Auton Robot (2017) 41:495–511
DOI 10.1007/s10514-016-9574-0

RBDL: an efficient rigid-body dynamics library using recursive algorithms

Martin L. Felis[1]

Received: 20 October 2014 / Accepted: 13 May 2016 / Published online: 2 June 2016

Abstract In our research we use rigid-body dynamics and optimal control methods to generate 3-D whole-body walking motions. For the dynamics modeling and computation we created RBDL—the Rigid Body Dynamics Library. It is a self-contained free open-source software package that implements state of the art dynamics algorithms including external contacts and collision impacts. It is based on Featherstone's spatial algebra notation and is implemented in C++ using highly efficient data structures that exploit sparsities in the spatial operators. The library contains various helper methods to compute quantities, such as point velocities, accelerations, Jacobians, angular and linear momentum and others. A concise programming interface and minimal dependencies makes it suitable for integration into existing frameworks. We demonstrate its performance by comparing it with state of the art dynamics libraries both based on recursive evaluations and symbolic code generation.

Keywords Reduced coordinates · Rigid-body dynamics · Jacobian · Contact · Software

1 Introduction

A lot of research of rigid multibody dynamics algorithms has been done over the last decades. The theory of multibody dynamics has been the subject of various textbooks such as (Khalil and Dombre 2004; Craig 2005; Jain 2011), and others. Various algorithms have been derived that solve fundamental dynamics related problems or compute various quantities very efficiently. Not only the theory but also the formulation and notation of dynamics has been improved. Traditionally the equation of motion is described using two 3-D equations, one for linear movement and one for rotational movement. When analyzing systems of connected bodies, this quickly results in a large amount of equations (Luh et al. 1980; Armstrong 1979) that are both difficult to read and transform into programming code. To circumvent this, alternate formulations based on screw theory (Ball 1900) have been proposed and proved to be very useful to analyze derive and formulate various algorithms (Featherstone 1983; Rodriguez 1987; Rodriguez et al. 1988; Jain and Rodriguez 1993). An overview of state of the art dynamics algorithms and a brief historical overview on dynamics algorithms has been composed in Featherstone and Orin (2000).

There are in general two classes of dynamics algorithms: maximal coordinate algorithms on the one side and reduced coordinates algorithms on the other side. Maximal coordinate approaches model each body individually and use joint constraint equations to restrict the relative motions of the bodies. The methods allow to handle certain closed loops systems more easily however the constraint equations, including joint constraints, may not be fulfilled at all times and cause separation of connected bodies. Well known implementations of this approach is ODE and Bullet that are commonly used in video games as well as in robotic applications.

Reduced (or generalized) coordinate algorithms usually assume multibody systems for which the topology can be described by a tree. These algorithms have the advantage to only operate on the actual degrees of freedom of the system that fulfill the joint constraints at all times. They operate on much smaller systems than the maximal coordinate approaches however the algorithms are usually more complex and difficult to implement. A well known algorithm that

✉ Martin L. Felis
martin.felis@iwr.uni-heidelberg.de

[1] Research Group Optimization in Robotics and Biomechanics Interdisciplinary Center for Scientific Computing (IWR), ML 100, Berliner Str. 45, 69120 Heidelberg, Germany

usually is described using reduced coordinates is the Recursive Newton–Euler algorithm. Simulation of systems subject to external contacts violate the tree topology assumption and require special treatment that further increase complexity when implementing these.

Dynamics algorithms can be implemented in two ways: by implementing the algorithms directly as computer code or by implementing them in a computer algebra software to formulate symbolic expressions of the result of these algorithms and convert the expressions to computer code. We refer to the former approach as recursive methods and the latter as symbolic code generation.

Symbolic code generation tools can easily prune out expressions that are zero. For planar models this will automatically remove all computations that are not within the plane the model is defined. Disadvantages of these approaches are that the symbolic expressions are reduced to scalar expressions which may not be exploited by modern CPUs. Further, the generation itself can be time consuming and the output is very complex and can only be used as black box. Whenever a change of the model is required one has to generate the code anew.

Recursive methods have the advantage that they are self-contained and do not need a computer algebra to create code for a new model. New methods can easily be implemented and used by existing models. The code can be analyzed and shared expressions can be extracted into separate functions such that redundant computations can be avoided. This approach leads to vector valued expressions in the computer code that allows for fast evaluations using SIMD (*Single Instruction, Multiple Data*) instructions present in today's CPUs.

The aim of this paper is twofold: first we give an overview of available state of the art algorithms for reduced coordinate rigid-body dynamics with external contacts together with formulations to compute quantities that are often needed in simulation and control, such as Jacobians, system angular momentum, and others. Second we present RBDL – the Rigid Body Dynamics Library, which is a mature, thoroughly tested, and highly efficient publicly available open-source implementation of all the presented algorithms and formulations based implemented as recursive methods. We demonstrate its performance by various benchmarks, including comparison to a state of the art dynamics library and code generation tool for models of different sizes.

The structure of the paper is the following: Sect. 2 describes the basic formulation of rigid-body dynamics of arbitrarily branched trees, also with external contacts. Section 3 gives a brief introduction to spatial algebra and Sect. 4 describes how we model multibody systems. The following Sect. 5 presents the state of the art algorithms: the Recursive Newton–Euler algorithm (RNEA) for inverse dynamics, the Composite Rigid-Body algorithm (CRBA) to compute the joint space inertia matrix, and the Articulated Body Algorithm (ABA) to compute the forward dynamics. In Sect. 6 we show how various quantities such as 3-D point velocities and accelerations, Jacobians, system angular momentum and others can be computed. In Sect. 7 we discuss implementation details of our open-source dynamics library RBDL and in Sect. 8 we evaluate the performance of our implementation with that of other implementations both using recursive methods and symbolic code generation.

2 Theoretical background

The theory of rigid-body dynamics is the focus of many textbooks (for a small selection refer to the introduction). In the following we borrow large parts of the description and notation used in Featherstone (2008).

2.1 Equation of motion for rigid multibody systems

For a given rigid multibody model with $n_{dof} \in \mathbb{R}^n$ degrees of freedom we use generalized coordinates to describe the state of the system. We use the symbols $\mathbf{q}(t), \dot{\mathbf{q}}(t), \ddot{\mathbf{q}}(t), \boldsymbol{\tau}(t) \in \mathbb{R}^{n_{dof}}$ to denote the generalized positions, generalized velocities, generalized accelerations, and generalized forces of the system at the current time $t \in \mathbb{R}$. Throughout this paper we consider the dynamics at the current instant and we therefore omit the argument t of the generalized variables.

The dynamics of rigid multibody systems using generalized coordinates is described by the equation:

$$\boldsymbol{H}(\boldsymbol{q})\ddot{\boldsymbol{q}} + \boldsymbol{C}(\boldsymbol{q}, \dot{\boldsymbol{q}}) = \boldsymbol{\tau} \tag{1}$$

The matrix $\boldsymbol{H}(\boldsymbol{q}) \in \mathbb{R}^{n_{dof} \times n_{dof}}$ is the symmetric and positive definite joint space inertia matrix or generalized inertia matrix. The term $\boldsymbol{C}(\boldsymbol{q}, \dot{\boldsymbol{q}}) \in \mathbb{R}^{n_{dof}}$ is the Coriolis term or generalized bias force and $\boldsymbol{\tau}$ the generalized force applied at the joints. For underactuated systems with $n_{actuated} < n_{dof}$ actuated degrees of freedom these generalized forces can be written as $\boldsymbol{\tau} = \boldsymbol{T}\boldsymbol{u}$, where $\boldsymbol{u} \in \mathbb{R}^{n_{actuated}}$ are the forces of the actuated joints and the matrix $\boldsymbol{T} \in \mathbb{R}^{n_{dof} \times n_{actuated}}$ maps those onto the generalized forces.

The generalized bias force (also sometimes called "nonlinear effects") is equal to the amount of generalized force that has to be applied to the system such that the resultant generalized acceleration $\ddot{\boldsymbol{q}}$ is zero. It also contains the forces due to Coriolis and centrifugal forces, gravity, and other forces acting on the system that are not caused by $\boldsymbol{\tau}$.

There are two fundamental dynamics problems related to rigid multibody systems: inverse dynamics and forward dynamics. Inverse dynamics computes the generalized forces $\boldsymbol{\tau}$ required to produce the given $\boldsymbol{q}, \dot{\boldsymbol{q}}, \ddot{\boldsymbol{q}}$:

$$\boldsymbol{\tau} = ID(\boldsymbol{q}, \dot{\boldsymbol{q}}, \ddot{\boldsymbol{q}}). \tag{2}$$

Forward dynamics computes the acceleration $\ddot{\boldsymbol{q}}$ of the rigid multibody system for given $\boldsymbol{q}, \dot{\boldsymbol{q}}, \boldsymbol{\tau}$. It can be formalized by:

$$\ddot{\boldsymbol{q}} = FD(\boldsymbol{q}, \dot{\boldsymbol{q}}, \boldsymbol{\tau}). \tag{3}$$

Both problems can be directly evaluated using (1), however for sufficiently large n_{dof} much more efficient recursive methods exist.

2.2 Equation of motion with external contacts

Equation (1) describes the dynamics of a rigid multibody system that is not subject to an external constraint. If one or more points of the model are in contact with the environment, then the motion of the model is constrained by these contacts. Additionally the contacts cause forces acting on the model. Modeling of contacts is a very complex topic in its own especially when taking unilateral contacts and friction into account (Pang and Trinkle 1996; Pfeiffer and Glocker 2008). In this work we restrict ourselves to bilateral holonomic scleronomic constraints without friction. However the quantities derived in this section play an important role for more general contacts. A bilateral contact can be expressed as

$$\boldsymbol{g}(\boldsymbol{q}) = 0. \tag{4}$$

For a point constraint that fixes a body point at a specific location the function $\boldsymbol{g}(\boldsymbol{q})$ expresses the difference of the point and its desired position. In this case $\boldsymbol{g}(\boldsymbol{q}) \in \mathbb{R}^3$ where the first entry would be the points X-coordinate, the second its Y-coordinate, and the last its Z-coordinate. If the constraint only acts along one axis for a single point then $\boldsymbol{g}(\boldsymbol{q}) \in \mathbb{R}$. If there are point constraints on n bodies then $\boldsymbol{g}(\boldsymbol{q}) \in \mathbb{R}^{3n}$.

The equation of motion for a rigid multibody model that is subject to external contacts is described by

$$\boldsymbol{H}(\boldsymbol{q})\ddot{\boldsymbol{q}} + \boldsymbol{C}(\boldsymbol{q}, \dot{\boldsymbol{q}}) = \boldsymbol{\tau} + \boldsymbol{G}(\boldsymbol{q})^T\boldsymbol{\lambda}, \tag{5a}$$
$$\boldsymbol{g}(\boldsymbol{q}) = \boldsymbol{0}, \tag{5b}$$

where $\boldsymbol{G}(\boldsymbol{q}) = \frac{d}{dt}\boldsymbol{g}(\boldsymbol{q})$ is the so-called *contact Jacobian* and $\boldsymbol{\lambda}$ is the contact force. Equation (5a) is a differential algebraic equation of (differential) index 3. By differentiating the constraint Eq. (5b) twice we obtain:

$$\boldsymbol{H}(\boldsymbol{q})\ddot{\boldsymbol{q}} + \boldsymbol{C}(\boldsymbol{q}, \dot{\boldsymbol{q}}) = \boldsymbol{\tau} + \boldsymbol{G}(\boldsymbol{q})^T\boldsymbol{\lambda}, \tag{6a}$$
$$\boldsymbol{G}(\boldsymbol{q})\ddot{\boldsymbol{q}} + \dot{\boldsymbol{G}}(\boldsymbol{q})\dot{\boldsymbol{q}} = \boldsymbol{0}, \tag{6b}$$

which we can rewrite as a linear system of the unknowns $\ddot{\boldsymbol{q}}, \boldsymbol{\lambda}$:

$$\begin{bmatrix} \boldsymbol{H}(\boldsymbol{q}) & \boldsymbol{G}(\boldsymbol{q})^T \\ \boldsymbol{G}(\boldsymbol{q}) & \boldsymbol{0} \end{bmatrix} \begin{bmatrix} \ddot{\boldsymbol{q}} \\ -\boldsymbol{\lambda} \end{bmatrix} = \begin{bmatrix} -\boldsymbol{C}(\boldsymbol{q}, \dot{\boldsymbol{q}}) + \boldsymbol{\tau} \\ \boldsymbol{\gamma}(\boldsymbol{q}, \dot{\boldsymbol{q}}) \end{bmatrix} \tag{7}$$

This system is always solvable if $\boldsymbol{G}(\boldsymbol{q})$ has full rank, which is the case if the constraints in $\boldsymbol{g}(\boldsymbol{q})$ are not redundant. The term $\boldsymbol{\gamma}(\boldsymbol{q}, \dot{\boldsymbol{q}})$ is the negative right summand of (6b) and is also called *contact Hessian* or sometimes *Jacobian derivative*.

To ensure equivalence of (5a) and (7) we have to ensure that the invariants of the constraints are fulfilled:

$$\boldsymbol{g}(\boldsymbol{q}) = \boldsymbol{0} \tag{8}$$
$$\boldsymbol{G}(\boldsymbol{q})\dot{\boldsymbol{q}} = \boldsymbol{0}. \tag{9}$$

It is sufficient to ensure that these conditions are fulfilled only at the beginning of the contact as due to (6b) and therefore (7) the condition will be already fulfilled on the acceleration level.

When using numerical integration equation (6b) will not be fulfilled exactly and hence errors will be accumulated eventually. This is especially true for large step sizes and/or long simulation durations. In this case one can use Baumgarte stabilization Ascher et al. (1994).

2.3 Collision impacts

The transition from a rigid-body system without contacts to a system that has contacts is called a contact gain. During a contact gain very high forces act in a very short time on the body. In the real world these high forces cause the body to first compress and then expand. After the compression phase, depending on the physical properties of the body, the body remains either in contact (perfect inelastic collision) or bounces off.

Advanced collision models consider the compression and expansion phase (Uchida et al. 2015), however it is often neglected and the contact gain is handled instantaneously using a collision. The physical properties of the body are described by the parameter of restitution $e \in [0, 1]$. For $e = 0$ the collision is a perfect inelastic collision, whereas for $e = 1$ the collision is perfectly elastic.

The contact gain is a discontinuous change in the generalized velocity variables from $\dot{\boldsymbol{q}}^-$ to $\dot{\boldsymbol{q}}^+$, i.e. the velocity *before* the collision to the velocity *after* the collision. The change can be computed using:

$$\begin{bmatrix} \boldsymbol{H}(\boldsymbol{q}) & \boldsymbol{G}(\boldsymbol{q})^T \\ \boldsymbol{G}(\boldsymbol{q}) & \boldsymbol{0} \end{bmatrix} \begin{bmatrix} \dot{\boldsymbol{q}}^+ \\ -\boldsymbol{\Lambda} \end{bmatrix} = \begin{bmatrix} \boldsymbol{H}(\boldsymbol{q})\dot{\boldsymbol{q}}^- \\ -e\boldsymbol{G}(\boldsymbol{q})\dot{\boldsymbol{q}}^- \end{bmatrix} \tag{10}$$

where $\boldsymbol{\Lambda}$ is the contact impulse. The upper part of this equation implies:

$$\boldsymbol{H}(\boldsymbol{q})\dot{\boldsymbol{q}}^+ - \boldsymbol{H}(\boldsymbol{q})\dot{\boldsymbol{q}}^- = \boldsymbol{G}(\boldsymbol{q})^T\boldsymbol{\Lambda} \tag{11}$$

which is the change of momentum of the system due to the collision. The lower part

$$\boldsymbol{G}(\boldsymbol{q})\dot{\boldsymbol{q}}^{+} = -e\boldsymbol{G}(\boldsymbol{q})\dot{\boldsymbol{q}}^{-} \tag{12}$$

states what the contact velocity is after the collision. For $e = 0$ this is equivalent to a contact velocity of zero, a perfect inelastic collision.

In the following we may omit the arguments $\boldsymbol{q}, \dot{\boldsymbol{q}}$ for the quantities when their requirement should be clear from the context.

3 Spatial algebra

Traditionally the dynamics of rigid bodies are described by two sets of 3-D equations: one 3-D equation for the linear (translational) motions and forces and one 3-D equation for the rotational (angular) motions and forces. Spatial algebra is a notation for rigid body motions and dynamics that embeds the two types of 3-D equations in a single 6-D equations. A more complete introduction can be found in Featherstone (2010).

3.1 Elements of spatial algebra

The fundamental elements of spatial algebra are spatial motions $\hat{\boldsymbol{v}} \in \mathsf{M}^6$, spatial forces $\hat{\boldsymbol{f}} \in \mathsf{F}^6$ and spatial inertias $\hat{\boldsymbol{I}} : \mathsf{M}^6 \rightarrow \mathsf{F}^6$. By using Plücker bases $\mathcal{D}_O, \mathcal{E}_O$ Featherstone (2006) we have a mapping $\mathcal{D}_O : \mathbb{R}^6 \rightarrow \mathsf{M}^6$ and $\mathcal{E}_O : \mathbb{R}^6 \rightarrow \mathsf{F}^6$.

3.1.1 Motion and force vectors

A spatial velocity vector describes both linear and angular velocity of a rigid body and is of the form:

$$\hat{\boldsymbol{v}} = \begin{bmatrix} \boldsymbol{\omega} \\ \boldsymbol{v}_O \end{bmatrix} = \begin{bmatrix} \omega_x, \omega_y, \omega_z, v_{Ox}, v_{Oy}, v_{Oz}, \end{bmatrix}^T \tag{13}$$

where O is an arbitrary but fixed reference point. The 3-D vector $\boldsymbol{\omega} \in \mathbb{R}^3$ describes the angular velocity of the body about an axis that passes through O and $\boldsymbol{v}_O \in \mathbb{R}^3$ is the linear velocity of the body point that currently coincides with the reference point O.

A spatial force vector describes both linear and rotational force components that act on a body and is of the form:

$$\hat{\boldsymbol{f}} = \begin{bmatrix} \boldsymbol{n}_O \\ \boldsymbol{f} \end{bmatrix} = \begin{bmatrix} n_{Ox}, n_{Oy}, n_{Oz}, f_x, f_y, f_z, \end{bmatrix}^T \tag{14}$$

where $\boldsymbol{n}_O \in \mathbb{R}^3$ describes the total moment about the point that coincides with O and $\boldsymbol{f} \in \mathbb{R}^3$ the linear force along an axis passing through O.

3.1.2 Spatial inertia

The spatial inertia can be expressed as a 6×6 matrix:

$$\hat{\boldsymbol{I}} = \begin{bmatrix} \boldsymbol{I}_C + m\boldsymbol{c} \times \boldsymbol{c}\times^T & m\boldsymbol{c}\times \\ m\boldsymbol{c}\times^T & m\boldsymbol{1} \end{bmatrix} \tag{15}$$

where $\boldsymbol{I}_C$ is the inertia of the body at the center of mass, $\boldsymbol{c}$ is the 3-D coordinate vector of the center of mass, and m is the mass of the body.

For a body with spatial velocity $\hat{\boldsymbol{v}}$ and spatial inertia $\hat{\boldsymbol{I}}$ we can compute the spatial momentum as $\hat{\boldsymbol{h}} = \hat{\boldsymbol{I}}\hat{\boldsymbol{v}}$. One should note that $\hat{\boldsymbol{h}} \in \mathsf{F}^6$.

3.1.3 Spatial transformations

In practice it is useful to use different coordinate frames, such as body local coordinate frames, instead of performing computations in one single coordinate frame. Spatial algebra provides a convenient way to transform spatial quantities such as motions, forces, and inertias between coordinate frames.

If A and B are two coordinate frames where B is translated by $\boldsymbol{r} \in \mathbb{R}^3$ and also rotated as described by the orthonormal matrix $\boldsymbol{E} \in \mathbb{R}^{3\times3}$ we can then define the following spatial transformations:

$${}^B\boldsymbol{X}_A = \begin{bmatrix} \boldsymbol{E} & \boldsymbol{0} \\ -\boldsymbol{E}\boldsymbol{r}\times & \boldsymbol{E} \end{bmatrix}, \quad {}^B\boldsymbol{X}_A^* = \begin{bmatrix} \boldsymbol{E} & -\boldsymbol{E}\boldsymbol{r}\times \\ \boldsymbol{0} & \boldsymbol{E} \end{bmatrix} \tag{16}$$

with $\boldsymbol{r}\times$ being the skew symmetric matrix of the vector:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} \times = \begin{bmatrix} 0 & -z & y \\ z & 0 & -x \\ -y & x & 0 \end{bmatrix}. \tag{17}$$

Elements of M^6 are transformed using ${}^B\boldsymbol{X}_A$ whereas elements of F^6 are transformed using ${}^B\boldsymbol{X}_A^*$. Similarly the inverse transformations ${}^A\boldsymbol{X}_B$ and ${}^A\boldsymbol{X}_B^*$ can be defined. For given $\boldsymbol{E}$ and $\boldsymbol{r}$ we may also write the spatial transformation as $\boldsymbol{X}(\boldsymbol{E}, \boldsymbol{r})$.

The motion vector ${}^A\hat{\boldsymbol{m}} \in \mathsf{M}^6$ and the force vector ${}^A\hat{\boldsymbol{f}} \in \mathsf{F}^6$ that are described in coordinate frame A can be transformed to the coordinate frame B using

$${}^B\hat{\boldsymbol{m}} = {}^B\boldsymbol{X}_A\,{}^A\hat{\boldsymbol{m}} \tag{18}$$

$${}^B\hat{\boldsymbol{f}} = {}^B\boldsymbol{X}_A^*\,{}^A\hat{\boldsymbol{f}}. \tag{19}$$

As spatial inertias are mappings from M^6 to F^6 the transformation of spatial inertia ${}^A\hat{\boldsymbol{I}}$ that is described in frame A can be transformed to frame B by:

$${}^B\hat{\boldsymbol{I}} = {}^B\boldsymbol{X}_A^*\,{}^A\hat{\boldsymbol{I}}\,{}^A\boldsymbol{X}_B. \tag{20}$$

3.1.4 Spatial cross products

When looking at rigid motions using 3-D equations the cross product has a special property: if we have a vector ${}^A\boldsymbol{u} \in \mathbb{R}^3$ that is fixed in a frame A moves with an angular velocity of ${}^A\boldsymbol{\omega}_A$ then the derivative of ${}^A\boldsymbol{u}$ is:

$$\dot{\boldsymbol{u}} = {}^A\boldsymbol{\omega}_A \times {}^A\boldsymbol{u}(t). \tag{21}$$

In spatial algebra there are two cross product operators: one for motion vectors:

$$\hat{\boldsymbol{m}}_O \times = \begin{bmatrix} \boldsymbol{\omega} \\ \boldsymbol{v}_O \end{bmatrix} \times = \begin{bmatrix} \boldsymbol{\omega}\times & \mathbf{0} \\ \boldsymbol{v}\times & \boldsymbol{\omega}\times \end{bmatrix}, \tag{22}$$

and one for force vectors:

$$\hat{\boldsymbol{v}} \times^* = \begin{bmatrix} \boldsymbol{\omega} \\ \boldsymbol{v}_O \end{bmatrix} \times^* = \begin{bmatrix} \boldsymbol{\omega}\times & \boldsymbol{v}_O\times \\ \mathbf{0} & \boldsymbol{\omega}\times \end{bmatrix}, \tag{23}$$

3.1.5 Spatial equation of motion

The equation of motion for a single rigid-body expressed using Spatial Algebra is stated by:

$$\hat{\boldsymbol{f}} = \hat{\boldsymbol{I}}\hat{\boldsymbol{a}} + \hat{\boldsymbol{v}} \times^* \hat{\boldsymbol{I}}\hat{\boldsymbol{v}} \tag{24}$$

$$= \hat{\boldsymbol{I}}\hat{\boldsymbol{a}} + \hat{\boldsymbol{p}}. \tag{25}$$

Here $\hat{\boldsymbol{f}} = \left[\boldsymbol{n}_O, \boldsymbol{f}\right]^T$ is the spatial force acting on a body where $\boldsymbol{n}_O$ is the total moment applied at point O and $\boldsymbol{f}$ the linear force that is acting on a line passing through O. The spatial acceleration $\hat{\boldsymbol{a}} = [\dot{\boldsymbol{\omega}}, \dot{\boldsymbol{v}}_O]^T$ consists of $\dot{\boldsymbol{\omega}}$ which is the rotational acceleration around an axis passing through O and the linear acceleration $\dot{\boldsymbol{v}}_O$. The quantity $\hat{\boldsymbol{p}}$ is sometimes called the spatial *bias force*, which equals to the spatial force that has to be applied to the body, such that no acceleration is produced on the body.

3.1.6 Comparison with the traditional 3-D notation

The spatial algebra equation of motion of a single rigid-body embeds the traditional 3-D equations of motion by:

$$\begin{bmatrix} \boldsymbol{n}_C \\ \boldsymbol{f} \end{bmatrix} = \begin{bmatrix} \boldsymbol{I}_C & \mathbf{0} \\ \mathbf{0} & m\mathbf{1} \end{bmatrix} \begin{bmatrix} \dot{\boldsymbol{\omega}} \\ \ddot{\boldsymbol{c}} - \boldsymbol{\omega} \times \boldsymbol{v}_C \end{bmatrix} + \begin{bmatrix} \boldsymbol{\omega}\times & \boldsymbol{v}_C\times \\ \mathbf{0} & \boldsymbol{\omega}\times \end{bmatrix} \begin{bmatrix} \boldsymbol{I}_C & \mathbf{0} \\ \mathbf{0} & m\mathbf{1} \end{bmatrix} \begin{bmatrix} \boldsymbol{\omega} \\ \boldsymbol{v}_C \end{bmatrix} \tag{26}$$

$$= \begin{bmatrix} \boldsymbol{I}_C\dot{\boldsymbol{\omega}} + \boldsymbol{\omega} \times \boldsymbol{I}_C\boldsymbol{\omega} \\ m\ddot{\boldsymbol{c}} \end{bmatrix}. \tag{27}$$

Here the reference point C is assumed to be coincident with the center of mass of the body and $\ddot{\boldsymbol{c}}$ is the linear acceleration of the center of mass.

It is important to emphasize that spatial algebra is not simply a matter of notation that concatenates the two 3-D equations. Instead, it performs operations on a different level of abstraction. Traditional 3-D equations express the motion of rigid bodies using quantities expressed at points that are moving with the body, e.g. the linear velocity of the center of mass and the angular velocity. In spatial algebra however the quantities are expressed at arbitrary but fixed reference points in space. Spatial velocities describe the flow of the body at the reference point, i.e. the angular and linear velocity of the point body that currently coincides with the point. Spatial accelerations describe the rate of change of this flow, i.e. how the angular and linear velocity of the point that coincide with the reference point change over time.

The difference of spatial velocity and acceleration compared to the classical formulation can seen by considering a body with constant angular velocity $\boldsymbol{\omega}$. In the classical 3-D formulation one would describe the motion of a body in terms of linear and angular velocity ($\dot{\boldsymbol{r}}_P$ and $\boldsymbol{\omega}$) and linear and angular acceleration ($\ddot{\boldsymbol{r}}_P$ and $\dot{\omega}$) of a point P that moves with the body. If that point P does not lie on the axis of rotation then the linear acceleration $\ddot{\boldsymbol{r}}_P \neq \mathbf{0}$ and points towards the axis of rotation. In spatial algebra however, the spatial acceleration $\hat{\boldsymbol{a}}_i$ of body i is the time derivative of the spatial velocity $\hat{\boldsymbol{v}}_i$. A constant $\hat{\boldsymbol{v}}_i$ therefore results in $\hat{\boldsymbol{a}}_i = \mathbf{0}$. For a thorough comparison of classical and spatial accelerations we refer to Featherstone (2001).

4 Modeling of rigid multibody systems

In this section we present the rigid multibody model formulation that is used in the description of algorithms below and as the main data structure of the software RBDL. It follows the notation and formulation presented in Featherstone (2008).

A rigid multibody system (MBS) consists of a set of bodies that are interconnected by joints. A joint always connects two bodies and in general affects the relative motion of the bodies.

The variables that describe a rigid multibody model with $n_B \in \mathbb{N}$ bodies are listed in Table 1

4.1 Coordinate frames and transformations

There are multiple coordinate frames involved when describing a multibody system. First there is the *global reference frame* which we name 0. This system is fixed and does not move. The root body B_0 is attached to this global reference frame and also defines a reference frame that is defined for the body B_0. We call this frame the *body (local) reference frame*. In the case of B_0 the global reference frame and the body reference frame of B_0 are the same.

When we have a system with a single moving body then this body B_1 is attached via joint J_1 to the base body B_0.

Table 1 Variables of a loop-free rigid multibody model

λ_i	Index of the parent body index for joint i that connects body i with body λ_i
$\kappa(i)$	The set of joints that influence (i.e. support) body i
$\mu(i)$	The set of children of joint i
$\nu(i)$	The subtree that starts at joint i
$\boldsymbol{S}_i$	Motion space matrix for joint i
$\boldsymbol{v}_i$	Spatial velocity of body i
$\boldsymbol{c}_i$	Velocity dependent acceleration term of body i
$\boldsymbol{a}_i$	Spatial acceleration of body i
$\boldsymbol{f}_i$	Spatial force body i is acting on the parent body λ_i via the connecting joint
${}^i\boldsymbol{X}_0$	Spatial transformation from the global frame to the frame of body i
${}^i\boldsymbol{X}_{\lambda_i}$	Spatial transformation from the parent of body i to body i
$\boldsymbol{X}_{\mathbf{T}i}$	Spatial transformation from the parent of body i to the frame of joint i
$\boldsymbol{I}$	Spatial inertia of body i
$\boldsymbol{I}^c$	Composite body inertia of body i
$\boldsymbol{I}^A$	Articulated body inertia of body i
$\boldsymbol{p}^A$	Articulated bias force of body i

Usually joints are not located at the origin of the parent body. Instead they are translated and/or rotated relative to the coordinate frame of the parent body. This gives rise to the *joint location frame*. The transformation from the parent body frame to the joint location frame is denoted as $\boldsymbol{X}_{\mathbf{T}i}$. It is fixed and specified in the parent's body reference frame.

When a joint is moving an additional frame is required. I.e. a revolute joint changes the orientation whereas a prismatic joint results in a translation of the coordinate frame. The *joint motion frame* is the frame that changes when the joint moves and is denoted as $\boldsymbol{X}_{\mathbf{J}i}$.

The transformation from the parent body to the child body is then given by

$$ {}^i\boldsymbol{X}_{\lambda_i} = \boldsymbol{X}_{\mathbf{J}i}\boldsymbol{X}_{\mathbf{T}i}. $$

4.2 Bodies

A single body in a rigid multibody system is described by its mass m, the location of the center of mass $\boldsymbol{c} \in \mathbb{R}^3$ and its inertia tensor $\boldsymbol{I}_C \in \mathbb{R}^{3\times3}$ at the center of mass. These values are taken to construct the spatial inertia matrix $\hat{\boldsymbol{I}}_i \in \mathbb{R}^{6\times6}$ for each body i.

4.3 Joints

A joint always connects two bodies with each other and in general limits the relative motion between them. A joint may allow between 0 and 6∘ of freedom, for which 0∘ of freedom mean that it is a fixed joint (i.e. the two bodies are rigidly connected with each other) and a joint with 6∘ of freedom (also called free-flyer joint) does not impose any restriction on the relative motion.

For each joint a specific subset of the generalized state variables are associated with the joint. For joint i with n_s degrees of freedom the value $\boldsymbol{q}_i \in \mathbb{R}^{n_s}$ are the associated generalized joint positions of joint J_i and similarly $\dot{\boldsymbol{q}}_i, \ddot{\boldsymbol{q}}_i, \boldsymbol{\tau}_i \in \mathbb{R}^{n_s}$ are the associated generalized joint velocities, accelerations and forces for that joint. $\boldsymbol{\tau}_i$ are more specifically the forces that are transmitted via the joint i. The tuple $(\boldsymbol{q}_i, \dot{\boldsymbol{q}}_i, \ddot{\boldsymbol{q}}_i, \boldsymbol{\tau}_i)$ is referred to as the joint state.

If the generalized joint positions for joint i are nonzero, then the transformation caused by the joint is represented by joint motion frame transformation $\boldsymbol{X}_{\mathbf{J}i}$. It is a spatial transformation of the form (16) where the entries $\boldsymbol{E}$ and $\boldsymbol{r}$ depend on the type of joint.

4.3.1 Joint models

The motion space of a joint is described by the so-called *motion subspace matrix* $\boldsymbol{S}$ which defines a mapping $\boldsymbol{S} : \mathbb{R}^{n_s} \to \mathsf{M}^6$. The subspace matrices for all joints with only 1∘ of freedom that are either revolute ($\boldsymbol{S}_{R.}$) or prismatic ($\boldsymbol{S}_{T.}$) around or along the coordinate axes are:

$$ \boldsymbol{S}_{Rx} = \begin{bmatrix} 1\\0\\0\\0\\0\\0 \end{bmatrix}, \boldsymbol{S}_{Ry} = \begin{bmatrix} 0\\1\\0\\0\\0\\0 \end{bmatrix}, \boldsymbol{S}_{Rz} = \begin{bmatrix} 0\\0\\1\\0\\0\\0 \end{bmatrix}, $$

$$ \boldsymbol{S}_{Tx} = \begin{bmatrix} 0\\0\\0\\1\\0\\0 \end{bmatrix}, \boldsymbol{S}_{Ty} = \begin{bmatrix} 0\\0\\0\\0\\1\\0 \end{bmatrix}, \boldsymbol{S}_{Tz} = \begin{bmatrix} 0\\0\\0\\0\\0\\1 \end{bmatrix}. $$

It is also possible to define motion subspace matrices for helical motions. In that case there are nonzero entries in both the upper three and the lower three entries.

Once the motion subspace matrix is defined the joint velocities and accelerations can be expressed as:

$$ \boldsymbol{v}_{\mathbf{J}i} = \boldsymbol{S}_i\dot{\boldsymbol{q}}_i \tag{28} $$

$$ \boldsymbol{a}_{\mathbf{J}i} = \boldsymbol{c}_{\mathbf{J}i} + \boldsymbol{S}_i\ddot{\boldsymbol{q}}_i \tag{29} $$

with

$$ \boldsymbol{c}_{\mathbf{J}i} = \dot{\boldsymbol{S}}_i\dot{\boldsymbol{q}}_i = \left(\frac{\mathrm{d}\boldsymbol{S}_i}{\mathrm{d}t} + \boldsymbol{v}_i \times \boldsymbol{S}_i\right)\dot{\boldsymbol{q}}_i \tag{30} $$

The value of $\dot{\boldsymbol{S}}_i$ is affected by two factors: first, the change of the motion subspace matrix over time itself and second, a change of the motion subspace matrix due to the motion of the frame of body i. For the simple single degree of freedom joints above, however this value is always zero. The term $\boldsymbol{c}_{\mathbf{J}i}$ is also called the *velocity dependent spatial acceleration term*.

4.3.2 Joints with multiple degrees of freedom

Models that contain joints with multiple degrees of freedom (DOF) can be treated in two different ways: emulate multiple degrees of freedom using multiple single degree of freedom joints and bodies with zero inertia and mass or the use of proper multiple degrees of freedom joints. The former have the advantage that they are very simple to implement however this is at the cost of performance as the algorithms will need more iterations to perform the same computations.

Multiple degrees of freedom joints occur in robotics mainly in the first movable body of the system from which the remaining multibody system is spanned. In humanoid robotics this body is often referred to as the *Floating Base* and the joint as *Floating Base Joint*, or *Freeflyer Joint*. In 3-D it has 6∘ of freedom and is not actuated, i.e. the generalized forces corresponding to it are always zero.

In models for human characters such as in animations also other joints such as those of hip, ankle, and shoulder can be modeled using 3 DOF joints. This makes joints with 3 DOF of particular interest. Furthermore when using 3 DOF joints the floating-base can be emulated using two joints instead of six.

For joints with multiple degrees of freedom we need to derive the expressions for $\boldsymbol{E}(\boldsymbol{q}_i), \boldsymbol{r}(\boldsymbol{q}_i)$ to describe the joint motion transformation and $\boldsymbol{S}(\boldsymbol{q}_i)$, and $\boldsymbol{c}_{\mathbf{J}i}(\boldsymbol{q}_i, \dot{\boldsymbol{q}}_i)$ to express the motion subspace of the joint, and the velocity dependent spatial acceleration term of the joint.

A 3 DOF joint that describes the translation along the X–, Y–, and Z– axis is described by:

$$\boldsymbol{E}(\boldsymbol{q}_i) = \mathbf{1} \in \mathbb{R}^{3\times 3}, \tag{31}$$

$$\boldsymbol{r}(\boldsymbol{q}_i) = (\boldsymbol{q}_{i,0}, \boldsymbol{q}_{i,1}, \boldsymbol{q}_{i,2})^T, \tag{32}$$

$$\boldsymbol{S}(\boldsymbol{q}_i) = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \tag{33}$$

$$\boldsymbol{c}_{\mathbf{J}i}(\boldsymbol{q}_i, \dot{\boldsymbol{q}}_i) = \mathbf{0} \in \mathsf{M}^6. \tag{34}$$

Rotational joints are slightly more involved due to the amount of sinus and cosine expressions. For a spherical joint that uses the Euler–Cardan angle convention XYZ we use the following notation $s_j = sin(\boldsymbol{q}_{i,j})$ and $c_j = cos(\boldsymbol{q}_{i,j})$ and furthermore $\dot{\boldsymbol{q}}_j = \dot{\boldsymbol{q}}_{i,j}$. This allows us to write the joint model dependent expressions as:

$$\boldsymbol{E}(\boldsymbol{q}_i) = \begin{bmatrix} c_2c_1 & s_2c_0 + c_2s_1s_0 & s_2s_0 - c_2s_1c_0 \\ -s_2c_1 & c_2c_0 - s_2s_1s_0 & c_2s_0 + s_2s_1c_0 \\ s_1 & -c_1s_0 & c_1c_0 \end{bmatrix}, \tag{35}$$

$$\boldsymbol{r}(\boldsymbol{q}_i) = \mathbf{0} \in \mathbb{R}^3, \tag{36}$$

$$\boldsymbol{S}(\boldsymbol{q}_i) = \begin{bmatrix} c_2c_1 & s_2 & 0 \\ -s_2c_1 & c_2 & 0 \\ s_1 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \tag{37}$$

$$\boldsymbol{c}_{\mathbf{J}i}(\boldsymbol{q}_i, \dot{\boldsymbol{q}}_i) = \begin{bmatrix} -s_2c_1\dot{\boldsymbol{q}}_2\dot{\boldsymbol{q}}_0 & -c_2s_1\dot{\boldsymbol{q}}_1\dot{\boldsymbol{q}}_0 & +c_2 * \dot{\boldsymbol{q}}_2 * \dot{\boldsymbol{q}}_1 \\ -c_2c_1\dot{\boldsymbol{q}}_2\dot{\boldsymbol{q}}_0 & +s_2s_1\dot{\boldsymbol{q}}_1\dot{\boldsymbol{q}}_0 & -s_2\dot{\boldsymbol{q}}_2\dot{\boldsymbol{q}}_1 \\ c_1\dot{\boldsymbol{q}}_1\dot{\boldsymbol{q}}_0 & & \\ 0 & & \\ 0 & & \\ 0 & & \end{bmatrix}. \tag{38}$$

Joints for other conventions or more degrees of freedom can similarly be derived. To note here is that the presented EulerXYZ joint suffers from singularities. An alternative is a joint that e.g. uses Quaternions Shoemake (1985), which is described in Featherstone (2008). A difficulty here is that the generalized velocities are no more the derivatives of the generalized positions.

As suggested in Featherstone (2008) it is very convenient to encapsulate the computations of the joint type and $\boldsymbol{q}, \dot{\boldsymbol{q}}$ dependent computations in a so-called *joint model calculation routine*:

$$\left[\boldsymbol{X}_{\mathbf{J}i}, \boldsymbol{S}_i, \boldsymbol{c}_{\mathbf{J}i}\right] = \text{jcalc}(\text{jtype}(i), \boldsymbol{q}_i, \dot{\boldsymbol{q}}_i). \tag{39}$$

The method jtype(i) returns a type of joint that will then be used e.g. in a **if** ... **elseif** block to chose the actual code depending of the type.

4.3.3 Joint numbering

The connection of bodies via joints in a loop-free system can also be seen as a directed graph, where the joints are the directed edges and the bodies are the nodes. Apart from the root body there are the same number of bodies B_i as joints J_i with $i = 1, \ldots, n_B$.

Every joint connects always exactly two bodies. Joint i connects bodies B_{λ_i} with body i. The body with index λ_i is also called the *parent body* and λ is called the *parent array*. The bodies and joints are numbered such that

$$\lambda_i < i \tag{40}$$

always holds. If $\kappa(i)$ is the set of joint indices that influence body i then this numbering has the property $j < i,\ \forall j \in \kappa(i)$.

This numbering becomes especially useful when computations require that all parent joints (i.e. all joints in $\kappa(i)$) must have been performed before a computation for body B_i can be done, as it is sufficient having computed the values for all bodies with $j < \lambda_i$.

Another important set of indices is $\mu(i)$ which contains the indices of all joints that are contained in the subtree starting at index i.

5 Dynamics algorithms

In this section we present algorithms that allow us to compute the quantities required for the evaluation of the equation of motion presented in Sect. 2.1. Some of the algorithms are the most efficient algorithms available for their tasks and all algorithms and methods are implemented in the software package RBDL.

5.1 Recursive Newton–Euler algorithm

The Recursive Newton–Euler algorithm (RNEA) is a highly efficient algorithm for the computation of inverse dynamics of a branched rigid-body model. It has an algorithmic complexity of $O(n_B)$ where n_B is the number of movable bodies in the multibody system.

It performs the computation in three steps:

1. Compute the position, velocity, and acceleration of all bodies.
2. Use (25) to compute for each body the net force that causes the previously computed acceleration.
3. Compute the force that is transmitted over each joint.

Pseudo code for the whole algorithm is presented in Algorithm 1. The acceleration of the root body is initialized with $\boldsymbol{a}_g = \left[\boldsymbol{0}, \boldsymbol{g}\right]^T \in \mathsf{M}^6$, where $\boldsymbol{g} \in \mathbb{R}^3$ is the gravity vector in global coordinates. As the accelerations are propagated from joint to joint this ensures that gravity is applied to every body.

The presented algorithm can be applied on kinematic trees and for all scleronomic joints including revolute, prismatic, and helical joints. Furthermore it allows to compute the inverse dynamics in presence of external forces $\boldsymbol{f}_i^x \in \mathsf{F}^6$. The algorithm can also be used to compute the generalized bias term in (1) $\boldsymbol{C}(\boldsymbol{q}, \dot{\boldsymbol{q}})$ by setting $\ddot{\boldsymbol{q}} = \boldsymbol{0}$.

$\boldsymbol{v}_0 = \boldsymbol{0}$
$\boldsymbol{a}_0 = -\boldsymbol{a}_g$
for $i = 1, \ldots, n_B$ **do**
  $\left[\boldsymbol{X}_{\mathbf{J}i}, \boldsymbol{S}_i, \boldsymbol{c}_{\mathbf{J}i}\right] = \text{jcalc}(\text{jtype}(i), \boldsymbol{q}_i, \dot{\boldsymbol{q}}_i)$
  ${}^i\boldsymbol{X}_{\lambda_i} = \boldsymbol{X}_{\mathbf{J}i}\boldsymbol{X}_{\mathbf{T}i}$
  if $\lambda_i \neq 0$ **then**
    ${}^i\boldsymbol{X}_0 = {}^i\boldsymbol{X}_{\lambda_i}\,{}^{\lambda_i}\boldsymbol{X}_0$
  end
  $\boldsymbol{v}_{\mathbf{J}i} = \boldsymbol{S}_i\dot{\boldsymbol{q}}_i$
  $\boldsymbol{a}_{\mathbf{J}i} = \boldsymbol{c}_{\mathbf{J}i} + \boldsymbol{S}_i\ddot{\boldsymbol{q}}_i$
  $\boldsymbol{v}_i = {}^i\boldsymbol{X}_{\lambda_i}\boldsymbol{v}_{\lambda_i} + \boldsymbol{v}_{\mathbf{J}i}$
  $\boldsymbol{a}_i = {}^i\boldsymbol{X}_{\lambda_i}\boldsymbol{a}_{\lambda_i} + \boldsymbol{a}_{\mathbf{J}i}$
  $\boldsymbol{f}_i = \boldsymbol{I}_i\boldsymbol{a}_i + \boldsymbol{v}_i \times^* \boldsymbol{I}_i\boldsymbol{v}_i - {}^i\boldsymbol{X}_0^*\boldsymbol{f}_i^x$
end
for $i = n_B, \ldots, 1$ **do**
  $\boldsymbol{\tau}_i = \boldsymbol{S}_i^T\boldsymbol{f}_i$
  if $\lambda_i \neq 0$ **then**
    $\boldsymbol{f}_{\lambda_i} = \boldsymbol{f}_{\lambda_i} + {}^{\lambda_i}\boldsymbol{X}_i^*\boldsymbol{f}_i$
  end
end

Algorithm 1: Recursive Newton–Euler algorithm

5.2 Composite rigid-body algorithm

The Composite Rigid-Body Algorithm (CRBA) computes the joint space inertia matrix $\boldsymbol{H}(\boldsymbol{q})$. The algorithm proceeds recursively and only computes the entries of the matrix that are nonzero. The algorithm can be motivated by looking at the computation for the multibody system's kinetic energy which can be stated as:

$$T = \frac{1}{2}\dot{\boldsymbol{q}}^T\boldsymbol{H}\dot{\boldsymbol{q}} = \frac{1}{2}\sum_{i=1}^{n_{dof}}\sum_{j=1}^{n_{dof}} H_{ij}\dot{q}_i\dot{q}_j \tag{41}$$

Alternatively, the kinetic energy of the system can be seen as the sum of the kinetic energy of the individual bodies:

$$T = \frac{1}{2}\sum_{i=1}^{n_B} \boldsymbol{v}_i^T\boldsymbol{I}_i\boldsymbol{v}_i. \tag{42}$$

Without loss of generality we can assume that all motion subspace matrices are expressed in the global reference frame. This allows us to write (42) as:

$$T = \frac{1}{2}\sum_{k=1}^{n_B}\sum_{i\in\kappa(k)}\sum_{j\in\kappa(k)} \dot{\boldsymbol{q}}_i^T\boldsymbol{S}_i^T\boldsymbol{I}_k\boldsymbol{S}_j\dot{\boldsymbol{q}}_j. \tag{43}$$

This equation expresses the kinetic energy as a sum over all combinations where body k is supported by both the joint i and j. This can be reformulated to

$$T = \frac{1}{2}\sum_{i=1}^{n_B}\sum_{j=1}^{n_B}\sum_{k\in\nu(i)\cap\nu(j)} \dot{\boldsymbol{q}}_i^T\boldsymbol{S}_i^T\boldsymbol{I}_k\boldsymbol{S}_j\dot{\boldsymbol{q}}_j \tag{44}$$

$$= \frac{1}{2} \sum_{i=1}^{n_B} \sum_{j=1}^{n_B} \dot{\boldsymbol{q}}_i^T \boldsymbol{H}_{ij} \dot{\boldsymbol{q}}_j \tag{45}$$

with

$$\boldsymbol{H}_{ij} = \sum_{k \in \nu(i) \cap \nu(j)} \boldsymbol{S}_i^T \boldsymbol{I}_k \boldsymbol{S}_j. \tag{46}$$

Equation (43) can be seen as assembling all the required quantities from k up to the root of the graph, whereas equation (44) performs the same computation by assembling the quantities down the subtrees starting at the joint that is influenced by both i and j.

The expressions $\boldsymbol{H}_{ij}$ and $\boldsymbol{H}_{ji}$ correspond to the block of the joint space inertia matrix that are affected by the joint variables of joints i and j. Due to the chosen joint numbering we can simplify the union of the two sets using:

$$\nu(i) \cap \nu(j) = \begin{cases} \nu(i) \text{ if } i \in \nu(j) \\ \nu(j) \text{ if } j \in \nu(i) \\ \emptyset \quad \text{otherwise} \end{cases} \tag{47}$$

Together with

$$\boldsymbol{I}_i^c = \sum_{j \in \nu(i)} \boldsymbol{I}_j \tag{48}$$

we can further simplify $\boldsymbol{H}_{ij}$ as:

$$\boldsymbol{H}_{ij} = \begin{cases} \boldsymbol{S}_i^T \boldsymbol{I}_i^c \boldsymbol{S}_j \text{ if } i \in \nu(j) \\ \boldsymbol{S}_i^T \boldsymbol{I}_j^c \boldsymbol{S}_j \text{ if } j \in \nu(i) \\ \boldsymbol{0} \quad \text{otherwise.} \end{cases} \tag{49}$$

The Composite Rigid-Body Algorithm is presented as pseudo code in Algorithm 2.

```
H = 0
for i = 1, ..., n_B do
    I_i^c = I_i
end
for i = n_B, ..., 1 do
    if λ_i ≠ 0 then
        I_{λ_i}^c = I_{λ_i}^c + {}^{λ_i}X_i^* I_i^c {}^iX_{λ_i}
    end
    F = I_i^c S_i
    H_ii = S_i^T F
    j = i
    while λ_i ≠ 0 do
        F = {}^{λ_i}X_j^* F
        j = λ_i
        H_ij = F^T S_j
        H_ji = H_ij^T
    end
end
```

Algorithm 2: Composite Rigid-Body Algorithm

The presented algorithm assumes that the spatial transformations from parent to child ${}^i\boldsymbol{X}_{\lambda_i}$ and the joint motion subspace matrices $\boldsymbol{S}_i$ have already been computed. If this is not the case then this could be done in the first loop. Furthermore, the quantities $\boldsymbol{S}_i$ and $\boldsymbol{I}_i^c$ are expressed in body local coordinates.

5.3 Articulated body algorithm

The articulated body algorithm (ABA) allows to compute the forward dynamics of a kinematic tree with $O(n_B)$ operations. It was first described by Featherstone in Featherstone (1983) however different variants have been described by others as shown in Jain (1991). We only outline the algorithm here. For a complete derivation including some variation in the formulation we refer to Featherstone (2008).

One of its key concepts is the articulated-body inertia which captures the dynamic properties of a body that is connected to other bodies and how it reacts when a force is applied to it while taking the connecting bodies into account. The equation of motion for an articulated body is given by:

$$\boldsymbol{f} = \boldsymbol{I}^A \boldsymbol{a} + \boldsymbol{p}^A \tag{50}$$

where $\boldsymbol{I}^A$ is the articulated body inertia and $\boldsymbol{p}^A$ is the articulated bias force.

The articulated body inertia and articulated bias force can be recursively computed. For body i the computations are:

$$\boldsymbol{I}_i^A = \boldsymbol{I}_i + \sum_{j \in \mu(i)} \boldsymbol{I}_j^a \tag{51}$$

$$\boldsymbol{p}_i^A = \boldsymbol{p}_i + \sum_{j \in \mu(i)} \boldsymbol{p}_j^a \tag{52}$$

with

$$\boldsymbol{I}_j^a = \boldsymbol{I}_j^A - \boldsymbol{I}_j^A \boldsymbol{S}_j (\boldsymbol{S}_j \boldsymbol{I}_j^A \boldsymbol{S}_j)^{-1} \boldsymbol{S}_j^T \boldsymbol{I}_j^A \tag{53}$$

$$\boldsymbol{p}_j^a = \boldsymbol{p}_j^A + \boldsymbol{I}_j^a \boldsymbol{c}_j + \boldsymbol{I}_j^A \boldsymbol{S}_j (\boldsymbol{S}_j^T \boldsymbol{I}_j^A \boldsymbol{S}_j)^{-1} (\boldsymbol{\tau}_j - \boldsymbol{S}_j^T \boldsymbol{p}_j^A). \tag{54}$$

The algorithm proceeds in three steps:

1. Compute positions, velocities and rigid-body bias forces from base outwards.
2. Compute the articulated body inertias and articulated bias forces.
3. From base outwards: use $\boldsymbol{I}_i^A$ and $\boldsymbol{p}_i^A$ computed in the previous step to compute the joint accelerations using:

$$\ddot{\boldsymbol{q}}_i = (\boldsymbol{S}_i^T \boldsymbol{I}_i^A \boldsymbol{S}_i^T)^{-1} (\boldsymbol{\tau}_i - \boldsymbol{S}_i^T (\boldsymbol{I}_i^A \boldsymbol{a}_{\lambda_i} + \boldsymbol{c}_i) - \boldsymbol{S}_i^T \boldsymbol{p}_i^A) \tag{55}$$

Pseudocode for the algorithm is shown in Algorithm 3. The code presented here avoids redundant computations of shared terms in Eqs. (53) and (54).

$v_0 = \mathbf{0}$
$a_0 = -a_g$
for $i = 1, \ldots, n_B$ **do**
  $\left[X_J, S_i, c_{Ji}\right] = \text{jcalc}(\text{jtype}(i), q_i, \dot{q}_i)$
  ${}^iX_{\lambda_i} = X_J X_{Ti}$
  if $\lambda_i \neq 0$ **then**
    ${}^iX_0 = {}^iX_{\lambda_i}\,{}^{\lambda_i}X_0$
  end
  $v_{Ji} = S_i \dot{q}_i$
  $v_i = {}^iX_{\lambda_i} v_{\lambda_i} + v_{Ji}$
  $c_i = c_{Ji} + v_i \times v_{Ji}$
  $I_i^A = I_i$
  $p_i^A = v_i \times^* I_i v_i - {}^iX_0^* f_i^x$
end
for $i = n_B, \ldots, 1$ **do**
  $U_i = I_i^A S_i$
  $D_i = S_i^T U_i$
  $u_i = \tau_i - S_i^T p_i^A$
  if $\lambda_i \neq 0$ **then**
    $I^a = I_i^A - U_i D_i^{-1} U_i^T$
    $p^a = p_i^A + I^a c_i + U_i D_i^{-1} u_i$
    $I_{\lambda_i}^A = I_{\lambda_i}^A + {}^{\lambda_i}X_i^* I^a\,{}^iX_{\lambda_i}$
    $p_{\lambda_i}^A = p_{\lambda_i}^A + {}^{\lambda_i}X_i^* p_a$
  end
end
for $i = 1, \ldots, n_B$ **do**
  $a' = {}^iX_{\lambda_i} a_{\lambda_i} + c_i$
  $\ddot{q}_i = D_i^{-1}(u_i - U_i^T a')$
  $a_i = a' + S_i \ddot{q}_i$
end

Algorithm 3: Articulated Body Algorithm

The Articulated Body Algorithm can be modified to yield a $O(n_B)$ solution operator to solve the linear system $\boldsymbol{H}(\boldsymbol{q})\ddot{\boldsymbol{q}} = \boldsymbol{\tau}$ for the accelerations $\ddot{\boldsymbol{q}}$. To do so the acceleration due to gravity and all velocity dependent variables in the first loop must be set to zero. Only the articulated bias forces $\boldsymbol{p}^A$ are influenced by $\boldsymbol{\tau}$. When solving the system for multiple values of $\ddot{\boldsymbol{q}}$ one should therefore avoid the costly re-evaluation of the articulated body inertias $\boldsymbol{I}^A$ and only compute them once.

6 Kinematics, energy, and momentum computations

In simulations and analysis of rigid multibody systems, it is often required to compute 3-D coordinates, 3-D linear velocities, and 3-D linear accelerations of points that are fixed on a body. This requires some clarifications, as spatial motion vectors describe the *flow* of motion at a specified point.

In this section the point P that is rigidly attached to and therefore moving with the body i in body local coordinates ${}^i\boldsymbol{r}_P \in \mathbb{R}^3$. In this section we are interested in computing the classical 3-D expressions of linear velocity, linear acceleration of the point P. Furthermore we want to formulate the contact Jacobian and contact Hessian for this point.

6.1 Point velocities

The velocity of the fixed body point can be computed using:

$$ {}^{P'}\hat{\boldsymbol{v}}_i = \begin{bmatrix} {}^{P'}\boldsymbol{\omega} \\ {}^{P'}\boldsymbol{v}_{P'} \end{bmatrix}_i = \boldsymbol{X}({}^0\boldsymbol{E}_i^T, {}^i\boldsymbol{r}_P)^i\boldsymbol{v}_i \tag{56} $$

where ${}^0\boldsymbol{E}_i$ is the orientation of body i with respect to the orientation of the global frame. The spatial motion vector $\boldsymbol{v}_i$ is transformed to a coordinate frame P' for which the reference point coincides with the global coordinates of P and also has the same orientation as the global reference frame. The lower part of ${}^{P'}\hat{\boldsymbol{v}}_P$ is therefore expressed at the point of body i that currently coincides with P and therefore we have ${}^{P'}\boldsymbol{v}_{P'} = {}^0\dot{\boldsymbol{r}}_P$.

In RBDL the function `CalcPointVelocity (model, q, qdot, body_id, point_position)` performs this computition and returns the 3-D vector ${}^{P'}\boldsymbol{v}_{P'}$.

6.2 Point accelerations

To compute the acceleration $\ddot{\boldsymbol{r}}_P$ of the body fixed point P we can use:

$$ {}^{P'}\hat{\boldsymbol{a}}_i' = \begin{bmatrix} {}^{P'}\boldsymbol{\omega} \\ {}^{P'}\boldsymbol{a}_{P'} \end{bmatrix}_i = \boldsymbol{X}({}^0\boldsymbol{E}_i^T, {}^i\boldsymbol{r}_P)^i\boldsymbol{a}_i + \begin{bmatrix} \mathbf{0} \\ {}^{P'}\boldsymbol{\omega} \times {}^{P'}\boldsymbol{v}_{P'} \end{bmatrix}. \tag{57} $$

The first term on the right-hand side transforms the spatial acceleration from body i coordinates to the coordinate frame of P' (i.e. the same coordinate frame as above). The second term compensates for the fact that spatial acceleration is *not* the acceleration of the reference point of the coordinate frame but instead it is *the rate of change of flow* at the reference point of the coordinate system.

In RBDL the function `CalcPointAcceleration (model, q, qdot, qddot, body_id, point_position)` performs this computition and returns the 3-D vector ${}^{P'}\boldsymbol{a}_{P'}$.

6.3 Contact jacobians

For a given joint configuration $\boldsymbol{q}$, the contact Jacobian from (5a) maps from the generalized velocities $\dot{\boldsymbol{q}}$ to the spatial velocity of a body expressed in some coordinate frame A:

$$ {}^A\boldsymbol{v}_j = {}^A\hat{\boldsymbol{G}}(\boldsymbol{q})\dot{\boldsymbol{q}} = \begin{bmatrix} {}^A\boldsymbol{G}_\omega(\boldsymbol{q}) \\ {}^A\boldsymbol{G}_{v_A}(\boldsymbol{q}) \end{bmatrix}\dot{\boldsymbol{q}} \tag{58} $$

${}^A\hat{\boldsymbol{G}}(\boldsymbol{q})$ is a $6 \times n_{dof}$ matrix, where the upper three rows ${}^A\boldsymbol{G}_\omega(\boldsymbol{q})$ map onto the angular velocity of the body and the bottom three rows ${}^A\boldsymbol{G}_{v_A}(\boldsymbol{q})$ map onto the linear velocity of ${}^A\boldsymbol{v}_j$. All

columns with index $i \notin \kappa(j)$ are zero. The values of the non-zero columns can be obtained by rewriting (58) as:

$$^{A}\hat{\boldsymbol{G}}(\boldsymbol{q})\dot{\boldsymbol{q}} = \sum_{i \in \kappa(j)} {}^{A}\boldsymbol{X}_i \boldsymbol{v}_i = \sum_{i \in \kappa(j)} {}^{A}\boldsymbol{X}_i \boldsymbol{S}_i \dot{q}_i . \tag{59}$$

The nonzero columns of the Jacobian can therefore be obtained by transforming the joint motion space matrices to the coordinate frame A. By using the coordinate frame P' from the previous paragraphs one can compute the Jacobian for the body fixed point P.

6.4 Contact Hessians

When differentiating the algebraic constraint equation twice we obtain:

$$\frac{\mathrm{d}^2}{\mathrm{d}t^2}\boldsymbol{g}(\boldsymbol{q}) = \boldsymbol{G}(\boldsymbol{q})\ddot{\boldsymbol{q}} + \dot{\boldsymbol{G}}(\boldsymbol{q})\dot{\boldsymbol{q}} \tag{60}$$

$$= \boldsymbol{G}(\boldsymbol{q})\ddot{\boldsymbol{q}} - \boldsymbol{\gamma}(\boldsymbol{q}, \dot{\boldsymbol{q}}) \tag{61}$$

where $\boldsymbol{\gamma}(\boldsymbol{q}, \dot{\boldsymbol{q}})$ is also called the contact Hessian. In RBDL it is computed using the same equations as the point acceleration Eqs. (57) and setting $\ddot{\boldsymbol{q}} = \boldsymbol{0}$.

6.5 System mass, center of mass, linear, angular, and centroidal momentum

Using spatial algebra we can also compute the system's spatial inertia which is the inertia the system would have if we lumped all bodies to a single body. It is obtained using:

$$^{0}\boldsymbol{I} = \sum_{i=1}^{n_B} {}^{0}\boldsymbol{X}_i^{*} \boldsymbol{I}_i {}^{i}\boldsymbol{X}_0 \tag{62}$$

The diagonal elements in lower right 3×3 submatrix of $^{0}\boldsymbol{I}$ will all contain the value of the system's total mass. The lower left 3×3 matrix contains the expression $m\boldsymbol{c}\times^{T}$, where $\boldsymbol{c}$ is the location of the system's Center of Mass (CoM) for which the 3-D vector can easily be extracted.

For a single rigid body its spatial momentum is $\boldsymbol{h} = \boldsymbol{I}\boldsymbol{v}$ which is an element of F^6. Together with the spatial transformations we can write the system's spatial momentum expressed at the origin of the global reference frame as:

$$^{0}\boldsymbol{h} = \sum_{i=1}^{n_B} {}^{0}\boldsymbol{X}_i^{*} \boldsymbol{I}_i \boldsymbol{v}_i . \tag{63}$$

The linear part of this vector will contain the linear momentum of the system which allows us to obtain the linear velocity of the CoM by dividing it by the system's total mass. The upper part contains the system's angular momentum expressed at the origin of the global coordinate system. Using the CoM's location we can get the Centroidal momentum Orin and Goswami (2008) as the upper 3-D vector of:

$$^{CoM}\boldsymbol{X}_0 {}^{0}\boldsymbol{h}. \tag{64}$$

7 Implementation notes

RBDL is implemented in a small subset of C++ and the programming interface could be relatively easily exposed to pure ANSI C. The library is self-contained and does not require any additional packages. For optimal performance it is however advised to use the C++ linear algebra package Eigen3 (Guennebaud et al. 2010). The programming interface itself has a model-based focused structure meaning that most calls take the model as their first argument. This also emphasizes the independence of model and the algorithms. The library also allows loading of models described in the URDF format.

The library is also extensively tested. At the time of writing there are more than 200 automated tests that ensure correctness of the library. Each test ensures the correctness of specific properties of our implementation, e.g. that the structure exploiting spatial algebra operators give the same results as the full 6-D formulation, the inverse dynamics computation is consistent with the forward dynamics computation. There is a wide range of different models that are used in the tests: from simple pendulums to a humanoid model with 36○ of freedom. Having a wide range of tests and testing infrastructure has helped us considerably to fix errors and gives us great confidence in the correctness of the library, also when major code changes have to be done. The tests run in less than a second which allows us to run the tests very frequently during development and to detect errors early on.

An example for the RBDL programming interface is shown in Algorithm 4. It creates a 3-D triple pendulum and computes the forward dynamics using the Articulated Body Algorithm and prints out the generalized acceleration.

7.1 Model structure

The model structure contains all variables listed in Table 1 and additional variables such as optional names (i.e. strings) for each body. The individual entries of the model structure are stored in arrays which store the entries for each body sequentially in memory.

7.2 Constraint sets

External contacts are modeled in RBDL using so-called constraint sets. A constraint set contains the data necessary to formulate all external contact constraints that currently act on the body. This includes a list of bodies, body points, and world normals which are used to compute the contact Jacobians

```
#include "rbdl.h"
#include <iostream>

using namespace RigidBodyDynamics;
using namespace RigidBodyDynamics::Math;

int main (int argc, char* argv[]) {
  Model model;
  Joint joint_rot_zyx (
    SpatialVector (0., 0., 1., 0., 0., 0.),
    SpatialVector (0., 1., 0., 0., 0., 0.),
    SpatialVector (1., 0., 0., 0., 0., 0.)
  );
  Body body (0.1, Vector3d (0., 0., -1.),
   Matrix3d (
    0.1, 0., 0.,
    0., 0.1, 0.,
    0., 0., 0.1)
  );

  model.gravity = Vector3d (0., 0., -9.81);
  unsigned int body_1_id = model.AddBody (
    0,
    Xtrans (Vector3d ( 0., 0., 0.)),
    joint_rot_zyx,
    body);
  unsigned int body_2_id = model.AddBody (
    body_1_id,
    Xtrans (Vector3d ( 0., 0., -1.)),
    joint_rot_zyx,
    body);
  unsigned int body_3_id = model.AddBody (
    body_2_id,
    Xtrans (Vector3d ( 0., 0., -1.)),
    joint_rot_zyx,
    body);

  VectorNd q =
    VectorNd::Zero (model.q_size);
  VectorNd qdot =
    VectorNd::Zero (model.qdot_size);
  VectorNd qddot =
    VectorNd::Zero (model.qdot_size);
  VectorNd tau =
    VectorNd::Zero (model.tau_size);

  ForwardDynamics (model,
    q, qdot, tau, qddot);

  std::cout « qddot.transpose()
    « std::endl;

  return 0;
}
```

Algorithm 4: A complete RBDL example the modeling and forward dynamics computation of a triple pendulum with spherical joints that are parameterized with EulerZYX angles.

and contact Hessians. Additionally it contains preallocated workspace for the linear system that has to be solved during the contact dynamics computation.

An example code is shown in Algorithm 5. Here we add three constraints to a body. The constraints all act on the same body point and constrain the motion of the point in all cardinal directions. After this we bind the constraint set to the model. This causes the workspace to be allocated and must be performed before computing the contact dynamics by calling `ForwardDynamicsContactsDirect (...)`.

```
ConstraintSet CS;

CS.AddConstraint (body_3_id,
  Vector3d (0., 0., -1.),
  Vector3d (1., 0., 0.)
);
CS.AddConstraint (body_3_id,
  Vector3d (0., 0., -1.),
  Vector3d (0., 1., 0.)
);
CS.AddConstraint (body_3_id,
  Vector3d (0., 0., -1.),
  Vector3d (0., 0., 1.)
);

CS.Bind(model);

ForwardDynamicsContactsDirect (model,
    q, qdot, tau, CS, qddot);

std::cout « qddot.transpose() « std::endl;
std::cout « CS.force.transpose()
    « std::endl;
```

Algorithm 5: Code example for the creation of a constraint set that adds an external point contact to body with id `body_3_id`. The point is located at $(0., 0., -1.)^T$ in the local coordinates of the body.

7.3 Structure exploiting spatial algebra

The pseudo code presented in Algorithm 1 and Algorithm 2 operate on 6-D spaces, however faster formulations can be derived for many spatial operations Featherstone (2008).

An expression like $\boldsymbol{X}\boldsymbol{v}$ would cost 36 multiplications and 30 additions using 6-D arithmetic. However the same operation can be performed by only 24 multiplications and 18 additions which can be seen by rewriting the multiplication using:

$$\begin{bmatrix} \boldsymbol{E} & \boldsymbol{0} \\ -\boldsymbol{E}\boldsymbol{r}\times & \boldsymbol{E} \end{bmatrix} \begin{bmatrix} \boldsymbol{\omega} \\ \boldsymbol{v}_O \end{bmatrix} = \begin{bmatrix} \boldsymbol{E}\boldsymbol{\omega} \\ -\boldsymbol{E}\boldsymbol{r} \times \boldsymbol{\omega} + \boldsymbol{E}\boldsymbol{v}_O \end{bmatrix} \tag{65}$$

$$= \begin{bmatrix} \boldsymbol{E}\boldsymbol{\omega} \\ \boldsymbol{E}(\boldsymbol{v}_O - \boldsymbol{r} \times \boldsymbol{\omega}) \end{bmatrix}. \tag{66}$$

The product $\boldsymbol{E}\boldsymbol{\omega}$ costs 9 multiplications and 6 additions, $\boldsymbol{r} \times \boldsymbol{\omega}$ costs 6 multiplications and 3 additions, $\boldsymbol{v}_O - \boldsymbol{v}'$ with $\boldsymbol{v}' =$

$\boldsymbol{r} \times \boldsymbol{\omega}$ costs 3 additions and the multiplication with $\boldsymbol{E}$ costs another 9 multiplications and 6 additions which totals in 24 multiplications and 18 additions.

At various points in the algorithms, it is required to concatenate spatial transformations (16). Featherstone introduces a compact representation $\text{plx}(\boldsymbol{E}, \boldsymbol{r})$ of the spatial transform that only stores the orientation matrix $\boldsymbol{E}$ and the linear displacement $\boldsymbol{r}$ instead of a full 6×6 matrix ("plx" here stands for "Plücker transformation"). The product of two spatial transformations

$$\boldsymbol{X}_i \boldsymbol{X}_j = \begin{bmatrix} \boldsymbol{E}_i & \boldsymbol{0} \\ -\boldsymbol{E}_i \boldsymbol{r}_i \times & \boldsymbol{E}_i \end{bmatrix} \begin{bmatrix} \boldsymbol{E}_j & \boldsymbol{0} \\ -\boldsymbol{E}_j \boldsymbol{r}_j \times & \boldsymbol{E}_j \end{bmatrix} \tag{67}$$

can then also be written as[1]:

$$\text{plx}(\boldsymbol{E}_i, \boldsymbol{r}_i)\ \text{plx}(\boldsymbol{E}_j, \boldsymbol{r}_j) = \text{plx}(\boldsymbol{E}_i \boldsymbol{E}_j, \boldsymbol{r}_j + \boldsymbol{E}_j^T \boldsymbol{r}_i) \tag{68}$$

which only needs 33 multiplications and 24 additions instead of 216 multiplications and 180 additions. The $\text{plx}(\cdot, \cdot)$ data structure can also be used to efficiently evaluate expressions $\boldsymbol{X}\boldsymbol{v}$, $\boldsymbol{X}^{-1}\boldsymbol{v}$, $\boldsymbol{X}^* \boldsymbol{f}$, and $\boldsymbol{X}^T \boldsymbol{f}$.

Another expression for which drastic improvements can be achieved is the following:

$${}^{\lambda_i}\boldsymbol{X}_i^* \boldsymbol{I}_i^c\ {}^i\boldsymbol{X}_{\lambda_i} \tag{69}$$

which can be found in the Composite Rigid-Body Algorithm (see Algorithm 2). It transforms the spatial inertia from reference frame of body i to that of body λ_i. When using 6-D operations this would result in multiplication of two 6×6 matrices which requires 432 multiplications and 360 additions. An alternative operation can be derived that performs the same computation using 52 multiplications and 56 additions. The faster method requires to store the spatial inertia more compact. The 6×6 matrix

$$\boldsymbol{I}^c = \begin{bmatrix} \boldsymbol{I} & m\boldsymbol{c}\times \\ m\boldsymbol{c}\times^T & m\boldsymbol{1} \end{bmatrix} \tag{70}$$

can be stored using 10 floating point values: 6 for the lower triangular part of the symmetric inertia matrix $\boldsymbol{I}$, 3 for $m\boldsymbol{c}\times$, and an additional floating point value for m. In Featherstone (2008) this compact structure is called $rbi(m, \boldsymbol{h}, lt(\boldsymbol{I}))$ where $lt(\cdot)$ denotes the lower triangular matrix of its argument. Using the $rbi(\cdot)$ structure furthermore results in fewer operations when rigid body inertias are being summed (10 additions instead of 36).

These structure exploiting operators and compact spatial data structures yield two benefits: fewer instructions and also reduced memory consumption. The latter also results in fewer cache misses on current CPU architectures that further improve performance as fewer values have to be fetched from slower memory hierarchies.

RBDL implements all of the mentioned compact structures and operators. Additional structures and exploiting operators can be found in Featherstone (2008).

7.4 Reuse of computed values

Computing the forward dynamics using (1) requires to compute $\boldsymbol{H}(\boldsymbol{q})$ and $\boldsymbol{C}(\boldsymbol{q}, \dot{\boldsymbol{q}})$. The former would be computed using the Composite Rigid-Body Algorithm (Algorithm 2) whereas the latter using the Recursive Newton–Euler algorithm (Algorithm 1) by setting $\ddot{\boldsymbol{q}} = \boldsymbol{0}$. Additional computations are required when computing forward dynamics with external contacts, such as the contact Jacobians and Hessians.

Instead of computing each from scratch one can reuse a considerable amount of values. By first calling the Recursive Newton–Euler Algorithm the values ${}^i\boldsymbol{X}_{\lambda_i}$ and $\boldsymbol{S}_i$ are already computed and can be reused by the Composite Rigid-Body Algorithm. Also values for $\boldsymbol{v}_i$ are already computed that are required for the contact Jacobians. Similarly all dependent variables for the contact Hessians are already precomputed.

The algorithms, the model structure and the programming interface in RBDL allow to reduce redundant computations in various functions using an optional boolean argument. E.g. in RBDL the function to compute the Jacobian for a point is defined using the following declaration:

```
void CalcPointJacobian (
Model &model,
const VectorNd &q,
unsigned int body_id,
const Vector3d &point_position,
MatrixNd &G,
bool update_kinematics = true
);
```

The last parameter `update_kinematics` has a default value of `true`. If it is not explicitly provided, it uses the default value `true`. When providing `false` as the last parameter RBDL reuses the existing values for ${}^A\boldsymbol{X}_i$ and $\boldsymbol{v}_i$ that are currently stored in the model structure.

8 Performance evaluation

Performance comparisons of implementations is generally difficult as different libraries have been built with different applications in mind. The aim of RBDL is to efficiently evaluate the equations of motion (1) and (7) and to compute forward and inverse dynamics of rigid multibody systems. As performance index we use the time duration it takes to evaluate specific quantities for a given number of times. As

[1] The derivation exploits $\boldsymbol{E}_j^T \boldsymbol{r}_i \times \boldsymbol{E}_j = (\boldsymbol{E}_j \boldsymbol{r}_i)\times$.

dynamics libraries that are based on maximal coordinates generally do not compute quantities explicitly we only consider the comparison of reduced coordinate libraries.

8.1 Floating base joint: emulated versus multi-DOF

Many real-world systems such as humanoid robots, vehicles, and biomechanical full-body human models are underactuated. There is one designated body that has all 6∘ of freedom that is not actuated (e.g. the Pelvis body) and all other bodies are connected to this so-called floating-base body. The 6-D joint is referred to as floating-base joint (or sometimes freeflyer joint).

In RBDL this floating-base joint can be modeled either using six 1-D joints or using two 3-D joints. To evaluate how the two modeling approaches affect performance we created two mathematically equivalent models of a humanoid robot with 35∘ of freedom. We used the sample humanoid robot from the OpenHRP framework (Kanehiro et al. 2004). The "emulated" model which uses six 1-DOF joints for the free-flyer joint and the "multi-DOF" model which uses two 3-DOF joints to describe an equivalent model. For both models we compare the performance of the $O(n_{dof})$ forward dynamics, the forward dynamics computation by explicitly formulating (1) and solving it using a LL^T factorization, and the computation of all quantities of (1) without solving it. All computations were performed in three batches of one million evaluations and the fastest values were used for the comparison. The results are shown in Fig. 1.

The emulated model the computations of one million calls to the Articulated Body Algorithm took 13.83 s whereas for the multi-DOF model it were 12.58 s. The alternate way to compute the forward dynamics is to directly solve Eq. (1). The fastest method we can use here solving the system using a LL^T decomposition as the joint-space inertia matrix is symmetric and positive definite. For the emulated model it took 21.51 s to build and solve the system whereas for the multi-DOF model it was 19.48 s. We also plotted the durations it takes when only building the systems: 12.24 s for the emulated model and 9.69 s for the multi-DOF model. Overall using two 3-DOF joints instead of six 1-DOF joints improves performance by about 9 % when computing forward dynamics and about 10 % when evaluating $\boldsymbol{H}(\boldsymbol{q})$ and $\boldsymbol{C}(\boldsymbol{q}, \dot{\boldsymbol{q}})$.

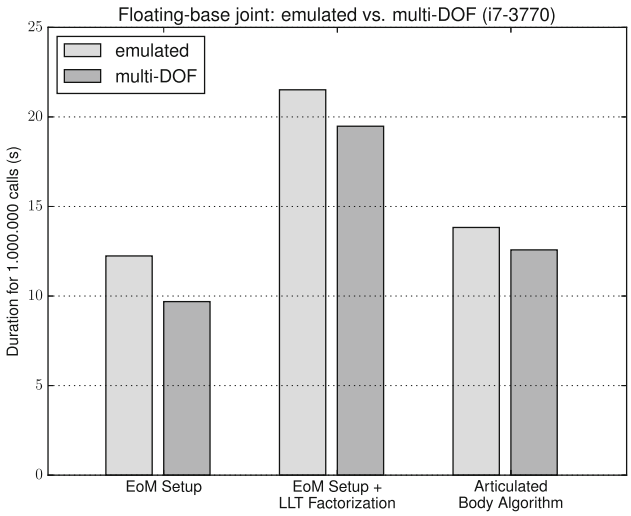

Fig. 1 Performance comparison for different modeling approaches of the floating-base joint for a 35-DOF humanoid model. Mathematically the same humanoid model was used, however the emulated version uses 6 1-DOF joints to emulate the floating-base joint and the multi-DOF version uses two 3-DOF joints. We show the durations for one million evaluations to compute the quantities of the equation of motion (1), compute the forward dynamics by explicitly formulating equation of motion and solving it with a LL^T factorization, and computing the forward dynamics using the $O(n_{dof})$ Articulated Body Algorithm. A performance gain when using multi-DOF joints can be seen for all evaluations

8.2 Comparison with existing code

We compared the performance of RBDL with SIMBODY (Sherman et al. 2011). SIMBODY is a state of the art rigid multibody simulation library that uses reduced coordinates and recursive algorithms to compute the multibody system dynamics using the formulations described in Rodriguez et al. (1991). The library includes a sophisticated caching mechanism to ensure that redundant computations are avoided and has a fine grained programming interface that include special methods to apply common operations such as $\boldsymbol{H}^{-1}\boldsymbol{\tau}$ for a given $\boldsymbol{\tau}$ or $\boldsymbol{G}(\boldsymbol{q})^T\boldsymbol{\lambda}$ without explicitly formulating the involved matrices.

We compare the performance of RBDL and SIMBODY by comparing their implementations of a $O(n)$ forward dynamic operator, a linear time solution operator for the expression $\boldsymbol{H}(\boldsymbol{q})^{-1}\boldsymbol{\tau}$, and repeated calls to the former operator by calling it multiple times to compute the inverse of the full joint space inertia matrix. The last comparison measures how well both implementations are able to reduce computations by evaluating articulated body inertias only once. The computations are performed on planar chains of varying lengths in the range of 5–100∘ of freedom. As both RBDL and SIMBODY internally use a 3-D formulation the benchmark generalizes to 3-D models.

For the $O(n)$ forward dynamics comparison we use the method `calcAccelerationsIgnoringConstraints()` of SIMBODY and `ForwardDynamics()` of RBDL. Both libraries have designated operators to solve the expression $\boldsymbol{H}(\boldsymbol{q})^{-1}\boldsymbol{\tau}$ in $O(n)$ time: in SIMBODY it is implemented in `multiplyByMInv()` and in RBDL it is called `CalcMInvTimesTau()`. Both implementations of this operator allow to efficiently solve the systems for multiple arguments of $\boldsymbol{\tau}$ by evaluating the articulated body inertias only once. This is exploited in SIMBODY in the function

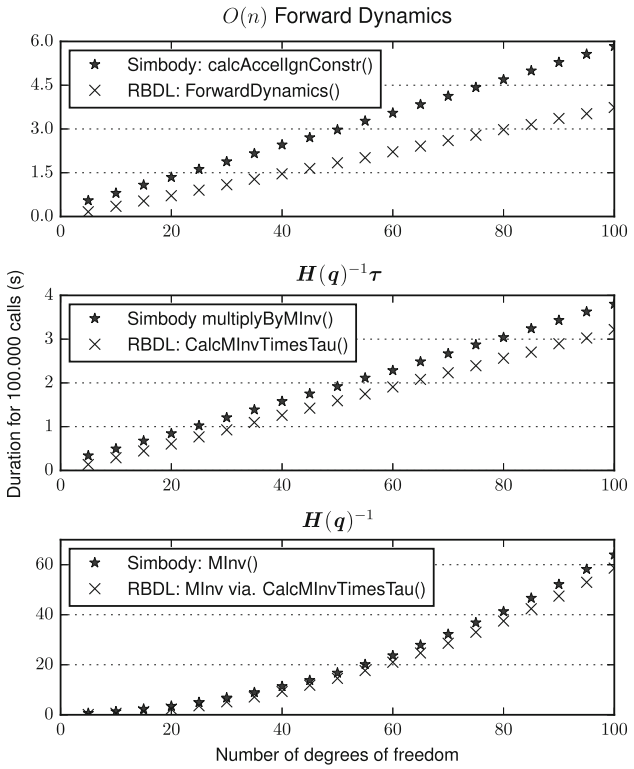

Fig. 2 Performance comparisons of RBDL with Simbody for models with increasing numbers of degrees of freedom. All computations were performed on a i7-3770

`MInv()` that computes the inverse of the joint space inertia matrix. RBDL does not contain a specific function to compute $\boldsymbol{H}(\boldsymbol{q})^{-1}$ but we implemented a similar function as in SIMBODY specifically for the benchmark in this paper. All computations were performed in three batches of 100.000 evaluations. The fastest batch was used to create a data point in the plots shown in Fig. 2.

The top plot shows the comparison of the $O(n)$ forward dynamics computation. Both implementations show as expected a linear increase of computation time when increasing the numbers of degrees of freedom of the benchmark model. The data points of RBDL are consistently below those of SIMBODY. For a model with 35∘ of freedom RBDL uses on average 12.75 μ s for the forward dynamics whereas it takes 21.71 μ s using SIMBODY. Overall RBDL is around 38 % faster than SIMBODY for the $O(n)$ forward dynamics computations.

The second plot compares the computation times for the $\boldsymbol{H}(\boldsymbol{q})^{-1}\boldsymbol{\tau}$ solution operator when calling it for a single vector of generalized torques $\boldsymbol{\tau}$. Similarly as the previous comparison one can see the linear algorithmic complexity of both implementations and again RBDL is consistently faster than SIMBODY. The difference here is not as dramatic as before and RBDL is around 15 % faster than SIMBODY. We assume that more optimization went into this routine for SIMBODY as the library makes use of this function when simulating constrained systems.

In the bottom plot we compare multiple invocations of the solution operator for $\boldsymbol{M}^{-1}\boldsymbol{\tau}$ where the articulated body inertias $\boldsymbol{I}^A$ are only computed once and reused in subsequent calls. Again RBDL shows better performance albeit being here only around 9 %.

Overall benchmarks show that RBDL performs very well compared to a well established dynamics code such as SIMBODY.

8.3 Comparison with symbolic code generation

We evaluated the performance of RBDL with symbolically generated code. that compute $\boldsymbol{H}(\boldsymbol{q})$, $\boldsymbol{C}(\boldsymbol{q}, \dot{\boldsymbol{q}})$, $\boldsymbol{G}(\boldsymbol{q})$, and $\boldsymbol{\gamma}(\boldsymbol{q}, \dot{\boldsymbol{q}})$ for the humanoid robot model with 35∘ of freedom that we already used in Sect. 8.1. One should note that the computational efforts of $\boldsymbol{C}(\boldsymbol{q}, \dot{\boldsymbol{q}})$ is only dependent on the number of degrees of freedom, however for $\boldsymbol{H}(\boldsymbol{q})$ also the structure of the robot is relevant. E.g. the pendulum used in the previous section results in a dense matrix, whereas a tree structured model, such as the humanoid model will have zeros at specific entries of the matrix.

The generated code was produced with the symbolic code generation tool DYNAMOD that was developed in our research group and is implemented in Maple ™. It uses the same algorithms as above to construct expressions and generate code for $\boldsymbol{H}(\boldsymbol{q})$, $\boldsymbol{C}(\boldsymbol{q}, \dot{\boldsymbol{q}})$, etc. DYNAMOD also caches expressions for every line of the algorithms. This was required as we experienced crashing of the computer algebra software when generating code for complex models. We assume that the generated symbolic equations became too large as we observed excessive memory usage of the software. Another benefit of the caching is that it it allows to avoid calculations of redundant computations, e.g. the expression of $\boldsymbol{f}_i$ in Algorithm 1 is only evaluated once and then reuses its value. This results in more compact and faster code. At the time of writing DYNAMOD has not been made available publicly but is planned for the near future.

For the computations with RBDL used two variants: one in which it simply computes the values and the other in which it reuses precomputed values as described in Sect. 7.4. We assessed the performance when evaluating the quantities required for contact dynamics using RBDL and compare it to the performance of generated C code. For this both RBDL and DYNAMOD compute the 6-D contact Jacobian $\boldsymbol{G}(\boldsymbol{q})$ and 6-D contact Hessian $\boldsymbol{\gamma}(\boldsymbol{q}, \dot{\boldsymbol{q}})$ expressed at the origin of the constrained bodies. For the benchmark model these are the bodies labeled `RLEG_LINK6` and `LLEG_LINK6` which represent the bodies of the right and left foot. To formulate for constraints expressed at other points on these bodies one can use spatial transformations from Sect. 3.1.3.

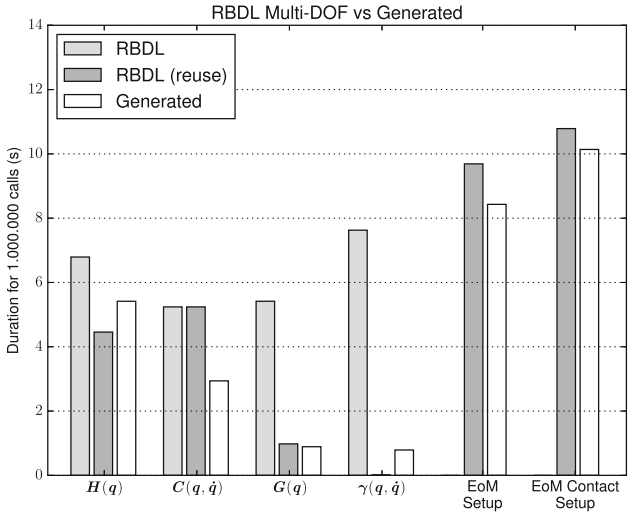

Fig. 3 Comparisons for the evaluation of quantities of the equation of motion for RBDL multi-DOF model without and with reusing of computed values and the equivalent computations performed by generated C code

The results of the benchmark are shown in Fig. 3 where evaluations of the respective functions have been performed one million times on PC running Ubuntu 14.04 on a Intel i7-3770 processor with 16 GB RAM. The vertical axis represents the duration in seconds, whereas the different function evaluations are listed on the horizontal axis. The last two groups of bar plots with the labels "EoM Setup" and "EoM Contact Setup" are benchmarks for computing all quantities to construct the linear systems of Eqs. (1) and (7) respectively by reusing as many computations as possible. Please note that this benchmark does not include the time required to actually solve these systems, instead it only is concerned on the computation on the required quantities.

Overall one can see that evaluation using symbolically generated code is faster when values are not reused. However if values can be reused the recursive methods can reach the efficiency of generated code, such as for the computation of $\boldsymbol{H}(\boldsymbol{q})$ where RBDL needs 6.76 s, the reuse variant 4.42 s, and the generated code 5.49 s. For the evaluation of $\boldsymbol{C}(\boldsymbol{q}, \dot{\boldsymbol{q}})$ unfortunately no computations can be saved and the generated code is with 2.94 s almost twice as fast as the 5.25 s of RBDL. Reusing of computed values is particularly beneficial for $\boldsymbol{G}(\boldsymbol{q})$ and $\boldsymbol{\gamma}(\boldsymbol{q}, \dot{\boldsymbol{q}})$. For the former RBDL needs 5.57 and 1.15 s when reusing values, it comes close to the evaluation of the generated code which used 0.90 s. The evaluation of the contact Hessian is particularly fast for the reusing RBDL variant: it needs 0.01 s, compared to the non-reusing 7.72 and the 0.77 for the generated code. This is due to the specific choice of the coordinate system for which the Hessian is expressed: it is simply $\boldsymbol{a}_i$ when $\ddot{\boldsymbol{q}} = \boldsymbol{0}$. Therefore no computations have to be made and the durations are simply due to the copying of the values. Computations of $\boldsymbol{H}(\boldsymbol{q})$ and $\boldsymbol{C}(\boldsymbol{q}, \dot{\boldsymbol{q}})$ while reusing the values amounts to 9.54 s for RBDL and 8.52 s for the generated code which makes it about 13 % faster. The evaluation of the quantities required for contact dynamics takes 10.91 s for RBDL when reusing values and 10.21 s for the generated code. Here the advantage of generated code is reduced to about 7 %. In the case of computing the forward dynamics these numbers roughly half as factorizations of the linear system will consume a large portion of the computation time.

One interesting observation here is that the sum of the timings for the individual quantities is less than the timings for "EoM Contact Setup". This is probably due to effects related to the instruction cache of the CPU. For the benchmarks of the individual quantities we run the specific functions over and over whereas for "EoM Setup" and "EoM Contact Setup" we repeatedly first compute $\boldsymbol{C}(\boldsymbol{q}, \dot{\boldsymbol{q}})$, then $\boldsymbol{H}(\boldsymbol{q})$ and then the contact Jacobians and Hessians if required. This results in larger reading and writing to a larger range of memory which has to be retrieved from slower memory hierarchies.

The comparisons show that faster code can be obtained when using a symbolic code generation approach. However in practical situations the performance of RBDL is likely to be better as the Jacobians and Hessians can also be evaluated for only one foot. The code generation approach used here always evaluates the Jacobians and Hessians for all bodies. One might suggest to generate code for each body individually, however this will increase the number of redundant computations as the Jacobians of the bodies share expressions.

9 Conclusion and outlook

In this paper we have given an introduction into the equations related to rigid body dynamics together with the formulation of the dynamics using spatial algebra. We have summarized some of the most efficient and powerful algorithms for generalized coordinate formulations of multibody systems along with a complete description of a model for any loop free rigid multibody system. We have shown how other quantities, such as Jacobians, linear point velocities and accelerations, system angular momentum, and others can be expressed using the model description. We have described how structures occurring in spatial algebra can be exploited and how computations can be reused to accomplish further performance gains. A performance comparison of RBDL with a state of the art rigid multibody dynamics libraries based on recursive methods and code generation library was given for models of varying sizes.

The comparisons show that the performance our implementation of recursive methods is faster than that of the well established SIMBODY library. The comparison of RBDL with symbolic code generation indicate that the latter is faster, however one should not underestimate the ease of use of

recursive methods used in RBDL and their independence of a separate code generation step. Also the flexibility of recursive methods in terms of extension, selective evaluation, and model modification during runtime likely outweigh the small performance increase of the generated code.

For future work we plan to extend the implemented contact formulation to allow unilateral contacts and friction. Another area of work is to expose the computations of RBDL to other computational environments such as MATLAB™ and PYTHON.

Acknowledgements The author gratefully acknowledges the financial support and the inspiring environment provided by the Heidelberg Graduate School of Mathematical and Computational Methods for the Sciences, funded by DFG (Deutsche Forschungsgemeinschaft) and the support by the European Commission under the FP7 projects ECHORD (Grant No 231143) and Koroibot (Grant No 611909). The author furthermore wants to thank Katja Mombaur for the opportunity to work in the stimulating environment of her research group Optimization in Robotics and Biomechanics and to Henning Koch for creating the generated code using his powerful DYNAMOD package.

References

Armstrong, W. W. (1979). Recursive solution to the equations of motion of an *n*-link manipulator. In *P*roceeding of the 5th World Congress on Theory of Machines and Mechanisms, (pp 1343–1346).

Ascher, U. M., Chin, H., Petzold, L. R., & Reich, S. (1994). Stabilization of constrained mechanical systems with daes and invariant manifolds. *Journal of Structural Mechanics*, *23*, 135–157.

Ball, R. S. (1900). *A treatise on the theory of screws*. Cambridge: Cambridge University Press.

Craig, J. (2005). Introduction to Robotics: Mechanics and Control. Addison-Wesley series in electrical and computer engineering: control engineering, Pearson Education, Incorporated.

Featherstone, R. (1983). The calculation of robot dynamics using articulated-body inertias. *The International Journal of Robotics Research*, *2*(1), 13–30. doi:10.1177/027836498300200102.

Featherstone, R. (2001). The acceleration vector of a rigid body. *The International Journal of Robotics Research*, *20*(11), 841–846. doi:10.1177/02783640122068137.

Featherstone, R. (2006). Plucker basis vectors. In: ICRA, (pp. 1892–1897), doi:10.1109/ROBOT.2006.1641982.

Featherstone, R. (2008). *Rigid body dynamics algorithms*. New York: Springer.

Featherstone, R. (2010). A beginner's guide to 6-d vectors (part 1). *Robotics Automation Magazine, IEEE*, *17*(3), 83–94. doi:10.1109/MRA.2010.937853.

Featherstone, R., & Orin, D. (2000). Robot dynamics: equations and algorithms. In: Robotics and Automation (ICRA). *P*roceedings of IEEE International Conference, (Vol. 1, pp. 826–834), doi:10.1109/ROBOT.2000.844153.

Guennebaud, G. et al. (2010). Eigen v3. http://eigen.tuxfamily.org.

Jain, A. (1991). Unified formulation of dynamics for serial rigid multibody systems. *Journal of Guidance, Control, and Dynamics*, *14*(3), 531–542. doi:10.2514/3.20672.

Jain, A. (2011). *Robot and multibody dynamics*. New York: Springer. doi:10.1007/978-1-4419-7267-5.

Jain, A., & Rodriguez, G. (1993). An analysis of the kinematics and dynamics of underactuated manipulators. *Robotics and Automation, IEEE Transactions on*, *9*(4), 411–422. doi:10.1109/70.246052.

Kanehiro, F., Hirukawa, H., & Kajita, S. (2004). OpenHRP: Open architecture humanoid robotics platform. *The International Journal of Robotics Research*, *23*(2), 155–165. doi:10.1177/0278364904041324.

Khalil, W., & Dombre, E. (2004). *Modeling*. Kogan Page Science paper edition, Elsevier Science: Identification and Control of Robots.

Luh, J. Y. S., Walker, M. W., & Paul, R. P. C. (1980). On-line computational scheme for mechanical manipulators. *Journal of Dynamic Systems, Measurement, and Control*, *102*, 69–102. doi:10.1115/1.3149599.

Orin, D., & Goswami, A. (2008). Centroidal momentum matrix of a humanoid robot: Structure and properties. In: Intelligent Robots and Systems, 2008. IROS 2008. *I*EEE/RSJ International Conference on, (pp. 653–659), doi:10.1109/IROS.2008.4650772.

Pang, J. S., & Trinkle, J. C. (1996). Complementarity formulations and existence of solutions of dynamic multi-rigid-body contact problems with coulomb friction. *Mathematical Programming*, *73*(2), 199–226. doi:10.1007/BF02592103.

Pfeiffer, F., & Glocker, C. (2008). *Multibody dynamics with unilateral contacts wiley series in nonlinear science*. New York: Wiley.

Rodriguez, G. (1987). Kalman filtering, smoothing, and recursive robot arm forward and inverse dynamics. *Robotics and Automation, IEEE Journal of*, *3*(6), 624–639. doi:10.1109/JRA.1987.1087147.

Rodriguez, G., Kreutz, K., & Milman, M. (1988). A spatial operator algebra for manipulator modeling and control. In: Intelligent Control, 1988. *P*roceedings of thr IEEE International Symposium on, (pp. 418–423), doi:10.1109/ISIC.1988.65468.

Rodriguez, G., Jain, A., & Kreutz-Delgado, K. (1991). A spatial operator algebra for manipulator modeling and control. *The International Journal of Robotics Research*, *10*(4), 371–381.

Sherman, M. A., Seth, A., & Delp, S. L. (2011). Simbody: Multibody dynamics for biomedical research. *Procedia IUTAM, IUTAM Symposium on Human Body Dynamics*, (Vol. 2, pp. 241–261), doi:10.1016/j.piutam.2011.04.023.

Shoemake, K. (1985). Animating rotation with quaternion curves. In *P*roceedings of the 12th Annual Conference on Computer Graphics and Interactive Techniques, ACM, New York, NY, SIGGRAPH '85, (pp. 245–254), doi:10.1145/325334.325242.

Uchida, T. K., Sherman, M. A., Delp, S. L. (2015). Making a meaningful impact: modelling simultaneous frictional collisions in spatial multibody systems. *P*roceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences, (Vol. 471), doi:10.1098/rspa.2014.0859.

Martin L. Felis studied mathematics and computer science at Heidelberg University (Germany) and Otago University (New Zealand) and received his diploma (Masters equiv.) with specialization in optimal control and simulation from Heidelberg University in 2009. He received his Ph.D. from Heidelberg University in 2015 under the supervision of Prof. Katja Mombaur and Prof. Alain Berthoz (Collège de France, Paris). He is an alumni of the Heidelberg Graduate School of Mathematical and Computational Methods for the Sciences and former member of the research group Optimization in Robotics and Biomechanics at Heidelberg University. He now works in the video game industry on real-time character animation systems.