Technical Report

UCAM-CL-TR-683
ISSN 1476-2986

Number 683

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Simulation of colliding constrained rigid bodies

Martin Kleppmann

April 2007

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

This technical report is based on a final-year project dissertation for the Computer Science Tripos. It was submitted in May 2006. The dissertation was marked as the best dissertation in its year and was awarded the AT&T prize.

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

http://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

Abstract

I describe the development of a program to simulate the dynamic behaviour of interacting rigid bodies. Such a simulation may be used to generate animations of articulated characters in 3D graphics applications. Bodies may have an arbitrary shape, defined by a triangle mesh, and may be connected with a variety of different joints. Joints are represented by constraint functions which are solved at run-time using Lagrange multipliers. The simulation performs collision detection and prevents penetration of rigid bodies by applying impulses to colliding bodies and reaction forces to bodies in resting contact.

The simulation is shown to be physically accurate and is tested on several different scenes, including one of an articulated human character falling down a flight of stairs.

An appendix describes how to derive arbitrary constraint functions for the Lagrange multiplier method. Collisions and joints are both represented as constraints, which allows them to be handled with a unified algorithm. The report also includes some results relating to the use of quaternions in dynamic simulations.

Contents

Chapter 1

Introduction

This project is concerned with the dynamics, i.e. the behaviour over time, of a general class of mechanical systems in three-dimensional space. In this report I describe the development and evaluation of a program to simulate such physical systems. The primary domain of this work is computer graphics, where physically-based modelling is a useful tool for creating realistic animations, although a wider range of applications in robotics and engineering can be envisaged.

1.1 Motivation

3D animated graphics have become an everyday part of our lives through films, advertising and computer games. Recent years have seen not only the production of feature-length animated films, but also the widespread use of animation in combination with traditionally shot footage. The techniques are now so refined that computer generated and recorded pictures are sometimes indistinguishable even to experts, thus opening up many new artistic possibilities.

Currently computer animation still involves a large amount of manual effort, despite commonly being termed "computer generated". It is said that in large-budget film productions every single frame is edited by hand, and many animations are completely handcrafted. There are artistic reasons for this (sometimes an animation is deliberately required to be physically impossible to achieve a special effect), and also economic reasons (animators are cheaper to hire than computer scientists). However, as the complexity of animated scenes increases, manual animation becomes unfeasible. Large crowds, fluids, cloth and some types of deformable bodies are therefore commonly animated using simulation techniques.

Unfortunately such scenes with an unbounded number of degrees of freedom are difficult to simulate well. Fluid dynamics is very complicated and a subject of ongoing research; the physics of deformable bodies (so-called "soft matter") is not even completely understood. Graphical simulations in these areas tend to resort to means which look good but are physically meaningless. This project aims to address the more well-defined subject of character animation, i.e. the movement of humans, many species of animal, and some types of alien creatures. Their common feature is that their bodies are deformable on the basis of a skeleton. Such bodies are called *articulated bodies* in computer graphics, and it is usually assumed that this is the only type of deformation – they do not have "soft flesh". *Skeleton* is not used as an anatomically accurate term, because more or fewer bones might be used, depending on the animation requirements.

Rigid body dynamics, which will form the basis for physically based character animation, is well understood and has a solid theoretical underpinning against which the simulation results can be checked. Hence physically-based modelling lends itself more to an objective evaluation

than many other areas of computer graphics, making it well suited as a Part II project.

While a simulation of a single rigid body can be fairly simple, dealing with multiple non-penetrating and interacting bodies is surprisingly difficult. In fact, creating such simulations was a subject of PhD theses only 10–15 years ago [2, 15, 20] and the topic has not yet settled enough to be described in textbooks[1]. This means that a project in this area will inevitably involve a research component in addition to the software engineering. This constitutes an exciting challenge.

1.2 Scope and requirements

Simulations of dynamic systems are employed in a wide range of environments:

1. games and related real-time applications, where fast computation is paramount and anything which "looks good" is acceptable;
2. professional animation systems for high quality graphics, in which calculations are done offline;
3. engineering and quantitative research, where accuracy and knowledge of errors are the main concerns.

The project proposal identifies itself with the second area, a trade-off between accuracy and speed. Once the scope was set, a more detailed analysis of the requirements had to be undertaken. Based on the project proposal, the program must

- read several polygon meshes, with skeletons and physical properties, from a file;
- provide a general framework for handling interactions between objects;
- ensure, as a particular type of interaction, that rigid bodies do not penetrate each other;
- combine forces, torques and impulses from interactions and hence calculate the change to the system over time;
- output the simulation state over time in rendered form on screen, and into a file such that it can be processed by other applications.

With respect to physical laws, the simulation should include

- the momentary state of a rigid body (position and orientation) and its velocities (linear and angular),
- changes to a body's state through forces, torques, linear and angular impulses,
- inertial mass, gravity and collisions between bodies,
- appropriate handling for articulated bodies, including limits on the range of valid angles for each joint.

This list of requirements differs slightly from those stated in the proposal:

[1] Some books [23, 6] give an introduction to the subject, but not in enough depth that the reader could actually implement a simulation.

- The simulation of friction forces was removed from the requirements, because the literature suggested that it was a very difficult problem [2] and I decided it was beyond the scope of a Part II project.
- Muscular forces are easy to implement but involve a large amount of manual effort to control [10], so I shifted the emphasis towards "passive" systems with no external forces apart from gravity.
- The requirements were extended from one body to multiple bodies – not many interesting simulations involve just a single body.
- Output of the results to a file was added to ease evaluation.

1.3 Overview

The dynamics of rigid bodies are physically described by ordinary differential equations (ODEs), and a simulation essentially comes down to numerically finding their solution, given a set of initial conditions and a set of constraints. Hence this project can be broken into six components:

1. A numerical solver for ordinary differential equations. This is a well researched area, and a range of good standard algorithms exist. This component is described in section 2.2.
2. An implementation of the equations of motion for rigid bodies. This is standard physics but has some implementation quirks. See sections 2.1 and 2.3.
3. Algorithms to ensure the correct handling of articulated bodies. See section 2.4.
4. Detection of collision between bodies. A vast amount of research has been done in this area, and most algorithms are concerned with speed. However, since run-time efficiency is not my primary concern, this part was kept quite simple. See section 3.2.
5. Handling of collisions, that is computation of the correct impulses between colliding bodies. This must also work in the context of articulated bodies! See section 3.3.3.
6. Handling of resting contact, that is the computation of forces between bodies which are in contact but not colliding. This is probably the most challenging part of the project. See section 3.3.2.

Chapter 2

Preparation

The preparatory work for this project splits into two parts:

- Surveying literature, learning the theoretical background and understanding the existing algorithms for my task. The outcome of this preparation is discussed in sections 2.1 to 2.4.
- Setting up and learning to use the software environment for development. This is described in section 2.5.

2.1 Rigid body dynamics

On the most theoretical level, a rigid body is a collection of k point masses ($k \geq 3$, can be infinite) subject to the constraint that the distance between any pair of masses must remain constant. More conveniently, we can regard a rigid body as an object with a three-dimensional shape, a non-zero volume and some mass distribution, and assume that it does not change shape.

At a particular point in time, a rigid body is fully determined by four non-scalar quantities: the *position* of its centre of mass, its *orientation*, and its *linear* and *angular velocities*. These values usually change over time. Each of these can be easily represented as a three-component vector, except for orientation, which is discussed in section 2.3. By convention a right-handed Cartesian coordinate system is used.

To characterize the body's dynamic behaviour, we also need to know its *inertial mass*, a scalar quantity, and its *moment of inertia*, which is a rank-2 tensor (commonly written as a 3×3 matrix). While the mass stays constant, the moment of inertia may depend on the body's orientation – it is, however, constant when expressed with respect to the body's *principal axes* [8, 9].

Angular velocity appears to be a straightforward way of describing the body's rotation: the direction of the vector gives the axis of rotation, while its magnitude is the rate of rotation in radians per second. Unfortunately, in an asymmetric body, the angular velocity may vary over time even if there are no external influences on the body, due to an effect known as *free precession* or *Poinsot motion* [9, 12]. It is therefore more convenient to use *angular momentum* instead, which is conserved in the absence of torques.

Forces and torques acting on a body may change the momenta of a body as described in table 2.1. A force $\mathbf{F}$ may be applied to any point $\mathbf{r}'$ of a body. We can treat this as if the force had been applied to the centre of mass $\mathbf{r}$, and add an additional torque given by $\boldsymbol{\tau} = (\mathbf{r}' - \mathbf{r}) \times \mathbf{F}$. If multiple forces and torques are applied, all forces (applied to the centre of mass) may be added into a single vector, and similarly all torques may be added.

	Linear	*Angular*
Resistance to change	Mass m	Moment of inertia I
Stationary state	Centre of mass position $\mathbf{r}$	*see section 2.3*
Velocity	Linear velocity $\mathbf{v} = \dot{\mathbf{r}}$	Angular velocity $\boldsymbol{\omega}$
Momentum	Linear momentum $\mathbf{p} = m\mathbf{v}$	Angular momentum $\mathbf{L} = \mathsf{I}\boldsymbol{\omega}$
External influence	Force $\mathbf{F} = \dot{\mathbf{p}}$	Torque $\boldsymbol{\tau} = \dot{\mathbf{L}}$

Table 2.1: Summary of the physical quantities describing a rigid body.

To solve the differential equations of motion, forces are integrated over time to find linear momentum, and torques are integrated to find angular momentum. From each of these we calculate linear and angular velocities, which are in turn integrated to find the position and orientation over the course of time. It is important that we integrate over torques (and not angular accelerations), otherwise the simulation will not correctly conserve angular momentum.

2.2 Solving ordinary differential equations (ODEs)

Undergraduate mathematics courses cover various methods for solving ODEs analytically. These work well for simple systems and deliver an exact solution. For example, if a constant force is applied to a body, its displacement is a quadratic (parabolic) function of time; if the force is negatively proportional to the displacement, we observe simple harmonic oscillation. Unfortunately, when systems become only slightly more complicated, the equations become intractable – even a simple pendulum falls in this category. General solutions can then only be approximated numerically.

The general scheme for numerical ODE solvers is to take the value of a function $f(t')$ and its derivative $\frac{\mathrm{d}f}{\mathrm{d}t}(t')$ at some point in time t'. From these the algorithm extrapolates what the value of the function is expected to be some time step h later. The simplest, most frequently-quoted and worst algorithm for this purpose is Euler's method,

$$f(t' + h) = f(t') + h\frac{\mathrm{d}f}{\mathrm{d}t}(t'), \tag{2.1}$$

which performs linear extrapolation. Figure 2.1 illustrates its operation and the errors introduced in the solution.

A popular and robust alternative is given by the *Runge-Kutta* family of ODE solvers, which evaluate $\frac{\mathrm{d}f}{\mathrm{d}t}$ at different times and use these values to fit a polynomial curve to the exact solution over the range of one time step. For example, fourth-order Runge-Kutta (RK4) [17] uses four values of the derivative to fit a fourth-order polynomial – contrast this to the first-order polynomial (linear function) used by Euler's method. This amounts to using the first five terms of the Taylor series of the exact solution, including the h^4 term, so we expect the error in each time step to be $O(h^5)$ – a minute error even if h is only moderately small. The step size can be much longer than in Euler's method while achieving the same accuracy, which makes Runge-Kutta significantly more efficient.

Further improvements can be made by adapting the step size h to the situation. In this project I chose to use the embedded Cash-Karp/Runge-Kutta method described in [17], which

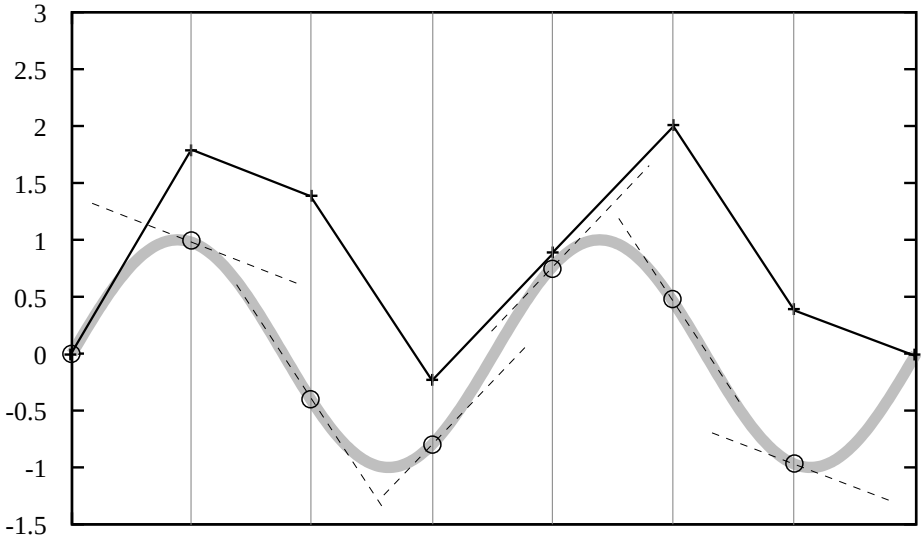

Figure 2.1: Demonstration of Euler's method for solving the ODE of simple harmonic oscillation. The broad grey line represents the exact solution. At each time step, the method takes the derivative (gradient) of the function and uses it to extrapolate linearly.

uses six evaluations of the derivative to calculate both a fourth-order and a fifth-order extrapolation. If the difference between them is small, the total error is likely to be small, and the next step will be longer. If the absolute difference lies above some threshold, the current time step is discarded and retried with a smaller h. Thus the ODE solver can rapidly skip over periods in which there is little happening, without sacrificing accuracy in moments of sudden change.

2.3 Quaternions

2.3.1 The need for Quaternions

Besides its position, each rigid body in 3D space may have an orientation, in which there are three degrees of freedom (three independent axes to rotate about). While the position of a body can be neatly represented using Cartesian coordinates, there is no obvious best way of describing an orientation. The most common schemes describe it in terms of a rotation operation which transforms a vector in the body's local coordinates into world coordinates (or *vice versa*). But again, there are various different approaches to representing this rotation, all of which have advantages and disadvantages.

Euler angles are probably the most intuitive representation of a 3D rotation, describing it as a series of three rotations about different axes. These axes are fixed by convention, so it suffices to specify the three angles of rotation. However, this scheme has a number of drawbacks [20, 24]: amongst other things, it is possible that rotation about one of the axes freezes during an animation ("Gimbal lock").

Rotation matrices are commonly used because they are well understood and allow efficient combination with other linear transformations (scaling and shearing – also translation if homogeneous coordinates are employed). However, ODEs over rotation matrices are difficult to implement correctly, since this representation uses nine numbers (a 3×3 matrix) to represent three degrees of freedom, thus introducing six additional side conditions which must be

maintained. Not doing so causes skew through numerical drift [20].

Quaternions [24, 5, 27] are a popular alternative to the two previous schemes, and they are used extensively in this project.

2.3.2 Definition and properties

Mathematically, quaternions can be regarded as numbers with one real part and three distinct imaginary parts:

$$\mathsf{q} = q_w + q_x\mathsf{i} + q_y\mathsf{j} + q_z\mathsf{k} \tag{2.2}$$

where q_w, q_x, q_y and q_z are real numbers and i, j and k satisfy

$$\mathsf{i}^2 = \mathsf{j}^2 = \mathsf{k}^2 = \mathsf{ijk} = -1. \tag{2.3}$$

From this follows that $\mathsf{ij} = -\mathsf{ji} = \mathsf{k}$ and $\mathsf{jk} = -\mathsf{kj} = \mathsf{i}$ and $\mathsf{ki} = -\mathsf{ik} = \mathsf{j}$. Note that multiplication is not commutative.

We will also need the conjugate and the inverse of a quaternion. In analogy to complex numbers, these are given respectively by

$$\overline{\mathsf{q}} = q_w - q_x\mathsf{i} - q_y\mathsf{j} - q_z\mathsf{k} \tag{2.4}$$

$$\mathsf{q}^{-1} = \frac{\overline{\mathsf{q}}}{|\mathsf{q}|^2} \tag{2.5}$$

$$\text{where} \qquad |q_w + q_x\mathsf{i} + q_y\mathsf{j} + q_z\mathsf{k}| = \sqrt{q_w^2 + q_x^2 + q_y^2 + q_z^2} \tag{2.6}$$

Sometimes we will need to relate a 3D vector to a quaternion with zero real part. For a given vector $\mathbf{u} = (u_1, u_2, u_3)^T$ we define the corresponding quaternion to be

$$\tilde{\mathbf{u}} = u_1\mathsf{i} + u_2\mathsf{j} + u_3\mathsf{k}. \tag{2.7}$$

The complex constants i, j and k are required for the algebra only, therefore we can represent a quaternion as four numbers (q_w, q_x, q_y, q_z). It turns out that a quaternion with unit magnitude ($|\mathsf{q}| = 1$) neatly represents an arbitrary rotation in 3D space, similarly to the way that an ordinary complex number represents a rotation in the 2D Argand diagram. The condition $|\mathsf{q}| = 1$ reduces the number of degrees of freedom to three, as required.

Every unit quaternion represents a rotation of angle θ about an arbitrary axis. If the axis passes through the origin and has a direction given by the vector $\mathbf{a}$ with $|\mathbf{a}| = 1$, the quaternion describing this rotation is

$$\mathsf{q} = \cos\left(\frac{\theta}{2}\right) + \tilde{\mathbf{a}}\sin\left(\frac{\theta}{2}\right). \tag{2.8}$$

It can easily be verified that this quaternion always has unit magnitude. It shall be assumed throughout this project that the rotation thus described is clockwise (as seen when looking in the direction of the vector $\mathbf{a}$) in a right-hand coordinate system, i.e. that it is given by the "right-hand rule".

Two rotations can be concatenated by multiplying their quaternions together. The order in which these rotations are applied is significant, and quaternion multiplication is not commutative, so the semantics match. The operations are, however, associative. The quaternion product is obtained simply by multiplying out the components, observing the rules for multiplying i, j and k:

$$\begin{aligned}(p_w + p_x\mathsf{i} + p_y\mathsf{j} + p_z\mathsf{k})(q_w + q_x\mathsf{i} + q_y\mathsf{j} + q_z\mathsf{k}) = \\ (p_wq_w - p_xq_x - p_yq_y - p_zq_z) + (p_wq_x + p_xq_w + p_yq_z - p_zq_y)\,\mathsf{i} + \\ (p_wq_y + p_yq_w + p_zq_x - p_xq_z)\,\mathsf{j} + (p_wq_z + p_zq_w + p_xq_y - p_yq_x)\,\mathsf{k}\end{aligned} \tag{2.9}$$

To rotate a vector $\mathbf{v} = (v_1, v_2, v_3)^T$ by a quaternion q, we first convert it into its corresponding quaternion $\tilde{\mathbf{v}}$ as defined in equation 2.7 and then calculate the quaternion product

$$\tilde{\mathbf{v}}' = \mathsf{q}\tilde{\mathbf{v}}\mathsf{q}^{-1} \tag{2.10}$$

If we expand this formula, we find that the real part of the result is always zero, and that the rotated vector $\mathbf{v}' = (v_1', v_2', v_3')^T$ corresponds to $\tilde{\mathbf{v}}'$ (i.e. $\mathbf{v}'$ is contained in the three complex parts of the quaternion product).

Some authors, notably Shoemake [24], choose to define the product in equation 2.10 in the reverse order. The choice is a matter of convention, since it merely changes the effect of this operation from being a clockwise to a counter-clockwise rotation. I chose the clockwise convention because it is consistent with the usual definition of the angular velocity vector in physics.

Observe that under this convention, if q is itself a product of quaternions $\mathsf{q} = \mathsf{q}_n\mathsf{q}_{n-1}\cdots\mathsf{q}_1$, the result is that of first applying the q_1 rotation, then q_2 etc. In other words, the rotations in a quaternion product are applied from right to left. To verify that this is the case, the identity $\overline{\mathsf{p}\,\mathsf{q}} = \overline{\mathsf{q}}\;\overline{\mathsf{p}}$ is useful [27].

2.3.3 Quaternion integration

We have seen that given the torques on a body, a numerical ODE solver can treat each component of the vector separately to obtain the new angular momentum. Now that we are representing orientation as a quaternion, how do we compute the change in orientation given the body's angular velocity?

The instantaneous rate of change of a quaternion q over time is usually [4, 6, 20] given as

$$\dot{\mathsf{q}}(t) = \frac{1}{2}\tilde{\boldsymbol{\omega}}(t)\mathsf{q}(t) \tag{2.11}$$

where $\tilde{\boldsymbol{\omega}}$ is the quaternion corresponding to the angular velocity vector $\boldsymbol{\omega}$. The quaternion $\dot{\mathsf{q}}$ can be fed into an ODE solver which can handle its four components separately. However, when this is done it is observed that the new orientation q' no longer has unit magnitude. This is an inherent property of the definition in equation 2.11, and not, as sometimes claimed [6], merely a matter of numerical round-off (proof in appendix B.1.1). Usually this problem is 'solved' by renormalizing the quaternion:

$$\mathsf{q}(t+h) = \frac{\mathsf{q}'}{|\mathsf{q}'|} \qquad \text{where} \quad \mathsf{q}' = \mathsf{q}(t) + h\dot{\mathsf{q}}(t) \tag{2.12}$$

(using Euler's method for clarity; q' would be appropriately redefined when using e.g. RK4). This 'solution' seemed quite *ad hoc* to me, and I demonstrated that it returns erroneous results if the angular velocity is large (see appendix B.1.2). I derive an exact algorithm for quaternion integration in appendix B.1. Such an algorithm may be important in aerospace applications, as the NASA patent [28] suggests.

2.4 Articulated bodies

2.4.1 Approaches to constraints

The methods developed so far allow the simulation of a single rigid body with forces acting on it. Further challenges arise when we consider multibody systems in which there are conditions

which must not be violated. In an articulated body, for example, each segment must be modelled as an individual rigid body, under the condition that joints must not be pulled apart.

There are various strategies for simulating articulated bodies:

- *Penalty methods* conceptually join bodies together with springs so that when they begin to separate, a restoring force causes them to move back together again. These methods are simple to implement initially but hard to get right: if the springs are too weak, the bodies will separate too far; if they are too stiff, very large forces can arise suddenly, causing the simulation to become unstable.

- Mechanics formulations using *generalized coordinates* [11, 9, 7, 29] build constraints firmly into the system: the number of generalized coordinates equals the actual number of degrees of freedom after the constraints have been applied. The state of the system is specified only in terms of these generalized coordinates, and they are chosen such that the system will always be in a legal state, irrespective of what the values of the coordinates are. These algorithms allow efficient and reliable computation. However, the equations are only tractable for so-called *holonomic constraints* (definition in [11]), which excludes many interesting systems.

- This project makes a compromise between the flexibility of penalty methods and the stability of generalized coordinates. As in the penalty method, we initially treat each segment of an articulated body separately. The constraints are then formulated in such a way that we can not only tell whether the system is currently in an illegal state, but also whether it is moving towards one, and even whether it is accelerating towards one. Using the method of *Lagrange multipliers* we can determine what forces must be applied to the bodies to nullify this unwanted acceleration before it ever occurs. This compromise comes at the expense of higher computational cost, because simultaneous linear equations have to be solved at run-time.

The mathematical formulation of Lagrange multipliers is derived in [4] and extended in [20] (also see [3] for an optimized algorithm), and I only state the results here.

2.4.2 Lagrange multipliers

Each constraint effectively reduces the number of degrees of freedom by one, and is expressed as a function c which is zero when the constraint is satisfied. (A ball-and-socket joint, which allows three modes of rotation but no separation, reduces the number of d.o.f. by three and is therefore expressed as a vector of three scalar constraint functions.) All constraint functions can then be concatenated into a single constraint vector $\mathbf{c}$.

The constraints can be satisfied by solving the equation[1]

$$-\mathsf{J}\mathsf{M}^{-1}\mathsf{J}^T\boldsymbol{\lambda} = \dot{\mathsf{J}}\dot{\mathbf{x}} + \mathsf{J}\mathsf{M}^{-1}(\boldsymbol{\Phi} + \boldsymbol{\Phi}_p) + k\mathbf{c} + d\dot{\mathbf{c}}. \tag{2.13}$$

The values of all variables except for the vector $\boldsymbol{\lambda}$ in this equation are known, and they are explained in appendix B.3.1. In brief, they are:

[1] Here the sign convention of [4] is adopted, which is the opposite of [20].

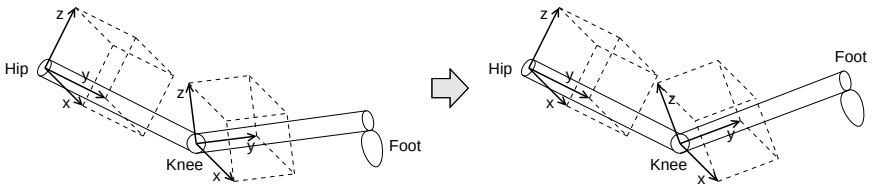

Figure 2.2: Two bones connected by a revolute joint, e.g. a knee. Rotation is constrained to be possible only about the local x axis of the knee, but not the local y axis (the lower leg itself) or the local z axis (sideways).

$\mathbf{c}$, $\dot{\mathbf{c}}$	Constraint vector and its first derivative w.r.t. time.
J, $\dot{\mathsf{J}}$	Jacobian matrices derived from $\mathbf{c}$ (see appendix B.3.3).
M	Mass-inertia matrix combining the masses and moments of inertia of all bodies in the scene.
$\dot{\mathbf{x}}$	Combined linear and angular velocity vector of all bodies.
$\mathbf{\Phi}$	Combined forces and torque vectors acting on all bodies.
$\mathbf{\Phi}_p$	Effects of free precession.
k, d	Scalar constants regulating error compensation, discussed in [4].

For us it will suffice to consider this equation to be a 'black box' which can be given to a linear equation solver to obtain $\boldsymbol{\lambda}$.

$\boldsymbol{\lambda}$ is an m-row vector of so-called *Lagrange multipliers* (after which this algorithm is named), where m is the number of constraints. Once we have solved for it, we can compute the expression

$$\mathbf{\Phi}_c = \mathsf{J}^T \boldsymbol{\lambda}. \tag{2.14}$$

$\mathbf{\Phi}_c$ is the vector of constraint restoring forces and torques for all bodies. Speaking in physical terms, these are the reaction forces which balance the action $\mathbf{\Phi}$. All we therefore need to do is to compute $\mathbf{\Phi} + \mathbf{\Phi}_c$ and to feed the result into our ODE solver, and the motion which ensues will satisfy the constraints. $\mathbf{\Phi}_p$ must not be added to this term[2].

2.4.3 Modelling the human body

The human body's selection of joint types is quite limited compared to the range of connectors found in machines [13]: for example, there are no sliding joints or screws. The simple types of joint found in human bodies are ball-and-socket joints allowing rotation about three axes (like shoulders and hips), and revolute joints limited to a single axis (like knees and elbows). The issue is made more complicated by compound joints like the spine, and motion which occurs along the length of a segment rather than at a joint (like rotating the wrist about the axis of the lower arm, which occurs through a relative shift of the ulna and radius bones [1]).

To make the model manageable, I chose to be anatomically ignorant and assumed that two adjacent segments of an articulated body are connected by a single joint, and that rotation occurs only about this joint. Each bone has a local coordinate system whose origin lies at the joint to the parent bone. This is illustrated in figure 2.2.

There are different ways of formulating this model. My approach is to assume initially that every joint is a ball-and-socket type. For those joints which are not, additional constraints are added to restrict the permitted axes of rotation.

[2] provided the moments of inertia are recalculated based on the bodies' orientation at every time step.

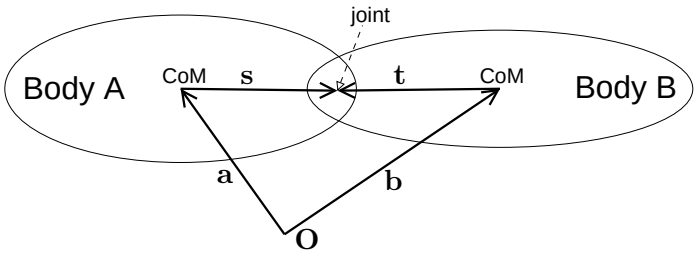

Figure 2.3: A valid ball-and-socket joint configuration. $\mathbf{s}$ is constant in body A's frame, while $\mathbf{t}$ is constant in the frame of body B. In the world frame, these two vectors therefore rotate according to their respective body's orientation.

2.4.4 Ball-and-socket joints

The position of a joint is a particular offset from the centre of mass in one body's frame, and a different offset from the other body's centre of mass. While the bodies may move around, these offsets stay constant (as seen in the body's frame).

Figure 2.3 gives an example: here, the configuration of bodies satisfies the ball-and-socket constraint iff $\mathbf{a} + \mathbf{s} = \mathbf{b} + \mathbf{t}$, i.e. if the joint has not separated. We can rewrite this to give the constraint function

$$\mathbf{c} = \mathbf{a} + \mathbf{s} - \mathbf{b} - \mathbf{t} \tag{2.15}$$

which equals $\mathbf{0}$, the three-dimensional null vector, iff the constraint is satisfied.

2.4.5 Rotation constraints

In the case of an elbow or a knee, we also need to prohibit the rotation about particular axes. After some thought I came up with the constraint function

$$c = \Re(\tilde{\mathbf{n}}\mathsf{p}^{-1}\mathsf{q}) \tag{2.16}$$

in which p is the quaternion of orientation for body A, q the quaternion for body B, and $\mathbf{n}$ a vector pointing along the prohibited axis in body A's frame. Setting c to zero ensures that no rotation occurs about the axis $\mathbf{n}$. If two different axes are prohibited with two separate constraints, then rotation is only possible about a single axis orthogonal to the two prohibited axes. Further details on this constraint are given in appendix B.4.3.

2.4.6 Angle limitation

The constraints described so far capture most of the features of the human skeleton, with one exception: if rotation is permitted about some axis, there is no limit to the amount of rotation that can occur. For example, permitting a human to look left and right would allow the simulation to rotate his head by an arbitrary amount. Imposing limits on the angle of rotation as well as the axis is possible but more complicated, and will be explained in section 3.3.4. It is worth noting that this is our first example of a non-holonomic constraint, and thus lies beyond the capabilities of generalized-coordinate approaches.

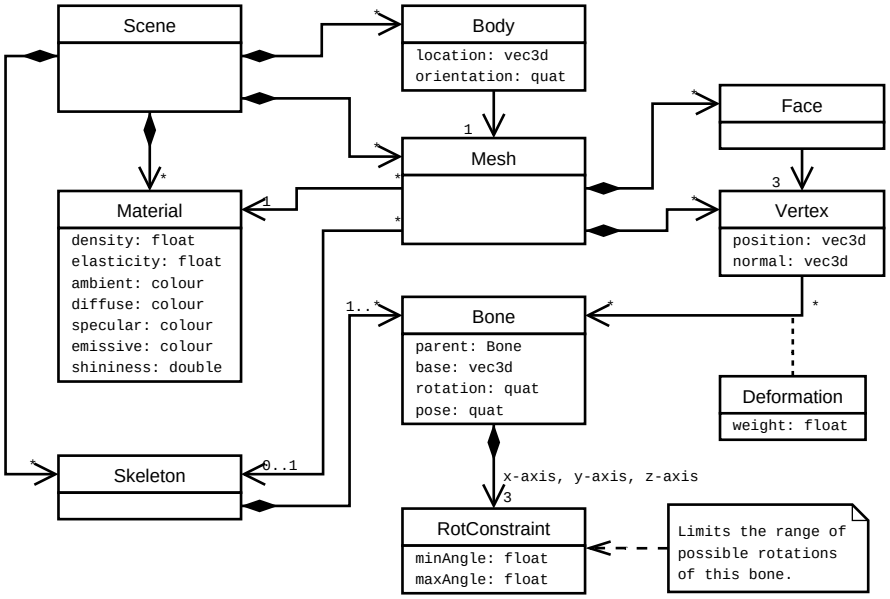

Figure 2.4: Structure of the input data to a simulation, using UML syntax.

2.5 Software preparation

2.5.1 Data structures and file formats

Two types of data files are mainly used in this project: an input file, specifying the bodies, their geometry and initial configuration in a scene, and an output file for the simulation results.

Input data

The input data has a graph structure which is visually represented in figure 2.4. Geometric objects are represented as *meshes* of triangles. (Polygons with a larger number of sides must first be split into triangles; this is always possible [25].) Each triangle (or *face*) of the object's surface is delimited by three *vertices*. Non-manifold or open meshes are not permitted because they would cause difficulties with the geometric algorithms and generally do not represent 'real' objects. Thus the mesh describes the surface of a polyhedron. This simple structure is sufficient to represent a great variety of shapes, and although a large number of vertices is necessary to adequately approximate curved surfaces, the ease of handling this data outweighs the costs caused by its quantity.

Several *bodies* in the scene may share the same mesh if they are identical up to rotation and translation. A mesh is also associated with a *material* which is used to determine the visual rendering (colour) and the physical behaviour (density, elasticity).

If the scene contains articulated bodies, *skeletons* must be defined in the file. A skeleton is a tree of *bones* in which each bone has an offset and a rotation relative to its parent. If one bone – say an upper arm – in this structure moves, all children of this bone – up to the tip of the little finger – will be rotated equally. This matches the natural behaviour of a skeleton. There can also be limits on the maximum angles of rotation for each joint; these are discussed in section 3.3.4.

In an articulated mesh, each vertex is associated with one or more bones of a particular skeleton. Normally it will be associated with one bone, but a weighted sum of several adjacent

bones' transformations may be used to create the illusion of smoothly deforming joints.

Since this data structure is designed to be very general-purpose, I wanted to use a file format which could easily be generated by 3D modelling applications or manually edited. I chose to use *XML*[3] as a basis since it is easy to read both by humans and by *XML* parsing libraries. I could not find a format which suited my needs – all existing formats are either far too complicated[4] or not powerful enough – and therefore designed a custom schema. I created scenes in this format by modelling them in *Blender* (see below) and exporting them to the *Cal3D*[5] format; since *Cal3D* contains less information than required by the simulation, I could semi-automatically convert it to my format using an *XSLT*[6] stylesheet, and then add the missing information manually.

Output data

The output of a simulation is much simpler than the input data: All that is required is the position and orientation, and possibly the velocities, of each body at each time step. For articulated bodies, the orientation of each bone is also required. This suggests a simple matrix layout in which bodies and bones are sorted along one dimension and time along the other.

The output file is such a matrix, arranged in a simple ASCII file format which can directly be imported by *Octave* (see below). *Octave* can then be used for further processing or evaluation of the results.

2.5.2 Tools

Programming languages

I chose *Java 1.5*[7] as language for the implementation of the bulk of the project. *Java*'s strictly object-oriented paradigm and its package concept facilitate the management of large software engineering projects; it is mature, well supported and benefits from an extensive run-time library. I particularly appreciate its good type safety (and hence bug prevention) compared to languages like *C*. I also make heavy use of the *generic* type polymorphism added in version 1.5.

Java is a verbose language though, so I decided that it would be beneficial to prototype the algorithmically tricky parts in a concise language first. For this purpose I used *GNU Octave*[8], a scripting language optimized for numerical computation on matrices and vectors. I implemented almost all numerical algorithms in this project in *Octave* first; this version was too slow to be of much practical use, but turned out to be extremely useful to guide the subsequent *Java* implementation. *Octave* was also used for processing of simulation output, for example to generate plots of the simulation behaviour.

Libraries

Rendered output is specified in the project requirements, and the most common choice for displaying 3D graphics in *Java* programs is the *Java3D*[9] library. *Java3D* provides a clean

[3] eXtensible Markup Language, `http://www.w3.org/XML/`

[4] for example X3D, `http://www.web3d.org/`

[5] Character Animation Library, `http://cal3d.sourceforge.net/`

[6] eXtensible Stylesheet Language with Transformations, `http://www.w3.org/TR/xslt`

[7] Using Sun's J2SE JDK 5.0, `http://java.sun.com/`

[8] GNU Octave – `http://www.octave.org/` – uses a language mostly compatible to MATLAB – `http://www.mathworks.com/products/matlab/`

[9] `https://java3d.dev.java.net/`

object-oriented interface to either a software renderer or accelerated graphics hardware, and is also quite well documented.

Processing of the input file was done using an *XML data binding tool* which I had developed independently in summer 2005. This tool takes a grammar for an *XML* schema in *RELAX NG*[10] format and generates a set of *Java* classes implementing a run-time representation of that schema. The actual low-level parsing is done by the *SAX*[11] parser implementation in *Xerces*[12].

Some of the numerical algorithms required by the simulation could have been obtained through libraries. However, none of them are particularly complicated – their *C* implementation in [17] is at most a few pages long and is easy to translate into *Java*. I therefore chose to implement the algorithms myself, giving me the freedom to combine them intimately (see section 3.1.3).

Tool chain

For *Java* development, I set up the *Eclipse*[13] environment, and a dual build system using the *Ant*[14] tool was kept as fallback in case of problems with *Eclipse*. To analyse the run-time performance of the *Java* implementation, *EJP*[15] was used.

I created input data for the simulation using *Blender*[16], an open source 3D modelling/animation/rendering application. *Blender* supports internal scripting in *Python*[17] to extend its functionality. This allowed me to write some short *Python* scripts which imported the results of a simulation as an animation back into *Blender*, acting on the original models. This was useful to review simulation results separately from the internal *Java3D* output. *Blender* can also export animations to the *YafRay*[18] raytracer to produce high-quality video files. These AVI files were converted to MPEG-1 video streams using *FFmpeg*[19].

Some types of constraint involve very messy algebraic expressions (appendix B.4). I found it useful to automatically generate optimized source code implementing these expressions using *Maple*[20].

Backup strategy

All work was done either on PWF machines or my own computer. All project files were kept under version control using *CVS*[21], and regular snapshots of the whole repository were transferred to the *Pelican* backup service. A working build environment was maintained both on the PWF and my computer so that work could continue seamlessly in case of disaster.

[10] `http://www.relaxng.org/`
[11] Simple API for XML, `http://www.saxproject.org/`
[12] Apache Xerces2 Java, `http://xerces.apache.org/xerces2-j/`
[13] `http://www.eclipse.org/`
[14] `http://ant.apache.org/`
[15] Extensible Java Profiler, `http://ejp.sourceforge.org/`
[16] `http://www.blender.org/`
[17] `http://www.python.org/`
[18] `http://www.yafray.org/`
[19] `http://ffmpeg.sourceforge.net/`
[20] `http://www.maplesoft.com/products/maple/`
[21] `http://www.nongnu.org/cvs/`

Figure 2.5: Alfred, one of the meshes used as input data. From left to right: The triangle mesh in wire-frame view; a raytraced perspective view; and the skeleton in its rest position.

Example scenes

I modelled all input data for the simulations in *Blender*. The most interesting mesh is that of *Alfred*, a humanoid articulated body (figure 2.5), which I created using human anatomical data [1]. It has 2295 vertices, 4524 faces and is bound to a skeleton of 25 bones, also shown in figure 2.5. There are some bugs in this mesh – for example, strange cracks appear at the shoulders when the arms are lifted high – but since creating good example data is not a primary objective of this project, these problems should be disregarded.

Dissertation

LaTeX was the definite choice for writing this dissertation because of its reliability, good handling of cross-references, beautiful layout and many other benefits. I created the figures using *Dia*[22], *Gnuplot*[23] and the *GIMP*[24].

[22] `http://www.gnome.org/projects/dia/`
[23] `http://www.gnuplot.info/`
[24] `http://www.gimp.org/`

Chapter 3

Implementation

In section 1.3, six core components of this project were identified. The first three – ODE solving, rigid body dynamics and articulated bodies – were introduced in the last chapter. In this chapter I give details of how the whole simulation was implemented in software, and which challenges I faced along the way (section 3.1). I then elaborate on the three remaining components – collision detection (section 3.2), and handling of colliding and resting contacts (section 3.3).

3.1 Casting theory into code

My final implementation comprises 700 lines of *Octave* code (prototyping and numerical tools) and 10500 lines of *Java* code for the main implementation (including 3500 lines automatically generated by the XML data binding tool), plus a few fragments in other languages.

3.1.1 The engineering process

The primary management challenge was to control the uncertainties caused by my lack of prior knowledge of the algorithmic details required. The first four milestones, up to collision detection, bore no major uncertainties, and could therefore be accomplished on schedule. I started surveying the options for implementing articulated bodies during milestone 2; at this point I realized that it was going to be much more challenging than expected, and that I would have to adapt my strategy.

I decided to suspend coding after milestone 4 and to work on the theory until I was certain that I understood it. If I tried to implement my ideas immediately, I believe the result would have been chaotic and a waste of time. I set myself a new hierarchy of tasks arranged in a rectangular grid. From left to right, the columns were the simulation components as listed in section 1.3 (except for item 4, which was already completed). From top to bottom, the rows were labelled

1. search for literature relevant to the subject;
2. read and completely understand it;
3. if it is insufficient and no more literature can be found, work out a new algorithm for doing it;
4. argue or prove that the algorithm works correctly;

5. write down a good explanation of the algorithm, as if it were for another person to read, to ensure I fully understand it;

6. write a prototype in *Octave*.

The idea of this grid is that each cell – one of these six tasks applied to one of the five simulation components – depends on those above and to the left of it. Thus I could start in the top left-hand corner and work towards the bottom right-hand corner; backtracking was possible when I got stuck, but there was a clear objective and I could track progress. I decided that only once I had covered the whole grid would I lay out and implement the *Java* program. Completing the grid took me seven weeks, from mid-December until the end of January.

The *Java* implementation, testing and debugging were completed in the following seven weeks, slightly less than the nine weeks originally allocated to this part of the project. Including the seven additional weeks required to complete the "theory grid", the project was now five weeks behind schedule.

Writing the dissertation, again a task with few uncertainties, took the allotted length of time. With five weeks delay carried over and one week of holiday, the project finished six weeks behind schedule. Given the early completion date (19 March) originally envisaged, this still allowed me to submit well before the deadline.

3.1.2 Application architecture

The *Java* implementation consists of five packages:

Scene. Facilities to load a scene description from an XML input file, and data structures to represent it at run-time. This package is automatically generated by the data binding tool.

Maths. General-purpose implementations of vectors, matrices (including sparse matrices), quaternions, solver for ordinary differential equations (variable-step-size Runge-Kutta with Cash-Karp parameters [17]), and a solver for systems of linear equations (biconjugate gradient method, also taken from [17]).

Geometry. Handling of triangle meshes at run-time: deformation of articulated meshes, rendering of meshes using *Java3D*, collision detection. (Depends on Scene and Maths)

Dynamics. Implementations of a rigid body and all the different constraint types. Architecture for handling interactions between objects. Articulated body code combining rigid bodies and constraints, and implementation of the collision/contact handling algorithms. (Depends on Scene, Maths and Geometry)

Main. Main application class and test cases. (Depends on Scene and Dynamics)

The application follows a clean object-oriented design throughout. Some excerpts from its class/interface hierarchy are shown in figure 3.1. This diagram indicates how extensible the architecture is: any two SimulationObjects can interact, generating an Interaction object – this could be anything from a repulsive Coulomb force to some sort of sentient behaviour – without having to change any of the rest of the system.

This design encapsulates algorithms in a way which is both clear and efficient. To give but two examples:

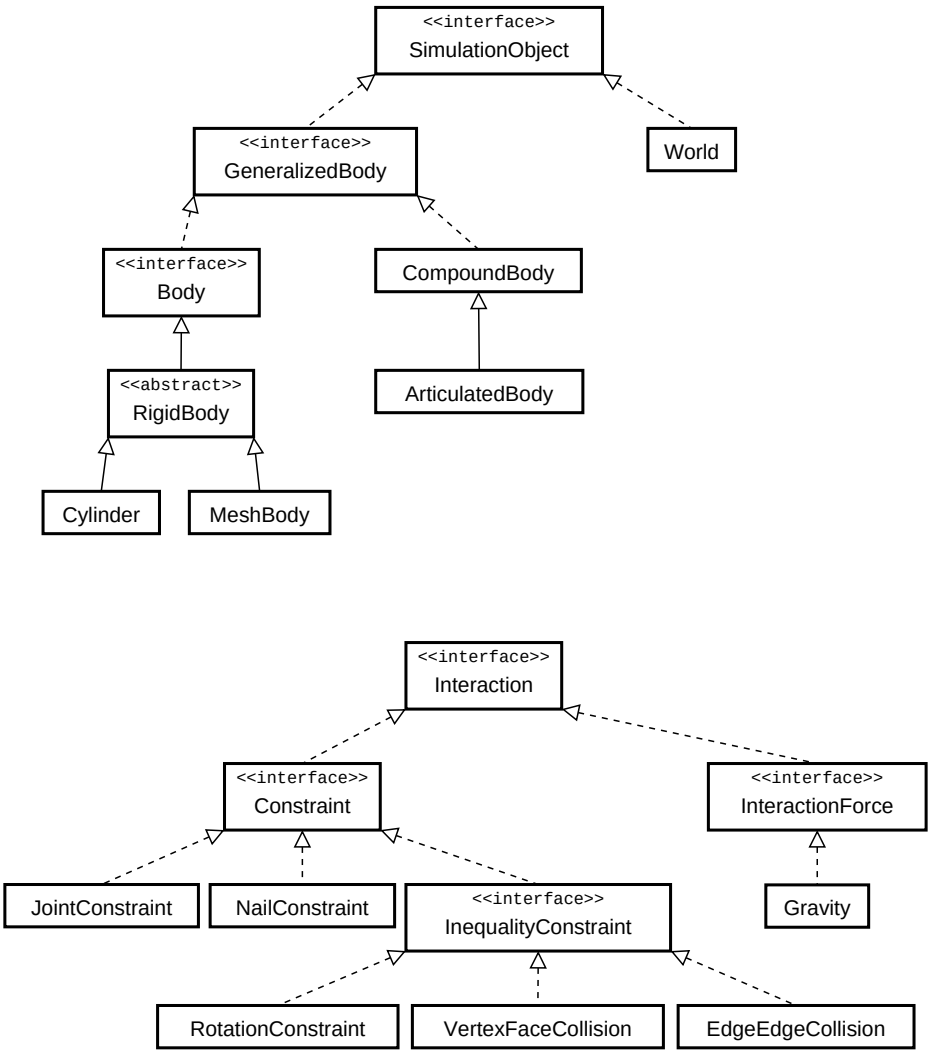

Figure 3.1: Two class inheritance hierarchies from the Dynamics package. Methods are omitted for better readability.

- the ODE solver operates on the Vector interface. The RigidBody's state vector implements the add method in such a way that quaternions are automatically normalized when appropriate. Thus the ODE solver needs to know nothing about quaternions, and a RigidBody object can be handled by any ODE solver.

- the biconjugate gradient method for solving linear equations only multiplies Matrix objects with Vectors. This operation is implemented efficiently for sparse matrices, so the algorithm can run quickly even though it knows nothing about sparsity itself.

3.1.3 Making it work

A flow chart of the main simulation loop is given in figure 3.2. The main algorithmic building blocks are indicated on the right margin. The order in which the steps is performed is important, and it took me several attempts to get this right – despite careful thought beforehand – because the bugs were quite subtle. The step size can be varied both by the ODE solver error estimation and by the Newton-Raphson collision time search. This would not have been possible with a library implementation of either algorithm.

There was only one major design mistake which I had to correct during the debugging phase: I had originally decided to store the current state of the simulation as values in each RigidBody object. This caused errors when the simulation had to backtrack and retry a step, because part of the body state had already been overwritten by the aborted step and could not be regained without introducing tight coupling with the ODE solver. I changed the architecture such that all time-varying state of a body is kept in a separate, immutable state vector, and all operations require the current state as an argument, returning a new state. These side-effect-free semantics made juggling different simulation states much easier, but the rewriting effort to incorporate this architecture change took almost a week – it would have been no difficulty if I had initially designed it this way.

I did not invest much effort in improving the run-time performance, but I did perform some profiling which showed that a very large proportion of the time was being spent in the biconjugate gradient solver for linear equations. I had originally implemented the *preconditioned* version of this algorithm described in [17], but noticed in the profiler that computing the preconditioner matrix was more expensive than its benefit. Removing the preconditioning immediately increased the execution speed of the whole simulation by a factor of 18.

3.1.4 Extensions

Being significantly behind schedule towards the end of the project, I decided not to add many optional features – I merely wrote a short but exceedingly useful script to import completed simulations into *Blender*, in order to create raytraced video files.

3.2 Collision detection

I now turn to the fourth main component of the system (as listed in section 1.3). An important requirement of the simulation is that bodies must not penetrate each other. In order to enforce this constraint, the program must first be able to detect penetration.

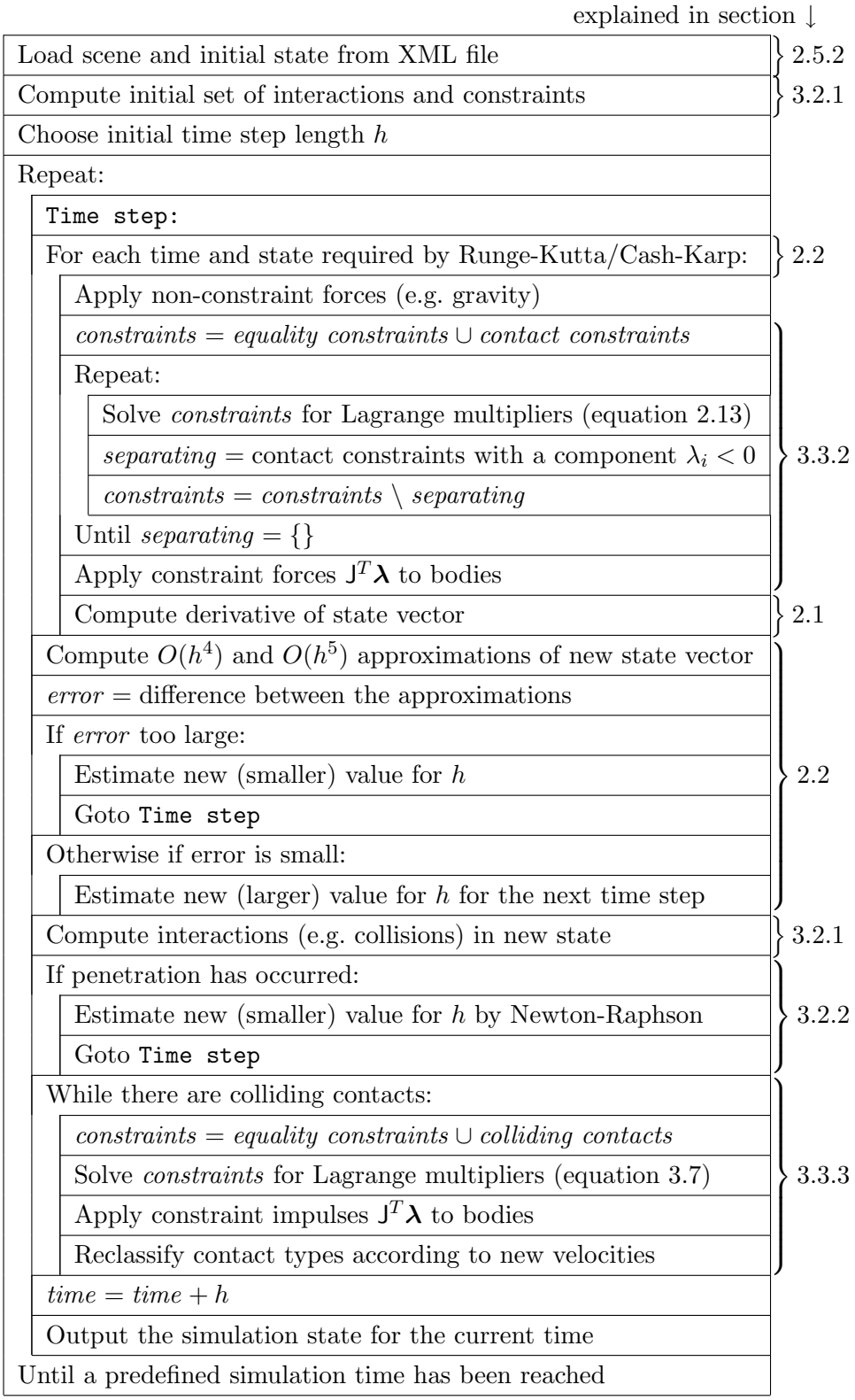

Figure 3.2: Overall (slightly simplified) structure of the simulation algorithm.

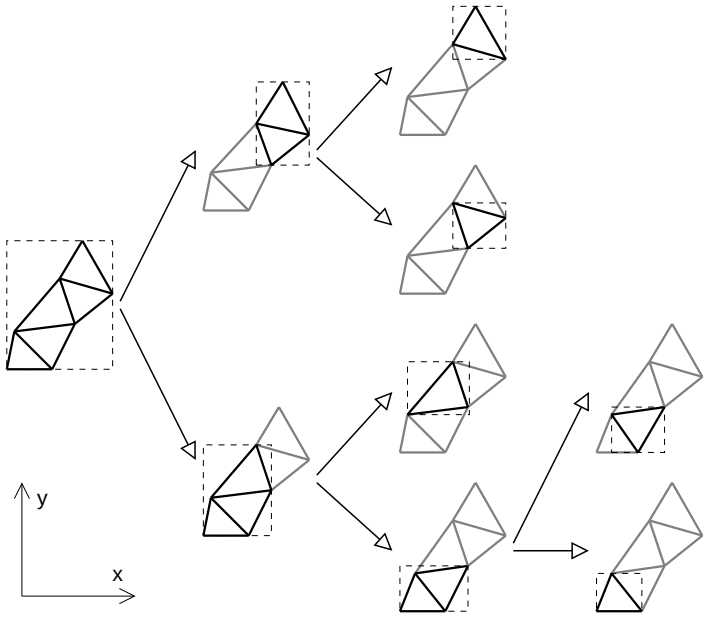

Figure 3.3: Example tree of axis-aligned bounding boxes (AABBs).

3.2.1 Intersection of triangle meshes

Collision detection determines which meshes are geometrically in contact, and what type of contact is present. The simplest algorithm for this purpose would be to take each triangle of one mesh and test it for intersection [16] against each triangle of the other mesh. This $O(N^2)$ algorithm is clearly very inefficient; most research in collision detection endeavours to create algorithms which are as close as possible to $O(1)$ complexity while remaining semantically equivalent.

Fast computation is only a minor concern in this project, therefore I shall describe my solution only briefly. This algorithm was suggested by Dr. Breton Saunders [21] and is simpler than most published algorithms, but nevertheless effective.

For each mesh, the closest-fitting Axis-Aligned Bounding Box (AABB) is computed. The set of triangles is split approximately in half along the axis of the longest bounding box side, and an AABB is computed for each subset of triangles (the two new AABBs may overlap). The subdivision continues recursively until each box contains only a single triangle. Thus we obtain a binary tree of bounding boxes (see figure 3.3).

AABBs can efficiently be tested for intersection. To determine whether two meshes are in contact, the boxes at the roots of their AABB trees are tested against each other. If the boxes intersect, all four pairings of their immediate children are tested. This procedure continues recursively as long as the boxes intersect. At the leaves of the tree, the actual triangles are tested for intersection.

3.2.2 Finding the time of contact

This collision detection algorithm returns nothing if meshes are separated merely by a very small distance. Thus it is likely that bodies which were separate in one time step are suddenly found to have interpenetrated in the next. There are elaborate solutions which predict the time of impact using proximity detection and the bodies' velocities. I take a simpler approach: if, at

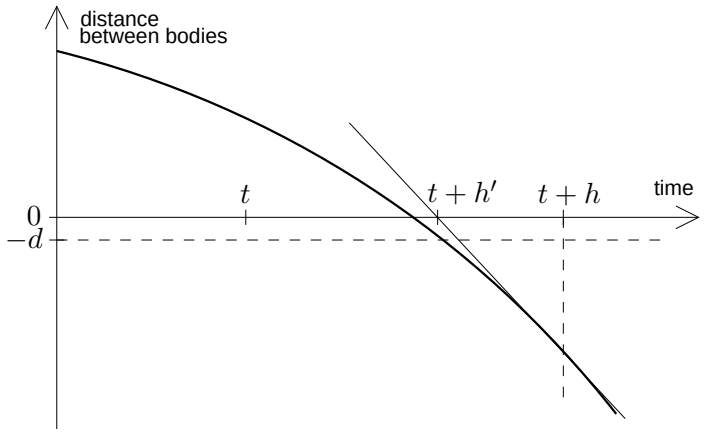

Figure 3.4: Estimating a new step length h' after penetration occurred when using step length h. The iteration stops when the penetration depth lies within the tolerance d.

some time, meshes have penetrated beyond some tolerance threshold, the current simulation step is discarded and retried using a smaller step length.

Retrying a step is quite expensive, so it is desirable to find the time of first contact with a minimum number of retries. The depth of penetration can usually be estimated from the collision geometry, and the velocities of the bodies are known from the simulation. Hence I can use the Newton-Raphson algorithm [17] for root finding (see figure 3.4) to make a good guess at the next step size. This method converges rapidly in most cases, but occasionally it is unstable, so my program automatically resorts to binary search [22] if Newton-Raphson diverges or converges too slowly.

3.2.3 Types of contact

We are now in a state where the polygon meshes are in non-penetrative contact, i.e. there is some intersection between their surfaces, but the intersection of their volumes is very small or zero. Between polyhedral bodies we can identify several basic types of contact, and the most common – *vertex/face* and *edge/edge* contact – are exemplified in figure 3.5. I have derived expressions for handling these two types of contact in appendices B.4.5 and B.4.6.

There are many other types of contact which occur, particularly when complicated meshes collide. It is unclear how these might best be handled; regrettably I could not find any results of research in this area. It seems intuitively plausible that it might be possible to decompose any contact between polyhedral bodies into a set of vertex/face and edge/edge contacts, as figure 3.6 demonstrates. However, no such algorithm is known to me, and my own attempts at creating one turned out to be unfruitful – the problem is extraordinarily difficult to tackle in general.

Instead, I chose to use a heuristic if a collision cannot be classified as either vertex/face or edge/edge: the colliding surfaces are approximated by spheres, planes and points, for which handling expressions can be derived. This approximation can cause unrealistic behaviour in some cases, but I have not found it to be a problem in practice.

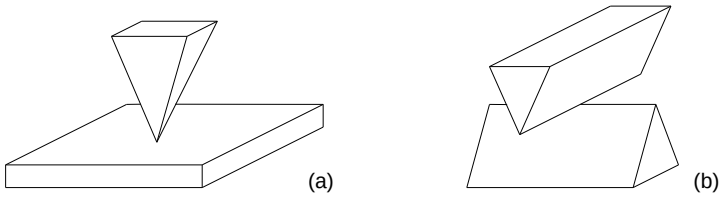

Figure 3.5: Basic types of polyhedral contact: *(a)* vertex/face, *(b)* edge/edge.

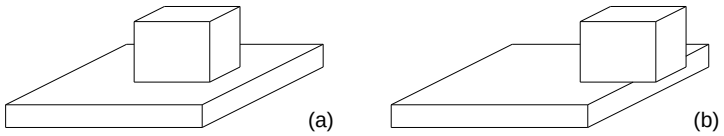

Figure 3.6: Two examples of composite contacts between polyhedral bodies: *(a)* four vertex/face contacts (one for each corner on the upper cube's bottom face), *(b)* two vertex/face and two edge/edge contacts.

3.3 Collision and contact handling

3.3.1 Overview

In the last section I explained how to detect contact between meshes. To prevent the bodies from penetrating each other during the simulation, the collision that has been detected must be handled in a physical way.

Let us distinguish two cases of contact between bodies: resting contact and colliding contact. Resting contact is given when the relative velocity at the contact point is zero (or very small) when projected onto the contact plane. Bodies in colliding contact are moving towards each other.

Creating a simulation involving contacts is generally seen as a difficult task. Baraff [4] derives equations to handle collisions, and outlines (with a number of errors) a way of handling resting contact. Unfortunately his collision handling does not work in combination with constraints, and his resting contact computation relies on a complicated and uncommon numerical routine. In this project, I developed a new method of handling contacts which is simpler to implement, more powerful, and works perfectly together with constraints like those required for articulated bodies.

3.3.2 Resting contact

Bodies in resting contact exert equal and opposite forces on each other so that they do not interpenetrate. This force must exactly balance the outside force, otherwise they will accelerate apart. The direction of this force is perpendicular to the plane of contact.

We need an algorithm which computes the contact forces in an arbitrary system, where there may be objects stacked on top of each other, different forces and torques acting, etc. Let us employ a constraint function c whose value is positive when the bodies are separated, zero when they are in contact, and negative when they have penetrated. In resting contact we have $\dot{c} = 0$. The function becomes interesting when we consider its second derivative, $\ddot{c}$. The non-penetration

constraint for a resting contact can be written as

$$\ddot{c} \geq 0. \tag{3.1}$$

More exactly, if it is the case that $\ddot{c} < 0$, we need to apply forces and torques to the bodies such that afterwards we have $\ddot{c} = 0$. When $\ddot{c} \geq 0$, no force is applied (otherwise we would be glueing the bodies together). But how do we solve this inequality in practice? At this point we diverge from Baraff's method [4], which expresses equation 3.1 and additional side conditions as a *quadratic program* and uses a specialized numerical solver.

Let us take a much simpler approach and pretend for now that (3.1) contained an equals sign. Then this is a constraint like any other, which we can satisfy by using the Lagrange multiplier method of section 2.4.2. Moreover, if there are multiple points of resting contact in the system, and also 'real' equality constraints like joints between bodies, we can solve all of these simultaneously without having to think twice.

However, having solved equation 2.13 for $\boldsymbol{\lambda}$, we need to take care. Observe equation 2.14, in which the constraint forces are computed:

$$\boldsymbol{\Phi}_c = \mathsf{J}^T\boldsymbol{\lambda} = \sum_{i=1}^{m} (\mathbf{J}_i)^T \lambda_i \tag{3.2}$$

Since each row of J is generated by a particular constraint, and the ith row of J is scaled with component λ_i of $\boldsymbol{\lambda}$ in the linear combination of constraint forces, we can say that each each component λ_i specifies the 'amount' of constraint i to apply to the system. Moreover, a positive value of the λ_i component generates forces and torques which push the bodies apart, while a negative value pulls them together (the opposite sign convention is equivalent, but one convention must be used consistently). It is such a negative λ_i that we must avoid.

This gives us a basis for an algorithm which turns (3.1) back into what it actually is – an inequality: set up constraint functions for all contact points, and solve the system of linear equations for the Lagrange multiplier $\boldsymbol{\lambda}$. In this vector, find all components relating to inequality constraints whose value is negative. (Equality constraints are left untouched whatever their $\boldsymbol{\lambda}$ component value is.) If there are none, calculate $\mathsf{J}^T\boldsymbol{\lambda}$ and add the forces and torques to the system – then we are done. If there are negative components, completely discard the constraints to which they belong (since the bodies are accelerating away from each other in this point, the contact will break in the next time step), then repeat the solving for $\boldsymbol{\lambda}$ with the reduced set of constraints. The process may take several iterations before terminating, but always terminates because in each iteration we either reduce the number of active constraints or exit.

I have not yet been successful in proving this novel algorithm correct, but despite extensive pondering and testing I have not found any examples in which it fails either.

3.3.3 Colliding contact

While points of resting contact had the property that $\dot{c} = 0$, colliding contact is characterized by $\dot{c} < 0$. (If $\dot{c} > 0$ then the bodies are separating. Such a contact can be ignored for now, but we cannot fully discard it, as explained below.) The bodies' linear and angular momenta must be modified instantaneously to satisfy $\dot{c}' \geq 0$, otherwise the bodies will interpenetrate in the next time step.

Physically, this operation is expressed in terms of an *impulse*. An impulse $\mathbf{u}$ has the same dimensions as linear momentum $\mathbf{p}$ (kg m/s) and is directly added to the momentum of the body to which it is applied: $\mathbf{p}' = \mathbf{p} + \mathbf{u}$. If the impulse is applied at the point $\mathbf{s}$, and $\mathbf{r}$ is the body's

centre of mass, it also changes the angular momentum to $\mathbf{L}' = \mathbf{L} + (\mathbf{s} - \mathbf{r}) \times \mathbf{u}$. This behaviour is analogous to that of a force.

A collision between two bodies must conserve the total momentum of the system [8]. This implies that the impulses acting on the bodies at a point of collision must be equal and opposite. However, the collision need not necessarily conserve energy. A collision which does conserve energy is called *fully elastic*, and may be given for example by an ideal rubber ball which, when dropped, bounces up to exactly the same height as it was dropped from. At the other extreme, an *inelastic* collision is one which reduces the energy of the system as much as possible – the bodies do not 'bounce apart' again.

Most real collisions are somewhere in between, depending on the materials of the colliding objects. The elasticity (also called *coefficient of restitution*) is a scalar $0 \leq \varepsilon \leq 1$, where $\varepsilon = 0$ indicates an inelastic collision, and $\varepsilon = 1$ full elasticity.

How do we now find the impulses we need to apply to the colliding bodies, and how do we simultaneously handle constraints? Once again we can use a Lagrange multiplier method, but a new equation must be derived which operates on impulses rather than on forces.

Set up a vector $\mathbf{c}$ of concatenated constraint functions including all equality constraints and all points of colliding contact. Points of resting contact and separating contacts must not be included. Now for all points i of colliding contact we have $\dot{c}_i < 0$. Remember that this value is the relative velocity of the bodies in the point of contact, projected onto the normal of the contact plane. Then the relative velocity $\dot{c}_i'$ after the collision is given by $\dot{c}_i' = -\varepsilon_i \dot{c}_i$. The components of the relative velocity parallel to the contact plane remain unchanged.

Assume for convenience that the first m components of $\mathbf{c}$ come from equality constraints, and that the remaining k components are due to inequalities. Then $\dot{\mathbf{c}}'$ is the required value of $\dot{\mathbf{c}}$ after the collision:

$$\dot{\mathbf{c}}' = \begin{bmatrix} \left.\begin{matrix} 0 \\ \vdots \\ 0 \end{matrix}\right\} m \\ -\varepsilon_1 \dot{\mathbf{c}}_{m+1} \\ \vdots \\ -\varepsilon_k \dot{\mathbf{c}}_{m+k} \end{bmatrix} \tag{3.3}$$

i.e. we require all equalities to be satisfied, and all inequalities to have 'bounced' by the right amount. We can rewrite this as $\dot{\mathbf{c}}' = \dot{\mathbf{c}} - \mathsf{E}\dot{\mathbf{c}}$, where E is the diagonal $m + k \times m + k$ matrix

$$\mathsf{E} = \begin{bmatrix} \underbrace{\begin{matrix} 1 & & \\ & \ddots & \\ & & 1 \end{matrix}}_{m} & & & \\ & 1 + \varepsilon_1 & & \\ & & \ddots & \\ & & & 1 + \varepsilon_k \end{bmatrix} \tag{3.4}$$

Let us now concatenate the linear and angular momenta of all n bodies into a single vector

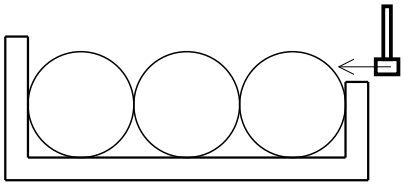

Figure 3.7: Fully elastic spheres are packed tightly in a rigid box, with no space between them or at either end. When one is tapped horizontally with a hammer, the impulse travels up and down the chain forever, always being reflected at the walls of the box.

$\mathbf{\Psi}$, like we previously did for forces and torques (equation B.15):

$$\mathbf{\Psi} = \begin{bmatrix} \mathbf{p}_1 \\ \mathbf{L}_1 \\ \vdots \\ \mathbf{p}_n \\ \mathbf{L}_n \end{bmatrix} \tag{3.5}$$

From the definition of M^{-1} (equation B.16) we can deduce that $\dot{\mathbf{x}} = \mathsf{M}^{-1}\mathbf{\Psi}$. We are seeking a vector $\mathbf{\Psi}_c$ which, when added to $\mathbf{\Psi}$, causes the constraints to return the value $\dot{\mathbf{c}}'$:

$$\dot{\mathbf{c}} = \mathsf{J}\dot{\mathbf{x}} = \mathsf{J}\mathsf{M}^{-1}\mathbf{\Psi} \quad \Longrightarrow \quad \dot{\mathbf{c}}' = \mathsf{J}\mathsf{M}^{-1}(\mathbf{\Psi} + \mathbf{\Psi}_c) \tag{3.6}$$

(cf. equation B.19). It turns out[1] that this vector can be written as $\mathbf{\Psi}_c = \mathsf{J}^T\boldsymbol{\lambda}$ for some $m+k$-row vector $\boldsymbol{\lambda}$. Substituting the previously defined expressions into equation 3.6:

$$\begin{aligned} \dot{\mathbf{c}}' &= \mathsf{J}\mathsf{M}^{-1}(\mathbf{\Psi} + \mathsf{J}^T\boldsymbol{\lambda}) \\ \dot{\mathbf{c}} - \mathsf{E}\dot{\mathbf{c}} &= \mathsf{J}\mathsf{M}^{-1}\mathbf{\Psi} + \mathsf{J}\mathsf{M}^{-1}\mathsf{J}^T\boldsymbol{\lambda} \\ \mathsf{J}\mathsf{M}^{-1}\mathbf{\Psi} - \mathsf{E}\mathsf{J}\mathsf{M}^{-1}\mathbf{\Psi} &= \mathsf{J}\mathsf{M}^{-1}\mathbf{\Psi} + \mathsf{J}\mathsf{M}^{-1}\mathsf{J}^T\boldsymbol{\lambda} \\ -\mathsf{J}\mathsf{M}^{-1}\mathsf{J}^T\boldsymbol{\lambda} &= \mathsf{E}\mathsf{J}\mathsf{M}^{-1}\mathbf{\Psi} \end{aligned} \tag{3.7}$$

All variables in equation 3.7 are known except for $\boldsymbol{\lambda}$, so this is just a system of linear equations which we can solve for $\boldsymbol{\lambda}$. The resulting vector $\mathbf{\Psi}_c = \mathsf{J}^T\boldsymbol{\lambda}$ of impulses can be directly added to the rigid bodies' momenta.

We are not quite finished yet. Having modified the bodies' momenta, the state of other contact points in the system may have changed. While the previously colliding contacts will have turned into resting or separating contacts, other (previously unconsidered) contacts may now be colliding! Thus we must repeat the whole process described in this section, ignoring the previously colliding points, and considering the new ones instead. If there are no new colliding contacts, we have finished.

The bad news is that this algorithm is not guaranteed to terminate – indeed figure 3.7 shows an example in which it will run indefinitely. An easy way of solving this problem is by not allowing ε to be too close to 1; then if a loop occurs, the system will eventually run out of energy and return to a resting contact.

[1] The argument is analogous to the derivation of the force vector in terms of Lagrange multipliers; see [4] for details.

3.3.4 Generalized collisions

In section 2.4.6 I mentioned that the simulation still needed a way of limiting the range of valid angles for a joint of an articulated body. I could not find any literature explaining how one might realize such a constraint, but I was excited to notice one day that I already had all the tools in hand, and merely needed to combine them.

An angle limitation is simply another type of inequality constraint. Usually two constraints are required for each axis about which rotation is possible:

$$\begin{aligned} c_1 &= \textit{current angle} - \textit{minimum angle} \\ c_2 &= \textit{maximum angle} - \textit{current angle} \end{aligned} \tag{3.8}$$

The two inequalities $c_1 \geq 0$ and $c_2 \geq 0$ then require the current angle to always stay within the range bracketed by the minimum and the maximum. Once c_1 and c_2 have been formulated in an appropriate way, these inequalities can be given directly to the algorithms for resting and colliding contact, alongside any other constraints generated by meshes in contact or by joints. These algorithms will ensure that all constraints are satisfied simultaneously (irrespective of their origin), and the result actually behaves just as expected!

The continuation of the good news is that I have already given a suitable formula to determine the current angle of rotation about a particular axis in equation 2.16 (section 2.4.5). This formula can simply be reused for inequalities.

There is just one problem: it is fiendishly hard to visualize what a particular restriction on a 3D rotation actually means. The best I could come up with is to imagine a 3D phase space in which each coordinate is the angle of rotation about one of the segment's local (Cartesian) axes. The set of valid rotations for one joint is then a subspace of some shape; despite the linear look of inequalities 3.8, this space is generally not polyhedral; in fact it is often not even compact. Maple cannot visualize such a solution space of inequalities, so I wrote myself a little program for this task (figure 3.8) and used it to set up the joint limits for Alfred's skeleton.

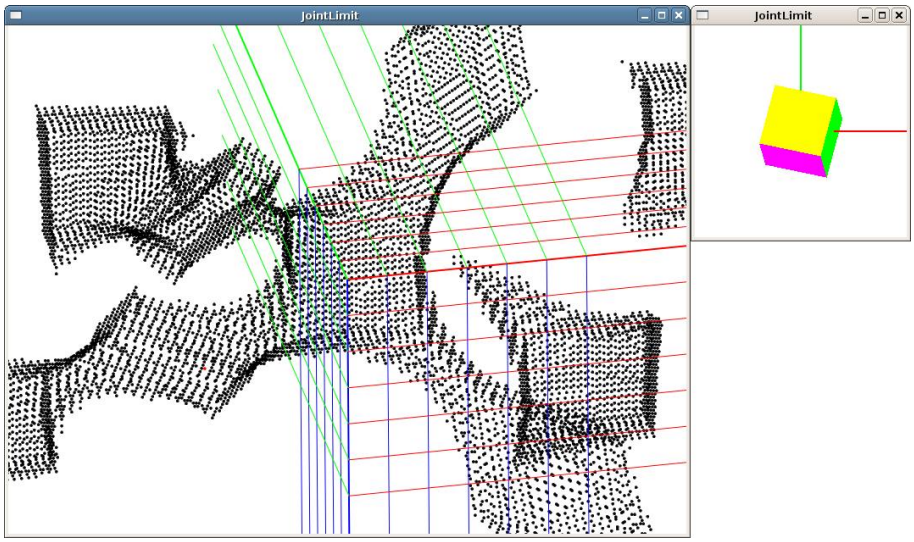

Figure 3.8: Screenshot of a tool to visualize the solution space of simultaneous inequalities in three variables (here, the rotation about three different axes). If a point in the solution space is selected, the three angles of rotation corresponding to this point are applied to the cube on the right-hand side; this allows the inequalities to be adjusted to fit the desired set of rotations.

Chapter 4

Evaluation

The first three chapters explain in detail how the simulation application was developed. I now turn to its evaluation and explain how I showed that the program works as required.

4.1 Testing strategy

An application as large as the one developed in this project is almost impossible to get right entirely by advance planning. Although I took much care during the preparation and implementation, I anticipated that testing and debugging would be necessary.

As mentioned in section 3.1.1, I prototyped most numerical algorithms in *Octave* which allowed rapid interpretation of results through its plotting facilities, and hence a rapid edit–test–debug cycle. For each major algorithmic part of the project I modelled a physical system which placed a particular emphasis on one part of the simulation, thus allowing me to test and debug each feature before working on the next feature:

- a gyroscope (section 4.2.1) to test rigid body dynamics,
- a double pendulum to test articulated bodies/constraints,
- Newton's cradle (section 4.2.2) to test colliding contact handling,
- a simulation of boxes falling onto a table to test resting contact handling.

I developed the *Java* version of each feature only after it was working to satisfaction in the *Octave* prototype. The *Java* implementation usually introduced new bugs, which I could locate by implementing the same test cases in *Java*, and using step-by-step comparison with the *Octave* computation. This testing and debugging strategy turned out to be fruitful and effective.

4.2 Quantitative evaluation

I performed several quantitative tests on the program by simulating simple mechanical systems and comparing their numerical outcome to the physical predictions.

4.2.1 Gyroscope simulation

The first set of tests simulates a *gyroscope* (figure 4.1), which was also used as a general test case (section 4.1). A gyroscope consists of a single rotating rigid body and a 'nail' constraint

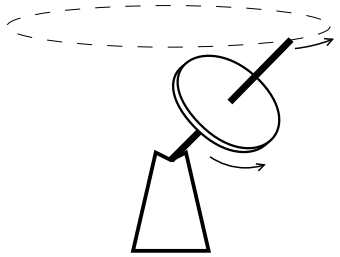

Figure 4.1: Schematic drawing of a gyroscope. The disc rapidly rotates about its own axis, and gravity causes a slower precession movement (shown as a dotted line) about a vertical axis.

(appendix B.4.1) holding one end of its axis in place. In a gravitational field the gyroscope exhibits a precession movement. Although this behaviour seems counter-intuitive at first, it can be characterized analytically [12]. There are not many interesting systems of rigid bodies which have an exact solution, so a gyroscope is a good choice for quantitative evaluation.

I set up initial conditions similar to the values one might find in a toy gyroscope (20 revolutions per second about the gyroscope axis, one full circle of precession in 8 seconds). I then ran the simulation for 8 s, using an average time step length of about $2.3 \cdot 10^{-4}$ s. The simulation performed one full circle of precession in 7.953 s, which is within 0.6 % of the theoretical value. Over the course of 8 s, the body rotated by 320.36π radians about its own axis, which differs from the theoretical value by only 0.1 %. These errors varied little even in simulations using larger time steps. The effects of nutation[1] were small for the chosen initial conditions but may have contributed towards the errors.

It is interesting to also observe a different error, namely the amount by which the constraint drifts apart. Usually this drift is compensated in the Lagrange multiplier method so that it never manifests itself, but temporarily deactivating this correction[2] makes the error introduced by the ODE solver observable.

Figure 4.2 shows by what distance the gyroscope's 'nail' constraint drifted apart after 8 s of simulation time, for a wide range of different step sizes. There are some noteworthy features about this plot:

- The logarithmic axes are scaled such that one order of magnitude in the horizontal has the same length as five orders of magnitude in the vertical. Observe that in this scaling, the solid line (error per time step) is an almost perfect straight line with gradient 1. This shows that the error is indeed an $O(h^5)$ function of the step size, as expected.

- Over a wide range of step sizes, the plot of the total accumulated error is parallel to the solid line. This means the total error is also $O(h^5)$, which is even better than expected: although the approximation in each time step is $O(h^5)$, the number of steps required is inversely proportional to the step length, so one might expect a larger overall error. This relationship indicates that the ODE solver's target error can in fact be used as a reliable estimate of the overall error to within a constant factor.

- As step sizes h become very small – below about $3 \cdot 10^{-4}$ s – the error in each step continues to scale order $O(h^5)$, but due to the huge number of steps, the accumulated error cannot

[1] Having nothing in common with *mutation*, *nutation* is an oscillation about an axis orthogonal to the two main axes of rotation. [8]

[2] by setting $k = d = 0$ in equation 2.13.

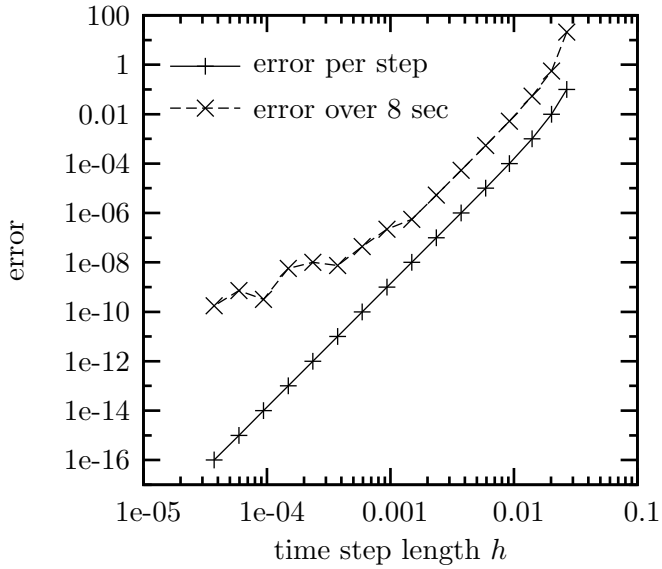

Figure 4.2: Errors introduced by the ODE solver for different step sizes h, as observed in the gyroscope simulation. Solid line: difference between $O(h^4)$ and $O(h^5)$ Runge-Kutta approximations for each time step. Dashed line: cumulative drift of the gyroscope's 'nail' constraint after 8 s simulation time.

be reduced much further. However, the errors here are in the range of nanometres, so they should be of little concern for computer graphics purposes.

In summary, the results for the simple gyroscope simulation inspire confidence that the implementation is reliable and will continue to produce realistic results for complicated systems which lack an exact solution. They also show that the target error can conveniently be adjusted to match the requirements, because more CPU time does – within sensible bounds – buy higher accuracy.

4.2.2 Collision handling

The analysis in the last section is relevant for continuous systems, but says nothing about simulations involving collisions. For this purpose I simulated a different kind of physics toy, *Newton's cradle* (figure 4.3). This system does not have an exact analytical solution, but it does have characteristic behaviour patterns which may be observed.

Newton's cradle works best when the elasticity is large ($\varepsilon \approx 1$). I simulated it using $\varepsilon = 1.0$ and $\varepsilon = 0.9$, with one ball initially raised and the other four at rest. The comparison of the two simulation results is shown in figure 4.4. The energy is calculated as the sum of potential, linear and angular kinetic energies of all five balls, with zero potential when all balls are at their equilibrium position. Fully elastic collisions conserve energy in the simulation (constant to within 1 part in 10^8), while imperfect collisions instantaneously dissipate energy.

The momentum of balls 2–4 stays zero (within 1 part in 10^{10}, except for transient peaks, which are immediately neutralized again) with full elasticity; with $\varepsilon = 0.9$, they increasingly begin to swing, as expected. The bottommost plots in figure 4.4 show how the step size is reduced to find the exact time of collision, and large time steps are taken otherwise.

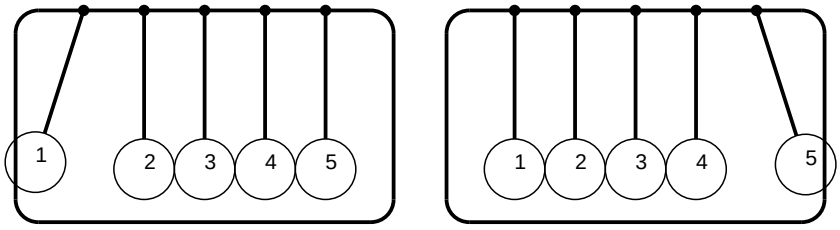

Figure 4.3: Newton's cradle. By conservation of momentum and energy, if k balls collide with one end of the chain of balls, the same number of balls bounce up on the opposite side. The other balls stay stationary.

All behaviour exhibited by this system matches the behaviour observed in reality, so it constitutes a good demonstration that the algorithm of section 3.3.3 works correctly: it respects the constraints which attach the balls to the frame of Newton's cradle, and it propagates impulses along the chain of contacts. The simulation works equally well with more than one ball in motion.

4.2.3 Run-time cost

Profiling of the application revealed that the simulation spends about 93 % of its time running the biconjugate gradient algorithm to solve the constraint equations. To round off the quantitative evaluation, I wanted to know how the computational cost relates to the size of the problem.

The biconjugate gradient algorithm is an iterative procedure which stops when some error criterion is met. In theory, if exact arithmetic was being used, a solution would always be found in $O(N)$ iterations for a system of N constraints[3] [17]. Each iteration requires a constant number of multiplications of a matrix with a vector, and these multiplications dominate (86 %) the cost of the algorithm. Such a multiplication has a cost of $O(N^2)$ for a general matrix, but only $O(N)$ for the type of sparse matrix we are dealing with. Hence a theoretical estimate of the overall cost of the algorithm would be $O(N \cdot N) = O(N^2)$.

In practice, the algorithm converges more slowly when using floating-point arithmetic. I measured the ratio of CPU time to simulation time for a range of different-sized systems and found an overall relationship of about $O(N^{2.5})$. Note that the use of sparse matrices still causes a significant benefit, since a simpler implementation would require the same number of iterations and hence be about $O(N^{3.5})$. In absolute terms, a small simulation – of a double pendulum, say – runs almost in real-time on my PC[4], while one with 100 constraints requires approximately one hour of CPU time per second of simulation time.

4.2.4 Numerical stability

I had very few problems with numerical stability throughout this project. The ODE solving turned out to be very robust; this is particularly satisfying since ODE stability would have been the greatest problem if a penalty method had been used instead of Lagrange multipliers (section 2.4.1). I occasionally observed divergence of the biconjugate gradient algorithm; this seemed to occur only if there were contradictory constraints in the system. In complicated collision geometries such contradictions did sometimes occur. I solved this problem by keeping

[3] or rigid bodies, but since we are dealing with articulated bodies, we can assume a linear dependence between the numbers of bodies and constraints.

[4] AMD Athlon XP 2000+

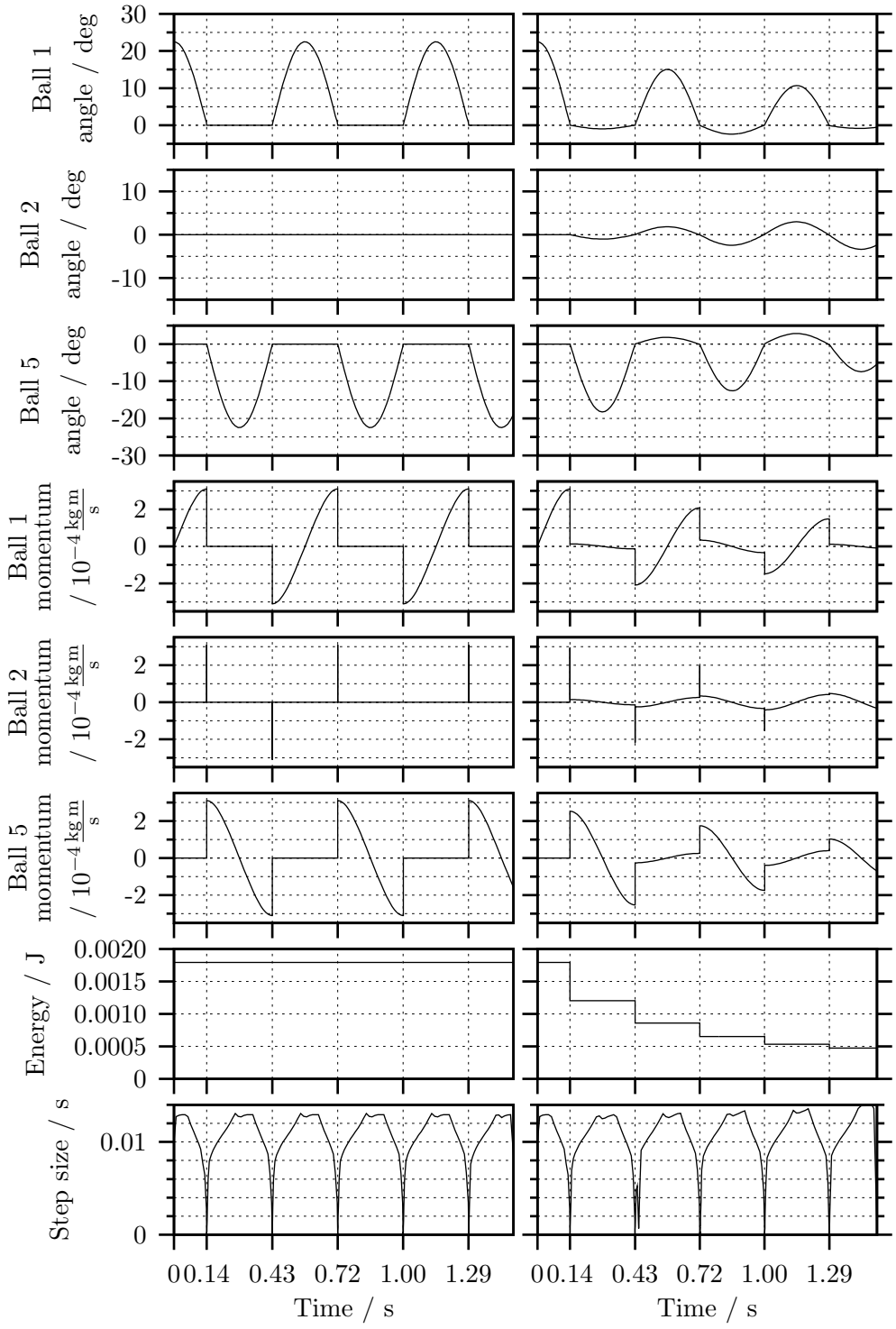

Figure 4.4: Plot of various time-varying properties of Newton's cradle. Left column: using ideal, fully elastic collisions ($\varepsilon = 1.0$). Right column: imperfect collisions ($\varepsilon = 0.9$).

track of the approximate solution with the smallest error amongst all iterations; if the algorithm starts diverging, it is aborted and the 'best guess' is used. This procedure is not mathematically justified, but in all my simulations it produced good results.

4.3 Simulation examples

In the end, the purpose of this project is to produce animations for computer graphics, and the last word must be given to the human observer who watches these animations. I would like to claim that the results are of a quality similar to what an animator might create, and that they are realistic in the sense that they do not violate the laws of physics or anatomy. However, such a claim is difficult to defend convincingly in print. I have therefore compiled a video of several example animations, and I urge the reader to watch it to gain a more personal impression of this project's results. The video can be found in the 3D graphics section of my web site `http://martin.kleppmann.de/`. The following sections describe how each of the simulated systems was implemented and point out noteworthy features.

4.3.1 Pendulum systems

The simplest physical system in the demonstration video is the animation of a double pendulum (a rigid pendulum with a second one attached to its end), and its extension to three, eight and 25 segments. The double pendulum is a commonly studied system in physics; although it does not have an exact analytic solution, an approximate solution can be found for small angles. In this case, one finds that the resulting movement is the sum of two simple harmonic oscillations at different frequencies, resulting in a mysterious-looking swinging movement.

The double, triple and eight-part pendulums are set up in a very similar way. Each segment is modelled as a rigid cylinder with constant density. The top end of the top segment is held in place by a 'nail' constraint. Each pair of adjacent segments is connected by a ball-and-socket joint. Initially, all segments are at rest, the first segment is rotated by 45 degrees anticlockwise from the equilibrium position, and all other segments hang straight down.

The simulation does not employ an XML input file; instead, the objects representing the bodies and the constraints are directly created by the Java test case. The simulation results were exported to Blender and rendered using the internal renderer and orthographic projection. I also performed a Fourier transform on the results of the double pendulum simulation, which indicated very clearly that there were two main frequency components.

The 25-segment pendulum is an attempt to simulate rope, and it is considerably more ambitious. It is modelled as a cylindrical mesh bound to a chain of 25 bones. The ground is a separate mesh, and in the simulation its position is fixed by three 'nail' constraints. This input data is represented as an XML file. Collision detection is performed on basis of the meshes, thus the rope can collide both with itself and with the ground.

The joints between adjacent segments of the rope are of ball-and-socket type, and their rotation is limited to a maximum of 15 degrees about each axis. The simulated rope is therefore very stiff. I am not entirely sure how realistic I should consider this simulation; its behaviour looks strange at first, but this is mainly due to the assumptions put into the model. I believe that the simulation correctly obeys the model, but the model would have to be refined in order to produce genuinely realistic-looking rope.

4.3.2 Newton's cradle

The next section shows Newton's cradle in a range of different animations. For full elasticity, various combinations of one, two and three balls in simultaneous motion are demonstrated, and all effects observed in a real toy cradle are reproduced. I also simulate what happens if collisions are not fully elastic: all balls gradually begin to swing, and the sequence of collisions becomes much more complicated.

Newton's cradle is modelled as five spherical bodies, each with its centre of mass at the centre of the ball (i.e. the wire connecting it to the frame is assumed to be massless). The joint to the frame is modelled as a 'nail' constraint, and the frame itself does not exist in the simulation. Collision detection is performed not on the basis of the meshes (since these are polyhedral approximations and not exact spheres), but using a special type of sphere/sphere constraint which computes its behaviour using the positions of the centres of the balls and their radii.

For the simulation to work accurately, a very low penetration tolerance must be set. The different examples are simulated in exactly the same way, independent of the number of balls moving or the elasticity of collisions.

4.3.3 Falling boxes

The third section demonstrates eleven interacting rigid bodies: ten boxes falling onto a table. The simulation is performed at two different elasticities: one close to what one would expect in reality ($\varepsilon = 0.2$), and one very bouncy as contrast ($\varepsilon = 1.0$).

Each of the ten boxes is modelled as a symmetric hollow body. All boxes and the table are specified as meshes in an XML input file. The table is held in place by three 'nail' constraints. These simulations exhibit a range of different collision types: each example starts with a vertex/face collision when the first box hits the table, and is followed by a sequence of vertex/face, edge/edge and compound collisions. Whenever a collision cannot be clearly classified as either vertex/face or edge/edge, the algorithm searches for a plane which contains as closely as possible the line of intersection between the two colliding meshes. This plane is then used as the contact plane for computing the impulses and resting contact forces.

The fully elastic version of this simulation looks unrealistic because it behaves as though the boxes were made of extremely bouncy rubber. The version with elasticity 0.2 is much closer to what one might expect in a real-life scene, but the contrast between the two is interesting. At the end of the low-elasticity simulation, several boxes can be seen in a frictionless glide over the surface of the table. They cannot penetrate the table due to a resting contact force.

4.3.4 Alfred falling down stairs

The final two sections show a humanoid model fall down a straight staircase and a spiral staircase respectively. I created these simulations to demonstrate a situation which did not require control of muscular forces (which are hard to control in a simulation [10]) but were nevertheless interesting.

The simulation on straight stairs and on the spiral staircase are set up in a very similar way. Each scene consists of two bodies, a polygon model of the staircase and Alfred himself. In each case, the staircase is held in place by three 'nail' constraints[5]. Alfred is an articulated body as

[5]It seems intuitively bizarre that one might make a whole staircase immobile by only three nails, but in the simulation the magnitude of the forces is, of course, almost irrelevant!

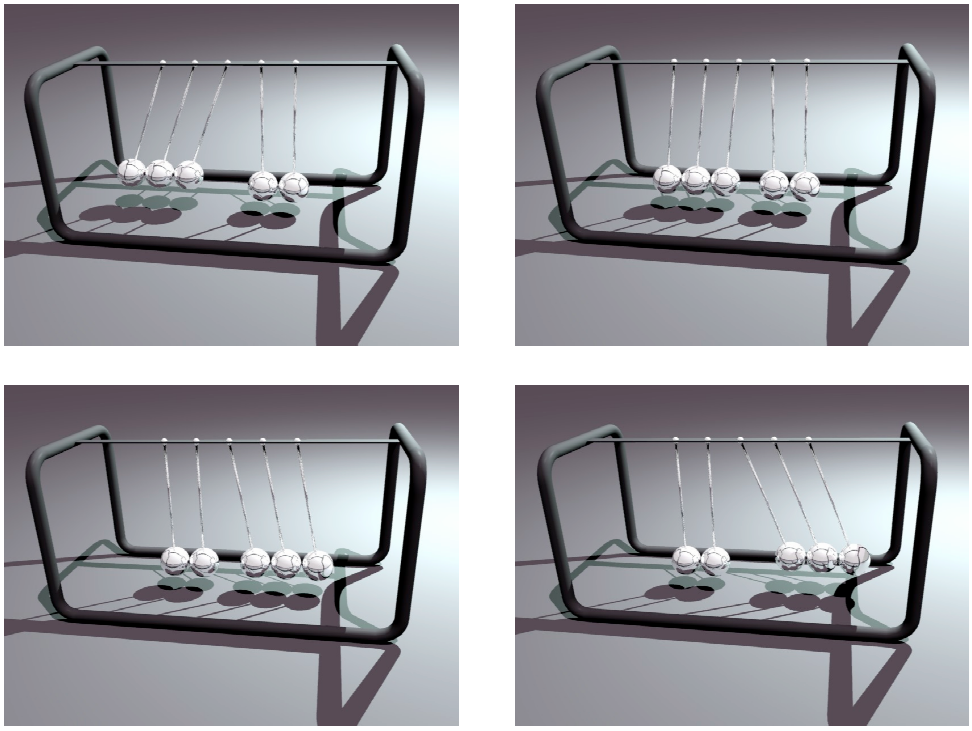

Figure 4.5: Animation of Newton's cradle.

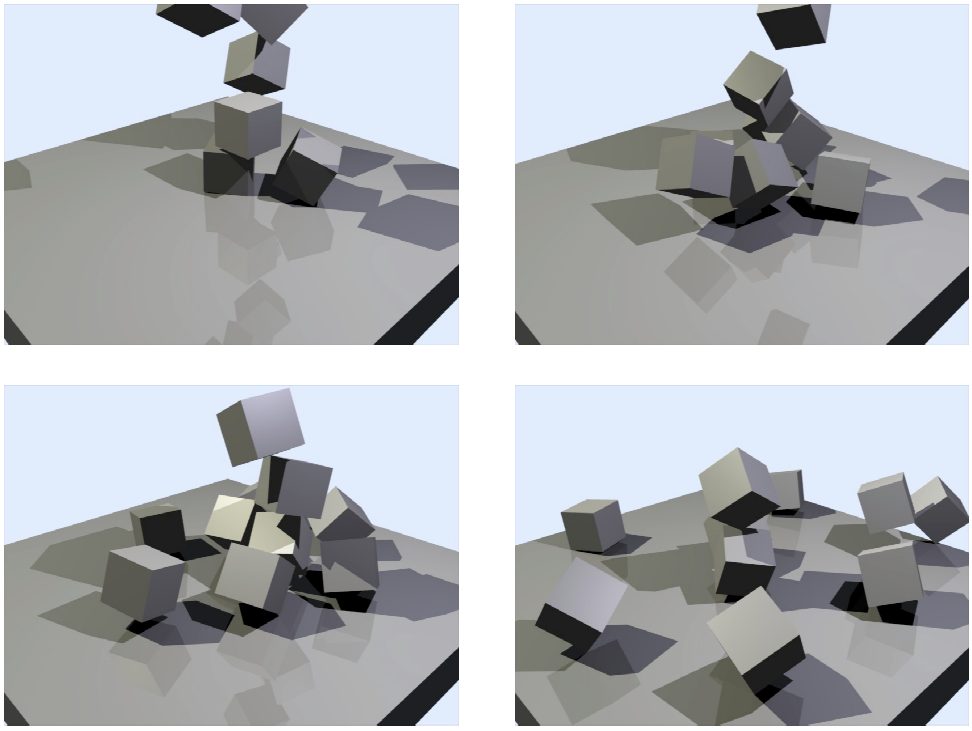

Figure 4.6: Animation of ten boxes falling onto a table.

described in section 2.5.2.

All meshes and skeletons are defined in an XML input file. All joints' rotation is restricted in a way which approximates the anatomical reality. Nonetheless the body sometimes enters poses which, although they are not impossible, look rather uncomfortable. This is because a pose does not gradually become more uncomfortable as the limit is approached, but action takes place only at the limit. The articulated body model would have to be extended to accommodate a notion of "comfort" or elastic limits.

For purposes of collision detection, the shape of the Alfred mesh is approximated by 252 little spheres. The staircase is not approximated. Thus this simulation does not make use of the usual vertex/face and edge/edge collisions; instead, sphere/face and sphere/edge collisions occur (on contact with the staircase), and sphere/sphere collisions (on contact between different parts of the articulated body, e.g. the hand against the chest). I found the use of spheres to be the most reliable technique for such a complicated mesh. Constraint functions for these types of collision can be derived fairly easily and handled using the usual algorithms for resting and colliding contact as described in section 3.3.

The camera movement and lighting was set up in Blender. After importing the simulation results, the animation was rendered using Blender's internal renderer/raytracer. Some of the mesh deformations look strange; this is unrelated to the simulation, but must be blamed on my lack of 3D art skills!

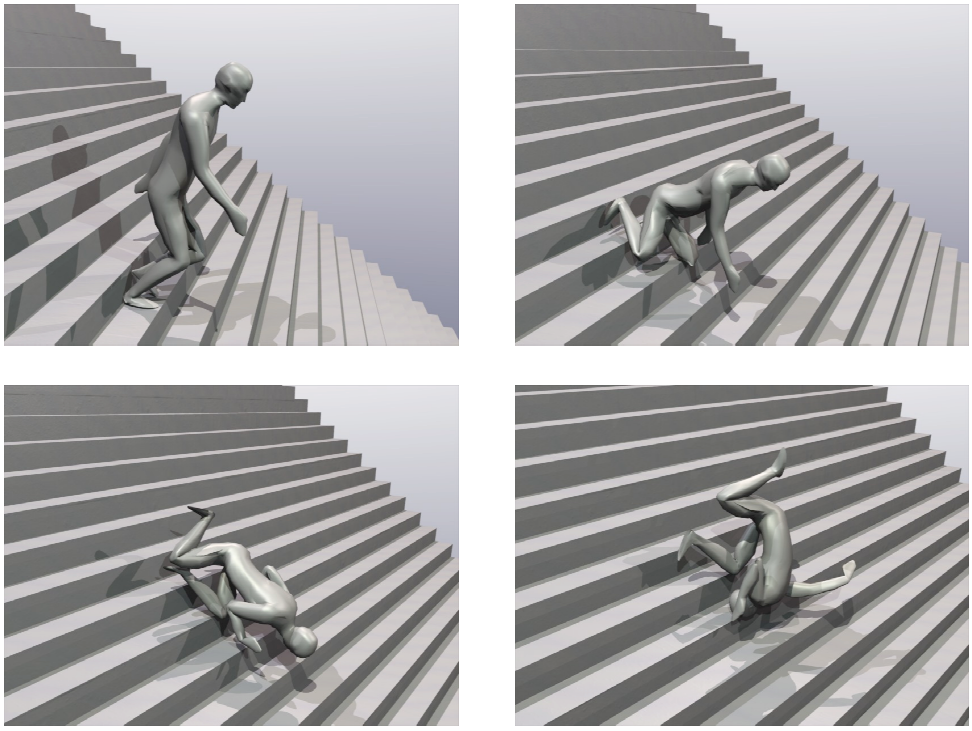

Figure 4.7: Animation of Alfred falling down a set of stairs.

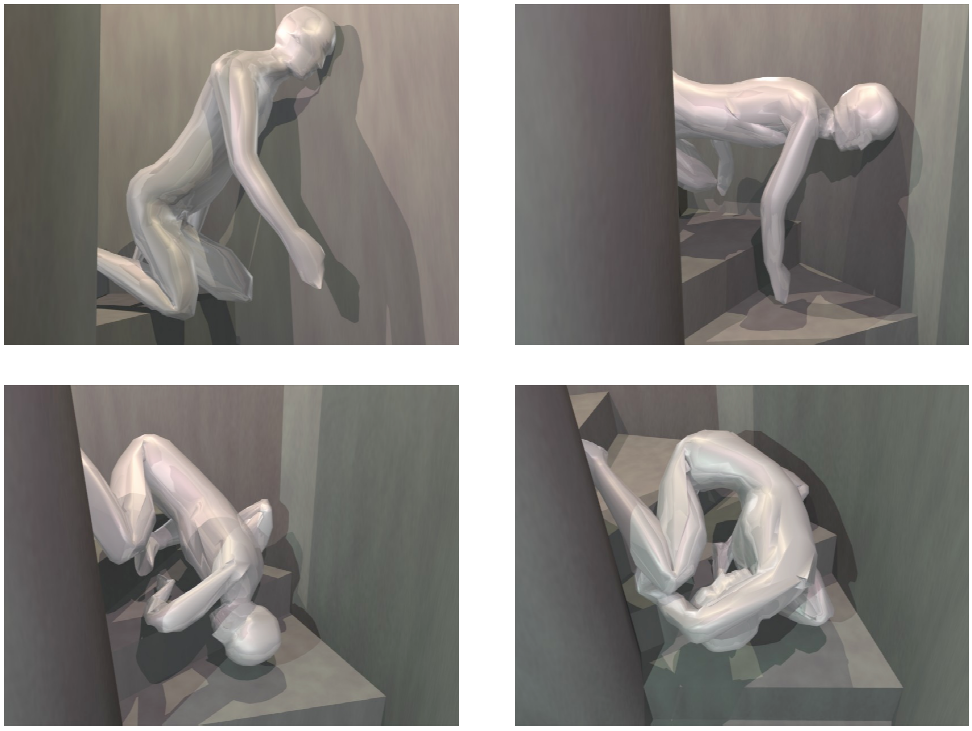

Figure 4.8: Animation of Alfred tumbling down a spiral staircase.

Chapter 5

Conclusions

5.1 Successes

In this project I have successfully created a general-purpose application which simulates the dynamic behaviour of many different mechanical systems. In particular, it handles all features necessary to animate articulated characters like humans or animals: an arbitrary-shaped exterior, a skeleton with many different types of joints, and collisions with other bodies. I have quantitatively verified the accuracy of the simulation results for simple mechanical systems, and demonstrated that the program produces realistic-looking animations in more complicated scenes involving a model of a human character.

All requirements of the project proposal have been met, except for the simulation of friction, which was ruled out at an early stage because it is too complicated. All other features were realized at a high quality standard. The implementation is carefully designed to use refined algorithms which

- keep numerical errors very small, doubtlessly within the bounds acceptable even for ambitious computer graphics purposes;
- give the simulation a high degree of stability and reliability;
- allow a reasonable run-time performance which can be traded off against the required accuracy by adjusting parameters.

The technology of this project is close to current research, and carrying it out required some algorithms which I could not find in any of the publications I searched. I therefore had to newly invent several algorithms, which took several weeks longer than anticipated in the original schedule. However, I succeeded in adapting my management strategy to the uncertainties facing this project, so that it could be completed in an organized way.

5.2 Limitations

There are only few explicit approximations in my simulation algorithms; this means that if they were to be run using ideal arithmetic (no rounding errors) and arbitrarily small time steps, the exact physical behaviour would be obtained. The exceptions are:

- Given a triangle mesh and the density of a body, I currently approximate its mass and moment of inertia using a simple heuristic. For a better simulation, this should be replaced by an exact analysis of the geometric properties [14].

- Simple collisions (vertex/face and edge/edge cases) can be handled in an exact way, but more complicated collision geometries are currently approximated by planes or bounding spheres. This produces plausible-looking but slightly incorrect animations. Ideally, exact handling of any sort of collision would be desirable, but I am unsure whether such an algorithm exists.

- As mentioned previously, static and sliding friction have been completely ignored. Algorithms to simulate these exist, but they are complicated [2].

Even with these added features, my program would not be a ready-to-market product. It particularly lacks user-friendliness: there are several simulation parameters (tolerances and error thresholds) which need to be configured differently depending on the situation in order to obtain good results. Ideally the program should be able to determine these automatically. A higher execution speed would also be beneficial, since the current cost – in my test cases, up to an hour of CPU time per second of simulation time – limits productivity. The algorithm in [3] could alleviate this.

5.3 Summary and outlook

In summary, I consider this project a clear success. I am glad to have chosen such an ambitious project because it was personally rewarding: in particular, the lack of existing algorithms forced me to contemplate the background theory in great detail, which has given me a good understanding of the subject area.

There are not many decisions that I would have taken differently in retrospect. In the schedule I should have left more time for understanding the theory; I had thought there would be a simple, "obvious" way of realizing articulated bodies, rather than having to deal with constraints. This meant I massively underestimated the problem. I would have also avoided the design mistake mentioned in section 3.1.3.

The application is not yet user-friendly enough for general use, but with some additional effort it might gain this potential. I can envisage working on it beyond the frame of this project, and will consider the possibility of publishing some of the new algorithms to a wider audience.

I would like to thank Dr. Brett Saunders [21] and Dr. Neil Dodgson (supervisor) for their ideas and discussions, particularly in the early phase of this project.

Appendix A

Notation

a	italic Latin letter	scalar
θ	italic Greek letter	angle
$\mathbf{a}$	boldface Greek/Latin letter	column vector
$\mathbf{0}$	boldface figure 0	null vector (of appropriate dimension)
M	sans serif upper-case letter	matrix
$\mathsf{1}$	sans serif figure 1	identity matrix (of appropriate dim.)
q	sans serif lower-case letter	quaternion
$\dot{c}$	dot	first derivative with respect to time
$\ddot{c}$	double dot	second derivative with respect to time
J^T	superscript T	matrix transpose
M^{-1}	superscript -1	matrix or quaternion inverse (eq. 2.5)
$\overline{\mathsf{q}}$	bar	quaternion conjugate (eq. 2.4)
$\lvert\mathbf{a}\rvert$	norm	vector or quaternion magnitude (eq. 2.6)
$\tilde{\mathbf{a}}$	tilde	corresponding quaternion (eq. 2.7)
$\mathbf{a}^*$	asterisk	dual tensor (eq. A.1)
$\mathbf{a} \times \mathbf{b}$	cross	vector cross product
$\mathbf{a} \cdot \mathbf{b}$	dot	inner product
$\Re(\mathsf{q})$	real	real part of a quaternion

Given any vector $\mathbf{a} = (a_1, a_2, a_3)^T$, we define its dual tensor, written as a 3×3 matrix, to be

$$\mathbf{a}^* = \begin{bmatrix} 0 & -a_3 & a_2 \\ a_3 & 0 & -a_1 \\ -a_2 & a_1 & 0 \end{bmatrix} \tag{A.1}$$

(see [18, 4] and also Kalra [13], who defines it to be the transpose of the expression above). The dual allows us to rewrite a vector cross product as a matrix multiplication:

$$\mathbf{a} \times \mathbf{b} = \mathbf{a}^* \, \mathbf{b} \tag{A.2}$$

Note that $(\mathbf{a}^*)^T = -\mathbf{a}^*$.

Let us also recall some basic identities of vector and matrix algebra [18]:

$$\begin{aligned} \mathbf{a} \times \mathbf{a} &= \mathbf{0} \quad \text{(the null vector)} \\ \mathbf{a} \times \mathbf{b} &= -\mathbf{b} \times \mathbf{a} \\ \mathbf{a} \times (\mathbf{b} + \mathbf{c}) &= \mathbf{a} \times \mathbf{b} + \mathbf{a} \times \mathbf{c} \end{aligned}$$

$$
\begin{aligned}
\mathbf{a} \cdot \mathbf{b} &= \mathbf{b} \cdot \mathbf{a} \\
\mathbf{a} \cdot \mathbf{b} &= \mathbf{a}^T \mathbf{b} \\
\mathbf{a} \cdot (\mathbf{b} + \mathbf{c}) &= \mathbf{a} \cdot \mathbf{b} + \mathbf{a} \cdot \mathbf{c} \\
\mathbf{a} \cdot (\mathbf{b} \times \mathbf{c}) &= \mathbf{b} \cdot (\mathbf{c} \times \mathbf{a}) \\
&= \mathbf{c} \cdot (\mathbf{a} \times \mathbf{b}) \\
\mathbf{a} \times (\mathbf{b} \times \mathbf{c}) &= \mathbf{b}(\mathbf{a} \cdot \mathbf{c}) - \mathbf{c}(\mathbf{a} \cdot \mathbf{b}) \\
(\mathsf{AB})\mathsf{C} &= \mathsf{A}(\mathsf{BC}) \\
\mathsf{A}(\mathsf{B} + \mathsf{C}) &= \mathsf{AB} + \mathsf{AC} \\
(\mathsf{AB})^T &= \mathsf{B}^T \mathsf{A}^T \\
\mathsf{A}\mathsf{A}^{-1} = \mathsf{A}^{-1}\mathsf{A} &= \mathsf{1} \quad \text{(the identity matrix)}
\end{aligned}
$$

Appendix B

Proofs and derivations

B.1 Quaternion integration

Let us first consider a geometric view on quaternions, taken from [24]. Treat the four components of a quaternion as Cartesian coordinates of a four-dimensional vector space. The set of unit quaternions is then the surface of a unit hypersphere (also called a *glome* [26]) in this vector space. Each point on this hypersphere corresponds to a particular rotation. It also turns out that each pair of opposite points on this sphere represent exactly the same rotation; hence all possible rotations are contained in one hemisphere, no matter where the sphere is cut in half.

B.1.1 Conservation of magnitude

Define the quaternion dot product, in analogy to the 4D vector dot product, to be

$$\mathsf{p} \cdot \mathsf{q} = (p_w + p_x\mathsf{i} + p_y\mathsf{j} + p_z\mathsf{k}) \cdot (q_w + q_x\mathsf{i} + q_y\mathsf{j} + q_z\mathsf{k}) = p_w q_w + p_x q_x + p_y q_y + p_z q_z \tag{B.1}$$

The dot product is commutative, contrary to the quaternion juxtaposition product.

The instantaneous rate of change is given [4, 6, 20] to be

$$\begin{aligned}
\dot{\mathsf{q}} &= \frac{1}{2}\tilde{\boldsymbol{\omega}}\mathsf{q} = \frac{1}{2}(\omega_1\mathsf{i} + \omega_2\mathsf{j} + \omega_3\mathsf{k})(q_w + q_x\mathsf{i} + q_y\mathsf{j} + q_z\mathsf{k}) \\
&= \frac{1}{2}(-\omega_1 q_x - \omega_2 q_y - \omega_3 q_z) + \frac{\mathsf{i}}{2}(\omega_1 q_w + \omega_2 q_z - \omega_3 q_y) + \\
&\quad \frac{\mathsf{j}}{2}(-\omega_1 q_z + \omega_2 q_w + \omega_3 q_x) + \frac{\mathsf{k}}{2}(\omega_1 q_y - \omega_2 q_x + \omega_3 q_w)
\end{aligned}$$

We now treat q and $\dot{\mathsf{q}}$ as 4D vectors and calculate their dot product:

$$\begin{aligned}
\mathsf{q} \cdot \dot{\mathsf{q}} &= \frac{1}{2}(-q_w\omega_1 q_x - q_w\omega_2 q_y - q_w\omega_3 q_z + q_x\omega_1 q_w + q_x\omega_2 q_z - q_x\omega_3 q_y \\
&\quad -q_y\omega_1 q_z + q_y\omega_2 q_w + q_y\omega_3 q_x + q_z\omega_1 q_y - q_z\omega_2 q_x + q_z\omega_3 q_w) \\
&= 0.
\end{aligned}$$

The rate of change is orthogonal to q, and therefore it is always a tangent to the sphere, touching it at the point corresponding to q. The set of all possible values of $\dot{\mathsf{q}}$ is thus a hyperplane (a three-dimensional subspace) tangential to the sphere at the point q in 4D space.

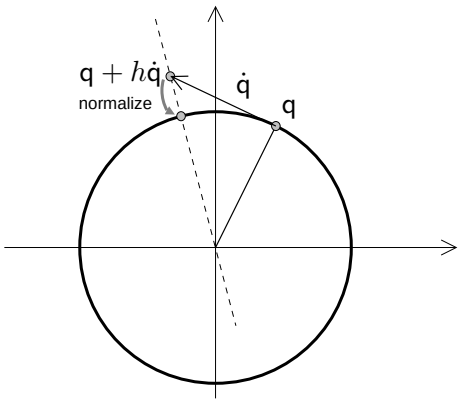

Figure B.1: Normalizing a quaternion after performing an ODE solving step.

We can determine the magnitude $|\dot{\mathsf{q}}|$ from the sum of squares of the components given above and find it to be $|\dot{\mathsf{q}}| = \frac{1}{2}|\boldsymbol{\omega}|\,|\mathsf{q}|$. Since we always require q to be a unit quaternion, we can reduce this to

$$|\dot{\mathsf{q}}| = \frac{1}{2}|\boldsymbol{\omega}|. \tag{B.2}$$

Now let us determine what happens if we calculate $\mathsf{q} + h\dot{\mathsf{q}}$ for some finite h. Note that this operation is required by all common numerical solvers of differential equations. Consider the magnitude of the result:

$$\begin{aligned} |\mathsf{q} + h\dot{\mathsf{q}}|^2 &= (\mathsf{q} + h\dot{\mathsf{q}}) \cdot (\mathsf{q} + h\dot{\mathsf{q}}) \\ &= \mathsf{q} \cdot \mathsf{q} + 2h\mathsf{q} \cdot \dot{\mathsf{q}} + h^2\dot{\mathsf{q}} \cdot \dot{\mathsf{q}} \\ &= 1 + 0 + \frac{h^2}{4}|\boldsymbol{\omega}|^2 \\ &> 1 \qquad \text{whenever} \quad |\boldsymbol{\omega}| > 0. \end{aligned}$$

Hence, if the body in question is rotating, it is not possible for a standard numerical ODE solver to preserve a quaternion's property of unit magnitude.

B.1.2 Normalization is not enough

One should think that given the derivative of a quaternion q (equation 2.11, page 13), one can find $\mathsf{q}(t+h)$ for some time step h within the accuracy of the ODE solver employed ($O(h^5)$ error for fourth-order Runge-Kutta). Unfortunately this is not the case. This shall be demonstrated using Euler's method; it should, however, be pointed out that more sophisticated methods like RK4 are also affected. Consider the value of q at the next time step, $\mathsf{q}(t+h) = \mathsf{q}(t) + h\dot{\mathsf{q}}(t)$. For any non-zero h and $\dot{\mathsf{q}}$ this point will always lie outside the unit quaternion sphere due to the orthogonality of q and $\dot{\mathsf{q}}$. This is usually compensated by renormalizing the quaternion after the ODE solving step. Geometrically, this renormalization can be understood as drawing a straight line through the origin and the point $\mathsf{q}(t) + h\dot{\mathsf{q}}(t)$, intersecting this line with the unit sphere and replacing $\mathsf{q}(t+h)$ by this point of intersection (see figure B.1).

Following the tangent to the sphere is a reasonable approximation to following its curve if the magnitude of $h\dot{\mathsf{q}}(t)$ is small compared to the curvature of the sphere. For large time steps or large magnitudes of $\boldsymbol{\omega}$, however, this gets increasingly erroneous. Consider the limiting case, a body rotating infinitely fast ($|\boldsymbol{\omega}| \to \infty$): after renormalisation, q will have moved merely

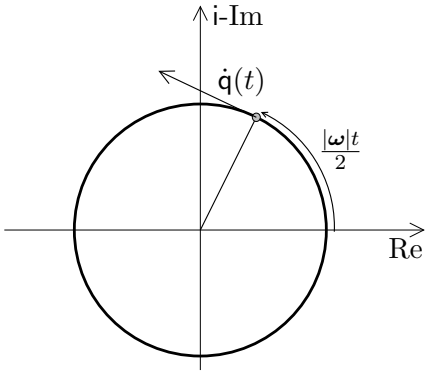

Figure B.2: Assumed situation for the derivation in section B.1.3.

a quarter of the way around the unit sphere, which equates to concatenating the rotation of quaternion q with some rotation by 180°. This is a strictly finite amount of rotation per time step, while it would actually have been correct to perform an infinite number of revolutions around the quaternion sphere.

If a polynomial approximation method like RK4 had been used instead of Euler's method, a parametric polynomial space curve would have been fitted to the surface of the sphere instead of the straight line. Note however that the Taylor series of the sin and cos functions are non-terminating, and that it is therefore not possible for a finite polynomial curve to lie exactly in the surface of a sphere. These ODE solvers will therefore suffer the same problems, albeit less pronounced.

B.1.3 Corrected quaternion integration

Assume that the body we are simulating is rotating at a constant angular velocity. (This assumption is later weakened by the use of a more sophisticated ODE solver, but for now we will stick with Euler's method.) Furthermore assume without loss of generality that the body is rotating clockwise about its x axis, which corresponds to the world's x axis, and that at time $t = 0$ the body's frame and the world frame coincide. Then the orientation of the body (the quaternion describing the linear transformation from the body's frame of reference to the world frame) is given as a function of time by

$$\mathsf{q}(t) = \cos\left(\frac{|\boldsymbol{\omega}|t}{2}\right) + \sin\left(\frac{|\boldsymbol{\omega}|t}{2}\right)\mathsf{i} \tag{B.3}$$

(cf. equation 2.8, figure B.2) and its angular velocity is

$$\boldsymbol{\omega} = (\omega_1, \omega_2, \omega_3)^T = (|\boldsymbol{\omega}|, 0, 0)^T \tag{B.4}$$

for some arbitrary $|\boldsymbol{\omega}|$, measured in radians per unit time.

Now assume w.l.o.g. that we take a time step from $t = 0$ to $t = h$. Then we require that the result returned by Euler's method for $\mathsf{q}(h)$ after renormalization be equal to its exact value in equation B.3:

$$\cos\left(\frac{|\boldsymbol{\omega}|h}{2}\right) + \sin\left(\frac{|\boldsymbol{\omega}|h}{2}\right)\mathsf{i} = \frac{\mathsf{q}(0) + h\dot{\mathsf{q}}(0)}{|\mathsf{q}(0) + h\dot{\mathsf{q}}(0)|} \tag{B.5}$$

We know from examining the 4D geometry that the value assigned to $\dot{\mathsf{q}}$ in equation 2.11 has the correct direction and merely needs to be corrected in magnitude. In other words, we are

searching for a scalar function $f(h, |\boldsymbol{\omega}|)$ which will allow $\dot{\mathsf{q}}$ to satisfy equation B.5:

$$\dot{\mathsf{q}}_h(t) = f\tilde{\boldsymbol{\omega}}(t)\mathsf{q}(t) \tag{B.6}$$

Observe that under the above assumptions $\mathsf{q}(0) = 1$, and thus $\dot{\mathsf{q}}_h(0) = f|\boldsymbol{\omega}|\mathsf{i}$. Substituting this into equation B.5 and considering only the real part:

$$\begin{aligned}
& \cos\left(\frac{|\boldsymbol{\omega}|h}{2}\right) = \left[1 + (f|\boldsymbol{\omega}|h)^2\right]^{-\frac{1}{2}} \\
\Leftrightarrow \quad & (f|\boldsymbol{\omega}|h)^2 = \frac{1}{\cos^2\left(\frac{|\boldsymbol{\omega}|h}{2}\right)} - 1 \\
\Leftrightarrow \quad & f(h, |\boldsymbol{\omega}|) = \frac{1}{|\boldsymbol{\omega}|h}\sqrt{\frac{1 - \cos^2\left(\frac{|\boldsymbol{\omega}|h}{2}\right)}{\cos^2\left(\frac{|\boldsymbol{\omega}|h}{2}\right)}} = \frac{1}{|\boldsymbol{\omega}|h}\tan\left(\frac{|\boldsymbol{\omega}|h}{2}\right)
\end{aligned}$$

To check, we substitute this result into the i-imaginary part of equation B.5:

$$\begin{aligned}
\sin\left(\frac{|\boldsymbol{\omega}|h}{2}\right) &= \tan\left(\frac{|\boldsymbol{\omega}|h}{2}\right)\left[1 + \tan^2\left(\frac{|\boldsymbol{\omega}|h}{2}\right)\right]^{-\frac{1}{2}} \\
&= \tan\left(\frac{|\boldsymbol{\omega}|h}{2}\right)\left[\frac{\cos^2\left(\frac{|\boldsymbol{\omega}|h}{2}\right) + \sin^2\left(\frac{|\boldsymbol{\omega}|h}{2}\right)}{\cos^2\left(\frac{|\boldsymbol{\omega}|h}{2}\right)}\right]^{-\frac{1}{2}} \\
&= \tan\left(\frac{|\boldsymbol{\omega}|h}{2}\right)\cos\left(\frac{|\boldsymbol{\omega}|h}{2}\right) \\
&= \sin\left(\frac{|\boldsymbol{\omega}|h}{2}\right)
\end{aligned}$$

Thus we establish the validity of this expression for f. Observe that by using L'Hospital's rule, we can find value of f for an infinitesimally small time step:

$$\lim_{h\to 0} f = \lim_{h\to 0} \frac{\frac{|\boldsymbol{\omega}|}{2}\cos^{-2}\left(\frac{|\boldsymbol{\omega}|h}{2}\right)}{|\boldsymbol{\omega}|} = \frac{1}{2}$$

i.e. we obtain the original equation 2.11 for the instantaneous rate of change.

Now let $\Delta\mathsf{q} = h\dot{\mathsf{q}} = \frac{h}{2}\tilde{\boldsymbol{\omega}}\mathsf{q}$. From equation B.2 we find that $|\Delta\mathsf{q}| = \frac{|\boldsymbol{\omega}|h}{2}$. Hence we can simplify the expression for the quaternion correcting factor by expressing it in terms of $\Delta\mathsf{q}$ as follows:

$$hf\tilde{\boldsymbol{\omega}}\mathsf{q} = \frac{h}{|\boldsymbol{\omega}|h}\tan\left(\frac{|\boldsymbol{\omega}|h}{2}\right)\tilde{\boldsymbol{\omega}}\mathsf{q} = \tan\left(|\Delta\mathsf{q}|\right)\frac{\Delta\mathsf{q}}{|\Delta\mathsf{q}|}$$

This expression now has a clear geometric interpretation with respect to the 4D geometry (see figure B.3): $|\Delta\mathsf{q}|$ is measured in radians, and it corresponds to the *correct* angle between the old and the new vector q. Since q and $\dot{\mathsf{q}}$ are orthogonal, we have a right-angled triangle between the origin, the old and the new points q, and hence we can use the tan function to evaluate the required length of the side in direction $\Delta\mathsf{q}$.

Finally we can combine this correction and the subsequent quaternion normalisation into a single function, which I call Quergs (for *Qu*at*er*nion inte*g*ration *s*tep)[1]:

$$\mathsf{q}(t+h) = \text{Quergs}(\mathsf{q}(t), \Delta\mathsf{q}) \quad = \quad \frac{\mathsf{q}(t) + \tan\left(|\Delta\mathsf{q}|\right)\frac{\Delta\mathsf{q}}{|\Delta\mathsf{q}|}}{|\mathsf{q}(t) + \tan\left(|\Delta\mathsf{q}|\right)\frac{\Delta\mathsf{q}}{|\Delta\mathsf{q}|}|}$$

[1]This naming follows the spirit of Shoemake [24], whose "Slerp" function is an 'acronym' of *S*pherical *l*inear int*erp*olation.

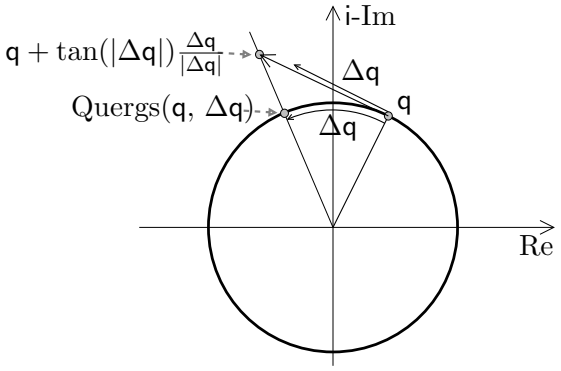

Figure B.3: Illustration of the operation of Quergs.

$$\begin{aligned} &= \frac{\mathsf{q}(t) + \tan(|\Delta\mathsf{q}|)\,\frac{\Delta\mathsf{q}}{|\Delta\mathsf{q}|}}{\sqrt{1+\tan^2(|\Delta\mathsf{q}|)}} \\ &= \left[\mathsf{q}(t) + \tan(|\Delta\mathsf{q}|)\,\frac{\Delta\mathsf{q}}{|\Delta\mathsf{q}|}\right]\cos(|\Delta\mathsf{q}|) \end{aligned}$$

The last expression is simplest (and again allows geometric interpretation), but probably the first of the three expressions is more useful for numerical evaluation, since it involves only one trigonometric function and minimizes numerical errors.

When implementing this formula, care must be taken around the discontinuities of the tan function, where numerical instability may occur. These discontinuities are reached whenever a body performs an odd multiple of half revolutions during a single time step.

B.2 Free precession

This argument is modelled after [19]. The moment of inertia $\mathbf{L}$ of a rigid body is defined as

$$\mathbf{L} = \mathsf{I}\boldsymbol{\omega} \tag{B.7}$$

where I is the inertia tensor and $\boldsymbol{\omega}$ is the angular velocity vector. Torque is the rate of change of angular momentum over time:

$$\boldsymbol{\tau} = \dot{\mathbf{L}} = \dot{\mathsf{I}}\boldsymbol{\omega} + \mathsf{I}\dot{\boldsymbol{\omega}} \tag{B.8}$$

We can further evaluate $\dot{\mathsf{I}}$ by writing I as a product with some rotation matrix R and its transpose:

$$\mathsf{I} = \mathsf{R}\mathsf{I}_{\text{body}}\mathsf{R}^T \tag{B.9}$$

It can be shown that such a decomposition of the inertia tensor always exists, and that I_{body} is a diagonal, time-invariant matrix containing the moments of inertia about the body's principal axes [8]. Hence we obtain

$$\dot{\mathsf{I}} = \dot{\mathsf{R}}\mathsf{I}_{\text{body}}\mathsf{R}^T + \mathsf{R}\mathsf{I}_{\text{body}}\frac{\mathrm{d}}{\mathrm{d}t}\mathsf{R}^T \tag{B.10}$$

Witkin [4] derives that $\dot{\mathsf{R}} = \boldsymbol{\omega}^*\,\mathsf{R}$ for a rotation matrix R and an angular velocity vector $\boldsymbol{\omega}$. Using this identity and taking the differential operator onto the inside of the transpose at the

end of equation B.10,

$$\begin{aligned} \dot{\mathsf{I}} &= \boldsymbol{\omega}^* \mathsf{R}\mathsf{I}_{\text{body}}\mathsf{R}^T + \mathsf{R}\mathsf{I}_{\text{body}}(\boldsymbol{\omega}^* \mathsf{R})^T \\ &= \boldsymbol{\omega}^* \mathsf{R}\mathsf{I}_{\text{body}}\mathsf{R}^T - \mathsf{R}\mathsf{I}_{\text{body}}\mathsf{R}^T\boldsymbol{\omega}^* \\ &= \boldsymbol{\omega}^* \mathsf{I} - \mathsf{I}\boldsymbol{\omega}^* \end{aligned} \tag{B.11}$$

Substituting equation B.11 back into B.8:

$$\begin{aligned} \boldsymbol{\tau} &= \mathsf{I}\dot{\boldsymbol{\omega}} + \boldsymbol{\omega}^* \mathsf{I}\boldsymbol{\omega} - \mathsf{I}\boldsymbol{\omega}^* \boldsymbol{\omega} \\ &= \mathsf{I}\dot{\boldsymbol{\omega}} + \boldsymbol{\omega}^* \mathsf{I}\boldsymbol{\omega} \end{aligned} \tag{B.12}$$

Equation B.12 corrects the similar expression in [20], page 34. This means that the angular acceleration of a rigid body is given by

$$\dot{\boldsymbol{\omega}} = \mathsf{I}^{-1}(\boldsymbol{\tau} - \boldsymbol{\omega}^* \mathsf{I}\boldsymbol{\omega}). \tag{B.13}$$

where $\boldsymbol{\tau}$ is the sum of all torque vectors acting on the body. This means that even if there are no torques, its angular velocity may change if I is not diagonal (i.e. if the body is somehow asymmetric). This effect is called *free precession*.

In a simulation, we usually integrate over torques to find angular momentum, and then calculate the angular velocity from the momentum in each time step using the current moment of inertia. In this case, the angular acceleration in equation B.13 is not needed. It is required only for purposes of computing constraint forces and torques.

B.3 Constrained rigid body dynamics

This section is based on material in [4] and [20]. The explanation in these sources is rather sketchy though, and the following details I worked out are more comprehensive.

B.3.1 Prerequisites of constraint solving

Consider a system of n rigid bodies at a particular point in time. Assume that we can define a vector $\mathbf{x}$ which represents the state of the whole system as a function of time[2]. $\mathbf{x}$ has $6n$ rows (one row for each d.o.f.) and its first and second derivative with respect to time are

$$\dot{\mathbf{x}} = \begin{bmatrix} \dot{\mathbf{r}}_1 \\ \boldsymbol{\omega}_1 \\ \vdots \\ \dot{\mathbf{r}}_n \\ \boldsymbol{\omega}_n \end{bmatrix} \quad \text{and} \quad \ddot{\mathbf{x}} = \begin{bmatrix} \ddot{\mathbf{r}}_1 \\ \dot{\boldsymbol{\omega}}_1 \\ \vdots \\ \ddot{\mathbf{r}}_n \\ \dot{\boldsymbol{\omega}}_n \end{bmatrix} \tag{B.14}$$

where $\mathbf{r}_i$ is the position of the centre of mass of body i, and $\boldsymbol{\omega}_i$ is its angular velocity. Thus $\dot{\mathbf{x}}$ is the concatenation of all bodies' linear and angular velocities, and $\ddot{\mathbf{x}}$ is that of the accelerations. We cannot write down an explicit expression for $\mathbf{x}$ since we do not have a representation of orientation whose derivative is $\boldsymbol{\omega}$. This is not a problem though, since we can work exclusively with $\dot{\mathbf{x}}$ and $\ddot{\mathbf{x}}$.

[2] Witkin [4] calls this vector $\mathbf{q}$; I use $\mathbf{x}$ to avoid confusion with quaternions.

We require two more vectors in the same format, containing the forces and torques already acting on the system, for example due to gravity or muscular activity:

$$\mathbf{\Phi} = \begin{bmatrix} \mathbf{F}_1 \\ \boldsymbol{\tau}_1 \\ \vdots \\ \mathbf{F}_n \\ \boldsymbol{\tau}_n \end{bmatrix} \qquad \mathbf{\Phi}_p = \begin{bmatrix} \mathbf{0} \\ -\boldsymbol{\omega}_1^* \mathsf{I}_1 \boldsymbol{\omega}_1 \\ \vdots \\ \mathbf{0} \\ -\boldsymbol{\omega}_n^* \mathsf{I}_n \boldsymbol{\omega}_n \end{bmatrix} \tag{B.15}$$

Here $\mathbf{F}_i$ denotes the sum of all forces acting on the centre of mass of body i, and $\boldsymbol{\tau}_i$ is the sum of all torque vectors acting on body i. The vector $\mathbf{\Phi}_p$ accounts for the fact that a body's moment of inertia is in general not constant in the world frame[3]; I_i is body i's moment of inertia in the world frame at the current point in time. See appendix B.2 for a derivation of $\mathbf{\Phi}_p$.

Following the same principle of concatenation, we set up the inverse mass-inertia matrix M^{-1}, a $6n \times 6n$ matrix of the form

$$\mathsf{M}^{-1} = \begin{bmatrix} \frac{1}{\mu_1}\mathsf{1} & & & & & & \\ & \mathsf{I}_1^{-1} & & & & & \\ & & \frac{1}{\mu_2}\mathsf{1} & & & & \\ & & & \mathsf{I}_2^{-1} & & & \\ & & & & \ddots & & \\ & & & & & \frac{1}{\mu_n}\mathsf{1} & \\ & & & & & & \mathsf{I}_n^{-1} \end{bmatrix} \tag{B.16}$$

μ_i denotes the (scalar) mass of body i, $\mathsf{1}$ stands for the 3×3 identity matrix, and I_i^{-1} is the inverse of the inertia tensor (written as a 3×3 matrix) of body i in the world frame. All empty regions are zero. While μ_i will typically stay constant over time, I_i may depend on the orientation of body i.[4] These variables have been chosen such that

$$\ddot{\mathbf{x}} = \mathsf{M}^{-1}(\mathbf{\Phi} + \mathbf{\Phi}_p). \tag{B.17}$$

B.3.2 Definition of the Jacobians

Now assume that we want to impose m constraints on this system. Express each constraint as a function c which is zero when the constraint is satisfied. We can then concatenate all constraint functions into a single m-row constraint vector $\mathbf{c}$. Each valid configuration of the system must satisfy $\mathbf{c} = \mathbf{0}$. $\dot{\mathbf{c}}$ and $\ddot{\mathbf{c}}$ must be calculated algebraically before implementation.

The Jacobian matrix [18] J contains all partial derivatives of $\mathbf{c}$ w.r.t. each coordinate of $\mathbf{x}$. Similarly, $\dot{\mathsf{J}}$ is the Jacobian of $\dot{\mathbf{c}}$:

$$J_{ij} = \frac{\partial c_i}{\partial x_j} \qquad \text{and} \qquad \dot{J}_{ij} = \frac{\partial \dot{c}_i}{\partial x_j} \tag{B.18}$$

Both J and $\dot{\mathsf{J}}$ are $m \times 6n$ matrices. However, since we do not have an explicit expression for $\mathbf{x}$, we cannot calculate them directly using partial differentiation. Instead we make use of the following analytic results:

$$\dot{\mathbf{c}} = \mathsf{J}\dot{\mathbf{x}} \qquad \text{and} \qquad \ddot{\mathbf{c}} = \dot{\mathsf{J}}\dot{\mathbf{x}} + \mathsf{J}\ddot{\mathbf{x}}. \tag{B.19}$$

[3] The subscript p of this vector stands for *precession*, since the variability of the moment of inertia causes free precession.

[4] If we know the principal axes of the body, we can express I in a time-invariant form combined with rotations into the principal axes frame and back again (see [4]), which saves us a lot of effort.

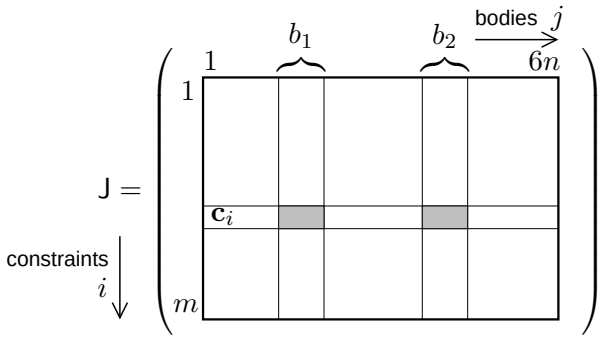

Figure B.4: Block structure of the Jacobian matrix of m constraints in a system of n bodies. For example, if $\mathbf{c}_i$ is a ball-and-socket joint between bodies b_1 and b_2, each of the two slices (shaded areas) is three rows high and six columns wide. The vertical position of $\mathbf{c}_i$ must match its offset in the concatenated constraint vector $\mathbf{c}$, and the horizontal positions of the slices must match the offset of bodies b_1 and b_2 in the vectors $\dot{\mathbf{x}}$, $\mathbf{\Phi}$ and $\mathbf{\Phi}_p$ (equations B.14 and B.15).

Any Jacobian component J_{ij} is certainly zero if constraint i does not mention body $\lfloor j/6 \rfloor$. Since each constraints usually involves only two bodies, J and $\dot{\mathsf{J}}$ each have a sparse block-structured form sketched in figure B.4. If there are many constraints, their slices of the Jacobian can be calculated independently and simply inserted at the right places in J and $\dot{\mathsf{J}}$. Appendix B.4 gives derivations of Jacobian slices for different constraint types.

B.3.3 Finding the Jacobians

Both [4] and [20] omit details on how to derive the Jacobians J and $\dot{\mathsf{J}}$ for a custom constraint. I deduced the following procedure after pondering over some source code implementing one type of constraint (specifically equations B.23 and B.24 below). The source code was kindly provided by Dr. Breton Saunders [21]. It was used only for this derivation and not copied otherwise. The implementation in this project is based on the following derivations.

We start by defining a function $\mathbf{c}$ which evaluates to zero (or the null vector) for all states which satisfy the constraint, and any non-zero value for all other states. We then calculate the first and second derivatives with respect to time, $\dot{\mathbf{c}}$ and $\ddot{\mathbf{c}}$, which must both exist.

For any sort of valid constraint, we should be able to massage both $\dot{\mathbf{c}}$ and $\ddot{\mathbf{c}}$ into a sum of products form. Moreover, in each summand in $\dot{\mathbf{c}}$, at least one of the variables should be either the linear velocity of the centre of mass of one of the bodies, or the angular velocity of one of the bodies. Use algebra to make this 'chosen variable' the rightmost variable in each product, and evaluate the rest of the product to a single matrix.

Now observe equation B.19 (page 54). We can easily factor our expression for $\dot{\mathbf{c}}$ into J and $\dot{\mathbf{x}}$ (since the latter contains the linear and angular velocities for all bodies). J has the same number of rows as there are constraints, and $6n$ columns for a system of n bodies. Each of the matrices in our expression for $\dot{\mathbf{c}}$ also has the same number of rows as there are constraints, and has 3 columns. All we need to do is to find the correct horizontal position of each of these matrices in J, depending on the location of the 'chosen variable' in $\dot{\mathbf{x}}$. Thus we obtain the matrix J.

Observe equation B.19 again. Now we know $\ddot{\mathbf{c}}$, $\dot{\mathbf{x}}$, J and $\ddot{\mathbf{x}}$ – no problem. Evaluate $\ddot{\mathbf{c}} - \mathsf{J}\ddot{\mathbf{x}}$, remembering that $\ddot{\mathbf{x}}$ is the vector of linear accelerations and angular accelerations. The result should be an expression which we can bring into just the same kind of sum of products form as we previously did with $\dot{\mathbf{c}}$. By exactly the same procedure we factor out the linear velocities and

angular velocities (not accelerations!), thus obtaining $\dot{\mathsf{J}}$.

In case you were in doubt, this procedure is of course executed by a human on paper (possibly assisted by a symbolic algebra system) prior to implementation. The simulation program will just plug numbers into the hard-coded expressions for $\mathbf{c}$, $\dot{\mathbf{c}}$, J and $\dot{\mathsf{J}}$ during each time step of the simulation.

B.4 A catalogue of constraint functions

Let us consider some examples of constraints to clarify the procedure.

B.4.1 Fixed point in space ('Nail')

This simple constraint 'nails' a particular point in a rigid body to a fixed point in world space. (It's a very flexible nail, because despite fixing the position, it allows all three modes of rotation.) Let $\mathbf{p}$ be the position of the centre of mass of our rigid body, $\mathbf{s}$ the vector from the centre of mass to the point in the body we want to attach, $\boldsymbol{\omega}$ the angular velocity of the rigid body, and $\mathbf{t}$ the coordinates in world space that we want to nail the point to. Then we can set up a simple constraint function,

$$\mathbf{c} = \mathbf{p} + \mathbf{s} - \mathbf{t} \tag{B.20}$$

which equals the null vector when $\mathbf{p} + \mathbf{s}$ and $\mathbf{t}$ coincide, as required. Since this is a three-dimensional vector equation, we are actually defining three constraints at once. $\mathbf{t}$ does not change over time, so we obtain

$$\begin{aligned} \dot{\mathbf{c}} &= \dot{\mathbf{p}} + \boldsymbol{\omega} \times \mathbf{s} \\ &= \dot{\mathbf{p}} - \mathbf{s}^* \boldsymbol{\omega} \qquad \text{(B.21)} \\ \ddot{\mathbf{c}} &= \ddot{\mathbf{p}} + \dot{\boldsymbol{\omega}} \times \mathbf{s} + \boldsymbol{\omega} \times (\boldsymbol{\omega} \times \mathbf{s}) \\ &= \ddot{\mathbf{p}} - \mathbf{s}^* \dot{\boldsymbol{\omega}} - (\boldsymbol{\omega} \times \mathbf{s})^* \boldsymbol{\omega} \qquad \text{(B.22)} \end{aligned}$$

(cf. similar derivations in [13]). We have already moved the 'chosen variables' to the rightmost position of each product. We will now factor $\dot{\mathbf{c}}$ and write out the components of J in terms of the vector components:

$$\dot{\mathbf{c}} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \dot{\mathbf{p}} + \begin{bmatrix} 0 & s_3 & -s_2 \\ -s_3 & 0 & s_1 \\ s_2 & -s_1 & 0 \end{bmatrix} \boldsymbol{\omega} \tag{B.23}$$

The two matrices in equation B.23 thus form two slices of J at the locations appropriate for $\dot{\mathbf{p}}$ and $\boldsymbol{\omega}$. We now continue to the next step of the procedure:

$$\begin{aligned} \dot{\mathsf{J}}\dot{\mathbf{x}} = \ddot{\mathbf{c}} - \mathsf{J}\ddot{\mathbf{x}} &= (\ddot{\mathbf{p}} - \mathbf{s}^* \dot{\boldsymbol{\omega}} - (\boldsymbol{\omega} \times \mathbf{s})^* \boldsymbol{\omega}) - (\ddot{\mathbf{p}} - \mathbf{s}^* \dot{\boldsymbol{\omega}}) \\ &= -(\boldsymbol{\omega} \times \mathbf{s})^* \boldsymbol{\omega} \\ &= \begin{bmatrix} 0 & \omega_1 s_2 - \omega_2 s_1 & \omega_1 s_3 - \omega_3 s_1 \\ \omega_2 s_1 - \omega_1 s_2 & 0 & \omega_2 s_3 - \omega_3 s_2 \\ \omega_3 s_1 - \omega_1 s_3 & \omega_3 s_2 - \omega_2 s_3 & 0 \end{bmatrix} \boldsymbol{\omega} \end{aligned} \tag{B.24}$$

We see that here there is only one slice for $\dot{\mathsf{J}}$; the slice belonging to $\ddot{\mathbf{p}}$ is zero. Since $\boldsymbol{\omega}$ also occurs inside the matrix, there are actually several alternative representations of this matrix which are equally valid.

B.4.2 Ball-and-socket joint

Two rigid bodies are attached together at a particular point in each of the bodies. They may not separate, but all three rotational degrees of freedom are permitted. This is a good representation e.g. of a human shoulder joint.

Let $\mathbf{a}$ and $\mathbf{b}$ be the positions of the centres of mass in the first and second rigid body respectively. Let $\mathbf{s}$ be the vector from $\mathbf{a}$ to the attachment point, and $\mathbf{t}$ the vector from $\mathbf{b}$ to the attachment point. Also let $\boldsymbol{\omega}$ be the angular velocity of the first body, and $\boldsymbol{\phi}$ that of the second. Then our constraint function and its derivatives are:

$$\mathbf{c} = \mathbf{a} + \mathbf{s} - \mathbf{b} - \mathbf{t} \tag{B.25}$$

$$\dot{\mathbf{c}} = \dot{\mathbf{a}} + \boldsymbol{\omega} \times \mathbf{s} - \dot{\mathbf{b}} - \boldsymbol{\phi} \times \mathbf{t} \tag{B.26}$$

$$\ddot{\mathbf{c}} = \ddot{\mathbf{a}} + \dot{\boldsymbol{\omega}} \times \mathbf{s} + \boldsymbol{\omega} \times (\boldsymbol{\omega} \times \mathbf{s}) - \ddot{\mathbf{b}} - \dot{\boldsymbol{\phi}} \times \mathbf{t} - \boldsymbol{\phi} \times (\boldsymbol{\phi} \times \mathbf{t}) \tag{B.27}$$

The rest of the derivation is very similar to the previous example. We obtain four matrix slices for J and two slices for $\dot{\mathsf{J}}$.

B.4.3 Rotation axis restriction ('Joystick')

We now have formulae to define a ball-and-socket joint. How can we express other types of joints? A good way of doing this is by augmenting the ball-and-socket constraint with additional constraints which restrict the set of valid rotations. In this section I derive expressions for a constraint which prohibits rotation about one particular axis – or, in other words, confines the axis of any valid rotation to a plane. In engineering terms, this is called a universal joint [23]. Let us define a unit vector $\mathbf{n}$ which points in the direction of the axis we want to prohibit; equivalently, this is the normal of the confinement plane.

It is not completely easy to visualize what this type of constraint means. One good way to look at it is to consider a standard two-axis joystick. If it is placed on a table, the two axes of rotation lie in a plane parallel to the surface of this table. But you cannot turn the stick about its own axis. Hence the normal of the constraint plane is orthogonal to the table surface. Don't be confused by the fact that the joystick handle happens to point in the direction of the normal – any sort of obscure shape may be substituted in its place without changing the nature of the constraint!

A more common sort of joint is the *hinge* or *revolute joint*, which we find in most doors, in our knees and elbows. It allows rotation only about one particular axis. We can conveniently express it by employing two 'joystick' constraints on the same body, each of which confines the axis to a plane. Provided the two planes are not parallel, the axis about which rotation may occur is just the line of intersection of these two planes. In summary, to make a revolute joint, we first add a ball-and-socket joint. Then we find two non-collinear vectors which are both orthogonal to the hinge axis, and use them as normal vectors for two 'joystick' constraints. This reduces the original number of six degrees of freedom to one – the angle of the hinge.

For this derivation I will use an alternative notation for quaternions, which is used by Shoemake [24], amongst others. Instead of using the complex constants i, j and k, a quaternion is

written as a pair consisting of a scalar (the real part) and a 3D vector (the three imaginary parts):

$$\mathsf{q} = q_w + q_x\mathsf{i} + q_y\mathsf{j} + q_z\mathsf{k} = [q_w, (q_x, q_y, q_z)^T] = [q_w, \mathbf{q}_v] \tag{B.28}$$

Using this notation, we can write the quaternion product in terms of vector dot and cross products:

$$\mathsf{pq} = [p_w,\ \mathbf{p}_v]\ [q_w,\ \mathbf{q}_v] = [p_w q_w - \mathbf{p}_v \cdot \mathbf{q}_v,\ p_w\mathbf{q}_v + q_w\mathbf{p}_v + \mathbf{p}_v \times \mathbf{q}_v] \tag{B.29}$$

We shall now consider the relative rotation of two rigid bodies. Say the first body has an orientation quaternion p and angular velocity $\boldsymbol{\phi}$, and the second body orientation q and angular velocity $\boldsymbol{\omega}$. Assume that each quaternion expresses the rotation required to transform from the body's frame to the world frame. Then the quaternion product $\mathsf{p}^{-1}\mathsf{q}$ is the rotation required to transform from the second body's frame to the first one's – that is, the relative rotation of the two bodies.

To confine the axis of rotation, we use the fact that the axis is contained in the imaginary parts of a quaternion (equation 2.8, page 12). We want the dot product of this axis and the normal vector $\mathbf{n}$ to be zero. Conveniently, the dot product happens to be implicitly present in the real part of the quaternion product (see equation B.29). Hence we can define our constraint function as follows:

$$c = \Re(\tilde{\mathbf{n}}\,\mathsf{p}^{-1}\mathsf{q}) \tag{B.30}$$

As with complex numbers, the function $\Re$ returns only the real part of its argument. Since the real part of $\tilde{\mathbf{n}}$ is zero by definition, this is just the dot product of $\mathbf{n}$ and the axis of $\mathsf{p}^{-1}\mathsf{q}$, with an extra minus sign in front (cf. equation B.29). The constraint function is a scalar because we are only losing one degree of freedom.

The derivative of q with respect to time is $\dot{\mathsf{q}} = \frac{1}{2}\tilde{\boldsymbol{\omega}}\mathsf{q}$. Pushing the differential operator onto the inside of a quaternion inverse produces a minus sign and reverses the order, provided we are dealing with a unit quaternion:

$$\frac{\mathrm{d}}{\mathrm{d}t}\mathsf{p} = \frac{1}{2}\tilde{\boldsymbol{\phi}}\,\mathsf{p} \iff \frac{\mathrm{d}}{\mathrm{d}t}(\mathsf{p}^{-1}) = -\frac{1}{2}\mathsf{p}\,\tilde{\boldsymbol{\phi}} \tag{B.31}$$

We now have everything in place to calculate the constraint function derivatives:

$$\dot{c} = \frac{1}{2}\Re(\tilde{\mathbf{n}}\,\mathsf{p}^{-1}\,\tilde{\boldsymbol{\omega}}\,\mathsf{q}) - \frac{1}{2}\Re(\tilde{\mathbf{n}}\,\mathsf{p}^{-1}\,\tilde{\boldsymbol{\phi}}\,\mathsf{q}) \tag{B.32}$$

$$\begin{aligned}\ddot{c} &= \frac{1}{2}\Re(\tilde{\mathbf{n}}\,\mathsf{p}^{-1}\,\tilde{\dot{\boldsymbol{\omega}}}\,\mathsf{q}) - \frac{1}{2}\Re(\tilde{\mathbf{n}}\,\mathsf{p}^{-1}\,\tilde{\dot{\boldsymbol{\phi}}}\,\mathsf{q}) \\ &\quad + \frac{1}{4}\Re(\tilde{\mathbf{n}}\,\mathsf{p}^{-1}\,\tilde{\boldsymbol{\omega}}\,\tilde{\boldsymbol{\omega}}\,\mathsf{q}) - \frac{1}{2}\Re(\tilde{\mathbf{n}}\,\mathsf{p}^{-1}\,\tilde{\boldsymbol{\phi}}\,\tilde{\boldsymbol{\omega}}\,\mathsf{q}) + \frac{1}{4}\Re(\tilde{\mathbf{n}}\,\mathsf{p}^{-1}\,\tilde{\boldsymbol{\phi}}\,\tilde{\boldsymbol{\phi}}\,\mathsf{q})\end{aligned} \tag{B.33}$$

We manipulate these equations into the form required to find J:

$$\begin{aligned}\Re(\tilde{\mathbf{n}}\,\mathsf{p}^{-1}\,\tilde{\boldsymbol{\omega}}\,\mathsf{q}) &= \Re([0,\ \mathbf{n}]\ [p_w,\ -\mathbf{p}_v]\ [0,\ \boldsymbol{\omega}]\ [q_w,\ \mathbf{q}_v]) \\ &= \Re([\mathbf{n}\cdot\mathbf{p}_v,\ p_w\mathbf{n} - \mathbf{n}\times\mathbf{p}_v]\ [-\boldsymbol{\omega}\cdot\mathbf{q}_v,\ q_w\boldsymbol{\omega} + \boldsymbol{\omega}\times\mathbf{q}_v]) \\ &= -(\mathbf{n}\cdot\mathbf{p}_v)(\boldsymbol{\omega}\cdot\mathbf{q}_v) - (p_w\mathbf{n} - \mathbf{n}\times\mathbf{p}_v)\cdot(q_w\boldsymbol{\omega} + \boldsymbol{\omega}\times\mathbf{q}_v) \\ &= -\mathbf{n}^T\mathbf{p}_v\mathbf{q}_v^T\boldsymbol{\omega} - (p_w\mathbf{n} + \mathbf{p}_v\times\mathbf{n})^T(q_w\boldsymbol{\omega} - \mathbf{q}_v\times\boldsymbol{\omega}) \\ &= -\mathbf{n}^T\mathbf{p}_v\mathbf{q}_v^T\boldsymbol{\omega} - (p_w\mathbf{n}^T - \mathbf{n}^T\mathbf{p}_v^*)(q_w\boldsymbol{\omega} - \mathbf{q}_v^*\,\boldsymbol{\omega}) \\ &= -\mathbf{n}^T(\mathbf{p}_v\mathbf{q}_v^T + (p_w 1 - \mathbf{p}_v^*)(q_w 1 - \mathbf{q}_v^*))\,\boldsymbol{\omega}\end{aligned}$$

Here 1 denotes the 3×3 identity matrix. The same derivation is valid if we substitute $\boldsymbol{\phi}$ for $\boldsymbol{\omega}$, hence we obtain the Jacobian

$$\begin{aligned} \mathsf{J}\dot{\mathbf{x}} &= -\frac{1}{2}\mathbf{n}^T(\mathbf{p}_v\mathbf{q}_v^T + (p_w 1 - \mathbf{p}_v^*)(q_w 1 - \mathbf{q}_v^*))\,\boldsymbol{\omega} \\ &\quad +\frac{1}{2}\mathbf{n}^T(\mathbf{p}_v\mathbf{q}_v^T + (p_w 1 - \mathbf{p}_v^*)(q_w 1 - \mathbf{q}_v^*))\,\boldsymbol{\phi} \end{aligned} \tag{B.34}$$

Now the first two terms of equation B.33 are generated by $\mathsf{J}\ddot{x}$, so for finding $\dot{\mathsf{J}}$ we need only consider the last three terms. Let us evaluate the penultimate term:

$$\begin{aligned} \Re(\tilde{\mathbf{n}}\,\mathsf{p}^{-1}\,\tilde{\boldsymbol{\phi}}\,\tilde{\boldsymbol{\omega}}\,\mathsf{q}) &= \Re([0,\,\mathbf{n}]\,[p_w,\,-\mathbf{p}_v]\,[0,\,\boldsymbol{\phi}]\,[0,\,\boldsymbol{\omega}]\,[q_w,\,\mathbf{q}_v]) \\ &= \Re([0,\,\mathbf{n}]\,[\mathbf{p}_v\cdot\boldsymbol{\phi},\,p_w\boldsymbol{\phi} - \mathbf{p}_v\times\boldsymbol{\phi}]\,[-\boldsymbol{\omega}\cdot\mathbf{q}_v,\,q_w\boldsymbol{\omega} + \boldsymbol{\omega}\times\mathbf{q}_v]) \\ &= \Re\Big(\ [\ \ \mathbf{n}\cdot(\mathbf{p}_v\times\boldsymbol{\phi}) - p_w\boldsymbol{\phi}\cdot\mathbf{n}, \\ &\qquad\quad (\mathbf{p}_v\cdot\boldsymbol{\phi})\mathbf{n} + p_w\mathbf{n}\times\boldsymbol{\phi} - \mathbf{n}\times(\mathbf{p}_v\times\boldsymbol{\phi})\ \ ] \\ &\qquad [\ \ -\boldsymbol{\omega}\cdot\mathbf{q}_v,\,q_w\boldsymbol{\omega} + \boldsymbol{\omega}\times\mathbf{q}_v\ \ ]\qquad\qquad\qquad\Big) \\ &= -(\mathbf{n}\cdot(\mathbf{p}_v\times\boldsymbol{\phi}) - p_w\boldsymbol{\phi}\cdot\mathbf{n})(\boldsymbol{\omega}\cdot\mathbf{q}_v) \\ &\quad -((\mathbf{p}_v\cdot\boldsymbol{\phi})\mathbf{n} + p_w\mathbf{n}\times\boldsymbol{\phi} - \mathbf{n}\times(\mathbf{p}_v\times\boldsymbol{\phi}))\cdot(q_w\boldsymbol{\omega} - \mathbf{q}_v\times\boldsymbol{\omega}) \\ &= -(\mathbf{n}\cdot(\mathbf{p}_v\times\boldsymbol{\phi}))(\mathbf{q}_v\cdot\boldsymbol{\omega}) + p_w(\mathbf{n}\cdot\boldsymbol{\phi})(\mathbf{q}_v\cdot\boldsymbol{\omega}) \\ &\quad +(\mathbf{n}\cdot(\mathbf{q}_v\times\boldsymbol{\omega}))(\mathbf{p}_v\cdot\boldsymbol{\phi}) - q_w(\mathbf{n}\cdot\boldsymbol{\omega})(\mathbf{p}_v\cdot\boldsymbol{\phi}) \\ &\quad -(\mathbf{n}\times(p_w\boldsymbol{\phi} - \mathbf{p}_v\times\boldsymbol{\phi}))\cdot(q_w\boldsymbol{\omega} - \mathbf{q}_v\times\boldsymbol{\omega}) \\ &= \mathbf{n}^T(p_w 1 - \mathbf{p}_v^*)\,\boldsymbol{\phi}\,\mathbf{q}_v^T\boldsymbol{\omega} - \mathbf{n}^T(q_w 1 - \mathbf{q}_v^*)\,\boldsymbol{\omega}\,\mathbf{p}_v^T\boldsymbol{\phi} \\ &\quad +(p_w\boldsymbol{\phi} - \mathbf{p}_v\times\boldsymbol{\phi})^T\mathbf{n}^*(q_w 1 - \mathbf{q}_v^*)\,\boldsymbol{\omega} \end{aligned}$$

Fortunately, the other two terms of equation B.33 we are interested in are similar, so we can obtain them by substitution in the last expression. This gives us the following expression for $\dot{\mathsf{J}}$:

$$\begin{aligned} \dot{\mathsf{J}}\dot{\mathbf{x}} &= \frac{1}{4}\Big(\mathbf{n}^T(p_w 1 - \mathbf{p}_v^*)\,\boldsymbol{\omega}\,\mathbf{q}_v^T - \mathbf{n}^T(q_w 1 - \mathbf{q}_v^*)\,\boldsymbol{\omega}\,\mathbf{p}_v^T \\ &\quad +(p_w\boldsymbol{\omega} - \mathbf{p}_v\times\boldsymbol{\omega})^T\mathbf{n}^*(q_w 1 - \mathbf{q}_v^*) \\ &\quad -2\mathbf{n}^T(p_w 1 - \mathbf{p}_v^*)\,\boldsymbol{\phi}\,\mathbf{q}_v^T - 2(p_w\boldsymbol{\phi} - \mathbf{p}_v\times\boldsymbol{\phi})^T\mathbf{n}^*(q_w 1 - \mathbf{q}_v^*)\Big)\,\boldsymbol{\omega} + \\ &\quad \frac{1}{4}\Big(\mathbf{n}^T(p_w 1 - \mathbf{p}_v^*)\,\boldsymbol{\phi}\,\mathbf{q}_v^T - \mathbf{n}^T(q_w 1 - \mathbf{q}_v^*)\,\boldsymbol{\phi}\,\mathbf{p}_v^T \\ &\quad +(p_w\boldsymbol{\phi} - \mathbf{p}_v\times\boldsymbol{\phi})^T\mathbf{n}^*(q_w 1 - \mathbf{q}_v^*) \\ &\quad +2\mathbf{n}^T(q_w 1 - \mathbf{q}_v^*)\,\boldsymbol{\omega}\,\mathbf{p}_v^T\Big)\,\boldsymbol{\phi} \end{aligned} \tag{B.35}$$

B.4.4 Rotation angle limitation

As mentioned in section 3.3.4, the constraint function of the last section can be used in an inequality context to limit the angle of rotation rather than just the axes. Adding a constant value to the constraint function makes no difference to its derivatives and therefore leaves the Jacobians derived in the last section unchanged.

So the implementation is easy, but the challenge is to understand what effect such an inequality actually has. Since quaternions are rather hard to visualize, let us translate the meaning of equation B.30 into Euler angles. Choose three orthogonal axes with basis vectors **a**, **b** and

$\mathbf{g}$ such that $\mathbf{a} \times \mathbf{b} = \mathbf{g}$ and $\mathbf{b} \times \mathbf{g} = \mathbf{a}$ and $\mathbf{g} \times \mathbf{a} = \mathbf{b}$ and $|\mathbf{a}| = |\mathbf{b}| = |\mathbf{g}| = 1$. Consider the rotation $\mathsf{p}^{-1}\mathsf{q}$ (the relative rotation between the two bodies), and decompose it into a sequence of three rotations: first a rotation of α about the $\mathbf{a}$ axis, then a rotation of β about the $\mathbf{b}$ axis, and finally a rotation of γ about the $\mathbf{g}$ axis. Thus we have

$$\begin{aligned} \mathsf{p}^{-1}\mathsf{q} &= \mathsf{gba} && \text{(B.36)} \\ \mathsf{a} &= \cos(\frac{\alpha}{2}) + \tilde{\mathbf{a}}\sin(\frac{\alpha}{2}) && \text{(B.37)} \\ \mathsf{b} &= \cos(\frac{\beta}{2}) + \tilde{\mathbf{b}}\sin(\frac{\beta}{2}) && \text{(B.38)} \\ \mathsf{g} &= \cos(\frac{\gamma}{2}) + \tilde{\mathbf{g}}\sin(\frac{\gamma}{2}) && \text{(B.39)} \end{aligned}$$

Now if we evaluate the constraint function about each of the axes $\mathbf{a}$, $\mathbf{b}$ and $\mathbf{g}$, we get (skipping some boring algebra):

$$\begin{aligned} \Re(\tilde{\mathbf{a}}\,\mathsf{gba}) &= -\sin(\frac{\alpha}{2})\cos(\frac{\beta}{2})\cos(\frac{\gamma}{2}) - \cos(\frac{\alpha}{2})\sin(\frac{\beta}{2})\sin(\frac{\gamma}{2}) && \text{(B.40)} \\ \Re(\tilde{\mathbf{b}}\,\mathsf{gba}) &= -\cos(\frac{\alpha}{2})\sin(\frac{\beta}{2})\cos(\frac{\gamma}{2}) - \sin(\frac{\alpha}{2})\cos(\frac{\beta}{2})\sin(\frac{\gamma}{2}) && \text{(B.41)} \\ \Re(\tilde{\mathbf{g}}\,\mathsf{gba}) &= -\cos(\frac{\alpha}{2})\cos(\frac{\beta}{2})\sin(\frac{\gamma}{2}) - \sin(\frac{\alpha}{2})\sin(\frac{\beta}{2})\cos(\frac{\gamma}{2}) && \text{(B.42)} \end{aligned}$$

These expressions make clear how interdependent the three axes are – it is generally not possible to change a constraint on one axis without affecting the others. The best way to proceed from here is to enter inequalities using formulae B.40 to B.42 into a program for graphical visualization, and to experimentally determine the values such that the desired behaviour is achieved.

B.4.5 Confinement to a plane (vertex/face collision)

We want to define a constraint function whose value is the distance between a point and a plane, where the point is attached to one rigid body, and the plane to another. The plane is defined by a point in the plane and a normal vector. The distance should be positive if the point is on the side of the plane pointed to by the normal, and negative if it is on the opposite side. If we put this constraint directly into the Lagrange multiplier equation, it will enforce the condition that the distance be zero – the point is confined to move only within the plane. But if we use the same constraint function in an inequality, we have a handler for the vertex/face collision case.

Say $\mathbf{a}$ is the centre of mass position of the body to which the plane is attached, and $\mathbf{s}$ is the vector from the centre of mass to an arbitrary point in the plane. The angular velocity of this body is $\boldsymbol{\omega}$. The plane has a unit normal vector $\hat{\mathbf{n}}$ ($|\hat{\mathbf{n}}| = 1$). The point we are interested in is $\mathbf{b} + \mathbf{t}$, where $\mathbf{b}$ is the centre of mass position of the body to which the point belongs. This body has angular velocity $\boldsymbol{\phi}$.

Then the constraint function and its derivatives are given by

$$\begin{aligned} c &= (\mathbf{b} + \mathbf{t} - \mathbf{a} - \mathbf{s}) \cdot \hat{\mathbf{n}} && \text{(B.43)} \\ \dot{c} &= (\dot{\mathbf{b}} + \boldsymbol{\phi} \times \mathbf{t} - \dot{\mathbf{a}} - \boldsymbol{\omega} \times \mathbf{s}) \cdot \hat{\mathbf{n}} + (\mathbf{b} + \mathbf{t} - \mathbf{a} - \mathbf{s}) \cdot (\boldsymbol{\omega} \times \hat{\mathbf{n}}) \\ &= \hat{\mathbf{n}}^T\dot{\mathbf{b}} - \hat{\mathbf{n}}^T\dot{\mathbf{a}} - \hat{\mathbf{n}}^T\mathbf{t}^*\boldsymbol{\phi} + (\mathbf{a} - \mathbf{b} - \mathbf{t})^T\hat{\mathbf{n}}^*\boldsymbol{\omega} && \text{(B.44)} \\ \ddot{c} &= (\ddot{\mathbf{b}} + \dot{\boldsymbol{\phi}} \times \mathbf{t} + \boldsymbol{\phi} \times (\boldsymbol{\phi} \times \mathbf{t}) - \ddot{\mathbf{a}}) \cdot \hat{\mathbf{n}} + \\ & \quad 2(\dot{\mathbf{b}} + \boldsymbol{\phi} \times \mathbf{t} - \dot{\mathbf{a}}) \cdot (\boldsymbol{\omega} \times \hat{\mathbf{n}}) + (\mathbf{b} + \mathbf{t} - \mathbf{a}) \cdot (\dot{\boldsymbol{\omega}} \times \hat{\mathbf{n}} + \boldsymbol{\omega} \times (\boldsymbol{\omega} \times \hat{\mathbf{n}})) \end{aligned}$$

$$= \hat{\mathbf{n}}^T\ddot{\mathbf{b}} - \hat{\mathbf{n}}^T\ddot{\mathbf{a}} - \hat{\mathbf{n}}^T\mathbf{t}^*\dot{\boldsymbol{\phi}} + (\mathbf{a}-\mathbf{b}-\mathbf{t})^T\hat{\mathbf{n}}^*\dot{\boldsymbol{\omega}} + \tag{B.45}$$
$$(2(\dot{\mathbf{a}} - \dot{\mathbf{b}} - \boldsymbol{\phi}\times\mathbf{t})^T\hat{\mathbf{n}}^* + (\mathbf{a}-\mathbf{b}-\mathbf{t})^T(\boldsymbol{\omega}\times\hat{\mathbf{n}})^*)\boldsymbol{\omega} - \hat{\mathbf{n}}^T(\boldsymbol{\phi}\times\mathbf{t})^*\boldsymbol{\phi}$$

from which J and $\dot{\mathsf{J}}$ can be read off as usual.

B.4.6 Edge/edge collision

In this section we require a constraint function whose value is the shortest distance between two straight lines. The problem is closely related to the one in section B.4.5. If used as an equality constraint, it could be used to simulate two metal rods which are joined together such that the join can move up or down either of the rods, but the rods always have to touch in one point – a kind of combination of a ball-and-socket joint with two one-dimensional sliding rails. However, the practical use of such a system is rather limited. Much more important is the use of this constraint as an inequality, where it can handle the collision situation in which two edges collide.

We assume that each straight line is connected to a rigid body. The first body's centre of mass is located at $\mathbf{a}$, its angular velocity is $\boldsymbol{\omega}$, $\mathbf{s}$ is a vector from the centre of mass to an arbitrary point on the line, and $\mathbf{u}$ is a vector pointing along the line (unit magnitude is not required). Similarly, the second body's CoM is $\mathbf{b}$, its angular velocity is $\boldsymbol{\phi}$, $\mathbf{t}$ points at the line and $\mathbf{v}$ points in the line's direction. In this derivation we shall assume that the lines are not parallel ($|\mathbf{u}\times\mathbf{v}| \neq \mathbf{0}$); the parallel case is special and needs to be handled separately.

If $\mathbf{u}$ and $\mathbf{v}$ are not parallel, we can find a unique plane which contains one line and is parallel to the other. This plane has a normal vector $\mathbf{n} = \mathbf{u}\times\mathbf{v}$ (or $\mathbf{n} = \mathbf{v}\times\mathbf{u}$). Interestingly this normal is the direction in which the force or impulse acts in the event of a collision. It is not obvious that this is the case, but by playing around with two books or similar it is possible to convince oneself. Then the closest distance between the two lines is given by

$$c = (\mathbf{b}+\mathbf{t}-\mathbf{a}-\mathbf{s})\cdot\frac{\mathbf{n}}{|\mathbf{n}|} \tag{B.46}$$

The derivation of the Jacobian matrices involves some of the most messy linear algebra in this project, so hold on tight. Let us first define a few auxiliary variables:

$$\mathbf{n} = \mathbf{u}\times\mathbf{v} \tag{B.47}$$
$$h = \frac{1}{|\mathbf{n}|} = (\mathbf{n}\cdot\mathbf{n})^{-\frac{1}{2}} \tag{B.48}$$
$$\begin{aligned}\mathbf{z} &= \dot{\mathbf{n}} = \dot{\mathbf{u}}\times\mathbf{v} + \mathbf{u}\times\dot{\mathbf{v}}\\ &= (\boldsymbol{\omega}\times\mathbf{u})\times\mathbf{v} + \mathbf{u}\times(\boldsymbol{\phi}\times\mathbf{v})\\ &= \mathbf{v}^*\mathbf{u}^*\boldsymbol{\omega} - \mathbf{u}^*\mathbf{v}^*\boldsymbol{\phi}\end{aligned} \tag{B.49}$$
$$\begin{aligned}\mathbf{y} &= \dot{\mathbf{z}} = \ddot{\mathbf{u}}\times\mathbf{v} + 2\,\dot{\mathbf{u}}\times\dot{\mathbf{v}} + \mathbf{u}\times\ddot{\mathbf{v}}\\ &= (\dot{\boldsymbol{\omega}}\times\mathbf{u})\times\mathbf{v} + (\boldsymbol{\omega}\times(\boldsymbol{\omega}\times\mathbf{u}))\times\mathbf{v} + 2\,(\boldsymbol{\omega}\times\mathbf{u})\times(\boldsymbol{\phi}\times\mathbf{v}) +\\ &\quad \mathbf{u}\times(\dot{\boldsymbol{\phi}}\times\mathbf{v}) + \mathbf{u}\times(\boldsymbol{\phi}\times(\boldsymbol{\phi}\times\mathbf{v}))\\ &= \mathbf{v}^*\mathbf{u}^*\dot{\boldsymbol{\omega}} - \mathbf{u}^*\mathbf{v}^*\dot{\boldsymbol{\phi}} + \mathbf{v}^*(\boldsymbol{\omega}\times\mathbf{u})^*\boldsymbol{\omega} - \mathbf{u}^*(\boldsymbol{\phi}\times\mathbf{v})^*\boldsymbol{\phi} + 2\,(\boldsymbol{\phi}\times\mathbf{v})^*\mathbf{u}^*\boldsymbol{\omega}\end{aligned} \tag{B.50}$$
$$\hat{\mathbf{n}} = h\mathbf{n} \tag{B.51}$$
$$\begin{aligned}\dot{\hat{\mathbf{n}}} &= h\mathbf{z} - \frac{1}{2}\mathbf{n}(\mathbf{n}\cdot\mathbf{n})^{-\frac{3}{2}}(\mathbf{z}\cdot\mathbf{n} + \mathbf{n}\cdot\mathbf{z})\\ &= h\mathbf{z} - h^3\,(\mathbf{n}\cdot\mathbf{z})\,\mathbf{n}\end{aligned} \tag{B.52}$$
$$\ddot{\hat{\mathbf{n}}} = h\mathbf{y} - 2h^3(\mathbf{n}\cdot\mathbf{z})\,\mathbf{z} - h^3(\mathbf{n}\cdot\mathbf{y})\,\mathbf{n} - h^3(\mathbf{z}\cdot\mathbf{z})\,\mathbf{n} + 3h^5(\mathbf{n}\cdot\mathbf{z})^2\,\mathbf{n} \tag{B.53}$$

Now we can turn to calculating the derivatives of c:

$$
\begin{aligned}
\dot{c} &= \hat{\mathbf{n}} \cdot (\dot{\mathbf{b}} + \boldsymbol{\phi} \times \mathbf{t} - \dot{\mathbf{a}} - \boldsymbol{\omega} \times \mathbf{s}) + (\mathbf{b} + \mathbf{t} - \mathbf{a} - \mathbf{s}) \cdot \dot{\hat{\mathbf{n}}} && \text{(B.54)} \\
&= h\,(\mathbf{u} \times \mathbf{v})^T (\dot{\mathbf{b}} + \boldsymbol{\phi} \times \mathbf{t} - \dot{\mathbf{a}} - \boldsymbol{\omega} \times \mathbf{s}) + \\
&\quad (\mathbf{b} + \mathbf{t} - \mathbf{a} - \mathbf{s})^T (h\mathbf{z} - h^3\,(\mathbf{n} \cdot \mathbf{z})\,\mathbf{n}) \\
&= h\mathbf{u}^T\mathbf{v}^*\dot{\mathbf{b}} - h\mathbf{u}^T\mathbf{v}^*\dot{\mathbf{a}} - h\mathbf{u}^T\mathbf{v}^*\mathbf{t}^*\boldsymbol{\phi} + h\mathbf{u}^T\mathbf{v}^*\mathbf{s}^*\boldsymbol{\omega} + \\
&\quad (\mathbf{b} + \mathbf{t} - \mathbf{a} - \mathbf{s})^T (h\mathbf{v}^*\mathbf{u}^*\boldsymbol{\omega} - h\mathbf{u}^*\mathbf{v}^*\boldsymbol{\phi} - h^3\mathbf{u}^*\mathbf{v}\mathbf{u}^T\mathbf{v}^*(\mathbf{v}^*\mathbf{u}^*\boldsymbol{\omega} - \mathbf{u}^*\mathbf{v}^*\boldsymbol{\phi})) \\
&= h\mathbf{u}^T\mathbf{v}^*\dot{\mathbf{b}} - h\mathbf{u}^T\mathbf{v}^*\dot{\mathbf{a}} \\
&\quad + \Big(h\mathbf{u}^T\mathbf{v}^*\mathbf{s}^* + h\,(\mathbf{b} + \mathbf{t} - \mathbf{a} - \mathbf{s})^T (1 - h^2\mathbf{u}^*\mathbf{v}\mathbf{u}^T\mathbf{v}^*)\mathbf{v}^*\mathbf{u}^*\Big)\,\boldsymbol{\omega} \\
&\quad - \Big(h\mathbf{u}^T\mathbf{v}^*\mathbf{t}^* + h\,(\mathbf{b} + \mathbf{t} - \mathbf{a} - \mathbf{s})^T (1 - h^2\mathbf{u}^*\mathbf{v}\mathbf{u}^T\mathbf{v}^*)\mathbf{u}^*\mathbf{v}^*\Big)\,\boldsymbol{\phi} && \text{(B.55)} \\
&= \mathsf{J}\dot{\mathbf{x}} \\
\ddot{c} &= \hat{\mathbf{n}} \cdot \Big(\ddot{\mathbf{b}} + \dot{\boldsymbol{\phi}} \times \mathbf{t} + \boldsymbol{\phi} \times (\boldsymbol{\phi} \times \mathbf{t}) - \ddot{\mathbf{a}} - \dot{\boldsymbol{\omega}} \times \mathbf{s} - \boldsymbol{\omega} \times (\boldsymbol{\omega} \times \mathbf{s})\Big) + \\
&\quad 2\left(\dot{\mathbf{b}} + \boldsymbol{\phi} \times \mathbf{t} - \dot{\mathbf{a}} - \boldsymbol{\omega} \times \mathbf{s}\right) \cdot \dot{\hat{\mathbf{n}}} + \left(\mathbf{b} + \mathbf{t} - \mathbf{a} - \mathbf{s}\right) \cdot \ddot{\hat{\mathbf{n}}} && \text{(B.56)} \\
&= h\mathbf{u}^T\mathbf{v}^*\ddot{\mathbf{b}} - h\mathbf{u}^T\mathbf{v}^*\ddot{\mathbf{a}} - h\mathbf{u}^T\mathbf{v}^*\mathbf{t}^*\dot{\boldsymbol{\phi}} + h\mathbf{u}^T\mathbf{v}^*\mathbf{s}^*\dot{\boldsymbol{\omega}} + \\
&\quad h\mathbf{u}^T\mathbf{v}^*(\boldsymbol{\omega} \times \mathbf{s})^*\boldsymbol{\omega} - h\mathbf{u}^T\mathbf{v}^*(\boldsymbol{\phi} \times \mathbf{t})^*\boldsymbol{\phi} \\
&\quad + 2h\,(\dot{\mathbf{b}} + \boldsymbol{\phi} \times \mathbf{t} - \dot{\mathbf{a}} - \boldsymbol{\omega} \times \mathbf{s})^T \Big(\mathbf{v}^*\mathbf{u}^*\boldsymbol{\omega} - \mathbf{u}^*\mathbf{v}^*\boldsymbol{\phi} \\
&\qquad - h^2\mathbf{u}^*\mathbf{v}\mathbf{u}^T\mathbf{v}^*(\mathbf{v}^*\mathbf{u}^*\boldsymbol{\omega} - \mathbf{u}^*\mathbf{v}^*\boldsymbol{\phi})\Big) \\
&\quad + (\mathbf{b} + \mathbf{t} - \mathbf{a} - \mathbf{s})^T \Big(\\
&\qquad h\,(\mathbf{v}^*\mathbf{u}^*\dot{\boldsymbol{\omega}} - \mathbf{u}^*\mathbf{v}^*\dot{\boldsymbol{\phi}} + \mathbf{v}^*(\boldsymbol{\omega} \times \mathbf{u})^*\boldsymbol{\omega} \\
&\qquad\quad - \mathbf{u}^*(\boldsymbol{\phi} \times \mathbf{v})^*\boldsymbol{\phi} + 2\,(\boldsymbol{\phi} \times \mathbf{v})^*\mathbf{u}^*\boldsymbol{\omega}) \\
&\qquad - 2h^3(\mathbf{v}^*\mathbf{u}^*\boldsymbol{\omega} - \mathbf{u}^*\mathbf{v}^*\boldsymbol{\phi})\mathbf{u}^T\mathbf{v}^*(\mathbf{v}^*\mathbf{u}^*\boldsymbol{\omega} - \mathbf{u}^*\mathbf{v}^*\boldsymbol{\phi}) \\
&\qquad - h^3\mathbf{u}^*\mathbf{v}\mathbf{u}^T\mathbf{v}^*(\mathbf{v}^*\mathbf{u}^*\dot{\boldsymbol{\omega}} - \mathbf{u}^*\mathbf{v}^*\dot{\boldsymbol{\phi}} + \mathbf{v}^*(\boldsymbol{\omega} \times \mathbf{u})^*\boldsymbol{\omega} \\
&\qquad\quad - \mathbf{u}^*(\boldsymbol{\phi} \times \mathbf{v})^*\boldsymbol{\phi} + 2\,(\boldsymbol{\phi} \times \mathbf{v})^*\mathbf{u}^*\boldsymbol{\omega}) \\
&\qquad - h^3\mathbf{u}^*\mathbf{v}(\boldsymbol{\omega}^T\mathbf{u}^*\mathbf{v}^* - \boldsymbol{\phi}^T\mathbf{v}^*\mathbf{u}^*)(\mathbf{v}^*\mathbf{u}^*\boldsymbol{\omega} - \mathbf{u}^*\mathbf{v}^*\boldsymbol{\phi}) \\
&\qquad + 3h^5\mathbf{u}^*\mathbf{v}\mathbf{u}^T\mathbf{v}^*(\mathbf{v}^*\mathbf{u}^*\boldsymbol{\omega} - \mathbf{u}^*\mathbf{v}^*\boldsymbol{\phi})\mathbf{u}^T\mathbf{v}^*(\mathbf{v}^*\mathbf{u}^*\boldsymbol{\omega} - \mathbf{u}^*\mathbf{v}^*\boldsymbol{\phi})\Big)
\end{aligned}
$$

Finally we subtract the terms generated by $\mathsf{J}\ddot{\mathbf{x}}$ from $\ddot{c}$ and separate into factors of $\boldsymbol{\omega}$ and $\boldsymbol{\phi}$ as usual:

$$
\begin{aligned}
\dot{\mathsf{J}}\dot{\mathbf{x}} \;=\; & \Big(h\mathbf{u}^T\mathbf{v}^*(\boldsymbol{\omega}\times\mathbf{s})^* \\
& \quad +2h\,(\dot{\mathbf{b}}+\boldsymbol{\phi}\times\mathbf{t}-\dot{\mathbf{a}}-\boldsymbol{\omega}\times\mathbf{s})^T(1-h^2\mathbf{u}^*\mathbf{v}\mathbf{u}^T\mathbf{v}^*)\mathbf{v}^*\mathbf{u}^* \\
& \quad +(\mathbf{b}+\mathbf{t}-\mathbf{a}-\mathbf{s})^T\Big(h\mathbf{v}^*(\boldsymbol{\omega}\times\mathbf{u})^*+2h\,(\boldsymbol{\phi}\times\mathbf{v})^*\mathbf{u}^* \\
& \qquad -h^3\mathbf{u}^*\mathbf{v}\mathbf{u}^T\mathbf{v}^*\mathbf{v}^*(\boldsymbol{\omega}\times\mathbf{u})^*-2h^3\mathbf{u}^*\mathbf{v}\mathbf{u}^T\mathbf{v}^*(\boldsymbol{\phi}\times\mathbf{v})^*\mathbf{u}^* \\
& \qquad -2h^3\,(\mathbf{v}^*\mathbf{u}^*\boldsymbol{\omega}-\mathbf{u}^*\mathbf{v}^*\boldsymbol{\phi})\mathbf{u}^T\mathbf{v}^*\mathbf{v}^*\mathbf{u}^* \\
& \qquad -h^3\mathbf{u}^*\mathbf{v}(\boldsymbol{\omega}^T\mathbf{u}^*\mathbf{v}^*-\boldsymbol{\phi}^T\mathbf{v}^*\mathbf{u}^* \\
& \qquad\quad -3h^2\mathbf{u}^T\mathbf{v}^*(\mathbf{v}^*\mathbf{u}^*\boldsymbol{\omega}-\mathbf{u}^*\mathbf{v}^*\boldsymbol{\phi})\mathbf{u}^T\mathbf{v}^*)\mathbf{v}^*\mathbf{u}^*\Big)\Big)\,\boldsymbol{\omega} \\
& -\Big(h\mathbf{u}^T\mathbf{v}^*(\boldsymbol{\phi}\times\mathbf{t})^* \\
& \quad +2h\,(\dot{\mathbf{b}}+\boldsymbol{\phi}\times\mathbf{t}-\dot{\mathbf{a}}-\boldsymbol{\omega}\times\mathbf{s})^T(1-h^2\mathbf{u}^*\mathbf{v}\mathbf{u}^T\mathbf{v}^*)\mathbf{u}^*\mathbf{v}^* \\
& \quad +(\mathbf{b}+\mathbf{t}-\mathbf{a}-\mathbf{s})^T\Big(h\mathbf{u}^*(\boldsymbol{\phi}\times\mathbf{v})^*-h^3\mathbf{u}^*\mathbf{v}\mathbf{u}^T\mathbf{v}^*\mathbf{u}^*(\boldsymbol{\phi}\times\mathbf{v})^* \\
& \qquad -2h^3\,(\mathbf{v}^*\mathbf{u}^*\boldsymbol{\omega}-\mathbf{u}^*\mathbf{v}^*\boldsymbol{\phi})\mathbf{u}^T\mathbf{v}^*\mathbf{u}^*\mathbf{v}^* \\
& \qquad -h^3\mathbf{u}^*\mathbf{v}(\boldsymbol{\omega}^T\mathbf{u}^*\mathbf{v}^*-\boldsymbol{\phi}^T\mathbf{v}^*\mathbf{u}^* \\
& \qquad\quad -3h^2\mathbf{u}^T\mathbf{v}^*(\mathbf{v}^*\mathbf{u}^*\boldsymbol{\omega}-\mathbf{u}^*\mathbf{v}^*\boldsymbol{\phi})\mathbf{u}^T\mathbf{v}^*)\mathbf{u}^*\mathbf{v}^*\Big)\Big)\,\boldsymbol{\phi}
\end{aligned}
\tag{B.57}
$$

Bibliography

[1] Peter H. Abrahams, Sandy C. Marks Jr., and Ralph Hutchings. *McMinn's Color Atlas of Human Anatomy*. Mosby/Elsevier, fifth edition, 2003.

[2] David Baraff. *Dynamic Simulation of Non-Penetrating Rigid Bodies*. PhD thesis, Cornell University, Department of Computer Science, March 1992.

[3] David Baraff. Linear-time dynamics using Lagrange multipliers. In *Proceedings of SIGGRAPH*, volume 23, pages 137–146. ACM, 1996.

[4] David Baraff and Andrew Witkin. Physically based modeling: Principles and practice. SIGGRAPH 1997 Course Notes. Available online at http://www.cs.cmu.edu/ baraff/sigcourse/.

[5] David H. Eberly. *3D Game Engine Design*. Morgan Kaufmann, 2001.

[6] David H. Eberly. *Game Physics*. Morgan Kaufmann series in interactive 3D technology. Morgan Kaufmann, 2004.

[7] Roy Featherstone. *Robot dynamics algorithms*. Kluwer international series in engineering and computer science. Kluwer, 1987.

[8] Richard P. Feynman, Robert B. Leighton, and Matthew Sands. *The Feynman Lectures on Physics*, volume 1. Addison-Wesley, 1963.

[9] Herbert Goldstein. *Classical Mechanics*. Addison-Wesley, second edition, 1980.

[10] Mark Green. Using dynamics in computer animation: Control and solution issues. In Norman I. Badler, Brian A. Barsky, and David Zeltzer, editors, *Making them Move*, chapter 14, pages 281–314. Morgan Kaufmann, 1991.

[11] Louis N. Hand and Janet D. Finch. *Analytical Mechanics*. Cambridge University Press, 1998.

[12] Stephen R. Julian. Mechanics and relativity. Lecture notes for Natural Sciences Tripos, Part 1A Physics, University of Cambridge, 2003.

[13] Devendra Kalra. A general formulation of rigid body assemblies for computer graphics modeling. Technical Report HPL-95-70, HP Labs, Palo Alto, CA, 1995.

[14] Brian Mirtich. Fast and accurate computation of polyhedral mass properties. *Journal of Graphics Tools*, 1(2):31–50, 1996.

[15] Brian V. Mirtich. *Impulse-based Dynamic Simulation of Rigid Body Systems*. PhD thesis, University of California at Berkeley, 1996.

[16] Tomas Möller. A fast triangle-triangle intersection test. *Journal of Graphics Tools*, 2(2):25–30, 1997.

[17] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C*. Cambridge University Press, second edition, 1992.

[18] Ken F. Riley, Mike P. Hobson, and Stephen J. Bence. *Mathematical Methods for Physics and Engineering*. Cambridge University Press, second edition, 2002.

[19] Wolf-Dietrich Ruf. Mechanische Systeme. In Edmund Schiessle, editor, *Mechatronik 2*, chapter 3, pages 211–264. Vogel Buchverlag, 2002.

[20] Breton M. Saunders. *Fast Animation Dynamics*. PhD thesis, University of Cambridge Computer Laboratory, May 2000.

[21] Breton M. Saunders. private communications, 2005.

[22] Robert Sedgewick. *Algorithms*. Addison-Wesley, 1983.

[23] Ahmed A. Shabana. *Computational Dynamics*. Wiley, 2001.

[24] Ken Shoemake. Animating rotation with quaternion curves. In *Proceedings of SIGGRAPH*, volume 19, pages 245–254. ACM, 1985.

[25] Stephen S. Skiena. *The Algorithm Design Manual*. Springer, 1998.

[26] Eric W. Weisstein. Four-dimensional geometry. From MathWorld, a Wolfram web resource. http://mathworld.wolfram.com/Four-DimensionalGeometry.html.

[27] Eric W. Weisstein. Quaternion. From MathWorld, a Wolfram web resource. http://mathworld.wolfram.com/Quaternion.html.

[28] Stephen A. Whitmore. Closed-form integrator for the quaternion (Euler angle) kinematics equations. United States Patent 6,061,611, May 2000. Assignee: The United States of America as represented by the Administrator of the National Aeronautics and Space Administration (NASA).

[29] Jane Wilhelms. Dynamic experiences. In Norman I. Badler, Brian A. Barsky, and David Zeltzer, editors, *Making them Move*, chapter 13, pages 265–279. Morgan Kaufmann, 1991.