fig

```

```

def SelectData(IMUstart_time, IMUend_time, ViconStart_time, ViconEnd_time, err_i_IMU, err_i_traj, t_IMU_initial, t_Vicon_initial, t_traj_initial, A_initial, A_rot_initial, V_initial, W_initial, W_rot_initial, Q_initial, X_initial, A_Vicon_all_initial, X_Vicon_all_initial, W_Vicon_all_initial, mGRF_left_initial):
    IMUstart_i = find_nearest(t_IMU_initial, IMUstart_time)
    ViconStart_i = find_nearest(t_Vicon_initial, ViconStart_time)
    IMUend_i = find_nearest(t_IMU_initial, IMUend_time)+1 # slice goes until stop-1
    ViconEnd_i = find_nearest(t_Vicon_initial, ViconEnd_time)+1
    TrajStart_i = find_nearest(t_traj_initial, ViconStart_time)
    TrajEnd_i = find_nearest(t_traj_initial, ViconEnd_time)+1

    t_IMU = SelectFrame(t_IMU_initial, IMUstart_i, IMUend_i, err_i = err_i_IMU)
    A = SelectFrame(A_initial, IMUstart_i, IMUend_i, err_i = err_i_IMU)
    A_rot = SelectFrame(A_rot_initial, IMUstart_i, IMUend_i, err_i = err_i_IMU)
    V = SelectFrame(V_initial, IMUstart_i, IMUend_i, err_i = err_i_IMU)
    W = SelectFrame(W_initial, IMUstart_i, IMUend_i, err_i = err_i_IMU)
    W_rot = SelectFrame(W_rot_initial, IMUstart_i, IMUend_i, err_i = err_i_IMU)
    Q = SelectFrame(Q_initial, IMUstart_i, IMUend_i, err_i = err_i_IMU)
    X = SelectFrame(X_initial, IMUstart_i, IMUend_i, err_i = err_i_IMU)

    t_Vicon = SelectFrame(t_Vicon_initial, ViconStart_i, ViconEnd_i)
    t_traj = SelectFrame(t_traj_initial, TrajStart_i, TrajEnd_i, err_i = err_i_traj)
    A_Vicon = SelectFrame(A_Vicon_all_initial, TrajStart_i, TrajEnd_i, err_i = err_i_traj)
    A_Vicon_all = SelectFrame(A_Vicon_all_initial, TrajStart_i, TrajEnd_i, err_i = err_i_traj)
    V_Vicon = SelectFrame(V_Vicon_initial, TrajStart_i, TrajEnd_i, err_i = err_i_traj)
    V_Vicon_all = SelectFrame(V_Vicon_all_initial, TrajStart_i, TrajEnd_i, err_i = err_i_traj)
    X_Vicon = SelectFrame(X_Vicon_initial, TrajStart_i, TrajEnd_i, err_i = err_i_traj)
    X_Vicon_all = SelectFrame(X_Vicon_all_initial, TrajStart_i, TrajEnd_i, err_i = err_i_traj)
    W_Vicon = SelectFrame(W_Vicon_initial, TrajStart_i, TrajEnd_i, err_i = err_i_traj)
    W_Vicon_all = SelectFrame(W_Vicon_all_initial, TrajStart_i, TrajEnd_i, err_i = err_i_traj)

    mGRF_left = SelectFrame(mGRF_left_initial, ViconStart_i, ViconEnd_i)

```

```

mGRF_right = SelectFrame(mGRF_right_initial, ViconStart_i, ViconEnd_i)
mGRF = SelectFrame(mGRF_initial, ViconStart_i, ViconEnd_i)
mGRM_left = SelectFrame(mGRM_left_initial, ViconStart_i, ViconEnd_i)
mGRM_right = SelectFrame(mGRM_right_initial, ViconStart_i, ViconEnd_i)
mGRM = SelectFrame(mGRM_initial, ViconStart_i, ViconEnd_i)
mCoP_left = SelectFrame(mCoP_left_initial, ViconStart_i, ViconEnd_i)
mCoP_right = SelectFrame(mCoP_right_initial, ViconStart_i, ViconEnd_i)
mCoP = SelectFrame(mCoP_initial, ViconStart_i, ViconEnd_i)

if OpenSim_data == True:
    t_OpenSim = SelectFrame(t_OpenSim_initial, TrajStart_i, TrajEnd_i, err_i = err_i_traj)
    X_OpenSim = SelectFrame(X_OpenSim_initial, TrajStart_i, TrajEnd_i, err_i = err_i_traj)
    V_OpenSimGF = SelectFrame(V_OpenSimGF_initial, TrajStart_i, TrajEnd_i, err_i = err_i_traj)
    V_OpenSimBF = SelectFrame(V_OpenSimBF_initial, TrajStart_i, TrajEnd_i, err_i = err_i_traj)
    W_OpenSimGF = SelectFrame(W_OpenSimGF_initial, TrajStart_i, TrajEnd_i, err_i = err_i_traj)
    W_OpenSimBF = SelectFrame(W_OpenSimBF_initial, TrajStart_i, TrajEnd_i, err_i = err_i_traj)
    A_OpenSimGF = SelectFrame(A_OpenSimGF_initial, TrajStart_i, TrajEnd_i, err_i = err_i_traj)
    A_OpenSimBF = SelectFrame(A_OpenSimBF_initial, TrajStart_i, TrajEnd_i, err_i = err_i_traj)
    alpha_OpenSimGF = SelectFrame(alpha_OpenSimGF_initial, TrajStart_i, TrajEnd_i, err_i = err_i_traj)
    alpha_OpenSimBF = SelectFrame(alpha_OpenSimBF_initial, TrajStart_i, TrajEnd_i, err_i = err_i_traj)
    return t_IMU, A, A_rot, V, W, W_rot, X, Q, t_Vicon, t_traj, A_Vicon, A_Vicon_all, W_Vicon, W_Vicon_all

return t_IMU, A, A_rot, V, W, W_rot, X, Q, t_Vicon, t_traj, A_Vicon, A_Vicon_all, V_Vicon, V_Vicon_all,

```

```

def AlignViconAndIMU(M, WF, A_rot_initial, A_Vicon_initial, mGRF_initial, t_Vicon_initial, ViconStart_time,
# optimise cross-correlation between IMUs and Vicon

```

```

E = np.arange(-50, 50, 1) # range of 1s (-0.5s, 0.5s) for IMUs and 0.85s (-0.42s, 0.42s) for markers
highest_r_IMU = [0]
highest_r_Vicon = [0]
highest_r_diff = [0]

ViconStart_i = find_nearest(t_Vicon_initial, ViconStart_time)
ViconEnd_i = find_nearest(t_Vicon_initial, ViconEnd_time)
IMUstart_i = find_nearest(t_IMU_initial, IMUstart_time)
IMUend_i = find_nearest(t_IMU_initial, IMUend_time)
TrajStart_i = find_nearest(t_traj_initial, ViconStart_time)
TrajEnd_i = find_nearest(t_traj_initial, ViconEnd_time)

fs_IMU = 1/(t_IMU_initial[1] - t_IMU_initial[0])
fs_Vicon = 1/(t_Vicon_initial[1] - t_Vicon_initial[0])
fs_traj = fs_Vicon/8

for err_i in E:
    print(err_i)

    # Select the frame of interest
    A = SelectFrame(A_rot_initial, IMUstart_i, IMUend_i, err_i)
    A_Vicon = SelectFrame(A_Vicon_initial, TrajStart_i, TrajEnd_i, err_i)
    mGRF = SelectFrame(mGRF_initial, ViconStart_i, ViconEnd_i)

    # Filter
    mGRFxyz_filtered = FilterData(mGRF, fs_Vicon, 2, 30, 'lowpass', 0)
    mGRFres_filtered = np.sqrt(mGRFxyz_filtered[:, 0]**2 + mGRFxyz_filtered[:, 1]**2 + mGRFxyz_filtered[:, 2]**2)
    mGRF_filtered = np.concatenate((mGRFxyz_filtered, np.expand_dims(mGRFres_filtered, axis=1)), axis=1)
    A_filtered = FilterData(A, fs_IMU, 3, 6.5, 'lowpass', 1)
    A_Vicon_filtered = FilterData(A_Vicon, fs_traj, 3, 6.5, 'lowpass', 1)

    # upsample the IMU and marker data to the treadmill data
    A_filtered = scipy.signal.resample(A_filtered, mGRF_filtered.shape[0], axis=1)
    A_Vicon_filtered = scipy.signal.resample(A_Vicon_filtered, mGRF_filtered.shape[0], axis=1)

    # estimate the GRF
    eGRF_IMU = LinkSegmentModel(M, WF, A_filtered)

```

```

eGRF_Vicon = LinkSegmentModel(M, WF, A_Vicon_filtered)

# compare with the mGRF
r_IMU, RMSE_IMU, rRMSE_IMU = CalculateMetrics(eGRF_IMU, mGRF_filtered)
print(r_IMU)
r_Vicon, RMSE_Vicon, rRMSE_Vicon = CalculateMetrics(eGRF_Vicon, mGRF_filtered)
print(r_Vicon)
r_diff, RMSE_diff, rRMSE_diff = CalculateMetrics(A_filtered[4, :, :], A_Vicon_filtered[4, :, :])
print(r_diff)
print('-'*50)

if r_IMU[2] > highest_r_IMU[-1]:
    highest_r_IMU.append(r_IMU[2])
    err_i_IMU_best = err_i

if r_Vicon[2] > highest_r_Vicon[-1]:
    highest_r_Vicon.append(r_Vicon[2])
    err_i_Vicon_best = err_i

if r_diff[2] > highest_r_diff[-1]:
    highest_r_diff.append(r_diff[2])
    err_i_diff_best = err_i

return err_i_IMU_best, highest_r_IMU[-1], err_i_Vicon_best, highest_r_Vicon[-1], err_i_diff_best, highe

```

```

# Participant 0

```

```

# The Vicon dataset (treadmill and markers) started 33.21 s later than the IMU dataset (accelerations)

```

```

IMUstart_time = 113 # op't zicht
ViconStart_time = 80.25833
IMUend_time = IMUstart_time + (t_traj_initial[-1] - ViconStart_time) # trajectories have the shortest record
IMUinterval_time = IMUend_time - IMUstart_time
ViconEnd_time = ViconStart_time + IMUinterval_time
# err_i_IMU, r_IMU, err_i_Vicon, r_Vicon, err_i_diff, r_diff = AlignViconAndIMU(M, WF_W, A_rot_initial, A_Vi
# print((err_i_IMU, r_IMU, err_i_Vicon, r_Vicon, err_i_diff, r_diff)) # 46, 0.99, -2, 0.98
err_i_IMU = 46
err_i_traj = -2

```

```

t_IMU, A, A_rot, V, W, W_rot, X, Q, t0_Vicon, t_traj, A_Vicon, A_Vicon_all, W_Vicon, W_Vicon_all, V_Vicon, V

```

```

# 7 km/h

```

```

IMUstart_time = 88+33.21
ViconStart_time = 88
ViconEnd_time = 148
running_period = ViconEnd_time - ViconStart_time
IMUend_time = IMUstart_time + running_period
# err_i_IMU, r_IMU, err_i_traj, r_traj, err_i_diff, r_diff = AlignViconAndIMU(M, WF_W, A_rot_initial, A_Vi
# print((err_i_IMU, r_IMU, err_i_traj, r_traj, err_i_diff, r_diff))
err_i_IMU = -1
err_i_traj = -1

```

```

t_7_IMU, A_7, A_7_rot, V_7, W_7, W_7_rot, X_7, Q_7, t0_7_Vicon, t_7_traj, A_7_Vicon, A_7_Vicon_all, W_7_Vicon

```

```

# 8 km/h

```

```

IMUstart_time = 158+33.21
ViconStart_time = 158
ViconEnd_time = 216
running_period = ViconEnd_time - ViconStart_time
IMUend_time = IMUstart_time + running_period
# err_i_IMU, r_IMU, err_i_Vicon, r_Vicon, err_i_diff, r_diff = AlignViconAndIMU(M, WF_W, A_rot_initial, A_Vi
# print((err_i_IMU, r_IMU, err_i_Vicon, r_Vicon, err_i_diff, r_diff))
err_i_IMU = -1
err_i_traj = -1

```

```

t_8_IMU, A_8, A_8_rot, V_8, W_8, W_8_rot, X_8, Q_8, t0_8_Vicon, t_8_traj, A_8_Vicon, A_8_Vicon_all, W_8_Vicon

# 9 km/h
IMUstart_time = 228+33.21
ViconStart_time = 228
ViconEnd_time = 288
running_period = ViconEnd_time - ViconStart_time
IMUend_time = IMUstart_time + running_period
# err_i_IMU, r_IMU, err_i_Vicon, r_Vicon, err_i_diff, r_diff = AlignViconAndIMU(M, WF_W, A_rot_initial, A_Vi
# print((err_i_IMU, r_IMU, err_i_Vicon, r_Vicon, err_i_diff, r_diff))
err_i_IMU = -1
err_i_traj = -1

t_9_IMU, A_9, A_9_rot, V_9, W_9, W_9_rot, X_9, Q_9, t0_9_Vicon, t_9_traj, A_9_Vicon, A_9_Vicon_all, W_9_Vicon

# 10 km/h
IMUstart_time = 298+33.21
ViconStart_time = 298
ViconEnd_time = 358
running_period = ViconEnd_time - ViconStart_time
IMUend_time = IMUstart_time + running_period
# err_i_IMU, r_IMU, err_i_Vicon, r_Vicon, err_i_diff, r_diff = AlignViconAndIMU(M, WF_W, A_rot_initial, A_Vi
# print((err_i_IMU, r_IMU, err_i_Vicon, r_Vicon, err_i_diff, r_diff))
err_i_IMU = -1
err_i_traj = -1

t_10_IMU, A_10, A_10_rot, V_10, W_10, W_10_rot, X_10, Q_10, t0_10_Vicon, t_10_traj, A_10_Vicon, A_10_Vicon_all, W_10_Vicon

# TO DO: check waarom de opgenomen tijd van de trajectories korter is dan van de treadmill?

# 11 km/h
# IMUstart_time = 366+33.21
# ViconStart_time = 366
# ViconEnd_time = 426
# running_period = ViconEnd_time - ViconStart_time
# IMUend_time = IMUstart_time + running_period
# print(IMUend_time)
# err_i_IMU, r_IMU, err_i_Vicon, r_Vicon, err_i_diff, r_diff = AlignViconAndIMU(M, WF_W, A_rot_initial, A_Vi
# print((err_i_IMU, r_IMU, err_i_Vicon, r_Vicon, err_i_diff, r_diff))
# err_i_IMU = 0
# err_i_traj = 0

# t_11_IMU, A_11, A_11_rot, V_11, W_11, W_11_rot, X_11, Q_11, t_11_Vicon, t_11_traj, A_11_Vicon, A_11_Vicon_all, W_11_Vicon

# 12 km/h
# IMUstart_time = 438+33.21
# ViconStart_time = 438
# ViconEnd_time = 499
# running_period = ViconEnd_time - ViconStart_time
# IMUend_time = IMUstart_time + running_period
# err_i_IMU, r_IMU, err_i_Vicon, r_Vicon, err_i_diff, r_diff = AlignViconAndIMU(M, WF_W, A_rot_initial, A_Vi
# print((err_i_IMU, r_IMU, err_i_Vicon, r_Vicon, err_i_diff, r_diff))
# err_i_IMU = 0
# err_i_traj = 0

# t_12_IMU, A_12, A_12_rot, V_12, W_12, W_12_rot, X_12, Q_12, t_12_Vicon, t_12_traj, A_12_Vicon, A_12_Vicon_all, W_12_Vicon

# Participant 2

# start running at 8 km/h (see metadata2.txt)
# Vicon2Start_time = 56 # seconds

i_max_IMU = np.argmax(A2_rot_initial[4, :, 2])
IMUstart_time = t2_IMU_initial[i_max_IMU]
i_max_Vicon = np.argmax(mGRF_initial2[17000:20000, 2])

```

```

ViconStart_time = t_Vicon2_initial[17000+i_max_Vicon]
diff = IMUstart_time - ViconStart_time
ViconEnd_time = 258 # 4min18
running_period = ViconEnd_time - ViconStart_time
IMUend_time = IMUstart_time + running_period
err_i_IMU = -23
err_i_traj = -1

t2_IMU, A2, A2_rot, V2, W2, W2_rot, X2, Q2, t2_Vicon, t2_traj, A2_Vicon, A2_Vicon_all, V2_Vicon, V2_Vicon_a

# 8 km/h
# The Vicon dataset (treadmill and markers) started 37.1 s later than the IMU dataset (accelerations)
IMUstart_time = 56+diff
ViconStart_time = 56
ViconEnd_time = 56+60
running_period = ViconEnd_time - ViconStart_time
IMUend_time = IMUstart_time + running_period
err_i_IMU = 0
err_i_traj = -1

t2_8_IMU, A2_8, A2_8_rot, V2_8, W2_8, W2_8_rot, X2_8, Q2_8, t2_8_Vicon, t2_8_traj, A2_8_Vicon, A2_8_Vicon_a

# 10 km/h
# The Vicon dataset (treadmill and markers) started 26.6 s later than the IMU dataset (accelerations)
IMUstart_time = 126+diff
ViconStart_time = 126
ViconEnd_time = 126+60
running_period = ViconEnd_time - ViconStart_time
IMUend_time = IMUstart_time + running_period
err_i_traj = -1

t2_10_IMU, A2_10, A2_10_rot, V2_10, W2_10, W2_10_rot, X2_10, Q2_10, t2_10_Vicon, t2_10_traj, A2_10_Vicon, A

# 12 km/h
# The Vicon dataset (treadmill and markers) started 35.1 s later than the IMU dataset (accelerations)
IMUstart_time = 198+diff
ViconStart_time = 198
ViconEnd_time = 198+60
running_period = ViconEnd_time - ViconStart_time
IMUend_time = IMUstart_time + running_period
err_i_IMU = -24
err_i_traj = -1

t2_12_IMU, A2_12, A2_12_rot, V2_12, W2_12, W2_12_rot, X2_12, Q2_12, t2_12_Vicon, t2_12_traj, A2_12_Vicon, A

# jump
IMUstart_time = 22.42
ViconStart_time = 22.42-diff
ViconEnd_time = 31.59-diff
running_period = ViconEnd_time - ViconStart_time
IMUend_time = IMUstart_time + running_period
err_i_IMU = -1
err_i_traj = 0
t2_jump_IMU, A2_jump, A2_jump_rot, V2_jump, W2_jump, W2_jump_rot, X2_jump, Q2_jump, t2_jump_Vicon, t2_jump_

# Participant 3

# 8 km/h
# The Vicon dataset (treadmill and markers) started 37.1 s later than the IMU dataset (accelerations)
IMUstart_time = 83.2 # assuming that time starts from zero!
ViconStart_time = 46.09375
ViconEnd_time = 133.4979
running_period = ViconEnd_time - ViconStart_time
IMUend_time = IMUstart_time + running_period
# err_i_IMU, r_IMU, err_i_traj, r_traj, err_i_diff, r_diff = AlignViconAndIMU(M, WF_W, A3_8_rot_initial, A3

```

```

# print((err_i_IMU, r_IMU, err_i_traj, r_traj, err_i_diff, r_diff))
err_i_IMU = -7
err_i_traj = -1

t3_8_IMU, A3_8, A3_8_rot, V3_8, W3_8, W3_8_rot, X3_8, Q3_8, t3_8_Vicon, t3_8_traj, A3_8_Vicon, A3_8_Vicon_a

# 10 km/h
# The Vicon dataset (treadmill and markers) started 26.6 s later than the IMU dataset (accelerations)
# Problem: the treadmill has not recorded for a few seconds!
# Idee: knip deze seconden er later uit?
IMUstart_time = 70.32
ViconStart_time = 43.7218
ViconEnd_time = 154.8042
running_period = ViconEnd_time - ViconStart_time
IMUend_time = IMUstart_time + running_period
# err_i_IMU, r_IMU, err_i_traj, r_traj, err_i_diff, r_diff = AlignViconAndIMU(M, WF_W, A3_10_rot_initial, A
# print((err_i_IMU, r_IMU, err_i_traj, r_traj, err_i_diff, r_diff))
err_i_IMU = -39
err_i_traj = -48
t3_10_IMU, A3_10, A3_10_rot, V3_10, W3_10, W3_10_rot, X3_10, Q3_10, t3_10_Vicon, t3_10_traj, A3_10_Vicon, A

# 12 km/h
# The Vicon dataset (treadmill and markers) started 35.1 s later than the IMU dataset (accelerations)
# Problem: the treadmill has not recorded for a few seconds!
IMUstart_time = 74.66
ViconStart_time = 39.55625
ViconEnd_time = 181.8094
running_period = ViconEnd_time - ViconStart_time
IMUend_time = IMUstart_time + running_period
# err_i_IMU, r_IMU, err_i_traj, r_traj, err_i_diff, r_diff = AlignViconAndIMU(M, WF_W, A3_12_rot_initial, A
# print((err_i_IMU, r_IMU, err_i_traj, r_traj, err_i_diff, r_diff))
err_i_IMU = -4
err_i_traj = -1
t3_12_IMU, A3_12, A3_12_rot, V3_12, W3_12, W3_12_rot, X3_12, Q3_12, t3_12_Vicon, t3_12_traj, A3_12_Vicon, A

# jump
# Problem: the treadmill has not recorded for a few seconds!
IMUstart_time = 25.02
ViconStart_time = 4.866667
ViconEnd_time = 32.90729
running_period = ViconEnd_time - ViconStart_time
IMUend_time = IMUstart_time + running_period
# err_i_IMU, r_IMU, err_i_traj, r_traj, err_i_diff, r_diff = AlignViconAndIMU(M, WF_W, A3_jump_rot_initial,
# print((err_i_IMU, r_IMU, err_i_traj, r_traj, err_i_diff, r_diff))
err_i_IMU = 6
err_i_traj = 2
t3_jump_IMU, A3_jump, A3_jump_rot, V3_jump, W3_jump, W3_jump_rot, X3_jump, Q3_jump, t3_jump_Vicon, t3_jump_

```

```

# Participant 4

# 8 km/h
# The Vicon dataset (treadmill and markers) started 25.85 s later than the IMU dataset (accelerations)
IMUstart_time = 70.59
ViconStart_time = 44.74167
ViconEnd_time = 144.8781
running_period = ViconEnd_time - ViconStart_time
IMUend_time = IMUstart_time + running_period
# err_i_IMU, r_IMU, err_i_traj, r_traj, err_i_diff, r_diff = AlignViconAndIMU(M, WF_W, A4_8_rot_initial, A4
# print((err_i_IMU, r_IMU, err_i_traj, r_traj, err_i_diff, r_diff))
err_i_IMU = -40
err_i_traj = -47
t4_8_IMU, A4_8, A4_8_rot, V4_8, W4_8, W4_8_rot, X4_8, Q4_8, t4_8_Vicon, t4_8_traj, A4_8_Vicon, A4_8_Vicon_a

# 10 km/h
# The Vicon dataset (treadmill and markers) started 13.15 s later than the IMU dataset (accelerations)

```

```

IMUstart_time = 48.75
ViconStart_time = 35.59896
ViconEnd_time = 153.7625
running_period = ViconEnd_time - ViconStart_time
IMUend_time = IMUstart_time + running_period
# err_i_IMU, r_IMU, err_i_traj, r_traj, err_i_diff, r_diff = AlignViconAndIMU(M, WF_W, A4_10_rot_initial, A
# print((err_i_IMU, r_IMU, err_i_traj, r_traj, err_i_diff, r_diff))
err_i_IMU = -39
err_i_traj = -46
t4_10_IMU, A4_10, A4_10_rot, V4_10, W4_10, W4_10_rot, X4_10, Q4_10, t4_10_Vicon, t4_10_traj, A4_10_Vicon, A

# 12 km/h
# The Vicon dataset (treadmill and markers) started 57.30 s later than the IMU dataset (accelerations)
IMUstart_time = 102.36
ViconStart_time = 45.05938
ViconEnd_time = 165.4771
running_period = ViconEnd_time - ViconStart_time
IMUend_time = IMUstart_time + running_period
# err_i_IMU, r_IMU, err_i_traj, r_traj, err_i_diff, r_diff = AlignViconAndIMU(M, WF_W, A4_12_rot_initial, A
# print((err_i_IMU, r_IMU, err_i_traj, r_traj, err_i_diff, r_diff))
err_i_IMU = -1
err_i_traj = -1
t4_12_IMU, A4_12, A4_12_rot, V4_12, W4_12, W4_12_rot, X4_12, Q4_12, t4_12_Vicon, t4_12_traj, A4_12_Vicon, A

# jump
IMUstart_time = 12.7
ViconStart_time = 11.00729
ViconEnd_time = 42.99063
running_period = ViconEnd_time - ViconStart_time
IMUend_time = IMUstart_time + running_period
# err_i_IMU, r_IMU, err_i_traj, r_traj, err_i_diff, r_diff = AlignViconAndIMU(M, WF_W, A4_jump_rot_initial,
# print((err_i_IMU, r_IMU, err_i_traj, r_traj, err_i_diff, r_diff))
err_i_IMU = 13
err_i_traj = 0
t4_jump_IMU, A4_jump, A4_jump_rot, V4_jump, W4_jump, W4_jump_rot, X4_jump, Q4_jump, t4_jump_Vicon, t4_jump_

```

Participant 5

```

# 8 km/h
# The Vicon dataset (treadmill and markers) started 21.2 s later than the IMU dataset (accelerations)
IMUstart_time = 73.43
ViconStart_time = 52.23438
ViconEnd_time = 135.7906
running_period = ViconEnd_time - ViconStart_time
IMUend_time = IMUstart_time + running_period
# err_i_IMU, r_IMU, err_i_traj, r_traj, err_i_diff, r_diff = AlignViconAndIMU(M, WF_W, A5_8_rot_initial, A5
# print((err_i_IMU, r_IMU, err_i_traj, r_traj, err_i_diff, r_diff))
err_i_IMU = -1
err_i_traj = -1
t5_8_IMU, A5_8, A5_8_rot, V5_8, W5_8, W5_8_rot, X5_8, Q5_8, t5_8_Vicon, t5_8_traj, A5_8_Vicon, A5_8_Vicon_a

# 10 km/h
# The Vicon dataset (treadmill and markers) started 32.50 s later than the IMU dataset (accelerations)
IMUstart_time = 79.19
ViconStart_time = 46.69271
ViconEnd_time = 195.31
running_period = ViconEnd_time - ViconStart_time
IMUend_time = IMUstart_time + running_period
# err_i_IMU, r_IMU, err_i_traj, r_traj, err_i_diff, r_diff = AlignViconAndIMU(M, WF_W, A5_10_rot_initial, A
# print((err_i_IMU, r_IMU, err_i_traj, r_traj, err_i_diff, r_diff))
err_i_IMU = -3
err_i_traj = -1
t5_10_IMU, A5_10, A5_10_rot, V5_10, W5_10, W5_10_rot, X5_10, Q5_10, t5_10_Vicon, t5_10_traj, A5_10_Vicon, A

# 12 km/h

```

```

# The Vicon dataset (treadmill and markers) started 30.75 s later than the IMU dataset (accelerations)
IMUstart_time = 74.92
ViconStart_time = 44.17083
ViconEnd_time = 152.7802
running_period = ViconEnd_time - ViconStart_time
IMUend_time = IMUstart_time + running_period
# err_i_IMU, r_IMU, err_i_traj, r_traj, err_i_diff, r_diff = AlignViconAndIMU(M, WF_W, A5_12_rot_initial, A
# print((err_i_IMU, r_IMU, err_i_traj, r_traj, err_i_diff, r_diff))
err_i_IMU = -2
err_i_traj = -1
t5_12_IMU, A5_12, A5_12_rot, V5_12, W5_12, W5_12_rot, X5_12, Q5_12, t5_12_Vicon, t5_12_traj, A5_12_Vicon, A

# jump
IMUstart_time = 20.51
ViconStart_time = 2.589583
ViconEnd_time = 10.99271
running_period = ViconEnd_time - ViconStart_time
IMUend_time = IMUstart_time + running_period
# err_i_IMU, r_IMU, err_i_traj, r_traj, err_i_diff, r_diff = AlignViconAndIMU(M, WF_W, A5_jump_rot_initial,
# print((err_i_IMU, r_IMU, err_i_traj, r_traj, err_i_diff, r_diff))
err_i_IMU = -4
err_i_traj = 0
t5_jump_IMU, A5_jump, A5_jump_rot, V5_jump, W5_jump, W5_jump_rot, X5_jump, Q5_jump, t5_jump_Vicon, t5_jump_

```

```

# Participant 6

```

```

# 8 km/h
# The Vicon dataset (treadmill and markers) started 36.15 s later than the IMU dataset (accelerations)
# Problem: missing values in recording of treadmill
# Therefore, lower accuracy than expected because of distortion by the linear interpolation
IMUstart_time = 77.70
ViconStart_time = 41.55417
ViconEnd_time = 119.3615
running_period = ViconEnd_time - ViconStart_time
IMUend_time = IMUstart_time + running_period
# err_i_IMU, r_IMU, err_i_traj, r_traj, err_i_diff, r_diff = AlignViconAndIMU(M, WF_W, A6_8_rot_initial, A6
# print((err_i_IMU, r_IMU, err_i_traj, r_traj, err_i_diff, r_diff))
err_i_IMU = 41
err_i_traj = 6
t6_8_IMU, A6_8, A6_8_rot, V6_8, W6_8, W6_8_rot, X6_8, Q6_8, t6_8_Vicon, t6_8_traj, A6_8_Vicon, A6_8_Vicon_a

```

```

# 10 km/h
# The Vicon dataset (treadmill and markers) started 29.35 s later than the IMU dataset (accelerations)
IMUstart_time = 71.48
ViconStart_time = 42.12917
ViconEnd_time = 142.251
running_period = ViconEnd_time - ViconStart_time
IMUend_time = IMUstart_time + running_period
# err_i_IMU, r_IMU, err_i_traj, r_traj, err_i_diff, r_diff = AlignViconAndIMU(M, WF_W, A6_10_rot_initial, A
# print((err_i_IMU, r_IMU, err_i_traj, r_traj, err_i_diff, r_diff))
err_i_IMU = -3
err_i_traj = -1
t6_10_IMU, A6_10, A6_10_rot, V6_10, W6_10, W6_10_rot, X6_10, Q6_10, t6_10_Vicon, t6_10_traj, A6_10_Vicon, A

```

```

# 12 km/h
# The Vicon dataset (treadmill and markers) started 53.95 s later than the IMU dataset (accelerations)
# Problem: treadmill lacks some values
IMUstart_time = 92.20
ViconStart_time = 38.24583
ViconEnd_time = 157.3375
running_period = ViconEnd_time - ViconStart_time
IMUend_time = IMUstart_time + running_period
# err_i_IMU, r_IMU, err_i_traj, r_traj, err_i_diff, r_diff = AlignViconAndIMU(M, WF_W, A6_12_rot_initial, A
# print((err_i_IMU, r_IMU, err_i_traj, r_traj, err_i_diff, r_diff))
err_i_IMU = 0

```

```

err_i_traj = 0
t6_12_IMU, A6_12, A6_12_rot, V6_12, W6_12, W6_12_rot, X6_12, Q6_12, t6_12_Vicon, t6_12_traj, A6_12_Vicon, A

# jump
IMUstart_time = 21.58
ViconStart_time = 11.9875
ViconEnd_time = 33.55417
running_period = ViconEnd_time - ViconStart_time
IMUend_time = IMUstart_time + running_period
# err_i_IMU, r_IMU, err_i_traj, r_traj, err_i_diff, r_diff = AlignViconAndIMU(M, WF_W, A6_jump_rot_initial,
# print((err_i_IMU, r_IMU, err_i_traj, r_traj, err_i_diff, r_diff))
err_i_IMU = 23 # Nog niet ideaal (rho=0.5)
err_i_traj = -1
t6_jump_IMU, A6_jump, A6_jump_rot, V6_jump, W6_jump, W6_jump_rot, X6_jump, Q6_jump, t6_jump_Vicon, t6_jump_

```

```

# Participant 7

# 8 km/h
# The Vicon dataset (treadmill and markers) started 17.35 s later than the IMU dataset (accelerations)
IMUstart_time = 58.24
ViconStart_time = 40.88854
ViconEnd_time = 130.8292
running_period = ViconEnd_time - ViconStart_time
IMUend_time = IMUstart_time + running_period
# err_i_IMU, r_IMU, err_i_traj, r_traj, err_i_diff, r_diff = AlignViconAndIMU(M, WF_W, A7_8_rot_initial, A7
# print((err_i_IMU, r_IMU, err_i_traj, r_traj, err_i_diff, r_diff))
err_i_IMU = -1
err_i_traj = -2
t7_8_IMU, A7_8, A7_8_rot, V7_8, W7_8, W7_8_rot, X7_8, Q7_8, t7_8_Vicon, t7_8_traj, A7_8_Vicon, A7_8_Vicon_a
# _, A7_8, A7_8_rot, W7_8, W7_8_rot, Q7_8, V7_8, X7_8 = AlignViconAndIMU3(mGRF_initial7_8, ViconStart_time,

# 10 km/h
# The Vicon dataset (treadmill and markers) started 28.20 s later than the IMU dataset (accelerations)
IMUstart_time = 71.45
ViconStart_time = 43.25
ViconEnd_time = 136.5792
running_period = ViconEnd_time - ViconStart_time
IMUend_time = IMUstart_time + running_period
# err_i_IMU, r_IMU, err_i_traj, r_traj, err_i_diff, r_diff = AlignViconAndIMU(M, WF_W, A7_10_rot_initial, A
# print((err_i_IMU, r_IMU, err_i_traj, r_traj, err_i_diff, r_diff))
err_i_IMU = -2
err_i_traj = -2
t7_10_IMU, A7_10, A7_10_rot, V7_10, W7_10, W7_10_rot, X7_10, Q7_10, t7_10_Vicon, t7_10_traj, A7_10_Vicon, A
# _, A7_10, A7_10_rot, W7_10, W7_10_rot, Q7_10, V7_10, X7_10 = AlignViconAndIMU3(mGRF_initial7_10, ViconSta

# 12 km/h
# The Vicon dataset (treadmill and markers) started 25.75 s later than the IMU dataset (accelerations)
IMUstart_time = 122.94
ViconStart_time = 48.63021
ViconEnd_time = 167.6219
running_period = ViconEnd_time - ViconStart_time
IMUend_time = IMUstart_time + running_period
# err_i_IMU, r_IMU, err_i_traj, r_traj, err_i_diff, r_diff = AlignViconAndIMU(M, WF_W, A7_12_rot_initial, A
err_i_IMU = -22
err_i_traj = -2
t7_12_IMU, A7_12, A7_12_rot, V7_12, W7_12, W7_12_rot, X7_12, Q7_12, t7_12_Vicon, t7_12_traj, A7_12_Vicon, A

# jump
IMUstart_time = 30.26
ViconStart_time = 2.417708
ViconEnd_time = 38.49167
running_period = ViconEnd_time - ViconStart_time
IMUend_time = IMUstart_time + running_period
# err_i_IMU, r_IMU, err_i_traj, r_traj, err_i_diff, r_diff = AlignViconAndIMU(M, WF_W, A7_jump_rot_initial,
# print((err_i_IMU, r_IMU, err_i_traj, r_traj, err_i_diff, r_diff))

```

```

err_i_IMU = -22
err_i_traj = -1

t7_jump_IMU, A7_jump, A7_jump_rot, V7_jump, W7_jump, W7_jump_rot, X7_jump, Q7_jump, t7_jump_Vicon, t7_jump_

# Note
# Negatieve err_i: curve schuift naar links
# Positieve err_i: curve schuift naar rechts

```

✓ 4. Filter, resample and plot the data

```

def PlotFrequencies(t, signal, fs):
    # Applying FFT
    fft_result = np.fft.fft(signal)
    freq = np.fft.fftfreq(t.shape[-1], d=1/fs) # frequency spectrum is symmetric
    start = int(len(freq)/2)

    # Plotting the spectrum
    plt.plot(freq[start:], np.abs(fft_result)[start:])
    plt.title('FFT of the signal')
    plt.xlabel('Frequency (Hz)')
    plt.ylabel('Amplitude')
    plt.show()

```

```

def FilterData(data, fs, N, fc, btype, filter_axis, plot=False, t=None, nr_frames=None):

    """
    Apply a Butterworth filter on a dataset.
    - data: data that you want to filter
    - fs: sampling frequency of the data
    - N: order of the butterworth filter
    - fc: critical frequency or frequencies
    - btype: type of the butterworth filter ('lowpass', 'highpass', 'bandpass').
    For example, a low-pass filter (LPF) only passes signals below its cutoff frequency while attenuating all
    - filter_axis: axis where the filter should be applied (accelerations: 1, mGRF: 0)
    """

    # filter
    # numerator ('b') and denominator ('a') polynomials of the IIR filter
    b, a = scipy.signal.butter(N, fc, btype, analog=False, fs=fs)
    # filtfilt: bidirectional filter, to have zero phase difference (zero lag)
    data_filtered = scipy.signal.filtfilt(b, a, data, axis=filter_axis)

    return data_filtered

```

```

def ResampleData(data, axis, new_len):

    # resample
    data_resampled = scipy.signal.resample(data, new_len, axis=axis)

    return data_resampled

```

```

def FilterAndResampleData(data, fs, N, fc, btype, axis, new_len):

    """
    Apply a Butterworth filter on a dataset. Afterwards, the data is resampled to the length of the mGRF.
    - data: data that you want to filter
    - fs: sampling frequency of the data
    - N: order of the butterworth filter
    - fc: critical frequency or frequencies

```

```

- btype: type of the butterworth filter ('lowpass', 'highpass', 'bandpass').
For example, a low-pass filter (LPF) only passes signals below its cutoff frequency while attenuating a
- filter_axis: axis where the filter should be applied (accelerations: 1, mGRF: 0)
- mGRF
"""

# filter
# numerator ('b') and denominator ('a') polynomials of the IIR filter
b, a = scipy.signal.butter(N, fc, btype, analog=False, fs=fs)
# filtfilt: bidirectional filter, to have zero phase difference (zero lag)
data_filtered = scipy.signal.filtfilt(b, a, data, axis=axis)

# resample
data_resampled = scipy.signal.resample(data_filtered, new_len, axis=axis)

return data_resampled

```

4.1 IMU

```

def CorrectPositionIMU(AM, X):
    X_corr = np.zeros_like(X)
    X_corr[0, :, :] = X[0, :, :] + AM[0, :]*np.ones_like(X[0, :, :])
    X_corr[1, :, :] = X[1, :, :] + AM[1, :]*np.ones_like(X[1, :, :])
    X_corr[2, :, :] = X[2, :, :] + AM[0, :]*np.ones_like(X[2, :, :])
    X_corr[3, :, :] = X[3, :, :] + AM[1, :]*np.ones_like(X[3, :, :])
    X_corr[4, :, :] = X[4, :, :] + AM[2, :]*np.ones_like(X[4, :, :])
    return X_corr

# Participant 0
A0_8_filtered = FilterAndResampleData(A_8_rot, fs_IMU, 3, 6.5, 'lowpass', 1, mGRF_8.shape[0])
V0_8_filtered = FilterAndResampleData(V_8, fs_IMU, 3, [1, 6.5], 'bandpass', 1, mGRF_8.shape[0])
W0_8_filtered = FilterAndResampleData(W_8, fs_IMU, 3, 6.5, 'lowpass', 1, mGRF_8.shape[0])
W0_8_rot_filtered = FilterAndResampleData(W_8_rot, fs_IMU, 3, 6.5, 'lowpass', 1, mGRF_8.shape[0])
X0_8_filtered = FilterAndResampleData(X_8, fs_IMU, 3, [1, 6.5], 'bandpass', 1, mGRF_8.shape[0])
X0_8_filtered_corr = CorrectPositionIMU(AM0, X0_8_filtered)

A0_10_filtered = FilterAndResampleData(A_10_rot, fs_IMU, 3, 6.5, 'lowpass', 1, mGRF_10.shape[0])
V0_10_filtered = FilterAndResampleData(V_10, fs_IMU, 3, [1, 6.5], 'bandpass', 1, mGRF_10.shape[0])
W0_10_filtered = FilterAndResampleData(W_10, fs_IMU, 3, 6.5, 'lowpass', 1, mGRF_10.shape[0])
W0_10_rot_filtered = FilterAndResampleData(W_10_rot, fs_IMU, 3, 6.5, 'lowpass', 1, mGRF_10.shape[0])
X0_10_filtered = FilterAndResampleData(X_10, fs_IMU, 3, [1, 6.5], 'bandpass', 1, mGRF_10.shape[0])
X0_10_filtered_corr = CorrectPositionIMU(AM0, X0_10_filtered)
Q0_10_filtered = ResampleData(Q_10, 1, mGRF.shape[0])

# Participant 2
A2_filtered = FilterAndResampleData(A2_rot, fs_IMU, 3, 6.5, 'lowpass', 1, mGRF2.shape[0])
V2_filtered = FilterAndResampleData(V2, fs_IMU, 3, [1, 6.5], 'bandpass', 1, mGRF2.shape[0])
W2_filtered = FilterAndResampleData(W2, fs_IMU, 3, 6.5, 'lowpass', 1, mGRF2.shape[0])
W2_rot_filtered = FilterAndResampleData(W2_rot, fs_IMU, 3, 6.5, 'lowpass', 1, mGRF2.shape[0])
X2_filtered = FilterAndResampleData(X2, fs_IMU, 3, [1, 6.5], 'bandpass', 1, mGRF2.shape[0])
X2_filtered_corr = CorrectPositionIMU(AM2, X2_filtered)

A2_8_filtered = FilterAndResampleData(A2_8_rot, fs_IMU, 3, 6.5, 'lowpass', 1, mGRF2_8.shape[0])
V2_8_filtered = FilterAndResampleData(V2_8, fs_IMU, 3, [1, 6.5], 'bandpass', 1, mGRF2_8.shape[0])
W2_8_filtered = FilterAndResampleData(W2_8, fs_IMU, 3, 6.5, 'lowpass', 1, mGRF2_8.shape[0])
W2_8_rot_filtered = FilterAndResampleData(W2_8_rot, fs_IMU, 3, 6.5, 'lowpass', 1, mGRF2_8.shape[0])
X2_8_filtered = FilterAndResampleData(X2_8, fs_IMU, 3, [1, 6.5], 'bandpass', 1, mGRF2_8.shape[0])
X2_8_filtered_corr = CorrectPositionIMU(AM2, X2_8_filtered)

A2_10_filtered = FilterAndResampleData(A2_10_rot, fs_IMU, 3, 6.5, 'lowpass', 1, mGRF2_10.shape[0])
V2_10_filtered = FilterAndResampleData(V2_10, fs_IMU, 3, [1, 6.5], 'bandpass', 1, mGRF2_10.shape[0])
W2_10_filtered = FilterAndResampleData(W2_10, fs_IMU, 3, 6.5, 'lowpass', 1, mGRF2_10.shape[0])
W2_10_rot_filtered = FilterAndResampleData(W2_10_rot, fs_IMU, 3, 6.5, 'lowpass', 1, mGRF2_10.shape[0])

```

```

X2_10_filtered = FilterAndResampleData(X2_10, fs_IMU, 3, [1, 6.5], 'bandpass', 1, mGRF2_10.shape[0])
X2_10_filtered_corr = CorrectPositionIMU(AM2, X2_10_filtered)

A2_12_filtered = FilterAndResampleData(A2_12_rot, fs_IMU, 3, 6.5, 'lowpass', 1, mGRF2_12.shape[0])
V2_12_filtered = FilterAndResampleData(V2_12, fs_IMU, 3, [1, 6.5], 'bandpass', 1, mGRF2_12.shape[0])
W2_12_filtered = FilterAndResampleData(W2_12, fs_IMU, 3, 6.5, 'lowpass', 1, mGRF2_12.shape[0])
W2_12_rot_filtered = FilterAndResampleData(W2_12_rot, fs_IMU, 3, 6.5, 'lowpass', 1, mGRF2_12.shape[0])
X2_12_filtered = FilterAndResampleData(X2_12, fs_IMU, 3, [1, 6.5], 'bandpass', 1, mGRF2_12.shape[0])
X2_12_filtered_corr = CorrectPositionIMU(AM2, X2_12_filtered)

A2_jump_filtered = FilterAndResampleData(A2_jump_rot, fs_IMU, 3, 6.5, 'lowpass', 1, mGRF2_jump.shape[0])
V2_jump_filtered = FilterAndResampleData(V2_jump, fs_IMU, 3, [1, 6.5], 'bandpass', 1, mGRF2_jump.shape[0])
W2_jump_filtered = FilterAndResampleData(W2_jump, fs_IMU, 3, 6.5, 'lowpass', 1, mGRF2_jump.shape[0])
W2_jump_rot_filtered = FilterAndResampleData(W2_jump_rot, fs_IMU, 3, 6.5, 'lowpass', 1, mGRF2_jump.shape[0])
X2_jump_filtered = FilterAndResampleData(X2_jump, fs_IMU, 3, [1, 6.5], 'bandpass', 1, mGRF2_jump.shape[0])
X2_jump_filtered_corr = CorrectPositionIMU(AM2, X2_jump_filtered)

# Participant 3
A3_8_filtered = FilterAndResampleData(A3_8_rot, fs_IMU, 3, 6.5, 'lowpass', 1, mGRF3_8.shape[0])
V3_8_filtered = FilterAndResampleData(V3_8, fs_IMU, 3, [1, 6.5], 'bandpass', 1, mGRF3_8.shape[0])
W3_8_filtered = FilterAndResampleData(W3_8, fs_IMU, 3, 6.5, 'lowpass', 1, mGRF3_8.shape[0])
W3_8_rot_filtered = FilterAndResampleData(W3_8_rot, fs_IMU, 3, 6.5, 'lowpass', 1, mGRF3_8.shape[0])
X3_8_filtered = FilterAndResampleData(X3_8, fs_IMU, 3, [1, 6.5], 'bandpass', 1, mGRF3_8.shape[0])
X3_8_filtered_corr = CorrectPositionIMU(AM3, X3_8_filtered)

A3_10_filtered = FilterAndResampleData(A3_10_rot, fs_IMU, 3, 6.5, 'lowpass', 1, mGRF3_10.shape[0])
V3_10_filtered = FilterAndResampleData(V3_10, fs_IMU, 3, [1, 6.5], 'bandpass', 1, mGRF3_10.shape[0])
W3_10_filtered = FilterAndResampleData(W3_10, fs_IMU, 3, 6.5, 'lowpass', 1, mGRF3_10.shape[0])
W3_10_rot_filtered = FilterAndResampleData(W3_10_rot, fs_IMU, 3, 6.5, 'lowpass', 1, mGRF3_10.shape[0])
X3_10_filtered = FilterAndResampleData(X3_10, fs_IMU, 3, [1, 6.5], 'bandpass', 1, mGRF3_10.shape[0])
X3_10_filtered_corr = CorrectPositionIMU(AM3, X3_10_filtered)

A3_12_filtered = FilterAndResampleData(A3_12_rot, fs_IMU, 3, 6.5, 'lowpass', 1, mGRF3_12.shape[0])
V3_12_filtered = FilterAndResampleData(V3_12, fs_IMU, 3, [1, 6.5], 'bandpass', 1, mGRF3_12.shape[0])
W3_12_filtered = FilterAndResampleData(W3_12, fs_IMU, 3, 6.5, 'lowpass', 1, mGRF3_12.shape[0])
W3_12_rot_filtered = FilterAndResampleData(W3_12_rot, fs_IMU, 3, 6.5, 'lowpass', 1, mGRF3_12.shape[0])
X3_12_filtered = FilterAndResampleData(X3_12, fs_IMU, 3, [1, 6.5], 'bandpass', 1, mGRF3_12.shape[0])
X3_12_filtered_corr = CorrectPositionIMU(AM3, X3_12_filtered)

A3_jump_filtered = FilterAndResampleData(A3_jump_rot, fs_IMU, 3, 6.5, 'lowpass', 1, mGRF3_jump.shape[0])
V3_jump_filtered = FilterAndResampleData(V3_jump, fs_IMU, 3, [1, 6.5], 'bandpass', 1, mGRF3_jump.shape[0])
W3_jump_filtered = FilterAndResampleData(W3_jump, fs_IMU, 3, 6.5, 'lowpass', 1, mGRF3_jump.shape[0])
W3_jump_rot_filtered = FilterAndResampleData(W3_jump_rot, fs_IMU, 3, 6.5, 'lowpass', 1, mGRF3_jump.shape[0])
X3_jump_filtered = FilterAndResampleData(X3_jump, fs_IMU, 3, [1, 6.5], 'bandpass', 1, mGRF3_jump.shape[0])
X3_jump_filtered_corr = CorrectPositionIMU(AM3, X3_jump_filtered)

A3_filtered = np.concatenate((A3_8_filtered, A3_10_filtered, A3_12_filtered), axis=1)
V3_filtered = np.concatenate((V3_8_filtered, V3_10_filtered, V3_12_filtered), axis=1)
X3_filtered = np.concatenate((X3_8_filtered, X3_10_filtered, X3_12_filtered), axis=1)
W3_filtered = np.concatenate((W3_8_filtered, W3_10_filtered, W3_12_filtered), axis=1)
W3_rot_filtered = np.concatenate((W3_8_rot_filtered, W3_10_rot_filtered, W3_12_rot_filtered), axis=1)

# Participant 4
A4_8_filtered = FilterAndResampleData(A4_8_rot, fs_IMU, 3, 6.5, 'lowpass', 1, mGRF4_8.shape[0])
V4_8_filtered = FilterAndResampleData(V4_8, fs_IMU, 3, [1, 6.5], 'bandpass', 1, mGRF4_8.shape[0])
W4_8_filtered = FilterAndResampleData(W4_8, fs_IMU, 3, 6.5, 'lowpass', 1, mGRF4_8.shape[0])
W4_8_rot_filtered = FilterAndResampleData(W4_8_rot, fs_IMU, 3, 6.5, 'lowpass', 1, mGRF4_8.shape[0])
X4_8_filtered = FilterAndResampleData(X4_8, fs_IMU, 3, [1, 6.5], 'bandpass', 1, mGRF4_8.shape[0])
X4_8_filtered_corr = CorrectPositionIMU(AM4, X4_8_filtered)

A4_10_filtered = FilterAndResampleData(A4_10_rot, fs_IMU, 3, 6.5, 'lowpass', 1, mGRF4_10.shape[0])
V4_10_filtered = FilterAndResampleData(V4_10, fs_IMU, 3, [1, 6.5], 'bandpass', 1, mGRF4_10.shape[0])
W4_10_filtered = FilterAndResampleData(W4_10, fs_IMU, 3, 6.5, 'lowpass', 1, mGRF4_10.shape[0])
W4_10_rot_filtered = FilterAndResampleData(W4_10_rot, fs_IMU, 3, 6.5, 'lowpass', 1, mGRF4_10.shape[0])
X4_10_filtered = FilterAndResampleData(X4_10, fs_IMU, 3, [1, 6.5], 'bandpass', 1, mGRF4_10.shape[0])

```

```

X4_10_filtered_corr = CorrectPositionIMU(AM4, X4_10_filtered)

A4_12_filtered = FilterAndResampleData(A4_12_rot, fs_IMU, 3, 6.5, 'lowpass', 1, mGRF4_12.shape[0])
V4_12_filtered = FilterAndResampleData(V4_12, fs_IMU, 3, [1, 6.5], 'bandpass', 1, mGRF4_12.shape[0])
W4_12_filtered = FilterAndResampleData(W4_12, fs_IMU, 3, 6.5, 'lowpass', 1, mGRF4_12.shape[0])
W4_12_rot_filtered = FilterAndResampleData(W4_12_rot, fs_IMU, 3, 6.5, 'lowpass', 1, mGRF4_12.shape[0])
X4_12_filtered = FilterAndResampleData(X4_12, fs_IMU, 3, [1, 6.5], 'bandpass', 1, mGRF4_12.shape[0])
X4_12_filtered_corr = CorrectPositionIMU(AM4, X4_12_filtered)

A4_jump_filtered = FilterAndResampleData(A4_jump_rot, fs_IMU, 3, 6.5, 'lowpass', 1, mGRF4_jump.shape[0])
V4_jump_filtered = FilterAndResampleData(V4_jump, fs_IMU, 3, [1, 6.5], 'bandpass', 1, mGRF4_jump.shape[0])
W4_jump_filtered = FilterAndResampleData(W4_jump, fs_IMU, 3, 6.5, 'lowpass', 1, mGRF4_jump.shape[0])
W4_jump_rot_filtered = FilterAndResampleData(W4_jump_rot, fs_IMU, 3, 6.5, 'lowpass', 1, mGRF4_jump.shape[0])
X4_jump_filtered = FilterAndResampleData(X4_jump, fs_IMU, 3, [1, 6.5], 'bandpass', 1, mGRF4_jump.shape[0])
X4_jump_filtered_corr = CorrectPositionIMU(AM4, X4_jump_filtered)

A4_filtered = np.concatenate((A4_8_filtered, A4_10_filtered, A4_12_filtered), axis=1)
V4_filtered = np.concatenate((V4_8_filtered, V4_10_filtered, V4_12_filtered), axis=1)
X4_filtered = np.concatenate((X4_8_filtered, X4_10_filtered, X4_12_filtered), axis=1)
W4_filtered = np.concatenate((W4_8_filtered, W4_10_filtered, W4_12_filtered), axis=1)
W4_rot_filtered = np.concatenate((W4_8_rot_filtered, W4_10_rot_filtered, W4_12_rot_filtered), axis=1)

# Participant 5
A5_8_filtered = FilterAndResampleData(A5_8_rot, fs_IMU, 3, 6.5, 'lowpass', 1, mGRF5_8.shape[0])
V5_8_filtered = FilterAndResampleData(V5_8, fs_IMU, 3, [1, 6.5], 'bandpass', 1, mGRF5_8.shape[0])
W5_8_filtered = FilterAndResampleData(W5_8, fs_IMU, 3, 6.5, 'lowpass', 1, mGRF5_8.shape[0])
W5_8_rot_filtered = FilterAndResampleData(W5_8_rot, fs_IMU, 3, 6.5, 'lowpass', 1, mGRF5_8.shape[0])
X5_8_filtered = FilterAndResampleData(X5_8, fs_IMU, 3, [1, 6.5], 'bandpass', 1, mGRF5_8.shape[0])
X5_8_filtered_corr = CorrectPositionIMU(AM5, X5_8_filtered)

A5_10_filtered = FilterAndResampleData(A5_10_rot, fs_IMU, 3, 6.5, 'lowpass', 1, mGRF5_10.shape[0])
V5_10_filtered = FilterAndResampleData(V5_10, fs_IMU, 3, [1, 6.5], 'bandpass', 1, mGRF5_10.shape[0])
W5_10_filtered = FilterAndResampleData(W5_10, fs_IMU, 3, 6.5, 'lowpass', 1, mGRF5_10.shape[0])
W5_10_rot_filtered = FilterAndResampleData(W5_10_rot, fs_IMU, 3, 6.5, 'lowpass', 1, mGRF5_10.shape[0])
X5_10_filtered = FilterAndResampleData(X5_10, fs_IMU, 3, [1, 6.5], 'bandpass', 1, mGRF5_10.shape[0])
X5_10_filtered_corr = CorrectPositionIMU(AM5, X5_10_filtered)

A5_12_filtered = FilterAndResampleData(A5_12_rot, fs_IMU, 3, 6.5, 'lowpass', 1, mGRF5_12.shape[0])
V5_12_filtered = FilterAndResampleData(V5_12, fs_IMU, 3, [1, 6.5], 'bandpass', 1, mGRF5_12.shape[0])
W5_12_filtered = FilterAndResampleData(W5_12, fs_IMU, 3, 6.5, 'lowpass', 1, mGRF5_12.shape[0])
W5_12_rot_filtered = FilterAndResampleData(W5_12_rot, fs_IMU, 3, 6.5, 'lowpass', 1, mGRF5_12.shape[0])
X5_12_filtered = FilterAndResampleData(X5_12, fs_IMU, 3, [1, 6.5], 'bandpass', 1, mGRF5_12.shape[0])
X5_12_filtered_corr = CorrectPositionIMU(AM5, X5_12_filtered)

A5_jump_filtered = FilterAndResampleData(A5_jump_rot, fs_IMU, 3, 6.5, 'lowpass', 1, mGRF5_jump.shape[0])
V5_jump_filtered = FilterAndResampleData(V5_jump, fs_IMU, 3, [1, 6.5], 'bandpass', 1, mGRF5_jump.shape[0])
W5_jump_filtered = FilterAndResampleData(W5_jump, fs_IMU, 3, 6.5, 'lowpass', 1, mGRF5_jump.shape[0])
W5_jump_rot_filtered = FilterAndResampleData(W5_jump_rot, fs_IMU, 3, 6.5, 'lowpass', 1, mGRF5_jump.shape[0])
X5_jump_filtered = FilterAndResampleData(X5_jump, fs_IMU, 3, [1, 6.5], 'bandpass', 1, mGRF5_jump.shape[0])
X5_jump_filtered_corr = CorrectPositionIMU(AM5, X5_jump_filtered)

A5_filtered = np.concatenate((A5_8_filtered, A5_10_filtered, A5_12_filtered), axis=1)
V5_filtered = np.concatenate((V5_8_filtered, V5_10_filtered, V5_12_filtered), axis=1)
X5_filtered = np.concatenate((X5_8_filtered, X5_10_filtered, X5_12_filtered), axis=1)
W5_filtered = np.concatenate((W5_8_filtered, W5_10_filtered, W5_12_filtered), axis=1)
W5_rot_filtered = np.concatenate((W5_8_rot_filtered, W5_10_rot_filtered, W5_12_rot_filtered), axis=1)

# Participant 6
A6_8_filtered = FilterAndResampleData(A6_8_rot, fs_IMU, 3, 6.5, 'lowpass', 1, mGRF6_8.shape[0])
V6_8_filtered = FilterAndResampleData(V6_8, fs_IMU, 3, [1, 6.5], 'bandpass', 1, mGRF6_8.shape[0])
W6_8_filtered = FilterAndResampleData(W6_8, fs_IMU, 3, 6.5, 'lowpass', 1, mGRF6_8.shape[0])
W6_8_rot_filtered = FilterAndResampleData(W6_8_rot, fs_IMU, 3, 6.5, 'lowpass', 1, mGRF6_8.shape[0])
X6_8_filtered = FilterAndResampleData(X6_8, fs_IMU, 3, [1, 6.5], 'bandpass', 1, mGRF6_8.shape[0])
X6_8_filtered_corr = CorrectPositionIMU(AM6, X6_8_filtered)

```

```

A6_10_filtered = FilterAndResampleData(A6_10_rot, fs_IMU, 3, 6.5, 'lowpass', 1, mGRF6_10.shape[0])
V6_10_filtered = FilterAndResampleData(V6_10, fs_IMU, 3, [1, 6.5], 'bandpass', 1, mGRF6_10.shape[0])
W6_10_filtered = FilterAndResampleData(W6_10, fs_IMU, 3, 6.5, 'lowpass', 1, mGRF6_10.shape[0])
W6_10_rot_filtered = FilterAndResampleData(W6_10_rot, fs_IMU, 3, 6.5, 'lowpass', 1, mGRF6_10.shape[0])
X6_10_filtered = FilterAndResampleData(X6_10, fs_IMU, 3, [1, 6.5], 'bandpass', 1, mGRF6_10.shape[0])
X6_10_filtered_corr = CorrectPositionIMU(AM6, X6_10_filtered)

A6_12_filtered = FilterAndResampleData(A6_12_rot, fs_IMU, 3, 6.5, 'lowpass', 1, mGRF6_12.shape[0])
V6_12_filtered = FilterAndResampleData(V6_12, fs_IMU, 3, [1, 6.5], 'bandpass', 1, mGRF6_12.shape[0])
W6_12_filtered = FilterAndResampleData(W6_12, fs_IMU, 3, 6.5, 'lowpass', 1, mGRF6_12.shape[0])
W6_12_rot_filtered = FilterAndResampleData(W6_12_rot, fs_IMU, 3, 6.5, 'lowpass', 1, mGRF6_12.shape[0])
X6_12_filtered = FilterAndResampleData(X6_12, fs_IMU, 3, [1, 6.5], 'bandpass', 1, mGRF6_12.shape[0])
X6_12_filtered_corr = CorrectPositionIMU(AM6, X6_12_filtered)

A6_jump_filtered = FilterAndResampleData(A6_jump_rot, fs_IMU, 3, 6.5, 'lowpass', 1, mGRF6_jump.shape[0])
V6_jump_filtered = FilterAndResampleData(V6_jump, fs_IMU, 3, [1, 6.5], 'bandpass', 1, mGRF6_jump.shape[0])
W6_jump_filtered = FilterAndResampleData(W6_jump, fs_IMU, 3, 6.5, 'lowpass', 1, mGRF6_jump.shape[0])
W6_jump_rot_filtered = FilterAndResampleData(W6_jump_rot, fs_IMU, 3, 6.5, 'lowpass', 1, mGRF6_jump.shape[0])
X6_jump_filtered = FilterAndResampleData(X6_jump, fs_IMU, 3, [1, 6.5], 'bandpass', 1, mGRF6_jump.shape[0])
X6_jump_filtered_corr = CorrectPositionIMU(AM6, X6_jump_filtered)

A6_filtered = np.concatenate((A6_8_filtered, A6_10_filtered, A6_12_filtered), axis=1)
V6_filtered = np.concatenate((V6_8_filtered, V6_10_filtered, V6_12_filtered), axis=1)
X6_filtered = np.concatenate((X6_8_filtered, X6_10_filtered, X6_12_filtered), axis=1)
W6_filtered = np.concatenate((W6_8_filtered, W6_10_filtered, W6_12_filtered), axis=1)
W6_rot_filtered = np.concatenate((W6_8_rot_filtered, W6_10_rot_filtered, W6_12_rot_filtered), axis=1)

# Participant 7
A7_8_filtered = FilterAndResampleData(A7_8_rot, fs_IMU, 3, 6.5, 'lowpass', 1, mGRF7_8.shape[0])
V7_8_filtered = FilterAndResampleData(V7_8, fs_IMU, 3, [1, 6.5], 'bandpass', 1, mGRF7_8.shape[0])
W7_8_filtered = FilterAndResampleData(W7_8, fs_IMU, 3, 6.5, 'lowpass', 1, mGRF7_8.shape[0])
W7_8_rot_filtered = FilterAndResampleData(W7_8_rot, fs_IMU, 3, 6.5, 'lowpass', 1, mGRF7_8.shape[0])
X7_8_filtered = FilterAndResampleData(X7_8, fs_IMU, 3, [1, 6.5], 'bandpass', 1, mGRF7_8.shape[0])
X7_8_filtered_corr = CorrectPositionIMU(AM7, X7_8_filtered)

A7_10_filtered = FilterAndResampleData(A7_10_rot, fs_IMU, 3, 6.5, 'lowpass', 1, mGRF7_10.shape[0])
V7_10_filtered = FilterAndResampleData(V7_10, fs_IMU, 3, [1, 6.5], 'bandpass', 1, mGRF7_10.shape[0])
W7_10_filtered = FilterAndResampleData(W7_10, fs_IMU, 3, 6.5, 'lowpass', 1, mGRF7_10.shape[0])
W7_10_rot_filtered = FilterAndResampleData(W7_10_rot, fs_IMU, 3, 6.5, 'lowpass', 1, mGRF7_10.shape[0])
X7_10_filtered = FilterAndResampleData(X7_10, fs_IMU, 3, [1, 6.5], 'bandpass', 1, mGRF7_10.shape[0])
X7_10_filtered_corr = CorrectPositionIMU(AM7, X7_10_filtered)

A7_12_filtered = FilterAndResampleData(A7_12_rot, fs_IMU, 3, 6.5, 'lowpass', 1, mGRF7_12.shape[0])
V7_12_filtered = FilterAndResampleData(V7_12, fs_IMU, 3, [1, 6.5], 'bandpass', 1, mGRF7_12.shape[0])
W7_12_filtered = FilterAndResampleData(W7_12, fs_IMU, 3, 6.5, 'lowpass', 1, mGRF7_12.shape[0])
W7_12_rot_filtered = FilterAndResampleData(W7_12_rot, fs_IMU, 3, 6.5, 'lowpass', 1, mGRF7_12.shape[0])
X7_12_filtered = FilterAndResampleData(X7_12, fs_IMU, 3, [1, 6.5], 'bandpass', 1, mGRF7_12.shape[0])
X7_12_filtered_corr = CorrectPositionIMU(AM7, X7_12_filtered)

A7_jump_filtered = FilterAndResampleData(A7_jump_rot, fs_IMU, 3, 6.5, 'lowpass', 1, mGRF7_jump.shape[0])
V7_jump_filtered = FilterAndResampleData(V7_jump, fs_IMU, 3, [1, 6.5], 'bandpass', 1, mGRF7_jump.shape[0])
W7_jump_filtered = FilterAndResampleData(W7_jump_rot, fs_IMU, 3, 6.5, 'lowpass', 1, mGRF7_jump.shape[0])
X7_jump_filtered = FilterAndResampleData(X7_jump, fs_IMU, 3, [1, 6.5], 'bandpass', 1, mGRF7_jump.shape[0])
X7_jump_filtered_corr = CorrectPositionIMU(AM7, X7_jump_filtered)

A7_filtered = np.concatenate((A7_8_filtered, A7_10_filtered, A7_12_filtered), axis=1)
V7_filtered = np.concatenate((V7_8_filtered, V7_10_filtered, V7_12_filtered), axis=1)
X7_filtered = np.concatenate((X7_8_filtered, X7_10_filtered, X7_12_filtered), axis=1)
W7_filtered = np.concatenate((W7_8_filtered, W7_10_filtered, W7_12_filtered), axis=1)
W7_rot_filtered = np.concatenate((W7_8_rot_filtered, W7_10_rot_filtered, W7_12_rot_filtered), axis=1)

```

```

# Participant 0

# 8 km/h
mGRFxyz_8_filtered = FilterData(mGRF_8, 960, 2, 30, 'lowpass', 0)
mGRFres_8_filtered = np.sqrt(np.sum(mGRFxyz_8_filtered**2, axis=1))
mGRF0_8_filtered = np.concatenate((mGRFxyz_8_filtered, np.expand_dims(mGRFres_8_filtered, axis=1)), axis=1)

mGRFxyz_8_filtered = FilterData(mGRF_left_8, 960, 2, 30, 'lowpass', 0)
mGRFres_8_filtered = np.sqrt(np.sum(mGRFxyz_8_filtered**2, axis=1))
mGRF_left0_8_filtered = np.concatenate((mGRFxyz_8_filtered, np.expand_dims(mGRFres_8_filtered, axis=1)), axis=1)

mGRFxyz_8_filtered = FilterData(mGRF_right_8, 960, 2, 30, 'lowpass', 0)
mGRFres_8_filtered = np.sqrt(np.sum(mGRFxyz_8_filtered**2, axis=1))
mGRF_right0_8_filtered = np.concatenate((mGRFxyz_8_filtered, np.expand_dims(mGRFres_8_filtered, axis=1)), axis=1)

mGRM0_8_filtered = FilterData(mGRM_8, fs_Vicon, 2, 30, 'lowpass', 0)

mCoP_left0_8_filtered = FilterData(mCoP_left_8, fs_Vicon, 2, 1, 'highpass', 0)
mCoP_right0_8_filtered = FilterData(mCoP_right_8, fs_Vicon, 2, 1, 'highpass', 0)

# 10 km/h
mGRFxyz_10_filtered = FilterData(mGRF_10, 960, 2, 30, 'lowpass', 0)
mGRFres_10_filtered = np.sqrt(np.sum(mGRFxyz_10_filtered**2, axis=1))
mGRF0_10_filtered = np.concatenate((mGRFxyz_10_filtered, np.expand_dims(mGRFres_10_filtered, axis=1)), axis=1)

mGRFxyz_10_filtered = FilterData(mGRF_left_10, 960, 2, 30, 'lowpass', 0)
mGRFres_10_filtered = np.sqrt(np.sum(mGRFxyz_10_filtered**2, axis=1))
mGRF_left0_10_filtered = np.concatenate((mGRFxyz_10_filtered, np.expand_dims(mGRFres_10_filtered, axis=1)), axis=1)

mGRFxyz_10_filtered = FilterData(mGRF_right_10, 960, 2, 30, 'lowpass', 0)
mGRFres_10_filtered = np.sqrt(np.sum(mGRFxyz_10_filtered**2, axis=1))
mGRF_right0_10_filtered = np.concatenate((mGRFxyz_10_filtered, np.expand_dims(mGRFres_10_filtered, axis=1)), axis=1)

mGRM0_10_filtered = FilterData(mGRM_10, fs_Vicon, 2, 30, 'lowpass', 0)

mCoP_left0_10_filtered = FilterData(mCoP_left_10, fs_Vicon, 2, 1, 'highpass', 0)
mCoP_right0_10_filtered = FilterData(mCoP_right_10, fs_Vicon, 2, 1, 'highpass', 0)

# all
mGRF0_filtered = np.concatenate((mGRF0_8_filtered, mGRF0_10_filtered), axis=0)
mGRF_left0_filtered = np.concatenate((mGRF_left0_8_filtered, mGRF_left0_10_filtered), axis=0)
mGRF_right0_filtered = np.concatenate((mGRF_right0_8_filtered, mGRF_right0_10_filtered), axis=0)

```

```

# Participant 2

# 8 km/h
mGRFxyz2_8_filtered = FilterData(mGRF2_8, 960, 2, 30, 'lowpass', 0)
mGRFres2_8_filtered = np.sqrt(np.sum(mGRFxyz2_8_filtered**2, axis=1))
mGRF2_8_filtered = np.concatenate((mGRFxyz2_8_filtered, np.expand_dims(mGRFres2_8_filtered, axis=1)), axis=1)

mGRFxyz_8_filtered = FilterData(mGRF_left2_8, 960, 2, 30, 'lowpass', 0)
mGRFres_8_filtered = np.sqrt(np.sum(mGRFxyz_8_filtered**2, axis=1))
mGRF_left2_8_filtered = np.concatenate((mGRFxyz_8_filtered, np.expand_dims(mGRFres_8_filtered, axis=1)), axis=1)

mGRFxyz_8_filtered = FilterData(mGRF_right2_8, 960, 2, 30, 'lowpass', 0)
mGRFres_8_filtered = np.sqrt(np.sum(mGRFxyz_8_filtered**2, axis=1))
mGRF_right2_8_filtered = np.concatenate((mGRFxyz_8_filtered, np.expand_dims(mGRFres_8_filtered, axis=1)), axis=1)

mGRM2_8_filtered = FilterData(mGRM_8, fs_Vicon, 2, 30, 'lowpass', 0)

mCoP_left2_8_filtered = FilterData(mCoP_left2_8, fs_Vicon, 2, 1, 'highpass', 0)
mCoP_right2_8_filtered = FilterData(mCoP_right2_8, fs_Vicon, 2, 1, 'highpass', 0)

# 10 km/h

```

```

mGRFxyz2_10_filtered = FilterData(mGRF2_10, 960, 2, 30, 'lowpass', 0)
mGRFres2_10_filtered = np.sqrt(np.sum(mGRFxyz2_10_filtered**2, axis=1))
mGRF2_10_filtered = np.concatenate((mGRFxyz2_10_filtered, np.expand_dims(mGRFres2_10_filtered, axis=1)), ax

mGRFxyz_10_filtered = FilterData(mGRF_left2_10, 960, 2, 30, 'lowpass', 0)
mGRFres_10_filtered = np.sqrt(np.sum(mGRFxyz_10_filtered**2, axis=1))
mGRF_left2_10_filtered = np.concatenate((mGRFxyz_10_filtered, np.expand_dims(mGRFres_10_filtered, axis=1)),

mGRFxyz_10_filtered = FilterData(mGRF_right2_10, 960, 2, 30, 'lowpass', 0)
mGRFres_10_filtered = np.sqrt(np.sum(mGRFxyz_10_filtered**2, axis=1))
mGRF_right2_10_filtered = np.concatenate((mGRFxyz_10_filtered, np.expand_dims(mGRFres_10_filtered, axis=1))

mGRM2_10_filtered = FilterData(mGRM2_10, fs_Vicon, 2, 30, 'lowpass', 0)

mCoP_left2_10_filtered = FilterData(mCoP_left2_10, fs_Vicon, 2, 1, 'highpass', 0)
mCoP_right2_10_filtered = FilterData(mCoP_right2_10, fs_Vicon, 2, 1, 'highpass', 0)

# 12 km/h
mGRFxyz2_12_filtered = FilterData(mGRF2_12, 960, 2, 30, 'lowpass', 0)
mGRFres2_12_filtered = np.sqrt(np.sum(mGRFxyz2_12_filtered**2, axis=1))
mGRF2_12_filtered = np.concatenate((mGRFxyz2_12_filtered, np.expand_dims(mGRFres2_12_filtered, axis=1)), ax

mGRFxyz_12_filtered = FilterData(mGRF_left2_12, 960, 2, 30, 'lowpass', 0)
mGRFres_12_filtered = np.sqrt(np.sum(mGRFxyz_12_filtered**2, axis=1))
mGRF_left2_12_filtered = np.concatenate((mGRFxyz_12_filtered, np.expand_dims(mGRFres_12_filtered, axis=1)),

mGRFxyz_12_filtered = FilterData(mGRF_right2_12, 960, 2, 30, 'lowpass', 0)
mGRFres_12_filtered = np.sqrt(np.sum(mGRFxyz_12_filtered**2, axis=1))
mGRF_right2_12_filtered = np.concatenate((mGRFxyz_12_filtered, np.expand_dims(mGRFres_12_filtered, axis=1))

mGRM2_12_filtered = FilterData(mGRM2_12, fs_Vicon, 2, 30, 'lowpass', 0)

mCoP_left2_12_filtered = FilterData(mCoP_left2_12, fs_Vicon, 2, 1, 'highpass', 0)
mCoP_right2_12_filtered = FilterData(mCoP_right2_12, fs_Vicon, 2, 1, 'highpass', 0)

# jump
mGRFxyz_jump_filtered = FilterData(mGRF_left2_jump, 960, 2, 30, 'lowpass', 0)
mGRFres_jump_filtered = np.sqrt(np.sum(mGRFxyz_jump_filtered**2, axis=1))
mGRF_left2_jump_filtered = np.concatenate((mGRFxyz_jump_filtered, np.expand_dims(mGRFres_jump_filtered, axi

mGRFxyz_jump_filtered = FilterData(mGRF_right2_jump, 960, 2, 30, 'lowpass', 0)
mGRFres_jump_filtered = np.sqrt(np.sum(mGRFxyz_jump_filtered**2, axis=1))
mGRF_right2_jump_filtered = np.concatenate((mGRFxyz_jump_filtered, np.expand_dims(mGRFres_jump_filtered, ax

mGRFxyz2_jump_filtered = FilterData(mGRF2_jump, 960, 2, 30, 'lowpass', 0)
mGRFres2_jump_filtered = np.sqrt(np.sum(mGRFxyz2_jump_filtered**2, axis=1))
mGRF2_jump_filtered = np.concatenate((mGRFxyz2_jump_filtered, np.expand_dims(mGRFres2_jump_filtered, axis=1

mGRFxyz_jump_filtered = FilterData(mGRF_left2_jump, 960, 2, 30, 'lowpass', 0)
mGRFres_jump_filtered = np.sqrt(np.sum(mGRFxyz_jump_filtered**2, axis=1))
mGRF_left2_jump_filtered = np.concatenate((mGRFxyz_jump_filtered, np.expand_dims(mGRFres_jump_filtered, axi

mGRFxyz_jump_filtered = FilterData(mGRF_right2_jump, 960, 2, 30, 'lowpass', 0)
mGRFres_jump_filtered = np.sqrt(np.sum(mGRFxyz_jump_filtered**2, axis=1))
mGRF_right2_jump_filtered = np.concatenate((mGRFxyz_jump_filtered, np.expand_dims(mGRFres_jump_filtered, ax

mGRM2_jump_filtered = FilterData(mGRM2_jump, fs_Vicon, 2, 30, 'lowpass', 0)

mCoP_left2_jump_filtered = FilterData(mCoP_left2_jump, fs_Vicon, 2, 1, 'highpass', 0)
mCoP_right2_jump_filtered = FilterData(mCoP_right2_jump, fs_Vicon, 2, 1, 'highpass', 0)

# all
mGRF2_filtered = np.concatenate((mGRF2_8_filtered, mGRF2_10_filtered, mGRF2_12_filtered), axis=0)
mGRF_left2_filtered = np.concatenate((mGRF_left2_8_filtered, mGRF_left2_10_filtered, mGRF_left2_12_filtered
mGRF_right2_filtered = np.concatenate((mGRF_right2_8_filtered, mGRF_right2_10_filtered, mGRF_right2_12_filt

```

```

# Participant 3

# 8 km/h
mGRFxyz3_8_filtered = FilterData(mGRF3_8, 960, 2, 30, 'lowpass', 0)
mGRFres3_8_filtered = np.sqrt(np.sum(mGRFxyz3_8_filtered**2, axis=1))
mGRF3_8_filtered = np.concatenate((mGRFxyz3_8_filtered, np.expand_dims(mGRFres3_8_filtered, axis=1)), axis=

mGRFxyz_8_filtered = FilterData(mGRF_left3_8, 960, 2, 30, 'lowpass', 0)
mGRFres_8_filtered = np.sqrt(np.sum(mGRFxyz_8_filtered**2, axis=1))
mGRF_left3_8_filtered = np.concatenate((mGRFxyz_8_filtered, np.expand_dims(mGRFres_8_filtered, axis=1)), ax

mGRFxyz_8_filtered = FilterData(mGRF_right3_8, 960, 2, 30, 'lowpass', 0)
mGRFres_8_filtered = np.sqrt(np.sum(mGRFxyz_8_filtered**2, axis=1))
mGRF_right3_8_filtered = np.concatenate((mGRFxyz_8_filtered, np.expand_dims(mGRFres_8_filtered, axis=1)), a

mGRM3_8_filtered = FilterData(mGRM3_8, fs_Vicon, 2, 30, 'lowpass', 0)

mCoP_left3_8_filtered = FilterData(mCoP_left3_8, fs_Vicon, 2, 1, 'highpass', 0)
mCoP_right3_8_filtered = FilterData(mCoP_right3_8, fs_Vicon, 2, 1, 'highpass', 0)

# 10 km/h
mGRFxyz3_10_filtered = FilterData(mGRF3_10, 960, 2, 30, 'lowpass', 0)
mGRFres3_10_filtered = np.sqrt(np.sum(mGRFxyz3_10_filtered**2, axis=1))
mGRF3_10_filtered = np.concatenate((mGRFxyz3_10_filtered, np.expand_dims(mGRFres3_10_filtered, axis=1)), ax

mGRFxyz_10_filtered = FilterData(mGRF_left3_10, 960, 2, 30, 'lowpass', 0)
mGRFres_10_filtered = np.sqrt(np.sum(mGRFxyz_10_filtered**2, axis=1))
mGRF_left3_10_filtered = np.concatenate((mGRFxyz_10_filtered, np.expand_dims(mGRFres_10_filtered, axis=1))),

mGRFxyz_10_filtered = FilterData(mGRF_right3_10, 960, 2, 30, 'lowpass', 0)
mGRFres_10_filtered = np.sqrt(np.sum(mGRFxyz_10_filtered**2, axis=1))
mGRF_right3_10_filtered = np.concatenate((mGRFxyz_10_filtered, np.expand_dims(mGRFres_10_filtered, axis=1)))

mGRM3_10_filtered = FilterData(mGRM3_10, fs_Vicon, 2, 30, 'lowpass', 0)

mCoP_left3_10_filtered = FilterData(mCoP_left3_10, fs_Vicon, 2, 1, 'highpass', 0)
mCoP_right3_10_filtered = FilterData(mCoP_right3_10, fs_Vicon, 2, 1, 'highpass', 0)

# 12 km/h
mGRFxyz3_12_filtered = FilterData(mGRF3_12, 960, 2, 30, 'lowpass', 0)
mGRFres3_12_filtered = np.sqrt(np.sum(mGRFxyz3_12_filtered**2, axis=1))
mGRF3_12_filtered = np.concatenate((mGRFxyz3_12_filtered, np.expand_dims(mGRFres3_12_filtered, axis=1)), ax

mGRFxyz_12_filtered = FilterData(mGRF_left3_12, 960, 2, 30, 'lowpass', 0)
mGRFres_12_filtered = np.sqrt(np.sum(mGRFxyz_12_filtered**2, axis=1))
mGRF_left3_12_filtered = np.concatenate((mGRFxyz_12_filtered, np.expand_dims(mGRFres_12_filtered, axis=1))),

mGRFxyz_12_filtered = FilterData(mGRF_right3_12, 960, 2, 30, 'lowpass', 0)
mGRFres_12_filtered = np.sqrt(np.sum(mGRFxyz_12_filtered**2, axis=1))
mGRF_right3_12_filtered = np.concatenate((mGRFxyz_12_filtered, np.expand_dims(mGRFres_12_filtered, axis=1)))

mGRM3_12_filtered = FilterData(mGRM3_12, fs_Vicon, 2, 30, 'lowpass', 0)

mCoP_left3_12_filtered = FilterData(mCoP_left3_12, fs_Vicon, 2, 1, 'highpass', 0)
mCoP_right3_12_filtered = FilterData(mCoP_right3_12, fs_Vicon, 2, 1, 'highpass', 0)

# jump
mGRFxyz3_jump_filtered = FilterData(mGRF3_jump, 960, 2, 30, 'lowpass', 0)
mGRFres3_jump_filtered = np.sqrt(np.sum(mGRFxyz3_jump_filtered**2, axis=1))
mGRF3_jump_filtered = np.concatenate((mGRFxyz3_jump_filtered, np.expand_dims(mGRFres3_jump_filtered, axis=1

mGRFxyz_jump_filtered = FilterData(mGRF_left3_jump, 960, 2, 30, 'lowpass', 0)
mGRFres_jump_filtered = np.sqrt(np.sum(mGRFxyz_jump_filtered**2, axis=1))
mGRF_left3_jump_filtered = np.concatenate((mGRFxyz_jump_filtered, np.expand_dims(mGRFres_jump_filtered, axi

```

```

mGRFxyz_jump_filtered = FilterData(mGRF_right3_jump, 960, 2, 30, 'lowpass', 0)
mGRFres_jump_filtered = np.sqrt(np.sum(mGRFxyz_jump_filtered**2, axis=1))
mGRF_right3_jump_filtered = np.concatenate((mGRFxyz_jump_filtered, np.expand_dims(mGRFres_jump_filtered, ax

mGRM3_jump_filtered = FilterData(mGRM3_jump, fs_Vicon, 2, 30, 'lowpass', 0)

mCoP_left3_jump_filtered = FilterData(mCoP_left3_jump, fs_Vicon, 2, 1, 'highpass', 0)
mCoP_right3_jump_filtered = FilterData(mCoP_right3_jump, fs_Vicon, 2, 1, 'highpass', 0)

# all
mGRF3_filtered = np.concatenate((mGRF3_8_filtered, mGRF3_10_filtered, mGRF3_12_filtered), axis=0)
mGRF_left3_filtered = np.concatenate((mGRF_left3_8_filtered, mGRF_left3_10_filtered, mGRF_left3_12_filtered
mGRF_right3_filtered = np.concatenate((mGRF_right3_8_filtered, mGRF_right3_10_filtered, mGRF_right3_12_filt

mGRM3_filtered = np.concatenate((mGRM3_8_filtered, mGRM3_10_filtered, mGRM3_12_filtered), axis=0)
mCoP_left3_filtered = np.concatenate((mCoP_left3_8_filtered, mCoP_left3_10_filtered, mCoP_left3_12_filtered
mCoP_right3_filtered = np.concatenate((mCoP_right3_8_filtered, mCoP_right3_10_filtered, mCoP_right3_12_filt

```

```

# Participant 4

```

```

# 8 km/h

```

```

mGRFxyz4_8_filtered = FilterData(mGRF4_8, 960, 2, 30, 'lowpass', 0)
mGRFres4_8_filtered = np.sqrt(np.sum(mGRFxyz4_8_filtered**2, axis=1))
mGRF4_8_filtered = np.concatenate((mGRFxyz4_8_filtered, np.expand_dims(mGRFres4_8_filtered, axis=1)), axis=

mGRFxyz_8_filtered = FilterData(mGRF_left4_8, 960, 2, 30, 'lowpass', 0)
mGRFres_8_filtered = np.sqrt(np.sum(mGRFxyz_8_filtered**2, axis=1))
mGRF_left4_8_filtered = np.concatenate((mGRFxyz_8_filtered, np.expand_dims(mGRFres_8_filtered, axis=1)), ax

mGRFxyz_8_filtered = FilterData(mGRF_right4_8, 960, 2, 30, 'lowpass', 0)
mGRFres_8_filtered = np.sqrt(np.sum(mGRFxyz_8_filtered**2, axis=1))
mGRF_right4_8_filtered = np.concatenate((mGRFxyz_8_filtered, np.expand_dims(mGRFres_8_filtered, axis=1)), a

mGRM4_8_filtered = FilterData(mGRM4_8, fs_Vicon, 2, 30, 'lowpass', 0)

mCoP_left4_8_filtered = FilterData(mCoP_left4_8, fs_Vicon, 2, 1, 'highpass', 0)
mCoP_right4_8_filtered = FilterData(mCoP_right4_8, fs_Vicon, 2, 1, 'highpass', 0)

```

```

# 10 km/h

```

```

mGRFxyz4_10_filtered = FilterData(mGRF4_10, 960, 2, 30, 'lowpass', 0)
mGRFres4_10_filtered = np.sqrt(np.sum(mGRFxyz4_10_filtered**2, axis=1))
mGRF4_10_filtered = np.concatenate((mGRFxyz4_10_filtered, np.expand_dims(mGRFres4_10_filtered, axis=1)), ax

mGRFxyz_10_filtered = FilterData(mGRF_left4_10, 960, 2, 30, 'lowpass', 0)
mGRFres_10_filtered = np.sqrt(np.sum(mGRFxyz_10_filtered**2, axis=1))
mGRF_left4_10_filtered = np.concatenate((mGRFxyz_10_filtered, np.expand_dims(mGRFres_10_filtered, axis=1))),

mGRFxyz_10_filtered = FilterData(mGRF_right4_10, 960, 2, 30, 'lowpass', 0)
mGRFres_10_filtered = np.sqrt(np.sum(mGRFxyz_10_filtered**2, axis=1))
mGRF_right4_10_filtered = np.concatenate((mGRFxyz_10_filtered, np.expand_dims(mGRFres_10_filtered, axis=1)))
mGRM4_10_filtered = FilterData(mGRM4_10, fs_Vicon, 2, 30, 'lowpass', 0)

mCoP_left4_10_filtered = FilterData(mCoP_left4_10, fs_Vicon, 2, 1, 'highpass', 0)
mCoP_right4_10_filtered = FilterData(mCoP_right4_10, fs_Vicon, 2, 1, 'highpass', 0)

```

```

# 12 km/h

```

```

mGRFxyz4_12_filtered = FilterData(mGRF4_12, 960, 2, 30, 'lowpass', 0)
mGRFres4_12_filtered = np.sqrt(np.sum(mGRFxyz4_12_filtered**2, axis=1))
mGRF4_12_filtered = np.concatenate((mGRFxyz4_12_filtered, np.expand_dims(mGRFres4_12_filtered, axis=1)), ax

mGRFxyz_12_filtered = FilterData(mGRF_left4_12, 960, 2, 30, 'lowpass', 0)
mGRFres_12_filtered = np.sqrt(np.sum(mGRFxyz_12_filtered**2, axis=1))
mGRF_left4_12_filtered = np.concatenate((mGRFxyz_12_filtered, np.expand_dims(mGRFres_12_filtered, axis=1))),

```

```

mGRFxyz_12_filtered = FilterData(mGRF_right4_12, 960, 2, 30, 'lowpass', 0)
mGRFres_12_filtered = np.sqrt(np.sum(mGRFxyz_12_filtered**2, axis=1))
mGRF_right4_12_filtered = np.concatenate((mGRFxyz_12_filtered, np.expand_dims(mGRFres_12_filtered, axis=1))

mGRM4_12_filtered = FilterData(mGRM4_12, fs_Vicon, 2, 30, 'lowpass', 0)

mCoP_left4_12_filtered = FilterData(mCoP_left4_12, fs_Vicon, 2, 1, 'highpass', 0)
mCoP_right4_12_filtered = FilterData(mCoP_right4_12, fs_Vicon, 2, 1, 'highpass', 0)

# jump
mGRFxyz4_jump_filtered = FilterData(mGRF4_jump, 960, 2, 30, 'lowpass', 0)
mGRFres4_jump_filtered = np.sqrt(np.sum(mGRFxyz4_jump_filtered**2, axis=1))
mGRF4_jump_filtered = np.concatenate((mGRFxyz4_jump_filtered, np.expand_dims(mGRFres4_jump_filtered, axis=1)

mGRFxyz_jump_filtered = FilterData(mGRF_left4_jump, 960, 2, 30, 'lowpass', 0)
mGRFres_jump_filtered = np.sqrt(np.sum(mGRFxyz_jump_filtered**2, axis=1))
mGRF_left4_jump_filtered = np.concatenate((mGRFxyz_jump_filtered, np.expand_dims(mGRFres_jump_filtered, axi

mGRFxyz_jump_filtered = FilterData(mGRF_right4_jump, 960, 2, 30, 'lowpass', 0)
mGRFres_jump_filtered = np.sqrt(np.sum(mGRFxyz_jump_filtered**2, axis=1))
mGRF_right4_jump_filtered = np.concatenate((mGRFxyz_jump_filtered, np.expand_dims(mGRFres_jump_filtered, ax

mGRM4_jump_filtered = FilterData(mGRM4_jump, fs_Vicon, 2, 30, 'lowpass', 0)

mCoP_left4_jump_filtered = FilterData(mCoP_left4_jump, fs_Vicon, 2, 1, 'highpass', 0)
mCoP_right4_jump_filtered = FilterData(mCoP_right4_jump, fs_Vicon, 2, 1, 'highpass', 0)

# all
mGRF4_filtered = np.concatenate((mGRF4_8_filtered, mGRF4_10_filtered, mGRF4_12_filtered), axis=0)
mGRF_left4_filtered = np.concatenate((mGRF_left4_8_filtered, mGRF_left4_10_filtered, mGRF_left4_12_filtered
mGRF_right4_filtered = np.concatenate((mGRF_right4_8_filtered, mGRF_right4_10_filtered, mGRF_right4_12_filt

mGRM4_filtered = np.concatenate((mGRM4_8_filtered, mGRM4_10_filtered, mGRM4_12_filtered), axis=0)
mCoP_left4_filtered = np.concatenate((mCoP_left4_8_filtered, mCoP_left4_10_filtered, mCoP_left4_12_filtered
mCoP_right4_filtered = np.concatenate((mCoP_right4_8_filtered, mCoP_right4_10_filtered, mCoP_right4_12_filt

```

Participant 5

8 km/h

```

mGRFxyz5_8_filtered = FilterData(mGRF5_8, 960, 2, 30, 'lowpass', 0)
mGRFres5_8_filtered = np.sqrt(np.sum(mGRFxyz5_8_filtered**2, axis=1))
mGRF5_8_filtered = np.concatenate((mGRFxyz5_8_filtered, np.expand_dims(mGRFres5_8_filtered, axis=1)), axis=

mGRFxyz_8_filtered = FilterData(mGRF_left5_8, 960, 2, 30, 'lowpass', 0)
mGRFres_8_filtered = np.sqrt(np.sum(mGRFxyz_8_filtered**2, axis=1))
mGRF_left5_8_filtered = np.concatenate((mGRFxyz_8_filtered, np.expand_dims(mGRFres_8_filtered, axis=1)), ax

mGRFxyz_8_filtered = FilterData(mGRF_right5_8, 960, 2, 30, 'lowpass', 0)
mGRFres_8_filtered = np.sqrt(np.sum(mGRFxyz_8_filtered**2, axis=1))
mGRF_right5_8_filtered = np.concatenate((mGRFxyz_8_filtered, np.expand_dims(mGRFres_8_filtered, axis=1)), a

mGRM5_8_filtered = FilterData(mGRM5_8, fs_Vicon, 2, 30, 'lowpass', 0)

mCoP_left5_8_filtered = FilterData(mCoP_left5_8, fs_Vicon, 2, 1, 'highpass', 0)
mCoP_right5_8_filtered = FilterData(mCoP_right5_8, fs_Vicon, 2, 1, 'highpass', 0)

```

10 km/h

```

mGRFxyz5_10_filtered = FilterData(mGRF5_10, 960, 2, 30, 'lowpass', 0)
mGRFres5_10_filtered = np.sqrt(np.sum(mGRFxyz5_10_filtered**2, axis=1))
mGRF5_10_filtered = np.concatenate((mGRFxyz5_10_filtered, np.expand_dims(mGRFres5_10_filtered, axis=1)), ax

mGRFxyz_10_filtered = FilterData(mGRF_left5_10, 960, 2, 30, 'lowpass', 0)
mGRFres_10_filtered = np.sqrt(np.sum(mGRFxyz_10_filtered**2, axis=1))
mGRF_left5_10_filtered = np.concatenate((mGRFxyz_10_filtered, np.expand_dims(mGRFres_10_filtered, axis=1))),

```

```

mGRFxyz_10_filtered = FilterData(mGRF_right5_10, 960, 2, 30, 'lowpass', 0)
mGRFres_10_filtered = np.sqrt(np.sum(mGRFxyz_10_filtered**2, axis=1))
mGRF_right5_10_filtered = np.concatenate((mGRFxyz_10_filtered, np.expand_dims(mGRFres_10_filtered, axis=1))

mGRM5_10_filtered = FilterData(mGRM5_10, fs_Vicon, 2, 30, 'lowpass', 0)

mCoP_left5_10_filtered = FilterData(mCoP_left5_10, fs_Vicon, 2, 1, 'highpass', 0)
mCoP_right5_10_filtered = FilterData(mCoP_right5_10, fs_Vicon, 2, 1, 'highpass', 0)

# 12 km/h
mGRFxyz5_12_filtered = FilterData(mGRF5_12, 960, 2, 30, 'lowpass', 0)
mGRFres5_12_filtered = np.sqrt(np.sum(mGRFxyz5_12_filtered**2, axis=1))
mGRF5_12_filtered = np.concatenate((mGRFxyz5_12_filtered, np.expand_dims(mGRFres5_12_filtered, axis=1)), ax

mGRFxyz_12_filtered = FilterData(mGRF_left5_12, 960, 2, 30, 'lowpass', 0)
mGRFres_12_filtered = np.sqrt(np.sum(mGRFxyz_12_filtered**2, axis=1))
mGRF_left5_12_filtered = np.concatenate((mGRFxyz_12_filtered, np.expand_dims(mGRFres_12_filtered, axis=1)),

mGRFxyz_12_filtered = FilterData(mGRF_right5_12, 960, 2, 30, 'lowpass', 0)
mGRFres_12_filtered = np.sqrt(np.sum(mGRFxyz_12_filtered**2, axis=1))
mGRF_right5_12_filtered = np.concatenate((mGRFxyz_12_filtered, np.expand_dims(mGRFres_12_filtered, axis=1))

mGRM5_12_filtered = FilterData(mGRM5_12, fs_Vicon, 2, 30, 'lowpass', 0)

mCoP_left5_12_filtered = FilterData(mCoP_left5_12, fs_Vicon, 2, 1, 'highpass', 0)
mCoP_right5_12_filtered = FilterData(mCoP_right5_12, fs_Vicon, 2, 1, 'highpass', 0)

# jump
mGRFxyz5_jump_filtered = FilterData(mGRF5_jump, 960, 2, 30, 'lowpass', 0)
mGRFres5_jump_filtered = np.sqrt(np.sum(mGRFxyz5_jump_filtered**2, axis=1))
mGRF5_jump_filtered = np.concatenate((mGRFxyz5_jump_filtered, np.expand_dims(mGRFres5_jump_filtered, axis=1

mGRFxyz_jump_filtered = FilterData(mGRF_left5_jump, 960, 2, 30, 'lowpass', 0)
mGRFres_jump_filtered = np.sqrt(np.sum(mGRFxyz_jump_filtered**2, axis=1))
mGRF_left5_jump_filtered = np.concatenate((mGRFxyz_jump_filtered, np.expand_dims(mGRFres_jump_filtered, axi

mGRFxyz_jump_filtered = FilterData(mGRF_right5_jump, 960, 2, 30, 'lowpass', 0)
mGRFres_jump_filtered = np.sqrt(np.sum(mGRFxyz_jump_filtered**2, axis=1))
mGRF_right5_jump_filtered = np.concatenate((mGRFxyz_jump_filtered, np.expand_dims(mGRFres_jump_filtered, ax

mGRM5_jump_filtered = FilterData(mGRM5_jump, fs_Vicon, 2, 30, 'lowpass', 0)

mCoP_left5_jump_filtered = FilterData(mCoP_left5_jump, fs_Vicon, 2, 1, 'highpass', 0)
mCoP_right5_jump_filtered = FilterData(mCoP_right5_jump, fs_Vicon, 2, 1, 'highpass', 0)

# all
mGRF5_filtered = np.concatenate((mGRF5_8_filtered, mGRF5_10_filtered, mGRF5_12_filtered), axis=0)
mGRF_left5_filtered = np.concatenate((mGRF_left5_8_filtered, mGRF_left5_10_filtered, mGRF_left5_12_filtered
mGRF_right5_filtered = np.concatenate((mGRF_right5_8_filtered, mGRF_right5_10_filtered, mGRF_right5_12_filt

mGRM5_filtered = np.concatenate((mGRM5_8_filtered, mGRM5_10_filtered, mGRM5_12_filtered), axis=0)
mCoP_left5_filtered = np.concatenate((mCoP_left5_8_filtered, mCoP_left5_10_filtered, mCoP_left5_12_filtered
mCoP_right5_filtered = np.concatenate((mCoP_right5_8_filtered, mCoP_right5_10_filtered, mCoP_right5_12_filt

```

Participant 6

8 km/h

```

mGRFxyz6_8_filtered = FilterData(mGRF6_8, 960, 2, 30, 'lowpass', 0)
mGRFres6_8_filtered = np.sqrt(np.sum(mGRFxyz6_8_filtered**2, axis=1))
mGRF6_8_filtered = np.concatenate((mGRFxyz6_8_filtered, np.expand_dims(mGRFres6_8_filtered, axis=1)), axis=

mGRFxyz_8_filtered = FilterData(mGRF_left6_8, 960, 2, 30, 'lowpass', 0)
mGRFres_8_filtered = np.sqrt(np.sum(mGRFxyz_8_filtered**2, axis=1))
mGRF_left6_8_filtered = np.concatenate((mGRFxyz_8_filtered, np.expand_dims(mGRFres_8_filtered, axis=1)), ax

```

```

mGRFxyz_8_filtered = FilterData(mGRF_right6_8, 960, 2, 30, 'lowpass', 0)
mGRFres_8_filtered = np.sqrt(np.sum(mGRFxyz_8_filtered**2, axis=1))
mGRF_right6_8_filtered = np.concatenate((mGRFxyz_8_filtered, np.expand_dims(mGRFres_8_filtered, axis=1)), a

mGRM6_8_filtered = FilterData(mGRM6_8, fs_Vicon, 2, 30, 'lowpass', 0)

mCoP_left6_8_filtered = FilterData(mCoP_left6_8, fs_Vicon, 2, 1, 'highpass', 0)
mCoP_right6_8_filtered = FilterData(mCoP_right6_8, fs_Vicon, 2, 1, 'highpass', 0)

# 10 km/h
mGRFxyz6_10_filtered = FilterData(mGRF6_10, 960, 2, 30, 'lowpass', 0)
mGRFres6_10_filtered = np.sqrt(np.sum(mGRFxyz6_10_filtered**2, axis=1))
mGRF6_10_filtered = np.concatenate((mGRFxyz6_10_filtered, np.expand_dims(mGRFres6_10_filtered, axis=1)), ax

mGRFxyz_10_filtered = FilterData(mGRF_left6_10, 960, 2, 30, 'lowpass', 0)
mGRFres_10_filtered = np.sqrt(np.sum(mGRFxyz_10_filtered**2, axis=1))
mGRF_left6_10_filtered = np.concatenate((mGRFxyz_10_filtered, np.expand_dims(mGRFres_10_filtered, axis=1)),

mGRFxyz_10_filtered = FilterData(mGRF_right6_10, 960, 2, 30, 'lowpass', 0)
mGRFres_10_filtered = np.sqrt(np.sum(mGRFxyz_10_filtered**2, axis=1))
mGRF_right6_10_filtered = np.concatenate((mGRFxyz_10_filtered, np.expand_dims(mGRFres_10_filtered, axis=1))

mGRM6_10_filtered = FilterData(mGRM6_10, fs_Vicon, 2, 30, 'lowpass', 0)

mCoP_left6_10_filtered = FilterData(mCoP_left6_10, fs_Vicon, 2, 1, 'highpass', 0)
mCoP_right6_10_filtered = FilterData(mCoP_right6_10, fs_Vicon, 2, 1, 'highpass', 0)

# 12 km/h
mGRFxyz6_12_filtered = FilterData(mGRF6_12, 960, 2, 30, 'lowpass', 0)
mGRFres6_12_filtered = np.sqrt(np.sum(mGRFxyz6_12_filtered**2, axis=1))
mGRF6_12_filtered = np.concatenate((mGRFxyz6_12_filtered, np.expand_dims(mGRFres6_12_filtered, axis=1)), ax

mGRFxyz_12_filtered = FilterData(mGRF_left6_12, 960, 2, 30, 'lowpass', 0)
mGRFres_12_filtered = np.sqrt(np.sum(mGRFxyz_12_filtered**2, axis=1))
mGRF_left6_12_filtered = np.concatenate((mGRFxyz_12_filtered, np.expand_dims(mGRFres_12_filtered, axis=1)),

mGRFxyz_12_filtered = FilterData(mGRF_right6_12, 960, 2, 30, 'lowpass', 0)
mGRFres_12_filtered = np.sqrt(np.sum(mGRFxyz_12_filtered**2, axis=1))
mGRF_right6_12_filtered = np.concatenate((mGRFxyz_12_filtered, np.expand_dims(mGRFres_12_filtered, axis=1))

mGRM6_12_filtered = FilterData(mGRM6_12, fs_Vicon, 2, 30, 'lowpass', 0)

mCoP_left6_12_filtered = FilterData(mCoP_left6_12, fs_Vicon, 2, 1, 'highpass', 0)
mCoP_right6_12_filtered = FilterData(mCoP_right6_12, fs_Vicon, 2, 1, 'highpass', 0)

# jump
mGRFxyz6_jump_filtered = FilterData(mGRF6_jump, 960, 2, 30, 'lowpass', 0)
mGRFres6_jump_filtered = np.sqrt(np.sum(mGRFxyz6_jump_filtered**2, axis=1))
mGRF6_jump_filtered = np.concatenate((mGRFxyz6_jump_filtered, np.expand_dims(mGRFres6_jump_filtered, axis=1

mGRFxyz_jump_filtered = FilterData(mGRF_left6_jump, 960, 2, 30, 'lowpass', 0)
mGRFres_jump_filtered = np.sqrt(np.sum(mGRFxyz_jump_filtered**2, axis=1))
mGRF_left6_jump_filtered = np.concatenate((mGRFxyz_jump_filtered, np.expand_dims(mGRFres_jump_filtered, axi

mGRFxyz_jump_filtered = FilterData(mGRF_right6_jump, 960, 2, 30, 'lowpass', 0)
mGRFres_jump_filtered = np.sqrt(np.sum(mGRFxyz_jump_filtered**2, axis=1))
mGRF_right6_jump_filtered = np.concatenate((mGRFxyz_jump_filtered, np.expand_dims(mGRFres_jump_filtered, ax

mGRM6_jump_filtered = FilterData(mGRM6_jump, fs_Vicon, 2, 30, 'lowpass', 0)

mCoP_left6_jump_filtered = FilterData(mCoP_left6_jump, fs_Vicon, 2, 1, 'highpass', 0)
mCoP_right6_jump_filtered = FilterData(mCoP_right6_jump, fs_Vicon, 2, 1, 'highpass', 0)

# all
mGRF6_filtered = np.concatenate((mGRF6_8_filtered, mGRF6_10_filtered, mGRF6_12_filtered), axis=0)

```

```

mGRF_left6_filtered = np.concatenate((mGRF_left6_8_filtered, mGRF_left6_10_filtered, mGRF_left6_12_filtered)
mGRF_right6_filtered = np.concatenate((mGRF_right6_8_filtered, mGRF_right6_10_filtered, mGRF_right6_12_filt

mGRM6_filtered = np.concatenate((mGRM6_8_filtered, mGRM6_10_filtered, mGRM6_12_filtered), axis=0)
mCoP_left6_filtered = np.concatenate((mCoP_left6_8_filtered, mCoP_left6_10_filtered, mCoP_left6_12_filtered)
mCoP_right6_filtered = np.concatenate((mCoP_right6_8_filtered, mCoP_right6_10_filtered, mCoP_right6_12_filt

```

```

# Participant 7

```

```

# 8 km/h

```

```

mGRFxyz7_8_filtered = FilterData(mGRF7_8, 960, 2, 30, 'lowpass', 0)
mGRFres7_8_filtered = np.sqrt(np.sum(mGRFxyz7_8_filtered**2, axis=1))
mGRF7_8_filtered = np.concatenate((mGRFxyz7_8_filtered, np.expand_dims(mGRFres7_8_filtered, axis=1)), axis=

```

```

mGRFxyz_8_filtered = FilterData(mGRF_left7_8, 960, 2, 30, 'lowpass', 0)
mGRFres_8_filtered = np.sqrt(np.sum(mGRFxyz_8_filtered**2, axis=1))
mGRF_left7_8_filtered = np.concatenate((mGRFxyz_8_filtered, np.expand_dims(mGRFres_8_filtered, axis=1)), ax

```

```

mGRFxyz_8_filtered = FilterData(mGRF_right7_8, 960, 2, 30, 'lowpass', 0)
mGRFres_8_filtered = np.sqrt(np.sum(mGRFxyz_8_filtered**2, axis=1))
mGRF_right7_8_filtered = np.concatenate((mGRFxyz_8_filtered, np.expand_dims(mGRFres_8_filtered, axis=1)), a

```

```

mGRM7_8_filtered = FilterData(mGRM7_8, fs_Vicon, 2, 30, 'lowpass', 0)

```

```

mCoP_left7_8_filtered = FilterData(mCoP_left7_8, fs_Vicon, 2, 1, 'highpass', 0)
mCoP_right7_8_filtered = FilterData(mCoP_right7_8, fs_Vicon, 2, 1, 'highpass', 0)

```

```

# 10 km/h

```

```

mGRFxyz7_10_filtered = FilterData(mGRF7_10, 960, 2, 30, 'lowpass', 0)
mGRFres7_10_filtered = np.sqrt(np.sum(mGRFxyz7_10_filtered**2, axis=1))
mGRF7_10_filtered = np.concatenate((mGRFxyz7_10_filtered, np.expand_dims(mGRFres7_10_filtered, axis=1)), ax

```

```

mGRFxyz_10_filtered = FilterData(mGRF_left7_10, 960, 2, 30, 'lowpass', 0)
mGRFres_10_filtered = np.sqrt(np.sum(mGRFxyz_10_filtered**2, axis=1))
mGRF_left7_10_filtered = np.concatenate((mGRFxyz_10_filtered, np.expand_dims(mGRFres_10_filtered, axis=1))),

```

```

mGRFxyz_10_filtered = FilterData(mGRF_right7_10, 960, 2, 30, 'lowpass', 0)
mGRFres_10_filtered = np.sqrt(np.sum(mGRFxyz_10_filtered**2, axis=1))
mGRF_right7_10_filtered = np.concatenate((mGRFxyz_10_filtered, np.expand_dims(mGRFres_10_filtered, axis=1)))

```

```

mGRM7_10_filtered = FilterData(mGRM7_10, fs_Vicon, 2, 30, 'lowpass', 0)

```

```

mCoP_left7_10_filtered = FilterData(mCoP_left7_10, fs_Vicon, 2, 1, 'highpass', 0)
mCoP_right7_10_filtered = FilterData(mCoP_right7_10, fs_Vicon, 2, 1, 'highpass', 0)

```

```

# 12 km/h

```

```

mGRFxyz7_12_filtered = FilterData(mGRF7_12, 960, 2, 30, 'lowpass', 0)
mGRFres7_12_filtered = np.sqrt(np.sum(mGRFxyz7_12_filtered**2, axis=1))
mGRF7_12_filtered = np.concatenate((mGRFxyz7_12_filtered, np.expand_dims(mGRFres7_12_filtered, axis=1)), ax

```

```

mGRFxyz_12_filtered = FilterData(mGRF_left7_12, 960, 2, 30, 'lowpass', 0)
mGRFres_12_filtered = np.sqrt(np.sum(mGRFxyz_12_filtered**2, axis=1))
mGRF_left7_12_filtered = np.concatenate((mGRFxyz_12_filtered, np.expand_dims(mGRFres_12_filtered, axis=1))),

```

```

mGRFxyz_12_filtered = FilterData(mGRF_right7_12, 960, 2, 30, 'lowpass', 0)
mGRFres_12_filtered = np.sqrt(np.sum(mGRFxyz_12_filtered**2, axis=1))
mGRF_right7_12_filtered = np.concatenate((mGRFxyz_12_filtered, np.expand_dims(mGRFres_12_filtered, axis=1)))

```

```

mGRM7_12_filtered = FilterData(mGRM7_12, fs_Vicon, 2, 30, 'lowpass', 0)

```

```

mCoP_left7_12_filtered = FilterData(mCoP_left7_12, fs_Vicon, 2, 1, 'highpass', 0)
mCoP_right7_12_filtered = FilterData(mCoP_right7_12, fs_Vicon, 2, 1, 'highpass', 0)

```

```

# jump

```

```

mGRFxyz7_jump_filtered = FilterData(mGRF7_jump, 960, 2, 30, 'lowpass', 0)

```

```

mGRFres7_jump_filtered = np.sqrt(np.sum(mGRFxyz7_jump_filtered**2, axis=1))
mGRF7_jump_filtered = np.concatenate((mGRFxyz7_jump_filtered, np.expand_dims(mGRFres7_jump_filtered, axis=1)

mGRFxyz_jump_filtered = FilterData(mGRF_left7_jump, 960, 2, 30, 'lowpass', 0)
mGRFres_jump_filtered = np.sqrt(np.sum(mGRFxyz_jump_filtered**2, axis=1))
mGRF_left7_jump_filtered = np.concatenate((mGRFxyz_jump_filtered, np.expand_dims(mGRFres_jump_filtered, axi

mGRFxyz_jump_filtered = FilterData(mGRF_right7_jump, 960, 2, 30, 'lowpass', 0)
mGRFres_jump_filtered = np.sqrt(np.sum(mGRFxyz_jump_filtered**2, axis=1))
mGRF_right7_jump_filtered = np.concatenate((mGRFxyz_jump_filtered, np.expand_dims(mGRFres_jump_filtered, ax

mGRM7_jump_filtered = FilterData(mGRM7_jump, fs_Vicon, 2, 30, 'lowpass', 0)

mCoP_left7_jump_filtered = FilterData(mCoP_left7_jump, fs_Vicon, 2, 1, 'highpass', 0)
mCoP_right7_jump_filtered = FilterData(mCoP_right7_jump, fs_Vicon, 2, 1, 'highpass', 0)

# all
mGRF7_filtered = np.concatenate((mGRF7_8_filtered, mGRF7_10_filtered, mGRF7_12_filtered), axis=0)
mGRF_left7_filtered = np.concatenate((mGRF_left7_8_filtered, mGRF_left7_10_filtered, mGRF_left7_12_filtered
mGRF_right7_filtered = np.concatenate((mGRF_right7_8_filtered, mGRF_right7_10_filtered, mGRF_right7_12_filt

mGRM7_filtered = np.concatenate((mGRM7_8_filtered, mGRM7_10_filtered, mGRM7_12_filtered), axis=0)
mCoP_left7_filtered = np.concatenate((mCoP_left7_8_filtered, mCoP_left7_10_filtered, mCoP_left7_12_filtered
mCoP_right7_filtered = np.concatenate((mCoP_right7_8_filtered, mCoP_right7_10_filtered, mCoP_right7_12_filt

```

```

# Er is niet echt een consensus, maar het wordt aangeraden om te upsamplen.
# Je creëert dan spookpunten, maar je behoudt de data met de hoogste resolutie.

```

```

# Downsample
# mGRF_left = scipy.signal.resample(mGRF_left_filtered, A.shape[1], axis=0)
# mGRF_right = scipy.signal.resample(mGRF_right_filtered, A.shape[1], axis=0)
# mGRF = scipy.signal.resample(mGRF_filtered, A.shape[1], axis=0)

```

✓ 5. Choose the data that you want to analyse

```

_12_filtered, A4_8_filtered, A4_10_filtered, A4_12_filtered, A5_8_filtered, A5_10_filtered, A5_12_filtered, A
, A5_8_rot, A5_10_rot, A5_12_rot, A6_8_rot, A6_10_rot, A6_12_rot, A7_8_rot, A7_10_rot, A7_12_rot), axis=1)
, axis=1)

```

```

_12_filtered, W5_8_filtered, W5_10_filtered, W5_12_filtered, W6_8_filtered, W6_10_filtered, W6_12_filtered, W
_filtered, W4_10_rot_filtered, W4_12_rot_filtered, W5_8_rot_filtered, W5_10_rot_filtered, W5_12_rot_filtered,
_12_filtered, V5_8_filtered, V5_10_filtered, V5_12_filtered, V6_8_filtered, V6_10_filtered, V6_12_filtered, V
_filtered_corr, X4_10_filtered_corr, X4_12_filtered_corr, X5_8_filtered_corr, X5_10_filtered_corr, X5_12_fil

```

```

mGRF_filtered = np.concatenate((mGRF0_filtered, mGRF2_filtered, mGRF3_filtered, mGRF4_filtered, mGRF5_filt
mGRF_left_filtered = np.concatenate((mGRF_left0_filtered, mGRF_left2_filtered, mGRF_left3_filtered, mGRF_le
mGRF_right_filtered = np.concatenate((mGRF_right0_filtered, mGRF_right2_filtered, mGRF_right3_filtered, mGR

```

```

mGRF_jumping = np.concatenate((mGRF2_jump_filtered, mGRF3_jump_filtered, mGRF4_jump_filtered, mGRF5_jump_fi
mGRF_left_jumping = np.concatenate((mGRF_left2_jump_filtered, mGRF_left3_jump_filtered, mGRF_left4_jump_fil
mGRF_right_jumping = np.concatenate((mGRF_right2_jump_filtered, mGRF_right3_jump_filtered, mGRF_right4_jump

```

```

mGRM_filtered = np.concatenate((mGRM0_8_filtered, mGRM0_10_filtered, mGRM3_8_filtered, mGRM3_10_filtered, m

```

```

M = np.mean([M0, M2, M3, M4, M5, M6, M7])
print(M)
M_std = np.std([M0, M2, M3, M4, M5, M6, M7])
print(M_std)

```

```

l = np.mean([l0, l2, l3, l4, l5, l6, l7])
print(l)
l_std = np.std([l0, l2, l3, l4, l5, l6, l7])
print(l_std)
ages = np.mean([34, 26, 26, 21, 24, 27, 34])
print(ages)
ages_std = np.std([34, 26, 26, 21, 24, 27, 34])
print(ages_std)
AM = np.mean([AM0, AM2, AM3, AM4, AM5, AM6, AM7], axis=0)
print(AM)

```

```

70.23571428571428
6.339823920815055
1.7671428571428571
0.08630747123996127
27.428571428571427
4.531071864901059
[[0.12417052 0.12765466 0.42853976]
 [0.11170422 0.13277455 0.59236334]
 [0.31730684 0.20488649 0.74623976]]

```

6. Step detection

```

def CheckSteps(IC_eGRF, TO_eGRF, IC_mGRF, TO_mGRF):

    """
    This function ensures that we are comparing the same steps of the estimated GRF and the measured GRF.
    The function works under the assumption that eGRF and mGRF are recorded at the same time
    """

    step_length = np.mean(IC_mGRF[1:] - IC_mGRF[:-1]) # duration of one step, including flight phase

    # check if the IC's of the mGRF and the eGRF are close enough to each other
    i = 0
    IC_eGRF_final = []
    j = 0
    IC_mGRF_final = []
    while i < len(IC_eGRF) and j < len(IC_mGRF):
        if abs(IC_eGRF[i] - IC_mGRF[j]) <= step_length/2: # not too much difference between time of init
            IC_eGRF_final.append(IC_eGRF[i])
            i += 1
            IC_mGRF_final.append(IC_mGRF[j])
            j += 1
        elif (IC_eGRF[i] - IC_mGRF[j]) > step_length/2:
            j += 1
        elif (IC_mGRF[j] - IC_eGRF[i]) > step_length/2:
            i += 1

    # check if the TO's of the mGRF and the eGRF are close enough to each other
    p = 0
    TO_eGRF_final = []
    q = 0
    TO_mGRF_final = []
    while p < len(TO_eGRF) and q < len(TO_mGRF):
        if abs(TO_eGRF[p] - TO_mGRF[q]) <= step_length/2: # not too much difference between time of init
            TO_eGRF_final.append(TO_eGRF[p])
            p += 1
            TO_mGRF_final.append(TO_mGRF[q])
            q += 1
        elif (TO_eGRF[p] - TO_mGRF[q]) > step_length/2:
            q += 1
        elif (TO_mGRF[q] - TO_eGRF[p]) > step_length/2:
            p += 1
    
```

```
return np.array(IC_eGRF_final), np.array(TO_eGRF_final), np.array(IC_mGRF_final), np.array(TO_mGRF_fina
```

```
def GetSteps(IC, TO, A, GRF, fs): # be careful: IC and TO should be calculated on the same GRF
    # adjust the arrays beforehand, in order to start with initial contact
    """
    Divide the recorded trial in steps.

    - IC: initial contact
    - TO: toe-off
    - A: accelerations
    - GRF: ground reaction force
    - fs: sampling frequency
    """

    steps = []
    steps_left = []
    steps_right = []

    n_steps = len(IC)
    step_length = int(np.max(TO-IC))

    for i in range(n_steps-1):
        step = GRF[IC[i]:TO[i]] # one step is from IC until TO (without flight phase)
        steps.append(scipy.signal.resample(step, step_length)) # to compare the steps more easily, make th
        # determine if it is a left or a right step, based on the accelerations
        i_peak, peak = np.argmax(step), np.max(step) # indices restart counting in the step
        if (IC[i]+i_peak) < A.shape[1]: # avoid problems at the end
            Av_thigh_left = A[2, IC[i]+i_peak, 2]
            Av_thigh_right = A[0, IC[i]+i_peak, 2]
            if Av_thigh_left > Av_thigh_right: # als het rechterbeen op de grond staat (-> mGRF right), z
                steps_right.append(scipy.signal.resample(step, step_length))
            elif Av_thigh_right > Av_thigh_left:
                steps_left.append(scipy.signal.resample(step, step_length))

    # t = 1/fs * np.arange(0, step_length, 1)
    t = np.linspace(0, 100, step_length) # Define the time-axis as a percentage of one full gait cycle

    return np.array(t), np.array(steps), np.array(steps_left), np.array(steps_right)
```

```
def PlotSteps(t, steps, title):

    """
    Plot the mean step with standard deviation
    """

    if len(steps.shape) == 3:
        steps_x = steps[:, :, 0] # shape: (n_steps, step_length, direction)
        steps_y = steps[:, :, 1]
        steps_z = steps[:, :, 2]
        steps_res = steps[:, :, 3]

        # mean step and standard deviation
        mean_step_x, stdv_step_x = np.mean(steps_x, axis=0), np.std(steps_x, axis=0)
        mean_step_y, stdv_step_y = np.mean(steps_y, axis=0), np.std(steps_y, axis=0)
        mean_step_z, stdv_step_z = np.mean(steps_z, axis=0), np.std(steps_z, axis=0)
        mean_step_res, stdv_step_res = np.mean(steps_res, axis=0), np.std(steps_res, axis=0)

        # plot
        fig, ax = plt.subplots(nrows=1, ncols=3, figsize=(12,4))
        fig.suptitle(title)

        ax[0].fill_between(t, mean_step_x - stdv_step_x, mean_step_x + stdv_step_x, color='c', alpha=0.3)
        ax[0].plot(t, mean_step_x)
```

```

ax[0].set_xlabel('Time [s]')
ax[0].set_ylabel('GRFx [BW] ')
#ax[0].set_title('Mean estimated GRFx with one standard deviation')

ax[1].fill_between(t, mean_step_y - stdv_step_y, mean_step_y + stdv_step_y, color='c', alpha=0.3)
ax[1].plot(t, mean_step_y)
ax[1].set_xlabel('Time [s]')
ax[1].set_ylabel('GRFy [BW] ')
#ax[1].set_title('Mean estimated GRFy with one standard deviation')

ax[2].fill_between(t, mean_step_z - stdv_step_z, mean_step_z + stdv_step_z, color='c', alpha=0.3)
ax[2].plot(t, mean_step_z)
ax[2].set_xlabel('Time [s]')
ax[2].set_ylabel('GRFz [BW] ')
#ax[2].set_title('Mean estimated GRFz with one standard deviation')

#ax[3].fill_between(t, mean_step_res - stdv_step_res, mean_step_res + stdv_step_res, color='c', alp
#ax[3].plot(t, mean_step_res)
#ax[3].set_xlabel('Time [s]')
#ax[3].set_ylabel('GRFres / BW ')
#ax[3].set_title('Mean estimated GRFres with one standard deviation')

plt.tight_layout()
plt.show()

if len(steps.shape) == 2:
    steps_z = steps[:, :] # shape: (n_steps, step_length)

    # mean step and standard deviation
    mean_step_z, stdv_step_z = np.mean(steps_z, axis=0), np.std(steps_z, axis=0)

    # plot
    fig, ax = plt.subplots(nrows=1, ncols=4, figsize=(12,4))
    fig.suptitle(title)

    plt.fill_between(t, mean_step_z - stdv_step_z, mean_step_z + stdv_step_z, color='c', alpha=0.3)
    plt.plot(t, mean_step_z)
    plt.xlabel('Time [s]')
    plt.ylabel('GRFz [BW] ')
    plt.title('Mean estimated GRFz with one standard deviation')

    plt.tight_layout()
    plt.show()

```

```

def PlotSteps2(t1, steps1, t2, steps2, labels, title):

    """
    Plot the mean step with standard deviation of two variables
    """

    if len(steps1.shape) == 3:
        steps1_x = np.array(steps1[:, :, 0])
        steps1_y = np.array(steps1[:, :, 1])
        steps1_z = np.array(steps1[:, :, 2])
        #steps1_res = np.array(steps1[:, :, 3])

        steps2_x = np.array(steps2[:, :, 0])
        steps2_y = np.array(steps2[:, :, 1])
        steps2_z = np.array(steps2[:, :, 2])
        #steps2_res = np.array(steps2[:, :, 3])

        # mean step and standard deviation
        mean_step1_x, stdv_step1_x = np.mean(steps1_x, axis=0), np.std(steps1_x, axis=0)
        mean_step1_y, stdv_step1_y = np.mean(steps1_y, axis=0), np.std(steps1_y, axis=0)
        mean_step1_z, stdv_step1_z = np.mean(steps1_z, axis=0), np.std(steps1_z, axis=0)

```

```

#mean_step1_res, stdv_step1_res = np.mean(steps1_res, axis=0), np.std(steps1_res, axis=0)

mean_step2_x, stdv_step2_x = np.mean(steps2_x, axis=0), np.std(steps2_x, axis=0)
mean_step2_y, stdv_step2_y = np.mean(steps2_y, axis=0), np.std(steps2_y, axis=0)
mean_step2_z, stdv_step2_z = np.mean(steps2_z, axis=0), np.std(steps2_z, axis=0)
#mean_step2_res, stdv_step2_res = np.mean(steps2_res, axis=0), np.std(steps2_res, axis=0)

# plot
fig, ax = plt.subplots(nrows=1, ncols=3, figsize=(12,4))
fig.suptitle(title)

ax[0].fill_between(t1, mean_step1_x - stdv_step1_x, mean_step1_x + stdv_step1_x, color='c', alpha=0)
ax[0].plot(t1, mean_step1_x, label=labels[0])
ax[0].fill_between(t2, mean_step2_x - stdv_step2_x, mean_step2_x + stdv_step2_x, color='y', alpha=0)
ax[0].plot(t2, mean_step2_x, color='#ff7f01', label=labels[1])
ax[0].set_xlabel('Time [s]')
ax[0].set_ylabel('GRFx [BW] ')
#ax[0].legend(loc="lower right")
#ax[0].set_title('Mean estimated GRFx with one standard deviation')

ax[1].fill_between(t1, mean_step1_y - stdv_step1_y, mean_step1_y + stdv_step1_y, color='c', alpha=0)
ax[1].plot(t1, mean_step1_y, label=labels[0])
ax[1].fill_between(t2, mean_step2_y - stdv_step2_y, mean_step2_y + stdv_step2_y, color='y', alpha=0)
ax[1].plot(t2, mean_step2_y, color='#ff7f01', label=labels[1])
ax[1].set_xlabel('Time [s]')
ax[1].set_ylabel('GRFy [BW] ')
#ax[1].legend(loc="lower right")
#ax[1].set_title('Mean estimated GRFy with one standard deviation')

ax[2].fill_between(t1, mean_step1_z - stdv_step1_z, mean_step1_z + stdv_step1_z, color='c', alpha=0)
ax[2].plot(t1, mean_step1_z, label=labels[0])
ax[2].fill_between(t2, mean_step2_z - stdv_step2_z, mean_step2_z + stdv_step2_z, color='y', alpha=0)
ax[2].plot(t2, mean_step2_z, color='#ff7f01', label=labels[1])
ax[2].set_xlabel('Time [s]')
ax[2].set_ylabel('GRFz [BW]')
ax[2].legend(loc="lower right")
#ax[2].set_title('Mean estimated GRFz with one standard deviation')

#ax[3].fill_between(t1, mean_step1_res - stdv_step1_res, mean_step1_res + stdv_step1_res, color='c'
#ax[3].plot(t1, mean_step1_res, label=labels[0])
#ax[3].fill_between(t2, mean_step2_res - stdv_step2_res, mean_step2_res + stdv_step2_res, color='y'
#ax[3].plot(t2, mean_step2_res, color='#ff7f01', label=labels[1])
#ax[3].set_xlabel('Time [s]')
#ax[3].set_ylabel('GRFres [BW] ')
#ax[3].legend(loc="lower right")
#ax[3].set_title('Mean estimated GRFres with one standard deviation')

if len(steps1.shape) == 2:
    steps1_z = steps1[:, :] # shape: (n_steps, step_length)
    steps2_z = steps2[:, :] # shape: (n_steps, step_length)

    # mean step and standard deviation
    mean_step1_z, stdv_step1_z = np.mean(steps1_z, axis=0), np.std(steps1_z, axis=0)
    mean_step2_z, stdv_step2_z = np.mean(steps2_z, axis=0), np.std(steps2_z, axis=0)

    # plot
    fig, ax = plt.subplots(nrows=1, ncols=1, figsize=(12,4))
    fig.suptitle(title)

    plt.fill_between(t1, mean_step1_z - stdv_step1_z, mean_step1_z + stdv_step1_z, color='c', alpha=0.3)
    plt.plot(t1, mean_step1_z, label=labels[0])
    plt.fill_between(t2, mean_step2_z - stdv_step2_z, mean_step2_z + stdv_step2_z, color='y', alpha=0.3)
    plt.plot(t2, mean_step2_z, color='#ff7f01', label=labels[1])
    plt.xlabel('Time [s]')
    plt.ylabel('GRFz [BW] ')

```

```
plt.title('Mean estimated GRFz with one standard deviation')

plt.tight_layout()
plt.show()
```

✓ 6.1 Option 1: arbitrary threshold

A foot contact is detected if the vertical force crosses a certain threshold. For example:

- Flight-phase if the measured vertical GRF is <25N for longer than 0.05s,
- Stance-phase if the measured vertical GRF is >25N for longer than 0.05s. This method is typically used for force plates. It should be noticed that in this case, the GRF is not corrected for BW (I think).

Source: <https://www.sciencedirect.com/science/article/pii/S0021929021001998>

```
def StepDetector1(GRF, M, step_t, fs, threshold): # GRF must be a 1D-array

    """
    Detect initial contact (IC) and toe-off (TO) of the GRF, based on a threshold value.

    - GRF: ground reaction force
    - M: mass of the participant
    - step_t: length of the interval where the threshold value will be checked, typically 0.05s
    - fs: sampling frequency
    - threshold: threshold value, typically between 10 N and 50 N
    """

    g = 9.81
    step_i = int(step_t*fs)
    if len(GRF.shape) == 2:
        GRFv = GRF[:, 2]*M*g
    if len(GRF.shape) == 1:
        GRFv = GRF*M*g

    indices_s = [] # indices stance phase

    for i in range(step_i, len(GRFv)):
        if (np.mean(GRFv[i-step_i:i]) > threshold):
            indices_s.append(i) # all indices of the stance phase

    IC = []
    TO = []
    for i in range(1, len(indices_s)):
        if (indices_s[i] - indices_s[i-1]) > 1: # when the difference is bigger than 1, we have IC and TO
            IC.append(indices_s[i])
            TO.append(indices_s[i-1]-step_i) # TO will always be first
    IC = np.array(IC) # make arrays of equal length (see command below)
    TO = np.array(TO) # make sure that we start with IC

    # check for errors
    step_length = np.mean(IC[1:] - IC[:-1]) # duration of one step, including flight phase
    # make sure that we have the same amount of IC and TO, with IC first
    i = 0
    IC_final = []
    j = 0
    TO_final = []
    step_length = np.mean(IC[1:] - IC[:-1])

    while i < len(IC) and j < len(TO):
        if 2*step_i < TO[j] - IC[i] <= step_length: # not too close or too much difference between time
            IC_final.append(IC[i])
            i += 1
            TO_final.append(TO[j])
```

```

        j += 1
    elif TO[j]-IC[i] <= 2*step_i:
        j += 1
    elif (TO[j] - IC[i]) > step_length:
        i += 1

```

```

return np.array(IC_final), np.array(TO_final)

```

```

step_t = 0.05 # s
threshold = 15 # threshold values can range from 10N to 50N, according to literature
step_t_lr = 0.1
threshold_lr = 15 #TO DO: probeer een andere threshold-waarde

# Participant 0
IC_mGRF0_8, TO_mGRF0_8 = StepDetector1(mGRF0_8_filtered, M0, step_t, fs_Vicon, threshold)
t_mGRF0_8, steps_mGRF0_8, __, __ = GetSteps(IC_mGRF0_8, TO_mGRF0_8, A0_8_filtered, mGRF0_8_filtered, fs_Vicon)
IC_mGRF0_10, TO_mGRF0_10 = StepDetector1(mGRF0_10_filtered, M0, step_t, fs_Vicon, threshold)
t_mGRF0_10, steps_mGRF0_10, __, __ = GetSteps(IC_mGRF0_10, TO_mGRF0_10, A0_10_filtered, mGRF0_10_filtered, fs_Vicon)
# IC_mGRF0_12, TO_mGRF0_12 = StepDetector1(mGRF0_12_filtered, M0, step_t, fs_Vicon, threshold)
# t_mGRF0_12, steps_mGRF0_12, __, __ = GetSteps(IC_mGRF0_12, TO_mGRF0_12, A0_12_filtered, mGRF0_12_filtered, fs_Vicon)

# Participant 2
IC_mGRF2_8, TO_mGRF2_8 = StepDetector1(mGRF2_8_filtered, M2, step_t, fs_Vicon, threshold)
t_mGRF2_8, steps_mGRF2_8, __, __ = GetSteps(IC_mGRF2_8, TO_mGRF2_8, A2_8_filtered, mGRF2_8_filtered, fs_Vicon)
IC_mGRF2_10, TO_mGRF2_10 = StepDetector1(mGRF2_10_filtered, M2, step_t, fs_Vicon, threshold)
t_mGRF2_10, steps_mGRF2_10, __, __ = GetSteps(IC_mGRF2_10, TO_mGRF2_10, A2_10_filtered, mGRF2_10_filtered, fs_Vicon)
IC_mGRF2_12, TO_mGRF2_12 = StepDetector1(mGRF2_12_filtered, M2, step_t, fs_Vicon, threshold)
t_mGRF2_12, steps_mGRF2_12, __, __ = GetSteps(IC_mGRF2_12, TO_mGRF2_12, A2_12_filtered, mGRF2_12_filtered, fs_Vicon)

# Participant 3
IC_mGRF3_8, TO_mGRF3_8 = StepDetector1(mGRF3_8_filtered, M3, step_t, fs_Vicon, threshold)
t_mGRF3_8, steps_mGRF3_8, __, __ = GetSteps(IC_mGRF3_8, TO_mGRF3_8, A3_8_filtered, mGRF3_8_filtered, fs_Vicon)
IC_mGRF3_10, TO_mGRF3_10 = StepDetector1(mGRF3_10_filtered, M3, step_t, fs_Vicon, threshold)
t_mGRF3_10, steps_mGRF3_10, __, __ = GetSteps(IC_mGRF3_10, TO_mGRF3_10, A3_10_filtered, mGRF3_10_filtered, fs_Vicon)
IC_mGRF3_12, TO_mGRF3_12 = StepDetector1(mGRF3_12_filtered, M3, step_t, fs_Vicon, threshold)
t_mGRF3_12, steps_mGRF3_12, __, __ = GetSteps(IC_mGRF3_12, TO_mGRF3_12, A3_12_filtered, mGRF3_12_filtered, fs_Vicon)

# Participant 4
IC_mGRF4_8, TO_mGRF4_8 = StepDetector1(mGRF4_8_filtered, M4, step_t, fs_Vicon, threshold)
t_mGRF4_8, steps_mGRF4_8, __, __ = GetSteps(IC_mGRF4_8, TO_mGRF4_8, A4_8_filtered, mGRF4_8_filtered, fs_Vicon)
IC_mGRF4_10, TO_mGRF4_10 = StepDetector1(mGRF4_10_filtered, M4, step_t, fs_Vicon, threshold)
t_mGRF4_10, steps_mGRF4_10, __, __ = GetSteps(IC_mGRF4_10, TO_mGRF4_10, A4_10_filtered, mGRF4_10_filtered, fs_Vicon)
IC_mGRF4_12, TO_mGRF4_12 = StepDetector1(mGRF4_12_filtered, M4, step_t, fs_Vicon, threshold)
t_mGRF4_12, steps_mGRF4_12, __, __ = GetSteps(IC_mGRF4_12, TO_mGRF4_12, A4_12_filtered, mGRF4_12_filtered, fs_Vicon)

# Participant 5
IC_mGRF5_8, TO_mGRF5_8 = StepDetector1(mGRF5_8_filtered, M5, step_t, fs_Vicon, threshold)
t_mGRF5_8, steps_mGRF5_8, __, __ = GetSteps(IC_mGRF5_8, TO_mGRF5_8, A5_8_filtered, mGRF5_8_filtered, fs_Vicon)
IC_mGRF5_10, TO_mGRF5_10 = StepDetector1(mGRF5_10_filtered, M5, step_t, fs_Vicon, threshold)
t_mGRF5_10, steps_mGRF5_10, __, __ = GetSteps(IC_mGRF5_10, TO_mGRF5_10, A5_10_filtered, mGRF5_10_filtered, fs_Vicon)
IC_mGRF5_12, TO_mGRF5_12 = StepDetector1(mGRF5_12_filtered, M5, step_t, fs_Vicon, threshold)
t_mGRF5_12, steps_mGRF5_12, __, __ = GetSteps(IC_mGRF5_12, TO_mGRF5_12, A5_12_filtered, mGRF5_12_filtered, fs_Vicon)

# Participant 6
IC_mGRF6_8, TO_mGRF6_8 = StepDetector1(mGRF6_8_filtered, M6, step_t, fs_Vicon, threshold)
t_mGRF6_8, steps_mGRF6_8, __, __ = GetSteps(IC_mGRF6_8, TO_mGRF6_8, A6_8_filtered, mGRF6_8_filtered, fs_Vicon)
IC_mGRF6_10, TO_mGRF6_10 = StepDetector1(mGRF6_10_filtered, M6, step_t, fs_Vicon, threshold)
t_mGRF6_10, steps_mGRF6_10, __, __ = GetSteps(IC_mGRF6_10, TO_mGRF6_10, A6_10_filtered, mGRF6_10_filtered, fs_Vicon)
IC_mGRF6_12, TO_mGRF6_12 = StepDetector1(mGRF6_12_filtered, M6, step_t, fs_Vicon, threshold)
t_mGRF6_12, steps_mGRF6_12, __, __ = GetSteps(IC_mGRF6_12, TO_mGRF6_12, A6_12_filtered, mGRF6_12_filtered, fs_Vicon)

# Participant 7
IC_mGRF7_8, TO_mGRF7_8 = StepDetector1(mGRF7_8_filtered, M7, step_t, fs_Vicon, threshold)
t_mGRF7_8, steps_mGRF7_8, __, __ = GetSteps(IC_mGRF7_8, TO_mGRF7_8, A7_8_filtered, mGRF7_8_filtered, fs_Vicon)
IC_mGRF7_10, TO_mGRF7_10 = StepDetector1(mGRF7_10_filtered, M7, step_t, fs_Vicon, threshold)

```

```
t_mGRF7_10, steps_mGRF7_10, _, _ = GetSteps(IC_mGRF7_10, TO_mGRF7_10, A7_10_filtered, mGRF7_10_filtered, fs
IC_mGRF7_12, TO_mGRF7_12 = StepDetector1(mGRF7_12_filtered, M7, step_t, fs_Vicon, threshold)
t_mGRF7_12, steps_mGRF7_12, _, _ = GetSteps(IC_mGRF7_12, TO_mGRF7_12, A7_12_filtered, mGRF7_12_filtered, fs
```

✓ 6.2 Option 2: using the quaternions

Detect a step if w-component of quaternion is bigger than an arbitrary threshold. The quaternion of the IMU of the pelvis seems the best indicator.

```
def StepDetector2(Q, i_body, GRF, threshold):    # This is more a peak detector or leg detector...

    """
    Detect initial contact (IC) and toe-off (TO) of the estimated GRF, based on a threshold value for
    the orientation, defined by the quaternions, of a body segment.

    - Q: quaternions
    - i_body: body segment
    - GRF: ground reaction force
    - threshold: threshold value, typically > 10 000
    """

    qw = Q[i_body][:, 3]

    indices_s = []
    for i in range(len(qw)):
        if qw[i] > threshold:
            indices_s.append(i)

    # step detection
    IC = []
    TO = []
    for i in range(len(indices_s)):
        if (indices_s[i] - indices_s[i-1]) > 2:
            IC.append(indices_s[i-1])
            TO.append(indices_s[i])

    return IC, TO
```

✓ 6.3 Option 3: using the accelerations

According to literature (<https://pmc.ncbi.nlm.nih.gov/articles/PMC6480623/>), the peak downward velocity of the pelvis (PDVP) is a good indicator. The PDVP for the inertial measurement system is defined as the minimum of the integral of linear acceleration in the global vertical axis of the IMU on the sacrum.

```
def StepDetector3(fs, A):

    """
    Detect initial contact (IC) and toe-off (TO) of the estimated GRF, based on the peak downward velocity o
    - fs: sampling frequency
    - A: accelerations, measured by the IMUs
    """

    AvP = A[4, :, 2]+g # AvP: vertical acceleration of the pelvis
    VP = scipy.integrate.cumulative_trapezoid(AvP, dx=1/fs) # VP: vertical velocity of the pelvis
    IC = scipy.signal.argrelmin(VP)[0] # indices of minima
    TO = scipy.signal.argrelmax(VP)[0] # indices of maxima

    # make sure that we have the same amount of IC and TO, with IC first
    i = 0
```

```

IC_final = []
j = 0
TO_final = []
step_length = np.mean(IC[1:] - IC[:-1])

while i < len(IC) and j < len(TO):
    if 0 < TO[j] - IC[i] <= step_length:    # not too close or too much difference between time of init
        IC_final.append(IC[i])
        i += 1
        TO_final.append(TO[j])
        j += 1
    elif TO[j]-IC[i] <= 0:
        j += 1
    elif (TO[j] - IC[i]) > step_length:
        i += 1

return np.array(IC_final), np.array(TO_final)

```

```

# Participant 0
# IC_eGRF0_7, TO_eGRF0_7 = StepDetector3(fs_Vicon, A0_7_filtered)
# t_eGRF0_7, steps_eGRF0_7, __, __ = GetSteps(IC_eGRF0_7, TO_eGRF0_7, A0_7_filtered, mGRF0_7_filtered, fs_Vicon)
IC_eGRF0_8, TO_eGRF0_8 = StepDetector3(fs_Vicon, A0_8_filtered)
t_eGRF0_8, steps_eGRF0_8, __, __ = GetSteps(IC_eGRF0_8, TO_eGRF0_8, A0_8_filtered, mGRF0_8_filtered, fs_Vicon)
# IC_eGRF0_9, TO_eGRF0_9 = StepDetector3(fs_Vicon, A0_9_filtered)
# t_eGRF0_9, steps_eGRF0_9, __, __ = GetSteps(IC_eGRF0_9, TO_eGRF0_9, A0_9_filtered, mGRF0_9_filtered, fs_Vicon)
IC_eGRF0_10, TO_eGRF0_10 = StepDetector3(fs_Vicon, A0_10_filtered)
t_eGRF0_10, steps_eGRF0_10, __, __ = GetSteps(IC_eGRF0_10, TO_eGRF0_10, A0_10_filtered, mGRF0_10_filtered, fs_Vicon)
# IC_eGRF0_11, TO_eGRF0_11 = StepDetector3(fs_Vicon, A0_11_filtered)
# t_eGRF0_11, steps_eGRF0_11, __, __ = GetSteps(IC_eGRF0_11, TO_eGRF0_11, A0_11_filtered, mGRF0_11_filtered, fs_Vicon)
# IC_eGRF0_12, TO_eGRF0_12 = StepDetector3(fs_Vicon, A0_12_filtered)
# t_eGRF0_12, steps_eGRF0_12, __, __ = GetSteps(IC_eGRF0_12, TO_eGRF0_12, A0_12_filtered, mGRF0_12_filtered, fs_Vicon)

# Participant 2
IC_eGRF2_8, TO_eGRF2_8 = StepDetector3(fs_Vicon, A2_8_filtered)
t_eGRF2_8, steps_eGRF2_8, __, __ = GetSteps(IC_eGRF2_8, TO_eGRF2_8, A2_8_filtered, mGRF2_8_filtered, fs_Vicon)
IC_eGRF2_10, TO_eGRF2_10 = StepDetector3(fs_Vicon, A2_10_filtered)
t_eGRF2_10, steps_eGRF2_10, __, __ = GetSteps(IC_eGRF2_10, TO_eGRF2_10, A2_10_filtered, mGRF2_10_filtered, fs_Vicon)
IC_eGRF2_12, TO_eGRF2_12 = StepDetector3(fs_Vicon, A2_12_filtered)
t_eGRF2_12, steps_eGRF2_12, __, __ = GetSteps(IC_eGRF2_12, TO_eGRF2_12, A2_12_filtered, mGRF2_12_filtered, fs_Vicon)

# Participant 3
IC_eGRF3_8, TO_eGRF3_8 = StepDetector3(fs_Vicon, A3_8_filtered)
t_eGRF3_8, steps_eGRF3_8, __, __ = GetSteps(IC_eGRF3_8, TO_eGRF3_8, A3_8_filtered, mGRF3_8_filtered, fs_Vicon)
IC_eGRF3_10, TO_eGRF3_10 = StepDetector3(fs_Vicon, A3_10_filtered)
t_eGRF3_10, steps_eGRF3_10, __, __ = GetSteps(IC_eGRF3_10, TO_eGRF3_10, A3_10_filtered, mGRF3_10_filtered, fs_Vicon)
IC_eGRF3_12, TO_eGRF3_12 = StepDetector3(fs_Vicon, A3_12_filtered)
t_eGRF3_12, steps_eGRF3_12, __, __ = GetSteps(IC_eGRF3_12, TO_eGRF3_12, A3_12_filtered, mGRF3_12_filtered, fs_Vicon)

# Participant 4
IC_eGRF4_8, TO_eGRF4_8 = StepDetector3(fs_Vicon, A4_8_filtered)
t_eGRF4_8, steps_eGRF4_8, __, __ = GetSteps(IC_eGRF4_8, TO_eGRF4_8, A4_8_filtered, mGRF4_8_filtered, fs_Vicon)
IC_eGRF4_10, TO_eGRF4_10 = StepDetector3(fs_Vicon, A4_10_filtered)
t_eGRF4_10, steps_eGRF4_10, __, __ = GetSteps(IC_eGRF4_10, TO_eGRF4_10, A4_10_filtered, mGRF4_10_filtered, fs_Vicon)
IC_eGRF4_12, TO_eGRF4_12 = StepDetector3(fs_Vicon, A4_12_filtered)
t_eGRF4_12, steps_eGRF4_12, __, __ = GetSteps(IC_eGRF4_12, TO_eGRF4_12, A4_12_filtered, mGRF4_12_filtered, fs_Vicon)

# Participant 5
IC_eGRF5_8, TO_eGRF5_8 = StepDetector3(fs_Vicon, A5_8_filtered)
t_eGRF5_8, steps_eGRF5_8, __, __ = GetSteps(IC_eGRF5_8, TO_eGRF5_8, A5_8_filtered, mGRF5_8_filtered, fs_Vicon)
IC_eGRF5_10, TO_eGRF5_10 = StepDetector3(fs_Vicon, A5_10_filtered)
t_eGRF5_10, steps_eGRF5_10, __, __ = GetSteps(IC_eGRF5_10, TO_eGRF5_10, A5_10_filtered, mGRF5_10_filtered, fs_Vicon)
IC_eGRF5_12, TO_eGRF5_12 = StepDetector3(fs_Vicon, A5_12_filtered)
t_eGRF5_12, steps_eGRF5_12, __, __ = GetSteps(IC_eGRF5_12, TO_eGRF5_12, A5_12_filtered, mGRF5_12_filtered, fs_Vicon)

# Participant 6

```

```

IC_eGRF6_8, TO_eGRF6_8 = StepDetector3(fs_Vicon, A6_8_filtered)
t_eGRF6_8, steps_eGRF6_8, __, __ = GetSteps(IC_eGRF6_8, TO_eGRF6_8, A6_8_filtered, mGRF6_8_filtered, fs_Vicon)
IC_eGRF6_10, TO_eGRF6_10 = StepDetector3(fs_Vicon, A6_10_filtered)
t_eGRF6_10, steps_eGRF6_10, __, __ = GetSteps(IC_eGRF6_10, TO_eGRF6_10, A6_10_filtered, mGRF6_10_filtered, fs_Vicon)
IC_eGRF6_12, TO_eGRF6_12 = StepDetector3(fs_Vicon, A6_12_filtered)
t_eGRF6_12, steps_eGRF6_12, __, __ = GetSteps(IC_eGRF6_12, TO_eGRF6_12, A6_12_filtered, mGRF6_12_filtered, fs_Vicon)

# Participant 7
IC_eGRF7_8, TO_eGRF7_8 = StepDetector3(fs_Vicon, A7_8_filtered)
t_eGRF7_8, steps_eGRF7_8, __, __ = GetSteps(IC_eGRF7_8, TO_eGRF7_8, A7_8_filtered, mGRF7_8_filtered, fs_Vicon)
IC_eGRF7_10, TO_eGRF7_10 = StepDetector3(fs_Vicon, A7_10_filtered)
t_eGRF7_10, steps_eGRF7_10, __, __ = GetSteps(IC_eGRF7_10, TO_eGRF7_10, A7_10_filtered, mGRF7_10_filtered, fs_Vicon)
IC_eGRF7_12, TO_eGRF7_12 = StepDetector3(fs_Vicon, A7_12_filtered)
t_eGRF7_12, steps_eGRF7_12, __, __ = GetSteps(IC_eGRF7_12, TO_eGRF7_12, A7_12_filtered, mGRF7_12_filtered, fs_Vicon)

```

```

# Participant 0
IC_eGRF0_8_check, TO_eGRF0_8_check, IC_mGRF0_8_check, TO_mGRF0_8_check = CheckSteps(IC_eGRF0_8, TO_eGRF0_8, A6_8_filtered, mGRF0_8_filtered, fs_Vicon)
IC_eGRF0_10_check, TO_eGRF0_10_check, IC_mGRF0_10_check, TO_mGRF0_10_check = CheckSteps(IC_eGRF0_10, TO_eGRF0_10, A6_10_filtered, mGRF0_10_filtered, fs_Vicon)

# Participant 2
IC_eGRF2_8_check, TO_eGRF2_8_check, IC_mGRF2_8_check, TO_mGRF2_8_check = CheckSteps(IC_eGRF2_8, TO_eGRF2_8, A6_8_filtered, mGRF2_8_filtered, fs_Vicon)
IC_eGRF2_10_check, TO_eGRF2_10_check, IC_mGRF2_10_check, TO_mGRF2_10_check = CheckSteps(IC_eGRF2_10, TO_eGRF2_10, A6_10_filtered, mGRF2_10_filtered, fs_Vicon)
IC_eGRF2_12_check, TO_eGRF2_12_check, IC_mGRF2_12_check, TO_mGRF2_12_check = CheckSteps(IC_eGRF2_12, TO_eGRF2_12, A6_12_filtered, mGRF2_12_filtered, fs_Vicon)

# Participant 3
IC_eGRF3_8_check, TO_eGRF3_8_check, IC_mGRF3_8_check, TO_mGRF3_8_check = CheckSteps(IC_eGRF3_8, TO_eGRF3_8, A6_8_filtered, mGRF3_8_filtered, fs_Vicon)
IC_eGRF3_10_check, TO_eGRF3_10_check, IC_mGRF3_10_check, TO_mGRF3_10_check = CheckSteps(IC_eGRF3_10, TO_eGRF3_10, A6_10_filtered, mGRF3_10_filtered, fs_Vicon)
IC_eGRF3_12_check, TO_eGRF3_12_check, IC_mGRF3_12_check, TO_mGRF3_12_check = CheckSteps(IC_eGRF3_12, TO_eGRF3_12, A6_12_filtered, mGRF3_12_filtered, fs_Vicon)

# Participant 4
IC_eGRF4_8_check, TO_eGRF4_8_check, IC_mGRF4_8_check, TO_mGRF4_8_check = CheckSteps(IC_eGRF4_8, TO_eGRF4_8, A6_8_filtered, mGRF4_8_filtered, fs_Vicon)
IC_eGRF4_10_check, TO_eGRF4_10_check, IC_mGRF4_10_check, TO_mGRF4_10_check = CheckSteps(IC_eGRF4_10, TO_eGRF4_10, A6_10_filtered, mGRF4_10_filtered, fs_Vicon)
IC_eGRF4_12_check, TO_eGRF4_12_check, IC_mGRF4_12_check, TO_mGRF4_12_check = CheckSteps(IC_eGRF4_12, TO_eGRF4_12, A6_12_filtered, mGRF4_12_filtered, fs_Vicon)

# Participant 5
IC_eGRF5_8_check, TO_eGRF5_8_check, IC_mGRF5_8_check, TO_mGRF5_8_check = CheckSteps(IC_eGRF5_8, TO_eGRF5_8, A6_8_filtered, mGRF5_8_filtered, fs_Vicon)
IC_eGRF5_10_check, TO_eGRF5_10_check, IC_mGRF5_10_check, TO_mGRF5_10_check = CheckSteps(IC_eGRF5_10, TO_eGRF5_10, A6_10_filtered, mGRF5_10_filtered, fs_Vicon)
IC_eGRF5_12_check, TO_eGRF5_12_check, IC_mGRF5_12_check, TO_mGRF5_12_check = CheckSteps(IC_eGRF5_12, TO_eGRF5_12, A6_12_filtered, mGRF5_12_filtered, fs_Vicon)

# Participant 6
IC_eGRF6_8_check, TO_eGRF6_8_check, IC_mGRF6_8_check, TO_mGRF6_8_check = CheckSteps(IC_eGRF6_8, TO_eGRF6_8, A6_8_filtered, mGRF6_8_filtered, fs_Vicon)
IC_eGRF6_10_check, TO_eGRF6_10_check, IC_mGRF6_10_check, TO_mGRF6_10_check = CheckSteps(IC_eGRF6_10, TO_eGRF6_10, A6_10_filtered, mGRF6_10_filtered, fs_Vicon)
IC_eGRF6_12_check, TO_eGRF6_12_check, IC_mGRF6_12_check, TO_mGRF6_12_check = CheckSteps(IC_eGRF6_12, TO_eGRF6_12, A6_12_filtered, mGRF6_12_filtered, fs_Vicon)

# Participant 7
IC_eGRF7_8_check, TO_eGRF7_8_check, IC_mGRF7_8_check, TO_mGRF7_8_check = CheckSteps(IC_eGRF7_8, TO_eGRF7_8, A6_8_filtered, mGRF7_8_filtered, fs_Vicon)
IC_eGRF7_10_check, TO_eGRF7_10_check, IC_mGRF7_10_check, TO_mGRF7_10_check = CheckSteps(IC_eGRF7_10, TO_eGRF7_10, A6_10_filtered, mGRF7_10_filtered, fs_Vicon)
IC_eGRF7_12_check, TO_eGRF7_12_check, IC_mGRF7_12_check, TO_mGRF7_12_check = CheckSteps(IC_eGRF7_12, TO_eGRF7_12, A6_12_filtered, mGRF7_12_filtered, fs_Vicon)

```

```

# find peak GRF
def DetectActivePeak(GRF, A, fs, height=1, min_time_diff=None):

    if min_time_diff == None:
        min_i_diff = 1
    else:
        min_i_diff = min_time_diff * fs

    # Option 1: calculate the peaks from the GRF
    if len(GRF.shape) == 1:
        i_GRFpeak = scipy.signal.find_peaks(GRF, height=height, distance = min_i_diff)[0]

    elif len(GRF.shape) == 2:
        i_GRFpeak = scipy.signal.find_peaks(GRF[:, 2], height=height, distance = min_i_diff)[0]

```

```

# Option 2: calculate the peaks from the steps
elif len(GRF.shape) == 3:
    i_GRFpeak = []
    for i in range(GRF.shape[0]):
        i_GRFpeak.append(np.argmax(GRF[i, :, 2]))
    i_GRFpeak = np.array(i_GRFpeak)

# Compare the accelerations of the left and the right thigh to know if it is the left or the right leg
# Afterwards, just use the fact of consecutive alternating legs
Av_right_thigh = A[0, :, 2]
i_GRFpeak_right = []
Av_left_thigh = A[2, :, 2]
i_GRFpeak_left = []

if Av_right_thigh[0] > Av_left_thigh[0]:
    i_GRFpeak_right = [i_GRFpeak[i] for i in range(len(i_GRFpeak)) if i%2==0]
    i_GRFpeak_left = [i_GRFpeak[i] for i in range(len(i_GRFpeak)) if i%2==1]
elif Av_left_thigh[0] > Av_right_thigh[0]:
    i_GRFpeak_left = [i_GRFpeak[i] for i in range(len(i_GRFpeak)) if i%2==0]
    i_GRFpeak_right = [i_GRFpeak[i] for i in range(len(i_GRFpeak)) if i%2==1]

return np.array(i_GRFpeak), np.array(i_GRFpeak_left), np.array(i_GRFpeak_right)

```

7. Jump detection

```

def JumpDetector(step_t, fs, data, method):
    # https://pmc.ncbi.nlm.nih.gov/articles/PMC11257225/

    g = 9.81
    step_i = int(step_t*fs)

    if method == 'IMU':
        Av = data[4, :, 2]
    if method == 'GRF': #zorg ervoor dat de GRF al vermenigvuldigd is met M*g!
        GRFz = data[:, 2]

    marge_start = 1*fs
    marge_end = 1*fs
    indicesJ = [] # indices flight phase

    if method == 'IMU':
        for i in range(int(step_i+marge_start), int(len(Av)-marge_end-1)):
            if (np.mean(Av[i-step_i:i])) <= -g: # leave the ground when a < -g
                indicesJ.append(i-step_i) # all indices of the flight phase

    if method == 'GRF':
        for i in range(int(step_i+marge_start), int(len(GRFz)-marge_end-1)):
            if (np.mean(GRFz[i-step_i:i])) <= 15: # leave the ground when GRF < 15 (arbitrary treshold)
                indicesJ.append(i-step_i) # all indices of the flight phase

    startJ = []
    endJ = []
    for i in range(1, len(indicesJ)):
        if (indicesJ[i] - indicesJ[i-1]) > step_i+1: # when the difference is bigger than 1, we have IC an
            endJ.append(indicesJ[i-1])
            startJ.append(indicesJ[i]) # startJ will always be first
    startJ = np.array(startJ)
    endJ = np.array(endJ)

    # check for errors
    jump_length = int(2*fs) # duration of one jump, guess max 2s
    # make sure that we have the same amount of IC and T0, with IC first

```

```

i = 0
startJ_final = []
j = 0
endJ_final = []

while i < len(startJ) and j < len(endJ):
    if step_i < (endJ[j] - startJ[i]) <= jump_length:    # not too close or too much difference between
        startJ_final.append(startJ[i])
        i += 1
        endJ_final.append(endJ[j])
        j += 1
    elif (endJ[j]-startJ[i]) <= step_i:
        j += 1
    elif (endJ[j] - startJ[i]) > jump_length:
        i += 1

startJ_final = (np.array(startJ_final) - marge_start).astype(int)
endJ_final = (np.array(endJ_final) + marge_end).astype(int)
return startJ_final, endJ_final

```

```

def GetJumps(Start, End, GRF, fs): # be careful: IC and TO should be calculated on the same GRF
    # adjust the arrays beforehand, in order to start with initial contact
    """
    Divide the recorded trial in jumps.

    - Start: start of the jump
    - End: end of the jump
    - GRF: ground reaction force (measured or estimated)
    - fs: sampling frequency
    """

    jumps = []

    n_jumps = len(Start)
    jump_length = int(np.max(End-Start))

    for i in range(n_jumps-1):
        jump = GRF[Start[i]:End[i]] # one jump is from Start until End
        jumps.append(scipy.signal.resample(jump, jump_length)) # to compare the jumps more easily, make th

    t = 1/fs * np.arange(0, jump_length, 1)
    #t = np.linspace(0, 100, step_length) # Define the time-axis as a percentage of one full gait cycle

    return np.array(t), np.array(jumps)

```

```

step_t = 0.1
g = 9.81

# Participant 2
startJ_IMU2, endJ_IMU2 = JumpDetector(step_t, fs_Vicon, A2_jump_filtered, method='IMU')
startJ_mGRF2, endJ_mGRF2 = JumpDetector(step_t, fs_Vicon, mGRF2_jump_filtered*M2*g, method='GRF')

# Participant 3
startJ_IMU3, endJ_IMU3 = JumpDetector(step_t, fs_Vicon, A3_jump_filtered, method='IMU')
startJ_mGRF3, endJ_mGRF3 = JumpDetector(step_t, fs_Vicon, mGRF3_jump_filtered*M3*g, method='GRF')

# Participant 4
startJ_IMU4, endJ_IMU4 = JumpDetector(step_t, fs_Vicon, A4_jump_filtered, method='IMU')
startJ_mGRF4, endJ_mGRF4 = JumpDetector(step_t, fs_Vicon, mGRF4_jump_filtered*M4*g, method='GRF')

# Participant 5
startJ_IMU5, endJ_IMU5 = JumpDetector(step_t, fs_Vicon, A5_jump_filtered, method='IMU')

```

```

startJ_mGRF5, endJ_mGRF5 = JumpDetector(step_t, fs_Vicon, mGRF5_jump_filtered*M5*g, method='GRF')

# Participant 6
startJ_IMU6, endJ_IMU6 = JumpDetector(0.2, fs_Vicon, A6_jump_filtered, method='IMU')
startJ_mGRF6, endJ_mGRF6 = JumpDetector(step_t, fs_Vicon, mGRF6_jump_filtered*M6*g, method='GRF')

# Participant 7
startJ_IMU7, endJ_IMU7 = JumpDetector(step_t, fs_Vicon, A7_jump_filtered, method='IMU')
startJ_mGRF7, endJ_mGRF7 = JumpDetector(step_t, fs_Vicon, mGRF7_jump_filtered*M7*g, method='GRF')

```

✓ 8. Link-segment model

✓ 8.1 Build the model

✓ 8.1.1 Translate the accelerations to the CoM

Segment linear accelerations exported from the IMUs are expressed in the origin of each local coordinate system of the IMU on each segment. To apply these variables in the second law of Newton, a translation to the segment's centre of mass is required, defined as:

$$\mathbf{a}_i = \mathbf{a}_{i,o} + \dot{\boldsymbol{\omega}}_i \times \mathbf{r}_{oi} + \boldsymbol{\omega}_i \times (\boldsymbol{\omega}_i \times \mathbf{r}_{oi})$$

with $\mathbf{r}_{oi}$ the position vectors between the center of mass and the origin of each segment. Source: Karatsidis (2019)

Longitudinal center of mass positions are taken from the literature (David Winter or Paolo de Leva). For the sagittal and transversal coordinate, I assume the middle of the body segment's depth and width.

```

def PositionVectors(CoM, AM):

    # Define a matrix with position vectors
    # TO DO: improve by measuring the exact lengths on the pants
    # I assume now that the IMU is located in the middle of the body segment, on the lateral side

    # Based on the accelerations and for a right-handed coordinate system (see Vicon Nexus in the lab):
    # x_axis to the right: medio-lateral or width
    # y-axis backward: anterior-posterior or depth
    # z-axis downward: longitudinal or length
    # shift accelerations over a length r with a positive or negative sign

    w_t, d_t, l_t = AM[0, 0], AM[0, 1], AM[0, 2]
    w_s, d_s, l_s = AM[1, 0], AM[1, 1], AM[1, 2]
    w_p, d_p, l_p = AM[2, 0], AM[2, 1], AM[2, 2]

    CoM_t = CoM[0]
    CoM_s = CoM[1]
    CoM_sf = CoM[2]
    CoM_p = CoM[3]

    if AM.shape[0] == 4:
        w_f, d_f, l_f = AM[3, 0], AM[3, 1], AM[3, 2]
        CoM_f = CoM[4]

    # right thigh
    r1x = w_t/2          # move IMU more to the right for the IMU on the left leg
    r1y = 0

```

```

r1z = -(l_t/2 - (CoM_t * l_t))    # assumption: IMU is lower than CoM
# left thigh
r3x = -w_t/2
r3y = 0
r3z = -(l_t/2 - (CoM_t * l_t))

# left shank with foot included (IMU)
r2x = w_sf/2-1    # small correction, based on measurement on pants
r2y = 0
r2z = -(l_sf/2 - (CoM_sf * l_sf))    # CoM is located below the IMU    # Note: Should I remove the sign
# right shank with foot included (IMU)    # I think it is because
r4x = -w_sf/2
r4y = 0
r4z = -(l_sf/2 - (CoM_sf * l_sf))

# left shank (Vicon)
r2x_Vicon = w_s/2-1
r2y_Vicon = 0
r2z_Vicon = -(l_s/2 - (CoM_s * l_s))
# right shank (Vicon)
r4x_Vicon = -w_s/2
r4y_Vicon = 0
r4z_Vicon = -(l_s/2 - (CoM_s * l_s))

# pelvis
r5x = 0 # Assume that the IMU is in the middle of the pelvis
r5y = -d_p/2
r5z = -CoM_p * l_p

# Note: the matrix R has 5 rows (body segment) and 3 columns (position vectors)
R = (np.array([[r1x, r1y, r1z], [r2x, r2y, r2z], [r3x, r3y, r3z], [r4x, r4y, r4z], [r5x, r5y, r5z]]))

if AM.shape[0] == 4:
    # left foot (Vicon)
    r6x_Vicon = 0    # Assume that by averaging the toe, heel and ankle marker, we are already in the CO
    r6y_Vicon = d_f/2 - (CoM_f * d_f)    # this is zero
    r6z_Vicon = 0
    # right foot (Vicon)
    r7x_Vicon = 0
    r7y_Vicon = d_f/2 - (CoM_f * d_f)    # this is zero
    r7z_Vicon = 0

    R_Vicon = (np.array([[r1x, r1y, r1z], [r2x_Vicon, r2y_Vicon, r2z_Vicon], [r3x, r3y, r3z], [r4x_Vicon, r4y_Vicon, r4z_Vicon], [r5x, r5y, r5z]]))
    R = R_Vicon

return R

```

```

def TranslateAccelerations(A, W, AM, method):

```

```

    if method == "Winter":
        # CoM / Segment length (proximal: close to the body)
        CoM_p_W = 62.6 /100    # HAT (Head, Arms, Trunk)
        CoM_t_W = 43.3 /100
        CoM_sf_W = 60.6 /100
        CoM_s_W = 43.3 /100
        CoM_f_W = 50.0 /100
        CoM = [CoM_t_W, CoM_s_W, CoM_sf_W, CoM_p_W, CoM_f_W]

    if method == "De Leva women":
        CoM_p_W = 62.6 /100
        # thigh
        CoM_t_FdL = 36.12 /100
        # shank
        CoM_s_FdL = 43.52 /100
        CoM_sf_W = 60.6 /100

```

```

# foot
CoM_f_FdL = 40.14 /100
CoM= [CoM_t_FdL, CoM_s_FdL, CoM_sf_W, CoM_p_W, CoM_f_FdL] # no info for CoM of shank+foot and pelv

if method == "De Leva men":
    CoM_p_W = 62.6 /100
    CoM_t_MdL = 40.95 /100
    CoM_s_MdL = 43.95 /100
    CoM_sf_W = 60.6 /100
    CoM_f_MdL = 44.15 /100
    CoM = [CoM_t_MdL, CoM_s_MdL, CoM_sf_W, CoM_p_W, CoM_f_MdL] # no info for CoM of shank+foot and pel

# Translate accelerations
A_trans = np.zeros_like(A)
alpha = np.diff(W, n=1, axis=1, prepend=0) # angular acceleration (equals the derivative of the anglu
R = PositionVectors(CoM, AM)

for i in range(A.shape[0]): # for each body segment
    # Note: crossproduct is anti-commutative: a x b = - (b x a)
    A_trans[i, :, :] = A[i, :, :] + np.cross(alpha[i, :, :], R[i]) + np.cross(W[i, :, :], np.cross(W[i,
return A_trans

```

✓ 8.1.2 Determine the weight factors

GRF is a measure of external load on the whole body. The weight factors express the mass of each body segment as a percentage of the total body mass. Because the model represents the whole body, the sum of all weight factors should be one. Therefore, the weight factors of the human body are constrained to the condition:

$$2WF_t + 2WF_s + WF_p = 1$$

```

# Flight phase correction: acceleration in z-direction should be bigger than zero
# Otherwise, the estimated GRF would be negative
# As such, we avoid negative weight factors

```

```

def CorrectAccelerations(A):
    A_corr = copy.deepcopy(A)
    A_corr[:, :, 2][A_corr[:, :, 2] <= -g] = -g
    return A_corr

```

✓ 8.1.2.1 Option 1: Weight factors of de Leva

```

# weight factors for women
WF_Fthigh_dL = 14.78 /100 # percentage
WF_Fshank_dL = 4.81 /100
WF_Ffoot_dL = 1.29 /100
WF_Fpelvis_dL = 1 - 2*(WF_Fthigh_dL + WF_Fshank_dL + WF_Ffoot_dL)

WF_F_dL = np.array([WF_Fthigh_dL, WF_Fshank_dL+WF_Ffoot_dL, WF_Fthigh_dL, WF_Fshank_dL+WF_Ffoot_dL, WF_Fpelvis_dL, WF_Ffoot_dL])
WF_F_dL_Vicon = np.array([WF_Fthigh_dL, WF_Fshank_dL, WF_Fthigh_dL, WF_Fshank_dL, WF_Fpelvis_dL, WF_Ffoot_dL])

# weight factors for men
WF_Mthigh_dL = 14.16 /100
WF_Mshank_dL = 4.33 /100
WF_Mfoot_dL = 1.37 /100
WF_Mpelvis_dL = 1 - 2*(WF_Mthigh_dL + WF_Mshank_dL + WF_Mfoot_dL)

WF_M_dL = np.array([WF_Mthigh_dL, WF_Mshank_dL+WF_Mfoot_dL, WF_Mthigh_dL, WF_Mshank_dL+WF_Mfoot_dL, WF_Mpelvis_dL, WF_Mfoot_dL])
WF_M_dL_Vicon = np.array([WF_Mthigh_dL, WF_Mshank_dL, WF_Mthigh_dL, WF_Mshank_dL, WF_Mpelvis_dL, WF_Mfoot_dL])

```

```
print('The weight factors for men from the table of de Leva are:', WF_M_dL, 'with sum', np.sum(WF_M_dL), 'a
print('The weight factors for women from the table of de Leva are:', WF_F_dL, 'with sum', np.sum(WF_F_dL),
```

↪ The weight factors for men from the table of de Leva are: [0.1416 0.057 0.1416 0.057 0.6028] with sum
The weight factors for women from the table of de Leva are: [0.1478 0.061 0.1478 0.061 0.5824] with s

✓ 8.1.2.2 Option 2: Weight factors of David Winter

```
WF_pelvis_W = 67.8 /100
WF_thigh_W = 10 /100
WF_shank_W = 4.65 /100
WF_foot_W = 1.45 /100
```

```
WF_W = np.array([WF_thigh_W, WF_shank_W+WF_foot_W, WF_thigh_W, WF_shank_W+WF_foot_W, WF_pelvis_W])
WF_W_Vicon = np.array([WF_thigh_W, WF_shank_W, WF_thigh_W, WF_shank_W, WF_pelvis_W, WF_foot_W, WF_foot_W])
```

```
print('The weight factors from the table of David Winter are:', WF_W, 'with sum', np.sum(WF_W), 'and', WF_W
```

↪ The weight factors from the table of David Winter are: [0.1 0.061 0.1 0.061 0.678] with sum 1.0 and

✓ 8.1.2.3 Option 3: Least-squares estimation (with artificially added constraint)

Notice that the GRF estimation model is a linear model. In that case, the optimal parameters based on a given dataset are:

$$\beta = (X^T X)^{-1} X^T Y$$

Source: https://en.wikipedia.org/wiki/Linear_regression

```
def EstimateWF_LSE(M, A, mGRF, constraint=True, printer=False):

    if constraint == True:
        # adding additional constraint that 2*Wft + 2*WFs + Wfp = 1 with np.ones
        Ax = np.concatenate((np.expand_dims(np.ones(A.shape[0])), axis=1), A[:, :, 0]), axis=1)
        Ay = np.concatenate((np.expand_dims(np.ones(A.shape[0])), axis=1), A[:, :, 1]), axis=1)
        Az = np.concatenate((np.expand_dims(np.ones(A.shape[0])), axis=1), A[:, :, 2]), axis=1)

        mGRFx = np.append([1], mGRF[:, 0]*M*g)
        mGRFy = np.append([1], mGRF[:, 1]*M*g)
        mGRFz = np.append([1], mGRF[:, 2]*M*g - M*g)

    if constraint == False:
        Ax = A[:, :, 0]
        Ay = A[:, :, 1]
        Az = A[:, :, 2]

        mGRFx = mGRF[:, 0]*M*g
        mGRFy = mGRF[:, 1]*M*g
        mGRFz = mGRF[:, 2]*M*g - M*g

    # optimisation via linear regression
    # Note: From the implementation point of view, this is just the Least Squares Method (scipy.linalg.lstsq)
    # Note: Python does not want to fit matrix A on matrix mGRF (only 1D input)
    LSEx = sklearn.linear_model.LinearRegression(fit_intercept=False, positive=True).fit(Ax.T, mGRFx)
    Wfx_LSE = LSEx.coef_ /M
    LSEy = sklearn.linear_model.LinearRegression(fit_intercept=False, positive=True).fit(Ay.T, mGRFy)
    Wfy_LSE = LSEy.coef_ /M
    LSEz = sklearn.linear_model.LinearRegression(fit_intercept=False, positive=True).fit(Az.T, mGRFz)
```

```

WFz_LSE = LSEz.coef_ / M

if printer == True:
    print('Optimal weight factors for the x-direction:', WFx_LSE, ', with sum', np.sum(WFx_LSE))
    print('Optimal weight factors for the y-direction:', WFi_LSE, ', with sum', np.sum(WFi_LSE))
    print('Optimal weight factors for the z-direction:', WFz_LSE, ', with sum', np.sum(WFz_LSE))

return WFx_LSE, WFi_LSE, WFz_LSE

```

✓ 8.1.2.4 Option 4: own implementation of stochastic gradient descent

See notes for the calculations.

I tried to create my own stochastic gradient descent method, because it is not possible to add the constraint that the coefficients should sum to 1 in the predefined algorithms.

```

class customGDRegressor():
    def __init__(self, M, A, initial_WF, n_iter, learning_rate, mGRF, fs, method):

        self.M = M
        self.n_iter = n_iter
        self.learning_rate = learning_rate

        # initialize the weight factors
        self.WF = initial_WF # The better the first guess for the weight factors, the less iterations are r
                             # For example, use the weight factors of David Winter or de Leva

        self.A = A
        self.fs = fs
        self.mGRFv = mGRF[:, 2]*M*g - M*g

        self.method = method

        if self.method == 'peaks':
            self.i_mGRFpeak, self.i_mGRFpeak_left, self.i_mGRFpeak_right = DetectActivePeak(self.mGRFv, sel

    def cost_function(self):
        """
        Calculate the cost for the given data and parameters.

        Parameters
        -----
        - M: mass of the participant
        - A : {array-like}, shape (n_samples, n_features)
            Training data
        - mGRF : ndarray of shape (n_samples,)
            Target values
        - WF : float
            Coefficients or weight factors
        """

        g = 9.81

        WF1, WF2, WF3, WF4 = self.WF[0], self.WF[1], self.WF[2], self.WF[3]
        Av1, Av2, Av3, Av4, Av5 = self.A[0, :, 2], self.A[1, :, 2], self.A[2, :, 2], self.A[3, :, 2], self.
        eGRFv = WF1*self.M*Av1 + WF2*self.M*Av2 + WF3*self.M*Av3 + WF4*self.M*Av4 + (1-WF1-WF2-WF3-WF4)*sel
        RMSE = np.sqrt(1/len(self.mGRFv) * np.sum((eGRFv - self.mGRFv)**2))

        return RMSE

    def gradient_descent_step(self):

```

```

g = 9.81

WF1, WF2, WF3, WF4 = self.WF[0], self.WF[1], self.WF[2], self.WF[3]
Av1, Av2, Av3, Av4, Av5 = self.A[0, :, 2], self.A[1, :, 2], self.A[2, :, 2], self.A[3, :, 2], self.
eGRFv = WF1*self.M*Av1 + WF2*self.M*Av2 + WF3*self.M*Av3 + WF4*self.M*Av4 + (1-WF1-WF2-WF3-WF4)*sel

if self.method == 'no symmetry':
    WF1_new = WF1 - self.learning_rate * np.sum(2 * self.M*(Av1 - Av5) * (eGRFv - self.mGRFv))/len(
        # be careful: the update rule of W1 uses the old values of WF2, WF3 and WF4
    WF2_new = WF2 - self.learning_rate * np.sum(2 * self.M*(Av2 - Av5) * (eGRFv - self.mGRFv))/len(
    WF3_new = WF3 - self.learning_rate * np.sum(2 * self.M*(Av3 - Av5) * (eGRFv - self.mGRFv))/len(
    WF4_new = WF4 - self.learning_rate * np.sum(2 * self.M*(Av4 - Av5) * (eGRFv - self.mGRFv))/len(

if self.method == 'symmetry':
    WF1_new = WF1 - self.learning_rate * 1/len(self.mGRFv) * np.sum(2 * self.M*(Av1 + Av3 - 2*Av5)
    WF2_new = WF2 - self.learning_rate * 1/len(self.mGRFv) * np.sum(2 * self.M*(Av2 + Av4 - 2*Av5)
    WF3_new = WF1_new
    WF4_new = WF2_new

if self.method == 'peaks':
    i_eGRFpeak_corr = []
    i_mGRFpeak_corr = []
    # idea: use quaternions to find peaks OR scipy.signal.find_peaks
    i_eGRFpeak, i_eGRFpeak_left, i_eGRFpeak_right = DetectActivePeak(eGRFv, self.A, self.fs, height
    # compare the same peaks of eGRF and mGRF
    for i_e in i_eGRFpeak:
        for i_m in self.i_mGRFpeak:
            if abs(i_e - i_m) < 50: # 100 is marge
                i_eGRFpeak_corr.append(i_e)
                i_mGRFpeak_corr.append(i_m)
    WF1_new = WF1 - self.learning_rate * np.sum(2 * self.M*(Av1 - Av5)[i_eGRFpeak_corr] * (eGRFv[i_
    WF2_new = WF2 - self.learning_rate * np.sum(2 * self.M*(Av2 - Av5)[i_eGRFpeak_corr] * (eGRFv[i_
    WF3_new = WF3 - self.learning_rate * np.sum(2 * self.M*(Av3 - Av5)[i_eGRFpeak_corr] * (eGRFv[i_
    WF4_new = WF4 - self.learning_rate * np.sum(2 * self.M*(Av4 - Av5)[i_eGRFpeak_corr] * (eGRFv[i_

self.WF = [WF1_new, WF2_new, WF3_new, WF4_new]
return self.WF

def fit(self):
    costs = [np.inf]
    for i in range(self.n_iter):
        self.WF = self.gradient_descent_step()

        if (i % 50 == 0) or (i == (self.n_iter - 1)):
            cost = self.cost_function()
            print("Iteration {}/{}", RMSE : {}".format(i, self.n_iter, cost))

        if cost > costs[-1]:
            print("Divergence of the weighting factors!")
            return self.WF

        costs.append(cost)

def predict(self):
    A1, A2, A3, A4, A5 = self.A[0, :, :], self.A[1, :, :], self.A[2, :, :], self.A[3, :, :], self.A[4,
    WF1, WF2, WF3, WF4 = self.WF[0], self.WF[1], self.WF[2], self.WF[3]
    eGRF = (WF1*self.M*A1 + WF2*self.M*A2 + WF3*self.M*A3 + WF4*self.M*A4 + (1-WF1-WF2-WF3-WF4)*self.M*
    return eGRF

def coefficients(self):
    WF1, WF2, WF3, WF4 = self.WF[0], self.WF[1], self.WF[2], self.WF[3]
    WF5 = 1 - WF1 - WF2 - WF3 - WF4
    res = [WF1, WF2, WF3, WF4, WF5]

```

```
print('The optimised weight factors for the z-direction are:', res, ', with sum', np.sum(res))
return np.array(res)
```

✓ 8.1.2.5 Option 5: Bayesian estimator with ridge regression

BayesianRidge estimates a probabilistic model of the regression problem. The prior for the coefficient is given by a spherical Gaussian.

When fitting a curve with a polynomial by Bayesian ridge regression, the selection of initial values of the regularization parameters (alpha, lambda) may be important. This is because the regularization parameters are determined by an iterative procedure that depends on initial values.

When starting from the default values (alpha_init = 1.90, lambda_init = 1.), the bias of the resulting curve is large, and the variance is small. So, lambda_init should be relatively small (1.e-3) so as to reduce the bias. (=> Bias-variance trade-off)

Due to the Bayesian framework, the weights found are slightly different to the ones found by Ordinary Least Squares. However, Bayesian Ridge Regression is more robust to ill-posed problems.

```
def EstimateWF_BR(M, A, mGRF, constraint=True, printer=False):

    if constraint == True:
        # adding additional constraint that 2*Wf + 2*Wfs + Wfp = 1 with np.ones
        Ax = np.concatenate((np.expand_dims(np.ones(A.shape[0]), axis=1), A[:, :, 0]), axis=1)
        Ay = np.concatenate((np.expand_dims(np.ones(A.shape[0]), axis=1), A[:, :, 1]), axis=1)
        Az = np.concatenate((np.expand_dims(np.ones(A.shape[0]), axis=1), A[:, :, 2]), axis=1)

        mGRFx = np.append([1], mGRF[:, 0]*M*g)
        mGRFy = np.append([1], mGRF[:, 1]*M*g)
        mGRFz = np.append([1], mGRF[:, 2]*M*g - M*g)

    if constraint == False:
        Ax = A[:, :, 0]
        Ay = A[:, :, 1]
        Az = A[:, :, 2]

        mGRFx = mGRF[:, 0]*M*g
        mGRFy = mGRF[:, 1]*M*g
        mGRFz = mGRF[:, 2]*M*g - M*g

    # optimisation via linear regression
    # Note: From the implementation point of view, this is just the Least Squares Method (scipy.linalg.lsts
    # Note: Python does not want to fit matrix A on matrix mGRF (only 1D input)
    BR = sklearn.linear_model.BayesianRidge(fit_intercept=False, tol=1e-6, alpha_init=1.0, lambda_init=1e-6)
    BR.fit(Ax.T, mGRFz)
    WFx_BR = BR.coef_ /M
    BR.fit(Ay.T, mGRFz)
    Wfy_BR = BR.coef_ /M
    BR.fit(Az.T, mGRFz)
    WFz_BR = BR.coef_ /M

    # only for z-direction
    eps_BR = BR.alpha_ # estimated precision of the noise
    prec_BR = BR.lambda_ # estimated precision of the weights
    n_BR = BR.n_iter_ # actual number of iterations to reach the stopping criterion

    if printer == True:
        print('The optimised weight factors for the x-direction are:', WFx_BR, ', with sum', np.sum(WFx_BR))
        print('The optimised weight factors for the y-direction are:', Wfy_BR, ', with sum', np.sum(Wfy_BR))
        print('The optimised weight factors for the z-direction are:', WFz_BR, ', with sum', np.sum(WFz_BR))
        print('Estimated precision of the noise (z-direction):', eps_BR)
```

```

print('Estimated precision of the weights (z-direction):', prec_BR)
print('Actual number of iterations to reach the stopping criterion (z-direction):', n_BR)

return WfX_BR, WfY_BR, WfZ_BR

```

✓ 8.1.2.6 Option 6: Anthropometric measurements

Calculate the volume of the lower limbs by simplifying them to a truncated cone:

$$V = \frac{1}{3} * \pi * h * (r^2 + r * R + R^2)$$

with

- R : the radius of the base of the original cone (bottom surface)
- r : the radius of the top surface
- h : the height of the truncated cone.

The mass then follows from the density, which is taken from the literature (David Winter):

$$m = \rho * V$$

with

- ρ : density
- V : volume

```

def EstimateWF_AM(AM, M, printer=False):

    rho_s = 1.09 * 1000
    rho_t = 1.05 * 1000 # see David Winter
    rho_f = 1.10 * 1000

    # Calculate the volume of each segment from the anthropometric measurements
    # Estimate the volume as a cylinder
    # Less accurate than truncated cylinder, but less measurements required (only length)

    if AM.shape[0] == 3:
        w_t, d_t, l_t = AM[0, :]
        w_sf, d_sf, l_sf = AM[1, :]
        w_p, d_p, l_p = AM[2, :]

        V_t = np.pi * (w_t/2)**2 * l_t
        V_sf = np.pi * (w_sf/2)**2 * l_sf

        m_t = rho_t * V_t
        m_sf = rho_s * V_sf

        WF_t = m_t/M
        WF_sf = m_sf/M
        WF_p = 1-2*WF_sf-2*WF_t

        WF_AM = np.array([WF_t, WF_sf, WF_t, WF_sf, WF_p])

    if AM.shape[0] == 4:
        w_t, d_t, l_t = AM[0, :]
        w_s, d_s, l_s = AM[1, :]
        w_p, d_p, l_p = AM[2, :]
        w_f, d_f, l_f = AM[3, :]

        V_t = np.pi * (w_t/2)**2 * l_t
        V_s = np.pi * (w_s/2)**2 * l_s
        V_f = np.pi * (l_f/2)**2 * d_f

        m_t = rho_t * V_t
        m_s = rho_s * V_s

```

```

m_s = rho_s * V_s
m_f = rho_f * V_f

WF_t = m_t/M
WF_s = m_s/M
WF_f = m_f/M
WF_p = 1-2*WF_s-2*WF_t-2*WF_f

WF_AM = np.array([WF_t, WF_sf, WF_t, WF_sf, WF_p, WF_f, WF_f])

if printer == True:
    print('The weight factors obtained via anthropometric measurements are:', WF_AM, ', with sum', np.sum(WF_AM))

return WF_AM

```

```

# Participant 0
WF_AM = EstimateWF_AM(AM0, M0, printer=True)
print('-'*150)

# Participant 2
WF2_AM = EstimateWF_AM(AM2, M2, printer=True)
print('-'*150)

# Participant 3
WF3_AM = EstimateWF_AM(AM3, M3, printer=True)
print('-'*150)

# Participant 4
WF4_AM = EstimateWF_AM(AM4, M4, printer=True)
print('-'*150)

# Participant 5
WF5_AM = EstimateWF_AM(AM5, M5, printer=True)
print('-'*150)

# Participant 6
WF6_AM = EstimateWF_AM(AM6, M6, printer=True)
print('-'*150)

# Participant 7
WF7_AM = EstimateWF_AM(AM7, M7, printer=True)
print('-'*150)

```

➡ The weight factors obtained via anthropometric measurements are: [0.11413173 0.12596398 0.11413173 0.12596398 0.06420992 0.07538605 0.06420992 0.07538605 0.06761742 0.07938664 0.06761742 0.07938664 0.07511758 0.08819226 0.07511758 0.08819226 0.08215516 0.09645477 0.08215516 0.09645477 0.07442547 0.08737968 0.07442547 0.08737968 0.06979862 0.0819475 0.06979862 0.0819475]



```

def CalculateVolume(h, r, R):
    """
    Calculate the volume of the lower limbs by simplifying them to a truncated cone.
    - R: the radius of the base of the original cone (bottom surface)
    - r: the radius of the top surface
    """

```

```
- h: the height of the truncated cone.
"""

V = 1/3 * np.pi * h * (r**2 + r*R + R**2)

return V
```

```
rho_s = 1.09 * 1000 # kg/m^3?
rho_t = 1.05 * 1000 # see David Winter
rho_f = 1.10 * 1000

# Calculate the volume of each segment from the anthropometric measurements
r_t = w_t_lower / 2 # m
R_t = w_t_upper / 2
h_t = l_t
V_t = CalculateVolume(h_t, r_t, R_t) # m^3

r_s = w_sf_upper / 2
R_s = w_sf_lower / 2
h_s = l_sf - l_f
V_s = CalculateVolume(h_s, r_s, R_s)

r_f = (2 / 2) / 100 # from cm to m
R_f = (9.5 / 2) / 100
h_f = d_f
V_f = CalculateVolume(h_f, r_f, R_f)
```

```
# Calculate the mass and weight factor of each segment
m_t = V_t * rho_t
WF_t = m_t / M0

m_s = V_s * rho_s # m^3 * kg / m^3
m_f = V_f * rho_f
WF_s = (m_s + m_f) / M0

# m_p = M - 2*(m_t + m_s + m_f)
# WF_p = m_p / M
WF_p = 1 - 2*(WF_t + WF_s)

WF_AM_extra = np.array([WF_t, WF_s, WF_t, WF_s, WF_p])
print('The weight factors obtained via anthropometric measurements are:', WF_AM_extra, ', with sum', np.sum
```

↔ The weight factors obtained via anthropometric measurements are: [0.12307937 0.04338095 0.12307937 0.04



✓ 8.1.2.7 Comparison of weight factors

```
def CompareWF(WFz_LSE, WFz_LSE_trans, WFz_SGD_nosymm, WFz_SGD_nosymm_trans, WFz_SGD_symm, WFz_SGD_symm_tran

WF = [WFz_LSE, WFz_LSE_trans, WFz_SGD_nosymm, WFz_SGD_nosymm_trans, WFz_SGD_symm, WFz_SGD_symm_trans, W

print('LSE (original):', np.around(WFz_LSE, 4))
print('LSE (translated):', np.around(WFz_LSE_trans, 4))
print('SGD no symmetry (original):', np.around(WFz_SGD_nosymm, 4))
print('SGD no symmetry (translated):', np.around(WFz_SGD_nosymm_trans, 4))
print('SGD symmetry (original):', np.around(WFz_SGD_symm, 4))
print('SGD symmetry (translated):', np.around(WFz_SGD_symm_trans, 4))
print('SGD peaks (original):', np.around(WFz_SGD_peaks, 4))
print('SGD peaks (translated):', np.around(WFz_SGD_peaks_trans, 4))
print('BR (original):', np.around(WFz_BR, 4))
print('BR (translated):', np.around(WFz_BR_trans, 4))
print('AM:', np.around(WF_AM, 4))
```

```

print('-'*75)
print('De Leva (for men):',WF_MdL)
print('De Leva (for women):',WF_FdL)
print('Winter:', WF_W)

return WF

```

✓ 8.1.3 Create the link-segment model

✓ 8.1.3.1 Ground reaction force

Major assumption that we make is that the GRF is the only significant external load present.

$$GRF = \sum_{i=1}^N m_i(\mathbf{a}_i - \mathbf{g})$$

```

def LinkSegmentModel(M, WF, A, FlightPhaseCorr=True):    # Is the model independent of the mass?

    """
    Make an estimation of the ground reaction force, under the assumption that the IMU has measured the acc
    - M: mass of the participant
    - WF: weight factors
    - A: acceleration of each body segment (in column, for example position 0 is left thigh, ... ), measure
    """

    # only one body segment
    if len(A.shape) == 2:
        A_copy = copy.deepcopy(A) + G
        if FlightPhaseCorr==True:
            A_copy[:, 2][A_copy[:, 2] <= 0] = 0
            eGRFxyz = (WF * M * A_copy) / (M*g)

    # multiple body segments
    elif len(A.shape) == 3:
        A_copy = copy.deepcopy(A)
        if FlightPhaseCorr==True:
            A_copy[:, :, 2][A_copy[:, :, 2] <= -g] = -g
        F = np.zeros_like(A)
        for i in range(A.shape[0]):
            F[i, :] = WF[i] * M * A_copy[i, :, :]
        eGRFxyz = (np.sum(F, axis=0) + M*G) / (M*g)

    eGRFres = np.sqrt(eGRFxyz[:, 0]**2 + eGRFxyz[:, 1]**2 + eGRFxyz[:, 2]**2)
    eGRF = np.concatenate((eGRFxyz, np.expand_dims(eGRFres, axis=1)), axis=1)

    return eGRF

```

```

def CompareLinkSegmentModel(M, WF, A, A_trans):

    WFz_LSE, WFz_LSE_trans, WFz_SGD_nosymm, WFz_SGD_nosymm_trans, WFz_SGD_symm, WFz_SGD_symm_trans, WFz_SGD

    eGRF_LSE = LinkSegmentModel(M, WFz_LSE, A)
    eGRF_LSE_trans = LinkSegmentModel(M, WFz_LSE_trans, A_trans)
    eGRF_SGD_nosymm = LinkSegmentModel(M, WFz_SGD_nosymm, A)
    eGRF_SGD_nosymm_trans = eGRF_W = LinkSegmentModel(M, WFz_SGD_nosymm_trans, A_trans)
    eGRF_SGD_symm = LinkSegmentModel(M, WFz_SGD_symm, A)
    eGRF_SGD_symm_trans = LinkSegmentModel(M, WFz_SGD_symm_trans, A_trans)
    eGRF_SGD_peaks = LinkSegmentModel(M, WFz_SGD_peaks, A)
    eGRF_SGD_peaks_trans = LinkSegmentModel(M, WFz_SGD_peaks_trans, A_trans)
    eGRF_BR = LinkSegmentModel(M, WFz_BR, A)

```

```

eGRF_BR_trans = LinkSegmentModel(M, WFz_BR_trans, A_trans)
eGRF_W = LinkSegmentModel(M, WF_W, A)
eGRF_W_trans = LinkSegmentModel(M, WF_W, A_trans)
eGRF_MdL = LinkSegmentModel(M, WF_MdL, A)
eGRF_MdL_trans = LinkSegmentModel(M, WF_MdL, A_trans)
eGRF_FdL = LinkSegmentModel(M, WF_FdL, A)
eGRF_FdL_trans = LinkSegmentModel(M, WF_FdL, A_trans)

eGRF = [eGRF_LSE, eGRF_LSE_trans, eGRF_SGD_nosymm, eGRF_SGD_nosymm_trans, eGRF_SGD_symm, eGRF_SGD_symm_trans]

return eGRF

```

8.1.3.2 Ground reaction moment

In a similar way, we calculate the total external moment from Euler's equation:

$$\mathbf{GRM} = \sum_{i=1}^N [\mathbf{J}_i \dot{\boldsymbol{\omega}}_i + \boldsymbol{\omega}_i \times (\mathbf{J}_i \boldsymbol{\omega}_i)] - \sum_{i=1}^N \sum_{j=1}^{K_i} (\mathbf{r}_{ij} \times \mathbf{F}_{ij})$$

where $\mathbf{GRM}$ denotes the total three-dimensional external moment, K_i the number of end points in each segment (joints and external contact points), $\boldsymbol{\omega}_i$ and $\dot{\boldsymbol{\omega}}_i$ the angular velocities and angular accelerations of each segment, respectively. The inertia tensor around the center of mass of each segment is denoted by $\mathbf{J}_i$, the position vectors between the center of mass and the end points denoted by $\mathbf{r}_{ij}$ and the resultant force in the end points of each segment described by $\mathbf{F}_{ij}$. All variables are expressed in the global coordinate system of the IMC system.

Source: Karatsidis and https://en.wikipedia.org/wiki/Radius_of_gyration

8.1.3.2.1 Define inertia tensor

```

# Option 0: reference values
# see paper: G. Nikolova, D. Dantchev, and M. Tsveov, 'NEW RESULTS FOR THE MASS-INERTIAL PARAMETERS OF THE

Ixx_t = 1999.4 # kg*cm^2
Iyy_t = 1997.8
Izz_t = 413.4
J_t = np.diag((Ixx_t, Iyy_t, Izz_t)) / 10000 # from kg*cm^2 to kg*m^2

Ixx_s = 371.0
Iyy_s = 385.0
Izz_s = 64.6
J_s = np.diag((Ixx_s, Iyy_s, Izz_s)) / 10000

Ixx_f = 40.0
Iyy_f = 44.4
Izz_f = 10.3
J_f = np.diag((Ixx_f, Iyy_f, Izz_f)) / 10000

Ixx_p = 705.2 + 819.1 + 525.0 + 293.9 # calculated as the sum of the upper torso, middle torso, lower torso
Iyy_p = 1725.6 + 1280.8 + 656.8 + 272.1
Izz_p = 1454.5 + 1203.1 + 529.4 + 202.4
J_p = np.diag((Ixx_p, Iyy_p, Izz_p)) / 10000

J_ref = np.array([J_t, J_s, J_t, J_s, J_p, J_f, J_f])

print(J_ref)

```

```

[[[0.19994 0.      0.      ]
  [0.      0.19978 0.      ]
  [0.      0.      0.04134]]

  [[0.0371  0.      0.      ]
  [0.      0.0385  0.      ]
  [0.      0.      0.0103 ]]]

```

```

[0.      0.      0.00646]]

[[0.19994 0.      0.      ]
 [0.      0.19978 0.      ]
 [0.      0.      0.04134]]

[[0.0371 0.      0.      ]
 [0.      0.0385 0.      ]
 [0.      0.      0.00646]]

[[0.23432 0.      0.      ]
 [0.      0.39353 0.      ]
 [0.      0.      0.33894]]

[[0.004 0.      0.      ]
 [0.      0.00444 0.      ]
 [0.      0.      0.00103]]

[[0.004 0.      0.      ]
 [0.      0.00444 0.      ]
 [0.      0.      0.00103]]]

```

Option 1: De Leva (radius of gyration)

```

def CalculateInertiaTensor(M, WF, AM_Vicon, gender):

    w_t, d_t, l_t = AM_Vicon[0, :]
    w_s, d_s, l_s = AM_Vicon[1, :]
    w_p, d_p, l_p = AM_Vicon[2, :]
    w_f, d_f, l_f = AM_Vicon[3, :]

    WF_t, WF_s, WF_p, WF_f = WF[0], WF[1], WF[4], WF[5]

    if gender == 'M':
        # for men
        # thigh
        RG_t_MdL = np.array([32.9, 32.9, 14.9]) / 100      # transverse (M-L) r, sagittal (A-P) r, longitudinal r
        L_t = np.array([l_t, l_t, l_t])
        J_t_MdL = np.diag((M * WF_t) * (L_t * RG_t_MdL) ** 2) # element-wise multiplication
        # shank
        RG_s_MdL = np.array([24.6, 25.1, 10.2]) / 100
        L_s = np.array([l_s, l_s, l_s])
        J_s_MdL = np.diag((M * WF_s) * (L_s * RG_s_MdL) ** 2) # kg*m^2
        # foot
        RG_f_MdL = np.array([24.5, 25.7, 12.4]) / 100
        L_f = np.array([w_f, d_f, l_f])
        J_f_MdL = np.diag((M * WF_f) * (L_f * RG_f_MdL) ** 2)
        # pelvis (note: I approximated the moment of inertia of the pelvis as the moment of inertia of the
        RG_p_MdL = np.array([34.7, 37.2, 19.1]) / 100
        L_p = np.array([l_p, l_p, l_p])
        J_p_MdL = np.diag((M * WF_p) * (L_p * RG_p_MdL) ** 2)

        J_MdL = np.array([J_t_MdL, J_s_MdL, J_t_MdL, J_s_MdL, J_p_MdL, J_f_MdL, J_f_MdL])
        return J_MdL

    if gender == 'F':
        # for women
        # thigh
        RG_t_FdL = np.array([36.4, 36.9, 16.2]) / 100      # transverse (M-L) r, sagittal (A-P) r, longitudinal r
        L_t = np.array([l_t, l_t, l_t])
        J_t_FdL = np.diag((M * WF_t) * (L_t * RG_t_FdL) ** 2) # element-wise multiplication
        # shank
        RG_s_FdL = np.array([26.3, 26.7, 9.2]) / 100
        L_s = np.array([l_s, l_s, l_s])
        J_s_FdL = np.diag((M * WF_s) * (L_s * RG_s_FdL) ** 2) # kg*m^2
        # foot

```

```

RG_f_FdL = np.array([27.9, 29.9, 13.9]) /100
L_f = np.array([l_f, l_f, l_f])
J_f_FdL = np.diag((M * WF_f) * (L_f * RG_f_FdL)**2)
# pelvis (note: I approximated the moment of inertia of the pelvis as the moment of inertia of the
RG_p_FdL = np.array([33.9, 35.7, 17.1]) /100
L_p = np.array([l_p, l_p, l_p])
J_p_FdL = np.diag((M * WF_p) * (L_p * RG_p_FdL)**2)

J_FdL = np.array([J_t_FdL, J_s_FdL, J_t_FdL, J_s_FdL, J_p_FdL, J_f_FdL, J_f_FdL])
return J_FdL

```

Option 2: Estimate the inertia of each body segments as the inertia of a cylinder

```

def CalculateInertiaTensor2(M, WF, AM):

    """
    Calculate the inertia tensor, given
    - M: total body mass of the participant
    - WF: weighting factors to define the mass of each body segment
    - AM: body dimensions from anthropometric measurements
    """

    # D: body dimensions from anthropometric measurements, in the format (radius, height)
    # Note: radius is approximated as the mean of the width and the depth
    if AM.shape[0] == 3:
        w_t, d_t, l_t = AM[0, :]
        w_s, d_s, l_s = AM[1, :]
        w_p, d_p, l_p = AM[2, :]
        D = np.array([(w_t/2+d_t/2)/2, l_t), ((w_s/2+d_s/2)/2, l_s), ((w_t/2+d_t/2)/2, l_t), ((w_s/2+d_s/2)/2, l_s)]

    if AM.shape[0] == 4:
        w_t, d_t, l_t = AM[0, :]
        w_s, d_s, l_s = AM[1, :]
        w_p, d_p, l_p = AM[2, :]
        w_f, d_f, l_f = AM[3, :]
        D = np.array([(w_t/2+d_t/2)/2, l_t), ((w_s/2+d_s/2)/2, l_s), ((w_t/2+d_t/2)/2, l_t), ((w_s/2+d_s/2)/2, l_s), ((w_p/2+d_p/2)/2, l_p), ((w_f/2+d_f/2)/2, l_f)]

    # inertia tensor
    J = []

    for i in range(D.shape[0]):
        r, h = D[i][0], D[i][1]
        Ixx = 1/12 * WF[i] * M * (3*r**2 + h**2)
        Iyy = Ixx
        Izz = 1/2 * WF[i] * M * r**2
        J.append(np.diag([Ixx, Iyy, Izz]))

    return np.array(J)

```

✓ 8.3.2.2. Calculate ground reaction moment

```

def LinkSegmentModelExtended(fs, M, AM, WF, J, X, V, A, W, W_rot, FlightPhaseCorr=True):

    """
    Make an estimation of the ground reaction force, under the assumption that the IMU has measured the acc
    The model starts from the raw data.
    - M: mass of the participant
    - WF: weight factors
    - A: acceleration of each body segment (in column, for example position 0 is left thigh, ... ), measure
    - W: angular velocity of each segment
    - J: inertia tensor around the center of mass of each segment
    - R: position vectors between the center of mass and the end points of each segment (not relevant in ou
    """

```

```

# correct for the flight phase
A_corr = copy.deepcopy(A)
W_copy = copy.deepcopy(W_rot)
if FlightPhaseCorr==True:
    A_corr[A[:, :, 2] <= -g] = -g

#W_copy[:, :, 0] *= (-1)
W_copy[:, :, 1] *= (-1)

# Define the moment arm
R = np.abs(np.array([X[2, :, :]-X[4, :, :], X[1, :, :]-X[2, :, :]+X[1, :, :]]/2, X[0, :, :]-X[4, :, :])
# R = [afstand van bovenbeen IMU tot IMU heup, afstand van onderbeen IMU tot "knie", analoog andere zij
# Niet helemaal zeker van het onderbeen, omdat de knie meer uitsteekt dan het gemiddelde van het bovenb

# Calculate the ground reaction force and the ground reaction moment
F = np.zeros_like(A)
F_corr = np.zeros_like(A)
MM1 = np.zeros_like(A)
MM2 = np.zeros_like(A)

# force on the CoM of a segment
for i in range(A.shape[0]):
    F[i, :, :] = WF[i] * M * A[i, :, :]
    F_corr[i, :, :] = WF[i] * M * A_corr[i, :, :]
    # MM1: from the angular velocities
    MM1[i, :, 0] = (np.dot(W[i, :, :], J[i, :, :]) + np.cross(R[i, :, :], M * V[i, :, :]))[:, 0]
    # MM1[i, :, 0] = (np.dot(np.gradient(W[i, :, :], 1/fs, axis=1), J[i, :, :]) + np.cross(W[i, :, :],
    MM1[i, :, 1] = (np.dot(np.gradient(W[i, :, :], 1/fs, axis=1), J[i, :, :]) + np.cross(W[i, :, :], np
    MM1[i, :, 2] = (np.dot(np.gradient(W_copy[i, :, :], 1/fs, axis=1), J[i, :, :]) + np.cross(W_copy[i,

eGRFxyz = (np.sum(F, axis=0) + M*g) / (M*g)
eGRFres = np.sqrt(eGRFxyz[:, 0]**2 + eGRFxyz[:, 1]**2 + eGRFxyz[:, 2]**2)
eGRF = np.concatenate((eGRFxyz, np.expand_dims(eGRFres, axis=1)), axis=1)
eGRFxyz_corr = (np.sum(F_corr, axis=0) + M*g) / (M*g)
eGRFres_corr = np.sqrt(eGRFxyz_corr[:, 0]**2 + eGRFxyz_corr[:, 1]**2 + eGRFxyz_corr[:, 2]**2)
eGRF_corr = np.concatenate((eGRFxyz_corr, np.expand_dims(eGRFres_corr, axis=1)), axis=1)

# part 2: from the applied forces
# for thigh: approximate with pelvis and tibia
# right thigh
MM2[0, :, :] = - (np.cross(R[0, :, :], F[4, :, :]+WF[4]*M*g) + np.cross(R[0, :, :], F[3, :, :]+WF[3]*M*
# left thigh
MM2[2, :, :] = - (np.cross(R[2, :, :], F[4, :, :]+WF[4]*M*g) + np.cross(R[0, :, :], F[1, :, :]+WF[1]*M*
# for tibia: approximate with thigh and ankle (-eGRF - ...)?
# left tibia
MM2[1, :, :] = - (np.cross(R[1, :, :], F[2, :, :]+WF[2]*M*g) + np.cross(R[1, :, :], eGRFxyz*M*g)) # Ffo
# right tibia
MM2[3, :, :] = - (np.cross(R[3, :, :], F[0, :, :]+WF[0]*M*g) + np.cross(R[3, :, :], eGRFxyz*M*g))
# for pelvis
MM2[4, :, :] = - (np.cross(R[4, :, :], F[1, :, :]+F[0, :, :]+F[2, :, :]+F[3, :, :] + (WF[0]+WF[1]+WF[2]

l = AM[0,2] + AM[1,2] + AM[2,2]
eGRM = np.sum(MM1+MM2, axis=0) / (M*g*l)
eGRM[:, 0] = 1.5*(eGRM[:, 0]*(-1) + 0.1)
eGRM[:, 1] *= (-1)
eGRM[:, 2] /= 10

if FlightPhaseCorr == True:
    return eGRF_corr, eGRM

if FlightPhaseCorr == False:
    return eGRF, eGRM

```

✓ 8.1.3.3 Centre of Pressure

```
def CalculateCoP(eGRF, eGRM):

    eGRMx, eGRMy, eGRMz = eGRM[:, 0], eGRM[:, 1], eGRM[:, 2]
    eGRFx, eGRFy, eGRFz = eGRF[:, 0], eGRF[:, 1], eGRF[:, 2]

    eCoPx = np.zeros(eGRF.shape[0])
    eCoPy = np.zeros(eGRF.shape[0])
    eCoPz = np.zeros(eGRF.shape[0])

    for i in range(eGRF.shape[0]):
        if eGRFz[i] >= 0.0001:
            eCoPx[i] = - eGRMy[i]/eGRFz[i]
            if abs(eCoPx[i]) >= 20:
                eCoPx[i] = 0
        if eGRFy[i] >= 0.0001:
            eCoPy[i] = eGRMx[i]/eGRFy[i]
            if abs(eCoPy[i]) >= 60:
                eCoPy[i] = 0

    eCoP = np.array([eCoPx, eCoPy, eCoPz]).T
    return eCoP
```

✓ 9. Two mass model (Udofa and Clark)

```
def TwoMassModel(M, A, fs, a, b, c):

    g = 9.81
    m1 = 0.08 * M
    m2 = 0.92 * M

    IC, T0 = StepDetector3(fs, A)
    IC, T0, IC, T0 = CheckSteps(IC, T0, IC, T0)

    tc = (T0 - IC) / fs # contact time = time in contact with the ground
    #tc = tc[:-1] # solve the problem that tc contains one element more than ta
    ta = (IC[1:] - T0[:-1]) / fs # aerial time = time in flight phase
    ta = np.append(ta, np.mean(ta)) # solve the problem that tc contains one element more than ta

    Ftot_avg = M * g * (tc + ta) / tc # idea: add this impulse/force to link-segment model?
    Jtot = Ftot_avg * tc # definition of impulse: impulse = force * time

    At_left = A[2, :, 2]
    At_right = A[0, :, 2]
    V = VelocityFromIMU(fs, A)
    V_filtered = FilterData(V, fs_Vicon, 3, [1, 6.5], 'bandpass', 1)
    Vs_left = V_filtered[1, :, 2] # vertical velocity of the right shank
    Vs_right = V_filtered[3, :, 2]
    # find the values where the velocity of the shank slows to zero after initial contact
    Z_left = np.where((Vs_left[1:]>0) & (np.sign(Vs_left[1:]*Vs_left[:-1])== -1))[0]
    Z_right = np.where((Vs_right[1:]>0) & (np.sign(Vs_right[1:]*Vs_right[:-1])== -1))[0]

    # check that Z > IC and corresponding Z for IC
    mean_stance = np.mean(T0 - IC) # duration of one stance phase

    i = 0
    dt1 = []
    j = 0
    dv1 = []
```

```

while (i < len(IC)) and (j < len(Z_left) or j < len(Z_right)):
    # left leg is stance leg: a_thigh_left < a_thigh_right
    if At_right[IC[i]] > At_left[IC[i]]:
        if 0 < (Z_left[j] - IC[i]) < mean_stance:
            dv1.append(abs(Vs_left[Z_left[j]] - Vs_left[IC[i]]))
            dt1.append((Z_left[j] - IC[i]) / fs)
            i += 1
            j += 1
        elif (Z_left[j] - IC[i]) < 0:
            j += 1
        elif (Z_left[j] - IC[i]) > mean_stance:
            i += 1
    # right leg is stance leg
    elif At_left[IC[i]] > At_right[IC[i]]:
        if 0 < Z_right[j] - IC[i] < mean_stance:
            dv1.append(abs(Vs_right[Z_right[j]] - Vs_right[IC[i]]))
            dt1.append((Z_right[j] - IC[i]) / fs)
            i += 1
            j += 1
        elif (Z_right[j] - IC[i]) < 0:
            j += 1
        elif (Z_right[j] - IC[i]) > mean_stance:
            i += 1

dv1 = b*np.array(dv1)
dt1 = c*np.array(dt1)

# The acceleration a1 due to the impact of the lower limb mass m1 is determined
# from the vertical velocity Δv1 of the ankle marker slowing to zero over the time interval Δt1
# Because I do not have an ankle marker, I have taken the shank IMU (most close)

a1 = dv1 / dt1
F1_avg = m1 * (a1 + g)

# Impulse J1 corresponds to the vertical deceleration of m1 during surface impact
J1 = F1_avg * 2*dt1

# J2 corresponds to the acceleration of the remainder of the total mass.

# correctie om errors te vermijden
Jtot = Jtot[:len(J1)]
tc = tc[:len(J1)]

# Impulse J2
J2 = Jtot - J1
F2_avg = J2 / tc

# ground reaction force
A1 = 2*F1_avg
A2 = 2*F2_avg
B1 = dt1
B2 = a*tc
C1 = B1
C2L = B2
C2T = tc - B2

F1 = np.zeros(A.shape[1])
F2 = np.zeros(A.shape[1])
for i in range(len(J1)):
    t = np.arange(0, T0[i]-IC[i], 1) / fs # of t = np.arange(IC[i], T0[i]) / fs
    for j in range(len(t)):
        if t[j] < B1[i] - C1[i]:
            F1[IC[i]+j] = 0
        elif B1[i] - C1[i] <= t[j] <= B1[i] + C1[i]:
            F1[IC[i]+j] = A1[i]/2 * (1 + np.cos((t[j] - B1[i])/C1[i] * np.pi))

```

```

        elif t[j] > B1[i] + C1[i]:
            F1[IC[i]+j] = 0
        if t[j] < B2[i] - C2L[i]:
            F2[IC[i]+j] = 0
        elif B2[i] - C2L[i] <= t[j] <= B2[i]:
            F2[IC[i]+j] = A2[i]/2 * (1 + np.cos((t[j] - B2[i])/C2L[i] * np.pi))
        elif B2[i] < t[j] <= B2[i] + C2T[i]:
            F2[IC[i]+j] = A2[i]/2 * (1 + np.cos((t[j] - B2[i])/C2T[i] * np.pi))
        elif t[j] > B2[i] + C2T[i]:
            F2[IC[i]+j] = 0

    Ftot = F1 + F2

    return Ftot/(M*g)

```

```

a = 0.47
b = 2.2
c = 0.3

```

✓ 10. Combination of link-segment model and two-mass model

```

def CombolSMand2MM(b, c, fs, WF, M, A, FlightPhaseCorr=True):

    # Two-mass model (2MM)
    g = 9.81
    m1 = 0.08 * M
    m2 = 0.92 * M

    # calculation of the first impulse J1 (impact peak)
    IC, T0 = StepDetector3(fs, A)
    IC, T0, IC, T0 = CheckSteps(IC, T0, IC, T0)

    tc = (T0 - IC) / fs # contact time = time in contact with the ground
    tc = tc[:-1]
    ta = (IC[1:] - T0[:-1]) / fs # aerial time = time in flight phase
    # solve the problem that tc contains one element more than ta

    At_left = A[2, :, 2]
    At_right = A[0, :, 2]
    V = VelocityFromIMU(fs, A)
    V_filtered = FilterData(V, fs_Vicon, 3, [1, 6.5], 'bandpass', 1)
    Vs_left = V_filtered[1, :, 2] # vertical velocity of the right shank
    Vs_right = V_filtered[3, :, 2]
    # find the values where the velocity of the shank slows to zero after initial contact
    Z_left = np.where((Vs_left[1:]>0) & (np.sign(Vs_left[1:]*Vs_left[:-1])==-1))[0]
    Z_right = np.where((Vs_right[1:]>0) & (np.sign(Vs_right[1:]*Vs_right[:-1])==-1))[0]

    # check that Z > IC and corresponding Z for IC
    mean_stance = np.mean(T0 - IC) # duration of one stance phase
    i = 0
    dt1 = []
    j = 0
    dv1 = []

    while (i < len(IC)) and (j < len(Z_left) or j < len(Z_right)):
        if At_left[IC[i]] > At_right[IC[i]]: # left leg is stance leg
            if 0 < (Z_left[j] - IC[i]) < 1.5*mean_stance:
                dv1.append(abs(Vs_left[Z_left[j]] - Vs_left[IC[i]]))
                dt1.append((Z_left[j] - IC[i]) / fs)
                i += 1
                j += 1
            elif (Z_left[j] - IC[i]) < 0:

```

```

        j += 1
    elif (Z_left[j] - IC[i]) > 1.5*mean_stance:
        i += 1
    elif At_right[IC[i]] > At_left[IC[i]]: # right leg is stance leg
        if 0 < Z_right[j] - IC[i] < 1.5*mean_stance:
            dv1.append(abs(Vs_right[Z_right[j]] - Vs_right[IC[i]]))
            dt1.append((Z_right[j] - IC[i]) / fs)
            i += 1
            j += 1
        elif (Z_right[j] - IC[i]) < 0:
            j += 1
        elif (Z_right[j] - IC[i]) > 1.5*mean_stance:
            i += 1

dv1 = b*np.array(dv1)
dt1 = c*np.array(dt1)
# The acceleration a1 due to the impact of the lower limb mass m1 is determined
# from the vertical velocity Δv1 of the ankle marker slowing to zero over the time interval Δt1
# Because I do not have an ankle marker, I have taken the shank IMU (most close)

a1 = dv1 / dt1
F1_avg = m1 * (a1 + g)
# Impulse J1 corresponds to the vertical deceleration of m1 during surface impact
J1 = F1_avg * 2*dt1

# F1 derived from impulse J1
A1 = 2*F1_avg
B1 = dt1
C1 = B1

F1 = np.zeros(A.shape[1])
for i in range(len(J1)):
    t = np.arange(0, T0[i]-IC[i], 1) / fs # of t = np.arange(IC[i], T0[i]) / fs
    for j in range(len(t)):
        if t[j] < B1[i] - C1[i]:
            F1[IC[i]+j] = 0
        elif B1[i] - C1[i] <= t[j] <= B1[i] + C1[i]:
            F1[IC[i]+j] = A1[i]/2 * (1 + np.cos((t[j] - B1[i])/C1[i] * np.pi))
        elif t[j] > B1[i] + C1[i]:
            F1[IC[i]+j] = 0

# Part 2: link-segment model
A_copy = copy.deepcopy(A)
if FlightPhaseCorr==True:
    A_copy[:, :, 2][A_copy[:, :, 2] <= -g] = -g

F = np.zeros_like(A)
for i in range(A.shape[0]):
    F[i, :] = WF[i] * M * A_copy[i, :, :]
F2xyz = np.sum(F, axis=0) + M*G
F2res = np.sqrt(F2xyz[:, 0]**2 + F2xyz[:, 1]**2 + F2xyz[:, 2]**2)
F2 = np.concatenate((F2xyz, np.expand_dims(F2res, axis=1)), axis=1)

# Combination of LSM and 2MM
eGRF = np.zeros_like(F2)
eGRF[:, 0] = F2[:, 0]
eGRF[:, 1] = F2[:, 1]
eGRF[:, 2] = F1 + F2[:, 2]
eGRF[:, 3] = np.sqrt(eGRF[:, 0]**2 + eGRF[:, 1]**2 + eGRF[:, 2]**2)

return eGRF/(M*g)

```

```
a = 0.47
b = 2.2
c = 0.3
```

✓ 11. Mass-spring damper model

```
def ParametersMSD(AM, M, WF, X, V, A):

    # Initial conditions for the model's eight parameter values for reproducing GRF, taken from Verheul
    p1_0 = 0.015
    p2_0 = 0.0074

    v1_0 = -0.73
    v2_0 = -0.66

    w1_0 = 18.33 # w1_0 = 207 (Derrick)
    w2_0 = 58.32 * np.ones_like(X.shape[1]) # w2_0 = 626 (Derrick)

    zeta_0 = 0.31*np.ones_like(X.shape[1]) # zeta_0 = 0.35

    # upper mass m1 on a spring on top, representing the rest of the body
    m1 = (WF[0] + WF[3] + WF[4])*M # right leg + upper body

    # lower mass m2 on a spring and damper, representing the supporting leg
    m2 = (WF[1]+WF[2])*M # left shank OR (WF[1]+WF[2])

    # λ is the upper mass ratio relative to the lower mass
    lamda = m1/m2

    # position of the lower mass without any external load is described by x2
    # l2 is the natural length of the lower spring
    l2 = (AM[1, 2] + AM[0, 2])/2
    x2 = (X[1, :, 2] + X[2, :, 2])/2 # average of thigh and shank
    p2 = x2 - l2

    # position of the upper mass without any external load is described by x1
    # l1 is the natural length of the upper spring
    l1 = AM[0,2] + AM[1, 2] + AM[2, 2] # total body length? OR only upper body?
    x1 = X[4, :, 2]
    p1 = x1 - l1 - l2

    # ζ is the damper's damping ratio
    zeta = zeta_0 # zeta = c / (2 * np.sqrt(k2 * m2))

    # initial velocity of the upper mass
    v1 = V[4, :, 2]

    # initial velocity of the lower mass
    v2 = V[1, :, 2] # (V[1, :, 2] + V[2, :, 2])/2

    # acceleration of the upper mass
    a1 = A[4, :, 2]

    # acceleration of the lower mass
    a2 = A[1, :, 2] # (A[1, :, 2] + A[2, :, 2])/2

    def model_a1(w1, p1, p2): # acceleration of the upper mass
        g = -9.81
        return - w1**2 * (p1-p2) + g

    def diff_a1(w1, p1, p2, a1):
        return model_a1(w1, p1, p2) - a1
```

```

def grad_a1(w1, p1, p2, a1): # Opm: ik voegde a1 toe, want anders werkt de functie niet...
    # jacobian matrix
    J = np.zeros((p1.size, 1))
    J[:, 0] = -2*(p1-p2)*w1 # Derivative to variable w1 of model a1
    return J

def model_a2(X, w1, lamda, p1, p2, v2): # acceleration lower mass
    g = -9.81
    w2, zeta = X
    return -w2**2 * p2 + w1**2 * lamda * (p1-p2) - 2 * zeta * w2**2 * v2 + g

def diff_a2(X, w1, lamda, p1, p2, v2, a2):
    return model_a2(X, w1, lamda, p1, p2, v2) - a2

def grad_a2(X, w1, lamda, p1, p2, v2, a2):
    # jacobian matrix
    w2, zeta = X
    J = np.zeros((p1.size, 2))
    J[:, 0] = -2*w2*p2 - 2*zeta*v2 # derivative to w2
    J[:, 1] = -2*w2*v2 # derivative to zeta
    return J

res_a1 = scipy.optimize.least_squares(diff_a1, w1_0, jac=grad_a1, loss='soft_l1', f_scale=0.1, bounds=(
w1_opt = res_a1.x[0]
print(w1_opt)

g = -9.81
w1 = np.zeros_like(a1)
for i in range(len(a1)):
    root = (a1[i]-g) / (p1[i]-p2[i])
    if root > 0:
        w1[i] = 0
    elif root <= 0:
        w1[i] = np.sqrt(-root)
print(np.mean(w1)) # Feasible range: [0, 50] -> OK

w2 = np.zeros_like(a2)
for i in range(len(a2)):
    root = (a2[i]-(w1[i]**2)*lamda*(p1[i]-p2[i])-g) / (p2[i])
    if root > 0:
        w2[i] = 0
    elif root <= 0:
        w2[i] = np.sqrt(-root)
print(np.mean(w2))

x_init = [w2_0, zeta_0]
res_a2 = scipy.optimize.least_squares(diff_a2, x_init, jac=grad_a2, loss='soft_l1', f_scale=0.1, bounds
w2_opt, zeta_opt = res_a2.x # bounds=[(lower1, lower2), (upper1, upper2)]
print(w2_opt) # Feasible range: [0, 174] -> ?
print(zeta_opt) # Feasible range: [0.1, 1.5] -> ?

# linear spring stiffness constants for the upper spring is defined as k1
k1 = w1**2 * m1

# linear spring stiffness constants for the lower spring is defined as k2
k2 = w2**2 * m2

# c is the damper's damping coefficient.
c = zeta * 2 * np.sqrt(k2*m2)

return m1, m2, l1, l2, w1, w1_opt, w2, w2_opt, zeta_opt, lamda, c, p1, p2, v1, v2, a1, a2

```

According to "Biomechanics of Movement" (Uchida, Delp) the dimensionless leg stiffness is estimated as the ratio of the peak vertical ground reaction force normalised by the body weight (F_{peak}) to the change in leg length during that stance phase, normalised by leg length at foot contact (l_0), where l_{min} is the minimum leg length during stance? Leg length is defined as the distance between the center of pressure and the geometric center of the pelvis.

$$k_{leg} = \frac{F_{peak}}{(l_0 - l_{min})/l_0}$$

```
def EstimateW2(fs, A, v2, a2, w1, lamda, p1, p2, b, c):
    # calculation of the first impulse J1 (impact peak)
    IC, TO = StepDetector3(fs, A)

    tc = (TO - IC) / fs # contact time = time in contact with the ground
    tc = tc[:-1]
    ta = (IC[1:] - TO[:-1]) / fs # aerial time = time in flight phase
    # solve the problem that tc contains one element more than ta

    At_left = A[2, :, 2]
    At_right = A[0, :, 2]
    V = VelocityFromIMU(fs, A)
    V_filtered = FilterData(V, fs_Vicon, 3, [1, 6.5], 'bandpass', 1)
    Vs_left = V_filtered[1, :, 2] # vertical velocity of the right shank
    Vs_right = V_filtered[3, :, 2]
    # find the values where the velocity of the shank slows to zero after initial contact
    Z_left = np.where((Vs_left[1:]>0) & (np.sign(Vs_left[1:]*Vs_left[:-1])== -1))[0]
    Z_right = np.where((Vs_right[1:]>0) & (np.sign(Vs_right[1:]*Vs_right[:-1])== -1))[0]

    # check that Z > IC and corresponding Z for IC
    mean_stance = np.mean(TO - IC) # duration of one stance phase
    i = 0
    dt1 = []
    j = 0
    dv1 = []

    while (i < len(IC)) and (j < len(Z_left) or j < len(Z_right)):
        if At_left[IC[i]] > At_right[IC[i]]: # left leg is stance leg
            if 0 < (Z_left[j] - IC[i]) < 1.5*mean_stance:
                dv1.append(abs(Vs_left[Z_left[j]] - Vs_left[IC[i]]))
                dt1.append((Z_left[j] - IC[i]) / fs)
                i += 1
                j += 1
            elif (Z_left[j] - IC[i]) < 0:
                j += 1
            elif (Z_left[j] - IC[i]) > 1.5*mean_stance:
                i += 1
        elif At_right[IC[i]] > At_left[IC[i]]: # right leg is stance leg
            if 0 < Z_right[j] - IC[i] < 1.5*mean_stance:
                dv1.append(abs(Vs_right[Z_right[j]] - Vs_right[IC[i]]))
                dt1.append((Z_right[j] - IC[i]) / fs)
                i += 1
                j += 1
            elif (Z_right[j] - IC[i]) < 0:
                j += 1
            elif (Z_right[j] - IC[i]) > 1.5*mean_stance:
                i += 1

    dv1 = b*np.array(dv1)
    dt1 = c*np.array(dt1)
    # The acceleration a1 due to the impact of the lower limb mass m1 is determined
    # from the vertical velocity Δv1 of the ankle marker slowing to zero over the time interval Δt1
    # Because I do not have an ankle marker, I have taken the shank IMU (most close)

    a1 = dv1 / dt1
```

```

F1_avg = m1 * (a1 + g)
# Impulse J1 corresponds to the vertical deceleration of m1 during surface impact
J1 = F1_avg * 2*dt1

# F1 derived from impulse J1
A1 = 2*F1_avg
B1 = dt1
C1 = B1

F1 = np.zeros(A.shape[1])
for i in range(len(IC)-1):
    t = np.arange(0, TO[i]-IC[i], 1) / fs # of t = np.arange(IC[i], TO[i]) / fs
    for j in range(len(t)):
        if t[j] < B1[i] - C1[i]:
            F1[IC[i]+j] = 0
        elif B1[i] - C1[i] <= t[j] <= B1[i] + C1[i]:
            F1[IC[i]+j] = A1[i]/2 * (1 + np.cos((t[j] - B1[i])/C1[i] * np.pi))
        elif t[j] > B1[i] + C1[i]:
            F1[IC[i]+j] = 0

zeta = F1/2*v2

w2 = np.zeros_like(a2)
for i in range(len(a2)):
    root = (a2[i]-(w1[i]**2)*lamda*(p1[i]-p2[i])-g) / (p2[i]) #TO DO: voeg zeta toe!
    if root > 0:
        w2[i] =0
    elif root <= 0:
        w2[i] = np.sqrt(-root)

return w2, zeta

```

```

a = 0.47
b = 2.2
c = 0.3

```

```

def PredCorr(lamda, w1, w2_0, zeta_0, p2, v2, a2, t, eps=0.1):
    g = -9.81

    N = len(t)
    h = t[1] - t[0]

    # oplossingsvector X (N+1 rijen, 2 kolommen)
    X = np.zeros((N+1,2)) #X = [x1,x2] met kolom 1 is het verloop van w2 en kolom 2 is het verloop van z

    # startpunt
    X0 = np.asarray([w2_0, zeta_0]) # Merk op dat de tijd over de rijen loopt!
    X[0,:] = X0 # Dus elke nieuwe rij is een nieuwe stap in de tijd

    # Lotka-Volterra systeem
    f = lambda t,p2,v2,X : np.asarray([-2*X[0]*p2 - 2*X[1]*v2, -2*X[0]*v2]) # Voor die tijdstap
    # eerste en tweede differentiaalvergelijking ingeven
    errorL = []
    # Predictie - Correctie
    for j in range(N-1):
        # predictie met expliciete euler (een predictie van elke rij, op basis van de gecorrigeerde
        X[j+1,:] = X[j,:] + h*f(t[j], p2[j], v2[j], X[j,:]) #Merk op dat we de rijen van X aanpassen ipv
        #if a2[j] <= 0:
        #    X[j+1,0], X[j+1, 0] = 0,0
        # correcties met Adams-Moulton (2de orde) (we verbeteren nu elke predictie)
        a2_approx = -((X[j+1,0])**2) * p2[j] + w1[j]**2 * lamda * (p1[j]-p2[j]) - 2*X[j+1,1]*X[j+1,0]*v2[j]
        error = (a2_approx - a2[j])**2
        for i in range(10):

```

```

X[j+1,:] = X[j,:] + (h/2)*((f(t[j],p2[j],v2[j],X[j,:])) + f(t[j+1],p2[j], v2[j],X[j+1,:])) #Ha
a2_approx = -((X[j+1,0])**2) * p2[j] + w1[j]**2 * lamda * (p1[j]-p2[j]) - 2*X[j+1,1]*X[j+1,0]*v
error = np.abs(a2_approx - a2[j])
errorL.append(error)
# Eens we een voorspelling hebben, gaan we deze voorspelling verbeteren.
# Merk op: X[j+1,:] staat zowel links als rechts? Dit is logisch! Rechts is de voorspelde waarde. L
# Merk op: er staat geen i in de for-lus, maar toch itereren we twee keer. Zo kunnen we 2 correctie
# Alternatief: we corrigeren tot we onder een bepaalde drempelwaarde zitten
return errorL, X[1:, :]

```

```

w2_0 = 19
zeta_0 = 0.31

```

✓ 12. Model evaluation

12.1 Select time frame for metrics

✓ 12.2 Metrics

```

def Calculate_rRMSE(eGRF, mGRF, correction=False):
    RMSE = np.sqrt(mean_squared_error(eGRF, mGRF))
    rRMSE = round(RMSE / (max(mGRF) - min(mGRF)) * 100, 2)

    if correction == True:
        rRMSE = round(RMSE / (np.max(mGRF, initial=1) - np.min(mGRF, initial=0)) * 100, 2)
    return rRMSE

```

Metrics for all directions

```

def CalculateMetrics(eGRF, mGRF, printer=False):

    """
    Compare the GRF estimation (eGRF) of the model with the GRF measurement (mGRF) for each direction.
    - eGRF: array with eGRFx, eGRFy and eGRFz in the columns
    - mGRF: array with mGRFx, mGRFy and mGRFz in the columns
    """

    if len(eGRF.shape) == 2:
        eGRFx, eGRFy, eGRFz = eGRF[:, 0], eGRF[:, 1], eGRF[:, 2]
        eGRFres = np.sqrt(eGRFx**2 + eGRFy**2 + eGRFz**2)
        mGRFx, mGRFy, mGRFz = mGRF[:, 0], mGRF[:, 1], mGRF[:, 2]
        mGRFres = np.sqrt(mGRFx**2 + mGRFy**2 + mGRFz**2)

        # Pearson correlation coefficient
        # r[1] provides the p-value indicates the probability of an uncorrelated system producing datasets
        # correlation at least as extreme as the one computed from these datasets.
        r_x = round(pearsonr(eGRFx, mGRFx)[0], 2)
        r_y = round(pearsonr(eGRFy, mGRFy)[0], 2)
        r_z = round(pearsonr(eGRFz, mGRFz)[0], 2)
        r_res = round(pearsonr(eGRFres, mGRFres)[0], 2)
        r = np.array([r_x, r_y, r_z, r_res])

        RMSEx = round(np.sqrt(mean_squared_error(eGRFx, mGRFx)), 2)
        RMSEy = round(np.sqrt(mean_squared_error(eGRFy, mGRFy)), 2)
        RMSEz = round(np.sqrt(mean_squared_error(eGRFz, mGRFz)), 2)
        RMSEres = round(np.sqrt(mean_squared_error(eGRFres, mGRFres)), 2)
        RMSE = np.array([RMSEx, RMSEy, RMSEz, RMSEres])

```

```

rRMSEx = Calculate_rRMSE(eGRFx, mGRFx) # zie opmerking over het verwijderen van de flight phase..
rRMSEy = Calculate_rRMSE(eGRFy, mGRFy)
rRMSEz = Calculate_rRMSE(eGRFz, mGRFz, correction=True)
rRMSEres = Calculate_rRMSE(eGRFres, mGRFres)
rRMSE = np.array([rRMSEx, rRMSEy, rRMSEz, rRMSEres])

if printer==True:
    print("1. PEARSON CORRELATION COEFFICIENT")
    print('Pearson correlation coefficient of the GRF in the M-L direction:', r_x)
    print('Pearson correlation coefficient of the GRF in the A-P direction:', r_y)
    print('Pearson correlation coefficient of the GRF in the vertical direction:', r_z)
    print('Pearson correlation coefficient of the resultant GRF:', r_res)
    print('-'*100)

    print('2. ROOT MEAN SQUARE ERROR')
    print('RMSE of the GRF in the M-L direction:', RMSEx, 'N/kg')
    print('RMSE of the GRF in the A-P direction:', RMSEy, 'N/kg')
    print('RMSE of the GRF in the vertical direction:', RMSEz, 'N/kg')
    print('RMSE of the resultant GRF:', RMSEres, 'N/kg')
    print('-'*100)

    print('3. RELATIVE ROOT MEAN SQUARE ERROR') # Note: Better to call relative root mean square e
    print('rRMSE of the GRF in the M-L direction:', rRMSEx, '% BW')
    print('rRMSE of the GRF in the A-P direction:', rRMSEy, '% BW')
    print('rRMSE of the GRF in the vertical direction:', rRMSEz, '% BW')
    print('rRMSE of the resultant GRF:', rRMSEres, '% BW')

return r, RMSE, rRMSE

elif len(eGRF.shape) == 1:
    r = pearsonr(eGRF, mGRF)[0]
    RMSE = round(np.sqrt(mean_squared_error(eGRF, mGRF)), 2)
    rRMSE = Calculate_rRMSE(eGRF, mGRF, correction=True)

    if printer==True:
        print('Pearson correlation coefficient is:', r)
        print('RMSE is:', RMSE, 'N/kg')
        print('rRMSE is:', rRMSE, '% BW')

return r, RMSE, rRMSE

```

```

def Calculate_PeakError(eGRF, mGRF, A, fs, height=1, min_time_diff=None, printer=False):

    i_peaks_eGRF, i_peaks_eGRF_left, i_peaks_eGRF_right = DetectActivePeak(eGRF, A, fs, height, min_time_diff)
    i_peaks_mGRF, i_peaks_mGRF_left, i_peaks_mGRF_right = DetectActivePeak(mGRF, A, fs, height, min_time_diff)

    peaks_eGRF = []
    peaks_mGRF = []

    # check first if eGRF and mGRF have the same amount of peaks
    if len(eGRF.shape) == 1:
        step_length = np.mean(i_peaks_mGRF[1:] - i_peaks_mGRF[:-1])
        i = 0
        j = 0
        # additional check, in order to compare the same peaks
        while i < len(i_peaks_eGRF) and j < len(i_peaks_mGRF):
            if -step_length <= i_peaks_mGRF[j] - i_peaks_eGRF[i] <= step_length: # not too close or too
                peaks_eGRF.append(eGRF[i_peaks_eGRF[i]])
                i += 1
                peaks_mGRF.append(mGRF[i_peaks_mGRF[j]])
                j += 1
            elif (i_peaks_mGRF[j] - i_peaks_eGRF[i]) > step_length:
                i += 1
            elif (i_peaks_mGRF[j] - i_peaks_eGRF[i]) < -step_length:

```

```

        j += 1

elif len(eGRF.shape) == 2:
    step_length = np.mean(i_peaks_mGRF[1:] - i_peaks_mGRF[:-1])
    i = 0
    j = 0
    # additional check, in order to compare the same peaks
    while i < len(i_peaks_eGRF) and j < len(i_peaks_mGRF):
        if -step_length <= i_peaks_mGRF[j] - i_peaks_eGRF[i] <= step_length: # not too close or too far
            peaks_eGRF.append(eGRF[i_peaks_eGRF[i], 2])
            i += 1
            peaks_mGRF.append(mGRF[i_peaks_mGRF[j], 2])
            j += 1
        elif (i_peaks_mGRF[j] - i_peaks_eGRF[i]) > step_length:
            i += 1
        elif (i_peaks_mGRF[j] - i_peaks_eGRF[i]) < -step_length:
            j += 1

elif len(eGRF.shape) == 3: # steps
    for i in range(len(i_peaks_eGRF)):
        peaks_eGRF.append(eGRF[i, i_peaks_eGRF[i], 2])
        peaks_mGRF.append(mGRF[i, i_peaks_mGRF[i], 2])

peaks_eGRF = np.array(peaks_eGRF)
peaks_mGRF = np.array(peaks_mGRF)

# absolute peak error: https://en.wikipedia.org/wiki/Mean_absolute_error
abs_peak_error = np.abs(peaks_eGRF - peaks_mGRF)
mean_abs_peak_error = round(np.mean(abs_peak_error), 2)
stdv_abs_peak_error = round(np.std(abs_peak_error), 2)

# relative peak error
rel_peak_error = np.abs((peaks_eGRF - peaks_mGRF) / peaks_mGRF)*100
mean_rel_peak_error = round(np.mean(rel_peak_error), 2)
stdv_rel_peak_error = round(np.std(rel_peak_error), 2)

if printer==True:
    print('The peak error is:', mean_abs_peak_error, '+/-', stdv_abs_peak_error, 'N/kg')
    print('The peak error is:', mean_rel_peak_error, '+/-', stdv_rel_peak_error, '%')

return mean_abs_peak_error, stdv_abs_peak_error, mean_rel_peak_error, stdv_rel_peak_error

```

```

def Calculate>LoadingRateError(steps_eGRF, steps_mGRF, printer=False):

```

```

    VLR_eGRF = []
    VLR_mGRF = []

    # check first if eGRF and mGRF have the same amount of steps
    if len(steps_eGRF.shape) == 3:
        for i in range(steps_eGRF.shape[0]):
            max_i_eGRF = int(30/100 * len(steps_eGRF[i, :, 2]))
            d_eGRF = np.gradient(steps_eGRF[i, :, 2])
            VLR_eGRF.append(np.mean(d_eGRF[0:max_i_eGRF]))

            max_i_mGRF = int(30/100 * len(steps_mGRF[i, :, 2]))
            d_mGRF = np.gradient(steps_mGRF[i, :, 2])
            VLR_mGRF.append(np.mean(d_mGRF[0:max_i_mGRF]))

    if len(steps_eGRF.shape) == 2:
        for i in range(steps_eGRF.shape[0]):
            max_i_eGRF = int(30/100 * len(steps_eGRF[i, :]))
            d_eGRF = np.gradient(steps_eGRF[i, :])
            VLR_eGRF.append(np.mean(d_eGRF[0:max_i_eGRF]))

            max_i_mGRF = int(30/100 * len(steps_mGRF[i, :]))

```

```

        d_mGRF = np.gradient(steps_mGRF[i, :])
        VLR_mGRF.append(np.mean(d_mGRF[0:max_i_mGRF]))

VLR_eGRF = np.array(VLR_eGRF)
VLR_mGRF = np.array(VLR_mGRF)

# absolute error
abs_VLR_error = np.abs(VLR_eGRF - VLR_mGRF)
# relative error
rel_VLR_error = np.abs((VLR_eGRF - VLR_mGRF) / VLR_mGRF)*100

mean_abs_VLR_error = round(np.mean(abs_VLR_error), 2)
stdv_abs_VLR_error = round(np.std(abs_VLR_error), 2)
mean_rel_VLR_error = round(np.mean(rel_VLR_error), 2)
stdv_rel_VLR_error = round(np.std(rel_VLR_error), 2)

if printer==True:
    print('The loading rate error is:', mean_abs_VLR_error, '+/-', stdv_abs_VLR_error, 'N/kg')
    print('The loading rate error is:', mean_rel_VLR_error, '+/-', stdv_rel_VLR_error, '%')

return mean_abs_VLR_error, stdv_abs_VLR_error, mean_rel_VLR_error, stdv_rel_VLR_error

```

```

def Calculate_ImpulseError(steps_eGRF, steps_mGRF, printer=False):

    if len(steps_eGRF.shape) == 3:
        # calculate impulse
        impulse_eGRF = scipy.integrate.trapezoid(steps_eGRF[:, :, 2], axis=1)
        impulse_mGRF = scipy.integrate.trapezoid(steps_mGRF[:, :, 2], axis=1)

    if len(steps_eGRF.shape) == 2:
        # calculate impulse
        impulse_eGRF = scipy.integrate.trapezoid(steps_eGRF[:, :], axis=1)
        impulse_mGRF = scipy.integrate.trapezoid(steps_mGRF[:, :], axis=1)

    # absolute error
    abs_impulse_error = np.abs(impulse_eGRF - impulse_mGRF)
    # relative error
    rel_impulse_error = np.abs((impulse_eGRF - impulse_mGRF) / impulse_mGRF)*100

    mean_abs_impulse_error = round(np.mean(abs_impulse_error), 4)
    stdv_abs_impulse_error = round(np.std(abs_impulse_error), 4)
    mean_rel_impulse_error = round(np.mean(rel_impulse_error), 4)
    stdv_rel_impulse_error = round(np.std(rel_impulse_error), 4)

    if printer==True:
        print('The loading rate error is:', mean_abs_impulse_error, '+/-', stdv_abs_impulse_error, 'N/kg')
        print('The loading rate error is:', mean_rel_impulse_error, '+/-', stdv_rel_impulse_error, '%')

    return mean_abs_impulse_error, stdv_abs_impulse_error, mean_rel_impulse_error, stdv_rel_impulse_error

```

Metrics per step

```

def CalculateMetricsSteps(M, step_t, fs, threshold, A, eGRF, mGRF, height=1, min_time_diff=None, printer=False)
    """
    Compare the GRF estimation (eGRF) of the model with the GRF measurement (mGRF) per step.
    - eGRF: array with eGRFx, eGRFy and eGRFz in the columns
    - mGRF: array with mGRFx, mGRFy and mGRFz in the columns
    """

    IC_mGRF, TO_mGRF = StepDetector1(mGRF, M, step_t, fs, threshold)
    IC_eGRF, TO_eGRF = StepDetector3(fs, A)

    IC_eGRF_check, TO_eGRF_check, IC_mGRF_check, TO_mGRF_check = CheckSteps(IC_eGRF, TO_eGRF, IC_mGRF, TO_m

```

```

t, steps_mGRF, steps_mGRF_left, steps_mGRF_right = GetSteps(IC_mGRF_check, TO_mGRF_check, A, mGRF, fs)
t, steps_eGRF, steps_eGRF_left, steps_eGRF_right = GetSteps(IC_eGRF_check, TO_eGRF_check, A, eGRF, fs)

step_length = max(steps_eGRF.shape[1], steps_mGRF.shape[1])
steps_eGRF, steps_eGRF_left, steps_eGRF_right = scipy.signal.resample(steps_eGRF, step_length, axis=1),
steps_mGRF, steps_mGRF_left, steps_mGRF_right = scipy.signal.resample(steps_mGRF, step_length, axis=1),

r = []
RMSE = []
rRMSE = []
for i in range(steps_eGRF.shape[0]):
    if len(steps_eGRF.shape) == 3:
        r_step, RMSE_step, rRMSE_step = CalculateMetrics(steps_eGRF[i, :, :], steps_mGRF[i, :, :])
    if len(steps_eGRF.shape) == 2:
        r_step, RMSE_step, rRMSE_step = CalculateMetrics(steps_eGRF[i, :], steps_mGRF[i, :])
    r.append(r_step)
    RMSE.append(RMSE_step)
    rRMSE.append(rRMSE_step)
r = np.array(r)
RMSE = np.array(RMSE)
rRMSE = np.array(rRMSE)
mean_r, stdv_r = np.mean(r, axis=0), np.std(r, axis=0)
mean_RMSE, stdv_RMSE = np.mean(RMSE, axis=0), np.std(RMSE, axis=0)
mean_rRMSE, stdv_rRMSE = np.mean(rRMSE, axis=0), np.std(rRMSE, axis=0)

mean_abs_peak_error, stdv_abs_peak_error, mean_rel_peak_error, stdv_rel_peak_error = Calculate_PeakError
mean_abs_VLR_error, stdv_abs_VLR_error, mean_rel_VLR_error, stdv_rel_VLR_error = Calculate_LoadingRateE
mean_abs_impulse_error, stdv_abs_impulse_error, mean_rel_impulse_error, stdv_rel_impulse_error = Calcul

res = [[mean_r, stdv_r], [mean_RMSE, stdv_RMSE], [mean_rRMSE, stdv_rRMSE], [mean_abs_peak_error, stdv_a
header = ['Pearson correlation coefficient', 'Root mean square error', 'Relative root mean square error

if printer == True:
    for i in range(len(header)):
        print(header[i])
        print(res[i])

return res

```

✓ 12.3 Contribution of each segment

✓ 12.3.1 Correlation matrix

```

def PlotCorrelation(M, WF, A, mGRF_left, mGRF_right, mGRF):

    # creating the Numpy array of vertical accelerations: acc_z in each column
    Av = np.zeros((A.shape[1], A.shape[0]))
    for i in range(Av.shape[1]):
        Av[:, i] = A[i, :, 2]

    # creating a list of index names
    index_values = np.arange(0, Av.shape[0])

    # creating a list of column names
    column_values = ['acc_z1', 'acc_z2', 'acc_z3', 'acc_z4', 'acc_z5']

    # creating the dataframe
    df = pd.DataFrame(data = np.array(Av), columns = column_values)
    eGRF = LinkSegmentModel(M, WF, A)
    df['mGRFv_left'] = mGRF_left[:, 2] # add additional column with the mGRF of the left side of the treadmill
    df['mGRFv_right'] = mGRF_right[:, 2] # add additional column with the mGRF of the right side of the treadmill

```

```

df['mGRFv'] = mGRF[:, 2] # add additional column with the mGRF of the treadmill to the dataframe
df['eGRFv'] = eGRF[:, 2]
corr_matrix = df.corr(method='pearson').round(2)

# plotting correlation matrix
plt.imshow(corr_matrix, cmap='jet')
plt.figure(figsize=(11,8))
plt.imshow(corr_matrix, cmap='coolwarm')

# adding colorbar
plt.colorbar()

# extracting variable names
variables = ['right thigh', 'left shank', 'left thigh', 'right shank', 'pelvis', 'left mGRFv', 'right mG

# Adding labels to the matrix
plt.xticks(range(len(corr_matrix)), variables, rotation=45, ha='right', rotation_mode="anchor")
plt.yticks(range(len(corr_matrix)), variables)

for i in range(len(corr_matrix)):
    for j in range(len(corr_matrix)):
        text = plt.text(j, i, corr_matrix.values[i, j], ha="center", va="center", color="w")

# Display the plot
plt.show()

return corr_matrix

```

✓ 12.3.2 Principal component analysis

PCA basically searches a linear combination of variables so that we can extract maximum variance from the variables. Once this process completes it removes it and searches for another linear combination that gives an explanation about the maximum proportion of remaining variance which basically leads to orthogonal factors. In this method, we analyze total variance.

Source: <https://www.geeksforgeeks.org/principal-component-analysis-with-python/> and <https://www.datacamp.com/tutorial/principal-component-analysis-in-python>

```

from sklearn.decomposition import PCA

```

```

def PrincipalComponentAnalysis(M, WF, A, K):

    """
    Perform PCA on acceleration data.

    WF : array-like, shape (n_segments,)
    A : ndarray, shape (n_segments, n_timepoints, 3)
    K : how many principal components you want to take into account
    M : body mass

    Returns:
    pA_k, pA_1k, pGRF_k, pGRF_1k, explained_variance
    """

    g = 9.81
    G_vec = np.array([0, 0, g])

    n_segments, n_time, n_axes = A.shape
    A_reshaped = np.zeros((n_time, n_segments * n_axes))
    WF_reshaped = np.zeros(n_segments * n_axes)
    for i in range(n_segments):

```

```

A_segment = A[i, :, :]
A_reshaped[:, i*3:(i+1)*3] = A_segment

μ = np.mean(A_reshaped, axis=0)
A_centered = A_reshaped - μ
meanEnorm = np.mean(np.linalg.norm(A_centered, ord=2, axis=1))
A_norm = np.zeros_like(A_reshaped)
for i in range(n_segments):
    A_norm[:, i*3:(i+1)*3] = WF[i] * A_centered[:, i*3:(i+1)*3] / meanEnorm
    WF_reshaped[i*3:(i+1)*3] = WF[i]

pca = PCA(random_state=0, n_components=n_segments*n_axes)
pca.fit(A_norm)
explained_variance = pca.explained_variance_ratio_ * 100
print('Explained variance:', explained_variance)
cum_sum = np.cumsum(np.concatenate(([0], explained_variance)))

n_components = pca.n_components_
component_number = np.arange(1, n_components+1)
component_number2 = np.arange(0, n_components+1)

fig, ax = plt.subplots(1, 2, figsize=(10, 5))
ax[0].plot(component_number, explained_variance, linewidth=2, color='b')
ax[0].spines['top'].set_visible(False)
ax[0].spines['right'].set_visible(False)
ax[0].spines['bottom'].set_linewidth(2)
ax[0].spines['left'].set_linewidth(2)
ax[0].set_xticks(np.arange(1, min(n_components, 20)+1))
ax[0].set_yticks([0, 20, 40, 60, 80, 100])
ax[0].set_ylabel('Explained variance per PC (%)')
ax[0].set_xlabel('n_component')
ax[0].set_xlim(0, n_components)
ax[0].set_ylim(0, 100)

ax[1].plot(component_number2, cum_sum, linewidth=2, color='b')
ax[1].spines['top'].set_visible(False)
ax[1].spines['right'].set_visible(False)
ax[1].spines['bottom'].set_linewidth(2)
ax[1].spines['left'].set_linewidth(2)
ax[1].set_xticks(np.arange(1, min(n_components, 20)+1))
ax[1].set_yticks([0, 20, 40, 60, 80, 100])
ax[1].set_ylabel('Cumulative explained variance (%)')
ax[1].set_xlabel('n_component')
ax[1].set_xlim(0, n_components)
ax[1].set_ylim(0, 100)
plt.tight_layout()
plt.show()

evecs = pca.components_
evals = pca.explained_variance_
C = np.dot(A_norm, evecs.T)

pA_k = np.zeros((n_components, n_time, n_components))
pA_1k = np.zeros((n_components, n_time))

C_partial = np.zeros_like(C)
C_partial[:, :K] = C[:, :K] # De rest van C zijn nullen
A_recon_partial_norm = np.dot(C_partial, evecs)
pA_1k = 1/WF_reshaped * A_recon_partial_norm * meanEnorm + μ

for k in range(n_components):
    C_single = np.zeros_like(C)
    C_single[:, k] = C[:, k] # De rest van C zijn nullen
    A_recon_single_norm = np.dot(C_single, evecs) # vector heeft enkel op positie k getallen
    pA_k[k, :, :] = 1/WF_reshaped * A_recon_single_norm * meanEnorm + μ

```

```

pA_1k_reshaped = np.zeros((n_segments, n_time, 3))
pA_k_reshaped = np.zeros((n_segments, n_time, n_components, 3))
for i in range(n_segments):
    pA_1k_reshaped[i, :, :] = pA_1k[:, i*3:(i+1)*3]
    for k in range(n_components):
        pA_k_reshaped[i, :, k, :] = pA_k[k, :, i*3:(i+1)*3]

pF_1k = np.zeros((n_segments, n_time, 3))
pF_k = np.zeros((n_segments, n_time, n_components, 3))
for i in range(n_segments):
    pF_1k[i, :, :] = M * WF[i] * pA_1k_reshaped[i, :, :]
    for k in range(evecs.shape[1]):
        pF_k[i, :, k, :] = M * WF[i] * pA_k_reshaped[i, :, k, :]
pGRF_xyz_1k = (np.sum(pF_1k, axis=0) + M*g_vec) / (M*g)
pGRF_res_1k = np.sqrt(np.sum(pGRF_xyz_1k**2, axis=1))
pGRF_1k = np.concatenate((pGRF_xyz_1k, pGRF_res_1k[:, np.newaxis]), axis=1)

pGRF_k = np.zeros((evecs.shape[1], n_time, 4))
for k in range(evecs.shape[1]):
    pGRFxyz_k = (np.sum(pF_k[:, :, k, :], axis=0) + M*g_vec) / (M*g) # sum over the segments
    pGRFres_k = np.sqrt(pGRFxyz_k[:, 0]**2 + pGRFxyz_k[:, 1]**2 + pGRFxyz_k[:, 2]**2)
    pGRF_k[k, :, :] = np.concatenate((pGRFxyz_k, np.expand_dims(pGRFres_k, axis=1)), axis=1)

return pA_k, pA_1k, pGRF_k, pGRF_1k, explained_variance

```

✓ 12.3.3 Adding segments

```

def VaryingSegments(M, A, mGRF, method):
    # Model with varying segments
    A_t_right = A[0, :, :]
    A_s_left = A[1, :, :]
    A_t_left = A[2, :, :]
    A_s_right = A[3, :, :]
    A_p = A[4, :, :]

    # only pelvis
    print('pelvis')
    WF_p = 1
    eGRF_p = LinkSegmentModel(M, WF_p, A_p)
    r_p, RMSE_p, rRMSE_p = CalculateMetrics(eGRF_p, mGRF, printer=True)
    print('_'*100)

    # pelvis + thigh
    print('pelvis + thigh')
    WF_W_pt = [0.161, 0.161, 0.678]
    WF_MdL_pt = [0.1986, 0.1986, 0.6028]
    WF_FdL_pt = [0.2088, 0.2088, 0.5824]
    A_pt = np.array([A_t_left, A_t_right, A_p])
    if method == 'Winter':
        eGRF_W_pt = LinkSegmentModel(M, WF_W_pt, A_pt)
        r_W_pt, RMSE_W_pt, rRMSE_W_pt = CalculateMetrics(eGRF_W_pt, mGRF, printer=True)
    if method == 'de Leva men':
        eGRF_MdL_pt = LinkSegmentModel(M, WF_MdL_pt, A_pt)
        r_MdL_pt, RMSE_MdL_pt, rRMSE_MdL_pt = CalculateMetrics(eGRF_MdL_pt, mGRF, printer=True)
    if method == 'de Leva women':
        eGRF_FdL_pt = LinkSegmentModel(M, WF_FdL_pt, A_pt)
        r_FdL_pt, RMSE_FdL_pt, rRMSE_FdL_pt = CalculateMetrics(eGRF_FdL_pt, mGRF, printer=True)
    print('_'*100)

    # pelvis + tibia (shank)
    print('pelvis + shank')
    WF_W_ps = [0.161, 0.161, 0.678]

```

```

WF_MdL_ps = [0.1986, 0.1986, 0.6028]
WF_FdL_ps = [0.2088, 0.2088, 0.5824]
A_ps = np.array([A_s_left, A_s_right, A_p])

if method == 'Winter':
    eGRF_W_ps = LinkSegmentModel(M, WF_W_ps, A_ps)
    r_W_ps, RMSE_W_ps, rRMSE_W_ps = CalculateMetrics(eGRF_W_ps, mGRF, printer=True)
if method == 'de Leva men':
    eGRF_MdL_ps = LinkSegmentModel(M, WF_MdL_ps, A_ps)
    r_MdL_ps, RMSE_MdL_ps, rRMSE_MdL_ps = CalculateMetrics(eGRF_MdL_ps, mGRF, printer=True)
if method == 'de Leva women':
    eGRF_FdL_ps = LinkSegmentModel(M, WF_FdL_ps, A_ps)
    r_FdL_ps, RMSE_FdL_ps, rRMSE_FdL_ps = CalculateMetrics(eGRF_FdL_ps, mGRF, printer=True)
print('_'*100)

# pelvis + thigh + tibia
print('pelvis + thigh + shank')
WF_W_pts = [0.1, 0.061, 0.1, 0.061, 0.678]
WF_MdL_pts = [0.1416, 0.057, 0.1416, 0.061, 0.6028]
WF_FdL_pts = [0.1478, 0.061, 0.1478, 0.061, 0.5824]
A_pts = np.array([A_t_left, A_s_left, A_t_right, A_s_right, A_p])
if method == 'Winter':
    eGRF_W_pts = LinkSegmentModel(M, WF_W_pts, A_pts)
    r_W_pts, RMSE_W_pts, rRMSE_W_pts = CalculateMetrics(eGRF_W_pts, mGRF, printer=True)
if method == 'de Leva men':
    eGRF_MdL_pts = LinkSegmentModel(M, WF_MdL_pts, A_pts)
    r_MdL_pts, RMSE_MdL_pts, rRMSE_MdL_pts = CalculateMetrics(eGRF_MdL_pts, mGRF, printer=True)
if method == 'de Leva women':
    eGRF_FdL_pts = LinkSegmentModel(M, WF_FdL_pts, A_pts)
    r_FdL_pts, RMSE_FdL_pts, rRMSE_FdL_pts = CalculateMetrics(eGRF_FdL_pts, mGRF, printer=True)
print('_'*100)

# only thigh
print('thigh')
WF_t = [1/2, 1/2]
A_t = np.array([A_t_left, A_t_right])
eGRF_t = LinkSegmentModel(M, WF_t, A_t)
r_t, RMSE_t, rRMSE_t = CalculateMetrics(eGRF_t, mGRF, printer=True)
print('_'*100)

# thigh + tibia
print('thigh + shank')
WF_W_ts = [0.3413, 0.1587, 0.3413, 0.1587]
WF_MdL_ts = [0.3829, 0.1171, 0.3829, 0.1171]
WF_FdL_ts = [0.3772, 0.1228, 0.3772, 0.1228]
A_ts = np.array([A_t_left, A_s_left, A_t_right, A_s_right])

if method == 'Winter':
    eGRF_W_ts = LinkSegmentModel(M, WF_W_ts, A_ts)
    r_W_ts, RMSE_W_ts, rRMSE_W_ts = CalculateMetrics(eGRF_W_ts, mGRF, printer=True)
if method == 'de Leva men':
    eGRF_MdL_ts = LinkSegmentModel(M, WF_MdL_ts, A_ts)
    r_MdL_ts, RMSE_MdL_ts, rRMSE_MdL_ts = CalculateMetrics(eGRF_MdL_ts, mGRF, printer=True)
if method == 'de Leva women':
    eGRF_FdL_ts = LinkSegmentModel(M, WF_FdL_ts, A_ts)
    r_FdL_ts, RMSE_FdL_ts, rRMSE_FdL_ts = CalculateMetrics(eGRF_FdL_ts, mGRF, printer=True)
print('_'*100)

# only tibia
print('tibia')
WF_s = [1/2, 1/2]
A_s = np.array([A_s_left, A_s_right])
eGRF_s = LinkSegmentModel(M, WF_s, A_s)
r_s, RMSE_s, rRMSE_s = CalculateMetrics(eGRF_s, mGRF, printer=True)
print('_'*100)

```

```

if A.shape[0] == 7:
    A_f_left = A[5, :, :]
    A_f_right = A[6, :, :]

    # only foot
    print('foot')
    WF_f = [1/2, 1/2]
    A_f = np.array([A_f_left, A_f_right])
    eGRF_f = LinkSegmentModel(M, WF_f, A_f)
    r_f, RMSE_f, rRMSE_f = CalculateMetrics(eGRF_f, mGRF, printer=True)
    print('_'*100)

    # tibia + foot
    print('shank + foot')
    WF_W_sf = [0.3811, 0.1189, 0.3811, 0.1189]
    WF_MdL_sf = [0.3798, 0.1202, 0.3798, 0.1202]
    WF_FdL_sf = [0.3942, 0.1057, 0.3942, 0.1057]
    A_sf = np.array([A_s_left, A_s_right, A_f_left, A_f_right])
    if method == 'Winter':
        eGRF_W_sf = LinkSegmentModel(M, WF_W_sf, A_sf)
        r_W_sf, RMSE_W_sf, rRMSE_W_sf = CalculateMetrics(eGRF_W_sf, mGRF, printer=True)
    if method == 'de Leva men':
        eGRF_MdL_sf = LinkSegmentModel(M, WF_MdL_sf, A_sf)
        r_MdL_sf, RMSE_MdL_sf, rRMSE_MdL_sf = CalculateMetrics(eGRF_MdL_sf, mGRF, printer=True)
    if method == 'de Leva women':
        eGRF_FdL_sf = LinkSegmentModel(M, WF_FdL_sf, A_sf)
        r_FdL_sf, RMSE_FdL_sf, rRMSE_FdL_sf = CalculateMetrics(eGRF_FdL_sf, mGRF, printer=True)
    print('_'*100)

    # pelvis + tibia + foot
    print('pelvis + shank + foot')
    WF_W_psf = [0.1227, 0.1227, 0.678, 0.0383, 0.0383]
    WF_MdL_psf = [0.1223, 0.1223, 0.6028, 0.0387, 0.0387]
    WF_FdL_psf = [0.1270, 0.1270, 0.5824, 0.0340, 0.0340]
    A_psf = np.array([A_s_left, A_s_right, A_p, A_f_left, A_f_right])
    if method == 'Winter':
        eGRF_W_psf = LinkSegmentModel(M, WF_W_psf, A_psf)
        r_W_psf, RMSE_W_psf, rRMSE_W_psf = CalculateMetrics(eGRF_W_psf, mGRF, printer=True)
    if method == 'de Leva men':
        eGRF_MdL_psf = LinkSegmentModel(M, WF_MdL_psf, A_psf)
        r_MdL_psf, RMSE_MdL_psf, rRMSE_MdL_psf = CalculateMetrics(eGRF_MdL_psf, mGRF, printer=True)
    if method == 'de Leva women':
        eGRF_FdL_psf = LinkSegmentModel(M, WF_FdL_psf, A_psf)
        r_FdL_psf, RMSE_FdL_psf, rRMSE_FdL_psf = CalculateMetrics(eGRF_FdL_psf, mGRF, printer=True)
    print('_'*100)

    # pelvis + thigh + tibia + foot
    print('pelvis + thigh + shank + foot')
    WF_W_ptsf = [0.1, 0.0465, 0.1, 0.0465, 0.678, 0.0145, 0.0145]
    WF_MdL_ptsf = [0.1416, 0.0433, 0.1416, 0.0433, 0.6028, 0.0137, 0.0137]
    WF_FdL_ptsf = [0.1478, 0.0481, 0.1478, 0.0481, 0.5824, 0.0129, 0.0129]
    A_ptsf = np.array([A_t_left, A_s_left, A_t_right, A_s_right, A_p, A_f_left, A_f_right])
    if method == 'Winter':
        eGRF_W_ptsf = LinkSegmentModel(M, WF_W_ptsf, A_ptsf)
        r_W_ptsf, RMSE_W_ptsf, rRMSE_W_ptsf = CalculateMetrics(eGRF_W_ptsf, mGRF, printer=True)
    if method == 'de Leva men':
        eGRF_MdL_ptsf = LinkSegmentModel(M, WF_MdL_ptsf, A_ptsf)
        r_MdL_ptsf, RMSE_MdL_ptsf, rRMSE_MdL_ptsf = CalculateMetrics(eGRF_MdL_ptsf, mGRF, printer=True)
    if method == 'de Leva women':
        eGRF_FdL_ptsf = LinkSegmentModel(M, WF_FdL_ptsf, A_ptsf)
        r_FdL_ptsf, RMSE_FdL_ptsf, rRMSE_FdL_ptsf = CalculateMetrics(eGRF_FdL_ptsf, mGRF, printer=True)
    print('_'*100)

```

```

# thigh + foot
print('thigh + foot')
WF_W_tf = [0.4367, 0.4367, 0.0633, 0.0633]
WF_MdL_tf = [0.4559, 0.4559, 0.0441, 0.0441]
WF_FdL_tf = [0.4599, 0.4599, 0.0401, 0.0401]
A_tf = np.array([A_t_left, A_t_right, A_f_left, A_f_right])
if method == 'Winter':
    eGRF_W_tf = LinkSegmentModel(M, WF_W_tf, A_tf)
    r_W_tf, RMSE_W_tf, rRMSE_W_tf = CalculateMetrics(eGRF_W_tf, mGRF, printer=True)
if method == 'de Leva men':
    eGRF_MdL_tf = LinkSegmentModel(M, WF_MdL_tf, A_tf)
    r_MdL_tf, RMSE_MdL_tf, rRMSE_MdL_tf = CalculateMetrics(eGRF_MdL_tf, mGRF, printer=True)
if method == 'de Leva women':
    eGRF_FdL_tf = LinkSegmentModel(M, WF_FdL_tf, A_tf)
    r_FdL_tf, RMSE_FdL_tf, rRMSE_FdL_tf = CalculateMetrics(eGRF_FdL_tf, mGRF, printer=True)
print('_'*100)

# thigh + tibia + foot
print('thigh + shank + foot')
WF_W_tsf = [0.3106, 0.1444, 0.3106, 0.1444, 0.045, 0.045]
WF_MdL_tsf = [0.3565, 0.1444, 0.3565, 0.1444, 0.0345, 0.0345]
WF_FdL_tsf = [0.3539, 0.1152, 0.3539, 0.1152, 0.0309, 0.0309]
A_tsf = np.array([A_t_left, A_s_left, A_t_right, A_s_right, A_f_left, A_f_right])

if method == 'Winter':
    eGRF_W_tsf = LinkSegmentModel(M, WF_W_tsf, A_tsf)
    r_W_tsf, RMSE_W_tsf, rRMSE_W_tsf = CalculateMetrics(eGRF_W_tsf, mGRF, printer=True)
if method == 'de Leva men':
    eGRF_MdL_tsf = LinkSegmentModel(M, WF_MdL_tsf, A_tsf)
    r_MdL_tsf, RMSE_MdL_tsf, rRMSE_MdL_tsf = CalculateMetrics(eGRF_MdL_tsf, mGRF, printer=True)
if method == 'de Leva women':
    eGRF_FdL_tsf = LinkSegmentModel(M, WF_FdL_tsf, A_tsf)
    r_FdL_tsf, RMSE_FdL_tsf, rRMSE_FdL_tsf = CalculateMetrics(eGRF_FdL_tsf, mGRF, printer=True)
print('_'*100)

# pelvis + foot
print('pelvis + foot')
WF_W_pf = [0.678, 0.161, 0.161]
WF_MdL_pf = [0.6028, 0.1986, 0.1986]
WF_FdL_pf = [0.5824, 0.2088, 0.2088]
A_pf = np.array([A_p, A_f_left, A_f_right])

if method == 'Winter':
    eGRF_W_pf = LinkSegmentModel(M, WF_W_pf, A_pf)
    r_W_pf, RMSE_W_pf, rRMSE_W_pf = CalculateMetrics(eGRF_W_pf, mGRF, printer=True)
if method == 'de Leva men':
    eGRF_MdL_pf = LinkSegmentModel(M, WF_MdL_pf, A_pf)
    r_MdL_pf, RMSE_MdL_pf, rRMSE_MdL_pf = CalculateMetrics(eGRF_MdL_pf, mGRF, printer=True)
if method == 'de Leva women':
    eGRF_FdL_pf = LinkSegmentModel(M, WF_FdL_pf, A_pf)
    r_FdL_pf, RMSE_FdL_pf, rRMSE_FdL_pf = CalculateMetrics(eGRF_FdL_pf, mGRF, printer=True)
print('-'*100)

# pelvis + thigh + foot
print('pelvis + thigh + foot')
WF_W_ptf = [0.1, 0.1, 0.678, 0.061, 0.061]
WF_MdL_ptf = [0.1416, 0.1416, 0.6028, 0.057, 0.057]
WF_FdL_ptf = [0.1478, 0.1478, 0.5824, 0.061, 0.061]
A_ptf = np.array([A_t_left, A_t_right, A_p, A_f_left, A_f_right])

if method == 'Winter':
    eGRF_W_ptf = LinkSegmentModel(M, WF_W_ptf, A_ptf)
    r_W_ptf, RMSE_W_ptf, rRMSE_W_ptf = CalculateMetrics(eGRF_W_ptf, mGRF, printer=True)
if method == 'de Leva men':
    eGRF_MdL_ptf = LinkSegmentModel(M, WF_MdL_ptf, A_ptf)

```

```

r_MdL_ptf, RMSE_MdL_ptf, rRMSE_MdL_ptf = CalculateMetrics(eGRF_MdL_ptf, mGRF, printer=True)
if method == 'de Leva women':

```

▼ 12.4 Sensitivity Analysis

```

def SensitivityAnalysisVisual(step, start_f, stop_f, fs, M, WF, A, mGRF):

    fc = np.arange(start_f, stop_f+step, step)    # source: Karatsidis
    order = np.arange(1, 7, 1)
    o = 2

    best = []
    all = []
    previous_bestRMSE = [np.inf]

    for f in fc:
        print(f, 'Hz')
        # filter (Note: make sure that the data is already the frame of interest and resampled)
        A_filtered = FilterAndResampleData(A, fs, o, f, 'lowpass', 1, mGRF.shape[0])
        # estimate the GRF
        eGRF = LinkSegmentModel(M, WF, A_filtered)
        # compare with the mGRF
        r, RMSE, rRMSE = CalculateMetrics(eGRF, mGRF)
        print(RMSE)
        print(rRMSE)
        all.append([o, f, r, RMSE, rRMSE])

        if RMSE[2] < previous_bestRMSE:
            best.append([o, f, r, RMSE, rRMSE])
            previous_bestRMSE = RMSE[2]

    Plot2(eGRF[:, 2], mGRF[:, 2], ['eGRF', 'mGRF'], np.arange(0, eGRF.shape[0]/fs, 1/fs), 'Time [s]', '
    print('-'*150)
    return best, all

```

```

def SensitivityAnalysis(step, start_f, stop_f, fs, M, WF, A, mGRF, graph=False):

    fc = np.arange(start_f, stop_f+step, step)    # source: Karatsidis
    order = np.arange(1, 7, 1)

    best = []
    all = []
    previous_bestRMSE = [np.inf]

    for o in order:
        for f in fc:
            # filter (Note: make sure that the data is already the frame of interest and resampled)

```