Hyfydy User Manual

1. Introduction

Hyfydy is software for high-performance musculoskeletal simulation, with a focus on biomechanics research and predictive simulation of human and animal motion. It supports bodies, joints, contacts, musculotendon actuators, and torque actuators. Some of its key features include:

- **High performance**. Hyfydy simulations run approximately 100x faster than OpenSim [1], using the same muscle and contact models*. Hyfydy is also fast enough for use in real-time applications such as games.
- **Stable integration**. Efficient error-controlled variable-step integration methods ensure the simulation always runs stable at a user-specified accuracy level, without sacrificing performance.
- **Accurate models**. Hyfydy contains research-grade state-of-the-art models for musculotendon dynamics and contact forces.
- **Force-based constraints**. Joint constraints in Hyfydy are modeled through forces that mimic the effect of cartilage and ligaments. This allows for customizable joint stiffness and damping, and closed kinematic chains. Hyfydy is optimized to run at high simulation frequencies (>3000Hz) to efficiently handle stiff joint and contact constraint forces.
- **Intuitive model format** for easy model building and editing. A tool for converting OpenSim models is included (only a subset of OpenSim features is supported).

Hyfydy is currently available as a plugin for the open-source SCONE [2] simulation software. For more information on SCONE, please visit the SCONE website.

** Performance comparison between Hyfydy and OpenSim was based on a 10 second gait simulation using a reflex-based controller [3] with hill-type musculotendon dynamics [4] and non-linear contact forces [5]. Control parameters were optimized in SCONE [2] using 1000 generations of CMA-ES [6] on a 4-core i7 Intel CPU.*

1.1. Citation

If you use Hyfydy in a research project, please use the following citation:

```
@misc{Geijtenbeek2021Hyfydy,
   author = {Geijtenbeek, Thomas},
   title = {The {Hyfydy} Simulation Software},
   year = {2021},
   month = {11},
   url = {https://hyfydy.com},
   note = {\url{https://hyfydy.com}}
}
```

2. Model

2.1. File Format

Hyfydy employs an intuitive text-based model format (.hfd), which is designed to be easily edited or constructed by hand using a text editor. It consists of key-value pairs in the form `key = value`, curly braces `{ ... }` for groups and square brackets `[ ... ]` for arrays. Comments are supported through via `#` (one-line) or `/* */` (multi-line).

Example

```
model {
    name = my_model
    body {
        name = my_body
        mass = 3
        inertia = [ 1.5 1.5 1.5 ]
        pos = [ 0 1 0 ] # single-line comment
    }
    /* multi line
    comment */
}
```

2.2. Data Types

The `.hfd` file format uses several reoccurring data types, which are explained below.

2.2.1. string

Strings are used to identify components. Quotes are not required, unless an identifier contains whitespace or special characters (`{`, `}`, `[`, `]`, `\` or `=`):

```
name = my_name
another_name = "Name with special {characters}"
```

2.2.2. vector3

3D vectors are used to represent position, direction and linear / angular velocity. Values can be entered as an array, or using `x`, `y`, or `z` components:

```
position = [ 0 1 0 ]
velocity { x = 0 y = 1 z = 0 }
earth_gravity { y = -9.81 }
```

2.2.3. quaternion

Quaternions are used to represent rotations and orientations. They can be initialized using their w, x, y, z components directly, but also via an x-y-z Euler angle (recommended):

```
ori1 { z = 90 } # 90 degree rotation around z
ori2 = [ 45 0 90 ] # 45 degrees around x, then 90 degrees around z
ori3 { w = 1 x = 0 y = 0 z = 0 }
```

2.2.4. range

Ranges are used to describe any kind boundary, and are written as two numbers separated by two dots (`..`):

```
joint_limits { x = -10..10 y = 0..0 z = -45..90 }
more_joint_limits = [ -10..10 0..0 -45..90 ]
dof_range = 0..90
```

2.3. Core Components

2.3.1. model

The top-level `model` component is mostly a container for other components. A typical model looks like this:

```
model {
    name = my_model
    gravity = [ 0 -9.81 0 ]
    material { ... }
    model_options { ... }

    body { ... }
    joint { ... }
    geometry { ... }
    ...
}
```

A `model` can contain the following properties:

Identifier	Type	Description	Default
`name`	string	Name of the model	*empty*
`gravity`	vector3 [m/s^2]	Acceleration due to gravity	`[0 -9.81 0]`

Other components, including material and model_options, are described below.

2.3.2. body

Bodies are specified using the `body` component. They contain mass properties, position/orientation as well as linear and angular velocity. The body frame-of-reference always has its origin located at the center-of-mass and its axes must be aligned with the principle axes of inertia of the body.

A `body` component can contain the following properties:

Identifier	Type	Description	Default
`name`	string	Name of the body	*empty*
`mass`	number [kg]	Body mass	*required**
`inertia`	vector3	Diagonal components of the inertia matrix	*required**
`density`	number [kg/m^3]	Density used to calculate the body mass and inertia, together with shape	*required**
`shape`	Shape	Shape used to calculate the body mass and inertia, together with density	*required**
`pos`	vector3 [m]	Initial position of the body center-of-mass	*zero*
`ori`	quaternion	Initial orientation of the body	*identity*
`lin_vel`	vector3 [m/s]	Initial linear velocity of the body center-of-mass	*zero*
`ang_vel`	vector3 [rad/s]	Initial angular velocity of the body	*zero*

* A body must contain *either* `mass` and `inertia`, *or* `density` and `shape` in order to have valid mass properties.

Example

```
body {
    name = my_body
    mass = 3
    inertia = [ 1.5 1.5 1.5 ]
    pos = [ 0 1 0 ]
    ori = [ 0 0 45 ]
}
```

When specifying a `density` and `shape` and property, the mass and inertia are calculated automatically. Supported shape types are:

- `sphere` (with `radius` parameter)
- `cylinder` and `capsule` (with `radius` and `height` parameters)
- `box` (with `half_dim` parameter)

Example

```
body {
    name = my_cube
    density = 1000
    shape {
        type = box
        half_dim = [ 0.1 0.1 0.1 ]
    }
    pos = [ 0 1 0 ]
}
```

Note: a `body` component does not include geometry used for contact detection and response. These are defined separately via a geometry component.

2.3.3. joint

Joint components constrain the motion between two bodies. They can contain the following properties:

Identifier	Type	Description	Default
`name`	string	Name of the joint	*empty*
`parent`	string	Name of the parent body	*required*
`child`	string	Name of the child body	*required*
`pos_in_parent`	vector3 [m]	Position of the joint in the parent body frame-of-reference	*required*
`pos_in_child`	vector3 [m]	Position of the joint in the child body frame-of-reference	*required*
`ref_ori`	quaternion	Reference orientation of the child body wrt the parent body	*identity*
`stiffness`	number [N/m]	Stiffness property of the joint constraint force	model_options
`damping`	number	Damping property of the joint constraint force	model_options
`limits`	range [deg]	Rotational joint limits, expressed as a vector3 of ranges	*none*
`limit_stiffness`	number [Nm/rad]	Stiffness property of the joint limit force	model_options
`limit_damping`	number	Damping property of the joint limit force	model_options

There are no restrictions to the amount of joints a body can contain, and **kinematic loops are allowed**. However, adding superfluous joints will impact simulation performance.

Example

```
joint {
    name = knee_r
    parent = femur_r
    child = tibia_r
    pos_in_parent = [ 0 -0.226 0 ]
    pos_in_child = [ 0 0.1867 0 ]
    limits { x = 0..0 y = 0..0 z = -90..0 }
}
```

Alternatively, a `joint` component can be specified *inside* a body component, in which case the `child` property can be omitted:

```
body {
    name = tibia_r
    mass = 3.7075
    inertia { x = 0.0504 y = 0.0051 z = 0.0511 }
    joint {
        name = knee_r
        parent = femur_r
        child = tibia_r
        pos_in_parent = [ 0 -0.226 0 ]
        pos_in_child = [ 0 0.1867 0 ]
        limits { x = 0..0 y = 0..0 z = -90..0 }
    }
}
```

2.3.4. joint_6dof

The `joint_6dof` is a joint type that has translation limits in addition to rotation limits. Otherwise, it supports the same features as the regular `joint`. Example:

```
joint_6dof {
    name = sliding_knee_r
    parent = femur_r
    child = tibia_r
    pos_in_parent = [ 0 -0.226 0 ]
    pos_in_child = [ 0 0.1867 0 ]
    limits { x = 0..0 y = 0..0 z = -90..0 }
    translation_limits { x = 0..0 y = -0.05..0.05 z = 0..0 }
}
```

2.3.5. geometry

The `geometry` component defines the properties needed for determining contacts and subsequent contact forces. They can contain the following properties:

Identifier	Type	Description	Default
`name`	string	Name of the model	*empty*
`type`	Shape	Shape of the geometry	*required*
`body`	string	Name of the body to which the geometry is attached	*required*
`material`	string	Name of the material associated with the contact geometry	*default*
`pos`	vector3 [m]	Position of the geometry in the body frame-of-reference	`[0 0 0]`
`ori`	quaternion	Orientation of the geometry relative to the body	*identity*

Supported shape types for geometry are:

- `sphere` (with `radius` parameter)
- `capsule` (with `radius` and `height` parameters)

- `box` (with `half_dim` parameter)
- `plane` (with `normal` parameter)

An example of a geometry:

```
geometry {
    name = l_heel
    type = sphere
    radius = 0.03
    body = calcn_l
    pos { x = -0.085 y = -0.015 z = 0.005 }
    ori { x = 0 y = 0 z = 0 }
}
```

Alternatively, geometry components can be specified *inside* a body component, in which case the `body` property can be omitted. The shape of the geometry can also be used to automatically calculate the mass properties.

2.3.6. material

The `material` component describes material properties used to compute contact forces. They contain the following properties:

Identifier	Type	Description	Default
`name`	string	Name of the model	*empty*
`static_friction`	number	Value used to calculate the static friction force	`1`
`dynamic_friction`	number	Value used to calculate the dynamic friction force	`static_friction`
`stiffness`	number	Stiffness of the restitution force	`1e6`
`damping`	number	Damping of the restitution force	`1`

Note: the interpretation of the friction, stiffness and damping parameters depend on the type of contact force that is used in the simulation.

2.3.7. model_options

The properties defined in `model_options` are global defaults that apply to all components defined within the model. Their goal is to minimize duplication of common properties. The following properties can be specified within a `model_options` section:

Identifier	Type/Unit	Description	Default
`mirror`	boolean	Indicate whether components should be mirrored (0 or 1)	`0`
`scale`	number	Value by which the model is scaled	`1`
`density`	number [kg/m^3]	Default density used for bodies	`1000`
`joint_stiffness`	number [N/m]	Default stiffness for joints	`0`
`joint_damping`	number	Default damping for joints	`1`
`damping_mode`	choice	The way in which default damping is computed	`critical_mass`
`joint_limit_stiffness`	number [Nm/rad]	Default limit stiffness for joints	`0`
`joint_limit_damping`	number	Default limit damping for joints	`0`
`limit_damping_mode`	choice	The way in which default limit damping is computed	`critical_inertia`
`muscle_force_multiplier`	number	Factor by which muscle `max_isometric_force` is multiplied	`1`

2.4. Actuators

2.4.1. point_path_muscle

The `point_path_muscle` component specifies a musculotendon unit which path is defined through a series of via points. It contains the following properties:

Identifier	Type	Description	Default
`name`	string	Name of the model	*empty*
`max_isometric_force`	number [N]	Maximum isometric force of the musculotendon unit	*required*
`optimal_fiber_length`	number [m]	Optimal fiber length of the musculotendon unit	*required*
`tendon_slack_length`	number [m]	Tendon slack length of the musculotendon unit	*required*
`pennation_angle`	number [rad]	Pennation angle at optimal fiber length, **in radians**	`0`
`stiffness_multiplier`	number	Multiplier applied to passive tendon an muscle elastic forces	`1`

Note: the way in which muscle force is computed depends on the muscle force that is used in the simulation.

Example

```
point_path_muscle {
    name = hamstrings_r
    tendon_slack_length = 0.31
    optimal_fiber_length = 0.109
    max_isometric_force = 2594
    pennation_angle = 0
    path [
        { body = pelvis pos { x = -0.05526 y = -0.10257 z = 0.06944 } }
        { body = tibia_r pos { x = -0.028 y = 0.1667 z = 0.02943 } }
        { body = tibia_r pos { x = -0.021 y = 0.1467 z = 0.0343 } }
    ]
}
```

2.4.2. joint_point_path_muscle

The `joint_point_path_muscle` is identical to a point path muscle, with the exception that with these types of muscles, joint torques are applied instead of forces. The advantage of applying a torque is that it leads to less joint displacement, which in turn can improve performance. Applying torques instead of forces also makes the simulation more similar to reduced coordinate simulation engines, such as OpenSim. If instead accurate computation of joint displacement caused by muscle force is required, this type of component is not recommended.

The properties of the `joint_point_path_muscle` are identical to those of a point_path_muscle.

2.4.3. joint_motor

Joint motors produce a 3D joint torque based on joint angle and joint velocity. The amount of torque τ is based on base torque τ_o, stiffness k_p, damping k_d, orientation q, target orientation q_t, angular velocity ω, and target velocity ω_t:

$$\tau = \left[\tau_o + k_p r(q^{-1} q_t) + k_d(v_t - v)\right]^{\tau_{max}}$$

The notation $[\,]^{\tau_{max}}$ is used to indicate that the magnitude of the final torque is clamped between $[-\tau_{max}, \tau_{max}]$. The function $r : \mathbb{R}^4 \rightarrow \mathbb{R}^3$ converts a quaternion to a 3D *rotation vector*, which corresponds to the 3D *rotation axis* scaled by *rotation angle* in radians.

A `joint_motor` component can contain the following properties:

Identifier	Type	Description	Default
`joint`	string	name of the joint	*required*
`stiffness`	number [Nm/rad]	Stiffness for position-dependent torque (k_p)	`0`
`damping`	number [Ns/rad]	Damping for velocity-dependent torque (k_d)	`0`
`max_torque`	number [Nm]	Maximum torque magnitude that can be applied by the motor (τ_{max})	+infinity
`target_ori`	quaternion	Target orientation for the motor (q_t)	*identity*
`target_vel`	vector3 [rad/s]	Target angular velocity (v_t)	`[0 0 0]`
`torque_offset`	vector3 [Nm]	Base torque (τ_o)	`[0 0 0]`

Example

```
joint_motor {
    joint = hip_r
    stiffness = 1000
    damping = 10
    max_torque = 100
    target_ori = [ 0 0 90 ]
    target_vel = [ 0 0 0 ]
    torque_offset = [ 0 0 0 ]
}
```

Modifying joint_motor targets

Joint motors can be modified during the simulation, allowing them to be part of complex control strategies with shifting targets and other properties. In SCONE, joint motor components can be altered through the script interface, via the functions:

- `set_motor_target_ori( quaternion )`
- `set_motor_target_vel( vector3 )`
- `set_motor_stiffness( number )`
- `set_motor_damping( number)`
- `add_motor_torque( vector3 )`.

2.5. Auxiliary Components

Auxiliary components are not part of the simulation and therefore do not affect the outcome, but can be used by external software for analysis and visualization.

2.5.1. mesh

Components of type `mesh` can be used by client applications for visualizing bodies. In SCONE, `mesh` components are visualized in the 3D viewer window. They can contain the following properties:

Identifier	Type	Description	Default
`file`	string	Mesh filename	*empty*
`shape`	Shape	Mesh shape	*empty*
`pos`	vector3 [m]	Position of the mesh in the body reference frame	`[0 0 0]`
`ori`	quaternion	Orientation of the mesh in the body reference frame	*identity*
`color`	color	Color of the mesh, in format `[r g b a]`	*unspecified*

Example

```
mesh {
    file = example.obj
    pos = [ 0 0.1 0.1 ]
    ori = [ 0 90 0 ]
}
```

```
mesh {
    shape {
        type = cylinder
        height = 0.5
        radius = 0.05
    }
    pos = [ 0 0 0 ]
    ori = [ 0 0 0 ]
    color = [ 0.8 0.8 0.8 ]
}
```

```
mesh {
    shape {
        type = box
        dim = [ 0.3 0.1 0.1 ]
    }
    color = [ 0.8 0.8 0.8 ]
}
```

2.5.2. dof

Components of type `dof` can be used to describe a specific degree-of-freedom, which can be used by a client application for control and analysis. In SCONE, `dof` components are used to define model coordinates, which are used for analysis and control.

In Hyfydy, all bodies contain 6 degrees-of-freedom (3 translational and 3 rotational) – specifying specific degrees-of-freedom using a `dof` component does not affect the simulation.

The `dof` component can contain the following properties:

Identifier	Type	Description	Default
`name`	string	Name of the degree-of-freedom	*required*
`source`	string	Default name of the dof. This consists of the name of the body appended by `_rx`, `_ry` or `_rz` for rotation relative to its parent, or `_tx`, `_ty` and `_tz` for translation of a root body. Dof values can be inverted by prepending the source with a minus sign (`-`).	*required*
`default`	number [m or deg]	Default value of the dof	`0`
`range`	range [m or deg]	Minimum and maximum values of the dof. Note that these values are not enforced during the simulation, use a joint limit force for that instead.	`-90..90`

In Hyfydy, all bodies contain 6 degrees-of-freedom – 3 translational and 3 rotational. In practice, joints limit their movement…

```
dof {
    name = pelvis_tilt
    source = pelvis_rz
    default = 0
    range = -90..90
}
```

3. Forces

The motion of bodies is governed by forces and torques, which cause linear and angular accelerations on individual bodies. In Hyfydy, joint constraints are too enforced through explicit constraint forces. Each force is generated by a specific *force component*, which can be configured flexibly and at run-time, via a configuration script. It is possible to distinguish between different types of forces, including joint forces, contact forces, actuator forces and external forces.

Example

```
composite_force {
    planar_joint_force_pnld {}
    simple_collision_detection {}
    contact_force_hunt_crossley_sb { transition_velocity = 0.1 }
    muscle_force_m2012fast { xi = 0.1 }
}
```

Multiple force component specified, actual forces are applied in order of specification. Each individual force component has its own properties.

3.1. Joint Forces

Joint forces hold bodies together, like ligaments and cartilage do in human and animal joints. In addition, ligaments and bony structures also limit the range of motion between bodies – these forces become active after a joint rotates beyond a specific threshold. In Hyfydy, the former set of forces is referred to as *joint constraint forces*, while the latter is referred to as *joint limit forces*. Both joint constraint and joint limit forces are produces by a single force component.

3.1.1. joint_force_pnld

For each joint j, the `joint_force_pnld` component applies a force F_j based on joint displacement p_j and displacement velocity v_j:

$$F_j = -k_p p_j - k_v v_j$$

The force F_j is applied in opposite directions to the child and parent body of the joint, at their respective `pos_in_child` and `pos_in_parent` position. See joint for more details.

In addition to a joint constraint force, `joint_force_pnld` also applies a joint limit torque τ_j based on joint angle α_j and angular velocity ω_j. While the limit torque is proportional to joint displacement angle θ_j, the limit damping torque is non-linear. This prevents damping to become active immediately after a joint angle crosses its limit. Instead, Hyfydy follows similar considerations as for the Hunt-Crossley contact restitution force [5], and scales the limit damping by the limit displacement torque:

$$\tau_j = -k_\alpha \theta_j(\alpha_j, \alpha_{min}, \alpha_{max})(1 + k_\omega \omega_j)$$

in which the joint displacement angle θ_j is defined as:

$$\theta_j(\alpha_j, \alpha_{min}, \alpha_{max}) = \begin{cases} \alpha_j - \alpha_{min}, & \text{if } \alpha_j < \alpha_{min} \\ \alpha_j - \alpha_{max}, & \text{if } \alpha_j > \alpha_{max} \\ 0, & \text{otherwise} \end{cases}$$

The constants k_p, k_d, k_α and k_ω correspond to the `stiffness`, `damping`, `limit_stiffness` and `limit_damping` parameters, which can be specified individually per joint, or via defaults based on model_options. As a result, `joint_force_pnld` does not contain any options for itself and can simply be added as:

```
joint_force_pnld {}
```

3.1.2. joint_force_pd

This force is similar to `joint_force_pnld`, with the exception that the limit damping torque is linearly proportional to the angular velocity ω_j:

$$\tau_j = -k_\alpha \theta_j(\alpha_j, \alpha_{min}, \alpha_{max}) - k_\omega \omega_j$$

This damping component is active immediately after a joint angle crosses its limit angle, resulting in an immediate torque in the opposite direction. As a result, using `joint_force_pnld` is recommended instead.

3.1.3. Planar Joint Forces

All joint forces have *planar* variants, which generate forces only in the *x-y plane*, and torques only around the perpendicular *z-axis*. Planar forces greatly improve simulation performance for planar (2D) models.

The force component names of the planar joint forces are as follows:

Joint force component	Planar joint force equivalent
`joint_force_pnld`	`planar_joint_force_pnld`
`joint_force_pd`	`planar_joint_force_pd`

Important: planar forces should always be used in combination with a planar integrator.

3.2. Contact Forces

Contact forces are applied when the geometries of two bodies intersect. A contact force consists of a *restitution* and *friction* component, which are computed based on the penetration depth, the relative velocity of the bodies at the point of contact, and the `material` properties of the contact. The computation of contact forces consists of two phases:

1. **Collision detection**. Geometries are checked for intersections, and contacts between geometries are recorded.
2. **Collision response**. Restitution and friction forces are applied to colliding bodies, based on recorded contacts.

In Hyfydy, each phase is configured using a separate force component, even though technically, the collision detection component only generates contacts and no forces.

Important: both a collision detection and collision response force must be present in order for contact forces to work.

3.2.1. Collision Detection

Collision detection components detect intersections between the geometries of pairs of bodies. Therefore, it is required to have a `geometry` attached to at least two different bodies in order for collision detection to work.

3.2.1.1. simple_collision_detection

The `simple_collision_detection` component detects intersections by going through all relevant pairs of objects. While this is a reasonable strategy when not many geometries are present (as is the case with typical biomechanics simulations), efficiency quickly declines when the number of bodies increases. Therefore, it is important to keep the number of geometries at a minimum when using the `simple_collision_detection` component.

By default, the component is configured to only detect intersections between a static plane geometry and geometries attached to bodies – and not intersections between different bodies. This behavior can be changed by setting the `enable_collision_between_objects` property to 1.

Important: setting `enable_collision_between_objects = 1` when many geometries (>10) are present severely impacts simulation performance.

The following properties can be set for `simple_collision_detection`:

Identifier	Type	Description	Default
`enable_collision_between_objects`	bool	When set to 1, intersections between each pair of geometries are detected.	0

Example

```
simple_collision_detection { enable_collision_between_objects = 1 }
```

3.2.2. Combining Material Properties

When a collision is detected and a contact is generated, the material properties of both geometries are combined into a new set of parameters, which in turn are used by the collision response force.

Material Property	Hyfydy	Simbody
`stiffness` (k_1, k_2)	$\frac{k_1 k_2}{k_1+k_2}$	$\frac{k_1 k_2}{k_1+k_2}$
`damping` (c_1, c_2)	$\frac{c_1+c_2}{2}$	$\frac{k_1}{k_1+k_2}c_1 + \frac{k_2}{k_1+k_2}c_2$
`static_friction`, `dynamic_friction` (μ_1, μ_2)	$\sqrt{\mu_1 \mu_2}$	$2\frac{\mu_1 \mu_2}{\mu_1+\mu_2}$

3.2.3. Collision Response

Collision response components compute actual contact forces, based on the contact found during the collision detection phase.

3.2.3.1. contact_force_pd

The `contact_force_pd` produces a linear damped spring contact restitution force, also known as the Kelvin-Voigt contact model. Given the material stiffness coefficient k and dissipation coefficient c, the normal force F_n is derived from penetration depth d and normal velocity v_n, which is the contact velocity in the direction of contact normal $\vec{n}$:

$$F_n = kd - cv_n$$

The tangent force $\vec{F}_t$ depends on the static friction coefficient μ, `viscosity` parameter η and tangent velocity $\vec{v}_t$:

$$F_t = \frac{-\vec{v}_t}{\|\vec{v}_t\|} \min(\eta\|\vec{v}_t\|, \mu F_n)$$

The total contact force $\vec{F}_c$ corresponds to:

$$\vec{F}_c = F_n\vec{n} + \vec{F}_t$$

This force has no extra parameters and can be added by including:

```
contact_force_pd { viscosity = 1000 }
```

3.2.3.2. contact_force_hunt_crossley

The non-linear Hunt-Crossley [5] contact force provides a better model of the dependence of the coefficient of restitution on velocity, based on observed evidence. Given material stiffness k, damping c, penetration depth d and normal velocity v_n the contact restitution force F_n corresponds to:

$$F_n = kd^{\frac{3}{2}}(1 - \frac{3}{2}cv_n)$$

The friction force $\vec{F}_t$ and resulting contact force $\vec{F}_c$ are defined as in contact_force_pd.

3.2.3.3. Contact Stiffness Conversion

Traditionally, the Hunt-Crossley stiffness depends on the radius of the contact sphere. In Hyfydy, however, the Hunt-Crossley sphere radius is incorporated into the stiffness constant, making the stiffness only depent on penetration depth. The stiffness constant is updated automatically as part of the conversion from OpenSim to Hyfydy. For any contact sphere radius r, the relation between contact stiffness for OpenSim (k_{osim}) and Hyfydy (k_{hfd}) is as follows:

$$k_{hfd} = \left(\frac{4}{3}\sqrt{r}k_{osim}\right)^{\frac{2}{3}} \quad , \quad k_{osim} = \frac{3}{4\sqrt{r}}(k_{hfd})^{\frac{3}{2}}$$

3.2.3.4. contact_force_hunt_crossley_sb

Similar to `contact_force_hunt_crossley`, but with a friction model attributed to Michael Hollars and used by Simbody and OpenSim. This force introduces the `transition_velocity` property, which is described here. Lowering the transition velocity increases accuracy, at the cost of simulation performance as a result of requiring smaller integration time steps.

Identifier	Type	Description	Default
`transition_velocity`	number [m/s]	Transition velocity	`0.1`

Example

```
contact_force_hunt_crossley_sb {
    transition_velocity = 0.2
}
```

3.3. Actuator Forces

3.3.1. muscle_force_m2012fast

This is an optimized implementation of the Hill-type muscle model described by Millard et al. [4], which includes a passive damping term that allows velocity to be determined even when the muscle is deactivated. This is the **recommended muscle model** to be used in Hyfydy.

In order to increase performance, the Hyfydy implementation of the Millard Equilibrium Muscle Model [4] differs from the OpenSim implementation in two significant ways:

1. The curves that describe the force-length and force-velocity relations, as well as the curves for passive tendon and muscle forces, are defined through polynomials instead of splines. The resulting curves in Hyfydy therefore differ slightly from the curves used in the OpenSim implementation.

2. The muscle damping forces are approximated using an explicit method, instead of using an iterative method as suggested in the publication and used in the OpenSim implementation. While this approach greatly increases performance, it causes the resulting muscle damping forces in Hyfydy to differ slightly from the damping the forces produced in the OpenSim implementation.

The `muscle_force_m2012fast` can include the following properties:

Identifier	Type	Description	Default
`xi`	number	Muscle damping factor, in the paper referred to as β	0.1
`v_max`	number [optimal_fiber_length/s]	Maximum muscle contraction velocity	10
`use_pennation_during_equilibration`	bool	Use pennation angle during muscle equilibration	0

The tendon force F_T is defined as:

$$F_T(\epsilon) = F_{max}(\mathrm{c_{T1}}\epsilon^2 + \mathrm{c_{T2}}\epsilon)$$

in which F_{max} is the maximum isometric force of the muscle, $\mathrm{c_{T1}} = 260.972$, $\mathrm{c_{T2}} = 7.9706$, and ϵ is the tendon strain, based on tendon length l_t and tendon slack length l_s:

$$\epsilon = \begin{cases} \frac{l_t - l_s}{l_s} & \text{if } l_t > l_s \\ 0 & \text{otherwise} \end{cases}$$

The muscle force F_M is defined as:

$$F_M(a, l, v) = F_{max}\left(a f_L(l) f_V(v) + f_P(l) - \beta v\right)$$

in which a is the muscle activation, l is the normalized fiber length, v is the normalized fiber velocity, and β is the damping factor.

The active force-length multiplier is f_L is defined as:

$$f_L(l) = \begin{cases} \mathrm{c_{L1}}(l-1)^3 + \mathrm{c_{L2}}(l-1)^2 + 1 & \text{if } r_1 < l < r_2 \\ 0 & \text{otherwise} \end{cases}$$

in which $\mathrm{c_{L1}} = 1.5$, $\mathrm{c_{L2}} = -2.75$, $r_1 = 0.46899$ and $r_2 = 1.80528$.

The active force-velocity multiplier f_V is defined as:

$$f_V(v) = \begin{cases} 0 & \text{if } v \leq -1 \\ \frac{\mathrm{c_{V1}}(v+1)}{\mathrm{c_{V1}} - v} & \text{if } -1 < v < 0 \\ \frac{F_{vmax}v + \mathrm{c_{V2}}}{\mathrm{c_{V2}} + v} & \text{if } v \geq 0 \end{cases}$$

in which v is the normalized fiber velocity, $\mathrm{c_{V1}} = 0.227$, $\mathrm{c_{V2}} = 0.110$ and $F_{vmax} = 1.6$.

Finally, the passive force-length multiplier f_P used to compute the parallel elastic force is defined as:

$$f_P(l) = \begin{cases} \mathrm{c_{P1}}(l-1)^3 + \mathrm{c_{P1}}(l-1)^2 & \text{if } l > 1 \\ 0 & \text{otherwise} \end{cases}$$

in which $\mathrm{c_{P1}} = 1.08027$ and $\mathrm{c_{P2}} = 1.27368$.

3.3.2. muscle_force_gh2010

This is an implementation of the Hill-type muscle model described by Geyer & Herr [3]. It includes a passive spring that prevents the muscle from shortening after a specific length threshold. The force-velocity relationship is undefined at zero activation, as a result activation muscle be kept above a threshold (e.g. > 0.01) to ensure stability.

3.3.3. muscle_force_tj2003

This is an implementation of the muscle model described in a document authored by Chand T. John, which describes the implementation in OpenSim of the muscle model published by Thelen [7]. Despite its popularity, this model is best avoided, because its force-velocity relationship is poorly defined at low activation and relies heavily on its ad-hoc extrapolation of the force-velocity curve. We recommend using the `muscle_force_m2012fast` model instead.

3.3.4. joint_motor_force

The joint_motor_force component produces joint torques defined by joint_motor components, and needs to be included for models that use them. They contain no additional settings.

```
joint_motor_force {}
```

3.4. External Forces

External forces include gravity and perturbation. These forces are applied to automatically when defined in the model.

3.4.1. gravity

Gravity is applied automatically to all non-static bodies in the model. The magnitude and direction is determined by the `gravity` property defined inside the model. To disable gravity, simply add `gravity = [ 0 0 0 ]`.

3.4.2. perturbation

Perturbation forces are applied automatically based on `perturbation` components defined inside the model. Since perturbations usually change over time, `perturbation` components are typically not defined in the model itself, but are added by a client application.

In SCONE, perturbations and perturbation forces are enabled automatically when after specifying `enable_external_forces = 1` in the SCONE Model configuration.

4. Integrators

4.1. Overview

Integrators advance the simulation by updating the velocity and position/orientation of each body, based on the linear and angular acceleration that is the result of the sum of all forces. Formally, an integrator I_h updates a set of generalized coordinates q_t with derivatives $\dot{q}_t$ at time t based on accelerations $\ddot{q}_t$ and step size h:

$$\{q_{t+h}, \dot{q}_{t+h}\} = I_h(q_t, \dot{q}_t, \ddot{q}_t, h)$$

The integrator I_h is referred to as a *fixed-step integrator*: during each step, the state is advanced with with a constant time interval. While this approach is common among physics simulation engines, it leaves the difficulty of choosing the step-size h to the user. Step-size that are too small are inefficient, while step-sizes that are too large cause the simulation to become unstable. In practice, the optimal step-size often varies greatly during the course of a simulation, depending on the numerical stiffness at any point in time.

Variable-step integrators circumvent that issue by adapting the step-size to the current state of simulation. A variable-step integrator I_A computes the step-size h based on a set of accuracy requirements A:

$$\{h, q_{t+h}, \dot{q}_{t+h}\} = I_A(q_t, \dot{q}_t, \ddot{q}_t, A)$$

Variable-step integrators are an important part of Hyfydy, and allow for a stable and efficient trade-off between performance and accuracy.

4.2. Integration Methods

4.2.1. Forward Euler

The most straightforward integration method is the Forward Euler method, which updates positions q_t using velocities $\dot{q}_t$, which in turn are updated using accelerations $\ddot{q}_t$:

$$\begin{aligned} q_{t+h} &= q_t + h\dot{q}_t \\ \dot{q}_{t+h} &= \dot{q}_t + h\ddot{q}_t \end{aligned}$$

The downside of this approach is that the positions are updated using a poor estimate of the velocity during interval h. This can lead to an accumulation of errors and energy being added to system.

4.2.2. Symplectic Euler

The Symplectic Euler or semi-implicit Euler method improves over the Forward Euler method by updating velocities first and using the updated velocities $\dot{q}_{t+h}$ to update positions q_t:

$$\begin{aligned} \dot{q}_{t+h} &= \dot{q}_t + h\ddot{q}_t \\ q_{t+h} &= q_t + h\dot{q}_{t+h} \end{aligned}$$

This approach leads to increased stability and energy conservation over the Forward Euler method. However, the position update is still based on a poor estimate of the velocity during the step (even with constant acceleration), and as such this method does not improve accuracy.

4.2.3. Midpoint Euler

The Midpoint Euler method uses the average or midpoint velocity between to update the position, leading to a better position estimate:

$$\begin{aligned} \dot{q}_{t+h} &= \dot{q}_t + h\ddot{q}_t \\ q_{t+h} &= p_t + \frac{h}{2}(\dot{q}_t + \dot{q}_{t+h}) \end{aligned}$$

With this approach, both position and velocity updates are most accurate if acceleration $\ddot{q}_t$ remains constant over interval h. Despite being more accurate, it is typically outperformed by the Symplectic Euler method in terms of stability and energy conservation. However, in combination with a error-controlled variable-step integration strategy, the increased accuracy can be considered more important.

4.2.4. Higher-order Methods

In contrast to the first-order integrators described above, high-order integrators attempt to increase accuracy by evaluating the forces and resulting accelerations at different points in time t.

This section is under construction

4.3. Muscle Activation Dynamics

In addition to integrating body position and velocity, integrators in Hyfydy also handle muscle activation dynamics. The muscle activation and deactivate are properties that are part of the integrator:

Identifier	Type	Description	Default
`activation_rate`	number [s^{-1}]	The muscle activation rate	100
`deactivation_rate`	number [s^{-1}]	The muscle deactivation rate	25

Given muscle activation a_t and excitation u_t at time t, then activation is updated to a_{t+h} according to:

$$a_{t+h} = a_t + h(u_t - a_t)(c_1 u_t + c_2)$$

in which `activation_rate` corresponds to $c_1 + c_2$, while `deactivation_rate` corresponds to c_2.

4.4. Planar Integrators

Planar integrators only update positions in the *x-y plane*, and orientations in around the perpendicular *z-axis*. When using *planar models* with *planar forces*, these types of integrators increase simulation performance without loss of accuracy. See Integrator Configuraton for more details.

4.5. Integrator Configuration

In Hyfydy, the integrator can be configured via a configuration file. In SCONE, this configuration is directly added to the Model.

Integrator	Description	Properties
`forward_euler_integrator`	Fixed-step integrator using the Forward Euler method	
`symplectic_euler_integrator`	Fixed-step integrator using the Symplectic Euler method	
`midpoint_euler_integrator`	Fixed-step integrator using the Midpoint Euler method	
`planar_symplectic_euler_integrator`	Planar version of `symplectic_euler_integrator`	

This section is under construction

5. Comparisons

5.1. Hyfydy vs OpenSim

Even though Hyfydy is based on the same models as OpenSim [1], there are some differences in implementation that result in slightly different simulation outcomes. As a consequence, results from a Hyfydy simulation are not directly transferable to OpenSim, and vice-versa. While this does not seem ideal, it is important to stress that both simulators are equally valid, and it usually only takes a small number of optimization iterations to convert from one result to another. In the remainder of this section, we provide an overview of all significant differences between Hyfydy and OpenSim.

Joint constraints

OpenSim uses generalized or reduced coordinates to describe the positions and velocities of the articulated bodies in the system, which automatically enforces joint constraints. In Hyfydy, each body has six degrees of freedom, and joint constraints are enforced explicitly through forces that mimic the effect of cartilage, bony structures and tendons. This results in small displacements within joints – just as in biological joints. The amount of displacement is based on the joint stiffness and damping settings, which can be set individually per joint or computed automatically.

Joint limits

Hyfydy uses non-linear damping forces to enforce joint limits, where the amount of damping is proportional to the limit force. In contrast, the OpenSim `CoordinateLimitForce` uses a constant damping coefficient beyond the limit threshold. The behavior in Hyfydy is similar to damping in Hunt-Crossley contact forces [5], and has similar benefits over the OpenSim `CoordinateLimitForce`.

Friction force

The friction force in the OpenSim `HuntCrossleyFroce` is based on an unpublished model attributed to Michael Hollars. Hyfydy provides an exact implementation of this model via `contact_force_hunt_crossly_sb`, which can be used to emulate the behavior of OpenSim friction.

Muscle force

Hyfydy uses an optimized implementation of the Millard Equilibrium Muscle Model [4], which differs from the OpenSim implementation in two significant ways:

1. The curves that describe the force-length and force-velocity relations, as well as the curves for passive tendon and muscle forces, are defined through polynomials instead of splines. The resulting curves in Hyfydy therefore differ slightly from the curves used in the OpenSim implementation.
2. The muscle damping forces are computed explicitly instead through the iterative method in the original OpenSim implementation. The resulting muscle damping forces in Hyfydy therefore differ slightly from the forces produced in the OpenSim implementation.

Muscle activation dynamics

Hyfydy uses a different approach for computing muscle activation dynamics than OpenSim. As a result, muscle activation patterns can differ between Hyfydy and OpenSim -- even when excitation patterns are identical. This difference only occurs when the deactivation rate is different from the activation rate.

Numeric Integration

OpenSim and Hyfydy both implement several variable-step integrators with user-configurable error control. In Hyfydy, the accuracy criterium is based on the *highest error* found across all bodies, while OpenSim uses a weighted sum of errors to determine accuracy. As a result, only in Hyfydy the error of each body is guaranteed to be below the specified accuracy threshold.

6. Version History

Version history will be documented after the 1.0.0 release.

Version	Date	Description

7. References

1. Seth, A., Hicks, J. L., Uchida, T. K., Habib, A., Dembia, C. L., Dunne, J. J., … Delp, S. L. (2018). OpenSim: Simulating musculoskeletal dynamics and neuromuscular control to study human and animal movement. PLoS Computational Biology, 14(7). https://doi.org/10.1371/journal.pcbi.1006223 ↩ ↩

2. Geijtenbeek, T. (2019). SCONE: Open Source Software for Predictive Simulation of Biological Motion. Journal of Open Source Software, 4(38), 1421. https://doi.org/10.21105/joss.01421 ↩ ↩

3. Geyer, H., & Herr, H. (2010). A muscle-reflex model that encodes principles of legged mechanics produces human walking dynamics and muscle activities. IEEE Transactions on Neural Systems and Rehabilitation Engineering, 18(3), 263–273. **https://doi.org/10.1109/TNSRE.2010.2047592 ↩ ↩

4. Millard, M., Uchida, T., Seth, A., & Delp, S. L. (2013). Flexing computational muscle: modeling and simulation of musculotendon dynamics. Journal of Biomechanical Engineering, 135(2), 021005. https://doi.org/10.1115/1.4023390 ↩ ↩ ↩ ↩

5. Hunt, K. H., & Crossley, F. R. E. (1975). Coefficient of Restitution Interpreted as Damping in Vibroimpact. Journal of Applied Mechanics, 42(2), 440. ↩ ↩ ↩ ↩

6. Hansen, N. (2006). The CMA evolution strategy: a comparing review. Towards a New Evolutionary Computation, 75–102. ↩

7. Thelen, D. G. (2003). Adjustment of muscle mechanics model parameters to simulate dynamic contractions in older adults. Journal of Biomechanical Engineering, 125(1), 70–77. https://doi.org/10.1115/1.1531112 ↩