

# MuSkeMo Manual



**Pasha van Bijlert**

[pashavanbijlert@gmail.com](mailto:pashavanbijlert@gmail.com)

**Github Repository:** [Github](#)

**Bug Reports and Feature Requests:** Submit an issue on [Github](#)

Version 0.9.8  
October 20, 2025

**How to cite MuSkeMo:** I am preparing a publication to submit for peer-review describing MuSkeMo. Until that is available, please cite the preprint on bioRxiv if you used MuSkeMo for your work.

PA van Bijlert. *MuSkeMo: Open-source software to construct, analyze, and visualize human and animal musculoskeletal models and movements in Blender.* bioRxiv (preprint) 2024.12.10.627828; doi: 10.1101/2024.12.10.627828.

# Contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
1.1	Installation instructions . . . . .	4
1.2	Sample Datasets . . . . .	4
1.3	Units . . . . .	4
1.4	A Note on Precision . . . . .	5
1.5	Orientations . . . . .	5
<b>2</b>	<b>Using MuSkeMo</b>	<b>5</b>
2.1	Where MuSkeMo stores model data . . . . .	6
2.2	Model components and modifiers . . . . .	6
2.3	Blender Auto Save . . . . .	7
2.4	Default pose . . . . .	7
<b>3</b>	<b>MuSkeMo's Panels</b>	<b>8</b>
3.1	MuSkeMo Global Settings Panel . . . . .	8
3.2	Mesh tools panel . . . . .	8
3.2.1	Mesh alignment . . . . .	8
3.2.2	Checking geometry intersections . . . . .	9
3.3	Inertial properties panel . . . . .	9
3.3.1	Generate minimal convex hulls . . . . .	10
3.3.2	Expand convex hulls . . . . .	10
3.3.3	Whole body mass from convex hulls . . . . .	11
3.4	Body panel . . . . .	12
3.4.1	Visual geometry (meshes) . . . . .	12
3.5	Joint panel . . . . .	13
3.5.1	Fitting primitive objects for joints . . . . .	13
3.6	Muscle panel . . . . .	14
3.6.1	Creating a muscle and adding points . . . . .	14
3.6.2	Visualization radius . . . . .	14
3.6.3	View length in realtime . . . . .	15
3.6.4	Assigning wrapping . . . . .	15
3.6.5	Moment arms . . . . .	15
3.6.6	Plotting . . . . .	15
3.7	Anatomical & local reference frames panel . . . . .	16
3.8	Landmark & marker panel . . . . .	17
3.9	Contact panel . . . . .	18
3.10	Export panel . . . . .	18
3.11	Import panel . . . . .	18
3.11.1	OpenSim model import . . . . .	18
3.11.2	Gaitsym 2019 model import . . . . .	19
3.11.3	MuJoCo model import . . . . .	19
3.11.4	Other simulators . . . . .	21
3.12	Visualization panel . . . . .	21
3.12.1	Volumetric Muscles . . . . .	21
3.12.2	Trajectory import . . . . .	21
3.12.3	Visualization options . . . . .	22
3.12.4	Default colors . . . . .	23
3.13	Reflection panel . . . . .	23
3.14	MuSkeMo utilities . . . . .	23
3.14.1	OpenSim conversion . . . . .	24
3.14.2	Fitting muscle lines of action . . . . .	24
3.14.3	Compute closest point between objects . . . . .	24
3.14.4	Muscle length script . . . . .	24
3.14.5	Joint repositioning script . . . . .	24
3.14.6	Moment arms analysis . . . . .	25
3.14.7	Range of motion and pose sampling scripts . . . . .	25
3.14.8	Volumetric muscle visualization volume comparison . . . . .	26

<b>4 MuSkeMo Data Types</b>	<b>27</b>
4.1 BODY . . . . .	27
4.2 GEOMETRY . . . . .	27
4.3 MUSCLE . . . . .	28
4.3.1 Simple muscle visualization . . . . .	28
4.3.2 Volumetric muscle visualization . . . . .	28
4.3.3 Muscle wrapping . . . . .	31
4.4 JOINT . . . . .	32
4.5 LANDMARK . . . . .	33
4.6 CONTACT . . . . .	33
4.7 FRAME . . . . .	34
4.8 GEOM_PRIMITIVE . . . . .	34
4.9 WRAP . . . . .	34
<b>A Euler angles</b>	<b>34</b>
A.1 Conventions used by MuSkeMo . . . . .	34
A.2 Understanding decomposed successive rotations . . . . .	35
<b>B Rotation quaternions</b>	<b>38</b>
<b>C Axis angle rotation matrix</b>	<b>38</b>
<b>D Validation tests</b>	<b>39</b>
D.1 Mesh inertial properties . . . . .	39
D.2 Composite bodies and meshes . . . . .	40
D.3 Re-expressing matrices and vectors in arbitrary frames . . . . .	43
D.4 Muscle moment arms . . . . .	43
D.5 Pose sampler stress test . . . . .	45
<b>E Acknowledgements</b>	<b>47</b>
<b>References</b>	<b>47</b>

# 1 Introduction

MuSkeMo is a tool for musculoskeletal modeling in Blender. MuSkeMo allows you to translate biological 3D scans to useful biomechanical information, in the form of full models or model components. These can then be used for further analysis within Blender, or exported for simulations in other biomechanical software. Simulation trajectories can be imported back into MuSkeMo to make publication-ready stills and videos, using Blender's built-in raytracing rendering. MuSkeMo can also be used for 3D landmarking and for simply calculating inertia tensors from 3D scans.

This MuSkeMo documentation file is meant to complement the video tutorial series on YouTube. Users are encouraged to download the sample datasets and follow along with the video tutorials. This documentation is focused on features specific to MuSkeMo, and where necessary provides suggestions regarding Blender settings and features. Because Blender can be overwhelming at first, new users are recommended to follow the "Donut tutorial" series by BlenderGuru on YouTube. For Blender-specific questions, the reader is referred to the extensive Blender documentation. MuSkeMo currently officially supports Blender 4.1 - 4.5.

## 1.1 Installation instructions

To download: Go to the releases page on the Github repository, and download the most recent version of "MuSkeMo.zip". The .zip file is used to install the plugin, but also contains a folder with utility functions (e.g., for OpenSim conversion).

To install (Fig. 1): In Blender v4.1, go to Edit → preferences → Add-ons and then click "Install...", and then select your downloaded zip file. Then type in "MuSkeMo" in the search bar of the addon window, and enable the plugin by pressing the check mark. If using Blender 4.2+, the install button is replaced with a downwards arrow, which gives "Install from disk".

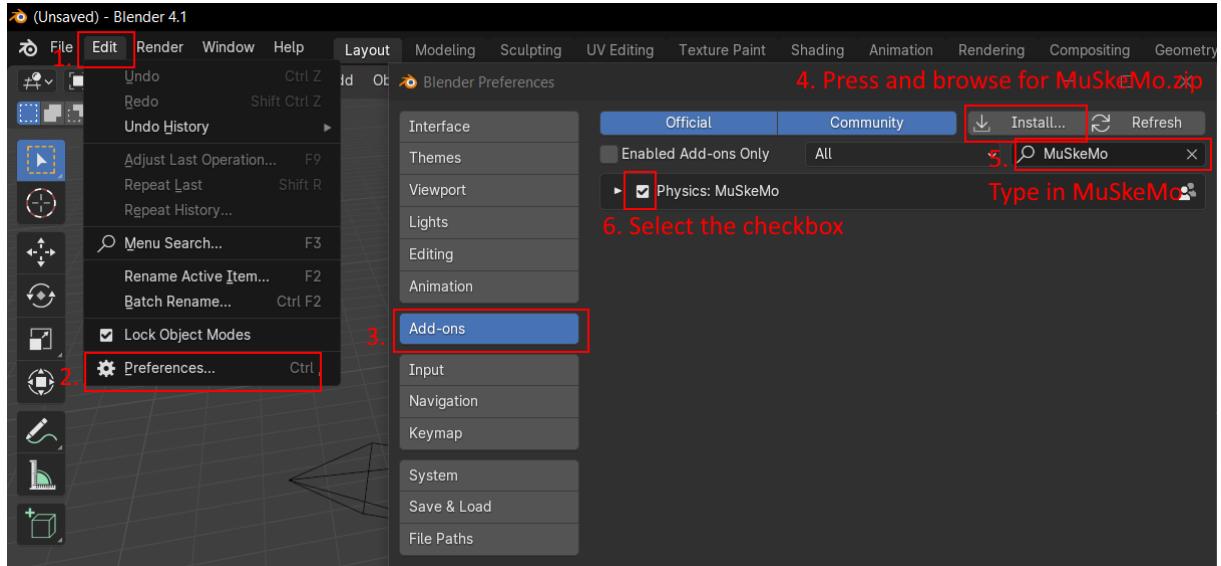


Figure 1: Installing MuSkeMo in Blender version 4.1. In 4.2+, Step 4 is replaced with pressing a downwards arrow and selection "Install from disk"

## 1.2 Sample Datasets

Some of the video tutorials use sample datasets, in which case they are also made available from Github. Currently, there is only one sample dataset, which includes two OpenSim models created using MuSkeMo, and compatible simulation trajectories so the user can create a video.

## 1.3 Units

MuSkeMo uses the following units:

- **Spatial distance/position:** metres (m)

- **Mass:** kilogram (kg)
- **Moment of inertia:** kilogram metre squared ( $\text{kg m}^2$ )
- **Force:** Newton (N)
- **Angle:** Degrees ( $^\circ$ , only for pennation angle). Internally MuSkeMo uses radians during computations.

**WARNING:** Blender’s default units are in metres. Many of the calculations in MuSkeMo are derived from the spatial position of individual points (vertices) of the 3D meshes. If you change the base units of Blender, these calculations are also implicitly scaled, and the units will no longer be correct (e.g., if you change Blender’s base units to centimetres, linear dimensions will be off by a factor of 100, but volumetrics and inertial properties will be off by a factor of  $100^3$ ). MuSkeMo currently only officially supports metres as the base unit, correct scaling of the outputs if using other units is up to the user.

## 1.4 A Note on Precision

MuSkeMo is built using the Blender Python API, which internally uses double precision numbers (64-bit floating point,  $\sim 16$  significant digits). However, Blender itself stores numbers using single precision (32-bit floating point,  $\sim 8$  significant digits), including all 3D point and mesh data. This means that precision beyond 8 significant digits cannot be expected when using MuSkeMo. Single precision floats nevertheless have a very large range of numbers that can be represented—the smallest number that can be accurately reported is in the order of  $1 \times 10^{-38}$ , approximately 1 million times smaller than the mass of an electron in kilograms.

Because computing inertial properties from 3D vertex data involves successive multiplications of single precision numbers, mass and inertial properties have an expected accuracy of 5-6 digits (see D.1).

## 1.5 Orientations

Orientations are represented as XYZ body-fixed Euler angles (see Appendix A) and as unit quaternions (see Appendix B). Internally, MuSkeMo avoids Euler angles where possible because they are prone to gimbal lock, but popular simulators often use Euler angles by default (e.g., OpenSim, SCONE).

# 2 Using MuSkeMo

MuSkeMo lives in Blender panels (Fig. 2A). Panels can be interactively resized and collapsed, which can be useful depending on your screen size. The panels contain buttons that perform operations on the data (implemented as Blender Operators). When hovering over a button, a tooltip appears that briefly explains the workings of the button in question.

Nearly all functions in MuSkeMo work by selecting the correct objects in the scene, and then pressing a button. If the selected objects are incompatible with the button in question, MuSkeMo will display an error message that explains what went wrong. To make the user aware of what MuSkeMo object (types) are currently selected, each has a live selection display for the relevant object types.

Although Blender does support using a trackpad, it is substantially more intuitive to navigate with a mouse, and a keyboard with a separate numberpad. There are also certain settings that make navigation easier (see Section 3.1).

**WARNING:** Ensure that 3D meshes (visual geometry and tissue outlines) have all their transformations applied before you start constructing the model. To do this, select the object, press **Control + A**, and select “All transforms”. This accounts for the way that Blender stores local transformations. When you move, rotate, and/or scale 3D models in Blender, these transformations are initially stored only locally in the object’s data (an extra storage layer in Blender). This means that the object’s position has not changed in 3D space, you have defined an extra local transformation that can easily be undone. Some of the functions in MuSkeMo may behave unpredictably unless transformations have first been applied.

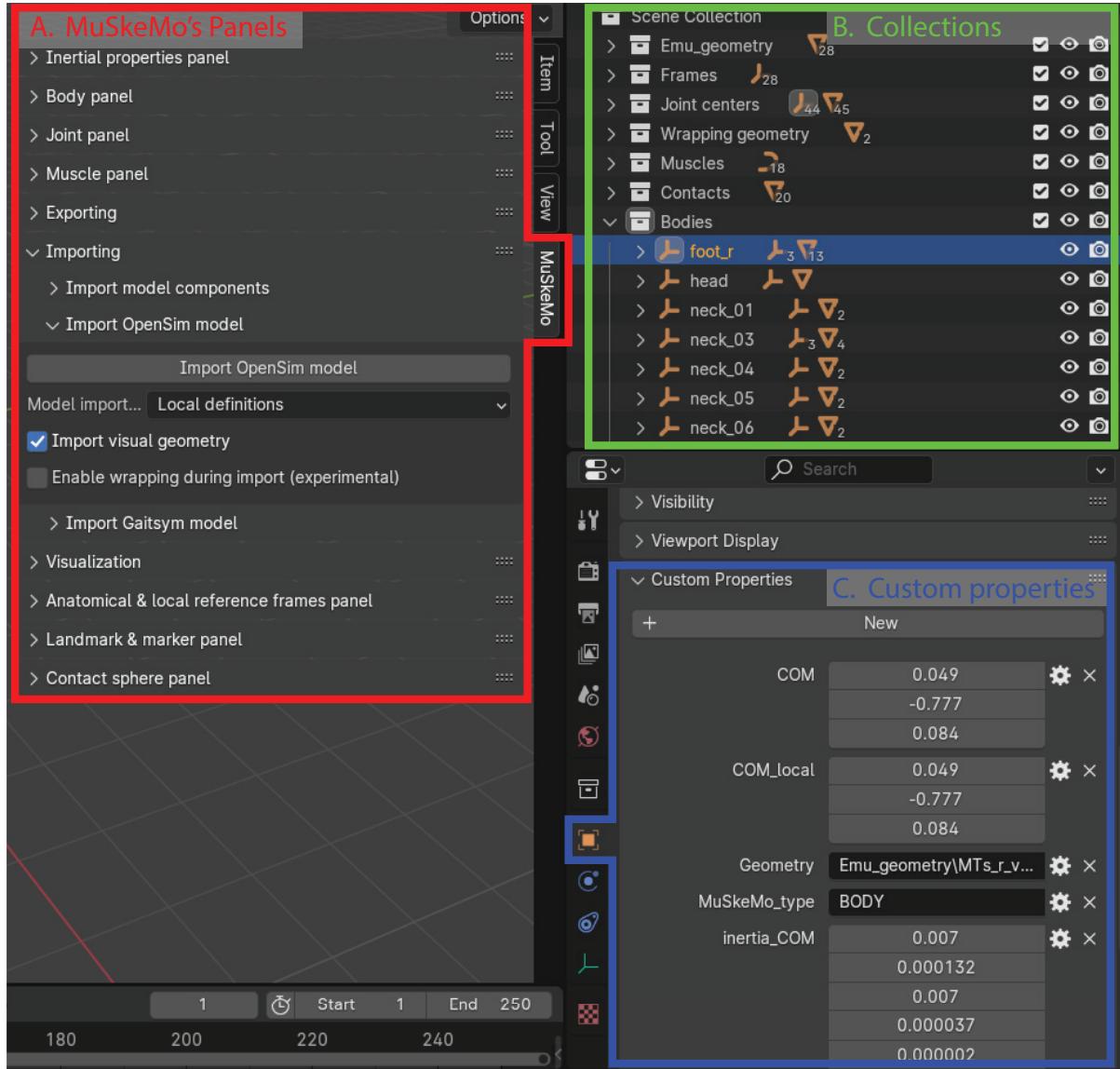


Figure 2: A. All of MuSkeMo’s functionality can be accessed via these panels. The panels can be interactively resized and collapsed. B. After creating or importing model components, they are stored in Blender collections. If child objects are not visible, turn off object children in the outliner filter (see Section 3.1). C. A component’s data are stored as custom properties. Press the yellow square, scroll all the way down, and open the Custom Properties dropdown

## 2.1 Where MuSkeMo stores model data

All the model components and other user-created objects are stored in Blender Collections (Fig. 2B), which are essentially just folders. The collection names all have sensible defaults in MuSkeMo (e.g., “Bodies” for bodies), but can be changed if desired. The collection names are also used as the default filenames during import/export, but this is also customizable. By default, child objects aren’t always visible in the collection that they’re actually in (see Section 3.1).

For each model component, data created by MuSkeMo are stored in Blender as Custom Properties assigned to each object in question (Fig. 2C, see also Section 4 on MuSkeMo Data Types).

## 2.2 Model components and modifiers

MuSkeMo creates a new Blender Object for each model component. Blender has many different object types, but MuSkeMo mainly uses “Mesh”, “Curve”, and “Empty” (see 4 for a full specification of all the used data types). Objects in Blender do not interact with other objects, unless the user somehow defines

their inter-relationships. MuSkeMo handles all of this under the hood. For instance, components such as contact spheres need to be parented to specific bodies in the model, and MuSkeMo handles this while also keeping track of these inter-relationships in "Custom properties" so they are exposed to the user (see 2.1).

Combining parenting and Blender's native object types does not provide all of the required functionality. MuSkeMo extensively makes use of Blender's Modifier system for both generating and visualizing components. For instance, individual muscle curve points cannot be attached to bodies using Blender's parenting system, so MuSkeMo achieves this by adding a hook modifier for each curve point (see 4.3).

MuSkeMo uses both regular modifiers (which perform a predefined operation/modification on the object, e.g. the Bevel modifier is used to round the edges of muscle visualizations), but also several custom-made Geometry nodes modifiers.

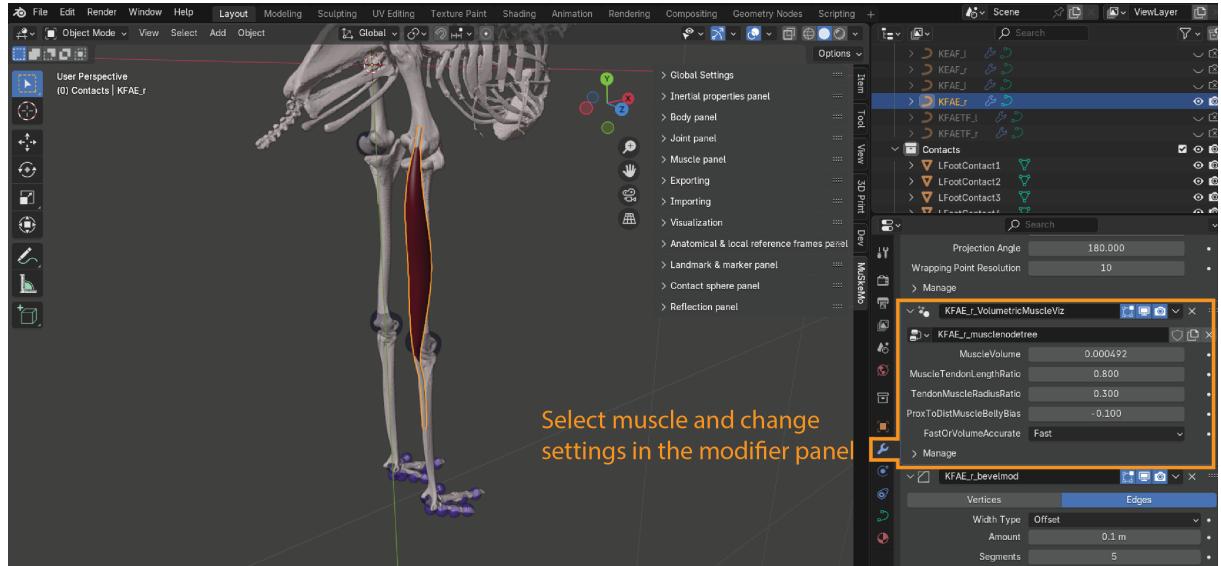


Figure 3: Some of MuSkeMo's custom Geometry Nodes modifiers have specific settings that can interactively change their behaviour (pictured is the Volumetric Muscle visualization node).

Geometry Nodes modifiers are a very powerful way to perform a customized sequence of operations on a specific object. MuSkeMo uses custom-made geometry nodes modifiers for muscle visualization (Fig. 3, sec. 4.3.1 and 4.3.2), muscle wrapping (sec. 4.3.3), creating parametric wrap geometry (sec. 4.9), amongst other features. The most important inputs for most MuSkeMo's custom Geometry nodes are accessible from the modifier panel itself (Fig. 3), but advanced Blender users can also dive into the nodes themselves for more in-depth modifications to their behavior. Where possible MuSkeMo tries to reuse node groups, which means that it is possible to make global changes to geometry node behavior by changing the parameters of a single node group. For example, the visualization of all the muscles can be changed by modifying the underlying node groups (4.3.1 and 4.3.2).

By default, MuSkeMo color codes each component (see 3.12.4 for details).

## 2.3 Blender Auto Save

Blender is quite stable, but it is possible for it to crash. Fortunately, by default, Blender has an Auto Save function. After a crash, restart Blender, and go to File, Recover, Auto Save. Remember to save the recovered file.

## 2.4 Default pose

Constructing a model in Blender provides the user with a lot of freedom to interactively reposition model elements during model construction. However, many of the computations are only valid in the specific pose they are computed in. For example, when computing mesh inertial properties using the inertial properties panel (3.3), the inertial properties in the global reference frame are only valid as long as the orientation and position (the pose) of the mesh do not change. Similarly, object positions and orientations

with respect to parent frames (4.7) are only valid during export if their relative positions and orientations remain unchanged.

To prevent incorrect exports, MuSkeMo keeps track of the 'default\_pose', which is the pose in which the relevant computation was performed. If the user tries to perform an operation that can only be performed in the default pose, the operation is cancelled (with an error message). In that situation, you must reposition the model components back into the default pose before exporting. MuSkeMo provides a button for this: "Reset to default pose", available from the Global settings panel (sec. 3.1), the Joint panel (sec. 3.5), the Body panel (sec. 3.4), the inertial properties panel (sec. 3.3), and the export panel (sec. 3.10). The button works by selecting one or multiple objects that have a default pose, and then pressing the button. If a single object is selected and it is part of a model, MuSkeMo traverses the model until it finds the root joint or body (i.e., the component at the top of the parent hierarchy), and then resets all the joints and bodies to their default pose. If multiple components of the same model are selected, MuSkeMo only performs the operation once (because they will both have the same root). If multiple disjointed objects are selected, MuSkeMo will reset the default pose for all the separate sections (including individual meshes with inertial properties).

Alternatively, if the user wishes to export the model in the changed pose, it is necessary to recompute all the model parameters in this new pose. For inertial properties and bodies, this means recomputing and reassigning the inertial properties, for other components it means detaching and reparenting the objects.

## 3 MuSkeMo's Panels

This section describes all of MuSkeMo's panels and how to use them. Most of MuSkeMo's features work by selecting the correct combination of objects, and then pressing one of the buttons in the panels. MuSkeMo's panels show you how many and what types of objects (e.g., "BODY") you have selected, and will give an explanatory error message if you try to use a panel incorrectly (e.g.,: assigning a parent body to a joint requires selecting one body and one joint, and MuSkeMo informs you if you didn't do this). Most of the panels also allow you to create new objects, which means you have to type in a (unique!) object name and press a button.

### 3.1 MuSkeMo Global Settings Panel

The Blender user interface has several default settings that do not work well with MuSkeMo. By default, Blender forces the Z-axis as the "up" direction in the viewport. The International Society of Biomechanics assumes Y as the up direction. To achieve this, the user should set the view rotation setting to "Trackball", and use "orbit around selection". The "MuSkeMo Global Settings" panel provides a button that does this automatically: "Set recommended navigation settings".

The panel also has a button named "Set object children visibility". This toggles off "Object Children" in the outliner. In Blender's behavior, objects are placed under their parents in the outliner, but this would for instance result in a body being placed under the parent joint in the "Joint collection", instead of in the "Body collection". Turning off "Object children" avoids this behavior. MuSkeMo automatically calls the "Set object children visibility" operator when the user creates a Body, Joint, or Muscle for the first, so in most cases, you will not need to set this yourself. However, if you find yourself in a situation where you cannot find your objects in the outliner, even though you can clearly see them in the scene, this button may solve your problem.

### 3.2 Mesh tools panel

This panel combines several tools that are generally useful when manipulating or repositioning meshes. Some of these are also available in different panels (e.g., the intersection checker is also made available in the visual geometry subpanel 3.4.1).

#### 3.2.1 Mesh alignment

Aligning 3D meshes in space is a common task when dealing with 3D biomechanical data. To use MuSkeMo's mesh alignment tool, simply select two meshes, designate which mesh is the target (stationary) object, and which one is the free mesh (which will be moved to achieve alignment), and then press "Align Meshes". The alignment assumes that you have approximately aligned the meshes yourself, and

that they have the same scale. You can align entire meshes, or selected sections of meshes (e.g., when trying to match articular surfaces).

The alignment uses an iterative closest point algorithm (point to plane, following [9]), using Blender’s KD-tree module to improve performance [43]. Performance is further improved with (random) subsampling (see [28]), and the subsampling is slowly ramped down after several iterations so the final alignment is based on the entire meshes. Performance will still be slow with heavy meshes.

The following settings are available:

1. **Free Object.** Selects which object is allowed to move during ICP point-to-plane mesh alignment.
2. **Alignment mode.** Toggle whether you want to align the entire mesh, or only selected portions of two meshes. When aligning selected mesh portions, both meshes should have selections set in Edit mode (select both meshes, hit TAB, and select the relevant mesh sections). Selecting portions of meshes can be useful if one (or both) of the meshes is incomplete, or when trying to match corresponding articular surfaces of bones.
3. **Max Iterations.** Maximum number of iterations performed during ICP point-to-plane mesh alignment. Higher values may improve accuracy but increase computation time.
4. **Tolerance.** Error tolerance used to determine convergence of the ICP alignment. Smaller values enforce stricter convergence at the cost of runtime. Values below  $10^{-8}$  are unlikely to improve results significantly.
5. **Start Sample Ratio.** Fraction of points used for sub-sampling at the beginning of the ICP process. A lower ratio speeds up early iterations while maintaining coarse alignment.
6. **End Sample Ratio.** Should be higher than Start Sample Ratio. Fraction of points used for sub-sampling at the end of the ICP process. If you set this to lower than 1, the final iterations will still use a sample ratio of 1 (ensuring that the final alignment is based on all points).
7. **No. iterations before end ratio.** Number of iterations before the sample ratio reaches the defined end value. Controls how quickly the sampling density increases during ICP.

### 3.2.2 Checking geometry intersections

MuSkeMo implements a mesh intersection checker, which is used by the pose sampling scripts in the MuSkeMo utilities folder (see sec. 3.14.7). This works via Blender’s BVHTree module, which is an implementation of Bounding Volume Hierarchies [15]. Essentially, the mesh is bounded by successively smaller objects in a nested tree structure, and Blender’s BVHTree traverses down the tree while checking whether these bounding objects intersect with other objects in the scene. By starting with large bounding volumes, and only traversing the tree if the larger objects trigger an intersection, this is a computationally efficient way of checking for intersections. The Wikipedia page on Bounding Volume Hierarchies also gives an accessible explanation on how this works. While intersection panel button is useful to interactively check for intersections (e.g., when articulating bones), it shines in scripting, because it allows between 50 to 200 pose checks per second (see sec. 3.14.7).

## 3.3 Inertial properties panel

The main goal of this panel is to compute inertial properties from 3D volumetric meshes, eg. from CT-segmentations or surface scans. Inertial properties are not dynamic, if you move the 3D meshes or would like to change their densities, you must recompute their inertial properties, otherwise COM, mass, and/or inertia can be outdated. MuSkeMo warns you if this has occurred, by tracking the ‘default\_pose’ of each mesh as a custom property (see 2.4). Density can only be changed by changing the ‘Segment density’ in the panel and recomputing the object’s inertial properties.

To compute the volumetric inertia tensor (with elements in the unit  $m^5$ ) of a triangular mesh, MuSkeMo implements the solution derived and presented in [14]. This gives an exact solution for the volumetric moments of inertia of a closed, triangulated mesh, based on the Divergence Theorem. This algorithm requires the mesh to be triangulated and watertight to provide meaningful results, and MuSkeMo alerts the user if this is not the case. The volumetric tensor is multiplied by density (in  $\text{kgm}^{-3}$ ) to acquire the inertial tensor elements (in units  $\text{kgm}^2$ ).

It is possible to change the density property of an object, after which you will have to select the object and rerun "Compute for selected meshes". See D.1 for a validation of the outputs.

### 3.3.1 Generate minimal convex hulls

Several studies have used convex hulls as the starting point for estimating inertial properties of (extinct) animals directly from full skeletons [30, 6, 8, 7, 11, 23, 38, 4]. In this subpanel, you can automatically generate convex hulls around (skeletal) meshes in a collection, and then apply methods from the literature to reconstruct the inertial properties.

You have to designate the "Skeletal mesh collection" where your skeletal meshes are located. This defaults to the "Geometry" collection, but it can also be a different collection. Each separate object in the collection will receive its own convex hull, so the meshes in the collection should represent functional body segments. See [30, 11, 23] for examples. Convex hulls are placed in a new collection.

The recommended workflow is to organize all the skeletal meshes in a single collection. Ensure that all the skeletal meshes contain the correct segment names in their full name (see below). Then, generate the minimal convex hulls, and use one of the expansion modes to expand those convex hulls.

### 3.3.2 Expand convex hulls

Both the expansion panels (arithmetic and logarithmic) work with segment names and corresponding scale factors or logarithmic parameters. If only "whole\_body" is typed in as Segment name 1 (with no other segment names defined), all the objects in the target collection will be treated as one - in arithmetic mode, all objects are scaled by a single factor, in logarithmic mode, all object volumes are summed before determining the whole-body scale factor.

If segment 1 is not "whole\_body", or if you have more than one segment name defined in the panel, MuSkeMo tries to match whatever segment names you have typed into the panel to the names of the objects in your target collection. E.g., if you've typed "neck" as one of the segment names in the panel, with a corresponding scale factor, all the objects in Convex hull collection that have the case-sensitive word "neck" in their name will be expanded by the scale factor (e.g., "neck\_1", "neck\_prox.obj", but not "Neck\_1" and also not "torso"). The "Expansion Template" dropdown menu gives several presets from the literature, and you can customize them if desired.

MuSkeMo assumes you want bilaterally symmetric models, and therefore when using the expansion functions, MuSkeMo generates bilaterally symmetric expanded hulls for the following segments: "head", "neck", "torso", "tail" (meaning the target skeletal objects should have those strings in their names). Furthermore, MuSkeMo also applies directional scaling. The following segments are scaled about the Y and Z axes (and thus not the X axis): "head", "neck", "torso", "tail", "forearm", "hand", "toe". All other segments are scaled about the X and Z axes. The resultant scaled shapes may not be very biologically realistic. You could add a "Maintain volume" constraint and change the shape in Blender (this was a suggestion by Matt Dempsey). Apply the constraint after you are satisfied.

Under **Expand convex hulls - arithmetic**, you can use arithmetic expansion factors (i.e., linear scale factors) to scale your hulls. If a segment initially has a volume of 1 m<sup>3</sup> and a scale factor of 1.2, the resultant segment will have a volume of 1.2 m<sup>3</sup>.

- **Custom:** Type in the segment names, and the desired scale factor.
- **Macaulay 2023 Bird:** The per-segment "Bird" average expansions from Macaulay et al. 2023 [23], supplementary data S6.
- **Macaulay 2023 Non-Avian Sauropsid:** The per-segment "Croc\_Lizard" average expansions from Macaulay et al. 2023 [23], supplementary data S6.
- **Macaulay 2023 Average (Bird and Non-Avian Sauropsid):** The per-segment "Av." average expansions from Macaulay et al. 2023 [23], supplementary data S6.
- **Sellers 2012 Large Mammals:** This is a "whole\_body" scale factor of 1.206 as described in [30]. This is the first introduction of this method, and the user is warned that the the Sellers 2012 curve was not calibrated using direct measurements of body mass for the analyzed skeletons [30].

Under **Expand convex hulls - logarithmic**, it is possible to use log-transformed regression equations from the literature to scale your hulls. The equations have the form:

$$y = 10^a \cdot x^b \cdot 10^{\frac{\ln(10)}{2} MSE} \quad (1)$$

This is known as an allometric power curve, and is the result of an ordinary least-squares regression of log-transformed data (see [1, 6]).  $a$  is the y-intercept of the regression,  $b$  is the slope of the regression,  $x$  is the input convex hull volume, and  $y$  is the scaled hull volume.  $MSE$  is the mean squared error of the regression, also called the variance of the residuals, or  $\sigma^2$ , with  $\sigma$  being the standard deviation or standard error of the residuals. The term  $10^{\frac{\ln(10)}{2} MSE}$  is used for a bias correction to account for the fact that we are transforming log-transformed data back to arithmetic units, see [3, 33, 32].  $MSE$  is assumed to be computed in base-10 logarithms, hence the extra factor  $\ln(10)$  [33, 32]. Where  $MSE$  was not or inconsistently reported by the original studies, MuSkeMo sets it to 0 in the expansion templates, which implies a small, systematic, underestimation of the predictions by the power curve. By unticking the box "Apply Bias Correction", the bias correction is not applied during expansion even if the template has a value pre-filled (i.e., setting  $MSE$  to zero during the computations).

- **Custom:** Type in the segment names, and the desired scale factor.
- **Macaulay 2023 Logarithmic Bird:** The logarithmic per-segment "Bird" average expansions from Macaulay et al. 2023 [23], supplementary data S4.
- **Macaulay 2023 Non-Avian Sauropsid:** The logarithmic per-segment "Non avian sauropsid" average expansions from Macaulay et al. 2023 [23], supplementary data S5.
- **Macaulay 2023 Average (Bird and Non-Avian Sauropsid):** The logarithmic per-segment "all taxa" average expansions from Macaulay et al. 2023 [23], supplementary data S3.
- **Coatham 2021 Logarithmic Mammal (rewritten volumetric):** The logarithmic per-segment mammal equations from Coatham et al. 2021 [11]. This implementation factors out the density to transform them into volumetric expansions, see below. MuSkeMo squares their "root mean squared errors" (RMSE) to acquire mean squared errors ( $MSE$ ) in eq. 1, but sets it to zero for the torso segment because the reported RMSE for the torso (3.42) leads to a bias correction factor of  $7 \cdot 10^5$ .

Coatham et al. directly regressed convex hull mass against CT-based tissue outline mass, assuming a constant density of  $\rho = 1000 \text{ kgm}^{-3}$ . Because this essentially gives a constant factor, these can be factored out to create volumetric equations as follows:

$$\begin{aligned} y \cdot \rho &= 10^a \cdot (x \cdot \rho)^b \cdot 10^{\frac{\ln(10)}{2} MSE} \\ y &= 10^a \cdot \rho^b / \rho \cdot x^b \cdot 10^{\frac{\ln(10)}{2} MSE} \\ y &= 10^a \cdot \rho^{b-1} \cdot x^b \cdot 10^{\frac{\ln(10)}{2} MSE} \end{aligned} \quad (2)$$

Here,  $y$  is still the expanded volume, and  $x$  is the convex hull volume. The original  $a$  can then be replaced with:

$$a_{\text{new}} = \log_{10} (10^a \cdot \rho^{b-1}) \quad (3)$$

### 3.3.3 Whole body mass from convex hulls

Several studies have regressed total body mass to summed convex hulls of the entire body, as an approach to estimate total body mass from skeletons [6, 8, 7, 38].

For the mass estimation equations,  $x$  is the summed volume of the convex hulls of all the body segments in  $\text{m}^3$ , and  $y$  is the computed total body mass in kg.

This subpanel sums all the convex hull volumes in a designated collection, then uses the user-selected empirical equation to compute total body mass (in kg). This assumes the convex hulls are of the complete skeleton (including both left and right sides). Available equations are:

- **Wright 2024 Logarithmic Tetrapods:** Regression equation from [38], based on primary data, and reused data from [11] and [30] (the latter includes estimates instead of measurements of total body mass)
- **Brassey 2018 Logarithmic Primates:** from [7].
- **Brassey 2016 Logarithmic Pigeons, Eviscerated:** from [8], using the eviscerated pigeons equation.
- **Brassey 2016 Logarithmic Pigeons, Combined:** from [8], using the combined pigeons equation.
- **Brassey 2014 Logarithmic Primates:** from [6].
- **Brassey 2014 Logarithmic Mammals:** from [6], this reuses data from [30] which includes body mass estimates instead of direct measurements.

The original Brassey et al. 2016 [8] pigeon equations assume volume is  $\text{mm}^3$ , and provide the mass in g. To convert the output to kilograms (multiply by  $10^{-3}$ ) and allow the input to be in  $\text{m}^3$  (multiply input by  $10^9$ ), it is possible to rewrite equation 1.

$$y = 10^{-3} \cdot 10^a \cdot (x \cdot 10^9)^b \cdot 10^{\frac{\ln(10)}{2} MSE} \quad (4)$$

After rewriting, this gives:

$$a_{new} = a - 3 + 9 \cdot b \quad (5)$$

Wright et al. [38] multiply the summed convex hull volume by a density of  $\rho = 1000 \text{ kgm}^{-3}$  before applying their power curve. This implies that mass scales with  $\rho^b$  in their predictive model.

### 3.4 Body panel

Create new bodies by typing in a (unique) name, and then pressing the 'Create new body' button. The newly created body will have all 'NaN's as inertial properties, and will be centered at the origin. Once you assign rigid body parameters, the body will be moved to the correct position in the global reference frame. A body's position is always its center of mass in the global frame, and its orientation is always aligned with the global frame.

You can assign rigid body parameters in two ways. The recommend approach is to assign values that were computed for meshes in the inertial properties panel (see 3.3). A single body can represent the combined inertial properties of more than one mesh. MuSkeMo computes the combined inertial properties using the parallel axes theorem [35, 27]. Simply select the target body, and 1+ source objects, and press 'Assign precomputed inertial properties'. MuSkeMo will give you an error message if you selected objects that don't have inertial properties precomputed, and the live selection boxes in the panel can help you with ensuring you have selected the correct (number of) objects. See D.2 for a validation of these features.

It is also possible to assign inertial properties manually. This is not recommended, but it is possible by manually typing the values in the Body's custom properties. If going this route, the COM position can be updated using the button in the "Assign inertial properties manually" subpanel.

Inertial properties are not dynamic, if you move the source objects that the rigid bodies were based on, or change their densities, you must recompute all inertial properties of the body (see 2.4). Otherwise COM, mass, and or inertia can be outdated. MuSkeMo warns you if you attempt to export inertial property data in a different pose than the default pose.

#### 3.4.1 Visual geometry (meshes)

In this panel, you can also attach visualization geometry (eg., bone meshes) to bodies. Select the meshes and the target body, and press the button.

The intersection checker (sec. 3.2.2) is also available here, for easy access when articulating bone meshes.

### 3.5 Joint panel

Create new joints by typing in a unique name and pressing 'Create new joint'. To and assign a parent or child body, select the joint and the target body, and press the respective button.

Until joints are parented, there is no 'default pose' state being tracked (see 2.4), and the global positions and orientations are listed as NaN in the custom properties (but are exported correctly if the user wants to export unparented joints). If a parent or child body has an anatomical (local) reference frame assigned, MuSkeMo automatically computes the relative positions and orientations in these frames as well. Orientations are stored as body-fixed, XYZ-Euler angles and as quaternions. All data that are created are included during export (if local frames are not assigned, these values will be NaN).

If you want to change a joint's position or orientation, detach the parent and child bodies first, or use the 'Match orientation' and 'Match position' buttons in the Joint utilities subpanel.

#### Joint coordinate names

It is possible to define coordinate names in the joint panel. After exporting from MuSkeMo, the model conversion scripts (e.g., MuSkeMo\_to\_OpenSim) will only add DOFs to model if they are named (e.g. hip\_angle\_r). If no coordinates are named for a joint, the joint is turned into an immobilized joint (e.g., WeldJoint in OpenSim).

#### 3.5.1 Fitting primitive objects for joints

In this panel, there are buttons that fit geometric primitives to a selected mesh (sphere, cylinder, ellipsoid, plane). Simply select a target mesh, and press one of the fitting buttons. The options are:

- **Sphere (geometric).** Implements [42].
- **Sphere (least squares).** Implements [19].
- **Cylinder.** Python implementation of the pseudocode in [13]. After implementing the fitting algorithm, MuSkeMo also uses the original vertex data to compute the height of the cylinder, based on the maximum z-span of the target vertices in the cylinder frame. The z-axis Euler angle is set to 0, because cylinders are rotationally symmetric about the long axis.
- **Ellipsoid.** This is a Python implementation provided courtesy of Mark Semple [31], who converted Yuri Petrov's Matlab implementation. I modified it further to ensure right-handed coordinate systems.
- **Plane.** Simple implementation based on taking the covariance matrix of all the recentered vertex positions, which is conceptually identical to a principal component analysis. The best fitting plane must pass through the centroid of the data points [25], which is used to recenter the data. The eigenvectors of the covariance matrix then encode the directions (principal components) of variance in the data [20]. The plane of best fit, in a least-squares sense, is orthogonal to the eigenvector corresponding to the lowest eigenvalue of the covariance matrix [20, 25]. This is taken as the Z-axis of the plane. The plane itself is taken as the basis spanned by the eigenvectors of the two largest eigenvalues. Right-handed coordinate systems are also ensured, by flipping the direction of one of the eigenvectors if necessary.

The functions will fit the entire mesh that you select, so if you want to fit a portion of the mesh (e.g., the femoral head), go through the following steps:

1. Select the target mesh
2. Press Tab to go into 'edit mode' in Blender
3. Select the portion of the mesh you would like to fit (easiest is to use face selection mode, and use the selection circle)
4. press shift D to duplicate the selection
5. press P (for seParate), and then select "by selection".
6. press Tab to go out of edit mode.

You have now copied a subsection of the mesh as a new mesh, which you can use as an input for one of the fitting functions. The duplicated section of the mesh is in the same collection as the original, and if the original mesh was attached to a BODY, so will this copied section. To prevent it being exported, move it to a different collection, and detach the mesh from the body using the button in the panel.

**Matching joints to fitted geometry.** It is possible to match the position and/or orientation of a joint to the fitted geometry, respectively. Simply select the target joint and a source geometry, and press the respective button. Child objects and parent objects are not transformed with the joint. Instead, only the joint's position or orientation is changed, and related data (e.g., pos\_in\_child) are recomputed automatically.

**Cycling through joint axes.** It is possible that the named axes of the fitted object do not correspond with how you would like the joint to behave (e.g., a cylinder's long axis is always the z-axis, but you may want that axis to align with the joint's x-axis). In this case, after using the match orientations button, use the "Cycle through joint axes" button. This button post-multiplies the joint's rotation matrix with the following permutation matrix (see also sec. A):

$$\mathcal{P} = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \quad (6)$$

Cycling through it three times restores the original orientation.

## 3.6 Muscle panel

In this panel, you can define path-point muscles, view their lengths in realtime, analyze moment arms, and assign wrapping. Muscle points are added to the 3D cursor location, and parented to the selected Body (so you have to define bodies before creating muscles). New muscle points are always attached to the selected body, and muscles are placed in the user-designated "Muscles" collection. New muscles points are always added to the 3D cursor location. Press shift + right mouse button to reposition the 3D cursor.

### 3.6.1 Creating a muscle and adding points

To create a new muscle, type a unique (unused) name in the text box, select a body, place the 3D cursor at the desired location, and press "Create new muscle". For each subsequent muscle points, select both the muscle and the target body, reposition the 3D cursor, and press "Add muscle point".

**Important:** MuSkeMo creates muscles using Blender's curves, and automatically hooks each point to the target body (see 4.3). It is impossible to reposition the whole muscle, it can only be repositioned one muscle point at a time. To do this, select the muscle, and press "TAB" on your keyboard to go into EDIT mode. In edit mode, you can select individual points and move them around by hitting G. Once you are satisfied, you can press TAB again to go out of edit mode.

It is possible to insert muscle points (instead of adding them at the end). To insert a point, select the point index after which you would like to insert a point (starting at 1), and press "Insert muscle point".

These instructions are also summarized in a "Muscle tooltips" box which can be hidden from view by deselecting the tick box.

### 3.6.2 Visualization radius

By default, muscles are visualized using a Geometry node modifier named "musc\_name.SimpleMuscleViz" (see section 4.3.1). If this visualization option is used, you can change the visualization radius of all muscles simultaneously by using the "Update visualization radius" button after selecting a desired radius. Alternatively, you can manually tune individual radii of muscles by changing the input in the "SimpleMuscleViz" modifier.

If the volumetric visualization option is used for the muscles, visualization parameters can be tuned in the "VolumetricMuscleViz" modifier (see section 4.3.2 for details).

### 3.6.3 View length in realtime

This button adds a geometry node modifier named "LiveLengthViewer" to the active muscle. This allows the user to interactively pose the model and see how the muscle lengths change. This can be useful during muscle construction, and for tuning musculotendon parameters. The user can change the placement and the size of the text from the modifier inputs. The node uses a custom node group named "LiveLengthViewerNodeGroup", which is shared between all muscles that have a length viewer modifier added to it. To disable the length viewer, remove the modifier from the muscle's modifier stack.

### 3.6.4 Assigning wrapping

You can use the "Wrapping" subpanel to perform all the necessary steps to create and assign wrapping to a muscle. This includes creating wrapping object geometry (currently, only cylinders are supported), assigning parent bodies, and assigning muscles to wrap around an object. The panel also performs the reverse operations. Note that wrapping is currently only physically accurate (i.e., a true tangent solution) if each wrapped section is separated by at least one muscle point on both sides.

Wrapping geometries are also generated using geometry nodes (see 4.9 for details), so that they can be interactively resized using the inputs in the wrapping object's modifier (see 2.2). When a WRAP is assigned to a muscle, it receives a very complex geometry nodes modifier (see 4.3.3 for details). Changing the wrapping geometry dimensions (e.g., the cylinder's radius) also affects the dimensions of the muscle wrapping. Information on the dimensions of the wrapping object is carried over to the Muscle by creating a Named Attribute in the Geometry node for the Wrapping object, which is then sampled in the Geometry node in the muscle wrapping node (see sec. 4.3.3).

### 3.6.5 Moment arms

MuSkeMo can compute a muscle's moment arms, about a single degree of freedom. Specify the muscle name in the top of the Muscle panel, and the "Active joint 1" name in the Moment arms panel. You can choose between rotating about local x, y, or z axes, and the range (in degrees) over which the moment arms should be computed. MuSkeMo assumes that 0 degrees rotation is the current model's position. Angle step size determines how fine or coarse the joint range will be sampled.

When a computation is successful, a Python dictionary named "length\_data" is added to the muscles's custom properties. MuSkeMo computes the length of the target muscle over the desired joint range. Moment arms are computed using the principle of virtual work [34, 2], where  $r = -dL/d\phi$  (see section D.4 in the appendix for more details and a validation of MuSkeMo's moment arms). Only lengths are stored, moment arms are computed during plotting or export. If "Generate plot" is selected, moment arms are plotted straight away (see also 3.6.6), which may be useful when constructing muscle paths. It is also possible to plot muscle lengths instead.

If "Export length and moment arm" is selected, a CSV (or other file, see 3.10) is exported during moment arm computation. This requires setting a target export directory.

**Warning:** While the moment arms computed by MuSkeMo are physically accurate (Fig. 4), they are slightly noisy when using wrapping, in the order of 0.015% of the value of the moment arm, see D.4 for details. Readers intending to perform simulations using the exported moment arms themselves (instead of exporting full models for simulations), may want to smooth the wrapping moment arms before use.

### 3.6.6 Plotting

The plotting subpanel provides the user with some flexibility to alter the plots. Change the parameters in the panel, and then press "(Re)generate a muscle plot", which updates the length or moment arm plot of the muscle in question. The plotter can only (re)generate a plot for muscles where "length\_data" were previously computed in the moment arms subpanel.

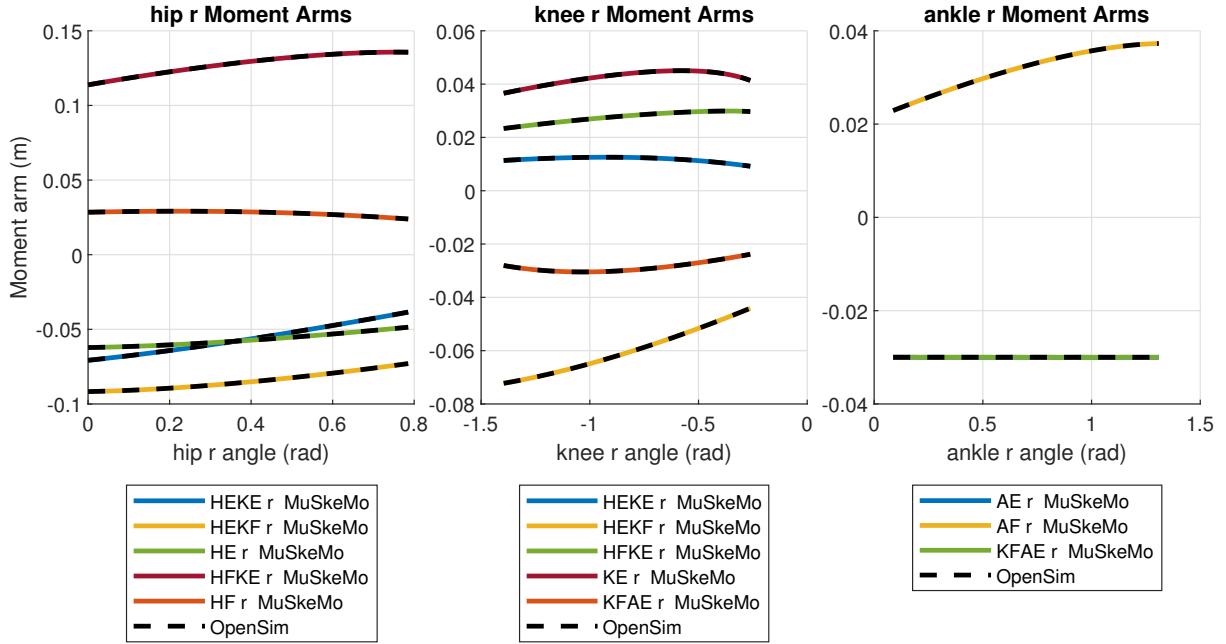


Figure 4: Moment arms of the emu model from [37], computed by MuSkeMo and exported as .csv files, compared to those computed by OpenSim 4.0 using the Matlab API. See section D.4 for details. A python script that performs a moment arm analysis is provided in the Utilities folder 3.14 . Note that this comparison gives identical results because the emu model uses only cylinder wrapping, and that each wrapping cylinder is separated by at least one muscle curve.

### 3.7 Anatomical & local reference frames panel

Construct anatomical and local reference frames (sec. 4.7) by assigning reference points to construct the axes directions, or via manual placement. The user defines a primary axis using two markers, and a temporary axis direction using a third, which together span a plane. The secondary axis is computed as perpendicular to the plane. The tertiary axis is mutually perpendicular to the first two axes, and lies on the same plane as the temporary axis. The reference frame is placed at the user-specified origin position. You have to assign four positions:

- Primary axis start
- Primary axis end
- Temp axis (to define the plane together with the primary axis)
- Frame origin (this can be the same marker as any of the previous three, if desired)

Each of these four positions is assigned by selecting a reference object, and pressing the corresponding "assign" button. The position of those objects will be sampled when constructing the actual frame (using the "Construct Frames from landmark positions" button). It is recommended (but not mandatory) to use Landmarks as reference objects (see sec. 3.8 and 4.5).

All three axes ( $X$ ,  $Y$  and  $Z$ ) can be selected as the primary axis, or the temporary axis to define the plane. It is also possible to place the new frame manually (in which case the frame is placed at the 3D cursor position). The panel thus has seven "Construction modes" which can be selected from a dropdown menu:

- "Specify X-axis and Y-temp direction (for XY plane)".
- "Specify X-axis and Z-temp direction (for XZ plane)".
- "Specify Y-axis and Z-temp direction (for YZ plane)".
- "Specify Y-axis and X-temp direction (for XY plane)".
- "Specify Z-axis and X-temp direction (for XZ plane)".

- "Specify Z-axis and Y-temp direction (for YZ plane)".
- "Manual placement (at 3D cursor position)"

As an example, when selecting the mode "Specify X-axis and Y-temp direction (for XY plane)": The frame will have its *X*-axis aligned to the direction specified by the primary axis start and end markers. The temp axis marker is used to define the *XY*-plane, and the *Z*-axis will be perpendicular to that plane (using a cross-product). The *Y* axis is found using another cross-product between the *Z* and *X* axes. This approach is commonly used to construct anatomical reference frames [39, 40, 41]. To use the Gatesy et al. [18], users will have to combine landmarks and fitted geometric primitives using the shape fitters (sec. 3.5.1).

Local reference frames have a position and an orientation with respect to the global reference frame. If the global frame is denoted with the letter  $\mathcal{G}$ , then the position of an arbitrary frame in the global reference frame can be written as the following vector:  $\vec{v}_{\mathcal{G}}$ . Internally, orientations are stored through rotation matrices. The rotation matrix that rotates a vector from the local  $\mathcal{B}$ -frame to the global  $\mathcal{G}$ -frame can be written as:  ${}^{\mathcal{G}}\mathbf{R}_{\mathcal{B}}$ . The vectors describing the axes directions in the global frame form three mutually orthogonal basis vectors, and they form the three columns of the orientation (rotation) matrix.

Orientations are exported as rotation (unit) quaternions (w, x, y, z), and also as body-fixed (intrinsic, active) XYZ-Euler angles (phi\_x, phi\_y, phi\_z, in rad) (see Appendices A & B). Euler angles are prone to gimbal lock.

Anatomical / local frames can be assigned to a body by selecting one body and one frame, and pressing "Assign parent body". During parent body assignment, MuSkeMo computes inertial properties, joint positions and orientations, contact positions, and muscle path points with respect to these frames. This requires the transpose of matrix  ${}^{\mathcal{G}}\mathbf{R}_{\mathcal{B}}$ , namely:  ${}^{\mathcal{B}}\mathbf{R}_{\mathcal{G}}$ . This rotates a vector from the global  $\mathcal{G}$ -frame to the local  $\mathcal{B}$ -frame.

For an arbitrary point  $p$  expressed in  $\mathcal{G}$ , MuSkeMo computes the transformation to  $\mathcal{B}$  as follows:

$$\vec{p}_{\mathcal{B}} = {}^{\mathcal{B}}\mathbf{R}_{\mathcal{G}} (\vec{p}_{\mathcal{G}} - \vec{v}_{\mathcal{G}}) \quad (7)$$

For an arbitrary matrix  $\mathbf{I}$  expressed in  $\mathcal{G}$  (e.g., an inertial tensor with respect to the body COM), MuSkeMo computes the transformation to  $\mathcal{B}$  as follows [35]:

$$\mathbf{I}_{\mathcal{B}} = {}^{\mathcal{B}}\mathbf{R}_{\mathcal{G}} \mathbf{I}_{\mathcal{G}} {}^{\mathcal{G}}\mathbf{R}_{\mathcal{B}} \quad (8)$$

Readers familiar with linear algebra will recognize this as a similarity transformation, using a change of basis matrix [22].

### 3.8 Landmark & marker panel

Similar to muscle points, landmarks (sec. 4.5) are added to the 3D cursor location. Type in a unique name for the landmark, place the 3D cursor in the desired location (shift + right mouse button), select the target mesh, and press the "create landmark" button. If the target mesh already has a parent body, you can also select the parent body instead.

Landmarks are always parented to bodies (sec. 4.1), not to the meshes themselves. Thus, if your target mesh does not have a parent body, MuSkeMo automatically creates one, and attaches the target mesh to that body as visual geometry (sec. 4.2). MuSkeMo warns the user when this has occurred. The newly created body will have the name "[meshname]\_pbody". When the user attempts to add a landmark to a mesh, MuSkeMo informs the user that the landmark was instead attached to the parent body.

If you are adding landmarks to visual geometry so that you can define local frames in the Anatomical & local reference frames panel (sec. 3.7), it is recommended that you first create bodies with your desired names, and attach those visual geometry to those bodies. Otherwise, MuSkeMo will automatically parent the meshes to newly created bodies, and you will have to detach them later on.

Some people make a distinction between a landmark (projected onto a bone surface) and a marker (any reference position), hence the name of the panel.

## 3.9 Contact panel

Similar to muscle points, contacts are added to the 3D cursor location. Type in a unique name for the landmark, place the 3D cursor in the desired location (shift + right mouse button), and press the "create contact" button. Contacts can also be assigned a parent body, similar to joints.

## 3.10 Export panel

You can export all the user-created datatypes via this panel. The individual exporters export all the data types from the user-designated collections (folders) in Blender. It is possible to export all the visual geometry to a subfolder.

MuSkeMo exports all the data with respect to both the global reference frame (origin), and body-fixed local reference frames. Orientations are exported as XYZ-Euler angle decompositions, and as quaternion decompositions.

Under export options, it is possible to configure other text-based filetypes for export (e.g., txt, bat), configure custom delimiters, and choose the number formatting in the exported files.

## 3.11 Import panel

You can currently import bodies, joints, muscles, frames, and contacts, if they are MuSkeMo-created CSV files.

### 3.11.1 OpenSim model import

MuSkeMo provides an OpenSim importer. This can import most of the components of an OpenSim model. The default behavior is to import models using local definitions, which means that all the local frames are created explicitly during model import. Each body's local 4.7 a distinct object in the scene. If you are unsure which import mode to use, this is the option to select (which is why it is the default).

It is also possible to import models using global definitions - this is only useful for *models created using MuSkeMo's conversion scripts, using the "Global definitions" option*. In essence, this import mode assumes that all the component positions are defined as their position in the global frame, and for joints specifically, transformations in parent and child are both set to the global position and orientation. Local frames are not created. Because this mode assumes a very particular OpenSim model setup, there is less robust support for error-checking of the Geometry folder, and "non-standard" transform axes are not supported (see below).

During import, cylinder wrapping is supported (see 4.3.3 and 4.9). MuSkeMo's wrapping definitions are different from OpenSim, and the importer tries to convert wrapping definitions. If the wrapping does not visualize correctly, it may be necessary to change the projection angle (combined with enabling "force wrap"), selecting "flip wrap", or "shortest wrap", depending on the situation. See 4.3.3 for a full description of MuSkeMo's wrapping.

**Warning:** Blender does not allow *any* object name to be reused. This means that all of your model components need to have unique names in MuSkeMo (so e.g., if a body is named 'shank\_r', and it has only one wrapping object, it still cannot be named 'shank\_r'. It would need a unique name such as 'shank\_wrap\_r'). If any component has a non-unique name, the OpenSim importer may fail. The importer currently checks whether frames and wrapping objects have non-unique names, but not other model components.

MuSkeMo does not support conditional or moving path points. These are automatically converted to regular path points during model import. For moving path points, the point position is selected that corresponds to the position when the controlling joint coordinate is equal to 0.

OpenSim joints have a "socket\_parent\_frame" and "socket\_child\_frame", which normally designates the names of the parent and child frames, respectively. The frames themselves are defined under the "frames" tag (usually PhysicalOffsetFrames), which also have names. It is possible that the "socket\_parent\_frame" or "socket\_child\_frame" does not correspond with either of the PhysicalOffsetFrames defined within the same joint. In this case, MuSkeMo assumes the user intends to use the actual PhysicalOffsetFrames as defined in the frames tag, and ignores the sockets. This is most likely a model error (inconsistent frame definitions), and MuSkeMo warns the user if this occurs during import.

OpenSim models with CustomJoints (and perhaps other joint types with a SpatialTransform) can have Transform Axes that are different from the joint's principal axes, or have a specific user-defined order. In essence, Custom Joints allow users to define custom rotation application order about arbitrary axes, and allows translations about arbitrary axes as well (and the translation and orientation axes do not need to match). Such flexibility is not currently possible in MuSkeMo, R1 is always coordinate\_Rx (about the joint's local (1,0,0) axis), R2 is always coordinate\_Ry, and R3 is always coordinate\_Rz, and Tx, Ty, and Tz are about the same axes as the rotational axes. If a CustomJoint defines a different rotation order than Rx - Ry - Rz, and/or defines any of the transformations about different axes than (1,0,0), (0,1,0), or (0,0,1) (in that specific order), then the transform axes are treated as "non standard". In this situation, MuSkeMo stores the name of the "rotation1" coordinate as the "coordinate\_Rx" (regardless of whether it originally encoded an x-axis rotation), "rotation2" as "coordinate\_Ry", and "rotation3" as "coordinate\_Rz", and the same for the translatory coordinates. The custom transform axes are stored as a Python dictionary in a custom property named "transform\_axes". These transform axes are mapped onto the joint rotations themselves when importing trajectories (see 3.12.2), using axis-angle rotations (see sec. C) in x-y-z order (thus preserving the user-defined custom order that was encoded in the CustomJoint). MuSkeMo warns the user during model import if such transform axes were detected and stored in a joint's custom data. Non-standard axes are also assumed when the root joint's default pose is not aligned with the global reference frame.

OpenSim models require a joint that connects the model to the ground that decides the global degrees of freedom (MuSkeMo calls this a root joint). If a user creates an OpenSim model without defining a root joint, OpenSim automatically creates one (or multiple) FreeJoints, without any frames defined within the joint. If such a model is imported into MuSkeMo using local definitions, MuSkeMo creates these implicit frames at the origin, and guesses the intended OpenSim naming convention.

OpenSim allows the characters ":" in the XML tags of their .osim format, which is not allowed in standard XML files. This is true for "HuntCrossleyForce" components, for instance. MuSkeMo automatically renames these tags during model import, although currently MuSkeMo does not import contact force parameters, so this is purely to prevent an XML import error.

### 3.11.2 Gaitsym 2019 model import

MuSkeMo includes a Gaitsym (2019) importer. It currently imports bodies, joints, muscles (Damped-Spring elements are treated as muscles), contacts, and markers (as frames). Muscles that include wrapping are currently not supported, but limited wrapping support is planned in a future update. Visual geometry can be imported, but requires the user to type the name of the containing folder in "Gaitsym geometry folder". The geometries must be in a subdirectory of the model directory, and the name of this subdirectory must currently be manually typed into the panel.

It is possible to automatically apply a rotation to the entire model during import. This can be convenient because Gaitsym is generally geared towards the Z-axis being the "up" axis, whereas ISB recommends Y-up. To rotate a model from Z-up to Y-up, apply a -90 °rotation about the x-axis. Points simply get rotated, MOI gets transformed according to eq. 8 (although technically the transformation is now from one global frame to another). The same change-of-basis transformation is also applied to orientations (of joints, frames, etc.). The result is that the old Z-axis becomes the new Y-axis, and the old Y-axis becomes the new Z-axis.

### 3.11.3 MuJoCo model import

MuSkeMo provides a MuJoCo importer. It should import most of the components of a MuJoCo model. It has only been tested on MuJoCo models converted from OpenSim using MyoConverter, made available here.

MuJoCo has a substantially different model format from most biomechanical simulators (see the documentation of their model format). Although for the most part it is possible to convert MuJoCo models to MuSkeMo, the conversion will not result in a perfect correspondence in all cases. This is because of the way joints are defined in the MuJoCo kinematic tree. In MuSkeMo, two BODIES (sec. 4.1) have to be connected by a joint, even if the joint has no Coordinates (degrees of freedom). In contrast, bodies in MuJoCo can be directly parented to each other (which welds them together, in essence creating a composite body). When joints are added to a model, they are listed as subelements of the body they act on in the xml, (even though they technically act as a parent joint to that body). MuJoCo also

allows more flexibility in their joint definitions, each MuJoCo joint adds one (or multiple) degrees of freedom, which in principle could each have a distinct position (joint center), and non-perpendicular axis to the other degrees of freedom. Because joints in MuJoCo are more akin to degrees of freedom, they are not equivalent to MuSkeMo joints, but instead are more akin to Coordinates defined within joints in MuSkeMo (see sec. 4.4). Below is a list of conversions that MuSkeMo performs during MuJoCo import:

- MuJoCo bodies are more equivalent to MuSkeMo FRAMES (see sec. 4.7). The actual rigid body information in a MuJoCo body is defined in the (optional) 'inertial' element. During conversion, MuSkeMo treats the 'inertial' element as the body, and creates a frame with the name "frame\_of\_[body\_name]" for each body in the MuJoCo model.
- Joints in MuJoCo define degrees of freedom, and during import are combined into a single MuSkeMo JOINT. The MuJoCo joint names are treated as the Coordinate names in MuSkeMo, and no combined Joint name is actually defined in the MuJoCo model. MuSkeMo automatically names the joint "[parent\_body]\_[child\_body].joint", unless the MuJoCo body has several joints defined, and they have several sections in their name in common. MuSkeMo also creates a joint if a MuJoCo body has no joints defined (because in MuJoCo, this implies two welded bodies, which in MuSkeMo is equivalent to a joint without coordinates).
- MuJoCo in principle allows each "joint" (i.e., degree of freedom) to have its own joint center (defined with the "pos" flag). This is not possible in MuSkeMo because the degrees of freedom are assigned to a single joint. MuSkeMo assumes all the degrees of freedom have the same center, and warns the user if the MuJoCo model defines it differently.
- MuJoCo in principle allows each "joint" (i.e., degree of freedom) to have an "axis" that is not mutually orthogonal to the other joint axes. During import, MuSkeMo checks whether all the axes are positive, principal unit vectors (so [1,0,0], [0,1,0], or [0,0,1]). If not, MuSkeMo creates a Python dict 'transform.axes' in the joint's custom properties that stores the axes as defined in the MuJoCo model. MuSkeMo will warn the user that this has occurred. The python dict can be used during trajectory import (using axis-angle representations) to correctly visualize trajectories (although this is currently not yet supported for MuJoCo models).
- MuJoCo models can have kinematic constraints that couple one coordinate's ("joint" in MuJoCo) motions to another. This is currently not yet possible in MuSkeMo, and the joint is thus imported assuming a coordinate value of 0. The user is warned if this occurred.
- DEPRECATED - WILL REMOVE AFTER TESTING MuJoCo allows joints to have user parameters, which the MyoConverter models appear to use to define a default coordinate value. MuSkeMo creates the model then uses the "user parameters" defined in the joints to transform the joints along their respective coordinate axes.
- Muscles are imported, but if the muscle points (called "Sites" in MuJoCo) were attached to MuJoCo bodies without inertial properties (i.e., frames without inertial properties), then currently the importer parents those muscle points to JOINTS. This is not officially supported by MuSkeMo, so currently such models cannot be exported, but this is adequate for visualizations. In MuJoCo, these points (called sites) can move using joint constraints, which is equivalent to 'MovingPathPoint' in OpenSim. These points were presumably added to the MuJoCo model to represent the wrapping surfaces present in the original OpenSim model that they were converted from.
- Optimal fiber lengths and tendon slack lengths are implicitly defined in MuJoCo, via the actuator length ranges ( $r_{length}$ ), and the normalized operating range ( $r_{norm}$ ) in the gain parameters (see the MuJoCo documentation on muscles). MuSkeMo calculates optimal fiber length  $L_0$  as:

$$L_0 = \frac{r_{length,2} - r_{length,1}}{r_{norm,2} - r_{norm,1}}. \quad (9)$$

Tendon slack length  $L_T$  is calculated as:

$$L_T = r_{length,1} - L_0 \cdot r_{norm,1}. \quad (10)$$

Note that because MuJoCo does not have elastic tendons in their muscle model, to achieve an adequate length range, apparently the optimal fiber lengths in the muscles are sometimes increased

to compensate. For a given  $F_{max}$ , this encodes higher muscle volumes in MuJoCo models. This can become apparent for certain muscles when using the volumetric muscle visualizations (sec. 4.3.2).

The MuJoCo model importer is still missing the following features:

- Conversion of moving path points to static path points.

#### 3.11.4 Other simulators

Future updates will include Hyfydy model support.

### 3.12 Visualization panel

Rendering in Blender can be a complicated process. It is capable of professional level video graphics rendering, and there are a lot of settings that the user can modify to achieve this. This panel provides some ease of use functions that preset the rendering settings that the author finds visually appealing, while also providing adequate performance. There exist thousands of video tutorials for creating renders in Blender. Until a MuSkeMo-specific rendering tutorial is recorded, it is recommended that you follow one of the many out there (e.g., the Donut tutorial referenced at the top of this document). **If your computer does not have a powerful graphics card (GPU), it may be necessary to tweak the recommended settings.**

#### 3.12.1 Volumetric Muscles

This subpanel allows you to toggle between the volumetric (sec. 4.3.2), and the tube style (sec. 4.3.1) muscle visualizations (Fig. 6). Use the **Convert to volumetric muscles** and **Convert to simple (tube) muscles** buttons to toggle between the two. The buttons will act on all muscles in the designated Muscle collection. Specific tension will affect the computed muscle volumes (assuming  $vol = F/\sigma \cdot L_0$ ), and the muscle radius affects the tube style. If you want to change these values, they will take into effect after toggling the visualizations (so e.g., to change specific tension for all muscles, first toggle it back to the tube style, change the specific tension, and then toggle back to volumetric).

For each muscle, the volumetric muscle visualization node has four inputs, next to the volume (see sec. 4.3.2). These can be adjusted for all muscles by selecting **Show volumetric muscle options**. You will have to toggle between the visualization modes for the changes to take into effect.

#### 3.12.2 Trajectory import

MuSkeMo enables you to import simulated trajectories back into Blender to create high-quality animations with complex camera movements. The trajectory importer sets a keyframe for each joint position and orientation, and keyframes the muscle activations by keyframing the color saturation. Frame 0 (in the playback timeline at the bottom of the screen) is always set as the default pose (see sec. 2.4). The actual time-base gets resampled according to the user-selected FPS (see below).

If you want to modify the import settings, you will need to undo the import, and then reimport using the new settings. It is also possible to delete the existing trajectory: navigate to Frame 0, press A to select all model components, then in the timeline below delete all the keyframes.

**OpenSim trajectory files.** MuSkeMo can import trajectories in the ‘.sto’ and ‘.mot’ filetypes.

**Custom trajectory files.** If set to custom, you can import any filetype you like (e.g., csv). The importer will use the column labels to match the trajectory to joint coordinates and muscles in the models. Guidelines:

- Specify a delimiter (including optional whitespaces within the delimiter).
- Specify on what row number the column labels start (starting with 1), all rows before that are skipped.
- Specify whether the data file is in degrees or in radians (default is in radians).
- The ‘time’ column is always the first data column.

- Labels for the joint coordinates should either only contain the coordinate names (see sec. 4.4), or they should contain the coordinate name and then end with ‘/value’. Examples: “pelvis\_ty” or “/jointset/groundPelvis/pelvis\_ty/value”.
- Muscle activations should vary between 0 and 1. Labels can be specified in three ways:
  1. Starts with [muscle name], followed by ‘/activation’. E.g., ‘gastrocnemius\_r/activation’
  2. Starts with [muscle name], followed by ‘.activation’. E.g., ‘gastrocnemius\_r.activation’
  3. Ends with ‘/’ [muscle name], followed by ‘/activation’.  
E.g., ‘/forceset/gastrocnemius\_r/activation’.

**Looping periodic trajectories.** If you are importing a periodic stride, it is possible to automatically loop these in sequence (by increasing the number of repetitions) to create a video using multiple strides, while automatically progressing the (selectable) forward progression coordinate. In this case, you have to define the root joint (MuSkeMo attempts to detect this automatically for OpenSim models, but you may have to set it manually) and the forward progression coordinate (default = coordinate\_Tx).

**Playback speed and FPS.** Using the user-selected frames per second and the trajectory’s timebase, MuSkeMo downsamples the trajectory to individual frames. The frames per second should be chosen during import with the final output speed in mind. For example: if you want to create a video at 30 frames per second, but also want to slow down the playback to 50% real time, then you should set 60 frames per second during import.

**Muscle activations.** The trajectory importer has two settings that affect how muscle activations are animated:

- **Scale activations to highest:** if this is ticked, then all activations in the trajectory will be mapped onto a visual scale that goes from 0 to the max activation (over all the muscles) in the trajectory. This ensures that in trajectories where the activations are low, the relative activation levels are still visible. This is achieved by dividing all the activations by the highest activation in the entire trajectory.
- **Baseline color saturation:** This decides how saturated an inactive muscle will look. The color saturation is computed as follows (after the optional max activations scaling):

$$saturation = baseline\_color\_saturation + (1 - baseline\_color\_saturation) \cdot activation. \quad (11)$$

Thus, if this parameter is set to 0, you use the full bandwidth of color saturation, but inactive muscles will be completely grey. The default value is 0.25, which gives a red-grey tint to the inactive muscles.

**Non-standard transform axes.** OpenSim CustomJoints can have user-defined custom transform axes (see 3.11.1). In the case of translations, these are simply added to the joint position in the directions specified by the transform axes. For rotations, MuSkeMo defines three successive axis-angle Rotation matrices, and applies them in XYZ order (see C).

MuJoCo models have a similar possibility (see Sec. 3.11.3.). MuJoCo trajectory import is not yet supported, but will be handled similarly in the future.

### 3.12.3 Visualization options

This subpanel includes several convenience tools to aid users who are new to animations in Blender. These are:

- **Create a ground plane:** adds a ground plane to the scene
- **Set recommended render settings:** sets the rendering engine to Cycles (see below), the device to GPU, Max samples to 1000, turns on persistent data (under performance), and sets the ”Look” to very high contrast (under Color Management). **WARNING:** the Cycles rendering engine is a path tracing (raytracing) rendering engine. If you do not have access to a computer with a powerful GPU, rendering performance will be incredibly poor. In that case, you should switch to the Eevee rendering engine.

- **Set black background gradient for renders:** creates a node setup in the compositor that de-emphasizes the background.

#### 3.12.4 Default colors

In Blender, objects get assigned a color (and other surface rendering properties, such as roughness) by assigning a material. Before importing and/or model component creation, you can define the desired default colors in this panel for muscles, visual geometry (bones), joints, contacts, markers, geometric primitives, and wrapping geometry. If an instance of the object has been already created (e.g., you have already created a joint), you can change the colors by changing the object's material in the properties tab or in the shader editor.

Importantly, the coloration in Blender depends on whichever Viewport Shading mode is active in Blender. "Solid" mode uses a material's Viewport Display, and during renders this mode is accessible by selecting the "Workbench" renderer. "Material Preview" and "Rendered" modes actually use the object's Material properties themselves, and the "Rendered" view is also what choosing "Eevee" or "Cycles" as the rendering engine. Eevee is faster than Cycles, but looks less cinematic because it is not a raytracing (or path tracing) engine. Both Eevee and Cycles will require setting up some lighting, and it is recommended that this lighting be parented to the camera (effectively acting as a camera flash). Choosing the "Workbench" renderer, combined with "Matcap" lighting, probably requires the least amount of set-up and lowest rendering times.

Joints, contacts, wrap geometry, geometric primitives, and markers are provided with a transparent material by default (for the rendered viewport shading mode, this is achieved with a mix shader node in the shader editor, for the solid viewport display, this is achieved by lowering the alpha).

Unlike all other object types, each individual muscle in the model gets a unique material. This is to ensure that its activation can be independently animated when importing trajectories. This means that if you want a different default muscle color, you need to define it before model import/creation, or you must change all the muscle-materials individually. When muscle activations are animated by the trajectory importer (sec. 3.12.2), the saturation of both the material, and the viewport display are changed simultaneously.

### 3.13 Reflection panel

Bodies, frames, joints, contacts, muscles, and wrapping geometry have a robust reflection approach.

The reflection panel enables symmetric model construction, by only constructing model components on one side, and then reflecting them across the mid-sagittal plane. You can choose what plane defines the midline (default = 'XY'), and change the strings with which you designate the left and right sides (default is '\_l' and '\_r', respectively, at the end of an object's name, e.g., "thigh\_r"). The script then checks for all components whether its other-side counterpart exists, and if not, creates it. The script checks this for both left and right-sided components simultaneously. Each component's reflection function searches within the collection that is designated next to the button.

**Warning:** All the reflection functions use a case-sensitive string search to check whether the side string is at the end of object's name. Because this can potentially cause conflicts (e.g., the name "dorsal\_rib\_l" contains both '\_l' and '\_r'), MuSkeMo only looks for the side string at the end of the object name. Because it is impossible to account for all naming conventions, a workaround can be to place the target component in its own collection and temporarily changing the name(s). In particular with muscles, where each point is parented to a body with a HOOK modifier (see 4.3), conflicts may need to be resolved manually.

Muscle point positions are reflected by multiplying the relevant component by -1, depending on which reflection plane is chosen. All other positional data are reflected by defining reflection matrix (a unit matrix with one diagonal equal to -1). Orientations are reflected by enacting a change of basis with the reflection matrix, using equation 8.

### 3.14 MuSkeMo utilities

The MuSkeMo.zip release contains a folder named MuSkeMo utilities, which includes several useful functions and scripts.

### **3.14.1 OpenSim conversion**

The Matlab script "MuSkeMo\_to\_OpenSim.m" converts your MuSkeMo outputs to an OpenSim model. You must have the OpenSim Matlab API installed, and "CreateOpenSimModelFunc.m" must be in the same directory. The script provides a graphical user interface that lets you select MuSkeMo-created csv files of the model components. It is possible to create a model using global model definitions (optional local frames are ignored, and position/orientation in parent and child are both set to the global position and orientation). It also possible to construct a model using local model definitions. This requires the user to assign a `local_frame` to each body.

### **3.14.2 Fitting muscle lines of action**

It can be useful to fit a curve to a 3D mesh of a muscle, for instance, acquired via DICE-CT scanning or surface scanning. "MuscleLineOfActionFitter.py" is a rudimentary fitting tool. It is not currently built into MuSkeMo yet, but the script can be opened in a Blender scene via the script editor. The user must fill in the target muscle name in the script, and ensure that two objects named 'origin' and 'insertion' are present in the scene. You can set the desired resolution.

The fitter works by slicing up the mesh into n-sections, whose heights are determined by the user-input resolution, and the origin-insertion distance. The slices are created by performing a boolean intersection between the target mesh, and a cuboid that progresses in n-steps from the origin position to the insertion position, aligned in this direction. The volumetric centroid is computed for each slice, and these centroids combined with the origin and insertion form the fitted curve.

High resolution results in slices with a smaller height and thus a smoother curve, with very high resolutions approximating thin cross-sections instead of volumetric slices. High resolutions can potentially result in clipping (ignoring) bits of the muscle during the fitting procedure: if due to the shape, sections of the muscle are proximal to the origin, or distal to the insertion, they are not included in the boolean intersection, and their volumes thus not represented in the line of action. It is up to the user to account for this, and this is why the default behavior is to display both the slice-cuboids, and the resultant slices, which can be compared to the input muscle.

Lower resolution gives a less smooth result, but in most cases will include the entire muscle during its computation.

The resultant objects are Blender curves. It is possible to resample these, if desired. By selecting a curve and right-clicking, it is possible to convert the curve to a mesh. This makes it possible to snap to the fitted curve when creating a MuSkeMo muscle based on the fitted curve.

### **3.14.3 Compute closest point between objects**

The Python script "ComputeClosestPointBetweenMeshes.py" can be run in the Blender script editor. It outputs the closest point between two meshes (in meters). This can be useful to compute minimal joint spacing while articulating a skeleton, or for instance in a series of XROMM frames. An example of how to extend it to work with multiple objects, loop through frames, and export distances as a CSV can be seen in "ComputeClosestPointRealExample.py". This script was written for Voeten et al. (in prep). If that paper has come out by when using this script, please consider citing it as the source.

### **3.14.4 Muscle length script**

Run this script in the Blender script editor (after filling in a target muscle) to get its current length. This is mainly provided because this can be used in more elaborate scripts and functions (e.g., for computing moment arms).

### **3.14.5 Joint repositioning script**

Run "MoveNeckJointsDistally.py" in the Blender script editor to programmatically move all the neck joints in the model by a specified "stretch factor". Intended as an example for programmatic model modifications.

### 3.14.6 Moment arms analysis

The Python script "moment\_arm\_analysis.py" can be opened in the Blender Python script editor to perform a moment arms analysis on the model in the scene. The script is set up to perform the analysis plotted in Figs. 4 & 10, and thus assumes the emu model from Sample Dataset 1 is imported into the scene. It exports the analysis to a subfolder called "muscle\_analysis" (which you can configure). The folder also contains a Matlab script that you can use to plot the outputs. The scripts can be used as a starting point for analyzing moment arms of your own models.

The moment arms analysis script loops through the target joints and finds all muscles that cross it (by checking which muscles change in length when moving the joint through its coordinates).

A second script called "moment\_arm\_hyperplane.py". This rotates the right hip joint and computes the moment arms over a combination of Rx and Rz angles, giving the result presented in sec. D.4 and fig. 11.

### 3.14.7 Range of motion and pose sampling scripts

The utilities folder contains two types of pose sampling scripts, making use of MuSkeMo and Blender's Python APIs in a variety of ways. Importantly, they both use MuSkeMo's intersection checker (see sec. 3.2.2), which is extremely fast, allowing up to 50-200 pose checks per second depending on the complexity of the scene (see below).

The pose sampling scripts are deliberately provided as scripts so that they remain extensible, and are intended as example scripts for your customised analysis. The first two scripts are designed to work with the Blender scene from Sample Dataset 2, which should give the outputs presented in Fig. 5. This is a modified version of the emu model from [37]. The third script is designed to work with VALIDATION SCENE GITHUB LINK. This Blender scene contains the meshes designed for [5], originally available from the associated data repository. See sec. D.5 for details.

The scripts are setup to output a .CSV file after completion, which gives the results of the analysis.

- **NeckJointRomExample.py** This script keeps rotating a joint about a specified axis until a bony intersection would occur, then moves on to the next most distal joint. It is specifically designed with tails or necks in mind (Fig. 5A), see the recently published [12] as an example. You can specify the axis of rotation. This script gives the ultimate range of motion about a specific anatomical axis (so it gives the angle before the intersection occurs). Note that because it works using skeletal intersections, if two successive geometries already intersect in the starting position, it will immediately skip to the next joint in the sequence. In its current version, it checks for intersections with all the proximal geometries one at a time. A performance boost could likely be achieved by using Blender's Dependency graph to combine all proximal geometries as a depsgraph evaluated object copy. This has not been implemented because the intersection checker is already very fast.
- **PoseSampleExample.py** This is a more elaborate pose sampling analysis, in the style of [24] (Fig. 5B). The idea is that instead of prescribing a specific axis of motion, you densely sample all available poses of a joint, and check whether they are viable. This script allows you to distinguish between skeletally non-viable (i.e., do the skeletal meshes intersect?) and non-viable due to soft-tissues (in this script: if a ligament length exceeds a specific threshold). To visualize the pose space, a landmark 4.5 is placed on a user-specified location on the distal segment, and in each pose, the position of this landmark is duplicated. These endpoint markers are color-coded based on viability (red: skeletally non-viable, orange: non-viable due to soft tissue constraint, blue: viable). MuSkeMo's reliance on Blender's implementation of bounding volume hierarchies ([15], see sec. 3.2.2) makes it computationally very efficient. The default settings test 1701 poses in 34 seconds (0.02 s per pose, or 50 poses per second) on an Intel i7-11850H 2.5 GHz Laptop processor. The script only checks rotational poses, see the next scripted example for a 6 degree of freedom pose analysis.
- **6DOFPoseSampleTest.py** This script performs the same analysis presented in D.5 at a lower sampling density. It works with VALIDATION SCENE, which uses the screw-motion pose sampling test developed for [5]. This script is intended to demonstrate how you might script up a 6DOF pose sampling analysis. The default settings of this script only find one viable pose (the starting pose), because the test is designed to require a high sampling density (see [5] and sec. D.5). The poses

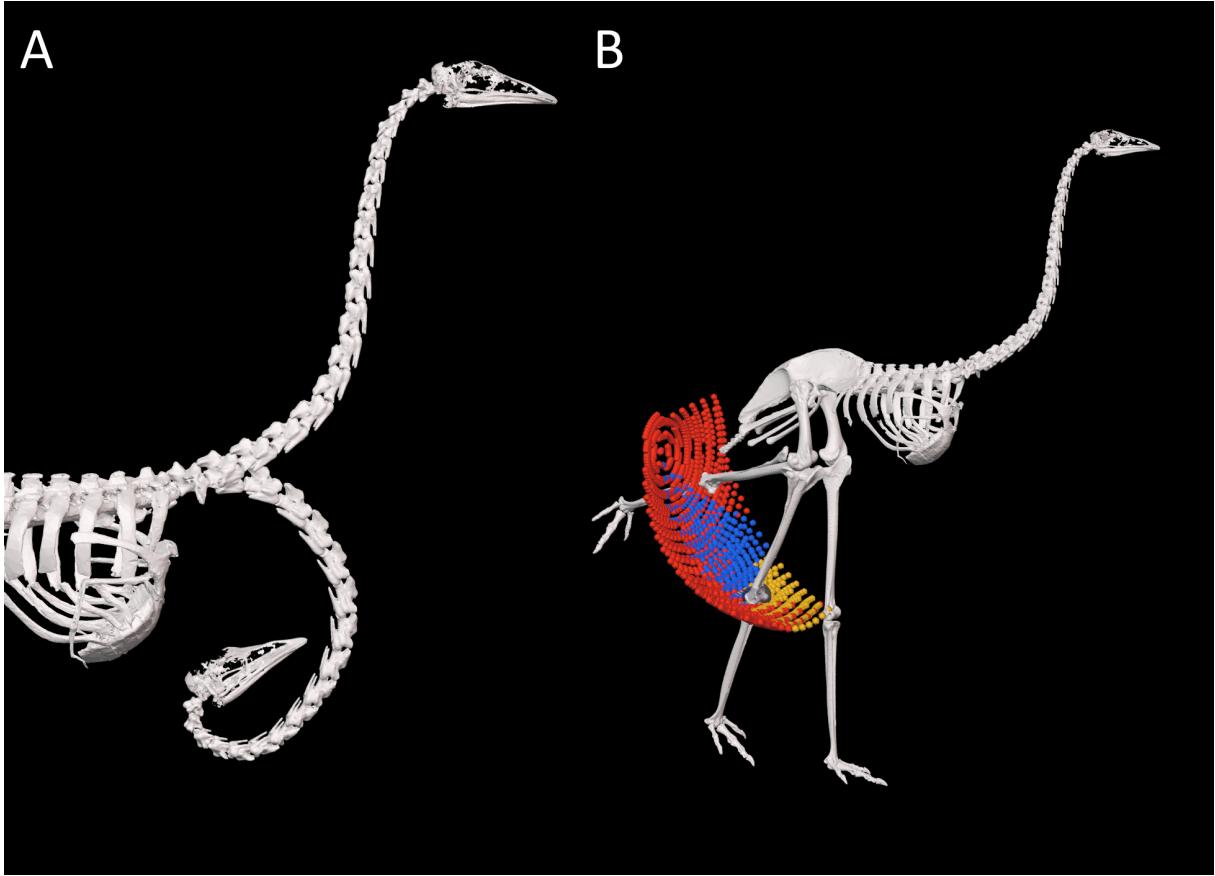


Figure 5: **Outputs of two joint range of motion analysis example scripts.** A. Ultimate neck flexion range of motion. The output is intended as a demonstration only. B. Knee joint pose sampling analysis. A LANDMARK is placed on the distal tibiotarsus. As the knee joint is moved into different poses, the script checks for intersections between the femur and the tibiotarsus, and checks if the length of the cruciate ligament exceeds a threshold. Based on this, the marker is duplicated in this position, and color-coded: red (skeletally non-viable), orange (non-viable due to soft tissue constraint), or blue (viable)).

are sampled at a rate of 200 poses per second, due to the relative simplicity of the meshes with respect to the emu model. If you want to perform the full test, set `sample_density` to a higher value (e.g., 4). This script does not use a soft-tissue constraint, but it is set up to allow it should you want to use one. You can also turn the endpoint visualizations on or off, and optionally suppress all the print statements, and choose to only keyframe viable poses.

The scripts are demonstrated using the Blender scene from Sample Dataset 2. This is a modified version of the emu model from [37]. The modified version of the model contains a cranial cruciate ligament, following [16], implemented in MuSkeMo as a MUSCLE so that its length can be used as a cutoff. The neck is also rearticulated slightly so that the neck geometries no longer intersect in the default pose. This version of the model is intended as a kinematic demonstration model only, so the rigid body parameters have been set to NAN.

### 3.14.8 Volumetric muscle visualization volume comparison

The Python script “`CompareVolMuscVizVolumes.py`” compares volumes of the Fast and VolumeAccurate modes of the volumetric muscle visualization node (sec. 4.3.2). This script should be executed in Blender’s script editor after loading a MuSkeMo model and enabling the volumetric muscle visualizations. It assumes that muscle object names end with `_r` (right side) or `_l` (left side), though this can be modified to suit other naming conventions.

For all muscles, the script sets the `TendonMuscleRadiusRatio` parameter to zero, disabling tendon geometry. Right-sided muscles are processed using the “Fast” volume mode, while left-sided muscles are processed using the “VolumeAccurate” mode. In each case, the actual mesh volume is computed using

Blender's `bmesh` API and compared to the Hill-type muscle volume encoded in the muscle's geometry node group.

The results are:

- Printed to the System Console (*Window, Toggle System Console*).
- Saved as a CSV file in the same directory as the current blend file.

For the emu model in the sample dataset (in the base posture), the Fast mode underestimated actual volume by between 0% and 3.6%, with an average underestimation of 2.5%. This discrepancy varies with posture and muscle visualization settings, and can be larger (e.g., up to 8%) for certain configurations. The VolumeAccurate mode provides a closer match but runs twice as slowly (on an Intel i7-11850H 2.5 GHz laptop processor).

The script requires minimal modification for other models, aside from adapting side-string detection if muscle naming conventions differ.

## 4 MuSkeMo Data Types

### 4.1 BODY

A rigid body. Rigid bodies have inertial properties, which can be computed during model creation from 3D scans:

- `mass` (kg)
- `COM` (center of mass (m) in the global reference frame)
- `inertia_COM` (moment of Inertia (kg m<sup>2</sup>) about the COM, in the global reference frame)

Inertial tensor elements are listed in the following order: (Ixx, Iyy, Izz, Ixy, Ixz, Iyz).

Bodies can have one or more optional attached `Geometry` meshes for visualization (e.g., 3D meshes of bones). These are delimited with a ';' if present, and usually preceded with the name of the subdirectory (default = '`Geometry`') in which the meshes will be exported. For example:

```
'Geometry/cranium.obj;Geometry/mandible.obj;'
```

COM is always reported in the global reference frame, and MOI is always computed with respect to the COM in the global reference frame and orientation. It is possible to assign a `local_frame` to a body (see section 4.7). This automatically computes the following properties:

- `COM_local` (center of mass (m) in the body's local reference frame)
- `inertia_COM_local` (moment of Inertia (kg m<sup>2</sup>) about the COM, in the body's local reference frame)

The underlying object type in Blender is an "Empty".

### 4.2 GEOMETRY

Visual geometry (e.g., bone meshes, or tissue outline meshes) can be attached to bodies for visualization purposes. In most simulators, these don't have a physical function, but are purely used for visualization purposes and for determining where muscle attachments are relative to the bodies.

**WARNING:** OpenSim has a visualization bug that prevents models from loading correctly if the total file size of the attached geometry exceeds 50MB. You can decimate the meshes to reduce them to about 50MB or attempt to load the model into OpenSim Creator.

If Decimating the model doesn't work, the visualization error can sometimes also be triggered by meshes that are composites of several different parts, or that have many loose triangles. Using MuSkeMo, first detach the visual geometry via the Geometry panel. Then, select the mesh and press TAB for edit mode, and then press P (for seParate). This separates the model by loose parts. The resulting meshes should be parented to the body individually. This will require you to re-export the geometries, re-export the bodies CSV, and regenerate the OpenSim model. However, it is usually the total filesize that triggers this issue, not composite meshes.

## 4.3 MUSCLE

A path-point muscle. Each muscle point is automatically parented to a body. Muscles have the following user-definable contractile properties:

- `F_max` (maximal contractile force of the muscle fibers, in Newtons)
- `optimal_fiber_length` (in meters)
- `pennation_angle` (in degrees)
- `tendon_slack_length` (in meters)

The underlying object type in Blender is a poly-curve. Each point in the curve is attached to a body using a hook-modifier.

### 4.3.1 Simple muscle visualization

Simple tube style muscle visualizations (Fig. 6) are achieved by adding a simple Geometry node setup to each muscle (curve in Blender). This node setup essentially lofts a curve with the user-specified radius (default = 0.015 m) across the entire length of the curve. The radius can be changed without going into the Geometry nodes setup, by selecting the correct modifier in the modifier stack.

Under "Geometry nodes", with the "SimpleMuscleViz" node of a muscle selected, the node setup is visible. The visualization setup itself is specified by the "SimpleMuscleNode" node group (select it and press 'TAB' to modify, this node group is shared by all muscles and thus modifications are applied to all muscles). The node setup also applies a material to each individual muscle, so that their colors can be animated individually.

### 4.3.2 Volumetric muscle visualization

You can convert all your muscles to the volumetric style in the Volumetric Muscles subpanel of the Visualization panel (sec. 3.12.1). Similar to the simple muscle visualizations, volumetric muscle visualizations are also achieved by adding a Geometry node to each muscle. The node-group itself is shared across muscles, but muscle visualizations are individualized through four parameters and an operation mode, which can be accessed directly in the "VolumetricMuscleViz" modifier (Fig. 6). These are:

- **MuscleVolume:** Computed using each muscle's `F_max`, `optimal_fiber_length`, and `specific_tension` (set in the Visualization panel, see section 3.12). The node setup adjusts the muscle's radius to keep the volume constant irrespective of the length.
- **MuscleTendonLengthRatio:** This decides the ratio of the muscle belly to the total musculo-tendon complex length. If set to 1, the muscle belly is stretched across the entire muscle's length.
- **TendonMuscleRadiusRatio:** This sets the relative ratio of the tendon with respect to the muscle's radius. Set to 0 if you want no tendon visualization.
- **ProxToDistMuscleBellyBias:** By default, the muscle belly is in the middle of the curve. If `MuscleTendonLengthRatio` is less than 1, you can shift the muscle belly more proximally or more distally using "ProxToDistMuscleBellyBias".
- **FastOrVolumeAccurate:** Toggle between a faster or more volumetrically precise visualization. See sec. 3.14.8 for a script that compares muscle volumes. The Fast mode may underestimate the volume by 1-5%, depending on the shape of the muscle. The VolumeAccurate node adds an extra volume checking and rescaling step to ensure that the visualized volume is correct with respect to the muscle-input volume. This mode is approximately twice as slow, because it essentially draws each muscle twice (once internally). In most cases, the discrepancy between the two modes is imperceptible. This is why the Fast mode is enabled by default, given that it is approximately twice as fast which may be noticeable on slow PC's when visualizing many muscles. Use the VolumeAccurate mode if high precision is required.

If you want to change the global defaults of these parameters, hit "Show volumetric muscle options" in the Volumetric Muscles subpanel (sec. 3.12.1), and toggle between the visualization modes for the changes to take into effect.

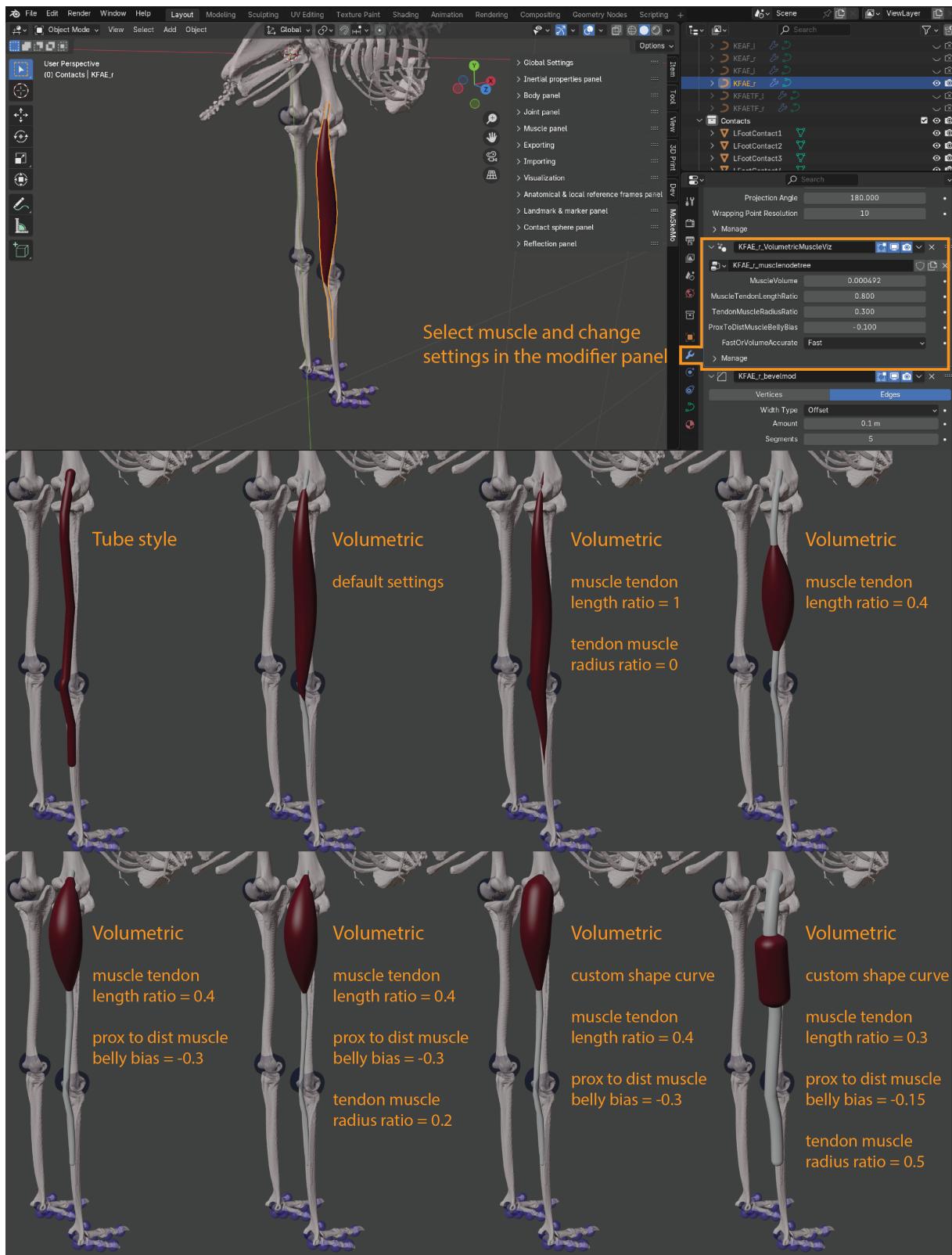


Figure 6: MuSkeMo’s tube and volumetric muscle visualization styles. The volumetric muscle visualization can be interactively changed using the modifier input, when the muscle is selected. How to adjust the shape curve is shown in Fig. 7.

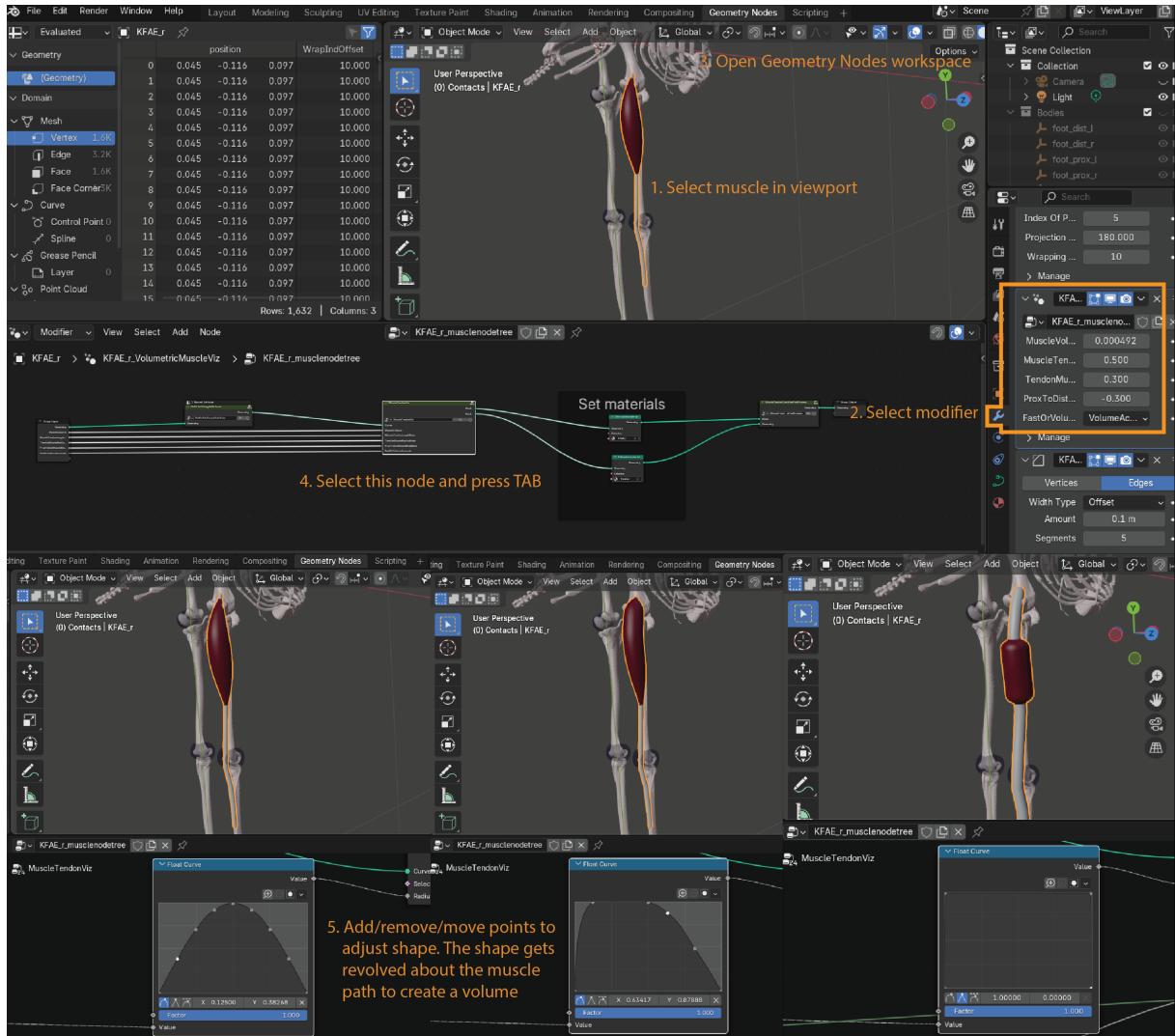


Figure 7: You can customize the shape curve of the volumetric muscle visualizer by going into Geometry Nodes, selecting the correct node, pressing “TAB”, and then adjusting the shape of the curve interactively.

**How it works:** The volumetric muscles are created using a Geometry nodes setup that sweeps (lofts) a circle across a portion of the length of the muscle curve, creating a volumetric object. A fixed fraction of the the whole muscle curve’s instantaneous length is assigned as the Muscle belly length, meaning the belly length changes with different postures. The radius of the swept circle is dynamically adjusted to maintain constant volume. Simply sweeping a circle across a curve would result in a cylinder, therefore the node setup further varies the radius of the swept circle to allow for a tapered muscle shape. The swept circle is also not a true circle, but an  $n$ -sided polygon (24 by default), which looks like a circle due to smooth shading in Blender. The node group corrects for the extra reduction in volume that the tapered shape and  $n$ -gon approximation introduce, but only the “VolumeAccurate” mode further corrects for the volume discrepancy introduced by not having a straight-line curve.

The full node setup performs the following steps:

1. The input polycurve is resampled for even spacing.
2. One copy of the curve becomes the *tendon*, with a constant-radius profile set by the tendon radius ratio.
3. The other copy of the curve is trimmed according to the length ratio and proximal-distal bias, then resampled again for smoothness
4. For both modes, an initial muscle belly radius is calculated from the target muscle volume and the trimmed belly length, using a cylindric approximation.

5. The computed muscle belly radius is then scaled by the curve's *Curve Radius* attribute, which can be edited using the Float Curve node, allowing you to adjust the muscle's shape (Fig. 7). In Blender, the *Curve Radius* is a per-control-point scalar property of a curve (Set Curve Radius node), which does not directly change the visual thickness of a curve in the viewport, but is used by operations such as Curve to Mesh to scale the cross-sectional profile. Thus, the Float curve node allows the user to draw a thickness profile of the muscle along its entire length (i.e., giving fine grained control over the thickness and the taper of the muscle). Note: Before version 4.5, the scale was set implicitly if the "Set curve radius" node was used before it in the node chain. Blender 4.5 added the scale as a dedicated input to the Curve to Mesh node, and automatically adds generates a "Named attribute" and "switch" node to connect to it, and to check whether "Set curve radius" was used earlier in the chain.
6. Because tapering reduces volume, the node setup calculates the theoretical swept-volume of circle profile along a straight line, with its radius equal to the initial muscle belly radius scaled by the relative height of the profile curve at any point across the muscles length. This computation accounts for the truncation error from sweeping an  $n$ -sided polygon instead of a true circle. This theoretical volume is compared to the target cylinder volume, and the initial muscle belly radius is scaled by the square root of the ratio to match the target.
7. The adjusted  $n$ -gon profile is swept along the curve to create the muscle mesh.
8. In **Fast mode**, this mesh is used directly.
9. In **Volume Accurate mode**, the swept mesh is subdivided, its volume is measured, and a second adjustment is applied to precisely match the target volume before generating the final mesh. Because the muscle is essentially drawn twice (once without visualizing it), this mode is approximately twice as slow. The within-node volume computation occurs with the "calc\_volume" node shared by user bebop.artist on the Blender Stackexchange forum, under a CC BY-SA 4.0 license.
10. A subdivision surface is applied to smooth the appearance and reduce visible creases.
11. Muscle and tendon receive separate materials (see sec. 3.12.4)

#### 4.3.3 Muscle wrapping

MuSkeMo implements [17] for cylindric wrapping, using a custom-designed Geometry node. Garner et al. [17] propose an iterative root finding method for multi-object wrapping between two curve points, which is currently very challenging to implement using Geometry nodes. **If you add multiple wrapping objects per pair of muscle points, MuSkeMo only provides true tangent curve solutions if each wrapped section is separated by a curve point.** MuSkeMo adds a Geometry nodes modifier to the muscle for each wrapping object.

**Wrapping will not work correctly if any of the muscle curve points clip into the wrapping object.**

Cylindric wrapping has the following settings, accessible via the wrapping modifiers:

- **Wrap Cylinder Radius:** Radius of the cylinder. When assigned to a muscle, also affects the wrapping radius by storing a Named Attribute in Blender.
- **Wrap Cylinder Height:** Height of the cylinder. When assigned to a muscle, also assigns the wrapping height by storing a Named Attribute in Blender.
- **Flip Wrap:** This allows you to flip the sidedness of the wrap manually.
- **Shortest Wrap:** This ensures the wrapping occurs on the side where the wrapped section would be shortest. Causes posture-dependent flipping if the curve moves to the other side of the wrapping geometry. Shortest wrap is ignored if Force Sided Wrap is enabled.
- **Force Sided Wrap:** Select this if you want to force a wrap in a specific projection direction, even when the curve is not intersecting the geometry. You have to specify the index of the pre-wrap point, and a projection angle. The wrap is active when the curve intersects the geometry, or when it is outside the geometry, on the opposite side of the projection direction. See below how this is achieved.

- **Index Of Pre Wrap Point Starting At 1:** Specify the point after which you would like the wrapping to occur. This setting is ignored unless you select Force Sided Wrap.
- **Projection Angle:** By default, the projection direction is in the positive X-direction in the frame of reference of the wrapping geometry. The projection angle rotates the projection direction about the Z-axis of the wrapping geometry.
- **Wrapping Point Resolution** Number of points projected onto the wrapping geometry (if wrapping occurs). This only affects visualizations, not length computations. Setting this very high can reduce performance.

**Projection Angle.** This can affect the wrap in two situations, and is used in comparison with the positions of the two points that span the wrap ( $\vec{p}_1$  and  $\vec{p}_2$ ).

1. If Force Sided Wrap is on, we find the closest point  $\vec{p}_c$  on the curve section spanned by  $\vec{p}_1$  and  $\vec{p}_2$ , from wrap object center  $\vec{c}_w$ , using:

$$\vec{p}_c = \frac{(\vec{c}_w - \vec{p}_2) \cdot (\vec{p}_1 - \vec{p}_2)}{\|\vec{p}_1 - \vec{p}_2\|^2} \cdot (\vec{p}_1 - \vec{p}_2) + \vec{p}_2 \quad (12)$$

Readers familiar with linear algebra will recognize this as an orthogonal projection [22]. An extra check is applied to account for situations where  $\vec{p}_1$  or  $\vec{p}_2$  is the closest point. All of this is computed in the wrap object's local frame. If the dot product between the projection direction vector and  $\vec{p}_c$  is negative, Force Sided Wrap ensures that the wrap is active. The above computation will work regardless of whether the curve is in contact with the wrap or not. This enables wrapping in situations where both points are on the other side of the wrap than the projection direction, without the curve being in contact with the wrap.

2. The vector  $\vec{p}_2 - \vec{p}_1$  points from the pre to the post wrapping point. If this vector points towards the first or second quadrant (using the projection-direction as the x-axis), the wrapping direction is flipped. This is achieved by checking whether  $\text{atan2}(\vec{p}_2 - \vec{p}_1) - \text{projectionangle}$  is between 0 and  $\pi$ .

**Pre wrap index.** Inside the node tree, the conditional wrapping is enforced in the node by passing -2 as an index if wrapping should not be active, and the actual pre-wrap index if it should be active. Pre-wrap index is either decided by a wrap intersection, or if there is no intersection, by checking if force wrap is enabled and the muscle is on the wrapping side.

**Shortest wrap.** First determines the vector  $-\vec{p}_1 \times \vec{p}_2$ , then computes the dot product between that vector and the cylinder z-axis. If this dot product is negative, the wrap is flipped. Shortest wrap is superceded by Force Sided Wrap.

## 4.4 JOINT

A joint in MuSkeMo represents the connection between two rigid bodies, allowing them to articulate relative to each other. The joint position and orientation can be expressed in XYZ Euler angles or quaternions.

`parent_body` and `child_body`: these are the two bodies connected by the joint. These must be defined by the user. When defined by the user, `pos_in_global` and `or_in_global` are both computed.

- `pos_in_global`: The position of the joint center in the global reference system (in meters).
- `or_in_global_XYZeuler`: The orientation of the joint in the global reference system (in radians) using an XYZ Euler decomposition.
- `or_in_global_quat`: The orientation of the joint in the global reference system, expressed as a quaternion (w, x, y, z).

These define the joint's position and orientation in the global reference frame. The position is given as a list of three floats (x, y, z) in meters. The orientation is expressed in both XYZ Euler angles (radians) and as a quaternion (w, x, y, z). If a joint has neither parent nor child body, the above are all set to nan, but exported correctly if the user exports Joint Data.

If a **parent** or **child** body also has a local FRAME defined, MuSkeMo automatically computes the transformations with respect to that frame, resulting in six more parameters:

- **pos\_in\_parent\_frame**: The position of the joint center in the local reference frame attached to the parent body (in meters).
- **or\_in\_parent\_frame\_XYZeuler**: The orientation of the joint in the local reference frame attached to the parent body (in radians) using an XYZ Euler decomposition.
- **or\_in\_parent\_frame\_quat**: The orientation of the joint center in the local reference frame attached to the parent body , expressed as a quaternion (w, x, y, z).
- **pos\_in\_child\_frame**: The position of the joint center in the local reference frame attached to the child body (in meters).
- **or\_in\_child\_frame\_XYZeuler**: The orientation of the joint in the local reference frame attached to the child body (in radians) using an XYZ Euler decomposition.
- **or\_in\_child\_frame\_quat**: The orientation of the joint center in the local reference frame attached to the child body , expressed as a quaternion (w, x, y, z).

Each joint also has six coordinates which can be named. These are meant to represent the generalized coordinates, or degrees of freedom, of the model. If they are named, this coordinate becomes a degree of freedom when using one of the provided scripts to convert a MuSkeMo model to a simulator (e.g., if you want **coordinate\_Rz** to be a degree of freedom in your OpenSim model, give **coordinate\_Rz** a name such as ‘hip\_flexion\_r’).

- **coordinate\_Tx**: Translation along the x-axis
- **coordinate\_Ty**: Translation along the y-axis
- **coordinate\_Tz**: Translation along the z-axis
- **coordinate\_Rx**: Rotation along the x-axis
- **coordinate\_Ry**: Rotation along the y-axis
- **coordinate\_Rz**: Rotation along the z-axis

The underlying object type in Blender is a UV sphere mesh. This object is essentially only for visualization purposes, but a future version of MuSkeMo may add the option to add axes as well.

## 4.5 LANDMARK

A **LANDMARK** in MuSkeMo refers to any user-specified point in 3D space, and can be used to help define (anatomical or local) reference frames (sec. 4.7). Landmarks are always parented to bodies (sec. 4.1), even when selecting a mesh to add landmarks to (see sec. 3.8 for details on how this is handled). Thus, if you are adding landmarks to a visual geometry so that you can define local reference frames, it is best to first attach the visual geometry to the body in question, because otherwise the Landmark panel will automatically create a body and attach the geometry to it.

The underlying Blender object type is a UV sphere mesh.

## 4.6 CONTACT

Contact geometry defines areas or regions where external forces might act on a rigid body. These points represent places where a body or object interacts with another object or the environment during simulations. MuSkeMo does not compute any contact forces, but the user can define contact positions if the model will be used for simulations. MuSkeMo currently only supports contact spheres. Contact the developer if you need more geometry types. Like joints, contact spheres can have a **pos\_in\_global** and a **pos\_in\_parent\_frame**.

The underlying object type in Blender is a UV sphere mesh.

## 4.7 FRAME

In MuSkeMo, a **FRAME** is a local coordinate system that can be assigned to any rigid body. It defines a reference position and orientation relative to which other properties of the body (such as the center of mass or moment of inertia) can be computed. Each rigid body can be assigned one (optional) local **FRAME**, which can represent an anatomical reference frame. If a local frame is assigned to a body, body segment parameters with respect to that frame are automatically computed, as are the local transformations of any parent and/or child joints. These are removed if the frame is detached from the body.

## 4.8 GEOM\_PRIMITIVE

In MuSkeMo, it is possible to fit geometric primitive shapes (spheres, cylinders, ellipsoids, and planes) to 3D meshes, see 3.5.1. This can be useful for defining joint centers of rotation using the skeletal geometry. The geometric details of the shapes are stored for reuse. These are:

- Sphere: `sphere_radius`
- Cylinder: `cylinder_height`
- Ellipsoid: `ellipsoid_radii` (x, y, and z components)
- Plane: `plane_dimensions` (x and y components)

The underlying object types in Blender are regular meshes (UV Sphere, cylinder, ellipsoid).

## 4.9 WRAP

Wrapping geometry can be user-created in the Wrapping subpanel of the Muscle panel. Wrapping geometry defined in OpenSim models (but currently not yet Gaitsym models) are included during model import. WRAP objects are not regular meshes like GEOM\_PRIMITIVES. WRAP geometry are generated using Geometry nodes that parametrically generate the desired object. The wrapping geometry dimensions (e.g., cylinder radius) can be altered in the modifier on the object (press the blue wrench in the interface, see sec. 2.2). Currently, only cylinders are supported.

Just like JOINTS, WRAPS have parent bodies, and orientations and positions with respect to parent frames if those are assigned to the body. Unique to WRAP are the following properties:

- `target_muscles`. A list of all the muscles (delimited with ';') that wrap around this object. Can be 'not\_assigned'.
- `wrap_type`. Currently only 'Cylinder'.

# A Euler angles

## A.1 Conventions used by MuSkeMo

Euler angles define rotation matrices that transform between reference frames. Conceptually, they are relatively straightforward: 3D rotations are defined by three successive rotations about different axes in space. However, Euler angles are ambiguous without explicitly defining what convention is being used. It is possible to choose twelve different sets of axes about which we perform the successive rotations (e.g., XYZ, XYX, etc.). It is furthermore possible to define these axes with respect to the rotating body (body-fixed, intrinsic), or the global frame (space-fixed, extrinsic). This determines the order of the application of the individual rotations. The rotations can also be defined to rotate the body itself (active rotations), or rotate the frame around a stationary body (passive rotations). Active and passive rotation matrices are related by being each others' transpose.

To enable cross-compatibility with OpenSim, MuSkeMo uses the same convention as OpenSim. This convention uses a body-fixed (intrinsic) XYZ-decomposition using active rotations (so the object is being rotated, not the reference frame). This is also sometimes referred to as  $X - Y' - Z''$ , to signify that the three successive rotations are about different frames. OpenSim is built on Simbody, and the Simbody Documentation cites page 423 of [21] as their source.

The full rotation matrix  ${}^G\mathbf{R}_B$  that MuSkeMo uses to rotate a vector from a body-fixed frame (the  $\mathcal{B}$ -frame) to the global reference frame (the  $\mathcal{G}$ -frame) is presented here to prevent ambiguity:

$${}^G\mathbf{R}_B = \begin{bmatrix} \cos \phi_y \cos \phi_z & -\cos \phi_y \sin \phi_z & \sin \phi_y \\ \cos \phi_x \sin \phi_z + \cos \phi_z \sin \phi_x \sin \phi_y & \cos \phi_x \cos \phi_z - \sin \phi_x \sin \phi_y \sin \phi_z & -\cos \phi_y \sin \phi_x \\ \sin \phi_x \sin \phi_z - \cos \phi_x \cos \phi_z \sin \phi_y & \cos \phi_z \sin \phi_x + \cos \phi_x \sin \phi_y \sin \phi_z & \cos \phi_x \cos \phi_y \end{bmatrix} \quad (13)$$

Here,  $\phi_x$ ,  $\phi_y$ , and  $\phi_z$  represent rotations about the body-fixed, right-handed  $X$ -,  $Y$ -, and  $Z$ -axes. This is equivalent to successive rotations by the same angles about global  $z$ -,  $y$ -, and  $x$ -axes (note the reversed order of the rotations). Brackets are omitted because each cosine and sine only has one term inside it.

A vector  ${}^B\vec{v}$  expressed in the  $B$ -frame (hence the  $B$ -prefix), can be rotated to the  $G$ -frame by pre-multiplying it with  ${}^G\mathbf{R}_B$ . In other words:

$${}^G\vec{v} = {}^G\mathbf{R}_B {}^B\vec{v} \quad (14)$$

To rotate a vector from the  $G$ -frame to the the  $B$ -frame, we need a rotation matrix that performs the inverse operation:

$${}^B\vec{v} = {}^B\mathbf{R}_G {}^G\vec{v} \quad (15)$$

Rotation matrices are orthogonal (the columns and rows are orthogonal unit vectors). Thus, the inverse or transpose of a rotation matrix performs the inverse of the rotation. This means that you can use both the inverse or the transpose of the rotation matrix to find the inverse operation

$${}^G\mathbf{R}_B^{-1} = {}^G\mathbf{R}_B^T = {}^B\mathbf{R}_G \quad (16)$$

So OpenSim, MuSkeMo, SCONE, etc., all implement Euler angles using an intrinsic (body-fixed), active (rotating the object) convention, in XYZ-order. Blender itself also implements Euler angles, but uses the convention of space-fixed (extrinsic) rotations. **As a result, what Blender refers to as ZYX-Euler angles is the same rotation matrix as the body-fixed XYZ-Euler angles used by MuSkeMo, OpenSim, SCONE, etc.**

## A.2 Understanding decomposed successive rotations

The different Euler-angle conventions in use were the source of much confusion for the author. The confusion comes from three characteristics of Euler-angles. First, there are many ways in which Euler-angles can be used to construct rotation matrices (technically, the XYZ convention is not an Euler-angle convention, but a Tait-Bryan convention — true Euler angles reuse the first axis as the third axis). The second source of confusion can come from the order of matrix application, whereas we write down the successive rotations in letter form (e.g., XYZ) from left to right. The third source of confusion is the unintuitive order in which the rotation matrices must be applied to achieve intrinsic rotations (about body-fixed axes).

A step-by-step explanation is provided in the next sections of this Appendix. For further reading, the reader is referred to textbooks on dynamics (e.g., [21, 35]), and D. Rose’s engineering notes website for a particularly accessible description of the conventions that exist for Euler angles [26].

### Rotation About the global $x$ -axis

We will use the lower case letters to signify rotations about global axes. So for example, the global  $x$ -axis (expressed in the  $G$ -frame). To rotate by  $\phi_x$  about the global  $x$ -axis (this is an **extrinsic** rotation), the rotation matrix  $\mathbf{R}_x$  is given by:

$$\mathbf{R}_x = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos \phi_x & -\sin \phi_x \\ 0 & \sin \phi_x & \cos \phi_x \end{pmatrix} \quad (17)$$

For example, applying this to a unit vector initially aligned with the the global  $y$ -axis direction:

$$\mathbf{R}_x \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \quad (18)$$

If  $\phi_x = \frac{\pi}{4}$ , this results in:

$$\begin{pmatrix} 0 \\ \cos \frac{\pi}{4} \\ \sin \frac{\pi}{4} \end{pmatrix} = \begin{pmatrix} 0 \\ \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{pmatrix} \quad (19)$$

Thus, the second column of  $\mathbf{R}_x$  (the  $y$  column of the matrix), gives the new direction of the new  $y$ -axis after rotation. This is true for all the columns of a rotation matrix: the columns give the direction vectors for the  $x$ ,  $y$ , and  $z$ -axes of the rotated object, expressed in the original frame (in this case, the global frame).

### Rotation About the global $y$ -axis

To rotate by  $\phi_y$  about the global  $y$ -axis, the rotation matrix  $\mathbf{R}_y$  is:

$$\mathbf{R}_y = \begin{pmatrix} \cos \phi_y & 0 & \sin \phi_y \\ 0 & 1 & 0 \\ -\sin \phi_y & 0 & \cos \phi_y \end{pmatrix} \quad (20)$$

Similarly, applying this to a unit vector initially pointing in the  $x$ -axis direction:

$$\mathbf{R}_y \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} \quad (21)$$

If  $\phi_y = \frac{\pi}{4}$ , this results in:

$$\begin{pmatrix} \cos \frac{\pi}{4} \\ 0 \\ -\sin \frac{\pi}{4} \end{pmatrix} = \begin{pmatrix} \frac{1}{\sqrt{2}} \\ 0 \\ -\frac{1}{\sqrt{2}} \end{pmatrix} \quad (22)$$

Thus, the first and third columns of  $\mathbf{R}_y$  give the  $x$ - and  $z$ -axes directions after the rotation.

### Rotation about the global $z$ -axis

To rotate by  $\phi_z$  about the global  $z$ -axis, the rotation matrix  $\mathbf{R}_z$  is:

$$\mathbf{R}_z = \begin{pmatrix} \cos \phi_z & -\sin \phi_z & 0 \\ \sin \phi_z & \cos \phi_z & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad (23)$$

Similarly, the first and second columns of  $\mathbf{R}_z$  give the  $x$ - and  $y$ -axes directions after rotating about the global  $z$ -axis.

### Three Successive Rotations about Body-Fixed $X$ , $Y$ , and then $Z$

The previous sections provided the elemental rotation matrices to rotate an object about axes in the **Global** reference frame, these were thus **extrinsic** rotations. These rotation matrices can nonetheless be chained to represent rotations about axes in a reference frame attached to the rotating body (body-fixed, the  $B$ -frame, **intrinsic** rotations as used by MuSkeMo).

In such a body-fixed convention, if we decide to use XYZ-Euler angles, we are in essence stating "I first align the body-fixed axes with the global reference frame. Then, I will first rotate my object about its

local X axis, then about its local Y axis, and lastly about its local Z axis, after which the body will be in its correct orientation with respect to the global reference frame”.

We can still use our previously defined rotation matrices about global axes to achieve the desired rotation:

$${}^g\mathbf{R}_{\mathcal{B}} = \mathbf{R}_x \mathbf{R}_y \mathbf{R}_z \quad (24)$$

**We said that we were starting with a rotation about  $X$ , but the vector will first be multiplied by  $R_z$  (remember the right to left application order of matrices). Why is this?.**

This occurs because instead of rotating about local-axes (expressed in the body-fixed frame  $\mathcal{B}$ ), we are rotating about the global axes. I will now demonstrate that this is simplifies to eq. 24.

Consider that we want to rotate a vector  $\vec{v}_{\mathcal{B}}$  (expressed in the body-fixed frame  $\mathcal{B}$ ) by arbitrary angles about body-fixed  $X$ ,  $Y$ , and  $Z$ -axes, in that order. After the rotations, we acquire  $\vec{v}_{\mathcal{G}}$ , the vector in the global frame  $\mathcal{G}$ . When the two frames are aligned (i.e., when all rotation angles are zero),  $\vec{v}_{\mathcal{B}} = \vec{v}_{\mathcal{G}}$ , but this trivial case gives no insight.

If we start by rotating about the body-fixed  $X$ -axis, we have:

$$\vec{v}_{\mathcal{G}} = \mathbf{R}_X \vec{v}_{\mathcal{B}} \quad (25)$$

Because frames were initially aligned, the body-fixed  $X$ -axis was coincident with the global  $x$ -axis. In other words (note the upper and lower case subscripts):

$$\mathbf{R}_X = \mathbf{R}_x \quad (26)$$

### Why Can't We Simply Add the Next Rotation by pre-multiplying the previous equation?

The next rotation becomes less intuitive. It may be tempting to add the rotation about the  $Y$ -axis as follows:

$$\underline{\mathbf{R}_y \mathbf{R}_X} \vec{v}_{\mathcal{B}} \quad (27)$$

However, this is incorrect! This formulation would result in a rotation about the global  $y$ -axis, as established previously. We need to re-express  $\mathbf{R}_y$  so that it acts about the local  $Y$ -axis. We can use eq. 8 to achieve this (filling in eq. 26 as well):

$$\mathbf{R}_Y = \mathbf{R}_X \mathbf{R}_y \mathbf{R}_X^T \quad (28)$$

This can be interpreted as "undoing" the first rotation (about  $X$ ), ensuring that local and global  $y$  axes coincide, which allows us to applying the global  $y$  rotation, and then redoing the first rotation.

Because rotation matrices are orthogonal matrices, multiplying by its own transpose gives the identity matrix. That means that our first two rotations can be described as:

$$\mathbf{R}_Y \mathbf{R}_X = \mathbf{R}_x \mathbf{R}_y \mathbf{R}_x^T \mathbf{R}_x = \mathbf{R}_x \mathbf{R}_y \quad (29)$$

In other words: successive rotations about body-fixed  $X$  and  $Y$  axes can be described by the same rotation matrices that describe successive rotations about global  $y$  and  $x$  axes! Once again, remember that matrix application should be read from right to left. Rotations about body-fixed axes are called *intrinsic*, whereas rotations about global axes are called *extrinsic*, but they can be described by applying the same rotation matrices in reverse order.

### Adding the Third Rotation

Before we can rotate our body about local  $Z$ , we need to re-express eq. 23 so that it acts about the body-fixed axis. Similar to eq. 28, we can apply a similarity transformation, now "undoing" the successive rotations about local  $X$  and  $Y$ :

$$\mathbf{R}_Z = \mathbf{R}_Y \mathbf{R}_X \mathbf{R}_z \mathbf{R}_X^T \mathbf{R}_Y^T \quad (30)$$

We can also fill in the result from eqs. 26 and 28 to expand the expression for  $\mathbf{R}_Z$  in terms of global rotation matrices, and simplify as much as possible:

$$\mathbf{R}_Y \mathbf{R}_X \mathbf{R}_z \mathbf{R}_X^T \mathbf{R}_Y^T = \mathbf{R}_X \mathbf{R}_y \mathbf{R}_X^T \mathbf{R}_X \mathbf{R}_z \mathbf{R}_X^T \mathbf{R}_X \mathbf{R}_y^T \mathbf{R}_X^T = \mathbf{R}_x \mathbf{R}_y \mathbf{R}_z \mathbf{R}_y^T \mathbf{R}_x^T \quad (31)$$

We can combine the results from eqs. 28 and 31 to write out the entire expanded matrix multiplication for our three successive rotations:

$$\mathbf{R}_Z \mathbf{R}_Y \mathbf{R}_X = \mathbf{R}_x \mathbf{R}_y \mathbf{R}_z \mathbf{R}_y^T \mathbf{R}_x^T \quad \mathbf{R}_x \mathbf{R}_y \mathbf{R}_x^T \quad \mathbf{R}_x = \mathbf{R}_x \mathbf{R}_y \mathbf{R}_z \quad (32)$$

Thus, a body-fixed  $XYZ$ -sequence of rotations (intrinsic) is identical to a global-frame  $zyx$ -sequence of (extrinsic) rotations!

Therefore, the total equation for the vector in the global frame after all three rotations is:

$$\vec{v}_{\mathcal{G}} = \mathbf{R}_x \mathbf{R}_y \mathbf{R}_z \vec{v}_{\mathcal{B}} \quad (33)$$

The inverse operation, which rotates from the global frame back to the body-fixed frame, is given by:

$${}^{\mathcal{B}} \mathbf{R}_{\mathcal{G}} = ({}^{\mathcal{G}} \mathbf{R}_{\mathcal{B}})^{-1} = ({}^{\mathcal{G}} \mathbf{R}_{\mathcal{B}})^T = \mathbf{R}_z^T \mathbf{R}_y^T \mathbf{R}_x^T \quad (34)$$

## B Rotation quaternions

The rotation matrix  ${}^{\mathcal{G}} \mathbf{R}_{\mathcal{B}}$  derived from a unit quaternion  $(w, x, y, z)$  is given by:

$${}^{\mathcal{G}} \mathbf{R}_{\mathcal{B}} = \begin{bmatrix} 1 - 2(z^2 + y^2) & 2(xy - wz) & 2(zx + wy) \\ 2(xy + wz) & 1 - 2(x^2 + z^2) & 2(yz - wx) \\ 2(zx - wy) & 2(yz + wx) & 1 - 2(x^2 + y^2) \end{bmatrix} \quad (35)$$

To convert a rotation matrix to a quaternion, MuSkeMo implements the algorithm described on page 537 of [14].

## C Axis angle rotation matrix

MuSkeMo uses the axis-angle approach described in [35], also known as Rodrigues' rotation formula, or Euler's finite rotation formula. It allows us to construct a rotation matrix from an axis vector  $\hat{\mathbf{n}}$ , and an angle  $\phi$  about which we would like to rotate.

We must first define the cross-product matrix:

$$\tilde{\mathbf{N}} = \begin{bmatrix} 0 & -n_z & n_y \\ n_z & 0 & -n_x \\ -n_y & n_x & 0 \end{bmatrix} \quad (36)$$

where  $\hat{\mathbf{n}} = (n_x, n_y, n_z)$  is the unit axis vector expressed in the global frame.

The identity matrix is:

$$\mathbf{I} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (37)$$

The rotation matrix is then acquired using:

$${}^g\mathbf{R}_{\mathcal{B}} = \mathbf{I} + (1 - \cos \phi) \tilde{\mathbf{N}} \tilde{\mathbf{N}} + \sin \phi \tilde{\mathbf{N}} \quad (38)$$

## D Validation tests

Many of MuSkeMo's modeling features rely on correctly computing positions and orientations with respect to local frames. Errors in some of these computations would be quite obvious during import (e.g., if the transformations are incorrectly propagated, resulting in incorrect model component positions and orientations). However, not all potential sources of errors can be detected visually (e.g., inertial properties inappropriately re-expressed in a different frame).

The following sections list several validation tests that rely on the correct application and propagation of all model construction and analysis steps. Section D.1 demonstrates that mesh inertial properties are computed correctly by MuSkeMo, and D.2 demonstrates these are correctly combined using the parallel axes theorem. Section D.2 also demonstrates internal consistency between rigid body parameters combined from different meshes ("composite body"), and rigid body parameters from combined meshes ("composite mesh"). Section D.3 demonstrates that MuSkeMo correctly re-expresses vectors and matrices in arbitrary frames. The scripts that construct the rotation matrices are reused by MuSkeMo for all transformations during model construction, import, and export. Section D.4 demonstrates that MuSkeMo can accurately compute moment arms from an imported OpenSim model. This validates MuSkeMo's implementation of cylindric wrapping from [17], and also demonstrates joint positions, orientations, and rotations are treated correctly by MuSkeMo, because the moment arms rely on their correct application.

All Blender scenes, files, scripts, and outputs described in the aforementioned sections are provided as a Github release. Where relevant, each Blender scene has a script in the script editor that manually performs all the calculations that are listed in the equations below.

MuSkeMo's pose sampling capabilities are tested in Section D.5. This section demonstrates that the intersection-checker (3.2.2) is computationally efficient (more than 10x faster than current methods, on a per-pose basis), and correctly recovers an a priori known viable pose space. This test was proposed by [5]. This test requires a separate Blender scene, also available on Github.

For further validation of MuSkeMo, the reader is referred to [37, 36], where animal models constructed using MuSkeMo were used for dynamic simulations, and simulator outputs were compared to empirical data from the animals in question.

### D.1 Mesh inertial properties

To compute inertial properties from arbitrary 3D meshes, MuSkeMo implements pseudo-code from [14] (see 3.3). I will validate the implementation in MuSkeMo by demonstrating that it: 1) accurately computes the inertial tensors for objects with known inertial tensors; 2) inertial properties of a complex mesh acquired from a CT scan match the outputs from Meshlab [10]. We will assume a density of  $1000 \text{ kgm}^{-3}$  for all computations, although this is arbitrary since all of the underlying computations are volumetric only (see 3.3).

The inertial properties are computed through successive multiplications of the 3D vertex coordinates of the input meshes. Because Blender stores these coordinates as 32 bit single precision digits (see 1.4), the repeated multiplications required to compute the inertial properties will reduce their precision to approximately 5-6 digits.

#### Icosahedron and sphere

If the edge lengths  $l_e$  are known, the volume of an icosahedron can be computed as [29]:

$$V = l_e^3 \cdot \frac{5}{6} \cdot \tau^2, \quad (39)$$

where  $\tau = \frac{\sqrt{5}+1}{2}$  is the golden ratio. Multiplying the volume by the density  $\rho$  gives the mass  $m$ .

In Blender, an icosahedron with a default radius of 1 has  $l_e = 1.05146$  m (Fig. 9). Using the algebraic equation, the computed mass is 2536.13 kg. MuSkeMo computes a mass of 2536.15 kg.

The principal moments of inertia of an icosahedron can be computed as [29]:

$$I = \frac{\tau^2}{10} \cdot l_e^2 \cdot m, \quad (40)$$

where all three principal moments are equal.

Using the algebraic equations, the computed moment of inertia is  $734.063 \text{ kgm}^2$ . MuSkeMo provides a moment of inertia value of  $734.070 \text{ kgm}^2$ .

The volume of a sphere with radius  $r_s$  can be computed as:

$$V = 4/3 \cdot \pi \cdot r_s^3 \quad (41)$$

Thus, a sphere with  $r_s = 1$  m weighs 4188.79 kg. MuSkeMo computes its mass as 4188.65 kg.

The principal moments of inertia of a sphere can be computed as [27]:

$$I = 2/5 \cdot m \cdot r_s^2, \quad (42)$$

again with all three principal moments equal. This gives  $1675.52 \text{ kgm}^2$  for our sphere. MuSkeMo computes it as  $1675.42 \text{ kgm}^2$ .

We will reuse the parameters of the icosahedron and the sphere in D.2.

### Complex tissue model from CT scan

3D models acquired from CT scans can be complex and irregular. The algorithm presented by Eberly [14] can deal with such meshes without problems. Coatham et al. [11] provided a dataset of CT scanned animals, with manually segmented tissue (skin) outlines. Fig. 8 shows the torso tissue outline of a cheetah, the "Torso.obj" mesh from the "skin" folder of the *Acinonyx jubatus* dataset. This is a relatively detailed mesh ( 78 MB), likely an unnecessary level of detail for a rigid body model.

Using the "Compute Geometric Measures" filter, Meshlab gives:  $V = 0.027241 \text{ m}^3$ , center of mass =  $(0.461789, -0.002036, 0.555943)$  m, and the volumetric inertia tensor as  $(0.000177, 0.002123, 0.002045, 0.000060, 0.000113, 0.000011) \text{ m}^5$ . MuSkeMo gives the following outputs (setting density at  $1000 \text{ kgm}^{-3}$ ):  $m = 27.2405 \text{ kg}$ , center of mass =  $(0.461789, -0.002036, 0.555943)$  m, and the mass inertial tensor as  $(0.176614, 2.12329, 2.0455, 0.060367, 0.113492, 0.010864)$ .

The numerical values are the same, save for a factor of 1000 representing the density.

## D.2 Composite bodies and meshes

MuSkeMo can combine the inertial properties of several source objects into a single composite body using the (3D) parallel axes theorem [35, 27]. Here, we will compute the values manually for the situation pictured in Fig. 9, and compare the computations by MuSkeMo.

Consider the icosahedron and sphere from D.1. The icosahedron has mass  $m_i = 2536.13 \text{ kg}$ , center of mass  $\vec{c}_i = (0, 0, 0)$  m, and its inertia tensor:

$$\mathbf{I}_{i/c, G} = \begin{pmatrix} 734.063 & 0 & 0 \\ 0 & 734.063 & 0 \\ 0 & 0 & 734.063 \end{pmatrix}. \quad (43)$$

For the inertia tensor, the subscript  $i$  denotes the icosahedron, and  $c$  signifies the inertia is given with respect to its center of mass. Like in A,  $G$  signifies the tensor is expressed in the global frame.

The sphere has mass  $m_s = 4188.79 \text{ kg}$ , center of mass  $\vec{c}_s = (4, 1, -1.5)$  m, and its inertia tensor:

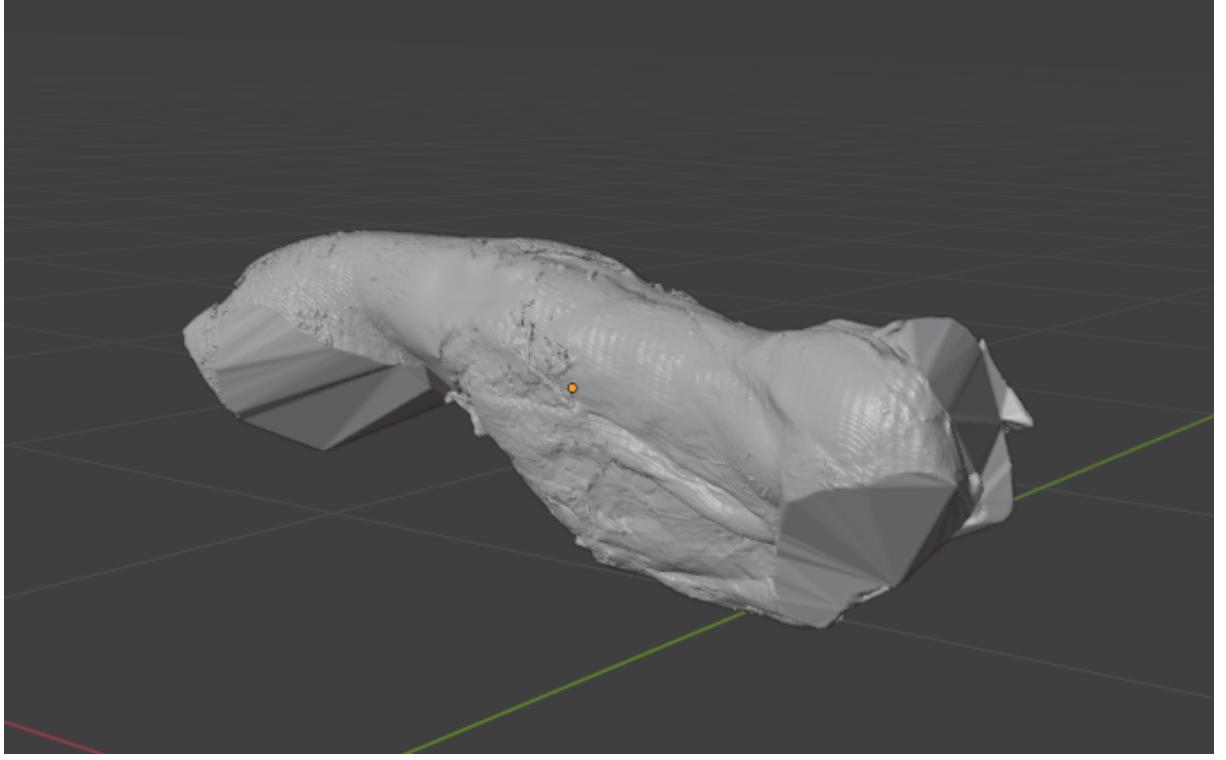


Figure 8: The "Torso.obj" mesh from [11], providing a 3D model of the torso of a cheetah, segmented from a CT-scan.

$$\mathbf{I}_{s/c,\mathcal{G}} = \begin{pmatrix} 1675.52 & 0 & 0 \\ 0 & 1675.52 & 0 \\ 0 & 0 & 1675.52 \end{pmatrix} \quad (44)$$

The combined center of mass  $\vec{c}_c$  is:

$$\frac{m_i \cdot \vec{c}_i + m_s \cdot \vec{c}_s}{m_i + m_s} = \vec{c}_c, \quad (45)$$

which is at (2.4915, 0.6229, -0.9343) m in the  $\mathcal{G}$ -frame when we compute it manually. In MuSkeMo, the inertial properties computed for the icosahedron and the sphere in D.1 can be assigned to a single body in the Body panel (see 3.4). When their rigid body parameters are assigned to a BODY named "composite.body", the combined center of mass is computed as (2.4915, 0.6229,-0.9343) m.

The combined inertial parameters can be calculated manually using the 3D parallel axes theorem [35, 27]. To fully write out the equation, it is useful to first define:

$$\vec{d}_{i/c} = \vec{c}_i - \vec{c}_c, \quad (46)$$

and a matrix constructed of its elements:

$$\mathbf{D}_{i//c,\mathcal{G}} = \begin{pmatrix} \vec{d}_{i/c,y}^2 + \vec{d}_{i/c,z}^2 & -\vec{d}_{i/c,x} \cdot \vec{d}_{i/c,y} & -\vec{d}_{i/c,x} \cdot \vec{d}_{i/c,z} \\ -\vec{d}_{i/c,x} \cdot \vec{d}_{i/c,y} & \vec{d}_{i/c,x}^2 + \vec{d}_{i/c,z}^2 & -\vec{d}_{i/c,y} \cdot \vec{d}_{i/c,z} \\ -\vec{d}_{i/c,x} \cdot \vec{d}_{i/c,z} & -\vec{d}_{i/c,y} \cdot \vec{d}_{i/c,z} & \vec{d}_{i/c,x}^2 + \vec{d}_{i/c,y}^2 \end{pmatrix}. \quad (47)$$

The matrix  $\mathbf{D}_{s//c,\mathcal{G}}$  is similarly constructed from  $\vec{d}_{s/c}$ . It is also possible to define  $\vec{d}_{c//i}$  and its respective matrix  $\mathbf{D}_{c//i,\mathcal{G}}$ , but because each term is either squared or multiplied with another term, this gives the same result.

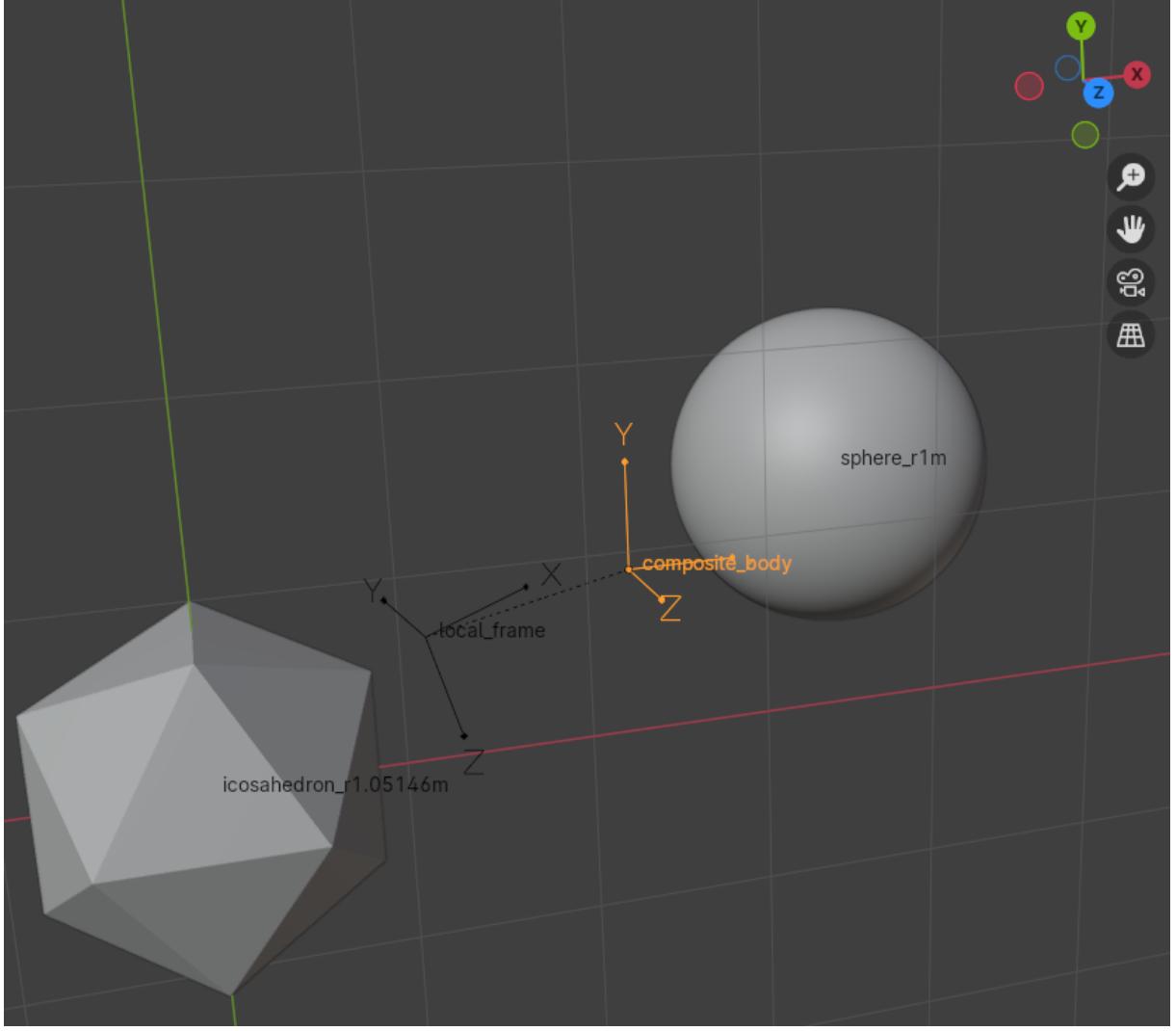


Figure 9: The icosahedron, sphere, composite body, and local frame described in sections D.1, D.2, and D.3

To calculate  $\mathbf{I}_{c/c,\mathcal{G}}$ , the combined inertial properties with respect to  $\vec{c}_c$  in the  $\mathcal{G}$ -frame, we apply the following:

$$\mathbf{I}_{c/c,\mathcal{G}} = \mathbf{I}_{i/c,\mathcal{G}} + m_i \cdot \mathbf{D}_{i//c,\mathcal{G}} + \mathbf{I}_{s/c,\mathcal{G}} + m_s \cdot \mathbf{D}_{s//c,\mathcal{G}}, \quad (48)$$

which is the 3D parallel axes theorem [35, 27].

Manually calculated, the inertial properties are: (7543.59, 31239.0, 29264.4, -6318.78, 9478.17, 2369.54)  $\text{kgm}^2$ , using the same element order as defined in 4.1. After applying the parallel axes theorem, MuSkeMo computes the inertial properties as: (7543.47, 31238.7, 29264.1, -6318.73, 9478.10, 2369.53)  $\text{kgm}^2$ .

As an extra internal validation step, it is also possible to combine the meshes from Fig. 9 into a single composite mesh, using Blender's join command. If inertial properties for this composite mesh are computed using MuSkeMo, MuSkeMo treats them as a single (but disjointed) mesh, and uses Eberly's algorithm to compute the combined rigid body parameters. This gives numerically identical results, even though the "composite body" approach uses the parallel axes theorem to combine two mesh inertial properties, whereas the "composite mesh" approach computes the inertial properties as if they were one single mesh.

### D.3 Re-expressing matrices and vectors in arbitrary frames

When assigning a local frame (see 4.7) to a BODY, MuSkeMo also expresses the parameters of all associated model components with respect to that local frame (see 4). The scripts that compute these matrices are reused throughout MuSkeMo, so we will demonstrate their implementation by expressing the composite rigid body parameters from D.2 with respect to a local frame.

The frame is pictured in 9. The frame's global position is  $\vec{r}_f = (1,1,1)$  m. Its Euler angles are  $45^\circ (\phi_x)$ ,  $10^\circ (\phi_y)$ , and  $25^\circ (\phi_z)$ , see A for a full treatment on Euler angles in MuSkeMo. Thus, the rotation matrix  ${}^g\mathbf{R}_{\mathcal{B}}$ , which rotates a vector from the local frame to the global frame, follows from equation 13 as:

$${}^g\mathbf{R}_{\mathcal{B}} = \begin{pmatrix} 0.892539 & -0.416198 & 0.173648 \\ 0.410120 & 0.588964 & -0.696364 \\ 0.187553 & 0.692749 & 0.696364 \end{pmatrix} \quad (49)$$

To express the combined center of mass  $\vec{c}_c$  from D.2 with respect to the local frame, we will start by explicitly defining in which frames the vectors are expressed,  $\mathcal{G}$  for the global frame, and  $\mathcal{B}$  for the local frame (the B stands for body-fixed). To compute the transformation manually, we use:

$$\vec{c}_{c,\mathcal{B}} = {}^B\mathbf{R}_{\mathcal{G}} \cdot (\vec{c}_{c,\mathcal{G}} - \vec{r}_{f,\mathcal{G}}). \quad (50)$$

Calculated manually, this gives: (0.813771, -2.18287, -0.825374) m. When assigning the local frame to the composite body, MuSkeMo computes: (0.813736, -2.18285, -0.825365) m.

To re-express  $\mathbf{I}_{c/c,\mathcal{G}}$  in the *mathcal{B}*-frame (but still with respect to the combined center of mass), we apply equation 8 (as can be found in standard mechanics textbooks [35]):

$$\mathbf{I}_{c/c,\mathcal{B}} = {}^B\mathbf{R}_{\mathcal{G}} \cdot \mathbf{I}_{c/c,\mathcal{G}} \cdot {}^g\mathbf{R}_{\mathcal{B}}. \quad (51)$$

Manual calculations give: (11205.0, 25752.7, 31089.3, 12358.1, 6113.85, -3495.72) kgm<sup>2</sup>. MuSkeMo computes: (11204.8, 25752.4, 31089.0, 12358.0, 6113.81, -3495.70) kgm<sup>2</sup>.

### D.4 Muscle moment arms

Several definitions of muscle moment arms exist [2]. MuSkeMo computes the moment arms using the principle of virtual work [2, 34]. For muscle  $m$  crossing a joint with joint angle  $\phi_j$ , the moment arm  $r_m$  is determined by changes in the muscle's length  $l_m$  as a function of the joint angle:

$$r_m = -dl_m/d\phi_j \quad (52)$$

The minus sign in eq. D.4 appears due to the convention in musculoskeletal simulators to define muscle contractile force as a positive scalar, even though muscle contraction results in a negative change in length.

Accurate computations of moment arms therefore require correct application of local frames, positions and orientations, require accurate changes in muscle point positions when the joints are rotated, and also require accurate computations of the instantaneous muscle lengths in all of the different joint poses. Fig. 10 compares the moment arms computed by MuSkeMo after importing an OpenSim model, to the moment arms of the same model computed by OpenSim 4.0 using the Matlab api. These are the same data that are combined into three plots in Fig. 4. Note that the moment arms computed by MuSkeMo will only match those computed by OpenSim if the model only uses wrapping cylinders, with each wrapped section separated by a curve point (or if the model uses no wrapping at all).

The OpenSim model used for this comparison was the emu model from [37], specifically `Dromaius_model_v4_intermed.osim`. Fig. 10 shows that all the moment arms are numerically identical within expected precision for Blender, as evidenced by the normalized root mean square differences (nrmsd) not exceeding

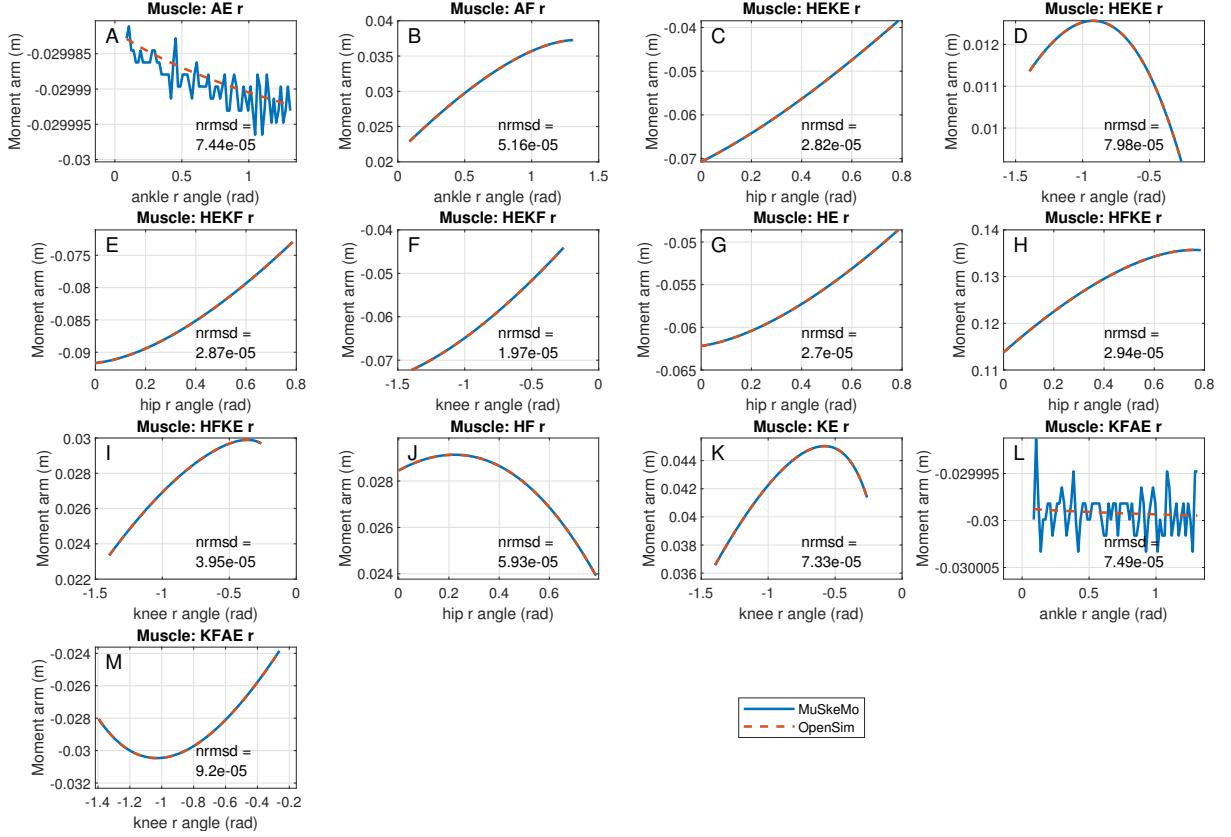


Figure 10: Individual muscle moment arms of the emu model from [37], computed by MuSkeMo and exported as .csv files, compared to those computed by OpenSim 4.0 using the Matlab API. nrmse stands for "normalized root mean squared differences", rmsd is normalized by the mean value of the moment arm. The average nrmse over all muscles was 0.0052% of the moment arm magnitudes. Panels A and L show that the moment arms are slightly noisy (in the order of 0.015% the magnitude of the moment arm) when wrapping determines the moment arms in question. As evidenced by similar values of nrmse, this noise is too small to be relevant in most practical applications (e.g., compare this figure to the same data plotted in Fig. 4, where the noise is not perceptible over a larger range). The python script that performs this moment arm analysis is provided in the Utilities folder 3.14

$9.2 \cdot 10^{-4}$ , or 0.0092% of the moment arm magnitudes, with an average of 0.0052% over all the muscles. To compute nrmse, rmsd was normalized by the mean of the moment arm of each muscle.

Fig. 10 also shows that the moment arms in panels A and L are somewhat noisy. Closer inspection of the vertical axes of those panels reveals that the moment arms are nearly constant for those muscles ( $\sim 0.03$  m), because the moment arms are determined by a cylinder with constant diameter radius (0.03 m). Thus, the noise is numerically insignificant (in the order of 0.015% of the moment arm magnitude). The noise is so small in magnitude that it is not visible when plotting multiple muscles in the same plot, with more informative ranges on the y axis (Fig. 4).

While the noise is small enough to make no difference in a practical sense, it is explicitly pointed out here, because readers who want to use MuSkeMo's moment arms directly for their own simulations may want to smooth the moment arms first. The noisy moment arms for wrapping muscles are likely caused by wrapping node (see sec. 4.3.3). A likely possibility is that the many mathematical operations that are performed in the node on single precision digits introduce floating point errors (see A Note on Precision). Another possible culprit may be inaccuracy of the Raycasting operation that geometrically determines intersections between a wrapping geometry and a muscle curve.

The python script ("moment\_arm\_analysis.py", to be run in Blender's script editor) that generates the muscle analysis is provided in MuSkeMo's utilities folder as an example script that calls MuSkeMo's API (sec. 3.14). The Matlab script that generates the plots from Figs. 4 and 10 is also provided

(moment\_arm\_analysis\_plot.m). This requires the OpenSim API to be installed.

### Moment arm hyperplane

Even though the emu model in the previous section only has degrees of freedom about the sagittal plane, because the paths themselves are in 3D, MuSkeMo computes everything in 3D. This is already demonstrated in sec. D.3, because all of the underlying 3D transformations in MuSkeMo use the same scripts. Thus, it does not matter what rotational axis is chosen, MuSkeMo computes the moment arms accurately. Fig. 11 confirms: this plot shows the moment arm hyperplane acquired over an arbitrary pose-space encompassing hip abduction (Rx) and hip flexion (Rz). The moment arms computed by MuSkeMo and OpenSim are essentially identical (normalized root mean squared difference = 0.006 %).

The analysis in MuSkeMo requires no further modifications to the model, and can be performed by running “moment\_arm\_hyperplane.py”. To generate the plot, the OpenSim model had to be modified (an Rx coordinate had to be added to the emu model). This version of the model, including the Matlab code to produce the figure, is available as a Github release.

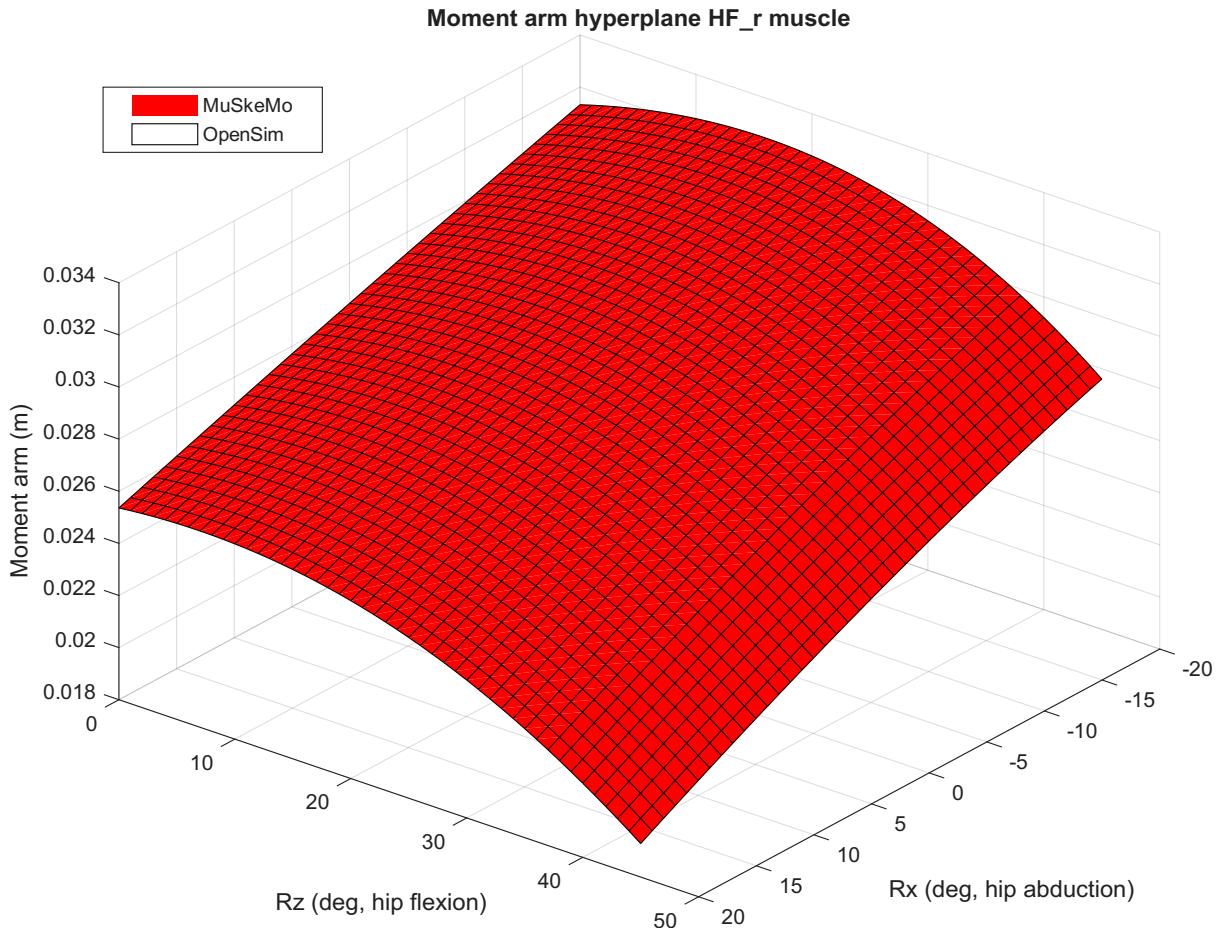


Figure 11: Moment arm hyperplane of the “HF\_r” hip flexor muscle from the emu model [37], comparison between MuSkeMo and OpenSim. The plot compares the muscle’s moment arm over different combinations Rx (hip abduction) and Rz (hip flexion). Normalized root mean squared differences were 0.006 %, meaning the outputs are essentially identical. The python script that performs this moment arm analysis is provided in the Utilities folder 3.14

### D.5 Pose sampler stress test

Bishop et al. [5] proposed an elegant stress test for pose sampling procedures. The test consists of a screw and a thread, with relatively little spacing between the two FIGURE. The screw has very little clearance to move through the thread, and requires a simultaneous rotation and translation to do so

without intersections. The goal of the test is to find the simultaneous coupled rotations and translations of the screw as it passes through the thread. Because the test requires a very high sample density, it is essentially designed to punish exhaustive search methods. I have nevertheless performed the test using an exhaustive search approach, to demonstrate MuSkeMo’s intersection checking speed.

A Blender scene with this testing setup is available on Github. In the script editor, **6DOFPoseSampleTest.py** is preloaded, but with the two `sample_density` parameters set to 1 to prevent long computation times when users simply want to try out the script (see sec. 3.14.7). Set `sample_density` to 3 to perform the full test. This test takes X HOURS to complete on an Intel i7-11850H 2.5 GHz Laptop processor, and tests XXXXXX samples in this time frame, thus giving a speed of XXX samples per second. The results are presented in FIGURE. This shows that MuSkeMo is correctly able to identify viable poses based on mesh intersections, and the per-pose sampling rate is at least an order of magnitude faster than other methods.

The total search time is still long, and the sample density is still lower than Bishop et al.’s reference results [5]. This is because we are exhaustively searching through all combinations of poses, which is not an efficient approach. A search method that explores future viable poses starting from existing viable poses would dramatically improve performance [5]. Parallelization provided Bishop et al. a further benefit [5], but this is currently not possible in Blender. However, because MuSkeMo’s intersection checker is so fast, simple pose sampling analyses may not require it (e.g., sec. 3.14.7). Most biological analyses will not require as fine a sampling density as the stress test.

The theoretical ideal translations and orientations plotted in FIGURE are derived in the supplementary information of [5]. The Matlab code that generates the figure is provided with the Blender scene on Github.

## E Acknowledgements

## References

- [1] R. McN Alexander. *Principles of Animal Locomotion*. Princeton NJ: Princeton University Press, 2006. 1-371. ISBN: 0-691-12634-8.
- [2] K. N. An et al. “Determination of Muscle Orientations and Moment Arms”. In: *Journal of Biomechanical Engineering* 106.3 (1984), pp. 280–282. ISSN: 15288951. DOI: 10.1115/1.3138494. PMID: 6492774.
- [3] G. L. Baskerville. “Use of Logarithmic Regression in the Estimation of Plant Biomass”. In: *Canadian Journal of Forest Research* 2.1 (Mar. 1, 1972), pp. 49–53. ISSN: 0045-5067, 1208-6037. DOI: 10.1139/x72-009. URL: <http://www.nrcresearchpress.com/doi/10.1139/x72-009> (visited on 10/26/2024).
- [4] Karl T. Bates et al. “Running Performance in *Australopithecus Afarensis*”. In: *Current Biology* 35.1 (Jan. 2025), 224–230.e4. ISSN: 09609822. DOI: 10.1016/j.cub.2024.11.025. URL: <https://linkinghub.elsevier.com/retrieve/pii/S0960982224015665> (visited on 01/24/2025).
- [5] Peter J. Bishop, Robert J. Brocklehurst, and Stephanie E. Pierce. “Intelligent Sampling of High-dimensional Joint Mobility Space for Analysis of Articular Function”. In: *Methods in Ecology and Evolution* 14.2 (Feb. 2023), pp. 569–582. ISSN: 2041-210X, 2041-210X. DOI: 10.1111/2041-210X.14016. URL: <https://besjournals.onlinelibrary.wiley.com/doi/10.1111/2041-210X.14016> (visited on 09/25/2025).
- [6] Charlotte A. Brassey and William I. Sellers. “Scaling of Convex Hull Volume to Body Mass in Modern Primates, Non-Primate Mammals and Birds”. In: *PLoS ONE* 9.3 (Mar. 11, 2014). Ed. by Alistair Robert Evans, e91691. ISSN: 1932-6203. DOI: 10.1371/journal.pone.0091691. URL: <https://dx.plos.org/10.1371/journal.pone.0091691> (visited on 10/26/2024).
- [7] Charlotte A. Brassey et al. “A Volumetric Technique for Fossil Body Mass Estimation Applied to *Australopithecus Afarensis*”. In: *Journal of Human Evolution* 115 (Feb. 2018), pp. 47–64. ISSN: 00472484. DOI: 10.1016/j.jhevol.2017.07.014. URL: <https://linkinghub.elsevier.com/retrieve/pii/S0047248417303196> (visited on 10/26/2024).
- [8] Charlotte A. Brassey et al. “Convex-Hull Mass Estimates of the Dodo (*Raphus Cucullatus*) : Application of a CT-based Mass Estimation Technique”. In: *PeerJ* 4 (Jan. 11, 2016), e1432. ISSN: 2167-8359. DOI: 10.7717/peerj.1432. URL: <https://peerj.com/articles/1432> (visited on 10/26/2024).
- [9] Y. Chen and G. Medioni. “Object Modeling by Registration of Multiple Range Images”. In: *Proceedings. 1991 IEEE International Conference on Robotics and Automation*. 1991 IEEE International Conference on Robotics and Automation. Sacramento, CA, USA: IEEE Comput. Soc. Press, 1991, pp. 2724–2729. ISBN: 978-0-8186-2163-5. DOI: 10.1109/ROBOT.1991.132043. URL: <http://ieeexplore.ieee.org/document/132043/> (visited on 08/31/2025).
- [10] Paolo Cignoni et al. *MeshLab: An Open-Source Mesh Processing Tool*. Eurographics Italian Chapter Conference. 2008. DOI: 10.2312/LOCALCHAPTEREVENTS/ITALCHAP/ITALIANCHAPCONF2008/129-136. URL: <http://diglib.eg.org/handle/10.2312/LocalChapterEvents.ItalChap.ItalianChapConf2008.129-136> (visited on 01/24/2025). Pre-published.
- [11] Samuel J. Coatham, William I. Sellers, and Thomas A. Püschel. “Convex Hull Estimation of Mammalian Body Segment Parameters”. In: *Royal Society Open Science* 8.6 (June 2021), p. 210836. ISSN: 2054-5703. DOI: 10.1098/rsos.210836. URL: <https://royalsocietypublishing.org/doi/10.1098/rsos.210836> (visited on 11/16/2021).
- [12] Verónica Díez Díaz et al. “Centres of Rotation and Osteological Constraints on Caudal Ranges of Motion in the Sauropod Dinosaur *Giraffatitan Brancai*”. In: *Royal Society Open Science* 12.8 (Aug. 2025), p. 250851. ISSN: 2054-5703. DOI: 10.1098/rsos.250851. URL: <https://royalsocietypublishing.org/doi/10.1098/rsos.250851> (visited on 08/25/2025).
- [13] David Eberly. *Least Squares Fitting of Data by Linear or Quadratic Structures*. URL: <https://www.geometrictools.com/Documentation/LeastSquaresFitting.pdf>.
- [14] David H. Eberly and Ken Shoemake. *Game Physics*. 1st ed. The Morgan Kaufmann Series in Interactive 3D Technology. Amsterdam ; Boston: Elsevier/Morgan Kaufmann, 2004. 776 pp. ISBN: 978-1-55860-740-8.
- [15] Christer Ericson. “Bounding Volume Hierarchies”. In: *Real-Time Collision Detection*. Morgan Kaufmann Series in Interactive 3D Technology. Amsterdam ; Boston: Elsevier, 2005. ISBN: 978-1-55860-732-3.

- [16] Franz K. Fuss and Chris R. Gasser. “Cruciate Ligaments of the Avian Knee: Insight into a Complex System”. In: *Journal of Morphology* 214.2 (Nov. 1992), pp. 139–151. ISSN: 0362-2525, 1097-4687. DOI: 10.1002/jmor.1052140204. URL: <https://onlinelibrary.wiley.com/doi/10.1002/jmor.1052140204> (visited on 08/25/2025).
- [17] Brian A. Garner and Marcus G. Pandy. “The Obstacle-Set Method for Representing Muscle Paths in Musculoskeletal Models”. In: *Computer Methods in Biomechanics and Biomedical Engineering* 3.1 (Jan. 2000), pp. 1–30. ISSN: 1025-5842, 1476-8259. DOI: 10.1080/10255840008915251. URL: <http://www.tandfonline.com/doi/abs/10.1080/10255840008915251> (visited on 10/11/2024).
- [18] Stephen M. Gatesy et al. “A Proposed Standard for Quantifying 3-D Hindlimb Joint Poses in Living and Extinct Archosaurs”. In: *Journal of Anatomy* 241.1 (2022), pp. 101–118. ISSN: 0021-8782, 1469-7580. DOI: 10.1111/joa.13635. URL: <https://onlinelibrary.wiley.com/doi/10.1111/joa.13635> (visited on 04/18/2024).
- [19] Charles F Jekel. “Digital Image Correlation on Steel Ball”. In: *Obtaining Non-Linear Orthotropic Material Models for Pvc-Coated Polyester via Inverse Bubble Inflation (MSc Thesis)*. Stellenbosch University, 2016, pp. 83–87. HDL: 10019.1/98627. URL: <https://hdl.handle.net/10019.1/98627>.
- [20] I T Jolliffe. “Mathematical and Statistical Properties of Sample Principal Components”. In: *Principal Component Analysis*. New York: Springer-Verlag, 2002, pp. 29–61. ISBN: 978-0-387-95442-4. DOI: 10.1007/0-387-22440-8\_3. URL: [http://link.springer.com/10.1007/0-387-22440-8\\_3](http://link.springer.com/10.1007/0-387-22440-8_3) (visited on 08/27/2025).
- [21] Thomas R. Kane, Peter W. Likins, and David A. Levinson. *Spacecraft Dynamics*. New York Hamburg: McGraw-Hill, 1983. 436 pp. ISBN: 978-0-07-037843-8.
- [22] David C. Lay, Steven R. Lay, and Judith McDonald. *Linear Algebra and Its Applications*. Fifth edition. Boston: Pearson, 2016. ISBN: 978-0-321-98238-4.
- [23] Sophie Macaulay et al. “Decoupling Body Shape and Mass Distribution in Birds and Their Dinosaurian Ancestors”. In: *Nature Communications* 14.1 (Mar. 22, 2023), p. 1575. ISSN: 2041-1723. DOI: 10.1038/s41467-023-37317-y. URL: <https://www.nature.com/articles/s41467-023-37317-y> (visited on 04/03/2023).
- [24] Armita R. Manafzadeh and Kevin Padian. “ROM Mapping of Ligamentous Constraints on Avian Hip Mobility: Implications for Extinct Ornithodirans”. In: *Proceedings of the Royal Society B: Biological Sciences* 285.1879 (May 30, 2018), p. 20180727. ISSN: 0962-8452, 1471-2954. DOI: 10.1098/rspb.2018.0727. URL: <https://royalsocietypublishing.org/doi/10.1098/rspb.2018.0727> (visited on 11/06/2024).
- [25] Karl Pearson. “LIII. On Lines and Planes of Closest Fit to Systems of Points in Space”. In: *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science* 2.11 (Nov. 1901), pp. 559–572. ISSN: 1941-5982, 1941-5990. DOI: 10.1080/14786440109462720. URL: <https://www.tandfonline.com/doi/full/10.1080/14786440109462720> (visited on 08/27/2025).
- [26] D Rose. *Rotations in Three-Dimensions: Euler Angles and Rotation Matrices*. Feb. 2015. URL: [https://danceswithcode.net/engineeringnotes/rotations\\_in\\_3d/rotations\\_in\\_3d\\_part1.html](https://danceswithcode.net/engineeringnotes/rotations_in_3d/rotations_in_3d_part1.html) (visited on 03/19/2025).
- [27] Andy Ruina and Rudra Pratap. *Mechanics Toolset, Statics and Dynamics*. 2019. 578-579. URL: <http://ruina.tam.cornell.edu/Book/>.
- [28] S. Rusinkiewicz and M. Levoy. “Efficient Variants of the ICP Algorithm”. In: *Proceedings Third International Conference on 3-D Digital Imaging and Modeling*. Third International Conference on 3-D Digital Imaging and Modeling. Quebec City, Que., Canada: IEEE Comput. Soc, 2001, pp. 145–152. ISBN: 978-0-7695-0984-6. DOI: 10.1109/IM.2001.924423. URL: <http://ieeexplore.ieee.org/document/924423/> (visited on 08/31/2025).
- [29] John Satterly. “The Moments of Inertia of Some Polyhedra”. In: *The Mathematical Gazette* 42.339 (Feb. 1958), pp. 11–13. ISSN: 0025-5572, 2056-6328. DOI: 10.2307/3608345. URL: [https://www.cambridge.org/core/product/identifier/S0025557200037682/type/journal\\_article](https://www.cambridge.org/core/product/identifier/S0025557200037682/type/journal_article) (visited on 01/24/2025).
- [30] W. I. Sellers et al. “Minimum Convex Hull Mass Estimations of Complete Mounted Skeletons”. In: *Biology Letters* 8.5 (2012), pp. 842–845. ISSN: 1744957X. DOI: 10.1098/rsbl.2012.0263. PMID: 22675141.
- [31] Mark Semple. *pyEllipsoid\_Fit*. URL: [https://github.com/marksemple/pyEllipsoid\\_Fit](https://github.com/marksemple/pyEllipsoid_Fit).
- [32] Richard J. Smith. “Logarithmic Transformation Bias in Allometry”. In: *American Journal of Physical Anthropology* 90.2 (Feb. 1993), pp. 215–228. ISSN: 0002-9483, 1096-8644. DOI: 10.1002/ajpa.

1330900208. URL: <https://onlinelibrary.wiley.com/doi/10.1002/ajpa.1330900208> (visited on 09/23/2025).
- [33] D. G. Sprugel. “Correcting for Bias in Log-Transformed Allometric Equations”. In: *Ecology* 64.1 (Feb. 1983), pp. 209–210. ISSN: 0012-9658, 1939-9170. DOI: 10.2307/1937343. URL: <https://esajournals.onlinelibrary.wiley.com/doi/10.2307/1937343> (visited on 09/23/2025).
- [34] Anthony Storace and Barry Wolf. “Functional Analysis of the Role of the Finger Tendons”. In: *Journal of Biomechanics* 12.8 (Jan. 1979), pp. 575–578. ISSN: 00219290. DOI: 10.1016/0021-9290(79)90076-9. URL: <https://linkinghub.elsevier.com/retrieve/pii/0021929079900769> (visited on 01/31/2025).
- [35] Heike Vallery and Arend L Schwab. *Advanced Dynamics*. Delft University of Technology, 2019. 481 pp. ISBN: 978-94-6186-948-7.
- [36] Pasha A van Bijlert et al. “Muscle-Driven Predictive Physics Simulations of Quadrupedal Locomotion in the Horse”. In: *Integrative And Comparative Biology* 64.3 (Sept. 27, 2024), pp. 694–714. ISSN: 1540-7063, 1557-7023. DOI: 10.1093/icb/icae095. URL: <https://academic.oup.com/icb/article/64/3/694/7713463> (visited on 10/17/2024).
- [37] Pasha A. Van Bijlert et al. “Muscle-Controlled Physics Simulations of Bird Locomotion Resolve the Grounded Running Paradox”. In: *Science Advances* 10.39 (Sept. 27, 2024), eado0936. ISSN: 2375-2548. DOI: 10.1126/sciadv.ado0936. URL: <https://www.science.org/doi/10.1126/sciadv.ado0936> (visited on 09/26/2024).
- [38] M A Wright, T J Cavanaugh, and S E Pierce. “Volumetric versus Element-scaling Mass Estimation and Its Application to Permo-Triassic Tetrapods”. In: *Integrative Organismal Biology* (Sept. 13, 2024), obae034. ISSN: 2517-4843. DOI: 10.1093/iob/obae034. URL: <https://academic.oup.com/iob/advance-article/doi/10.1093/iob/obae034/7756867> (visited on 09/22/2024).
- [39] Ge Wu and Peter R. Cavanagh. “ISB Recommendations for Standardization in the Reporting of Kinematic Data”. In: *Journal of Biomechanics* 28.10 (Oct. 1995), pp. 1257–1261. ISSN: 00219290. DOI: 10.1016/0021-9290(95)00017-C. URL: <https://linkinghub.elsevier.com/retrieve/pii/002192909500017C> (visited on 04/18/2024).
- [40] Ge Wu et al. “ISB Recommendation on Definitions of Joint Coordinate System of Various Joints for the Reporting of Human Joint Motion—Part I: Ankle, Hip, and Spine”. In: *Journal of Biomechanics* 35.4 (Apr. 2002), pp. 543–548. ISSN: 00219290. DOI: 10.1016/S0021-9290(01)00222-6. URL: <https://linkinghub.elsevier.com/retrieve/pii/S0021929001002226> (visited on 04/18/2024).
- [41] Ge Wu et al. “ISB Recommendation on Definitions of Joint Coordinate Systems of Various Joints for the Reporting of Human Joint Motion—Part II: Shoulder, Elbow, Wrist and Hand”. In: *Journal of Biomechanics* 38.5 (May 2005), pp. 981–992. ISSN: 00219290. DOI: 10.1016/j.jbiomech.2004.05.042. URL: <https://linkinghub.elsevier.com/retrieve/pii/S002192900400301X> (visited on 12/08/2024).
- [42] Sumith Yesudasan. “Fast Geometric Fit Algorithm for Sphere Using Exact Solution”. Version 1. In: *arXiv (preprint)* (2015). DOI: 10.48550/ARXIV.1506.02776. URL: <https://arxiv.org/abs/1506.02776> (visited on 10/11/2024).
- [43] Zhengyou Zhang. “Iterative Point Matching for Registration of Free-Form Curves and Surfaces”. In: *International Journal of Computer Vision* 13.2 (Oct. 1994), pp. 119–152. ISSN: 0920-5691, 1573-1405. DOI: 10.1007/BF01427149. URL: <http://link.springer.com/10.1007/BF01427149> (visited on 08/31/2025).