Univerzita Karlova v Praze

Matematicko-fyzikální fakulta

DIPLOMOVÁ PRÁCE

Marek Vondrák

Dynamika živých organismů

Kabinet software a výuky informatiky

Vedoucí diplomové práce: **RNDr. Josef Pelikán**

Studijní program: **Informatika**

Rád bych poděkoval vedoucímu diplomové práce RNDr. Josefu Pelikánovi za poskytnutí studijních materiálů a dále pak Mgr. Jaroslavu Semančíkovi za jeho ohromující motion capture výstup, zpracování motion capture dat a jejich zapůjčení pro účely předvedení možností animační knihovny, která je součástí této práce.

Prohlašuji, že jsem svou diplomovou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce.

V Praze dne 14. prosince 2005 Marek Vondrák

Contents

Název práce: Dynamika živých organismů
Autor: Marek Vondrák
Katedra: Kabinet software a výuky informatiky
Vedoucí diplomové práce: RNDr. Josef Pelikán
e-mail vedoucího: Josef.Pelikan@mff.cuni.cz
Abstrakt: Počítačová animace artikulovaných postav je jedna z nejzajímavějších a nejvíce se rozvíjejících oblastí moderní počítačové grafiky. Cílem této práce je seznámit čtenáře s teorií simulace tuhých těles s omezeními, která je následně použita ke konstrukci obecného simulátoru tuhých těles s omezeními a třením a animační knihovny pro artikulované postavy. Postavy jsou reprezentovány soustavami tuhých těles (segmentů) propojených klouby a jejich pohyb určen dynamikou příslušných segmentů. Dodatečná omezení, předepisující např. požadované úhly v kloubech nebo pozice vybraných bodů na povrchu segmentů, umožňují řídit pohyb postav. Bohaté interaktivní demonstrační příklady předvádí vlastnosti samotného simulátoru a možnosti animační knihovny pro zpracování motion capture dat (přehrávání animací, přizpůsobování animací externím vlivům, mapování "syrových" motion capture dat na pohyb segmentů, atd.).
Klíčová slova: tuhé těleso, simulace, dynamika, omezení, artikulovaná postava, řízení pohybu

Title: Dynamics of Vertebratea
Author: Marek Vondrák
Department: Kabinet software a výuky informatiky
Supervisor: RNDr. Josef Pelikán
Supervisor's e-mail: Josef.Pelikan@mff.cuni.cz
Abstract: Computer animation of articulated figures is one of the most interesting and the most developing areas of modern computer graphics. The goal of this thesis is to get reader acquainted with the theory of constrained rigid body simulation, which is subsequently used to construct a generic rigid body simulator with constraints and friction and the figure library suitable for the animation of articulated human-like figures. Articulated figures are represented by sets of rigid bodies (segments) connected by joints and their motion is determined by the dynamics of the corresponding segments. Additional constraints, specifying e.g. desired angles at joints or the positions of selected sites on the surfaces of the figure segments, allow to control the figure motion. A rich set of interactive demonstration examples presents the features of the actual simulator and the capabilities of the figure library to process motion capture data (replay motion capture data, adapt the data to external influences, map the "raw" motion capture data to the motion of figure segments, etc).
Keywords: rigid body, simulation, dynamics, constraint, articulated figure, motion control

Chapter 1

Introduction

Computer animation of human-like figures has always been an interesting and no doubt challenging topic pursued by many researchers and computer graphics since the very beginning of modern computer graphics. The purpose of this work is to get reader acquainted with the theory of physical-based animation of articulated figures, present the most recent results of the research in that area and to demonstrate that the technology is ready to be used in the production environment (animation systems, computer games). To prove this, industry standard dynamics simulator, capable of running at interactive rates on today's low-end computers, was implemented.

The thesis consists of a theoretical part, describing the dynamics theory and some algorithms, and an implementation part, describing the implementation of the simulator and a figure library built on top of it. No prior awareness of the theory is required to read this thesis, however basic knowledge of calculus and linear algebra are demanded.

This chapter provides the reader with a brief information on the character animation background, gets the reader acquainted with the topics covered by the thesis and its goals. It is relied upon the reader's acquaintance with the computer animation basics as we will advance rather quickly and informally.

1.1 Background

The mere purpose of this section is to identify the context of the area of dynamics discussed throughout this thesis. We will briefly survey the most commonly used animation techniques of human-like figures and point out their advantages and disadvantages.

We proceed from the oldest and the most primitive techniques to more advanced ones. Older techniques are more "low level" as animators have to control every aspect of the produced animation and their work much resembles the work of traditional animators. While "lower level" techniques are very close to the raw figure representation (often assorted polygonal mesh), "higher level" techniques attempt to abstract the figure representation and motion to be generated. Such techniques are often domain specific (fit for a particular use only) and always use special properties of the figure to be animated (for example, they are aware of the fact that the figure consists of a torso, limbs, etc. and take advantage of it).

The most primitive approach, *vertex key-framing*, treats the whole figure as a polygonal mesh, where each polygon vertex is animated independently on each other. Animators provide the animation system with positions of polygon vertices in time (animation curves), defined at *key-frames*, and the animation system computes *in-between-frames* automatically by interpolating data from the key-frames. This simplifies the animation process a little, compared to the lowest level approach ever possible, where animator has to animate all frames by hand, but the work is still quite laborious[1]. Probably the greatest drawback of the method is the fact that all vertices are treated the same and has to be animated.

It became clear soon that it was advantageous to separate the animation of the figure's *"skin"* from the animation of its *"skeleton"*. The idea is that the animator would control figure's posture by animating its skeleton and the skin would deform accordingly (for the time being we can think of the skeleton as of the actual figure's skeleton, consisting of rigid segments corresponding to bones and joints connecting the segments, and treat its skin as the skeleton envelope, drawn by the animation system). The animation of the "skin", driven by the animation of the skeleton, is called the *skeletal animation*. However since it is the animation of the skeleton what makes the overall impression of the motion, interests of many researchers and animators have focused at the animation of skeleton, in fact it is probably the area that has been developing the most in the last decade.

Depending on the way how figure's skeleton is animated, we distinguish *forward* and *backward kinematics* and *forward* and *backward dynamics*. All these techniques treat figure segments[2] as rigid body segments connected by joints that can restrict relative orientation and position of the attached segments. In 3D kinematics, a figure segment state consists of the segment position and orientation in the world space and can be parametrized by six scalar parameters called degrees-of-freedom, DOFs, where three scalars parametrize segment displacements along the three world space axes and the other three scalars parametrize angles about the axes[3]. Segment states are represented by vectors from $\mathbf{R}^6$ called the segment state space. If a segment gets connected to another segment, some of its DOFs will become constrained, that is their values will be determined by the values of other unconstrained DOFs in the system, and so the set of *valid*[4] states will reduce. The valid states of the constrained figure segments can be then parametrized either by a set of six constrained coordinates, called the *maximal coordinates*, or by a reduced set of < 6 unconstrained coordinates, called the *generalized coordinates*. In the latter case the constrained DOFs are actually removed from the system because the generalized coordinates can serve the purpose of the new unconstrained DOFs. By concatenating individual maximal or generalized coordinates of the figure segment states in a certain order, one gets the maximal or generalized parametrization of the whole figure state. That segment that was concatenated the first is called the root segment.

For example if we have a figure consisting of two segments A and B, where A is the root segment and B is attached to A by a joint that allows B only to rotate with respect to A along an axis attached to A (thus removing all translational and two rotational DOFs, leaving the one rotational DOF unconstrained), then we can write:

$$\begin{array}{rl}
(\underbrace{A_x, A_y, A_z}_{\text{translational DOFs}}, \underbrace{A_\alpha, A_\beta, A_\gamma}_{\text{rotational DOFs}}) & \text{state vector of } A \\
(A_x, A_y, A_z, A_\alpha, A_\beta, A_\gamma) & \text{parametrization of state of } A \\
(B_x, B_y, B_z, B_\alpha, B_\beta, B_\gamma) & \text{state vector of } B \\
(B_{angle}) & \text{parametrization of state of } B \\
 & \text{(in generalized coordinates)}
\end{array}$$

[1]There is an analogy with the traditional approach used by many animation studios in the 1950s, where senior animators drawn the key frames and less experienced animators drawn the inbetweens.

[2]Figure segments are skeleton segments with a geometry and possibly also volume and mass.

[3]More generally, any coordinates that parametrize a subset of segment state space are called DOFs. That way, if appropriate parametrization is chosen, we can define a DOF that controls both rotation and displacement.

[4]Physically consistent states that obey constraints imposed by figure's joints. When the segment is unconstrained, the set spans the whole $\mathbf{R}^6$ space.

$((A_x, A_y, A_z, A_\alpha, A_\beta, A_\gamma), (B_{angle}))$	parametrization of figure (in generalized coordinates)

Since B is attached to A, its state vector (B_x, B_y, B_z, B_α, B_β, B_γ) is a function of (A_x, A_y, A_z, A_α, A_β, A_γ, B_{angle}), where B_{angle} is the angle about the axis attached to A the segment B is rotated about with respect to the initial posture.

There is a fundamental difference between *maximal coordinate systems* and *reduced coordinate systems*. While maximal coordinate systems operate in terms of maximal coordinates and have to ensure that the figure remains in a valid state by considering the constraints (the set of valid states is defined implicitly), reduced coordinate systems operate on valid states only in terms of unconstrained generalized coordinates (the set of valid states is parametrized explicitly by generalized coordinates, granted the parametrization is available). Kinematics-only systems always use the reduced coordinates approach.

In the case of *forward kinematics*, animator specifies the figure state coordinates at given key-frames (usually a position and an orientation of the root segment and angles/displacements at joints, parametrization by generalized coordinates) and the animation system computes figure states, thus generating figure (skeleton) posture at each frame. Often it is more natural to specify desired positions of "end effectors" (zero DOF joints, "joints" with no children segments attached to them) and let the animation system compute appropriate state vectors, considering additional constraints such as joint angle limits, etc. Such an approach is called *backward (inverse) kinematics*. Unfortunately, present backward kinematics systems are often restricted to operate on a subset of the figure only (for example an arm)[5] and the rest of the figure has to be animated using forward kinematics techniques.

Dynamics-based approaches treat figure segments as volumes with certain mass properties that respond to forces and torques exerted at them[6]. In the case of *forward dynamics*, animator specifies forces and torques that act on figure segments at given times (virtually at the key-frames only) and the animation system produces a motion that is the result of the corresponding physical simulation. Clearly, such a system always produces physically correct motion, but it might be rather tedious to come up with appropriate forces to pose the figure as requested. *Backward (inverse) dynamics* systems are like forward dynamics systems but are also able to compute appropriate forces and torques according to constraints specified by animator (desired joint angles, joint angle limits, desired positions of end effectors, etc.)[7], yet able to produce physically correct motion and thus combining the good properties of both the dynamics and kinematics worlds.

Although both kinematic and dynamic approaches are able to produce plausible results, the quality of the generated motion mostly depends on the ability of the animator to supply appropriate inputs (say desired joint angles and/or positions of end effectors). While in kinematics-based approaches animator is responsible for every aspect of the generated animation, in dynamics-based approaches animator might want to control selected joints only (joints whose control is crucial for certain figure behaviour or to distinguish that behavior from others) and let the other joints move freely. For example, to animate a figure of a human who is leaning forward, it might be sufficient to explicitly animate the figure's spine only.

Recently *motion capture* has become a number one technology for animation of human-like figures. An actor, equipped with motion capture sensors, performs a desired motion that is to be captured by the motion capture device. Data from the motion capture sensors are analysed and mapped to skeletal motion (usually in the form of the trajectory of the root joint and joint angles), ready to be replayed using a forward kinematics technique. Although the motion capture

[5]To be more precise, they can operate on selected subsets of the figure, however each subset is treated as an isolated figure, not aware of the rest of the body.

[6]Segment states are extended by dynamics-specific quantities.

[7]Simulator described in this thesis can be treated as a backward dynamics system according to this classification.

is capable of producing realistic looking motions with a minimum effort it also has its hitches. It is relatively hard to retarget the captured motion to a different actor (say with a different body proportions than were those of the real actor who performed the motion, to say nothing of "capturing" a motion of non-existent figures or the motion of animals) or edit or even combine the captured motions. Furthermore, such a basic requirement as altering the motion of the arm to make it point in the specified direction might be a problem. Later on, it will be demonstrated that some of these problems can be solved using a backward dynamics system. Introductory information on the motion capture technology can be found in [19].

The problem of the inability to edit or combine motion capture segments to produce a composite animation of a desired expression can be solved by the use of *motion libraries*, which is an off-line pure kinematic approach. The library consists of a vast number of short motion sequences whose frames are annotated (described) by what the actor is doing and what their posture is (for example, right foot is now on the floor and left arm pointing up). Frames from all sequences are organized into a directed graph of frames (motion graph), where two frames are connected by an edge if the transition from the first frame to the second frame is possible[8], edges are annotated by the "costs" of transitions. Animator specifies a sequence of $(time, desiredannotation)$ pairs and the animation systems finds a frame sequence that approximates the desired motion expression best, that is, it finds a path in the motion graph with the minimum total cost of frame transitions that honors the specified motion constraints. It is an off-line optimization process.

Other researchers attempt to *generate motion* of certain type from parametric descriptions (where parameters might be the figure's sex, height, weight, age, motion speed, plausibility of the motion, ...) or simulate *human behavior* in particular situations, with the distant goal of creating an autonomous ultimate *digital actor*. Although these problems are very hard to solve, several classes of human motion have been described with a success, namely walking and running. These approaches often use algorithms borrowed from robotics and artificial intelligence.

1.2 Goals

This thesis focuses at the study of constrained rigid body dynamics whose equations of motion are formulated in terms of *maximal coordinates approach*. Constraints are enforced by introducing *constraint forces* into the system, rather than by removing the constrained DOFs.

As we will see later, such an approach will allow us to combine arbitrary set of constraints interactively, without the need for searching for the new parametrization of the set of valid states and reformulating equations of motion in terms of the new generalized coordinates. This is remarkably important for interactive computer graphics applications, such as games. No less important is the fact that maximal coordinates approach encourages a modular design of the simulation system, [9]. The primary goal is to describe such a system.

The other goal is the implementation of the simulation system utilizing the best (most recent) algorithms published by authorities in this area of computer graphics, allowing it to run at interactive rates. An extra attention is paid to the design and the implementation of the simulator, the abstraction of motion equation solver (differential equation solver, "integrator") and constraint solver. Ideally, both the performance capabilities of the simulator, its design and the quality of implementation should make it competitive with other simulators at the market.

The implementation is mostly based on the work and research of David Baraff. The theory behind (dynamics model) as well as the implementation ideas draw from his interesting papers,

[8]Either both the frames belong to the same motion sequence and the second frame immediately follows the first frame, or the two frames belong to different motion sequences but are "similar" to each other.

tutorials and dissertation thesis, [7], [9], [10], [6]. Several implementation tricks were adopted from the ODE library by Russell Smith, [22].

The implementation of a dynamics-based figure library suitable for interactive animation of human-like figures is our another goal. The library should present the capabilities of dynamics-based simulators to solve some of the problems associated with editing, retargetting or combining motion capture (or any other kinematics-based) data. Finally, we will show how raw optical motion capture data can be mapped to skeletal motion using the figure library.

Chapter 2

Differential Equations

This chapter will guide the reader through the problems of *ordinary differential equations* (ODEs) as required by their application in the practice. We will be mostly interested in the numerical solving of the equations and the theory will be avoided whenever possible. The derivations and discussion will be based on [10] and [14].

The area of ODEs is widely exploited in the practice, many real world problems can be modelled by the use of ODEs with a success. It is no surprise that the equations of motions, pursued by dynamics, are ODEs too. Therefore we find the problems of ODEs essential for understanding the dynamics theory and start off this way.

2.1 Definitions

Differential equation is any equation that involves an unknown function (to be solved for) and its derivative(s), representing a relation between the unknown function and the derivatives. When we talk about solving the equation we refer to the process that finds a function that satisfies the desired relation. We are particularly interested in a certain class of differential equations, called the *ordinary differential equations* and a special class of problems called the *initial value problems*.

Convention 1 *Vectors we will work with will be denoted by $\vec{\ }$ symbols, such as $\vec{a}, \vec{b}$, and will be considered as columns ($m \times 1$ matrices) unless stated otherwise. This will be also true when the vectors are specified by their components.*

For example $\vec{a} = (a_1, a_2, a_3)$ will refer to a column vector filled with a_1, a_2, a_3 values. One might as well write $\vec{a} = (a_1, a_2, a_3)^T$ to stress that $\vec{a}$ is really a column vector, but we will not do that, according to our convention.

Definition 1 *Ordinary differential equation is any equation of the form $\vec{y}'(t) = \vec{f}(t, \vec{y}(t))$, where $t \in \mathbf{R}$, $\vec{y} : \mathbf{R} \to \mathbf{R}^n$ is the unknown function of t, $\vec{f} : \mathbf{R}^{n+1} \to \mathbf{R}^n$ is a known function of t and $\vec{y}(t)$, $\vec{f}(t, \vec{y}(t)) = \vec{f}(t, y_1(t), \ldots, y_n(t))$ and $\vec{y}'(t) = \frac{\partial}{\partial t}\vec{y}(t) = \frac{\partial}{\partial t}(y_1(t), \ldots, y_n(t)) = (\frac{\partial}{\partial t}y_1(t), \ldots, \frac{\partial}{\partial t}y_n(t))$.*

The relation defined by an ODE equation can be interpreted as follows. Imagine that t represents time and $\vec{f}(\bullet, \vec{y})$ defines vectors of an n-dimensional vector field at time t ($\vec{f}(t, \vec{y})$ is the vector (value) of the vector field at point $\vec{y}$ of the field at time t[1]). The ODE then defines

[1]Since $\vec{f}$ is a function of time, vectors of the vector field may change over time.

a relation between a position $\vec{y}(t)$ of an imaginary particle in the vector field at time t and its instant velocity $\vec{y}'(t)$. It says that "if the particle is at position $\vec{y}(t)$ then its velocity $\vec{y}'(t)$ must equal $\vec{f}(t, \vec{y}(t))$". This is easy to visualize in the case of $n = 2$, [10].

The equation of the form $\vec{y}' = \vec{f}(t, \vec{y})$ does not determine the solution function $\vec{y}$ uniquely. Treating the function $\vec{y}(t)$ as a parametrization of the trajectory of an imaginary particle that "follows" vectors in the vector field, it is easily seen that the actual trajectory the particle goes along depends on the point where the particle is initially dropped to the vector field. To pick a particular trajectory one has to specify what the initial position of the particle should be, that is to specify $\vec{y}(t_0) = \vec{y}_0$.

Definition 2 *Ordinary differential equation $\vec{y}' = \vec{f}(t, \vec{y})$ with an additional condition $\vec{y}(t_0) = \vec{y}_0$, $t \geq t_0$ is called an initial value problem. The variable t is often interpreted as time, t_0 as an initial time, $\vec{y}(t)$ as a state of a system at time t and $\vec{y}_0$ as an initial state.*

The solution of the initial value problem is an integral curve that starts at $\vec{y}_0$ and is parametrized by $\vec{y}(t)$, where $t \geq t_0$. Vector $\vec{y}_0$ can be seen as an *initial state* of a system whose evolution in time is given by the ODE. Knowing the initial state $\vec{y}_0$, the ODE determines what the *system state* at time t is — this is the value of $\vec{y}(t)$. We thus have got two interpretations of ODE equations and $\vec{y}(t)$, either as a position of a particle in a vector field or a state of a system. It is usually a good idea to recall this when in doubt about the way certain concepts work.

If it is the first derivative of the unknown function $\vec{y}$ that appears in the ODE equation (that was the case in our definition), then it is said that it is a *first order* ODE. Equations with higher derivatives, however, occur frequently in the practice, but can be transformed into equivalent first order ODEs.

The trick is to make higher order derivatives part of the extended system state (position in the vector field). Consider an equation $x''' = const$ which contains a third order derivative. To make it a first order ODE, we will define $a = x$, $b = x'$, $c = x''$ ((x, x', x'') will be the new system state, a will be the solution to the original equation, b and c will be computed by-products), which will turn the original equation to a set of coupled first order equations:

$$(a', b', c') = (b, c, const)$$

This example is correct regardless of the fact whether x is a scalar or a vector. However in the case of vectors, we would actually get a set of vector equations which would have to be expanded by their components in order to match the ODE definition. The following paragraph elaborates more on this and it is the reason why it might be somewhat tedious to read.

Definition 3 *Ordinary differential equation of order m is any equation of the form $\vec{u}^{(m)} = \vec{f}(t, \vec{u}, \vec{u}', \ldots, \vec{u}^{(m-1)})$, where $t \in \mathbf{R}$, $\vec{u} : \mathbf{R} \to \mathbf{R}^n$ is the unknown function of t, $\vec{f} : \mathbf{R}^{n \cdot m+1} \to \mathbf{R}^n$ is a known function of t and $\vec{u}(t), \vec{u}'(t), \ldots, \vec{u}^{(m-1)}(t)$ and $\vec{f}(t, \vec{u}(t), \vec{u}'(t), \ldots, \vec{u}^{(m-1)}(t)) = \vec{f}(t, u_1(t), \ldots, u_n(t), u_1'(t), \ldots, u_n'(t), \ldots, u_1^{(m-1)}(t), \ldots, u_n^{(m-1)}(t))$. If a set of additional conditions $\vec{u}(t_0) = \vec{u}_0$, $\vec{u}'(t_0) = \vec{u}_0'$, $\ldots$, $\vec{u}^{(m-1)}(t_0) = \vec{u}_0^{(m-1)}$, $t \geq t_0$ is given, then this is an initial value problem of order m.*

Let us define a set of $n \cdot m$ unknowns $y_{11} = u_1, \ldots, y_{1n} = u_n$, $y_{21} = u_1', \ldots, y_{2n} = u_n', \ldots, y_{m1} = u_1^{(m-1)}, \ldots, y_{mn} = u_n^{(m-1)}$ and a vector $y = (y_{11}, \ldots, y_{mn}) \in \mathbf{R}^{n \cdot m}$. Then the following equation $(y_{11}', \ldots, y_{1n}', \ldots, y_{m-1,1}', \ldots, y_{m-1,n}', y_{m1}', \ldots, y_{mn}') = (y_{21}, \ldots, y_{2n}, \ldots, y_{m1}, \ldots, y_{mn}, f_1(t, \vec{y}), \ldots, f_n(t, \vec{y}))$ is the corresponding first order ordinary differential equation and the initial value condition of order m transforms to the first order condition $\vec{y}(t_0) = \vec{y}_0 = (u_1(t_0), \ldots, u_n(t_0), u_1'(t_0), \ldots, u_n'(t_0), \ldots, u_1^{(m-1)}(t_0), \ldots, u_n^{(m-1)}(t_0))$, $t \geq t_0$.

If $\vec{u}(t)$ is a solution of the system of order m, then $\vec{y}(t) = (u_1(t), \ldots, u_n(t), u_1'(t), \ldots, u_n'(t), \ldots, u_1^{(m-1)}(t), \ldots, u_n^{(m-1)})$ is a solution of the corresponding first order system. Contrary, if $\vec{y}(t)$ is a solution of the first order system, then $\vec{u}(t) = (y_{11}, \ldots, y_{1n})$ is a solution of the original system of order m.

In the rest of the chapter we will study first order systems only.

2.2 Numerical Solvers

Analytical solution of an ODE is a closed-form formula that can be evaluated at any point t. In contrast, numerical solution is a set of approximate values evaluated at a discrete set of points.

We restrict ourselves to first order initial value problems and the corresponding numerical solution methods. Traditionally, $\vec{y}(t)$ will be called the *system state* at time t and $\vec{y}_0 = \vec{y}(t_0)$ *initial state.*

The purpose of the numerical solver is to estimate $\vec{y}(t_{k+1})$ ($k+1 \geq h$, where $h \geq 1$ is a constant specific to the solver[2], $k \in \mathbf{N}_0$), given t_{k+1}, a sequence of h previous states together with the times when they were estimated, $\{(t_i, \vec{y}(t_i))\}_{i=k-h+1}^{k}$, and function $\vec{f}$ that can be evaluated for arbitrary t and $\vec{y}$. No other special knowledge of the function $\vec{f}$ is required by the solver.

To solve the equation, the solver proceeds in discrete time steps. At each step t_{k+1} one or more evaluations of $\vec{f}$ for a given $\vec{y}$ and t are performed (vector field is sampled) and the obtained information together with the information from previous h states utilized to approximate the system state change from the previous state $\vec{y}_k$ to $\vec{y}_{k+1}$, yielding an estimate of the new state at t_{k+1}. This process can be seen as a simulation of the evolution of the system from a specified initial state, as controlled by the ODE.

2.2.1 Euler Solver

Euler solver is the simplest numerical solver ever, often hard-coded to low-end numerical software. It uses a history of length 1, therefore the overall effect at each step is the same as if the solver was just "started" from the previous state, as if the previous state was the initial state. We can thus assume that $k = 0$.

Let us return to the analogy with the tracing of the trajectory of a particle. If an imaginary particle is dropped to $\vec{y}_0$, ODE will dictate what its velocity should be, which equals $\vec{f}(t_0, \vec{y}_0)$. Knowing the initial position $\vec{y}_0$, its desired velocity $\vec{f}(t_0, \vec{y}_0)$, a step size $h = t_1 - t_0$ and assuming that the desired velocity is constant within the $< t_0, t_1)$ range, the new position $\vec{y}_1$ at $t_1 = t_0 + h$ can be computed as follows

$$\vec{y}(t_0 + h) = \vec{y}_0 + \vec{y}'(t_0) \cdot h = \vec{y}_0 + \vec{f}(t_0, \vec{y}_0) \cdot h,$$

which is the formula used by the Euler solver when taking a step.

Approximation Error

While stepping from one discrete point to the next, due to various approximations made by the algorithm (here, it was assumed that $\vec{y}_0^{(k>1)}$ was zero), we incur some error and do not follow the integral curve precisely. In fact, our particle will drift and, at the end of the step, will end up on another curve, different from the curve we started from (at the beginning of the step).

[2] If $h > 1$ then the solution method is a multi-step method. Otherwise it is a single-step method.

This error, gained while taking a step, is called the *local error*. Contrary, the difference between the true (expected) solution (curve due to the position of the particle at the very beginning of the simulation) and the estimated solution is called the *global error*, [14]. Global error is always gained indirectly, by the accumulation of local error (it need not equal the sum of the local errors, though; an imprecision due to the used floating-point system is ignored). The goal is to keep global error small, but only local error can be controlled directly.

Local error at step k is the difference between an approximate solution $\vec{y}_{k+1}$ at t_{k+1} and a true solution $\vec{y}(t_{k+1})$ with $\vec{y}(t_k) = \vec{y}_k$ initial condition.

Global error at step k is the difference between an approximate solution $\vec{y}_{k+1}$ at t_{k+1} and a true solution $\vec{y}(t_{k+1})$ with $\vec{y}(t_0) = \vec{y}_0$ initial condition.

The process of solving an ODE is often called the integration of the ODE and the local error called the integration error per step.

Consider a 2D vector field $\vec{f}$, whose integral curves are concentric circles with their centers at the origin. An imaginary particle put to an arbitrary circle is supposed to orbit on that circle forever. Instead, with each Euler step, the particle will move in the direction tangent to the current circle ending up at a circle with a larger radius and local error is gained. As a result, the particle will follow a polygonal path that approximates an outward spiral (continuously drifting away from the origin) instead of the circle.

Further on we will discuss more accurate methods, capable of producing less local error with respect to a given step size. Such methods perform more than two system state derivative evaluations at each step, but the extra cost (cost per step) is compensated by the larger step size they allow, yet preserving the same local error. We will almost entirely restrict ourselves to methods that utilize histories of length 1, because such methods are *self-starting* and do not require a history to be specified (except for the initial state) in order to start.

2.2.2 Higher Order Solvers

Assuming that the unknown function $\vec{y}$ to be solved for is differentiable up to order $n+1$ in the range $< t_0, t_0 + h >$, we can approximate its value at the end of the step by its value and a set of its derivatives at the beginning of the step, in the form of *Taylor polynomial* of order n, so that:

$$\vec{y}(t_0 + h) = \vec{y}(t_0) + \vec{y}'(t_0) \cdot h + \vec{y}''(t_0) \cdot \frac{h^2}{2!} + \cdots + \vec{y}^{(n)}(t_0) \cdot \frac{h^n}{n!} + O(h^{n+1}), \tag{2.1}$$

where $O(h^{n+1})$ is the approximation error term of order[3] h^{n+1} (difference between the untruncated *Taylor series*[4] and the polynomial).

Taylor Series Based Solvers

Equation (2.1) serves as a basis for a specific class of solvers called *Taylor series based solvers*. In particular, if the equation of order n is used to update the states, one would get a solver of order n with a guaranteed local error (accuracy) of order $O(h^{n+1})$. For example by letting $n = 1$ we get the update formula used by Euler solver. It follows from this that Euler solver is a first order method with $O(h^2)$ accuracy and produces no local error only if $\vec{y}^{(k>1)}(t_0)$ were all zeroes — in other words $\vec{y}(t)$ had to be a linear function of t for $t \in < t_0, t_0 + h)$.

[3]Its constant is specific to $\vec{y}(t_0)$.

[4]Taylor polynomial where $n \to \infty$.

But where do we get the higher derivatives of $\vec{y}$ if we have a direct access to the first derivative only?[5] If $\vec{y}'(t) = \vec{f}(\vec{y}(t))$ then the derivatives of $\vec{y}$ can be expressed in terms of its lower order derivatives and the derivatives of $\vec{f}$. Although we can not evaluate derivatives of $\vec{f}$ directly (which would be impractical anyway), we can approximate them by evaluating $\vec{f}$ multiple times in the $< t_0, t_0 + h)$ range, that is sampling the vector field. By repeating the process, all derivatives can be eliminated and finally approximated in terms of $\vec{f}$ and $\vec{y}$. To summarize, the problem of evaluation of $\vec{y}$ derivatives can be reduced to the problem of evaluation of $\vec{f}$ derivatives and those derivatives can be approximated.

Let us show how a second order derivative of $\vec{y}$ can be computed from $\vec{f}$ and $\vec{f}'$, which will be approximated from $\vec{f}$ by utilizing its first order Taylor expansion at $\vec{y}(t)$ (differentiability of $\vec{f}$ is assumed at all points):

$$\vec{f}(\vec{y}(t) + \Delta\vec{y}) = \vec{f}(\vec{y}(t)) + \vec{f}'(\vec{y}(t)) \cdot \Delta\vec{y} + O(\|\Delta\vec{y}\|^2)$$

$$\vec{f}'(\vec{y}(t)) \approx \left[\vec{f}(\vec{y}(t) + \Delta\vec{y}) - \vec{f}(\vec{y}(t))\right] \cdot \frac{1}{\Delta\vec{y}}$$

Now we will use the approximation of $\vec{f}'$ to approximate $\vec{y}''$ in terms of $\vec{f}$ and $\vec{y}$:

$$\vec{y}'(t) = \vec{f}(\vec{y}(t)) \qquad \vec{y}''(t) = \vec{f}'(\vec{y}(t)) \cdot \vec{y}'(t)$$

$$\vec{y}''(t) \approx \left[\vec{f}(\vec{y}(t) + \Delta\vec{y}) - \vec{f}(\vec{y}(t))\right] \cdot \frac{\vec{f}(\vec{y}(t))}{\Delta\vec{y}} \tag{2.2}$$

If we let $\Delta\vec{y} = \vec{f}(\vec{y}(t_0)) \cdot \frac{h}{2}$, substitute to (2.2) and $\vec{y}''$ to (2.1) of order 2, we can predict $\vec{y}(t_0 + h)$ as

$$\vec{y}(t_0 + h) = \vec{y}(t_0) + \vec{y}'(t_0) \cdot h + \vec{y}''(t_0) \cdot \frac{h^2}{2} =$$

$$\vec{y}_0 + \vec{f}(\vec{y}_0) \cdot h + \left[\vec{f}(\vec{y}_0 + \vec{f}(\vec{y}_0) \cdot \frac{h}{2}) - \vec{f}(\vec{y}_0)\right] \cdot \frac{\vec{f}(\vec{y}_0)}{\vec{f}(\vec{y}_0) \cdot \frac{h}{2}} \cdot \frac{h^2}{2}$$

After rearranging, we will obtain an update formula for second order *midpoint* method[6]:

$$\vec{y}(t_0 + h) = \vec{y}(t_0) + \vec{f}(t_0 + \frac{h}{2}, \vec{y}(t_0) + \vec{f}(t_0, \vec{y}(t_0)) \cdot \frac{h}{2}) \cdot h$$

Midpoint solver first takes an Euler step of size $h/2$ to an *intermediate state*[7] $\vec{y}_0 + \vec{f}(\vec{y}_0) \cdot h/2$ called the midpoint, system state derivative is evaluated at the midpoint and an Euler step of size h from $\vec{y}_0$ in the direction of the midpoint's derivative is taken.

Runge-Kutta Methods

There is another class of self-starting higher order methods, generally called *Runge-Kutta* methods. These methods simulate the effect of higher order derivatives by sampling the vector field

[5]To simplify further discussion, we will assume that $\vec{f}$ depends on t only indirectly, through $\vec{y}(t)$, and will treat $\vec{f}$ as a function of a single variable $\vec{y}(t)$.

[6]From now on we will again treat $\vec{f}$ as a function of t.

[7]States $\vec{y}$, where the state derivative $\vec{y}'$ is evaluated just for the purposes of predicting $\vec{y}(t_0 + h)$, are generally called *intermediate states*. The other states are called *regular states*.

over $< t_0, t_0 + h)$ range, [14]. Probably the most popular Runge-Kutta method is a Runge-Kutta of order 4 (with a local error of $O(h^5)$):

$$\vec{k}_1 = \vec{f}(t_0, \vec{y}_0) \cdot h$$

$$\vec{k}_2 = \vec{f}(t_0 + \frac{h}{2}, \vec{y}_0 + \frac{\vec{k}_1}{2}) \cdot h$$

$$\vec{k}_3 = \vec{f}(t_0 + \frac{h}{2}, \vec{y}_0 + \frac{\vec{k}_2}{2}) \cdot h$$

$$\vec{k}_4 = \vec{f}(t_0 + h, \vec{y}_0 + \vec{k}_3) \cdot h$$

$$\vec{y}(t_0 + h) = \vec{y}_0 + \frac{1}{6}\vec{k}_1 + \frac{1}{3}\vec{k}_2 + \frac{1}{3}\vec{k}_3 + \frac{1}{6}\vec{k}_4$$

2.2.3 Step Size Control

Choosing an appropriate step size h is often a problem. Naturally, we would like to take as large steps as possible to minimize computational cost, but have to consider stability and accuracy. If we chose a fixed step, we could proceed only as fast as the "worst" sections of $\vec{y}$ allowed. The remedy seems to be to adapt h to the current section of $\vec{y}$ that is being evaluated at the moment.

Adaptive Time Stepping

Given a local error estimate $e(h)$ as a function of step size h and the maximum local error per step E we can tolerate, the idea of *adaptive time stepping* is to adjust h so that the desired accuracy is attained. If we know that the order of the underlying solver is n, then the local error is of order $O(h^{n+1})$ and thus $e(h) \leq k \cdot h^{n+1}$ for an appropriate constant k specific to the current step. We would like to find a step size shrink factor c so that $e(h' = h \cdot c) \leq E$ (h' would be the new step size). We will assume that $e(h')$ can grow as big as $k \cdot (h')^{n+1}$, thus $e(h') \leq k \cdot (h')^{n+1} \leq k \cdot h^{n+1} \cdot c^{n+1} = e(h) \cdot c^{n+1} \leq E$ and so $c \leq \left(\frac{E}{e(h)}\right)^{\frac{1}{n+1}}$ and conclude that we can take a step as large as

$$h' = h \cdot \left(\frac{E}{e(h)}\right)^{\frac{1}{n+1}}. \tag{2.3}$$

Local Error Estimation

There are multiple ways how local error can be *estimated*. These methods are often based on the evaluation of differences between results obtained using methods of different orders or step sizes.

Remember that local error at step k is the difference between an approximate solution $\vec{y}_{k+1}$ at t_{k+1} and a true (unknown) solution $\vec{y}(t_k + h_k)$ with $\vec{y}(t_k) = \vec{y}_k$ as the initial condition. Knowing $\vec{y}_{k+1}$, the local error gained during step k can be estimated by rolling back to $\vec{y}_k$ and using a different (and more accurate) method to advance to t_{k+1} — a higher order method or even the same method that takes multiple shorter stops would do. That way, we will be produced with two estimates of $\vec{y}(t_{k+1})$ and their difference can be used to estimate $e(h_k)$ at t_k.

In particular we might step from t_k to t_{k+1} and let $\vec{a} = \vec{y}_{k+1}$, rollback to t_k, step to $t_k + h_k/2$, step from this intermediate state to $t_k + h_k$, ending up at state $\vec{b}$. Having two estimates of the system state at t_{k+1} we let $e(h_k) = \|\vec{a} - \vec{b}\|$, update the current step size according to (2.3) and let $\vec{y}_{k+1} = \vec{b}$, because $\vec{b}$ is a better approximation of $\vec{y}(t_{k+1})$ than $\vec{a}$.

Alternatively, one can grasp that the local error more or less corresponds to the error due to truncation of the Taylor series in the Taylor polynomial of order n (2.1). It is well known that the truncation error can be expressed as $\vec{y}^{(n+1)}(\xi_k) \cdot \frac{h_k^{n+1}}{(n+1)!}$ for an appropriate $\xi_k \in < t_k, t_{k+1} = t_k + h_k >$. If we assume that $\xi_k = t_k$, we can approximate the local error at step k as $e(h_k) = \|\vec{y}^{(n+1)}(t_k)\| \cdot \frac{h_k^{n+1}}{(n+1)!}$, where $\vec{y}^{(n+1)}$ has to be approximated in terms of $\vec{f}$ and $\vec{y}$ or in terms of the state history (for example $\vec{y}'' \approx \frac{\vec{y}_k - \vec{y}_{k-1}}{t_k - t_{k-1}}$).

2.3 Abstraction

In this section we will discuss an abstraction of ODE equations with respect to solvers and present a common interface that will allow applications to use the same solvers to solve ODEs representing very different kinds of problems and, through the use of the common interface, allow to replace solvers without affecting the rest of the system. As we will see later, rigid body dynamics is governed by ODEs and this abstraction will the system to prefer accuracy over computational cost upon user's request, for example.

Although many programmers are aware of the ODE theory, they often design their simulators in such a way, that the simulation state (which corresponds to the state of the ODE) is kept in objects managed by the application and that it is the application who implements the simulation loop and takes the steps (there is no abstraction of the ODE equation). Used numerical solver method is hard-coded into the application's simulation loop, which has to be redesigned whenever the used solver needs be replaced by a different solver. Contrary, we will present an abstraction of ODEs and the interface to ODE solvers which manage the simulation state, implement the simulation loop and control the step size.

Assuming that the simulator will operate in terms of objects, whose states are to be controlled by an ODE, and noting that the *generic ODE solver* can operate merely in terms of vectors $\vec{y}$ and $\vec{f}$, treating $\vec{f}$ as a black-box function that can be evaluated for any given t and $\vec{y}(t)$, the interface will have to provide a way to map the object states to the components of $\vec{y}$ and back. This is the key idea of the abstraction. ODE state will be kept by the ODE solver and will be "installed" into application objects whenever the system state derivative is to be evaluated or the new state presented. To support this approach, simulator will have to provide the implementation of the following functions, invoked by the ODE solver:

- Retrieve the size of the vector $\vec{y}$ (**get_state_dimension**).

  ODE solver has to know what the dimension of system states and state derivatives is.

- Copy current object states to the state vector $\vec{y}$ (**get_state**), load object states from the state vector $\vec{y}$ (**set_state**).

  ODE solver has to retrieve the initial system state and install the new state at the end of the step. Moreover it might request to install an intermediate state in order to evaluate $\vec{f}$ at that point.

- Evaluate $\vec{f}$ at the current state (**evaluate_derivative**).

From the simulator's perspective, the ODE solver has to provide the following functions, called by the simulator:

- Restart the solver from the current state (**restart**).

Simulator has to tell the ODE solver what the new initial state is and that the state history should be discarded.

- Advance the system to a specified time (**advance**).

  Simulator has to tell the ODE solver when to advance and to what time.

- Set solver parameters (**set_parameters**).

  Simulator has to tell the ODE solver what the desired accuracy is. Parameters like maximum local error, minimum/maximum step size might be specified.

For example an unoptimized fixed-step Euler solver, as defined by (2.2.1), can be implemented in this way (implementation of **restart** and **advance** in pseudo-code is given):

```
restart( time )
    {
    n =get_state_dimension();
    resize_vector( current_state, n );
    resize_vector( current_state_derivative, n );
    get_state( current_state );
    evaluate_derivative( time, current_state_derivative );
    current_time =time;
    }

advance( time )
    {
    while( time > current_time )
        {
        step_size =min( time - current_time, maximum_step_size );
        scale_vector( current_state_derivative, step_size );
        add_vector( current_state, current_state_derivative );
        current_time =current_time + step_size;
        set_state( current_state );
        evaluate_derivative( current_time, current_state_derivative );
        }
    }
```

Chapter 3

Particle Dynamics

Particle systems dynamics is a part of mechanics that studies the motion of particles, bodies that can be approximated by objects characterized by their position, velocity and mass (but with no spatial extents), in response to forces that act on them. Collections of particles are called the particle systems. Although being simple, particle systems can be used to model many interesting non-rigid structures such as soft bodies, cloths, fluids, etc.

The most of the particle systems theory generalize nicely for rigid body dynamics and so we will treat the theory as a starter for the latter. The concepts and derivations are based on [10] and [13].

3.1 Concepts

We will present some definitions first describing the structure of particles from the point of kinematics. We will intuitively treat particles as volume-less points that move in a reference space ("world"). Reference space together with a coordinate system will be simply called the system.

Definition 4 Particle *is a structure with mass* $m > 0$ *and position* $\vec{r}(t)$*, where* $\vec{r}(t)$ *is a vector pointing from the origin of the reference space towards the position of the particle. A curve in the reference space parametrized by* $\vec{r}(t)$*, where* t *is interpreted as time, is called the* trajectory of the particle *(in the reference space).*

By taking the time derivative of the position $\vec{r}(t)$, we get the particle *velocity* $\vec{v}(t)$, expressed as a function of t. Differentiating the velocity, we get the particle *acceleration* $\vec{a}(t)$, expressed as a function of t:

$$\vec{v}(t) = \lim_{\Delta t \to 0} \frac{\vec{r}(t + \Delta t) - \vec{r}(t)}{\Delta t} = \frac{\partial}{\partial t}\vec{r}(t) = \dot{\vec{r}}(t)$$

$$\vec{a}(t) = \lim_{\Delta t \to 0} \frac{\vec{v}(t + \Delta t) - \vec{v}(t)}{\Delta t} = \frac{\partial}{\partial t}\vec{v}(t) = \frac{\partial^2 \vec{r}(t)}{\partial t^2} = \ddot{\vec{r}}(t) = \dot{\vec{v}}(t).$$

It is said that the particle is *moving uniformly* if its velocity $\vec{v}(t)$ is constant, that is its acceleration is zero ($\vec{a}(t) = \vec{0}$). Note that particles with zero velocity (at rest) are considered to be moving uniformly.

Force $\vec{F}(t)$ is a vector physical quantity that characterizes interaction among particles. It is described by the particle it acts upon, a magnitude and a direction[1]. The magnitude and the

[1]And the point of work in the case of bodies with spatial extents.

direction capture the effects of the interaction of this particle with other particles in the system. Newton's laws stipulate how forces exerted on particles affect their motion.

We will say that the particle is isolated if there is no force acting on it.

Definition 5 (Newton's First Law) *Inertial systems exists. Isolated particles move uniformly in an inertial system.*

The first law says that if there is no force acting on a particle then it must move uniformly in an inertial system. If it does not move uniformly (accelerates or decelerates) then there must be a force acting on it or the system is not inertial (the "missing" force, that would have to be exerted on the particle in order to make it exhibit the observed behavior in an inertial system, is called the *inertial force*). We will restrict ourselves to inertial systems only.

Let us define the *linear momentum* $\vec{P}(t)$ of a particle at time t as

$$\vec{P}(t) = m \cdot \vec{v}(t). \tag{3.1}$$

The momentum characterizes the kinetic state of the particle and the first law can be then interpreted in that way that the linear momentum of a particle in an inertial system can only change in response to a force acting on that particle. The second Newton's law, defined below, elaborates more on this and defines what the actual relation between an exerted force and the change of the linear momentum is.

Definition 6 (Newton's Second Law) *If $\vec{F}(t)$ is a force acting on a particle with the linear momentum $\vec{P}(t)$ then*

$$\vec{F}(t) = \frac{\partial}{\partial t}\vec{P}(t) = \frac{\partial m \cdot \vec{v}(t)}{\partial t} = m \cdot \frac{\partial}{\partial t}\vec{v}(t) = m \cdot \vec{a}(t)$$

and the particle will gain an acceleration $\vec{a}(t) = \vec{F}(t)/m$.

If we have a set of forces $\vec{F}_1(t), \ldots, \vec{F}_n(t)$ acting on the same particle then the net effect equals the effect due to a single force $\vec{F}_{total}(t) = \vec{F}_1(t) + \ldots + \vec{F}_n(t)$ (*superposition principle*).

Definition 7 (Newton's Third Law) *If force $\vec{F}_{AB}(t)$ due to particle A is exerted on particle B by particle A (action force), then $\vec{F}_{BA}(t) = -\vec{F}_{AB}(t)$ is exerted on A by particle B (reaction force). Both the forces lie on the same line of action, and begin/stop acting at the same time.*

Particle system is a collection of particles. If the system consists of N particles with masses $m_1, \ldots, m_N$ and positions $\vec{r}_1(t), \ldots, \vec{r}_N(t)$ then the total mass M and the center of mass $\vec{r}_{cm}$ of the particle system are defined as

$$M = \sum_{i=1}^{N} m_i, \qquad \vec{r}_{cm}(t) = \frac{\sum_{i=1}^{N} m_i \cdot \vec{r}_i(t)}{M}.$$

The total mass is defined simply as a sum of the particle masses and the center of mass as a weighted sum (convex combination) of the particle positions. The sum weights (coefficients of the convex combination) are proportional to the particle masses relative to the total mass. If all particles had the same mass then the center of mass of the particle system would correspond to the geometric center (average) of the particle system.

Let $\vec{F}_i^j, 1 \leq j \leq n$ be all forces acting on particle i in the particle system and $\vec{F}_i^{total}(t)$ be the sum of those forces. This sum can be decomposed to the sum of *internal forces* (denoted $\vec{F}_i^{int}(t)$, the sum of all forces exerted on particle i by *other particles* $j \neq i$ in the particle system) and the sum of *external forces* (denoted $\vec{F}_i^{ext}(t)$, the sum of all forces exerted on particle i by particles *not belonging* to this particle system), that is

$$\vec{F}_i^{total}(t) = \vec{F}_i^1(t) + \ldots + \vec{F}_i^n(t) = \vec{F}_i^{ext}(t) + \vec{F}_i^{int}(t).$$

If we define the total force $\vec{F}_{total}^{system}(t)$ acting on the particle system as

$$\vec{F}_{total}^{system}(t) = \sum_{i=1}^{N} \vec{F}_i^{total}(t) = \sum_{i=1}^{N} \vec{F}_i^{ext}(t) + \sum_{i=1}^{N} \vec{F}_i^{int}(t),$$

then the Newton's Third Law tells us that $\vec{F}_{total}^{system} = \sum_{i=1}^{N} \vec{F}_i^{ext}(t)$ and so the internal forces cancel out each other. This is an important result that we will exploit later when we are going to model rigid bodies as a set of constrained particles.

3.2 Equations of Motion

In this section we will utilize our knowledge of Newton laws to formulate the equation of motion for a single particle and a set of particles (particle system) and will be interested in the way the equation can be used to simulate the system. For now, let us assume that we are provided with a mechanism to compute interaction forces.

If $\vec{r}(t)$ denotes the position of a particle with mass m, $\ddot{\vec{r}}(t) = \vec{a}(t)$ the acceleration of the particle and $\vec{F}_{total}(t)$ the total force acting on the particle then the Newton's second law mandates that

$$\ddot{\vec{r}}(t) = \vec{F}_{total}(t)/m. \tag{3.2}$$

This is called the particle's *equation of motion*. Assuming an inertial system, this equation describes the particle dynamics fully and obeys both the first and second Newton's laws. The third law has to be obeyed by the mechanism that computes $\vec{F}_{total}(t)$.

As it will turn out, when studying constrained particle dynamics, it is often advantageous to rewrite the equation of motion as

$$M \cdot \ddot{\vec{r}}(t) = \vec{F}_{total}(t), \tag{3.3}$$

where M is a $n \times n$ diagonal matrix with the particle's mass m on the diagonal, n is the dimension of the $\vec{a}(t)$ vector (usually $n = 3$) and $\vec{F}_{total}(t)$ is the net force exerted on the particle at time t. Matrix M is called the *mass matrix* of the particle

$$M = \begin{pmatrix} m & 0 & \ldots & 0 \\ 0 & m & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & m \end{pmatrix}.$$

Both the two equations (3.2) and (3.3) are equivalent second order ODEs, where the first equation is expressed in the canonical second order ODE form and the second equation is the rearranged first equation. When we will refer to an equation of motion we will mean either of these two equations.

To simulate the particle, we will have to track its position $\vec{r}(t)$ according to its equation of motion, say (3.2). Fortunately, the equation is an ODE where $\vec{r}(t)$ is the unknown function and so the numerical solving of the ODE corresponds to the motion simulation.

In order to solve (3.2) using our first order ODE solver, we have to convert it to a first order ODE by letting $\dot{\vec{r}} = \vec{v}$ and making the velocity of the particle a part of the ODE state. We also have to expand the obtained coupled vector equations by their components so that the resulting equations would match our canonical ODE definition[2]. The ODE state will be characterized by the components of the particle position $\vec{r}(t)$ and the velocity $\vec{v}(t)$ concatenated into a 6-vector $\vec{y}(t) = (r_1(t), r_2(t), r_3(t), v_1(t), v_2(t), v_3(t))$:

$$\dot{r}_1(t) = v_1(t) \qquad \dot{r}_2(t) = v_2(t) \qquad \dot{r}_3(t) = v_3(t)$$
$$\dot{v}_1(t) = (F_{total})_1(t)/m \qquad \dot{v}_2(t) = (F_{total})_2(t)/m \qquad \dot{v}_3(t) = (F_{total})_3(t)/m$$
$$\underbrace{(\dot{r}_1, \dot{r}_2, \dot{r}_3, \dot{v}_1, \dot{v}_2, \dot{v}_3)}_{\vec{y}'(t)} = \underbrace{(v_1, v_2, v_3, (F_{total})_1/m, (F_{total})_2/m, (F_{total})_3/m)}_{\vec{f}(t,\vec{y}(t))} \tag{3.4}$$

The equation of motion for a system of N particles is obtained by concatenating the equations of motion due to individual particles. The corresponding first order ODE needed for the simulation of the particle system is obtained by concatenating the individual scalar first order ODE equations.

If we define $\vec{r} = (r_1^1, r_1^2, r_1^3, \ldots, r_N^1, r_N^2, r_N^3)$, $\vec{F}_{total} =((\vec{F}_1^{total})_1, (\vec{F}_1^{total})_2, (\vec{F}_1^{total})_3, \ldots, (\vec{F}_N^{total})_1, (\vec{F}_N^{total})_2, (\vec{F}_N^{total})_3)$ and $\vec{m} = (m_1, m_1, m_1, \ldots, m_N, m_N, m_N)$, where $\vec{r}_i$ is the position of the i-th particle, $\vec{F}_i^{total}$ net force exerted on that particle and m_i is its mass, then the equation of motion for the particle system can be written as

$$\ddot{\vec{r}}(t) = \vec{F}_{total}(t) \cdot \left(E \cdot \vec{m}^{-1}\right),$$

where $\vec{m}^{-1} = (m_1^{-1}, m_1^{-1}, m_1^{-1}, \ldots, m_N^{-1}, m_N^{-1}, m_N^{-1})$ and E is a $3N \times 3N$ identity matrix.

If we define matrix M as a $3N \times 3N$ diagonal matrix with the values of the vector $\vec{m}$ on its diagonal, we can express (rearrange) the equation of motion for the particle system in terms of M, which is called the *mass matrix of the particle system*

$$M \cdot \ddot{\vec{r}}(t) = \vec{F}_{total}(t).$$

Convention 2 *To improve readability and avoid messy equations, we will often define matrices and vectors in a block fashion, meaning that the elements of the matrices and vectors can be other matrices and vectors "embedded" at the particular position of the enclosing matrix or vector. For example if we have vectors $\vec{a} = (a_1, a_2, a_3)$ and $\vec{b} = (b_1, b_2, b_3)$ then $\vec{c} = (\vec{a}, \vec{b})$,* defined in a block fashion*, will refer to a vector $(a_1, a_2, a_3, b_1, b_2, b_3)$.*

Operations on matrices and vectors defined in a block fashion are defined in terms of operations performed on the blocks. For example if $2 \cdot \vec{c}$ refers to a vector $\vec{c}$ whose components are to be multiplied by the scalar 2 then $2 \cdot \vec{c} = (2 \cdot \vec{a}, 2 \cdot \vec{b})$ which refers to $(2 \cdot a_1, 2 \cdot a_2, 2 \cdot a_3, 2 \cdot b_1, 2 \cdot b_2, 2 \cdot b_3)$ according to our convention.

If M_i are the mass matrices of the individual particles, $\vec{r} = (\vec{r}_1, \ldots, \vec{r}_N)$ and $\vec{F}_{total} = (\vec{F}_1^{total}, \ldots, \vec{F}_N^{total})$ are block vectors and M, called the mass matrix of the particle system, is a square diagonal block matrix with matrices M_i on the diagonal then the equation of motion of the particle system can be written as

$$\begin{aligned} M \cdot \ddot{\vec{r}}(t) &= \vec{F}_{total}(t) \qquad (3.5) \\ M &= \begin{pmatrix} M_1 & 0 & \ldots & 0 \\ 0 & M_2 & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & M_n \end{pmatrix} \end{aligned}$$

[2]We are assuming that the particle moves in $\mathbf{R}^3$.

and the corresponding first order ODE $\frac{\partial}{\partial t}\vec{y}(t) = \vec{f}(t, \vec{y}(t))$, defined in a block fashion, has the form of

$$\frac{\partial}{\partial t}((\vec{r}_1, \vec{v}_1), \ldots, (\vec{r}_N, \vec{v}_N)) = ((\vec{v}_1, \vec{F}_1^{total}/m_1), \ldots, (\vec{v}_N, \vec{F}_N^{total}/m_N)). \tag{3.6}$$

3.3 Abstraction

The abstraction of particles, real-world laws to model particle interaction by forces and the representation of forces in the simulator is discussed in this section. Granted we know how to compute appropriate forces that act on particles at given times ($\vec{F}_i^{total}(t)$ $\forall$ particle i) we have got an almost complete particle simulator — to simulate particles, we have just to provide and implement functions to manage the simulation state and evaluate state derivatives (compute acting forces) as required by our ODE solver (see page 13), the rest of the task is accomplished by the ODE solver. The implementation of these functions is simple and pretty straightforward though.

Generally, we will be using *force laws*, as opposed to the *laws of physics*, to model particle interaction by forces. Our laws will often be using approximations of real-world laws to simplify the force computation or will be purposefully designed to make the particle system exhibit a desired behavior. It will be up to our laws to ensure physical correctness. We will assume that the particle system is simulated according to (3.6) and that the following abstraction of particles, force laws and forces is used.

Particles are represented by objects so that the representation of particle i would

- store the particle's ODE state ($\vec{r}_i$ and $\vec{v}_i$ vectors)

  This state corresponds to a cached block of the global ODE state vector $\vec{y}$ that is associated with this particle. The most recent version of the block is always stored in $\vec{y}$ maintained by the ODE solver and mechanisms to synchronize the local state with the corresponding block in $\vec{y}$ are defined[3]:

  – ODE solver keeps and maintains the whole simulation state and pushes the state changes to individual particles as simulation advances.

  – Whenever a particle needs to change its state on its own behalf, it must notify the ODE solver so that it could restart self and pick up the new particle's state.

  The simulation state is thus fully managed by the ODE solver, can be handled globally and remains opaque to the application. This allows to implement certain advanced features such as adaptive time stepping, rolling back on penetration, support for multiple simulation groups as well as employ several optimizations.

- provide functions to copy the particle's ODE state to the corresponding block of the global state vector $\vec{y}$ and vice versa (**get_state**, **set_state**) and to evaluate the block of the global system state derivative vector (**evaluate_derivative**)

  These functions are invoked by the simulator to satisfy ODE solver requests described on page 13. Whenever such a request is intercepted, the list of all particles is traversed and the corresponding operations on the relevant blocks of $\vec{y}$ or $\vec{f}$ are performed. For example if the ODE solver triggers a **set_state** operation, **set_state** is called on individual particles to distribute the blocks of $\vec{y}$ to the particles.

[3]Particles do not access the content of the blocks directly but use a local cache. That is because the extra indirection at each access might be costy and would prevent certain optimizations to take place.

- hold $\vec{F}_i^{total}$ force accumulator that is used to accumulate the force (as a quantity) to be exerted on that particular particle at the current time. Note that we are only interested in the net force exerted on that particle by the superposition principle and so can get along with a single force accumulator per a particle.

Forces $\vec{F}_i^k$ that contribute to $\vec{F}_i^{total}$ accumulators are represented by force objects that record the force types, lists of "attached" particles that the forces act on and provide functions that can "apply" (add) the represented forces to the $\vec{F}_i^{total}$ accumulators of the attached particles[4]. Since all potential interactions are modelled by force objects, we know what particles can potentially affect other particles which can be used to partition particles to independent (isolated) simulation groups. Forces $\vec{F}_i^k$ are computed according to force laws.

Particle system maintains a list of all particles and force object structures, which allows it to cooperate with the ODE solver when managing simulation state and evaluating system state derivative. The evaluation of the state derivative involves the computation of $\vec{F}_i^{total}$ for all particles i. This is done by clearing force accumulators, traversing the list of all forces and adding corresponding force constributions to relevant accumulators.

According to the number of particles the particular force object acts or depends on (the number of particles whose accumulators are affected by the force object or whose states are read by the force object in order to compute and apply a force), the following categories of forces (or force laws)[5] can be defined:

- *Unary forces*, acting independently on each particle.

  Exerted force can be either a constant force or a force that can depend on the current time and/or the state (position, velocity) of the particle whose accumulator is to be updated.

  Gravity $\vec{f} = m \cdot \vec{g}$, where $\vec{g}$ is a gravitational constant, or viscous drag $\vec{f} = -k_d \cdot \vec{v}$, where k_d is a coefficient of drag, acting to resist particle motion, might be examples of unary forces.

  Although unary forces do not exist in a real world (all real forces are due to particle interaction), they are often used in simulators to simplify the real-world laws. For example our gravity force approximates the interaction between a particle and the Earth, viscous drag approximates the interaction of the particle and particles in the air, which are not represented in the particle system.

- *n-ary forces*, acting and depending on a fixed set of particles.

  These forces approximate forces due to interaction of a fixed set of particles or simulate mechanisms that are not represented by particles in the particle system. They are designed purposely to attract the system state to a desired configuration. Springs, or more generally penalty forces, can be examples of such forces.

- *Spatial interaction* forces, acting and depending on any n-tuple of particles.

  These forces are used to implement attraction or repulsion of particles. Unlike n-ary forces, spatial interaction forces can act on any n-tuple of particles, depending on their positions. Usually, local interaction (interaction of particles that are close to each other) is considered only.

  Dynamics of fluids can be approximated by spatially interacting particles.

[4]Multiple $\vec{F}_i^k$ forces can be represented by the same force object.

[5]We will often treat the terms force and force law as synonyms.

3.4 Penalty Forces

In this section we will describe a certain class of forces called the *penalty forces* that can be used to pull the system towards a desired state. Given a vector function $\vec{C}_p$ defined in terms of positions of n particles[6] that measures how much "bad" the current state is, we will present a procedure to derive a force law that will act on the n particles in order to attract the system state towards a state with a "lower badness", ideally to $\vec{C}_p = \vec{0}$. The particles will be guided to approach states where $\vec{C}_p = \vec{0}$, and their behavior thus controlled by $\vec{C}_p$, called the *behavior function*.

Penalty forces can be used to implement various kinds of springs, approximate (sloppy) constraints and such. However due to the nature of the forces, penalty forces will compete with other forces and it is not guaranteed that the system state will eventually reach (or even approach) a desired state.

Let $\vec{C}_p(\vec{r}_1(t), \ldots, \vec{r}_n(t))$ be a behavior function of a set of particles indexed from 1 to n, which returns state "badness" measured as a column vector from $\mathbf{R}^m$. We will define the potential *energy* of the set of particles in terms of vector dot product of $\vec{C}_p$ as

$$E(\vec{r}_1(t), \ldots, \vec{r}_n(t)) = \frac{\vec{C}_p^T \cdot \vec{C}_p}{2} \tag{3.7}$$

and the force due to the potential energy acting at particle i as

$$\vec{f}_i(t) = -\left(\frac{\partial E}{\partial \vec{r}_i(t)}\right)^T. \tag{3.8}$$

The energy function is nothing else than an appropriately scaled square magnitude of $\vec{C}_p$ and the force $\vec{f}_i$ is defined so that it points in the direction opposite to the energy gradient with respect to the particle position, thus making the particle "want to accelerate" towards a nearby state with a lower energy, [10], [13]. By taking the definition of the energy function E, realizing how vector dot product is differentiated[7] and assuming that $\vec{C}_p = (C_p^1, \ldots, C_p^m)$ and $\vec{r}_i = (r_i^1, r_i^2, r_i^3)$ we get

$$\begin{aligned}
\vec{f}_i(t) &= -\left(\frac{\partial E}{\partial \vec{r}_i(t)}\right)^T = -\left(\frac{\partial \vec{C}_p}{\partial \vec{r}_i(t)}\right)^T \cdot \vec{C}_p = -J_i^T \cdot \vec{C}_p \\
\vec{f}_i(t) &= -\left(\frac{\partial E}{\partial r_i^1}, \frac{\partial E}{\partial r_i^2}, \frac{\partial E}{\partial r_i^3}\right) = -\left(\frac{\partial \vec{C}_p^T}{\partial r_i^1} \cdot \vec{C}_p, \frac{\partial \vec{C}_p^T}{\partial r_i^2} \cdot \vec{C}_p, \frac{\partial \vec{C}_p^T}{\partial r_i^3} \cdot \vec{C}_p\right) \\
&= -\begin{pmatrix} \frac{\partial C_p^1}{\partial r_i^1} & \frac{\partial C_p^1}{\partial r_i^2} & \frac{\partial C_p^1}{\partial r_i^3} \\ \vdots & \vdots & \vdots \\ \frac{\partial C_p^m}{\partial r_i^1} & \frac{\partial C_p^m}{\partial r_i^2} & \frac{\partial C_p^m}{\partial r_i^3} \end{pmatrix}^T \cdot \begin{pmatrix} C_p^1 \\ \vdots \\ C_p^m \end{pmatrix} = -J_i^T \cdot \vec{C}_p,
\end{aligned} \tag{3.9}$$

where $J_i = \frac{\partial \vec{C}_p}{\partial \vec{r}_i}$ is called the *Jacobian matrix* of $\vec{C}_p$ due to particle i.

There are fundamental problems with $\vec{f}_i$:

- First, $\vec{f}_i$ will compete with other forces acting on the particle and its effects can be cancelled by the other forces.

[6]The $_p$ subscript is used to stress out that we are working with particle positions. Since the positions are a part of the system state, $\vec{C}_p$ assesses the system state with respect to the positions of the n particles.

[7]$(\vec{u} \cdot \vec{v})' = \vec{u}' \cdot \vec{v} + \vec{u} \cdot \vec{v}'$, $(\alpha \cdot \vec{u})' = \alpha \cdot \vec{u}'$.

- Second, since $\vec{C}_p$ is defined in terms of particle positions (and so is J_i), $\vec{f}_i$ merely suggests the direction the particle i should accelerate along. Its magnitude is determined by the current value of $\vec{C}_p$, which is independent of the particle mass[8] and other forces exerted on the particle. To "correct" this, user has to specify a *spring constant* k_s that is used to scale the former force to account for the particle mass. Constant k_s must be tuned by trial-and-error and it is extremely hard and tedious to make the system behave as expected. Very small changes made to the scene definition (parameter changes, introduction of new springs, etc.) often require the user to tune all k_s constants once again from scratch.

- Third, if $\vec{C}_p$ is meant to measure a constraint error, the constraint must be violated first ($\vec{C}_p$ must become non-zero) in order the force $\vec{f}_i$ would begin to act ($\vec{C}_p = \vec{0} \Rightarrow \vec{f}_i = \vec{0}$). This implies that the system will (at best) oscillate around $\vec{C}_p = \vec{0}$ and a damping force, controlled by a tuned *damping constant* k_d, must be added. An attempt to make the constraint more accurate by making the spring constant big, so that a very small constraint error would induce a very large restoration force, produces unstable/stiff ODEs and an extremely small time step must be taken in order to solve the ODE, [10].

To fight some of these problems, *penalty force* due to $\vec{C}_p$ acting at particle i is defined instead as $\vec{f}_i = J_i^T \cdot (-k_s \cdot \vec{C}_p - k_d \cdot \dot{\vec{C}}_p)$, where $\dot{\vec{C}}_p = \frac{\partial}{\partial t}\vec{C}_p$ and $J_i = \frac{\partial \vec{C}_p}{\partial \vec{r}_i}$.

A zero-length spring attached to particles 1 and 2 can be described by $\vec{C}_p = \vec{r}_1 - \vec{r}_2 = \vec{0}$ behavior function, which yields $\vec{f}_1 = -k_s \cdot (\vec{r}_1 - \vec{r}_2) - k_d \cdot (\dot{\vec{r}}_1 - \dot{\vec{r}}_2) = -k_s \cdot (\vec{r}_1 - \vec{r}_2) - k_d \cdot (\vec{v}_1 - \vec{v}_2)$, because J_1 is an identity matrix. Similarly we get $\vec{f}_2 = -\vec{f}_1$, because $J_2 = -J_1$.

[8]This implies that the behavior of the system will become totally different if the mass is changed.

Chapter 4

Constrained Particle Dynamics

In this chapter we will cover the basics of constrained particle dynamics, which will nicely generalize for the case of constrained rigid body dynamics, discussed later. The problem of constrained particle dynamics is to obey both Newton laws and a set of constraints specified by user.

There are two approaches to enforce constraints. The first approach called the *reduced coordinates approach* expresses equations of motion of the constrained particles in terms of new unconstrained DOFs, that parametrize (capture) the set of valid particle states. The second approach called the *maximal coordinates approach* expresses equations of motion in terms of maximal (full) coordinates, that are constrained by the set of constraints. Constraint forces are introduced into the system to enforce the constraints (make the particle states remain in the set of valid states). The derivations will follow [10], but the used notation will correspond to [9] so that the material could be generalized for rigid bodies.

4.1 Lagrange Multiplier Method

We will study a *Lagrange multiplier method* that operates in terms of maximal coordinates. This method allows to handle each constraint in the same way (by introducing a clever constraint abstraction) and allows to combine constraints automatically.

4.1.1 Method Background

Constraints will be described by implicit functions (equations) defined either in terms of particle positions or particle velocities so that given system state would be considered valid and the constraint maintained if the constraint function returns zero for that state (equation will hold)[1]. Granted the constraint is maintained already, *constraint solver* will compute an appropriate constraint force so that the simulator would not advance to an invalid state. Constraint forces are constructed so that they would cancel exactly those components of $\vec{F}_{total}$ accumulators that would make the particles accelerate towards invalid states, thus forcing the particles to remain in the set of valid states. To do this, constraints are reformulated to conditions on particle accelerations, which allows to compute appropriate constraint forces.

This is the principal difference from the penalty method, where constraints implemented by penalty forces must be violated first in order the restoration force would begin to act to correct the

[1]Constraints defined in this way are called *equality constraints*. In this section we will restrict ourselves to this class of constraints only.

constraint error (small constraint errors would induce large restoration forces and the underlying ODE will become unstable and stiff).

4.1.2 Constraint Formulation

Definition 8 *Let particles to be constrained are indexed by* $1, \ldots, n$*, then the implicit function (equation) of the form* $\vec{C}_p(\vec{r}_1(t), \ldots, \vec{r}_n(t)) = \vec{0} \in \mathbf{R}^m$ *defines a position-level constraint on the particles and* m *is called the dimension of the constraint or the number of DOFs constrained (removed from the system) by the constraint.*

Individual scalar equations ("constraint rows") $C_p^i = 0, 1 \leq i \leq m$ *correspond to the removed DOFs in such a way that the i-th equation corresponds to the i-th DOF removed from the system.*

Vectors $\vec{r}_1(t), \ldots, \vec{r}_n(t)$*, where* $\vec{C}_p(\vec{r}_1(t), \ldots, \vec{r}_n(t)) = \vec{0}$*, implicitly define the set of valid particle positions.*

To simplify the manipulation with vectors due to individual particles, let us conceptually concatenate vectors $\vec{r}_1, \ldots, \vec{r}_n$ to a single block vector $\vec{r} = (\vec{r}_1, \ldots, \vec{r}_n)$ and define $\vec{v} = \dot{\vec{r}}$, $\vec{a} = \ddot{\vec{r}}$. This allows us to manipulate all particles altogether, treat the constrained particles as a monolith and to write, for example, that vectors $\vec{r}$, where $\vec{C}_p(\vec{r}) = \vec{0}$, define the set of valid particle positions, meaning that the components of the first block $\vec{r}_1$ of such $\vec{r}$ vectors correspond to the valid positions of the first particle, etc. We will be working in $\mathbf{R}^3$ and so the dimensions of $\vec{r}_i$, $\vec{v}_i$ and $\vec{a}_i$ vectors will be 3×1 and the dimensions of the $\vec{r}$, $\vec{v}$ and $\vec{a}$ vectors $3n \times 1$.

If we differentiate $\vec{C}_p = \vec{0}$ with respect to time we will get the corresponding *velocity-level constraint* $\vec{C}_v = \vec{0}$ that imposes conditions on the valid particle velocities

$$\vec{C}_v(\vec{r}, \vec{v}) = \dot{\vec{C}}_p = \frac{\partial \vec{C}_p(\vec{r})}{\partial t} = \frac{\partial \vec{C}_p}{\partial \vec{r}} \cdot \frac{\partial \vec{r}}{\partial t} = J(\vec{r}) \cdot \vec{v} = \vec{0},$$

where $J(\vec{r}) = \frac{\partial \vec{C}_p}{\partial \vec{r}}$ is the Jacobian matrix of $\vec{C}_p$ with dimensions $m \times 3n$.

By differentiating the velocity-level equation $\vec{C}_v = \vec{0}$ with respect to time, we will get the corresponding *acceleration-level constraint* $\vec{C}_a = \vec{0}$ that imposes conditions on the valid particle accelerations

$$\vec{C}_a(\vec{r}, \vec{v}, \vec{a}) = \ddot{\vec{C}}_p = \frac{\partial^2 \vec{C}_p}{\partial^2 t} = \frac{\partial}{\partial t} \vec{C}_v(\vec{r}, \vec{v}) = \frac{\partial J \cdot \vec{v}}{\partial t} = \dot{J} \cdot \vec{v} + J \cdot \vec{a} = \vec{0}.$$

By the chain rule we also get

$$\frac{\partial}{\partial t} \vec{C}_v(\vec{r}, \vec{v}) = \left(\frac{\partial \vec{C}_v}{\partial \vec{r}} \cdot \frac{\partial \vec{r}}{\partial t} \right) + \left(\frac{\partial \vec{C}_v}{\partial \vec{v}} \cdot \frac{\partial \vec{v}}{\partial t} \right) = \left(\dot{J} \cdot \vec{v} \right) + \left(J \cdot \vec{a} \right),$$

which implies that

$$\dot{J} = \frac{\partial \vec{C}_v}{\partial \vec{r}}$$

$$J = \frac{\partial \vec{C}_v}{\partial \vec{v}} = \frac{\partial \vec{C}_p}{\partial \vec{r}} = \left(\frac{\partial \vec{C}_v}{\partial \vec{v}_1} \quad \ldots \quad \frac{\partial \vec{C}_v}{\partial \vec{v}_n} \right) = \left(\frac{\partial \vec{C}_p}{\partial \vec{r}_1} \quad \ldots \quad \frac{\partial \vec{C}_p}{\partial \vec{r}_n} \right) \tag{4.1}$$

and both the velocity-level and acceleration level equations use the same Jacobian matrix J, this time written by blocks due to individual particles. Finally, by rewriting and introducing

$\vec{c}(\vec{r},\vec{v}) = -\dot{J}\cdot\vec{v}$ we can express the acceleration-level constraint $\vec{C}_a(\vec{r},\vec{v},\vec{a}) = \vec{0}$ as $\vec{C}_a(\vec{r},\vec{v},\vec{a}) = J(\vec{r})\cdot\vec{a} - \vec{c}(\vec{r},\vec{v}) = \vec{0}$ or simply as

$$J(\vec{r})\cdot\vec{a} = \vec{c}(\vec{r},\vec{v}). \tag{4.2}$$

Having known $\vec{C}_p$, $\dot{\vec{C}}_p = \vec{C}_v$ and $\ddot{\vec{C}}_p = \vec{C}_a$, the constraint of the $\vec{C}_p = \vec{0}$ form can be enforced on the acceleration level by:

- starting from an initial state where $\vec{C}_p = \dot{\vec{C}}_p = \vec{0}$ and
- requiring that $\ddot{\vec{C}}_p = \vec{0}$ at each step.

The first condition ensures that the initial positions and velocities are consistent with the constraint and the second condition ensures that $\dot{\vec{C}}_p$ will remain equal to $\vec{0}$ (velocities will remain valid), which in turn ensures that $\vec{C}_p$ will remain equal to $\vec{0}$ (positions will remain valid). Moreover $\ddot{\vec{C}}_p = \vec{0}$ is a condition on the particle accelerations and will be used to derive the constraint force.

In the further discussion we will assume that the velocity-level constraint function is provided directly, instead of being derived from the position-level constraint function $\vec{C}_p$. The constraint is assumed to be defined as $\vec{C}_v(\vec{r},\vec{v}) = J(\vec{r})\cdot\vec{v} - \vec{k} = \vec{0}$, where $\vec{k}$ is a *constant* vector, or simply as

$$J(\vec{r})\cdot\vec{v} = \vec{k}. \tag{4.3}$$

This is more general because it allows to specify velocity-dependent (*non-holonomic*) constraints[2], that can not be formulated on the position-level, as well as position-dependent constraints, by taking the time derivative of $\vec{C}_p$. By differentiating (4.3) with respect to time we will get the corresponding acceleration-level constraint in the form of (4.2).

4.1.3 Constraint Force Basis

Lagrange multiplier method computes the constraint force $\vec{F}_c$ as a linear combination of basis vectors $\vec{F}_1,\ldots,\vec{F}_m$, derived from the constraint and known *a priori*. The coefficients of the linear combination are called the *Lagrange multipliers* and are concatenated into a vector $\vec{\lambda}$. Lagrange multipliers are the unknowns we have to solve for.

$$\begin{aligned} \vec{F}_c &= \lambda_1\cdot\vec{F}_1 + \ldots + \lambda_m\cdot\vec{F}_m \\ \vec{\lambda} &= (\lambda_1,\ldots,\lambda_m) \end{aligned} \tag{4.4}$$

$\vec{F}_c$ is a block vector organized similarly to $\vec{v}$, that is $\vec{F}_c = (\vec{F}^c_1,\ldots,\vec{F}^c_n)$, where $\vec{F}^c_i$ is the constraint force to be exerted on particle i. Basis vectors $\vec{F}_i$ are organized in the same way. We will now derive what the vectors $\vec{F}_i$ should be, given a velocity-level constraint (4.3).

Recall that vector pairs $(\vec{r},\vec{v})$ that satisfy equation (4.3) define the set of valid particle velocities $\vec{v}$ with respect to particle positions $\vec{r}$. Let us treat vectors $\vec{r}$ and $\vec{v}$ as points in multi-dimensional spaces called the position and velocity space, fix $\vec{r}$ to its current value $\vec{r}_\bullet$ (particle positions at the current time) and characterize the condition in terms of the velocity space, as $\vec{C}_v(\vec{r}_\bullet,\vec{v}) = \vec{C}(\vec{v}) = \vec{0}$. Since this is a vector equation, it can be rewritten by the components of $(C_1,\ldots,C_m) = (0,\ldots,0)$, yeilding m scalar equations $C_i = 0$ $(1 \le i \le m)$. Each such an equation is an implicit functions of $\vec{v}$ ($\vec{r}$ remains fixed). The sets of vectors $\vec{v}$ that satisfy a given $C_i = 0$ constraint correspond to hypersurfaces in the velocity space and the set of valid

[2]Such as $\vec{C}_v = \vec{v}_1 - \vec{v}_2 - \vec{k} = \vec{0}$, where $\vec{k}$ is a constant vector.

velocities with respect to the current particle positions thus corresponds to the intersection of the hypersurfaces (all $C_i = 0$, $1 \le i \le m$ constraints must hold).

Let us assume that $\vec{v}$ already lies on the intersection of the hypersurfaces. In order to maintain the constraint we must ensure that $\vec{r}$ will not accelerate in such a way so that $\vec{v}$ will be driven off the hypersurface i due to $C_i = 0$, $1 \le i \le m$, that is, the acceleration of $\vec{r}$ in the position space along the normal of the hypersurface i, which equals an appropriately scaled vector $\left(\frac{\partial C_i(\vec{v})}{\partial \vec{v}}\right)$, must be constrained[3]. This vector would thus be our $\vec{F_i}$. By utilizing equation (4.3) we can write that

$$\vec{F_i}(\vec{r}_\bullet)^T = \frac{\partial C_i}{\partial \vec{v}} = \frac{\partial \left[\vec{C_v}(\vec{r}_\bullet, \vec{v})\right]_i}{\partial \vec{v}} = \frac{\partial \left[J(\vec{r}_\bullet) \cdot \vec{v} - \vec{k}\right]_i}{\partial \vec{v}} = \left[J(\vec{r}_\bullet)\right]_i \cdot \frac{\partial \vec{v}}{\vec{v}} - \frac{\partial k_i}{\partial \vec{v}} = \left[J(\vec{r}_\bullet)\right]_i ,$$

or, more readily, that

$$\vec{F_1} = [J]_1^T, \ldots, \vec{F_m} = [J]_m^T,$$

where $[\;]_i$ operator is used to index rows[4]. The rows of the Jacobian matrix J will be the vectors $\vec{F_1}, \ldots, \vec{F_m}$ and these vectors will indeed form a basis because J is expected to have a full-rank (there would be infeasible or duplicate constraints otherwise). See figure (4.1) for illustration.

By substituting vectors $\vec{F_i}$ into equation (4.4) we get that the constraint force $\vec{F_c}$ can be expressed as

$$\vec{F_c} = J^T \cdot \vec{\lambda}. \tag{4.5}$$

Constraint force $\vec{F_i^c} = (J^T \cdot \vec{\lambda})_i = J_i^T \cdot \vec{\lambda}$ to be exerted on particle $1 \le i \le n$ resembles the penalty force equation (3.9). In both the cases, the constraint force and the penalty force are defined as a linear combination of a priori known basis vectors (the rows of the Jacobian matrix), but the multipliers used to combine the basis are different. In the case of penalty forces the multipliers are given by the current value of the behavior function, ignoring the effects of other forces. In the case of constraint forces, however, the multipliers are considered unknown and their values are determined globally so that all constraints would be maintained.

4.1.4 Lagrange Multipliers

At last we get to the computation of the constraint force $\vec{F_c}$ in terms of the multipliers $\vec{\lambda}$. But before we do so, let us summarize and interpret some facts that we have discovered so far.

- Position-level constraints and velocity-level constraints (with constant vectors $\vec{k}$ on the right-hand side) can be transformed to acceleration-level constraints in the form of

  $$J \cdot \vec{a} = \vec{c},$$

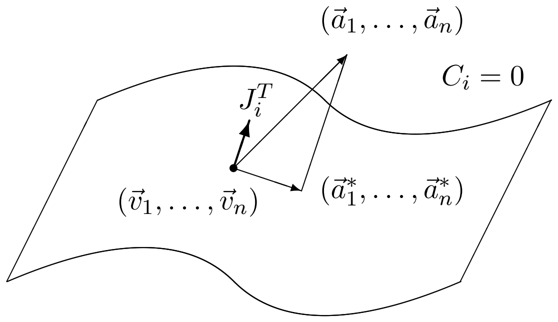
  where J is a $m \times 3n$ Jacobian matrix, m is the dimension of the constraint, n is the number of constrained particles and the particles are indexed $1, \ldots, n$ by differentiating the original constraint with respect to time once or twice.

- Position-level constraints and velocity-level constraints can be maintained

[3]To ensure that $\vec{v}$ is not driven from the hypersurface in the velocity space, $\dot{\vec{v}}$ along the hypersurface normal is constrained. Since $\ddot{\vec{r}} = \dot{\vec{v}} = \vec{a}$, the acceleration $\vec{a}$ of $\vec{r}$ in the position space along the hypersurface normal is constrained actually.

[4]This is just for clarification and to avoid confusion. In most other places J_i denotes the i-th block of J, the Jacobian due to particle i. If there are multiple constraints in effect, J_i denotes the Jacobian due to constraint i and $J_{i,j}$ the j-th block of J_i, the Jacobian due to constraint i and particle j.

Figure 4.1: Illustration of the hypersurface and the constraint force due to the i-th DOF of a constraint. Having a point $\vec{r} = (\vec{r}_1, \ldots, \vec{r}_n)$ in the position space describing the positions of particles, point $\vec{v} = \dot{\vec{r}}$ in the velocity space describing the particle velocities and already lying on the hypersurface due to $C_i = 0$ then $\vec{a} = \dot{\vec{v}}$ due to the total external force $\vec{F}_{total} = (\vec{F}_1^{total}, \ldots, \vec{F}_n^{total})$ must be corrected so that the new $\vec{a}^* = (\vec{a}_1^*, \ldots, \vec{a}_n^*)$ would not drive $\vec{v}$ from the hypersurface. This is done by introducing a constraint force $J_i^T \cdot \lambda_i$ that acts along the hypersurface normal.


$(\vec{a}_1, \ldots, \vec{a}_n)$
$C_i = 0$
J_i^T
$(\vec{v}_1, \ldots, \vec{v}_n)$
$(\vec{a}_1^*, \ldots, \vec{a}_n^*)$

   – by starting from a state that is consistent with the position-level[5] and velocity-level formulation of the constraint and
   – obeying the corresponding acceleration-level constraint instead of the original position-level or velocity-level constraint.

- Acceleration-level constraints are maintained by introducing a block vector constraint force $\vec{F}_c = (\vec{F}_1^c, \ldots, \vec{F}_n^c) = J^T \cdot \vec{\lambda}$, where $\vec{\lambda}$ is a vector of Lagrange multipliers of length m and $\vec{F}_i^c$ is the constraint force to be exerted on particle i. Multiplier λ_i defines a block vector constraint force that is exerted due to the i-th constraint row (due to the i-th DOF removed from the system) at the constrained particles and the exerted force equals $J_i^T \cdot \lambda_i$. In order to compute $\vec{F}_c$, multipliers $\vec{\lambda}$ must be computed.

Now let us assume that $\vec{F}_{total}$ is a block vector whose blocks consist of vectors of the net forces exerted on the individual particles, that is $\vec{F}_{total} = (\vec{F}_1^{total}, \ldots, \vec{F}_n^{total})$, where $\vec{F}_i^{total}$ is the net force exerted on particle i. We will compute $\vec{F}_c = J^T \cdot \vec{\lambda}$ so that the acceleration-level equation $J \cdot \vec{a} = \vec{c}$ would hold when constraint force $\vec{F}_c$ is exerted.

Dynamics of particles $1, \ldots, n$, with mass matrices $M_1, \ldots, M_n$ can be described in a block fashion by the following equation of motion (see (3.5))

$$M \cdot \ddot{\vec{r}}(t) = \vec{F}_{total}(t),$$

where M is the block diagonal matrix

$$M = \begin{pmatrix} M_1 & 0 & \ldots & 0 \\ 0 & M_2 & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & M_n \end{pmatrix}.$$

When $\vec{F}_c$ is exerted on the particles then the total force acting on the particles would be equal to $\vec{F}_{total} + \vec{F}_c$, therefore

$$M \cdot \ddot{\vec{r}} = \vec{F}_{total} + \vec{F}_c$$

[5]If the original constraint is a position-level constraint.

and thus

$$\vec{a} = \ddot{\vec{r}} = M^{-1} \cdot (\vec{F}_{total} + \vec{F}_c)$$
$$\vec{a} = M^{-1} \cdot \vec{F}_{total} + M^{-1} \cdot \vec{F}_c.$$

By utilizing the fact that $F_c = J^T \cdot \vec{\lambda}$ (4.5) and that $J \cdot \vec{a} = \vec{c}$ should hold when $\vec{F}_c$ is exerted (4.2), we get

$$\vec{a} = M^{-1} \cdot \vec{F}_{total} + M^{-1} \cdot J^T \cdot \vec{\lambda}$$
$$J \cdot \vec{a} = J \cdot M^{-1} \cdot \vec{F}_{total} + J \cdot M^{-1} \cdot J^T \cdot \vec{\lambda} = \vec{c}$$
$$\left(J \cdot M^{-1} \cdot J^T\right) \cdot \vec{\lambda} + \left(J \cdot M^{-1} \cdot \vec{F}_{total} - \vec{c}\right) = \vec{0},$$

finally producing a linear condition on the multipliers $\vec{\lambda}$ at the current time t:

$$A \cdot \vec{\lambda} + \vec{b} = \vec{0}, \tag{4.6}$$

where $A = J \cdot M^{-1} \cdot J^T$ is a $m \times m$ matrix and $\vec{b} = J \cdot M^{-1} \cdot \vec{F}_{total} - \vec{c}$ is a $m \times 1$ column vector and both A and $\vec{b}$ are evaluated with respect to the current time t, particle positions $\vec{r}$ and velocities $\vec{v}$[6].

Since J is assumed to be regular and M is positive-definite, A can be factored using standard factorization techniques to solve for $\vec{\lambda}$. Once $\vec{\lambda}$ multipliers are computed, constraint force $\vec{F}_c = J^T \cdot \vec{\lambda}$ can be exerted on the particles.

This is a theoretical result. In the practice, however, special properties of J and M matrices are utilized in order to compute $\vec{\lambda}$ efficiently and achieve interactive performance even for highly constrained systems.

4.1.5 An Example

In this section we will show how the Lagrange multiplier method can be used to constrain two particles with indices 1 and 2 ($n = 2$) so that their relative distance will remain fixed.

The discussed constraint can be formalized as a one-dimensional position constraint ($m = 1$) in the form of

$$C_p(\vec{r}_1, \vec{r}_2) = \|\vec{r}_1 - \vec{r}_2\|^2 - d^2 = 0,$$

where $d > 0$ is the desired relative particle distance.

To simplify further derivation we will define $\vec{x} = \vec{r}_1 - \vec{r}_2$, which will allow us to rewrite the constraint as

$$C_p = \vec{x} \cdot \vec{x} - d \cdot d = 0.$$

Now we have to produce the corresponding velocity-level and acceleration-level equations $\dot{C}_p$ and $\ddot{C}_p$

$$\dot{C}_p = 2 \cdot \dot{\vec{x}} \cdot \vec{x} = J \cdot \vec{v} = 0$$
$$\ddot{C}_p = 2 \cdot \ddot{\vec{x}} \cdot \vec{x} + 2 \cdot \dot{\vec{x}} \cdot \dot{\vec{x}} = J \cdot \vec{a} + \dot{J} \cdot \vec{v} = 0,$$

where $\vec{v} = (\vec{v}_1, \vec{v}_2)$ and $\vec{a} = (\vec{a}_1, \vec{a}_2)$ are defined in a block fashion and $\dot{\vec{x}} = \vec{v}_1 - \vec{v}_2$ and $\ddot{\vec{x}} = \vec{a}_1 - \vec{a}_2$. By rewriting we get

$$\begin{aligned} \dot{C}_p = 2 \cdot (\vec{v}_1 - \vec{v}_2) \cdot (\vec{r}_1 - \vec{r}_2) &= \\ 2 \cdot (\vec{r}_1 - \vec{r}_2) \cdot \vec{v}_1 - 2 \cdot (\vec{r}_1 - \vec{r}_2) \cdot \vec{v}_2 &= \end{aligned}$$

[6]For example, as we already stated, J is a function of $\vec{r}$.

$$\begin{aligned}
J \cdot \vec{v} &= 0 \\
\ddot{C}_p = (2 \cdot (\vec{r}_1 - \vec{r}_2) \cdot \vec{a}_1 - 2 \cdot (\vec{r}_1 - \vec{r}_2) \cdot \vec{a}_2) + & \\
(2 \cdot (\vec{v}_1 - \vec{v}_2) \cdot \vec{v}_1 - 2 \cdot (\vec{v}_1 - \vec{v}_2) \cdot \vec{v}_2)) &= \\
J \cdot \vec{a} + \dot{J} \cdot \vec{v} &= 0 \\
J \cdot \vec{a} &= c = -\dot{J} \cdot \vec{v}
\end{aligned}$$

and thus obtain

$$\begin{aligned}
J = (\, 2 \cdot (\vec{r}_1 - \vec{r}_2)^T \quad | \quad -2 \cdot (\vec{r}_1 - \vec{r}_2)^T \,) &= (\, J_1 \quad J_2 \,) \\
\dot{J} = (\, 2 \cdot (\vec{v}_1 - \vec{v}_2)^T \quad | \quad -2 \cdot (\vec{v}_1 - \vec{v}_2)^T \,) &= (\, \dot{J}_1 \quad \dot{J}_2 \,) \\
c = -\dot{J} \cdot \vec{v} &= -2 \cdot (\vec{v}_1 - \vec{v}_2) \cdot (\vec{v}_1 - \vec{v}_2),
\end{aligned}$$

where J and $\dot{J}$ are 1×6 matrices ($m \times 3n, m = 1, n = 2$) defined in a block fashion. As we already know $J_1 = \frac{\partial C_p}{\partial \vec{r}_1} = \begin{pmatrix} \frac{\partial C_p}{\partial r_1^1} & \frac{\partial C_p}{\partial r_1^2} & \frac{\partial C_p}{\partial r_1^3} \end{pmatrix}_{1\times 3} = \frac{\partial \dot{C}_p}{\partial \vec{v}_1} = \begin{pmatrix} \frac{\partial \dot{C}_p}{\partial v_1^1} & \frac{\partial \dot{C}_p}{\partial v_1^2} & \frac{\partial \dot{C}_p}{\partial v_1^3} \end{pmatrix}_{1\times 3}$ and $J_2 = \frac{\partial C_p}{\partial \vec{r}_2} = \frac{\partial \dot{C}_p}{\partial \vec{v}2}$.

Now if $\vec{F}_1^{total}$ and $\vec{F}_2^{total}$ are the net forces acting on the first and second particles and M_1 and M_2 their mass matrices due to masses m_1 and m_2, we can form equation (4.6), $A \cdot \lambda + b = 0$, and solve for λ (which is a scalar value likewise b, because $m = 1$).

$$\begin{aligned}
A &= (\, J_1 \quad J_2 \,) \cdot \begin{pmatrix} M_1 & 0 \\ 0 & M_2 \end{pmatrix}^{-1} \cdot \begin{pmatrix} J_1^T \\ J_2^T \end{pmatrix} = \left((\, J_1 \quad J_2 \,) \cdot \begin{pmatrix} M_1^{-1} & 0 \\ 0 & M_2^{-1} \end{pmatrix} \right) \cdot \begin{pmatrix} J_1^T \\ J_2^T \end{pmatrix} \\
&= (\, J_1 \cdot M_1^{-1} \quad J_2 \cdot M_2^{-1} \,) \cdot \begin{pmatrix} J_1^T \\ J_2^T \end{pmatrix} = J_1 \cdot M_1^{-1} \cdot J_1^T + J_2 \cdot M_2^{-1} \cdot J_2^T \\
b &= (\, J_1 \quad J_2 \,) \cdot \begin{pmatrix} M_1 & 0 \\ 0 & M_2 \end{pmatrix}^{-1} \cdot \begin{pmatrix} \vec{F}_1^{total} \\ \vec{F}_2^{total} \end{pmatrix} - c = J_1 \cdot M_1^{-1} \cdot \vec{F}_1^{total} + \\
& \quad J_2 \cdot M_2^{-1} \cdot \vec{F}_2^{total} - c
\end{aligned}$$

Since $J_2 = -J_1 = -2 \cdot \vec{x}^T$, $M_i = \begin{pmatrix} m_i^{-1} & 0 & 0 \\ 0 & m_i^{-1} & 0 \\ 0 & 0 & m_i^{-1} \end{pmatrix}$ and $c = 2 \cdot \dot{\vec{x}} \cdot \dot{\vec{x}}$ we have

$$\begin{aligned}
A &= \frac{1}{m_1} \cdot J_1 \cdot J_1^T + \frac{1}{m_2} \cdot J_1 \cdot J_1^T = (\frac{1}{m_1} + \frac{1}{m_2}) \cdot 4 \cdot \vec{x} \cdot \vec{x} \\
b &= \frac{1}{m_1} \cdot J_1 \cdot \vec{F}_1^{total} - \frac{1}{m_2} \cdot J_1 \cdot \vec{F}_2^{total} - 2 \cdot \dot{\vec{x}} \cdot \dot{\vec{x}} \\
&= \frac{1}{m_1} \cdot 2 \cdot \vec{x} \cdot \vec{F}_1^{total} - \frac{1}{m_2} \cdot 2 \cdot \vec{x} \cdot \vec{F}_2^{total} - 2 \cdot \dot{\vec{x}} \cdot \dot{\vec{x}} \\
\lambda &= A^{-1} \cdot (-b) = \frac{m_1 \cdot m_2 \cdot \dot{\vec{x}} \cdot \dot{\vec{x}} + \vec{x} \cdot \vec{F}_2^{total} \cdot m_1 - \vec{x} \cdot \vec{F}_1^{total} \cdot m_2}{2 \cdot (m_1 + m_2) \cdot \vec{x} \cdot \vec{x}}
\end{aligned}$$

If $C_p = \dot{C}_p = 0$ then $A \neq 0$ because $\|\vec{x}\| > 0$ ($d > 0$) and λ can be computed. Having known λ, constraint force $\vec{F}_c = (\vec{F}_1^c, \vec{F}_2^c) = J^T \cdot \lambda = (J_1^T \cdot \lambda, J_2^T \cdot \lambda)$ is evaluated, $\vec{F}_1^c$ applied to the first particle, $\vec{F}_2^c$ to the second particle and the simulator can advance further, obeying the constraint.

4.2 Constraint Abstraction

In the previous example we have shown how a particular constraint can be implemented by symbolically deriving the constraint function derivatives and building the equation (4.6) to compute the multipliers and the constraint force. What we actually did was that we hard-coded a certain particle model that would have to be updated (reprogrammed) whenever a new constraint had to be added or removed. Although simple static models could be implemented this way, it would not be practical. We would like to have the model dynamic so that constraints could be added or removed on the fly (this is essential for a proper implementation of contacts), the simulator would formulate equation (4.6) itself and would be able to compute and apply the appropriate constraint force automatically. In this section we will discuss a constraint abstraction that would allow to implement this model.

4.2.1 Combining Constraints

So far we assumed that there was a single global velocity-level constraint $\vec{C}_v = \vec{0}$ in effect. In order to make the dynamic behaviour possible, $\vec{C}_v$ will have to vary over time, constrain a variable set of particles and be (conceptually) reformulated at each step.

Let N be the number of particles in the particle system, $\vec{r}_i$ the position of the i-th particle, $\vec{v}_i$ the velocity of the i-th particle, c number of constraints. Let us concatenate the particle positions, velocities and accelerations to block vectors $\vec{r} = (\vec{r}_1, \ldots, \vec{r}_N), \vec{v} = \frac{\partial}{\partial t}\vec{r}$ and $\vec{a} = \frac{\partial}{\partial t}\vec{v}$ and define the i-th velocity-level constraint in the form of equation (4.3) as $\vec{C}^v_i(\vec{r}, \vec{v}) = \vec{0} \in \mathbf{R}^{m_i}$, where m_i is the dimension of the constraint, or equivalently as

$$J_i(\vec{r}) \cdot \vec{v} = \vec{k}_i,$$

where J_i is a $m_i \times 3N$ Jacobian matrix and $\vec{k}_i$ is a constant vector of length m_i. Note that each constraint might constrain all particles $1, \ldots, N$ and so the equation involves the whole $\vec{r}$ and $\vec{v}$ vectors.

According to (4.1), J_i Jacobians can be treated as block matrices consisting of $m_i \times 3$ blocks $J_{ij}, 1 \leq j \leq N$, where $J_{ij} = \frac{\partial \vec{C}^v_i}{\partial \vec{v}_j}$, and so the i-th constraint $\vec{C}^v_i = \vec{0}$ can be rewritten as

$$J_{i1} \cdot \vec{v}_1 + \ldots + J_{iN} \cdot \vec{v}_N = \vec{k}_i$$

The global constraint $\vec{C}_v = \vec{0}$ that acts on the particles at the given time and captures the effects of all individual constraints is simply a vector equation whose components are the components of the individual constraints $\vec{C}^v_i = \vec{0}$ that are currently in effect. This can be expressed in the block form as $\vec{C}_v(\vec{r}, \vec{v}) = (\vec{C}^v_1(\vec{r}, \vec{v}), \ldots, C^v_c(\vec{r}, \vec{v})) = (\vec{0}, \ldots, \vec{0}) = \vec{0}$, or equivalently as

$$\begin{aligned} \begin{pmatrix} J_{11} & \ldots & J_{1N} \\ \vdots & \ddots & \vdots \\ J_{c1} & \ldots & J_{cN} \end{pmatrix} \cdot \begin{pmatrix} \vec{v}_1 \\ \vdots \\ \vec{v}_N \end{pmatrix} &= \begin{pmatrix} \vec{k}_1 \\ \vdots \\ \vec{k}_c \end{pmatrix} \\ J(\vec{r}) \cdot \vec{v} &= \vec{k}, \end{aligned}$$

where $\vec{k}$ is defined by blocks $\vec{k}_i$ and the matrix J by blocks J_{ij}, called the Jacobian matrices due to the constraints i and the particles j. Note that J_{ij} is zero iff the constraint i does not constrain (affect) the particle j. Also since $J_{ij} = \frac{\partial \vec{C}^v_i}{\partial \vec{v}_j} = \frac{\partial (\vec{C}_v)_i}{\partial \vec{v}_j}$, it follows that $J = \frac{\partial \vec{C}_v}{\partial \vec{v}}$ and so J is the Jacobian matrix of $\vec{C}_v$.

This shows that the global velocity-level constraint $\vec{C}_v = \vec{0}$, that captures the effects of multiple $\vec{C}^v_i = \vec{0}$ constraints, uses a "global" Jacobian matrix J and a right-hand side vector $\vec{k}$, that are built from the Jacobian matrices J_i and the right-hand side vectors $\vec{k}_i$ of the individual constraints — the individual constraints contribute to the slices of the global J and $\vec{k}$. The same can be said about the Jacobian matrix J and the right-hand side vector $\vec{c}$ of the corresponding global acceleration-level constraint $\vec{C}_a = \dot{\vec{C}}_v = \vec{0}$ and the individual acceleration-level constraints $\vec{C}^a_i = \dot{\vec{C}}^v_i = \vec{0}$. That is, $\vec{C}_a(\vec{r},\vec{v},\vec{a}) = (\vec{C}^a_1(\vec{r},\vec{v},\vec{a}),\ldots,\vec{C}^a_c(\vec{r},\vec{v},\vec{a})) = (\vec{0},\ldots,\vec{0}) = \vec{0}$, or equivalently

$$\begin{aligned}\begin{pmatrix} J_{11} & \ldots & J_{1N} \\ \vdots & \ddots & \vdots \\ J_{c1} & \ldots & J_{cN}\end{pmatrix} \cdot \begin{pmatrix}\vec{a}_1 \\ \vdots \\ \vec{a}_N\end{pmatrix} &= \begin{pmatrix}\vec{c}_1 \\ \vdots \\ \vec{c}_c\end{pmatrix} \\ J(\vec{r}) \cdot \vec{a} &= \vec{c}(\vec{r},\vec{v}). \end{aligned} \tag{4.7}$$

Note that the vector of multipliers $\vec{\lambda}$ for the global acceleration-level constraint can be interpreted as a block vector $\vec{\lambda} = (\vec{\lambda}_1,\ldots,\vec{\lambda}_c)$ so that $\vec{F}^c_i = J^T_i \cdot \vec{\lambda}_i$ is the constraint force due to the i-th constraint.

4.2.2 Abstraction

In this section we will describe the abstraction of individual constraints and a simulator component called the *constraint solver* that uses this abstraction to compute constraint forces.

Constraints will be abstracted as a special kind of force objects. The representation of constraint i will provide information on

- the dimensionality m_i of the constraint,
- the list $p_i(j)$ of $n_i < N$ constrained particles so that $p_i(j)$, $1 \leq j \leq n_i$, would return the index of the j-th constrained particle, and
- the formulation of the constraint on the acceleration level:
  - $m_i \times 3N$ Jacobian matrix J_i, represented by an ordered set of n_i *non-zero* blocks $(J_{i,p(1)},\ldots,J_{i,p(n)})$ of dimensions $m_i \times 3$ and
  - the right-hand side vector $\vec{c}_i$ of length m_i.

This is a general enough representation because it allows to implement position constraints, velocity-dependent constraints (with constant right-hand side vector), and even acceleration-dependent constraints (by supplying the $J_i \cdot \vec{a} = \vec{c}_i$ equation directly), and, as we saw in the previous section, combine the constraints automatically.

The global matrix J is *sparse* because each constraint i usually constrains only a small subset of all particles ($n_i \ll N$). Non-zero blocks $J_{i,p(j)}$ due to constraints and constrained particles are stored only. Parameter n_i is often fixed to a certain constant value ($n_i = 2$) so that $\vec{\lambda}$ multipliers can be computed efficiently using specialized algorithms that exploit the special structure of J.

To enforce all constraints at time t, constraint solver will have to build the global Jacobian matrix J and vector $\vec{c}$, that is to formulate the global acceleration-level constraint (4.7) from the descriptions of the individual constraints, formulate equation (4.6) for the global constraint, solve for global $\vec{\lambda}$ and evaluate and apply the constraint force (4.5). We will now describe some of these steps in detail:

- First the constraint solver indexes all active constraints and particles and determines the total number of DOFs d removed from the system ($d = \sum_{i=1}^{c} m_i$). The constraint and particle indices together with the list of particles constrained by a given constraint define the dimensions and layout of the global matrix J ($d \times 3N$) and vector $\vec{c}$ ($d \times 1$) — the order of row and column blocks. Constraint indices induce the assignment of the components of the global $\vec{\lambda}$ ($d \times 1$) to the scalar rows of the global matrix J (each scalar row corresponds to a DOF removed from the system, each block row corresponds to a constraint).

- Jacobian blocks and right-hand side vectors due to the individual constraints are gathered to the global matrix J and vector $\vec{c}$. Equation (4.6) is formed, obtaining the matrix A ($d \times d$) and vector $\vec{b}$ ($d \times 1$). Global multipliers $\vec{\lambda}$ are computed.
- Global multipliers $\vec{\lambda}$ and the global matrix J are used to compute the constraint force. The force is added to the force accumulators of the constrained particles.

Let us consider an example with four particles indexed $1, 2, 3, 4$ and two constraints. The first constraint has index 1, dimensionality $m_1 = 2$ and will constrain particles 1 and 2. The second constraint has index 2, dimensionality $m_2 = 3$ and will constrain particles $1, 3$ and 4. According to our abstraction and the assignment of indices to particles and constraints, the constraints can be represented by the following structure

Index	DOFs removed	Particles	Jacobian blocks	Vector $\vec{c}$
1	2	1, 2	J_{11}, J_{12}	$\vec{c}_1$
2	3	1, 3, 4	J_{21}, J_{23}, J_{24}	$\vec{c}_2$

and the global matrix J has this layout

$$J = \begin{pmatrix} J_{11} & J_{12} & 0 & 0 \\ J_{21} & 0 & J_{23} & J_{24} \end{pmatrix}.$$

4.3 Constraint Stabilization

Until now we assumed that the underlying ODE solver was absolutely precise and no integration error was gained. This is however not true and, regardless of the precision of the computed constraint force, particles will "drift" due to the accumulation of the local integration (ODE) error, failing to obey the constraint. The problem is highlighted because constraints are supposed to be maintained incrementally, under an assumption that the constraints hold at the beginning of each time step.

Consider a position-level constraint $\vec{C}_p = \vec{0}$. Since the constraint is maintained on the acceleration-level, integration error creeps into $\dot{\vec{C}}_p$ which results in the fact that $\dot{\vec{C}}_p \neq \vec{0}$ at the end of the time step and the error propagates to $\vec{C}_p$. The corresponding acceleration-level constraint $\ddot{\vec{C}}_p = \vec{0}$ however assumes that $\vec{C}_p = \dot{\vec{C}}_p = \vec{0}$ at each step.

Now, for an illustration, assume that we have a position-level constraint that constrains two particles to occupy the same world-space position. Thus we have $\vec{C}_p = \vec{r}_1 - \vec{r}_2 = \vec{0}$, where $\vec{r}_1$ and $\vec{r}_2$ are the positions of the two particles. By differentiating the constraint with respect to time once and twice respectively we get the velocity-level and acceleration-level equations $\dot{\vec{C}}_p = \vec{v}_1 - \vec{v}_2 = \vec{0}$ and $\ddot{\vec{C}}_p = \vec{a}_1 - \vec{a}_2 = \vec{0}$. When $\vec{C}_p \neq \vec{0}$ or $\dot{\vec{C}}_p \neq \vec{0}$ we would like to bring these quantities back to a zero vector, that is to *stabilize* the constraint. We can do that as follows. Instead of requiring that $\ddot{\vec{C}}_p = \vec{0}$, we might request the particles to accelerate against each other with a non-zero acceleration so that $||\dot{\vec{C}}_p||$ and $||\vec{C}_p||$ would decrease. Therefore we might require that $\ddot{\vec{C}}_p = -\alpha \cdot (\vec{r}_1 - \vec{r}_2) - \beta \cdot (\vec{v}_1 - \vec{v}_2)$, where α, β are small positive constants.

This technique can luckily be generalized for any position-level $\vec{C}_p$ or velocity-level $\vec{C}_v$ constraint by adding extra terms to the corresponding acceleration-level equation right-hand side vector $\vec{c}$.

Let us first look at the stabilization of position-level constraints. We have $\vec{C}_p = \vec{0}$ (measuring position error, relative particle positions), $\dot{\vec{C}}_p = J \cdot \vec{v} = \vec{0}$ (measuring velocity error, relative particle velocities) and $\ddot{\vec{C}}_p = J \cdot \vec{a} - \vec{c} = \vec{0}$, where $\vec{c} = -\dot{J} \cdot \vec{v}$ (measuring acceleration error, relative particle accelerations). In order to stabilize the constraint, we would like to affect the relative acceleration so as to decrease the magnitudes of the components of $\vec{C}_p$ as well as of $\dot{\vec{C}}_p$. We might introduce an energy function analogously to (3.7) and request the particles to accelerate against the energy gradient, which is given by J (the acceleration speed along the gradient would be proportional to the current constraint error). Since the acceleration-level equation (4.2) actually measures the relative acceleration along the gradient (the acceleration equals $J \cdot \vec{a} - \vec{c}$), we have just to modify its right-hand side vector $\vec{c}$,

$$\begin{aligned} J \cdot \vec{a} &= \vec{c}, \\ \vec{c} = -\dot{J} \cdot \vec{v} - \alpha \cdot \vec{C}_p - \beta \cdot \dot{\vec{C}}_p &= -\dot{J} \cdot \vec{v} - \alpha \cdot \vec{C}_p - \beta \cdot J \cdot \vec{v}. \end{aligned} \tag{4.8}$$

This is the final acceleration-level equation with stabilization terms that corresponds to the original $\vec{C}_p = \vec{0}$ constraint. Scalars α, β are small positive constants whose purpose is to "absorb" the integration error. They determine how fast the constraint error should be corrected. Bigger values indicate faster restoration, but really big values might cause the stabilization to "overshoot" or make the simulation unstable. If the values are set to zero, then no stabilization will take place and the modified equation will just turn to (4.2). Consult [12] or [22] for more information.

Velocity-level constraints (4.3) of the form $\vec{C}_v = J \cdot \vec{v} - \vec{k} = \vec{0}$, where $\vec{k}$ is a constant vector, are stabilized similarly. We have $\dot{\vec{C}}_v = J \cdot \vec{a} + \dot{J} \cdot \vec{v} = J \cdot \vec{a} - \vec{c} = \vec{0}$, where $\vec{c} = -\dot{J} \cdot \vec{v}$. In order to stabilize the constraint, we would like to affect the relative acceleration so as to decrease the magnitudes of the components of $\vec{C}_v = J \cdot \vec{v} - \vec{k}$. We will do so by requesting the particles to accelerate against the gradient of the energy function due to $\vec{C}_v$, which is again given by J, with a speed proportional to the current constraint error $J \cdot \vec{v} - \vec{k}$. We obtain,

$$\begin{aligned} J \cdot \vec{a} &= \vec{c}, \\ \vec{c} = -\dot{J} \cdot \vec{v} - \beta \cdot \vec{C}_v &= -\dot{J} \cdot \vec{v} - \beta \cdot (J \cdot \vec{v} - \vec{k}), \end{aligned} \tag{4.9}$$

where β is a small positive constant.

This stabilization method is called the *Baumgarte stabilization* and is quite popular because of its simplicity, almost no additional computational overhead and ability to incorporate stabilization directly to the evaluation stage of vector $\vec{c}$, [12]. Let us note that *generalized coordinates* approaches derive the equations of motion with respect to a reduced set of unconstrained coordinates, the constrained particles can not get into an invalid state and so no stabilization is required.

Chapter 5

Rigid Body Dynamics

In this chapter we will generalize constrained particle systems to rigid body structures, make the reader acquainted with the most important rigid body concepts and finally present equations of motion for rigid bodies in the form of ODE. That way we will be able to simulate rigid bodies in the same way how we simulated particle systems. Later on, constraints will be incorporated in the form of constraint forces.

We will proceed quite informally, because the matter would be rather involved otherwise and, according to our opinion, exact mathematical derivations would not offer much insight anyway. Although everything what needs be explained will be covered by these notes, introductory level information on rigid body dynamics can be found in [11]. The concepts will be defined according to [10].

5.1 Concepts

Rigid bodies will be intuitively treated as solid structures with fixed shapes that can translate and rotate in response to forces exerted on them. Unlike particles, rigid bodies have assigned volumes to them and their spatial states are augmented by the representation of the volume orientation. Since rigid bodies can not deform, their shapes and volumes can be defined in terms of local coordinate systems, specific to particular bodies, called the *body spaces*. Positions and orientations of bodies in the world space are then characterized by transformations (rotations followed by translations) that map points from the body spaces to the points in the world space.

Rigid bodies are often imagined as constrained particle systems consisting of a large (infinite) number of particles constrained to remain at the same relative positions in the body space, where constraint forces correspond to the spatial interaction forces that prevent the body from being deformed. Many rigid body concepts are based on this view and can be derived from it. Rigid body dynamics abstracts this kind of constrained particle system.

5.1.1 Mass Properties

In this section we will define certain rigid body properties that are derived from the distribution of mass over the body volume. These properties are constant throughout the simulation and, as we will see, determine body response to an exerted force. In this sense they serve the same purpose like particle masses in particle systems.

Given a rigid body, let us assume that $\rho : \mathbf{R}^3 \rightarrow \mathbf{R}^+$ is a *density function* that characterizes the body's mass distribution over the points $\vec{r}_b$ from the body space. The density function is non-zero (and positive) for points that form the body's volume and zero outside the volume.

Definition 9 *If $\rho(\vec{r}_b)$ is a density function of a rigid body, then*

$$\begin{aligned}
\textit{total mass } M &= \int \rho(\vec{r}_b)\, d\vec{r}_b \\
\textit{center of mass in body space } \vec{r}^{\,b}_{cm} &= \frac{\int \vec{r}_b \cdot \rho(\vec{r}_b)\, d\vec{r}_b}{M} \\
\textit{moment of inertia about x axis } I_{xx} &= \int ((r^b_y)^2 + (r^b_z)^2) \cdot \rho(\vec{r}_b)\, d\vec{r}_b \\
\textit{moment of inertia about y axis } I_{yy} &= \int ((r^b_x)^2 + (r^b_z)^2) \cdot \rho(\vec{r}_b)\, d\vec{r}_b \\
\textit{moment of inertia about z axis } I_{zz} &= \int ((r^b_x)^2 + (r^b_y)^2) \cdot \rho(\vec{r}_b)\, d\vec{r}_b \\
\textit{product of inertia about x, y axes } I_{xy} &= \int r^b_x \cdot r^b_y \cdot \rho(\vec{r}_b)\, d\vec{r}_b \\
\textit{product of inertia about x, z axes } I_{xz} &= \int r^b_x \cdot r^b_z \cdot \rho(\vec{r}_b)\, d\vec{r}_b \\
\textit{product of inertia about y, z axes } I_{yz} &= \int r^b_y \cdot r^b_z \cdot \rho(\vec{r}_b)\, d\vec{r}_b,
\end{aligned}$$

where moments of inertia I_{xx}, I_{yy}, I_{zz} and products of inertia I_{xy}, I_{xz}, I_{yz} are expressed relative to the origin $\vec{0}$ of the body space. All quantities are computed in the body space.

As we already stated, the volume of a rigid body can be imagined as a discrete set of particles with fixed positions in the body space. Assume that we have N such particles with positions $\vec{r}^{\,b}_i$ in the body space and masses m_i. Mass properties can be then approximated by finite sums as

$$\begin{aligned}
M &= \sum_{i=1}^{N} m_i \qquad \vec{r}^{\,b}_{cm} = \frac{\sum_{i=1}^{N} m_i \cdot \vec{r}^{\,b}_i}{M} \qquad I_{xx} = \sum_{i=1}^{N} m_i \cdot ((\vec{r}^{\,b}_i)^2_y + (\vec{r}^{\,b}_i)^2_z) \\
I_{yy} &= \sum_{i=1}^{N} m_i \cdot ((\vec{r}^{\,b}_i)^2_x + (\vec{r}^{\,b}_i)^2_z) \qquad I_{zz} = \sum_{i=1}^{N} m_i \cdot ((\vec{r}^{\,b}_i)^2_x + (\vec{r}^{\,b}_i)^2_y) \\
I_{xy} &= \sum_{i=1}^{N} m_i \cdot (\vec{r}^{\,b}_i)_x \cdot (\vec{r}^{\,b}_i)_y \qquad I_{xz} = \sum_{i=1}^{N} m_i \cdot (\vec{r}^{\,b}_i)_x \cdot (\vec{r}^{\,b}_i)_z \\
I_{yz} &= \sum_{i=1}^{N} m_i \cdot (\vec{r}^{\,b}_i)_y \cdot (\vec{r}^{\,b}_i)_z .
\end{aligned}$$

Convention 3 *In order to simplify further derivations we will assume that the body space coordinates are designed so that the rigid body's center of mass lies at the origin of the body space, that is, $\vec{r}^{\,b}_{cm} = \vec{0}$.*

5.1.2 Position and Orientation

When we were working with particles it was sufficient to use a single vector $\vec{r}(t)$ to locate the particle in the world space. However for rigid bodies this is no longer true because rigid bodies have volumes assigned to them. In order to locate a rigid body in the world space, a world space position of its "reference point", chosen arbitrarily but fixed in the body space, and an orientation of body space axes in the world space must be given. We will formalize these concepts now a little.

By orientation of a rigid body we will mean a *rotation* (linear transformation) that maps the body space axes, attached to the rigid body, to world space vectors. Directions of the mapped vectors define the orientation of the body.

Convention 4 *We will always use* right-handed coordinate systems *and measure angles in a* counter-clockwise *way. If we are given a coordinate system with axes* $(\vec{x}, \vec{y}, \vec{z})$*, where* $\vec{x}$ *axis points to the* right*,* $\vec{y}$ *axis points* forward *and* $\vec{z}$ *axis points* up*, then the* $\vec{x}$ *axis will align to the* $\vec{y}$ *axis if the* $\vec{x}$ *axis is rotated by* $+\pi/2$ *about the* $\vec{z}$ *axis.*

Imagine that you grabbed the rotation axis in your right hand, so that the thumb would point in the direction of the axis. Then the direction of the fingers would determine the direction the positive angles of counter-clockwise rotations are measured along.

There are many ways to represent rotations — Euler angles, rotation matrices, quaternions and axis & angle pairs, to name some, [1], [15]. Although some representations proved to be extremely useful and better than another, in terms of both algebraic properties and computational efficiency, they were often hard to understand and certainly not intuitive. We can not expect the reader to be acquainted with the theory of quaternions, for example. On the other hand, the theory of rotation matrices seems to be well-known widely and yet pretty intuitive. Therefore we will use rotation matrices first with a note that we will later switch to quaternions.

Rigid body's center of mass (at the origin of the body space) will be used as the "reference point" whose position is tracked in the world space. As we will see, it will simplify the transformations of points and vectors from the body space to world space and allow to interpret body space axes vectors as vectors attached to the rigid body's center of mass and formulate the equations of motion in such a way, that the rotational part, describing the rotational effects due to an exerted force, can be separated from the linear part, describing the translational effects.

We will conclude this discussion by the following definition.

Definition 10 *The position and orientation of a rigid body in the world space is determined by an affine transformation that maps points from the body space to points in the world space.*

The transformation is defined by a world space position $\vec{x}(t)$ *of the rigid body's center of mass and a rotation* $R(t)$*, represented by a* 3×3 *rotation matrix, that maps vectors from the body space to vectors in the world space.*

If $\vec{p}_b$ *is a position of a point in the body space then the world space position* $\vec{p}(t)$ *of that point is given by*

$$\vec{p}(t) = R(t) \cdot \vec{p}_b + \vec{x}(t) \tag{5.1}$$

and the position $\vec{x}(t)$ *of the center of mass and rotation* $R(t)$ *of the body space are said to represent the position and orientation of the rigid body.*

Let us note that if $\vec{r}_x(t), \vec{r}_y(t), \vec{r}_z(t)$ are body space axes expressed in the world coordinates then the rotation $R(t)$, that represents the orientation of the corresponding body, can be written by blocks as

$$R(t) = \begin{pmatrix} \vec{r}_x(t) & \vec{r}_y(t) & \vec{r}_z(t) \end{pmatrix}.$$

5.1.3 Linear and Angular Velocity

The motion of particles, from the point of kinematics, is characterized by position vectors $\vec{r}(t)$ and their time derivatives — particles can undergo translation only. Contrary, the motion of rigid bodies is characterized by position vectors $\vec{x}(t)$, rotation matrices $R(t)$ and their time derivatives

— rigid bodies can translate and rotate. Naturally, we will be interested in how $\vec{x}(t)$ and $R(t)$ change over time.

First we define the *linear velocity* $\vec{v}(t)$ of a rigid body that characterizes the direction and speed of the body translation. If $\vec{x}(t)$ is the position of the rigid body's center of mass in the world space, then $\dot{\vec{x}}(t)$ is the velocity of an imaginary particle at the body's center of mass and this velocity is called the linear velocity,

$$\vec{v}(t) = \dot{\vec{x}}(t). \tag{5.2}$$

Although the rigid body motion can be arbitrarily complex, it can always be seen as a motion of the rigid body's center of mass and a rotation about a time varying axis that *passes through the center of mass.* This allows to define a concept of *angular velocity.*

Assume for a while that the body's center of mass is fixed to a certain position and so the body can rotate about an axis passing through the center of mass only. Angular velocity $\vec{\omega}(t)$ describes what the rotation axis and the rotation speed about the axis at the instant t is. Angular velocity is a limiting value, like linear velocity, and as such should be treated.

Rotation R of a rigid body can be factored to three rotations $R_x(\theta_x), R_y(\theta_y), R_z(\theta_z)$, where $R_x(\theta_x)$ represents the rotation about the world space axis X by an angle θ_x (analogously for R_y, R_z), so that $R = R_z \circ R_y \circ R_x$ (factors are composed in the X-Y-Z order). The angles $\theta_x, \theta_y, \theta_z$ are called the Euler angles about the fixed world space axes and form a vector $\vec{\theta} = (\theta_x, \theta_y, \theta_z)$. Angular velocity $\vec{\omega}(t)$ is then defined as

$$\vec{\omega}(t) = \dot{\vec{\theta}}(t) = \frac{\partial}{\partial t}\vec{\theta}(t) = (\frac{\partial}{\partial t}\theta_x(t), \frac{\partial}{\partial t}\theta_y(t), \frac{\partial}{\partial t}\theta_z(t)), \tag{5.3}$$

according to [13].

The direction of $\vec{\omega}(t)$ defines the instantaneous rotation axis, expressed in the world space coordinates, and the magnitude defines the instantaneous (counter-clockwise) rotation speed, that is $\vec{\omega}(t) = \overrightarrow{axis} \cdot speed$ in popular terms. If the body was rotating uniformly, $\vec{\omega}(t) = \overrightarrow{const}$, then $\|\vec{\omega}(t)\| = 2\pi \cdot f = \varphi$, where f (frequency) is the number of spins that the body would take per second and φ is the angle that the body would rotate by per second. $\|\vec{\omega}(t)\|$ is measured in $rad \cdot s^{-1}$.

Linear velocity $\vec{v}(t)$ and angular velocity $\vec{\omega}(t)$ of a rigid body can be used to characterize the velocity of an arbitrary point on the body. If we are given a world space position $\vec{p}(t)$ of such a point then

$$\begin{aligned} \vec{p}(t) &= \vec{x}(t) + \vec{r}(t) \\ \dot{\vec{p}}(t) &= \frac{\partial}{\partial t}\vec{x}(t) + \frac{\partial}{\partial t}\vec{r}(t) = \vec{v}(t) + \vec{\omega}(t) \times \vec{r}(t) \\ &= \vec{v}(t) + \vec{\omega}(t) \times (\vec{p}(t) - \vec{x}(t)), \end{aligned} \tag{5.4}$$

where $\dot{\vec{p}}(t)$ is the velocity of $\vec{p}(t)$, $\vec{r}(t) = \vec{p}(t) - \vec{x}(t)$ is a world space vector attached to the body's center of mass and $\dot{\vec{r}}(t) = \vec{\omega}(t) \times \vec{r}(t)$ is its time derivative[1]. As can be seen, it consists of two terms, a term due to the body translation and a term due to the rotation. Clearly, if the body was not rotating, then $\vec{\omega}(t) = \vec{0}$ and the point velocity would equal the velocity of the body's center of mass. If it was rotating, an extra term $\vec{\omega}(t) \times \vec{r}(t)$ due to the rotation would have to be added.

Let us focus at this term. If the linear velocity was $\vec{0}$ and angular velocity was constant then trajectory of the point $\vec{p}(t)$ would correspond to a circle with the center at $\vec{x}(t) + \|\vec{r}(t)\| \cdot \cos(\alpha) \cdot \vec{\omega}(t)$ and $\|\vec{r}(t)\| \cdot \sin(\alpha)$ radius, where α is the angle between $\vec{\omega}(t)$ and $\vec{r}(t)$. Then $\|\dot{\vec{p}}(t)\|$ would have to equal $\|\vec{\omega}(t)\| \cdot \|\vec{r}(t)\| \cdot \sin(\alpha)$

[1]This validity of this formula is verified in the next paragraph.

(this is the length of the circle arc swept by $\vec{p}(t)$ per second, hence the velocity magnitude). The direction of $\dot{\vec{p}}(t)$ would have to be perpendicular to both $\vec{\omega}(t)$ and $\vec{r}(t)$ and due to the counter-clockwise convention would have to equal $\frac{\vec{\omega}(t)\times\vec{r}(t)}{\|\vec{\omega}(t)\times\vec{r}(t)\|}$. Therefore, combining the two conditions on the velocity magnitude and the direction, $\dot{\vec{p}}(t) = (\|\vec{\omega}(t)\| \cdot \|\vec{r}(t)\| \cdot \sin(\alpha)) \cdot \frac{\vec{\omega}(t)\times\vec{r}(t)}{\|\vec{\omega}(t)\times\vec{r}(t)\|} = \vec{\omega}(t) \times \vec{r}(t)$. If $\vec{v}(t) \neq \vec{0}$ and angular velocity was still constant, then the trajectory of $\vec{p}(t)$ would no longer correspond to a circle, but the trajectory of $\vec{r}(t)$ would. Hence we could write that $\dot{\vec{r}}(t) = \vec{\omega}(t) \times \vec{r}(t)$. Finally, noting that the appropriate magnitude of $\dot{\vec{r}}(t)$ was derived from the instantaneous $\vec{\omega}(t)$, we can conclude that this equation holds even if the angular velocity is not constant.

In the previous paragraph we have shown that $\dot{\vec{r}}(t) = \vec{\omega}(t) \times \vec{r}(t)$ for vectors $\vec{r}(t)$ attached to the rigid body's center of mass. This time, we will assert that the same equation holds for vectors $\vec{r}(t)$ attached to arbitrary points on the rigid body. If $\vec{r}(t)$ is a world space vector attached to a world space point $\vec{p}(t)$ of a rigid body, then $\vec{a}(t) = \vec{p}(t)$ and $\vec{b}(t) = \vec{p}(t) + \vec{r}(t)$ are the world space positions of the origin and tip of $\vec{r}(t)$, $\vec{r}(t) = \vec{b}(t) - \vec{a}(t)$ and $\dot{\vec{r}}(t)$ equals

$$\dot{\vec{r}}(t) = \frac{\partial}{\partial t}\vec{b}(t) - \frac{\partial}{\partial t}\vec{a}(t) = \vec{\omega}(t) \times \vec{r}(t), \tag{5.5}$$

as can be verified by evaluating equation (5.4) for points $\vec{a}(t)$ and $\vec{b}(t)$.

We will now use this equation to derive a formula for $\frac{\partial}{\partial t}R(t) = \dot{R}(t)$. $R(t)$ can be written in a block fashion as $R(t) = (\vec{r}_x(t),\ \vec{r}_y(t),\ \vec{r}_z(t))$ and so $\dot{R}(t) = (\dot{\vec{r}}_x(t), \dot{\vec{r}}_y(t), \dot{\vec{r}}_z(t))$. Since $\vec{r}_x(t), \vec{r}_y(t), \vec{r}_z(t)$ are the body space axes expressed in the world coordinates and the axes are attached to the rigid body's center of mass, we get that $\dot{R}(t) = (\vec{\omega}(t)\times\vec{r}_x(t), \vec{\omega}(t)\times\vec{r}_y(t), \vec{\omega}(t)\times\vec{r}_z(t))$. If we define matrix $\vec{a}^*$ as

$$\vec{a}^* = \begin{pmatrix} 0 & -a_z & a_y \\ a_z & 0 & -a_x \\ -a_y & a_x & 0 \end{pmatrix}$$

then $\vec{a} \times \vec{b}$ can be expressed in the form of matrix multiplication as $\vec{a} \times \vec{b} = \vec{a}^* \cdot \vec{b}$, therefore $\dot{R}(t) = (\vec{\omega}(t)^* \cdot \vec{r}_x(t),\ \vec{\omega}(t)^* \cdot \vec{r}_y(t),\ \vec{\omega}(t)^* \cdot \vec{r}_z(t))$, finally obtaining

$$\dot{R}(t) = \vec{\omega}(t)^* \cdot R(t). \tag{5.6}$$

5.1.4 Force and Torque

When we were working with particle systems we tracked directions and magnitudes of forces exerted on the particles ($\vec{F}_i^{total}(t)$ vectors), but we did not care about the points of application, because they always coincided with the particle positions. The only effect a force exerted on a particle could produce was a translation. However since rigid bodies have volumes and can rotate, we need know not only what the direction and magnitude of the exerted force is, but also the point of its application so that both body translation and rotation could be affected by the force.

We will conceptually work with a rigid body that consists of N particles constrained to remain at the same relative positions in the body space. Particle positions in the body space will be fixed and characterized by vectors $\vec{r}_i^b, 1 \leq i \leq N$, their world space positions will be given by $\vec{r}_i(t) = R(t) \cdot \vec{r}_i^b + \vec{x}(t)$.

To simplify further derivations we will assume that the points of force application coincide with the positions of particles[2] the rigid body is made of. This will allow to characterize the force distribution over the volume of the body by a set of world space vectors $\vec{F}_i^{total}(t)$, where $\vec{F}_i^{total}(t)$ denotes the total force exerted at the rigid body's particle i, at the world space position $\vec{r}_i(t)$.

[2]We can always think that there happens to be a particle at the given world space position so that forces could act at arbitrary points in the world space.

If $\vec{F}_i(t)$ is a force acting on particle i, then the *torque* $\vec{\tau}_i(t)$ on particle i due to $\vec{F}_i(t)$ is defined as

$$\vec{\tau}_i(t) = (\vec{r}_i(t) - \vec{x}(t)) \times \vec{F}_i(t). \tag{5.7}$$

Intuitively torque can be imagined as a scale of the angular velocity that the rigid body would gain if $\vec{F}_i(t)$ was the only force acting on the body and the force was exerted at $\vec{r}_i(t)$. Torque is expressed in the world coordinates.

If $\vec{F}_i^{total}(t) = \vec{F}_i^1(t) + \ldots + \vec{F}_i^n(t)$ is the total force acting on particle i, then the *total torque* $\vec{\tau}_i^{total}(t)$ on particle i is defined as

$$\vec{\tau}_i^{total}(t) = \vec{\tau}_i^1(t) + \ldots + \vec{\tau}_i^n(t) = (\vec{r}_i(t) - \vec{x}(t)) \times \vec{F}_i^{total}(t), \tag{5.8}$$

where $\vec{\tau}_i^j(t)$ is the torque due to $\vec{F}_i^j(t)$.

By summing the total forces and torques acting on the particles that the rigid body is made of, we get the total force $\vec{F}_{total}(t)$ and total torque $\vec{\tau}_{total}(t)$ acting on the rigid body

$$\begin{aligned} \vec{F}_{total}(t) &= \sum_{i=1}^{N} \vec{F}_i^{total}(t) \\ \vec{\tau}_{total}(t) &= \sum_{i=1}^{N} \vec{\tau}_i^{total}(t). \end{aligned} \tag{5.9}$$

From the Newton's Third Law we get that $\vec{F}_{total}(t)$ and $\vec{\tau}_{total}(t)$ are the *total external force* and the *total external torque* exerted on the rigid body, because $\vec{F}_i^j(t)$ and $\vec{\tau}_i^j(t)$ terms due to internal forces (particle interaction) cancel each other and thus do not contribute to $\vec{F}_{total}(t)$ and $\vec{\tau}_{total}(t)$.

The total external force and the total external torque together with the body's mass properties determine the rigid body motion. The total external force captures the translational (*linear*) effects of the exerted force, the total external torque captures the rotational (*angular*) effects.

5.1.5 Linear and Angular Momentum

We will define the concept of a *linear* and an *angular* momentum which describe the translational and rotational (kinetic) state of a rigid body and allow to express the rigid body's equations of motion "nicely". As the names suggest linear momentum will deal with linear (translational) body properties, while angular momentum will deal with angular (rotational) properties. We still work with a rigid body model that treats the body as a set of N particles fixed in the body space.

Linear Momentum

The *linear momentum of particle* i is defined as $\vec{P}_i(t) = m_i \cdot \dot{\vec{r}}_i(t)$, where m_i is the mass of particle i (this corresponds to (3.1)) and the *linear momentum of the rigid body* $\vec{P}(t)$ is the sum of the linear momentums of the individual particles. Linear momentum describes the rigid body state with respect to the translational motion of the body's center of mass and it can be shown that

$$\vec{P} = \sum_{i=1}^{N} \vec{P}_i(t) = M \cdot \vec{v}(t), \tag{5.10}$$

where $M = \sum_{i=1}^{N} m_i$ is the mass of the rigid body and $\vec{v}(t)$ is the velocity of its center of mass. The time derivative of the linear momentum equals the total external force $\vec{F}_{total}(t)$ applied on the body

$$\dot{\vec{P}}(t) = \vec{F}_{total}(t). \tag{5.11}$$

Let us follow the derivation from [13]. From the definition of the linear momentum for particles and the rigid body we get $\vec{P}(t) = \sum_{i=1}^{N} m_i \cdot \dot{\vec{r}}_i(t)$. This can be rewritten according to (5.4) as $\vec{P}(t) = \sum_{i=1}^{N} m_i \cdot (\vec{v}(t) + \vec{\omega}(t) \times (\vec{r}_i(t) - \vec{x}(t))) = \sum_{i=1}^{N} m_i \cdot \vec{v}(t) + \vec{\omega}(t) \times \sum_{i=1}^{N} m_i \cdot (\vec{r}_i(t) - \vec{x}(t)) = (\sum_{i=1}^{N} m_i) \cdot \vec{v}(t) + \vec{\omega}(t) \times \vec{0} = M \cdot \vec{v}(t)$ because $\sum_{i=1}^{N} m_i \cdot (\vec{r}_i(t) - \vec{x}(t)) = \sum_{i=1}^{N} m_i \cdot R(t)\vec{r}_i^{\,b}(t) = R(t) \cdot \left(\frac{\sum_{i=1}^{N} m_i \cdot \vec{r}_i^{\,b}(t)}{M} \right) \cdot M = R(t) \cdot \vec{r}_{cm}^{\,b} \cdot M = \vec{0}$.

From the Newton's Second Law we have $\vec{F}_i^{total}(t) = m_i \cdot \ddot{\vec{r}}_i(t) = m_i \cdot \vec{a}_i(t)$ for any particle i and therefore $\vec{F}_{total}(t) = \sum_{i=1}^{N} \vec{F}_i^{total}(t) = \sum_{i=1}^{N} m_i \cdot \vec{a}_i(t)$. Since $\vec{x}(t) = \frac{\sum_{i=1}^{N} m_i \cdot \vec{r}_i(t)}{M}$, we get $\ddot{\vec{x}}(t) = \dot{\vec{v}}(t) = \frac{\sum_{i=1}^{N} m_i \cdot \vec{a}_i(t)}{M}$, finally obtaining $\vec{F}_{total} = M \cdot \dot{\vec{v}}(t) = \dot{P}(t)$.

Equations (5.10) and (5.11) describe the effect of exerted external force on the translational motion of the rigid body and "mimic" the Newton's Second Law for particles — this effect is the same as if the rigid body was replaced by a single particle with mass M, which $\vec{F}_{total}(t)$ acted on. Note that the external torque $\vec{\tau}_{total}(t)$ has no effect on the translational motion of the body. To describe the rotational effects we will define the concept of angular momentum.

Angular Momentum

The *angular momentum of particle i* is defined as $\vec{L}_i(t) = (\vec{r}_i(t) - \vec{x}(t)) \times (m_i \cdot \dot{\vec{r}}_i(t) - m_i \cdot \dot{\vec{x}}(t))$ and the *angular momentum of the rigid body* $\vec{L}(t)$ as the sum of angular momentums due to individual particles. Angular momentum describes the rigid body state with respect to the rotational motion of the body and it can be shown that

$$\vec{L} = \sum_{i=1}^{N} \vec{L}_i(t) = I(t) \cdot \vec{\omega}(t), \tag{5.12}$$

where $I(t)$ is a 3×3 positive definite matrix called the *world space inertia tensor*, which characterizes the distribution of mass over the body, relative to the body's center of mass, along the world space axes. $I(t)$ is independent of the body position $\vec{x}(t)$, but it depends on the orientation $R(t)$. Let us define I_{body} as

$$I_{body} = \begin{pmatrix} I_{xx} & -I_{xy} & -I_{xz} \\ -I_{xy} & I_{yy} & -I_{yz} \\ -I_{xz} & -I_{yz} & I_{zz} \end{pmatrix}, \tag{5.13}$$

where $I_{xx}, I_{yy}, I_{zz}, I_{xy}, I_{xz}, I_{yz}$ are the moments and products of inertia distilled from the rigid body's density function $\rho(\vec{r}_b)$. I_{body} is called the *body space inertia tensor*, it is evaluated in the body space and equals the world space inertia tensor $I(t)$ if the current body's orientation $R(t)$ is an identity matrix (I_{body} characterizes the distribution of mass over the body, relative to the body's center of mass, along the body space axes). It can be shown that the relation between the world space and body space tensors is given by

$$\begin{aligned} I(t) &= R(t) \cdot I_{body} \cdot R(t)^T \\ I^{-1}(t) &= R(t) \cdot I_{body}^{-1} \cdot R(t)^T, \end{aligned} \tag{5.14}$$

which provides us with a formula for the evaluation of $I(t)$ and $I^{-1}(t)$ for arbitrary body orientation because I_{body} is a constant matrix, [10]. Note that I_{body} is a positive definite matrix and so I_{body}^{-1}, $I(t)$ and $I^{-1}(t)$ are also positive definite for arbitrary orientation $R(t)$[3].

Since the body's angular momentum describes the body's kinetic state with respect to its rotational motion, we are naturally interested in what its time derivative is and what affects its

[3]This special property is utilized by the constraint solver to compute constraint forces more efficiently.

value. Erleben [13] shows that the time derivative of the angular momentum equals the total external torque $\vec{\tau}_{total}(t)$ exerted on the body,

$$\dot{L}(t) = \vec{\tau}_{total}(t), \tag{5.15}$$

which matches the relation between the linear momentum and the total external force.

Finally, let us present the following equation that is directly derived from (5.12) and might become handy during simulation

$$\vec{\omega}(t) \;=\; I^{-1}(t) \cdot \vec{L}(t). \tag{5.16}$$

Erleben [13] proceeds as follows to define the angular momentum. So far we have defined a torque with respect to the rigid body's center of mass. The total torque $\vec{\tau}_i^{total'}(t)$ on particle i with respect to the world space origin due to force $\vec{F}_i^{total}(t)$ is defined as $\vec{\tau}_i^{total'}(t) = \vec{r}_i(t) \times \vec{F}_i^{total}(t)$, where $\vec{r}_i(t)$ is the world space position of particle i. The angular momentum $\vec{L}_i'(t)$ of particle i with respect to the world space origin is then defined as $\vec{L}_i'(t) = \vec{r}_i(t) \times \vec{P}_i(t)$. Then

$$\begin{aligned}
\frac{\partial \vec{P}_i(t)}{\partial t} &= \vec{F}_i^{total}(t) \\
\frac{\partial \vec{L}_i'(t)}{\partial t} &= \frac{\partial}{\partial t}\vec{r}_i(t) \times \vec{P}_i(t) + \vec{r}_i(t) \times \frac{\partial}{\partial t}\vec{P}_i(t) \\
&= \dot{\vec{r}}_i(t) \times m_i \cdot \dot{\vec{r}}_i(t) + \vec{r}_i(t) \times \vec{F}_i^{total}(t) \\
&= \vec{0} + \vec{r}_i(t) \times \vec{F}_i^{total}(t) = \vec{\tau}_i^{total'}(t),
\end{aligned}$$

where the first equation is the well-known Newton's Second Law, relating the time derivative of linear momentum to a force, and the second equation is its "rotational version" called the Euler's Equation, relating the time derivative of angular momentum with respect to the world space origin to a torque with respect to the origin.

By summing the individual angular momentums and torques, with respect to the origin, due to particles of the rigid body, one would get the angular momentum of the rigid body and the total external torque with respect to the origin. These could be used to formulate the Euler's Equation for the rigid body in the form of $\frac{\partial}{\partial t}\vec{L}'(t) = \vec{\tau}_{total}'(t)$.

Unfortunately, the angular momentum $\vec{L}'$ of the rigid body defined in this way is hard to compute and it is better to define it differently, with respect to the rigid body's center of mass

$$\begin{aligned}
\vec{L} &= \sum_{i=1}^{N} \vec{L}_i(t), \\
\vec{L}_i(t) &= (\vec{r}_i(t) - \vec{x}(t)) \times (m_i \cdot \dot{\vec{r}}_i(t) - m_i \cdot \dot{\vec{x}}(t)).
\end{aligned}$$

Generally $\frac{\partial}{\partial t}\vec{L}_i(t) \neq \vec{\tau}_i^{total}(t)$, unlike $\frac{\partial}{\partial t}\vec{L}_i'(t) \neq \vec{\tau}_i^{total'}(t)$, but the Euler's Equation for the rigid body still holds and $\vec{L}$ is easy to compute and update, [13].

Equations (5.12) and (5.15) describe the effect of exerted external torque on the rotational motion of the rigid body. Note the correspondence to (5.10) and (5.11) and the fact that the external force $\vec{F}_{total}(t)$ has no effect on the rotational motion of the body, likewise the external torque $\vec{\tau}_{total}(t)$ has no effect on the translational motion.

5.2 Equations of Motion

We will summarize the rigid body concepts from the previous sections in the form of equations of motion for a rigid body and a set of bodies. The equations will be first order ODEs and hence the numerical solving of the equations would correspond to the simulation of the bodies. Two formulations will be presented — a formulation with linear and angular momentums and a formulation with linear and angular velocities.

5.2.1 Momentum Form

The formulation with linear and angular momentums treats the momentums as a part of the rigid body state. To complete the state, rigid body position and orientation must be added to the linear and angular momentum. Therefore the rigid body state will be described by a state (block) vector $\vec{y}(t)$,

$$\vec{y}(t) = (\vec{x}(t), R(t), \vec{P}(t), \vec{L}(t)), \tag{5.17}$$

where $\vec{x}(t)$ is the position of the rigid body's center of mass, $R(t)$ is the orientation of the body, $\vec{P}(t)$ is its linear momentum and $\vec{L}(t)$ the angular momentum. The state vector describes the rigid body state from the point of both kinematics (position and orientation) and dynamics (linear and angular momentums).

From equations (5.2), (5.6), (5.11), (5.15), (5.14) and (5.16) we get the *equation of motion for a rigid body in the momentum form*,

$$\frac{\partial}{\partial t}\vec{y}(t) = (\vec{v}(t), \vec{\omega}(t)^* \cdot R(t), \vec{F}_{total}(t), \vec{\tau}_{total}(t)), \tag{5.18}$$

where $\vec{F}_{total}(t)$ and $\vec{\tau}_{total}(t)$ are the total external force and torque exerted on the body and $\vec{v}(t)$ and $\vec{\omega}(t)$ are auxiliary quantities derived from the rigid body state $\vec{y}(t)$. These quantities, alongside with the additional auxiliary quantities $I(t)$ and $I^{-1}(t)$, are given by the following equations

$$\begin{aligned} \vec{v}(t) &= \frac{\vec{P}(t)}{M} \\ I(t) &= R(t) \cdot I_{body} \cdot R(t)^T \\ I^{-1}(t) &= R(t) \cdot I_{body}^{-1} \cdot R(t)^T \\ \vec{\omega}(t) &= I^{-1}(t) \cdot \vec{L}(t). \end{aligned}$$

Quantities $I_{body}, I_{body}^{-1}, M$ and M^{-1} characterize the rigid body mass properties and are constant throughout the simulation. The equation is a first order ODE and is expressed in a block form.

If we are given n rigid bodies with state vectors $\vec{y}_1(t), \ldots, \vec{y}_n(t)$ concatenated into a block vector $\vec{y}(t) = (\vec{y}_1(t), \ldots, \vec{y}_n(t))$, then the equation of motion for the set of bodies is given by $\frac{\partial}{\partial t}\vec{y}(t) = (\frac{\partial}{\partial t}\vec{y}_1(t), \ldots, \frac{\partial}{\partial t}\vec{y}_n(t))$.

5.2.2 Velocity Form

The equations of motion can be reformulated so that the linear and angular velocity are used as the state variables instead of the linear and angular momentum (position and orientation is tracked in the same way), yielding an equation that more resembles the equation of motion for a particle. The alternative state vector $\vec{y}(t)$ is defined as

$$\vec{y}(t) = (\vec{x}(t), R(t), \vec{v}(t), \vec{\omega}(t)), \tag{5.19}$$

where $\vec{x}(t)$ is the position of the rigid body's center of mass, $R(t)$ is the orientation of the body, $\vec{v}(t)$ is its linear velocity and $\vec{\omega}(t)$ angular velocity.

We will define the *linear acceleration* of a rigid body as the acceleration of the rigid body's center of mass, that is

$$\vec{a}(t) = \dot{\vec{v}}(t) = \ddot{\vec{x}}(t) = M^{-1} \cdot \vec{F}_{total}(t). \tag{5.20}$$

By differentiating equation (5.16) $\vec{\omega}(t) = I^{-1}(t) \cdot L(t)$ with respect to time, utilizing (5.14) and following the lengthy derivation from [10][4], we get the *angular acceleration* $\vec{\alpha}(t)$ of the body,

$$\begin{aligned} \vec{\alpha}(t) &= \dot{\vec{\omega}}(t) = \frac{\partial}{\partial t} I^{-1}(t) \cdot L(t) = \ \ldots \\ &= I^{-1}(t) \cdot (\vec{L}(t) \times \vec{\omega}(t) + \vec{\tau}_{total}(t)) \\ &= I^{-1}(t) \cdot ((I(t) \cdot \vec{\omega}(t)) \times \omega(t) + \vec{\tau}_{total}(t)) \\ &= I^{-1}(t) \cdot (\vec{\tau}_{coriolis}(t) + \vec{\tau}_{total}(t)), \end{aligned} \tag{5.21}$$

where $\vec{\tau}_{coriolis}(t) = \vec{L}(t) \times \vec{\omega}(t) = (I(t) \cdot \vec{\omega}(t)) \times \vec{\omega}(t)$ is a coriolis torque due to body rotation. The coriolis torque is an inertial velocity-dependent torque that exists "by itself" and can be imagined and treated as an implicit component of the total external torque $\vec{\tau}_{total}(t)$, [12]. However, to avoid confusion it is better to keep $\vec{\tau}_{total}(t)$ and $\vec{\tau}_{coriolis}(t)$ separated — $\vec{\tau}_{total}(t)$ is the total external torque due to explicit external forces, while $\vec{\tau}_{coriolis}(t)$ is the implicit inertial torque due to the body rotation[5]. This separation allows to switch between the two formulations of equations of motion as required. The momentum-based variant is not aware of any coriolis torque and it just happens that the coriolis torque is hidden by the concept of the angular momentum.

Having defined the linear and angular acceleration, we can define the *equation of motion for a rigid body in the velocity form* as

$$\frac{\partial}{\partial t}\vec{y}(t) = (\vec{v}(t), \vec{\omega}(t)^* \cdot R(t), M^{-1} \cdot \vec{F}_{total}(t), I^{-1}(t) \cdot (\vec{\tau}_{coriolis}(t) + \vec{\tau}_{total}(t)), \tag{5.22}$$

where $\vec{F}_{total}(t)$ and $\vec{\tau}_{total}(t)$ are the total explicit external force and torque, $\vec{\tau}_{coriolis}(t)$ is the coriolis torque defined in (5.21) and M^{-1} and $I^{-1}(t)$ are defined as in (5.18), noting that the equation of motion for a set of n bodies is obtained by $n\times$ "cloning" the equation for a single body.

We will use the momentum form of the equation of motion to simulate the motion of rigid bodies and the velocity form to formulate and derive constraint forces in the next chapter.

5.2.3 Mass Matrices

In this section we will elaborate on the velocity-form of the equation of motion, define the notation of *generalized velocities and forces* and the concept of mass matrices for rigid bodies, which will allow to treat rigid bodies as a kind of particles moving in $\mathbf{R}^6$, thus simplifying many equations, [9].

If we are given a rigid body with a linear velocity $\vec{v}(t)$ and an angular velocity $\vec{\omega}(t)$, then the *generalized velocity* $\vec{v}_{gen}(t)$ of the body is a 6×1 block vector $\vec{v}_{gen}(t) = (\vec{v}(t), \vec{\omega}(t))$, comprising of two 3×1 blocks due to the linear and angular velocity.

Similarly if $\vec{F}_{total}(t)$ and $\vec{\tau}_{total}(t)$ are the total external force and the total external torque acting on the body, then the *generalized external force* $\vec{F}^{total}_{gen}(t)$ acting on the body is a 6×1 block vector $\vec{F}^{total}_{gen}(t) = (\vec{F}_{total}(t), \vec{\tau}_{total}(t))$, comprising of two 3×1 blocks due to the force and torque.

In general, we will call any block vector consisting of a block due to a linear quantity and a block due to the corresponding angular quantity a generalized quantity. That way, we can define the generalized acceleration $\dot{\vec{v}}_{gen}(t)$, for example.

[4]Which we cowardly omit.

[5]It follows from this the angular acceleration $\vec{\alpha}(t)$ can change even if $\vec{\tau}_{total}(t) = \vec{0}$.

If m and $I(t)$ are the mass[6] and world space inertia tensor of the rigid body, then the *mass matrix of the rigid body* $M(t)$ is a 6×6 block diagonal matrix defined as

$$M(t) = \begin{pmatrix} m \cdot E & 0 \\ 0 & I(t) \end{pmatrix} = \begin{pmatrix} m & 0 & 0 & 0 & 0 & 0 \\ 0 & m & 0 & 0 & 0 & 0 \\ 0 & 0 & m & 0 & 0 & 0 \\ 0 & 0 & 0 & I_{11}(t) & I_{12}(t) & I_{13}(t) \\ 0 & 0 & 0 & I_{21}(t) & I_{22}(t) & I_{23}(t) \\ 0 & 0 & 0 & I_{31}(t) & I_{32}(t) & I_{33}(t) \end{pmatrix},$$

where E is a 3×3 identity matrix. Since $I(t)$ is positive definite and $m > 0$, $M(t)$ and $M^{-1}(t)$ are also positive definite. Since $M(t)$ is a block diagonal matrix, $M^{-1}(t)$ is also a block diagonal matrix whose blocks are the inverses of the corresponding blocks of $M(t)$,

$$M^{-1}(t) = \begin{pmatrix} m^{-1} \cdot E & 0 \\ 0 & I^{-1}(t) \end{pmatrix}.$$

From (5.20) and (5.21) we see that

$$\begin{aligned} m \cdot \vec{a}(t) &= \vec{F}_{total}(t) \\ I(t) \cdot \vec{\alpha}(t) &= \vec{\tau}_{total}(t) + \vec{\tau}_{coriolis}(t), \end{aligned}$$

which can be rewritten using the generalized notation and mass matrix M as

$$M(t) \cdot \dot{\vec{v}}_{gen}(t) = \vec{F}^{total}_{gen}(t) + \vec{F}^{coriolis}_{gen}(t),$$

where $\dot{\vec{v}}_{gen}(t) = (\vec{a}(t), \vec{\alpha}(t))$ is the time derivative of the generalized velocity $\vec{v}_{gen}(t)$, $\vec{F}^{total}_{gen}(t)$ is the generalized external force and $\vec{F}^{coriolis}_{gen}(t) = (\vec{0}, \vec{\tau}_{coriolis}(t))$ is the generalized coriolis force, or equivalently,

$$\dot{\vec{v}}(t) = M^{-1}(t) \cdot \vec{F}_{total}(t) + M^{-1}(t) \cdot \vec{F}_{coriolis}(t).$$

We have removed the $_{gen}$ subscripts this time to improve readability. Further on, when it is clear that the generalized notation is used, the subscripts will be removed implicitly and "generalized" adjective omitted as well ($\vec{v}$ in the generalized notation will refer to both the linear and angular velocities).

This equation describes how the acceleration of the rigid body changes if $\vec{F}_{total}(t)$ is applied to it. The acceleration $M^{-1}(t) \cdot \vec{F}_{coriolis}(t)$ due to the coriolis force is independent of other forces and can be treated as if it was a part of the acceleration due to the total external force, that is, the *coriolis force can be incorporated into $\vec{F}_{total}(t)$ and can be ignored elsewhere*[7]. From now on we will assume that the coriolis force is implicitly included into $\vec{F}_{total}(t)$ whenever working with generalized forces.

This allows to write

$$M(t) \cdot \dot{\vec{v}}(t) = \vec{F}(t) \tag{5.23}$$

for any force $\vec{F}(t)$ and acceleration $\dot{\vec{v}}(t)$ due to $\vec{F}(t)$. This equation generalizes (5.20) and (5.21) and describes the dynamics of the rigid body, [9]. As can be seen, it resembles the Newton's

[6]Until now we used M to denote the rigid body's mass. This would clash with our generalized notation and so the mass would be denoted m.

[7]This is generally true for any inertial force.

Second Law for particles and rigid bodies can thus be imagined as special particles with time-varying masses $M(t)$ that move in $\mathbf{R}^6$.

The formulation of rigid body dynamics with mass matrices will become very useful when the equations for constrained rigid body dynamics are developed because we will be able to reuse derivations from constrained particle dynamics that were formulated in terms of mass matrices for particles.

5.2.4 Numerical Issues

If we actually tried to simulate a rigid body, say according to (5.18), we would experience a numerical drift which would mostly affect the representation of the body orientation $R(t)$. The problem is that rigid bodies have 3 rotational degrees of freedom but rotation matrices $R(t)$ have 3×3 DOFs and the extra DOFs have to be neutralized by 6 additional conditions on the matrix rows, known as the *orthogonality conditions*. While updating $R(t)$, orthogonality conditions are violated and the matrix must be "orthogonalized" in order to represent a rotation again. Whenever the normalization takes place, rigid body state is changed directly, which makes it a new initial state and the ODE solver must be restarted.

There is, however, a better way to represent body orientations — by the use of unit quaternions, represented as $[scalar = s, \overrightarrow{vector} = (v_1, v_2, v_3)]$ pairs. Quaternions, unlike rotation matrices, have 4 DOFs and the extra DOF is neutralized by a condition that requests that the quaternion must have a unit length. Since there is only one constrained DOF, orientations represented by unit quaternions experience much less numerical drift than rotation matrices and the stabilization of the corresponding ODE is much simpler (corresponds to the normalization of the quaternion). We will assume that the reader is acquainted with quaternions and the representation of rotations by unit quaternions and present a modified equation of motion for a single rigid body straight off (one might read [1] to get introductory level information on quaternions). A variant of (5.18) with a different block row due to orientation representation by a quaternion is shown

$$\begin{aligned}\frac{\partial}{\partial t}\vec{y}(t) &= \frac{\partial}{\partial t}(\vec{x}(t), q(t), \vec{P}(t), \vec{L}(t)) \\ &= (\vec{v}(t), \frac{1}{2} \cdot [0, \vec{\omega}(t)] \cdot q(t), \vec{F}_{total}(t), \vec{\tau}_{total}(t)),\end{aligned}$$

where $q(t) = [\cos(\alpha/2), \sin(\alpha/2) \cdot \vec{v}]$ is a unit quaternion that represents the body orientation by a rotation about axis $\vec{v}$ by angle α. To utilize previous equations expressed in terms of $R(t)$, one might consider $R(t)$ as an additional auxiliary quantity, computed from the quaternion $q(t)$. Derivations can be found in [10].

5.3 Abstraction

In this section we will cover the abstraction and representation of rigid bodies and external forces that may act upon them. The abstractions generalize and built on top of the previous abstractions of particles and force objects described in the particle dynamics chapter.

We already know that the simulation of rigid bodies and particles corresponds to the numerical solving of the corresponding equations of motion, which are coupled ODEs. Since we have got multiple forms of equations of motion to choose from, we have to decide what variant of the equation of motion is to be used to drive the simulation and then implement the functions required by the ODE solver (see page 13) accordingly. The simulation of rigid bodies conceptually follows the same scheme like the simulation of particles and the only thing which makes it harder are the extra dimensions due to body orientations.

We will use the momentum-based formulation of the equation of motion (5.18) to simulate the bodies. This decision dictates what the layout of the body state vector $\vec{y}(t)$ is and what properties are recorded in the body state (equation (5.17)). Obviously, vector $\vec{y}(t)$ will be a part of our rigid body representation. The right-hand side of (5.18) tells us what quantities have to be processed when the body state derivative is requested to be evaluated by the ODE solver. Since the quantities are defined in terms of *auxiliary quantities* $\vec{v}(t), \vec{\omega}(t), \vec{I}(t)$ and $\vec{I}^{-1}(t)$, derived from the rigid body state, *constant quantities* $I_{body}, I_{body}^{-1}, M$ and M^{-1}, characterizing the rigid body mass properties, and *computed quantities* $\vec{F}_{total}(t), \vec{\tau}_{total}(t)$, storing the total external force and torque acting on the body at time t, all these quantities should be represented as well.

Let us focus at $\vec{F}_{total}(t)$ and $\vec{\tau}_{total}(t)$ quantities. When these quantities were defined (equations (5.7), (5.9)), we were conceptually working with a rigid body made of particles, correlated the positions where the forces were exerted with the positions of particles and defined a torque according to the position of the particle. Since we can imagine that there happens to be a particle at any world space position we are interested in, we can interpret $\vec{\tau}_i(t)$ from (5.7) as a torque due to an external force $\vec{F}_i(t)$ exerted on the body at the world space position $\vec{r}_i(t)$ and hence can write

$$\vec{F}_{total}(t) = \sum_{i=1}^{n} \vec{F}_i(t) \qquad \vec{\tau}_{total}(t) = \sum_{i=1}^{n} \vec{\tau}_i(t) = \sum_{i=1}^{n} (\vec{r}_i(t) - \vec{x}(t)) \times \vec{F}_i(t),$$

where $\vec{F}_i(t)$ is an external force exerted at point $\vec{r}_i(t)$ on the body.

This allows to implement $\vec{F}_{total}(t)$ and $\vec{\tau}_{total}(t)$ as a force and torque accumulators and handle the contributions to the accumulators in terms of **apply_force**$(\vec{r}, \vec{F})$ function that adds the exerted force $\vec{F}$ to the body's $\vec{F}_{total}(t)$ accumulator and the corresponding torque $(\vec{r} - \vec{x}(t)) \times \vec{F}$ due to $\vec{F}$ exerted at $\vec{r}$ to the $\vec{\tau}_{total}(t)$ accumulator. The contributions are represented by force objects that provide functions to exert the represented forces and torques in terms of **apply_force** function called on affected bodies.

Chapter 6

Constrained Rigid Body Dynamics

In this chapter we will generalize the Lagrange multiplier method for enforcing constraints on particles in particle systems to constraints imposed on rigid bodies and introduce some improvements, such as handling of inequality constraints and bounded equality constraints, which are needed for enforcing non-penetration and friction constraints generally called the contact constraints. We are in luck because the most of the theory was explained already. The remaining chapters will be devoted to the abstraction (and partial implementation) of the concepts presented in this chapter and the use of rigid body dynamics to animate human-like figures.

We will describe the (complete) constraint theory first and then show how it can be used to enforce certain types of constraints, such as contact constraints. We will not tackle the problem of contact constraints until the end of the chapter, unlike many other authors, because these are the most complex constraints we will ever handle and they are relatively hard to implement if they are to be handled correctly. If one is really inclined to cope with non-penetration constraints first, it is suggested to read [10], [13], [7], where non-penetration constraints are not abstracted and are formulated directly in the form of LCP (LCP problems will be presented in section (6.2.4) and non-penetration constraints with friction in sections (6.5.2) and (6.5.3)).

Throughout this chapter we will express vectors and matrices mostly in the block form and work with the generalized notation of forces, velocities and accelerations introduced on page 44. Its main advantage is that the equations we will develop will mimic the equations from the constrained particle dynamics chapter quite closely and allow us to proceed analogously to [9].

6.1 Beyond Particles

The purpose of this section is to transfer concepts from constrained particle dynamics to rigid bodies, establish a notation we will use throughout the chapter and make us ready for improvements in the further sections.

When we were working with particles, constraints were formulated in terms of position-level implicit functions $\vec{C}_p = \vec{0}$, velocity-level implicit functions $\vec{C}_v = \vec{0}$ or acceleration-level implicit functions $\vec{C}_a = \vec{0}$ — constraints were defined by *equalities.* These equalities parametrized the sets of valid particle positions, valid particle velocities or valid particle accelerations. The goal was to ensure that particle positions, velocities or accelerations remained valid. As we shown, position-level and velocity-level equality constraints could be implemented by corresponding acceleration-level constraints obtained by differentiating the original constraints once or twice with respect to time and shown how acceleration-level equality constraints could be maintained by introducing a constraint force.

We will more or less follow this road map and generalize the concepts for rigid bodies. We will however not define or derive the concepts once again from the ground up but utilize the fact that rigid bodies can be treated as a special kind of particles moving in $\mathbf{R}^6$, implicitly transferring the definitions, concepts and also results to rigid bodies.

6.1.1 Notation

We will use the following notation:

- generalized position $\vec{r}$ of a body

  A block vector $\vec{r}$ consisting of two blocks storing the position $\vec{x} \in \mathbf{R}^3$ of the rigid body's center of mass and orientation $\vec{\theta} \in \mathbf{R}^3$ by Euler angles, $\vec{r} = (\vec{x}, \vec{\theta}) \in \mathbf{R}^6$. Generalized position serves the same purpose like the vector $\vec{r}$ of particles and corresponds to our view of rigid bodies as particles in $\mathbf{R}^6$.

  The representation of orientation by Euler angles $\vec{\theta}$ stresses that rigid bodies have 3 rotational degrees of freedom and allows to define the generalized velocity as a time derivative of the generalized position (since angular velocity $\vec{\omega}$ is formally defined as $\dot{\vec{\theta}}$, see (5.3)).

  Note that this is a formalism only. We still maintain the orientation of rigid bodies by rotation matrices R or quaternions q and Euler angles $\vec{\theta}$ can always be extracted from R or q. Euler angles $\vec{\theta}$ are used to define the generalized position $\vec{r}$ which is in turn used to define position-level constraints for rigid bodies. Since it is often hard to express rotational constraints in terms of $\vec{\theta}$ on the position level, unlike of $\vec{\omega}$ on the velocity level, and constraints will finally be enforced on the acceleration level anyway, one might forget about $\vec{\theta}$ and formulate rotational constraints directly on the velocity or acceleration level, in terms of $\vec{\omega}$ or $\vec{\alpha}$. This is the reason why we do not study $\vec{\theta}$ further.

- generalized velocity $\vec{v}$ of a body

  A block vector $\vec{v}$ consisting of two blocks storing the rigid body's linear velocity $\dot{\vec{x}} \in \mathbf{R}^3$ and angular velocity $\vec{\omega} \in \mathbf{R}^3$, $\vec{v} = (\dot{\vec{x}}, \vec{\omega}) \in \mathbf{R}^6$. Generalized velocity serves the same purpose like vector $\dot{\vec{r}}$ of particles and in fact $\vec{v} = \dot{\vec{r}}$.

  To prevent clash with the generalized velocity $\vec{v}$ we will denote the linear velocity by $\dot{\vec{x}}$.

- generalized acceleration $\vec{a}$ of a body

  A block vector $\vec{a}$ consisting of two blocks storing the rigid body's linear acceleration $\ddot{\vec{x}} \in \mathbf{R}^3$ and angular acceleration $\vec{\alpha} \in \mathbf{R}^3$, $\vec{a} = (\ddot{\vec{x}}, \vec{\alpha}) \in \mathbf{R}^6$. Generalized acceleration serves the same purpose like vector $\ddot{\vec{r}}$ of particles and in fact $\vec{a} = \ddot{\vec{r}}$.

  To prevent clash with the generalized acceleration $\vec{a}$ we will denote the linear acceleration by $\ddot{\vec{x}}$.

- generalized force $\vec{F}$ exerted on a body

  A block vector $\vec{F}$ consisting of two blocks storing the force $\vec{f} \in \mathbf{R}^3$ (imagined as if acting at the body's center of mass) and torque $\vec{\tau} \in \mathbf{R}^3$ due to $\vec{f}$, $\vec{F} = (\vec{f}, \vec{\tau}) \in \mathbf{R}^6$. Generalized force serves the same purpose like force $\vec{F}$ exerted on a particle and in fact $\vec{F} = M \cdot \vec{a}$ for a generalized acceleration $\vec{a}$ due to $\vec{F}$, where M is the rigid body's mass matrix (see (5.23)).

  If $\vec{F}$ refers to the generalized total force exerted on the body, coriolis force is assumed to be included into $\vec{F}$.

We will restrict ourselves to constraints that constrain exactly two bodies, assuming that more complex constraints can be created by combining other constraints. This will make the implementation of efficient constraint solver possible.

6.1.2 Equality Constraints

In order to transfer definitions and equations regarding equality constraints from particle systems to rigid bodies we can simply copy the old material and adjust for the (minor) differences due to different vector lengths and the restriction that each constraint affects exactly two bodies — the generalized notation is of help here. To save space we will present the results straight off.

Constraint Formulation

For the purposes of constraint formulation in this section we will assume that the first body to be constrained has index A, the second body has index B and that $\vec{r}$, $\vec{v}$ and $\vec{a}$ are block vectors consisting of the generalized positions, velocities and accelerations of the two constrained bodies, so that

$$\begin{aligned}\vec{r} &= (\vec{r}_A, \vec{r}_B) = ((\vec{x}_A, \vec{\theta}_A), (\vec{x}_B, \vec{\theta}_B))\\ \vec{v} &= (\vec{v}_A, \vec{v}_B) = ((\dot{\vec{x}}_A, \vec{\omega}_A), (\dot{\vec{x}}_B, \vec{\omega}_B))\\ \vec{a} &= (\vec{a}_A, \vec{a}_B) = ((\ddot{\vec{x}}_A, \vec{\alpha}_A), (\ddot{\vec{x}}_B, \vec{\alpha}_B)),\end{aligned}$$

where $\vec{r}, \vec{v}, \vec{a}$ are 12×1 block vectors consisting of four 3×1 blocks organized in such a way so that odd blocks store linear quantities and even blocks store angular quantities.

Let $\vec{C}_p$ be a vector function of $\vec{r} = (\vec{r}_A, \vec{r}_B)$, then the equation of the form

$$\vec{C}_p(\vec{r}) = \vec{0} \in \mathbf{R}^m \tag{6.1}$$

defines a *position-level equality constraint* on bodies A and B due to $\vec{C}_p$, vectors $\vec{r}$ that satisfy the equation are the valid generalized positions of the constrained bodies (invalid positions must be avoided) and m is the dimensionality of the constraint or the number of DOFs removed from the bodies. Individual scalar $C_p^i(\vec{r}) = 0$ equations correspond to the removed DOFs.

By differentiating the position-level constraint with respect to time and generalizing for an arbitrary *constant right-hand side vector* $\vec{k}$, we get the *velocity-level equality constraint* $\vec{C}_v(\vec{r}, \vec{v}) = \vec{0}$ due to $\dot{\vec{C}}_p$ and $\vec{k}$ of dimension m in the form of $\vec{C}_v(\vec{r}, \vec{v}) = J(\vec{r}) \cdot \vec{v} - \vec{k} = \vec{0}$, or equivalently as

$$J(\vec{r}) \cdot \vec{v} = \vec{k}, \tag{6.2}$$

where $J(\vec{r}) = \frac{\partial \vec{C}_p}{\partial \vec{r}}$ is the Jacobian matrix of $\vec{C}_p$ with dimensions $m \times 12$ and $\vec{k}$ is a constant vector of length m. The velocity-level equality constraint parameterizes the set of valid generalized velocities of the constrained bodies with respect to their generalized positions. As before, invalid velocities must be avoided.

By differentiating the velocity-level constraint with respect to time and introducing $\vec{c}(\vec{r}, \vec{v}) = -\dot{J} \cdot \vec{v}$, we get the *acceleration-level equality constraint* $\vec{C}_a(\vec{r}, \vec{v}, \vec{a}) = \vec{0}$ due to $\dot{\vec{C}}_v$ of dimension m in the form of $\vec{C}_a(\vec{r}, \vec{v}, \vec{a}) = J(\vec{r}) \cdot \vec{a} - \vec{c}(\vec{r}, \vec{v}) = \vec{0}$, or equivalently as

$$J(\vec{r}) \cdot \vec{a} = \vec{c}(\vec{r}, \vec{v}), \tag{6.3}$$

where

$$J(\vec{r}) = \frac{\partial \vec{C}_v}{\partial \vec{v}} = \frac{\partial \vec{C}_p}{\partial \vec{r}} = \begin{pmatrix} \frac{\partial \vec{C}_v}{\partial \dot{\vec{x}}_A} & \frac{\partial \vec{C}_v}{\partial \vec{\omega}_A} & \frac{\partial \vec{C}_v}{\partial \dot{\vec{x}}_B} & \frac{\partial \vec{C}_v}{\partial \vec{\omega}_B} \end{pmatrix} = \begin{pmatrix} \frac{\partial \vec{C}_p}{\partial \vec{x}_A} & \frac{\partial \vec{C}_p}{\partial \vec{\theta}_A} & \frac{\partial \vec{C}_p}{\partial \vec{x}_B} & \frac{\partial \vec{C}_p}{\partial \vec{\theta}_B} \end{pmatrix} =$$

$$\begin{pmatrix} J_A^{linear} & J_A^{angular} & J_B^{linear} & J_B^{angular} \end{pmatrix} = (J_A \; J_B)$$

is the Jacobian matrix of function $\vec{C}_v$ or $\vec{C}_p$ with dimensions $m \times 12$, consisting of four $m \times 3$ Jacobian blocks $J_A^{linear}, J_A^{angular}, J_B^{linear}, J_B^{angular}$ due to linear and angular properties of bodies A and B or two $m \times 6$ Jacobian blocks $J_A = \begin{pmatrix} J_A^{linear} & J_A^{angular} \end{pmatrix}, J_B = \begin{pmatrix} J_B^{linear} & J_B^{angular} \end{pmatrix}$ due to bodies A and B, $\dot{J}$ is the time derivative of J and equals $\frac{\partial \dot{\vec{C}}_v}{\partial \vec{r}}$ and $\vec{c}(\vec{r}, \vec{v}) = -\dot{J} \cdot \vec{v}$. The acceleration-level equality constraint parameterizes the set of valid generalized accelerations of the constrained bodies with respect to their generalized positions and velocities. The goal is to compute an appropriate constraint force so that invalid accelerations would be avoided.

Analogously to particles, position-level equality constraints can be maintained incrementally, by starting from states that are consistent with the position-level and velocity-level formulations of the constraints and enforcing the corresponding acceleration-level constraints instead. Velocity-level equality constraints with constant right-hand side vectors can be maintained by starting from states that are consistent with the original velocity-level constraint and enforcing the corresponding acceleration-level constraints instead.

From now on we will work with multiple constraints and bodies. If there are a total number of n bodies, then vectors $\vec{r} = (\vec{r}_1, \ldots, \vec{r}_n)$, $\vec{v} = (\vec{v}_1, \ldots, \vec{v}_n)$ and $\vec{a} = (\vec{a}_1, \ldots, \vec{a}_n)$ will be block-vectors consisting of the generalized positions, velocities and accelerations of the individual bodies. Jacobian matrices $J = \frac{\partial \vec{C}_p}{\partial \vec{r}} = \frac{\partial \vec{C}_v}{\partial \vec{v}}$ of the individual constraints will be treated as block-matrices consisting of n blocks and the non-zero blocks due to constrained bodies A and B will be indexed A and B. These blocks will correspond to J_A and J_B matrices defined above.

Since there will be more constraints i in effect at the same time, Jacobian matrices will have to be subscripted by constraint indices, so that J_i will refer to the Jacobian matrix due to constraint i and J_{i,A_i} will refer to its block due to the constrained body A_i. This is not anything new, we did this already in section (4.2.1).

Constraint Force Basis

Now assume that we have n bodies and c acceleration-level constraints defined according to (6.3), so that the i-th constraint constrains bodies A_i and B_i, m_i is the dimensionality of the constraint, J_{i,A_i} and J_{i,B_i} are the Jacobian blocks due to the constraint and the constrained bodies of dimensions $m_i \times 6$ and $\vec{c}_i$ is the right-hand side vector of the constraint equation of length m_i. We thus have

$$J_{i,A_i} \cdot \vec{a}_{A_i} + J_{i,B_i} \cdot \vec{a}_{B_i} = \vec{c}_i \tag{6.4}$$

for $1 \le i \le c$, where $\vec{a}_j, 1 \le j \le n$ is the generalized acceleration of body j, or equivalently

$$J \cdot \vec{a} = \begin{pmatrix} J_{11} & \ldots & J_{1n} \\ \vdots & \ddots & \vdots \\ J_{c1} & \ldots & J_{cn} \end{pmatrix} \cdot \begin{pmatrix} \vec{a}_1 \\ \vdots \\ \vec{a}_n \end{pmatrix} = \begin{pmatrix} \vec{c}_1 \\ \vdots \\ \vec{c}_c \end{pmatrix}, \tag{6.5}$$

where J is a block Jacobian matrix whose non-zero blocks are given by J_{i,A_i} and J_{i,B_i} and $\vec{a}$, $\vec{c}$ are block vectors made up of vectors $\vec{a}_j$, $\vec{c}_i$.

Let $\vec{F_j}$ be a generalized force exerted on body j, $\vec{a}_j$ the generalized acceleration of the body due to $\vec{F_j}$ and M_j its mass matrix. From (5.23) we get that $M_j \cdot \vec{a}_j = \vec{F_j}$ and so if M is a square block diagonal matrix with individual mass matrices M_j on the diagonal, $\vec{F} = (\vec{F_1}, \dots, \vec{F_n})$ and $\vec{a} = (\vec{a}_1, \dots, \vec{a}_n)$ then the dynamics of the rigid body system can be described by

$$M \cdot \vec{a} = \vec{F}, \tag{6.6}$$

where

$$M = \begin{pmatrix} M_1 & 0 & \dots & 0 \\ 0 & M_2 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & M_n \end{pmatrix}$$

is the mass matrix of the rigid body system. For simplicity $\vec{F}$ is called the force exerted on the system and $\vec{a}$ the acceleration of the system due to $\vec{F}$ or the response of the system to $\vec{F}$.

Since rigid bodies can be seen as particles in $\mathbf{R}^6$, it follows from (4.5) and (4.7) that the generalized constraint forces $(\vec{F_c^i})_{A_i}$ and $(\vec{F_c^i})_{B_i}$ exerted on body A_i and B_i due to constraint i have the form

$$\begin{aligned} (\vec{F_c^i})_{A_i} &= J_{i,A_i}^T \cdot \vec{\lambda}_i \\ (\vec{F_c^i})_{B_i} &= J_{i,B_i}^T \cdot \vec{\lambda}_i, \end{aligned} \tag{6.7}$$

where $\vec{\lambda}_i$ is a $m_i \times 1$ vector of Lagrange multipliers due to constraint i. If J_i denotes the i-th block row of the Jacobian matrix J, the Jacobian matrix due to constraint i, then the constraint force $\vec{F_c^i}$ exerted on the system due to the constraint equals $\vec{F_c^i} = J_i^T \cdot \vec{\lambda}_i$, where $(\vec{F_c^i})_{A_i}$ and $(\vec{F_c^i})_{B_i}$ are the only non-zero blocks. By concatenating multiplier vectors $\vec{\lambda}_i$ to a block vector $\vec{\lambda} = (\vec{\lambda}_1, \dots, \vec{\lambda}_c)$, we can express the total constraint force $\vec{F_c}$ due to all constraints as $\vec{F_c} = \sum_{i=1}^{c} \vec{F_c^i} = J^T \cdot \vec{\lambda}$.

Dense Formulation

To compute the constraint force we have to compute the multipliers $\vec{\lambda}$. Luckily our formulation of constraints (6.5) matches the formulation of particle constraints (4.7) and the dynamics of the rigid body system (6.6) matches the particle systems dynamics. Therefore (4.6) is applicable and we can express $\vec{\lambda}$ as a solution to the linear system

$$A \cdot \vec{\lambda} + \vec{b} = \vec{0}, \tag{6.8}$$

where $A = J \cdot M^{-1} \cdot J^T$ is a $m \times m$ matrix, m is the total number of DOFs removed by the constraints, $m = \sum_{i=1}^{c} m_i$, $\vec{b} = J \cdot M^{-1} \cdot \vec{F}_{total} - \vec{c}$ is a $m \times 1$ vector, $\vec{c} = (\vec{c}_1, \dots, \vec{c}_c)$ and $\vec{F}_{total} = (\vec{F}_1^{total}, \dots, \vec{F}_n^{total})$ is the total external force exerted on the system (including the coriolis forces).

Matrix A can be treated as a $c \times c$ block matrix, whose block rows and block columns correspond to the individual constraints and the value of the (i_1, i_2)-th block $(m_{i_1} \times m_{i_2})$ is given by $A_{i_1,i_2} = \sum_{j=1}^{n} J_{i_1,j} \cdot M_j^{-1} \cdot (J_{i_2,j})^T$.

Since mass matrices M_j are non-zero, A_{i_1,i_2} is non-zero iff there exists a common body j that the both constraints i_1 and i_2 affect (that is $J_{i_1,j} \neq 0$ and $J_{i_2,j} \neq 0$). Although J is dense (there are two non-zero blocks per a block row) this shows that A can be arbitrarily dense[1]. However, unless there are too many constraints, $\vec{\lambda}$ can be computed simply in terms of A. M_j

[1]Imagine a branching structure with a common body and n other bodies attached to the common body.

and M_j^{-1} are positive definite, hence M and M^{-1} are positive definite and since J is assumed to be non-singular, A is also positive definite and can be factored using Cholesky factorization to solve for $\vec{\lambda}$, [9].

Equation (6.8) can be interpreted as a block equation consisting of c block-rows, where the i-th block row $A_i \cdot \vec{\lambda} + \vec{b}_i = \vec{0}$ imposes a condition on the blocks of $\vec{\lambda}$ due to the i-th constraint. The i-th block row of A and $\vec{b}$ tells us what the value of the i-th acceleration-level constraint function $J_i \cdot \vec{a} - \vec{c}_i$ would be if the constraint force $J^T \cdot \vec{\lambda}$ was exerted on the system, $J \cdot \vec{a} - \vec{c} = J \cdot (M^{-1} \cdot (J^T \cdot \vec{\lambda} + \vec{F}_{total})) - \vec{c} = A \cdot \vec{\lambda} + \vec{b}$, $J_i \cdot \vec{a} - \vec{c}_i = A_i \cdot \vec{\lambda} + \vec{b}_i$.

Sparse Formulation

Conditions on multipliers $\vec{\lambda}$ can be reformulated in terms of an always sparse matrix H,

$$H = \begin{pmatrix} M & -J^T \\ -J & 0 \end{pmatrix} \qquad H \cdot \begin{pmatrix} \vec{y} \\ \vec{\lambda} \end{pmatrix} + \begin{pmatrix} \vec{0} \\ -\vec{b} \end{pmatrix} = \vec{0}, \tag{6.9}$$

where $\vec{b} = J \cdot M^{-1} \cdot \vec{F}_{total} - \vec{c}$ and $\vec{y}$ is a helper vector, [9]. Indeed, the first block row yields $M \cdot \vec{y} - J^T \cdot \vec{\lambda} = \vec{0}$, hence $\vec{y} = M^{-1} \cdot J^T \cdot \vec{\lambda}$ ($\vec{y}$ equals the system response to the constraint force $J^T \cdot \vec{\lambda}$), and the second block row produces $-J \cdot \vec{y} - \vec{b} = \vec{0}$, hence $J \cdot M^{-1} \cdot J^T \cdot \vec{\lambda} + \vec{b} = \vec{0}$. Since the blocks of H consist of sparse matrices $M, -J, -J^T$, the matrix H is also sparse and sparse matrix methods can be used to solve for $\vec{\lambda}$.

6.2 Acceleration-Level Constraints

In this section we will generalize the acceleration-level formulation of equality constraints defined by (6.4) so that we would be able to handle inequalities and other types of constraints.

For the purposes of this chapter we will use the dense formulation (6.8) to base our conditions on $\vec{\lambda}$ on. As it will turn out, the other constraint types (inequalities, bounded equalities) will manifest themselves by other conditions on $\vec{\lambda}$ but they will still be expressed in terms of matrix A and vector $\vec{b}$.

The material presented in this section needs be seen in the context of the total number of c constraints and n bodies. Vector $\vec{r} = (\vec{r}_1, \ldots, \vec{r}_n)$ will denote a block vector of generalized body positions, $\vec{v} = \dot{\vec{r}}$ a block vector of generalized body velocities, $\vec{a} = \ddot{\vec{r}}$ a block vector of generalized body accelerations. We will say that $\vec{r}$ is a vector from the position space and $\dot{\vec{r}}$ a vector from the velocity space. J_i will denote the i-th block-row of the Jacobian matrix J (6.5), computed as if all constraints $1 \leq i \leq c$ were equality constraints (6.4). We will call J_i the Jacobian matrix due to constraint i and its non-zero blocks J_{i,A_i}, J_{i,B_i} the Jacobian blocks due to constraint i and constrained bodies A_i and B_i.

6.2.1 Equality Constraints

In the previous section we formulated acceleration-level equality constraints (6.4) in the form of

$$J_{i,A_i} \cdot \vec{a}_{A_i} + J_{i,B_i} \cdot \vec{a}_{B_i} = \vec{c}_i,$$

where i was the index of the constraint, and derived a condition on $\vec{\lambda}$ multipliers due to the equality constraint i

$$A_i \cdot \vec{\lambda} + \vec{b}_i = \vec{0}. \tag{6.10}$$

We will now derive similar equations for other types of constraints. Note that the equations have to be formulated in the context of all other active constraints, hence we have to index quantities specific to a given constraint by an appropriate index.

6.2.2 Inequality Constraints

For an illustration and motivation let us assume that we have an 1-DOF inequality position-level constraint i defined as $C_p(\vec{r}(t)) \geq 0$. We did not define inequality position-level constraints yet, but we can think of C_p as if it measured a "separation distance" between two objects, $\dot{C}_p$ as if it measured an "approaching velocity" and $\ddot{C}_p$ a "relative acceleration". For example, $C_p(\vec{r}(t)) \geq 0$ might be interpreted as a non-penetration constraint between a ball and a brick-wall, saying that "the ball should be in front of the wall". Let us assume that $C_p(\vec{r}(t)) = 0$ and $\dot{C}_p(\vec{r}(t)) = 0$ at the current time t. That is, the ball is in contact with the wall ($C_p(\vec{r}(t)) = 0$), but it did not strike on the wall (did not approach the wall with a non-zero velocity, $\dot{C}_p(\vec{r}(t)) = 0$) — more formally, $\vec{r}(t)$ is *resting at the boundary of the valid set of body positions* in the position product space. We would like to maintain the original position-level constraint.

Since $C_p(\vec{r}(t)) = 0$ we can ensure that the original constraint is not violated at the next time step by enforcing $\dot{C}_p(\vec{r}(t)) \geq 0$. Since $\dot{C}_p(\vec{r}(t)) = 0$ (the corresponding velocity-level constraint holds already) we achieve this by requiring $\ddot{C}_p(\vec{r}(t)) \geq 0$, which is an acceleration-level greater-or-equal constraint. According to our interpretation of C_p, $\ddot{C}_p(\vec{r}(t)) \geq 0$ merely says that the ball must not accelerate against the wall, which conforms to our physical intuition.

Greater-or-Equal Constraints

Acceleration-level greater-or-equal constraint i affecting bodies A_i and B_i is defined as

$$J_{i,A_i} \cdot \vec{a}_{A_i} + J_{i,B_i} \cdot \vec{a}_{B_i} \geq \vec{c}_i, \tag{6.11}$$

where J_{i,A_i} and J_{i,B_i} are $m_i \times 6$ Jacobian blocks due to the constraint i and the constrained bodies, m_i is the dimensionality of the constraint and $\vec{c}_i$ is the right-hand side vector of length m_i. The definition pretty much matches the definition of equality constraints (6.4), but $=$ is replaced by $\geq$.

We will now present some conditions on the constraint force $\vec{F}_c^i$ due to the constraint i. Recall that the goal of $\vec{F}_c^i$ is to cancel the components of $\vec{F}_{total}$ that would make the constrained bodies accelerate towards an invalid state in the velocity space. In the case of equality constraints (see page 25), the bodies were restricted to remain on the intersection of the hypersurfaces due to constraint DOFs in the velocity space and $\vec{F}_c^i$ cancelled any acceleration in the directions of the hypersurface normals. In the case of greater-or-equal constraints, however, the bodies can accelerate from the hypersurfaces in the directions of their normals but not in the opposite directions. The positive accelerations along the positive directions of the normals are unconstrained and therefore $\vec{\lambda}_i \geq \vec{0}$. If the bodies are already accelerating to the front of the hypersurface j, $(J_{i,A_i} \cdot \vec{a}_{A_i} + J_{i,B_i} \cdot \vec{a}_{B_i} - \vec{c}_i)_j > 0$, then the constraint force due to that hypersurface must vanish, that is $(\vec{\lambda}_i)_j = 0$.

To clarify, let us write these results in a more formalized way. For these purposes we will rewrite the constraint (6.11) as $\vec{C}_i \geq \vec{0}$, where $\vec{C}_i = J_i \cdot \vec{a} - \vec{c}_i$. Then the constraint i including the conditions on $\vec{\lambda}_i$ can be expressed as

$$\begin{aligned} \vec{C}_i &\geq \vec{0} \\ \vec{\lambda}_i &\geq \vec{0} \\ \vec{C}_i \cdot \vec{\lambda}_i &= 0, \end{aligned}$$

where $\vec{C}_i \cdot \vec{\lambda}_i = \sum_{j=1}^{m_i} (\vec{C}_i)_j \cdot (\vec{\lambda}_i)_j = 0$ in fact means that $(\vec{C}_i)_j \cdot (\vec{\lambda}_i)_j = 0$ for $1 \leq j \leq m_i$, because both the factors are required to be positive. Since $\vec{C}_i = J_i \cdot \vec{a} - \vec{c}_i = (J \cdot \vec{a} - \vec{c})_i =$

$(J \cdot (M^{-1} \cdot J^T \cdot \vec{\lambda} + M^{-1} \cdot \vec{F}_{total}) - \vec{c})_i = (A \cdot \vec{\lambda} + \vec{b})_i = A_i \cdot \vec{\lambda} + \vec{b}_i$, the conditions can be reformulated in terms of the i-th block row of matrix A and the i-th block of vector $\vec{b}$

$$\begin{aligned} A_i \cdot \vec{\lambda} + \vec{b}_i &\geq \vec{0} \\ \vec{\lambda}_i &\geq \vec{0} \\ (A_i \cdot \vec{\lambda} + \vec{b}_i) \cdot \vec{\lambda}_i &= 0, \end{aligned} \tag{6.12}$$

where A and $\vec{b}$ are from (6.10) and A_i refers to the block row of A due to constraint i. The last condition in (6.12) says that the components of $\vec{\lambda}_i$ are *complementary*[2] to the components of $A_i \cdot \vec{\lambda} + \vec{b}_i = \vec{C}_i = J_i \cdot \vec{a} - \vec{c}_i$ and together with the other two conditions forms the ultimate conditions on $\vec{\lambda}$ due to the greater-or-equal constraint i. Equation (6.12) serves the same purpose like (6.10) for equality constraints.

Less-or-Equal Constraints

Acceleration-level less-or-equal constraint i affecting bodies A_i and B_i is defined as

$$J_{i,A_i} \cdot \vec{a}_{A_i} + J_{i,B_i} \cdot \vec{a}_{B_i} \leq \vec{c}_i, \tag{6.13}$$

where J_{i,A_i} and J_{i,B_i} are $m_i \times 6$ Jacobian blocks due to the constraint i and the constrained bodies, m_i is the dimensionality of the constraint and $\vec{c}_i$ is the right-hand side vector of length m_i. The definition pretty much matches the definition of greater-or-equal constraint (6.11), but $\geq$ is replaced by $\leq$.

Using the notation and following the derivation scheme of greater-or-equal constraints, bodies in the velocity space can accelerate from the hypersurfaces in the directions opposite to the hypersurface normals only. That said, it is required that $\vec{\lambda}_i \leq \vec{0}$. Also if the bodies are already accelerating to the back of the hypersurface j, $(\vec{C}_i)_j < 0$, then the constraint force due to that hypersurface must vanish, that is $(\vec{\lambda}_i)_j = 0$. That way we obtain the conditions on $\vec{\lambda}$ due to the less-or-equal constraint i, expressed as

$$\begin{aligned} A_i \cdot \vec{\lambda} + \vec{b}_i &\leq \vec{0} \\ \vec{\lambda}_i &\leq \vec{0} \\ (A_i \cdot \vec{\lambda} + \vec{b}_i) \cdot \vec{\lambda}_i &= 0, \end{aligned} \tag{6.14}$$

where A_i and $\vec{b}_i$ are from (6.12). Again, the components of $\vec{\lambda}_i$ are complementary to the components of $A_i \cdot \vec{\lambda} + \vec{b}_i = \vec{C}_i = J_i \cdot \vec{a} - \vec{c}_i$.

6.2.3 Bounded-Equality Constraints

We would like to define a constraint that would behave like an equality constraint whose constraint force magnitude was tracked and when it grew beyond a specified limit the constraint would automatically turn into a corresponding greater-or-equal or less-or-equal constraint. Such a constraint could be used to implement various kinds of motors with limited motor forces so that the constraint would break if the maximum motor force magnitude was exceeded[3]. We will

[2]That is $(\vec{\lambda}_i)_j \neq 0 \Rightarrow (A_i \cdot \vec{\lambda} + \vec{b}_i)_j = 0$ and $(A_i \cdot \vec{\lambda} + \vec{b}_i)_j \neq 0 \Rightarrow (\vec{\lambda}_i)_j = 0$.

[3]Note that such a behavior can not be simulated by treating the constraint as an (unbounded) equality constraint, computing the constraint force and finally clamping the components of $\vec{\lambda}_i$ to specified ranges because other constraints influenced by this constraint would break (constraint forces due to the other constraints were computed with respect to the unclamped $\vec{\lambda}$).

define this constraint (as usually) in terms of a constraint force $\vec{F}_c^i = J_i^T \cdot \vec{\lambda}_i$ and conditions on $\vec{\lambda}_i$ multipliers.

If the dimensionality of the constraint was one, $m_i = 1$, then the magnitude of the constraint force $\|\vec{F}_c^i\|$ would be $\|J_i^T\| \cdot |(\vec{\lambda}_i)_1|$ and instead of limiting the force magnitude we could limit (the only component of) $\vec{\lambda}_i$. In the general case we will also limit the (multiple) components of $\vec{\lambda}_i$, but independently on each other, because the equations would not be linear otherwise. Henceforward we will assume that we are provided with m_i-vectors $\vec{\lambda}_i^{lo} \leq \vec{0}$ and $\vec{\lambda}_i^{hi} \geq \vec{0}$ so that the limits are specified as

$$(\vec{\lambda}_i^{lo})_j \leq (\vec{\lambda}_i)_j \leq (\vec{\lambda}_i^{hi})_j,$$

where $1 \leq j \leq m_i$.

According to the notation we already used while deriving conditions for greater-or-equal constraints, bodies in the velocity space will be constrained not to accelerate along the directions of the hypersurface normals due to the corresponding unbounded equality constraint. If the bodies began to accelerate off the hypersurface j due to the j-th constraint DOF in the direction of its normal, a negative $(\vec{\lambda}_i)_j$ would be required to cancel the acceleration. However if $(\vec{\lambda}_i)_j$ dropped below its lower limit $(\vec{\lambda}_i^{lo})_j$, $(\vec{\lambda}_i)_j$ would be clamped and the magnitude of the clamped value would not be big enough to cancel the prohibited acceleration, hence $(\vec{C}_i)_j > 0$. Similarly, if the bodies began to accelerate off the hypersurface in the opposite direction, a positive $(\vec{\lambda}_i)_j$ would be required to cancel the acceleration and if $(\vec{\lambda}_i)_j$ exceeded the upper limit $(\vec{\lambda}_i^{hi})_j$, $(\vec{\lambda}_i)_j$ would be clamped and hence $(\vec{C}_i)_j < 0$. Since $\vec{C}_i = A_i \cdot \vec{\lambda} + \vec{b}_i$, these conditions can be written as

$$\begin{array}{rcl} (\vec{\lambda}_i^{lo})_j & \leq & 0 \\ (\vec{\lambda}_i^{hi})_j & \geq & 0 \\ (\vec{\lambda}_i^{lo})_j \leq & (\vec{\lambda}_i)_j & \leq (\vec{\lambda}_i^{hi})_j \\ (\vec{\lambda}_i)_j = (\vec{\lambda}_i^{lo})_j & \Rightarrow & (A_i \cdot \vec{\lambda} + \vec{b}_i)_j \geq 0 \\ (\vec{\lambda}_i)_j = (\vec{\lambda}_i^{hi})_j & \Rightarrow & (A_i \cdot \vec{\lambda} + \vec{b}_i)_j \leq 0 \\ (\vec{\lambda}_i^{lo})_j < (\vec{\lambda}_i)_j < (\vec{\lambda}_i^{hi})_j & \Rightarrow & (A_i \cdot \vec{\lambda} + \vec{b}_i)_j = 0, \end{array} \tag{6.15}$$

where A_i and $\vec{b}_i$ are from (6.10). We can now define the constraint properly.

Acceleration-level bounded-equality constraint i affecting bodies A_i and B_i with $\vec{\lambda}_i$ limits $\vec{\lambda}_i^{lo} \leq \vec{0}$ and $\vec{\lambda}_i^{hi} \geq \vec{0}$ is defined as an equality constraint

$$J_{i,A_i} \cdot \vec{a}_{A_i} + J_{i,B_i} \cdot \vec{a}_{B_i} = \vec{c}_i, \tag{6.16}$$

where J_{i,A_i} and J_{i,B_i} are $m_i \times 6$ Jacobian blocks due to the constraint i and the constrained bodies, m_i is the dimensionality of the constraint, $\vec{c}_i$ is the right-hand side vector of length m_i and $\vec{\lambda}_i^{lo}$ and $\vec{\lambda}_i^{hi}$ are m_i-vectors that specify the lower and upper limits of the components of $\vec{\lambda}_i$ due to the constraint.

From (6.15) we get

$$\begin{array}{rcl} (A_i \cdot \vec{\lambda} + \vec{b}_i)_j > 0 & \Rightarrow & (\vec{\lambda}_i)_j = (\vec{\lambda}_i^{lo})_j \\ (A_i \cdot \vec{\lambda} + \vec{b}_i)_j < 0 & \Rightarrow & (\vec{\lambda}_i)_j = (\vec{\lambda}_i^{hi})_j, \end{array}$$

therefore if we set $\vec{\lambda}_i^{lo} = \vec{0}$ and $\vec{\lambda}_i^{hi} = \overrightarrow{+\infty}$ then the bounded equality constraint turns into a greater-or-equal constraint (6.12). Similarly, by setting $\vec{\lambda}_i^{lo} = \overrightarrow{-\infty}$ and $\vec{\lambda}_i^{hi} = \vec{0}$ we get a less-or-equal constraint (6.14). Finally, by setting $\vec{\lambda}_i^{lo} = \overrightarrow{-\infty}$ and $\vec{\lambda}_i^{hi} = \overrightarrow{+\infty}$ we get an unbounded equality constraint (6.10). This shows that all previous constraints can be abstracted by bounded-equality constraints and if we have a procedure that is able to solve bounded-equality constraints we can also solve unbounded equality and inequality constraints.

Definition 11 *Acceleration-level constraint i is described by its dimensionality m_i, $m_i \times 6$ Jacobian blocks J_{i,A_i}, J_{i,B_i} due to the constraint and the constrained bodies A_i, B_i, right-hand side m_i-vector $\vec{c}_i$ and $\vec{\lambda}_i$ limits $\vec{\lambda}_i^{lo} \leq \vec{0}$ and $\vec{\lambda}_i^{hi} \geq \vec{0}$, where*

$$J_{i,A_i} \cdot \vec{a}_{A_i} + J_{i,B_i} \cdot \vec{a}_{B_i} = \vec{c}_i \tag{6.17}$$

is the corresponding unbounded equality constraint and $\vec{F}_c^i = J_i^T \cdot \vec{\lambda}_i, \vec{\lambda}_i^{lo} \leq \vec{\lambda}_i \leq \vec{\lambda}_i^{hi}$ is the constraint force due to the constraint.

On the other side, although it is not practical, it can be shown that all previous constraints can be converted to a set of greater-or-equal constraints. This is obvious for less-or-equal constraints $\vec{C}_i \leq \vec{0}$, that can be converted by negating the constraint Jacobians and the right-hand side vector $\vec{c}_i$, obtaining $-\vec{C}_i \geq \vec{0}$, and unbounded equality constraints $\vec{C}_i = \vec{0}$, that can be converted to two greater-or-equal constraints $\vec{C}_i \geq \vec{0}$ and $-\vec{C}_i \geq \vec{0}$.

6.2.4 Reduction to LCP

In the previous sections we have developed conditions on $\vec{\lambda}$ due to a given constraint i. These conditions were different for different types of constraints. Equality constraints contributed (6.10), inequality constraints contributed (6.12) and (6.14) and bounded equality constraints contributed (6.15). Obviously, our concern is to compute the ultimate constraint force $\vec{F}_c$ to be exerted on the system and to do so, we have to solve for $\vec{\lambda}$, obeying the conditions due to individual constraints.

Conditions (6.12) are well-known and algorithms that can solve for $\vec{\lambda}$ constrained in this way exists. The general name for this problem is the *linear complementarity problem* (LCP). Since our constraints can be converted to a set of greater-or-equal constraints expressed in terms of (6.12), we could use those algorithms to compute $\vec{\lambda}$. We will, however, present an outline of an algorithm that can solve (6.15) directly, which is more practical.

From now on we will conceptually treat all constraint DOFs as c separate 1-DOF constraints that are ordered in such a way that unbounded equalities come first and are followed by greater-or-equal constraints or bounded-equality constraints[4]. Hence $\vec{\lambda} = (\lambda_1, \ldots, \lambda_c)$, $\vec{b} = (b_1, \ldots, b_c)$, $\vec{\lambda}^{lo} = (\lambda_1^{lo}, \ldots, \lambda_c^{lo})$ and $\vec{\lambda}^{hi} = (\lambda_1^{hi}, \ldots, \lambda_c^{hi})$ ($\lambda_i, b_i, \lambda_i^{lo}, \lambda_i^{hi}$ will be scalars and so will not be accented with $\vec{\ }$).

LCP problems

Let us look more closely at the ultimate conditions on $\vec{\lambda}$, combined from the conditions due to individual constraints. If all constraints are unbounded equalities (6.10), we get

$$A \cdot \vec{\lambda} + \vec{b} = \vec{0},$$

which is a linear system consisting of c conditions and can be solved by standard factorization techniques. If all constraints are greater-or-equal constraints (6.12), we get a *linear complementarity problem* (LCP, pure LCP) of the form

$$\begin{aligned} A \cdot \vec{\lambda} + \vec{b} &\geq \vec{0} \\ \vec{\lambda} &\geq \vec{0} \\ \vec{\lambda} \cdot (A \cdot \vec{\lambda} + \vec{b}) &= 0, \end{aligned}$$

[4]The special treatment of unbounded equalities will improve the solver performance.

which can be solved by a LCP solver, [12], [13], [7]. If there are k unbounded equalities and $c-k$ greater-or-equal constraints, we get a *mixed linear complementarity problem* (mixed LCP)

$$\begin{aligned} A_{eq}\cdot\vec{\lambda}+\vec{b}_{eq} &= 0 \\ A_{ineq}\cdot\vec{\lambda}+\vec{b}_{ineq} &\geq \vec{0} \\ \vec{\lambda}_{ineq} &\geq \vec{0} \\ \vec{\lambda}_{ineq}\cdot(A_{ineq}\cdot\vec{\lambda}+\vec{b}_{ineq}) &= 0, \end{aligned}$$

where X_{eq} denotes the rows of X due to equality constraints i $(1\leq i\leq k)$ and X_{ineq} denotes the rows of X due to greater-or-equal constraints i $(k+1\leq i\leq c)$, which can be solved by a mixed LCP solver, [7]. Finally, if there are k unbounded equalities and $c-k$ bounded-equality constraints (this includes inequalities with appropriately set $\vec{\lambda}_i$ limits), we get a *lo-hi linear complementarity problem* (lo-hi LCP, [22]),

$$\begin{aligned} A_{eq}\cdot\vec{\lambda}+\vec{b}_{eq} &= \vec{0} \\ \lambda_i^{lo}\leq &\ \lambda_i \leq \lambda_i^{hi} \\ \lambda_i=\lambda_i^{lo} &\Rightarrow A_i\cdot\vec{\lambda}+b_i\geq 0 \\ \lambda_i=\lambda_i^{hi} &\Rightarrow A_i\cdot\vec{\lambda}+b_i\leq 0 \\ \lambda_i^{lo}<\lambda_i<\lambda_i^{hi} &\Rightarrow A_i\cdot\vec{\lambda}+b_i=0, \end{aligned}$$

where i, $k+1\leq i\leq c$ is an index of a bounded equality constraint.

6.2.5 Dantzig LCP Solver

In this section we will present an outline of the Dantzig algorithm for solving mixed LCP problems generalized for solving lo-hi LCP problems. The algorithm is analytical, operates in terms of the dense matrix A and its implementation is based on the approach pursued by [7] and simplified according to the implementation from [22].

We will use the following notation. If I and J are (ordered) index sets, $I=\{i_1,\ldots,i_n\}$, $J=\{j_1,\ldots,j_m\}$ and $\vec{x}$ is a vector of length $>n$ and A is a matrix, whose number of rows $>n$ and number of columns $c>m$, then $\vec{x}_I$ denotes a slice of $\vec{x}$ due to I (a $n\times 1$ vector), A_I denotes a row-slice of A due to I (a $n\times c$ matrix) and A_{IJ} denotes a slice of A due to I,J (a $n\times m$ matrix), so that

$$\vec{x}_I=(\vec{x}_{i_1},\ldots,\vec{x}_{i_n}) \qquad A_I=\begin{pmatrix}A_{i_1}\\ \vdots\\ A_{i_n}\end{pmatrix} \qquad A_{IJ}=\begin{pmatrix}A_{i_1,j_1} & \ldots & A_{i_1,j_m}\\ \vdots & \ddots & \vdots\\ A_{i_n,j_1} & \ldots & A_{i_n,j_m}\end{pmatrix}.$$

To improve readability, it is often written A_i instead of $A_{\{i\}}$, etc.

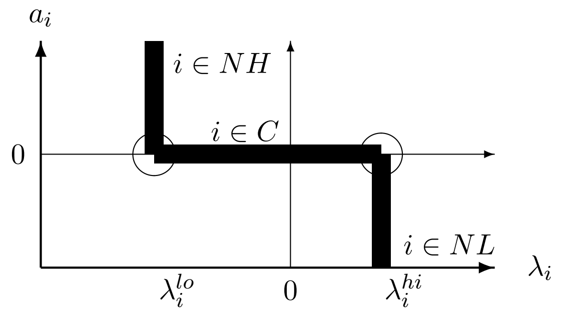
Overview

We can reformulate the lo-hi LCP problem as follows. Given a matrix A $(c\times c)$ and $c\times 1$ vectors $\vec{b}$, $\vec{\lambda}^{lo}\leq\vec{0}$ and $\vec{\lambda}^{hi}\geq\vec{0}$ we have to find a vector $\vec{\lambda}$ $(c\times 1)$ so that exactly one condition from the following set would hold for a given i (complementarity conditions)

$$\lambda_i=\lambda_i^{lo}\ \wedge\ a_i\geq 0 \qquad \lambda_i=\lambda_i^{hi}\ \wedge\ a_i\leq 0 \qquad \lambda_i^{lo}<\lambda_i<\lambda_i^{hi}\ \wedge\ a_i=0, \tag{6.18}$$

where for simplicity we refer to $A\cdot\vec{\lambda}+\vec{b}$ as to a vector $\vec{a}$ (note that $\vec{a}$ is a linear function of $\vec{\lambda}$).

Figure 6.1: Illustration of index sets, complementarity conditions and pivoting. Thick lines correspond to index sets — constraint i belongs to an index set if point (λ_i, a_i) lies on the thick line associated with the set. Circles indicate "singular points".



The algorithm starts with letting $\vec{\lambda} = \vec{0}$ (hence $\vec{a} = \vec{b}$) and then takes c iterations to update $\vec{\lambda}$. At the i-th iteration it computes λ_i so that (6.18) would hold for constraint i and in doing so it adjusts λ_j, $1 \leq j < i$ so that the conditions for constraints j, that were processed at previous iterations, would remain obeyed (constraints j, $j > i$ are ignored at the i-th iteration). When all c constraints are processed, all conditions are obeyed and $\vec{\lambda}$ contains the value we were solving for.

Index Sets

Index sets NH, NL, C are defined as a mutually disjoint subsets of constrained indices $\{1, \ldots, i\}$, where i is the number of the current iteration, that characterize what conditions of (6.18) are currently obeyed by constraints $j \leq i$. We will say that the constraint (index) j is

- *clamped* and belongs to a set C if $\lambda_j^{lo} < \lambda_j < \lambda_j^{hi} \ \wedge \ a_j = 0$.
- *non-clamped hi* and belongs to a set NH if $\lambda_j = \lambda_j^{lo} \ \wedge \ a_j > 0$.
- *non-clamped lo* and belongs to a set NL if $\lambda_j = \lambda_j^{hi} \ \wedge \ a_j < 0$.
- *non-clamped* and belongs to a set $NC = NH \cup NL$ if either $j \in NH$ or $j \in NL$.

Whenever updating λ_i, $\vec{a}$ changes as well and it must be ensured that each j, $j < i$ remains in one of the index sets so that (6.18) would not be violated for $j < i$ — j either remains in its current index set or transfers from an index set to another "adjacent" set (this is called *pivoting*).

Cases where $\lambda_j = \lambda_j^{lo} \ \wedge \ a_j = 0$ or $\lambda_j = \lambda_j^{hi} \ \wedge \ a_j = 0$ are "singular" and indicate that j should be transferred to a neighboring index set (see (6.1)). For example if $j \in C$ and λ_j reaches λ_j^{lo}, j is transferred to NH[5]. Constraint j can not be in NH (NL) if $\vec{\lambda}_j^{lo} = -\infty$ ($\vec{\lambda}_j^{hi} = +\infty$). Constraint j can not be in C if $\lambda_j^{lo} = \lambda_j^{hi} = 0$, such a constraint can be ignored.

Index sets and the pivoting process is illustrated by diagram (6.1) specific to a constraint i. Point (λ_i, a_i) in the diagram is restricted to lie on one of the three disjoint lines that correspond to index sets NH, C, NL and the pivoting corresponds to transfers of (λ_i, a_i) from one line to the other neighboring line when a "singular point" is reached.

[5]Due to the definition of NH, a_i must also be tweaked so that $a_i > 0$.

Iteration

The algorithm loops over i, at iteration i it computes λ_i and a_i while maintaining the complementarity conditions on λ_j and a_j for $j < i$. Indices $j > i$ are ignored at the i-th iteration. Let us look at the i-th iteration more closely.

When the i-th iteration is started, $\lambda_i = 0$ and $a_i = A_{i,C\cup NC} \cdot \vec{\lambda}_{C\cup NC} + b_i$. If (λ_i, a_i) already lies on a line of the complementarity diagram due to constraint i then (6.18) already holds for i, index i is assigned to the corresponding index set and the algorithm can proceed to the next iteration. Let us assume that (λ_i, a_i) is not on a line.

Our goal is to "drive" i to NH, C or NL. We will do so in multiple steps. At each step we compute a vector $\Delta\vec{\lambda}$, the adjustments to $\vec{\lambda}$, that will make (λ_i, a_i) "move towards" a line in the diagram. This will be looped until (λ_i, a_i) ends up on a line. Index i will be then assigned to the corresponding index set and the iteration will finish.

To ensure that conditions for constraints $j < i$ are not violated while taking a step during iteration i, we will require that clamped indices remain clamped and non-clamped indices remain non-clamped or a "singular point" is reached. If (λ_j, a_j) reaches a "singular point", j is transferred to the neighboring index set. We will now derive what $\Delta\vec{\lambda}$ should look like.

- Since we are not interested in constraints $j > i$ at the i-th iteration, $\Delta\vec{\lambda}_{j>i} = \vec{0}$.

- If $a_i > 0$ we would like to decrease λ_i so that i would eventually reach NH, hence $\Delta\lambda_i$ should be a negative non-zero value. If $a_i < 0$ we would like to increase λ_i so that i would eventually reach NL, hence $\Delta\lambda_i$ should be a positive non-zero value.

  However we do not know what the exact value of $\Delta\lambda_i$ should be. To tackle this problem we will write $\lambda_{dir} = \Delta\lambda_i = \mp 1$ (a λ "direction") with a note that we will actually add $\Delta\vec{\lambda} \cdot scale$ to $\vec{\lambda}$ and $scale > 0$ (a scaling factor) will be determined later.

- If $\Delta\vec{\lambda} \cdot scale$ was added to $\vec{\lambda}$, $\vec{a} = A \cdot \vec{\lambda} + \vec{b}$ would change by $\Delta\vec{a} \cdot scale = A \cdot \Delta\vec{\lambda} \cdot scale$. From the requirement that *clamped indices remain clamped or reach "singular points" and are transferred to* NH *or* NL we get a condition $\Delta\vec{a}_C = \vec{0}$ and from the requirement that *non-clamped indices remain non-clamped or reach "singular points" and are transferred to* C we get $\Delta\vec{\lambda}_{NC} = \vec{0}$. Putting these two requirements together we get that $\Delta\vec{\lambda}_C = -A_{CC}^{-1} \cdot A_{C,i} \cdot \Delta\lambda_i$.

  This is easy to show. Since we work with matrix slices we can assume (without loss of generality) that $C = \{1, \ldots, k\}$ and $NC = \{k+1, \ldots, i-1\}$. Constraints $j > i$ are ignored at this iteration and to simplify writing we can assume that this is the last iteration. Then, utilizing the requirement $\Delta\vec{\lambda}_{NC} = \vec{0}$, $\Delta\lambda_i = \lambda_{dir}$ and letting $\Delta\lambda_C = \vec{x}$,

$$A = \begin{pmatrix} A_{CC} & A_{C,NC} & A_{C,i} \\ A_{NC,C} & A_{NC,NC} & A_{NC,i} \\ A_{i,C} & A_{i,NC} & A_{ii} \end{pmatrix} \qquad \Delta\vec{\lambda} = \begin{pmatrix} \vec{x} \\ \vec{0} \\ \lambda_{dir} \end{pmatrix}$$

$$\Delta\vec{a} = A \cdot \Delta\vec{\lambda} = \begin{pmatrix} A_{CC} \cdot \vec{x} + A_{C,i} \cdot \lambda_{dir} \\ A_{NC,C} \cdot \vec{x} + A_{NC,i} \cdot \lambda_{dir} \\ A_{i,C} \cdot \vec{x} + A_{ii} \cdot \lambda_{dir} \end{pmatrix}$$

  and since it is required that $\Delta\vec{a}_C = \vec{0}$ we get that $\vec{x} = \Delta\lambda_C = -A_{CC}^{-1} \cdot A_{C,i} \cdot \lambda_{dir}$.

We thus have

$$\Delta\vec{\lambda}_C = -A_{CC}^{-1} \cdot A_{C,i} \cdot \lambda_{dir},\ \Delta\vec{\lambda}_{NC} = \vec{0},\ \Delta\vec{\lambda}_i = \lambda_{dir},\ \Delta\vec{\lambda}_{j>i} = 0 \tag{6.19}$$

and

$$\Delta\vec{a}_C = \vec{0},\ \Delta\vec{a}_{NC} = A_{NC,C} \cdot \Delta\vec{\lambda}_C + A_{NC,i} \cdot \lambda_{dir},\ \Delta\vec{a}_i = A_{i,C} \cdot \Delta\vec{\lambda}_C + A_{ii} \cdot \lambda_{dir} \tag{6.20}$$

and $\Delta\vec{a}_{j>i}$ can be treated as $\vec{0}$ because $\vec{a}_{j>i}$ will be computed at the beginning of the j-th iteration from $\vec{\lambda}$ and it will not be queried until then.

To take a step we have to determine an appropriate *scale* and then update $\vec{\lambda}$ and $\vec{a}$ by $\Delta\vec{\lambda} \cdot scale$ and $\Delta\vec{a} \cdot scale$. The goal is to ensure that i either ends up on an index set and the iteration can thus finish or $j < i$ reaches an end of the line in the diagram and needs be transferred to the neighboring line. We will now derive some conditions on *scale*. Note that $\Delta\vec{a}_C \cdot scale = \vec{0}$ and $\Delta\vec{\lambda}_{NC} \cdot scale = \vec{0}$ regardless of *scale* and thus we need be interested in $\Delta\vec{\lambda}_C \cdot scale, \Delta\vec{a}_{NC} \cdot scale, \Delta\lambda_i \cdot scale$ and $\Delta a_i \cdot scale$ only.

- If $\lambda_{dir} = -1$ $(a_i > 0)$ our desire is to push (λ_i, a_i) towards NH or C. Hence we have conditions $\lambda_i + \Delta\lambda_i \cdot scale \geq \lambda_i^{lo}$ and $a_i + \Delta a_i \cdot scale \geq 0$. If the first condition "wins"[6] then $(\lambda_i + \Delta\lambda_i \cdot scale = \lambda_i^{lo}, a_i + \Delta a_i \cdot scale > 0)$ and i should be assigned to NH. If the second condition "wins" then $(\lambda_i + \Delta\lambda_i \cdot scale > \lambda_i^{lo}, a_i + \Delta a_i \cdot scale = 0)$ and i should be assigned to C because $\Delta\lambda_i = \lambda_{dir} = -1$ and so also $\lambda_i < \lambda_i^{hi}$. Therefore

$$scale \leq \lambda_i - \lambda_i^{lo} \qquad \Delta a_i < 0 \Rightarrow scale \leq \frac{a_i}{\Delta a_i}.$$

- If $\lambda_{dir} = 1$ $(a_i < 0)$ our desire is to push (λ_i, a_i) towards NL or C. Hence we have conditions $\lambda_i + \Delta\lambda_i \cdot scale \leq \lambda_i^{hi}$ and $a_i + \Delta a_i \cdot scale \leq 0$. If the first condition "wins" then $(\lambda_i + \Delta\lambda_i \cdot scale = \lambda_i^{hi}, a_i + \Delta a_i \cdot scale < 0)$ and i should be assigned to NL. If the second condition "wins" then $(\lambda_i + \Delta\lambda_i \cdot scale < \lambda_i^{hi}, a_i + \Delta a_i \cdot scale = 0)$ and i should be assigned to C because $\Delta\lambda_i = \lambda_{dir} = 1$ and so also $\lambda_i > \lambda_i^{lo}$. Therefore

$$scale \leq \lambda_i^{hi} - \lambda_i \qquad \Delta a_i > 0 \Rightarrow scale \leq \frac{-a_i}{\Delta a_i}.$$

- If $j \in C$ then our desire is to keep j clamped or transfer it to NH or NL if a "singular point" is reached. Hence we have conditions $\lambda_j + \Delta\lambda_j \cdot scale \geq \lambda_j^{lo}$ and $\lambda_j + \Delta\lambda_j \cdot scale \leq \lambda_j^{hi}$ (a_j remains zero regardless of *scale*). If the first condition "wins" then $(\lambda_j + \Delta\lambda_j \cdot scale = \lambda_j^{lo}, a_j = 0)$ and j is transferred to NH (a_j is adjusted so that $a_j > 0$). If the second condition "wins" then $(\lambda_j + \Delta\lambda_j \cdot scale = \lambda_j^{hi}, a_j = 0)$ and j is transferred to NL (a_j is adjusted so that $a_j < 0$). Therefore

$$\Delta\lambda_j < 0 \Rightarrow scale \leq \frac{\lambda_j^{lo} - \lambda_j}{\Delta\lambda j} \qquad \Delta\lambda_j > 0 \Rightarrow scale \leq \frac{\lambda_j^{hi} - \lambda_j}{\Delta\lambda j}.$$

- If $j \in NH$ then our desire is to keep j in NH or transfer it to C if a "singular point" is reached. Hence we have a condition $a_j + \Delta a_j \cdot scale \geq 0$ (λ_j remains equal to λ_j^{lo} regardless of *scale*). If the condition "wins" then $(\lambda_j = \lambda_j^{lo}, a_j + \Delta a_j \cdot scale = 0)$ and j is transferred to C (λ_j is adjusted so that $\lambda_j > \lambda_j^{lo}$). Therefore

$$\Delta a_j < 0 \Rightarrow scale \leq \frac{-a_j}{\Delta a_j}.$$

[6] It is the condition on *scale* that limits it the most.

- If $j \in NL$ then our desire is to keep j in NL or transfer it to C if a "singular point" is reached. Hence we have a condition $a_j + \Delta a_j \cdot scale \leq 0$ (λ_j remains equal to λ_j^{hi} regardless of *scale*). If the condition "wins" then ($\lambda_j = \lambda_j^{hi}, a_j + \Delta a_j \cdot scale = 0$) and j is transferred to C (λ_j is adjusted so that $\lambda_j < \lambda_j^{hi}$). Therefore

$$\Delta a_j > 0 \Rightarrow scale \leq \frac{-a_j}{\Delta a_j}.$$

Any $scale > 0$, that satisfies the above conditions, ensures that complementarity conditions will remain valid for constraints $j < i$. Naturally, we would like to choose *scale* as big as possible so that i would reach an index set within one step and the iteration could thus finish — a condition on scale due to i would have to "win". This is not however always possible because the other conditions on *scale* may limit the required scale and another condition might "win". In that case there would exist an index j that would have to be transferred from one set to another set and we would have to take other step(s) to push i to an index set. We will call the maximum valid *scale* the *maximum step max_step* and the procedure that updates $\vec{\lambda}$, $\vec{a}$ and the index sets the *iteration step.*

Let us assume that we have procedures

- **compute_delta_lambda** and **compute_delta_a**, which compute $\Delta\vec{\lambda}$ and $\Delta\vec{a}$ according to (6.19) and (6.20) with respect to the current index sets C, NH, NL,
- **compute_max_step**, which computes the maximum step *max_step*, an index j that limited *max_step* the most (according to conditions on *scale*), an indicator of the index set S_{old} that the index j currently belongs to and an indicator S_{new} that the index j should be transferred or assigned to.

The work performed by the i-the iteration can then be summarized by the following pseudo-code.

```
iteration( i )
    {
    $a_i = A_{i,C\cup NC} \cdot \vec{\lambda}_{C\cup NC} + b_i$;
    if( $(\lambda_i, a_i)$ on $NL$ ) {$NL = NL \cup \{i\}$; return; }
    if( $(\lambda_i, a_i)$ on $NH$ ) {$NH = NH \cup \{i\}$; return; }
    if( $(\lambda_i, a_i)$ on $C$ ) {$C = C \cup \{i\}$; return; }
    while( true )
        {
        compute_delta_lambda( $\Delta\vec{\lambda}$ );
        compute_delta_a( $\Delta\vec{a}$ );
        compute_max_step( max_step, $j$, $S_{old}$, $S_{new}$ );
        $\vec{\lambda} = \vec{\lambda} + \Delta\vec{\lambda} \cdot max\_step$;
        $\vec{a} = \vec{a} + \Delta\vec{a} \cdot max\_step$;
        if( j == i ) {$S_{new} = S_{new} \cup \{i\}$; return; }
        $S_{old} = S_{old} \setminus \{j\}$;
        $S_{new} = S_{new} \cup \{j\}$;
        }
    }
```

Numerical Issues

Until now we have assumed that $max\_step > 0$. It can be shown that if all constraints are inequalities ($\lambda_j^{lo} = 0 \ \wedge \ \lambda_j^{hi} = +\infty$) and the constraints are feasible and not redundant, then $max\_step > 0$ and the algorithm always terminates, [7]. But we can not tell in advance whether all constraints can be satisfied or whether they are redundant. As a result, the algorithm can either fail to compute a valid *max_step* (either *max_step* $\leq \epsilon$ or *max_step* $= +\infty$ — no conditions on *scale* could be applied at the given context) or it might get stuck in an infinite loop transferring a certain index j between two index sets and not making any further progress. To correct this, extra checks that validate *max_step* and index transfers must be incorporated into the algorithm.

Finally, precision correction should be implemented. The purpose of the correction is to clamp (λ_j, a_j) so that if j should belong to an index set S then the point (λ_j, a_j) will really lie on the corresponding line in the diagram, avoiding the "singular points". For example if j transfers from C to NH it must be ensured that $a_j > 0$ after the transfer and a_j thus needs be tweaked so that $a_j \geq \epsilon$, where ϵ is a small positive constant.

The algorithm can be easily adapted to handle k unbounded equalities and $c-k$ bounded equalities, exploiting the special properties of the unbounded equalities. If constraints are indexed so that the first k constraints correspond to the unbounded equalities, we can let $C = \{1, \ldots, k\}$, $\vec{a}_C = \vec{0}$, $\vec{\lambda}_C = -A_{CC}^{-1} \cdot \vec{b}_C$ and start off with the $(k+1)$-th iteration directly.

Also note that we have to compute $\Delta\vec{\lambda}_C$ at each iteration step, which involves the evaluation of A_{CC}^{-1} (say, by a factorization of A_{CC}). At the end of the step either i is assigned to NH, C, NL or an index $j < i$ transfers from an index set to another set ($NH \to C$, $C \to NH$, $C \to NL$, $NL \to C$) — j is either added to or removed from C. Incorporating clever factorization techniques, the factorization of A_{CC} at the current iteration step can actually be computed from the factorization of $A_{C'C'}$ from the previous step (C' is the index set C from the previous step), thus improving the solver performance, [22].

6.3 Velocity-Level Constraints

In this section we will introduce a concept of impulses that allow to handle velocity-level constraints directly, including constraints with non-constant right-hand sides, without the need for going to the acceleration-level. We will use the notation from previous sections and base our discussion upon the work of [10] and [13].

6.3.1 Impulses

Let us return to the motivational example from section (6.2.2). We had an 1-DOF inequality position-level constraint i defined as $C_p(\vec{r}(t)) \geq 0$. We assumed that $C_p(\vec{r}(t)) = 0$ and $\dot{C}_p(\vec{r}(t)) = 0$ at the current time t and could enforce the original position-level constraint by maintaining the corresponding $\ddot{C}_p(\vec{r}(t)) \geq 0$ acceleration-level constraint.

Now let us assume that $\dot{C}_p(\vec{r}(t)) < 0$. In this case the rigid bodies *no longer rest at the boundary of the valid set of body positions* in the position product space, but the boundary is approached with a non-zero relative velocity. Corresponding $\dot{C}_p(\vec{r}(t)) \geq 0$ velocity-level constraint is violated at the current time t and the original (position-level) constraint can not be implemented on the acceleration level by maintaining $\ddot{C}_p(\vec{r}(t)) \geq 0$ (no matter how strong the constraint force is, it will require a non-zero time amount to bring $\dot{C}_p(\vec{r}(t))$ (and later on also $C_p(\vec{r}(t))$) to a positive value) — unless the body velocities are changed directly, the original constraint will be violated at the next time step.

For example if we (again) interpreted the position-level constraint as a non-penetration constraint between a ball and a brick wall then $C_p(\vec{r}(t)) = 0$ and $\dot{C}_p(\vec{r}(t)) < 0$ would indicate that the ball had struck the wall and the ball velocity would have to be changed abruptly at time t in order to prevent penetration at the next time step. In reality, however, the time duration of the collision is not zero, bodies involved in the collision deform and restoration forces with large magnitudes exerted during the period of the collision change the body velocity[7]. Since this would be hard to simulate (the corresponding ODE would have to be integrated with an extremely small time step) we will postulate a concept of impulsive force and torque (impulses) that will allow to change body velocities instantly without producing stiff ODEs. The collision in our example could then be treated as if its time duration was zero.

Impulsive Force and Torque

Let us assume that we have a rigid body with the center of mass at the world-space position $\vec{x}(t)$ and $\vec{P}(t)$ and $\vec{L}(t)$ are the body linear and angular momentum. *Impulsive force* (impulse) $\vec{J}_F$ is postulated as a force with "units of momentum". When $\vec{J}_F$ is applied to the body at the world-space position $\vec{r}$, then the linear momentum $\vec{P}(t)$ and the angular momentum $\vec{L}(t)$ change by $\Delta\vec{P}(t)$ and $\Delta\vec{L}(t)$, where

$$\Delta\vec{P}(t) = \vec{J}_F \qquad \Delta\vec{L}(t) = \vec{J}_\tau \qquad \vec{J}_\tau = (\vec{r} - \vec{x}(t)) \times \vec{J}_F \tag{6.21}$$

and $\vec{J}_\tau$ is the *impulsive torque* due to impulsive force $\vec{J}_F$ exerted on the rigid body at the world-space position $\vec{r}$. More precisely if $\vec{P}^-(t)$ and $\vec{L}^-(t)$ is the linear and angular momentum before the application of the impulse, then the linear and angular momentum $\vec{P}^+(t), \vec{L}^+(t)$ after the application of the impulse equals $\vec{P}^+(t) = \vec{P}^-(t) + \vec{J}_F$ and $\vec{L}^+(t) = \vec{L}^-(t) + (\vec{r} - \vec{x}(t)) \times \vec{J}_F$. Similarly to forces, $\vec{J}_F$ can be imagined as if it acted at the rigid body's center of mass ($\vec{J}_F$ affects the linear momentum only) and $\vec{J}_\tau$ as capturing the rotational effects due to $\vec{J}_F$ actually exerted at $\vec{r}$ ($\vec{J}_\tau$ affects the angular momentum only).

Impulsive forces and torques can be seen as "ordinary" forces and torques that change the body's linear and angular momentums directly, instead of affecting their time derivatives. Remember that if $\vec{F}_{total}(t)$ and $\vec{\tau}_{total}(t)$ are the total external force and torque exerted on the body, then $\frac{\partial}{\partial t}\vec{P}(t) = \vec{F}_{total}(t)$ and $\frac{\partial}{\partial t}\vec{L}(t) = \vec{\tau}_{total}(t)$. In the case of impulses, however, the impulsive force $\vec{J}_F$ and impulsive torque $\vec{J}_\tau$ change the linear and angular momentums directly, $\Delta\vec{P}(t) = \vec{J}_F$ and $\Delta\vec{L}(t) = \vec{J}_\tau$. Note the correspondence of the impulsive torque $\vec{J}_\tau = (\vec{r} - \vec{x}(t)) \times \vec{J}_F$ due to the impulsive force $\vec{J}_F$ exerted on the body at the world-space position $\vec{r}$ to the torque $\vec{\tau} = (\vec{r} - \vec{x}(t)) \times \vec{F}$ due to the force $\vec{F}$ exerted on the body at the world-space position $\vec{r}$.

Notation

We will extend our notation of generalized quantities by

- generalized momentum $\vec{F}^{total}_{imp}$ of a body

  A block vector $\vec{F}^{total}_{imp}$ consisting of two blocks storing the rigid body's linear momentum $\vec{P} \in \mathbf{R}^3$ and angular momentum $\vec{L} \in \mathbf{R}^3$, $\vec{F}^{total}_{imp} = (\vec{P}, \vec{L}) \in \mathbf{R}^6$. Generalized momentum serves the same purpose like the linear momentum $\vec{P}$ of particles.

  The reason for naming the generalized momentum in this way will become clear soon.

- generalized impulsive force (impulse) $\vec{F}_{imp}$ exerted on a body

  A block vector $\vec{F}_{imp}$ consisting of two blocks storing the impulsive force $\vec{J}_F \in \mathbf{R}^3$ (imagined as acting at the body's center of mass) and impulsive torque $\vec{J}_\tau \in \mathbf{R}^3$ due to $\vec{J}_F$, $\vec{F}_{imp} = (\vec{J}_F, \vec{J}_\tau) \in \mathbf{R}^6$.

[7]The restoration forces can be imagined as constraint forces due to $\ddot{C}_p(\vec{r}(t)) \geq 0$ constraints with position stabilization terms like in (4.8).

Impulsive Dynamics

If M is the mass matrix of the body and $\vec{v}$ is the rigid body's generalized velocity then, according to (5.10) and (5.12), we get that

$$M \cdot \vec{v} = \vec{F}_{imp}^{total}. \tag{6.22}$$

According to (6.21), if $\vec{F}_{imp}$ is applied to the body then the body's generalized momentum $\vec{F}_{imp}^{total}$ changes by $\Delta\vec{F}_{imp}^{total} = \vec{F}_{imp}$ and this change equals $\Delta\vec{F}_{imp}^{total} = \Delta(M \cdot \vec{v}) = M \cdot \Delta\vec{v}$, because impulses do not change M directly. We thus get that the change $\Delta\vec{v}$ of the body's velocity $\vec{v}$ due to $\vec{F}_{imp}$ equals

$$\Delta\vec{v} = M^{-1} \cdot \vec{F}_{imp}. \tag{6.23}$$

Equations (6.22) and (6.23) can be generalized by a single equation

$$M \cdot \vec{v} = \vec{F}_{imp}, \tag{6.24}$$

which describes the relation between an impulse $\vec{F}_{imp}$ and the change of velocity $\vec{v}$ due to the application of the impulse. Since the equation holds also for $\vec{F}_{imp} = \vec{F}_{imp}^{total}$ (thus capturing the relation between the momentum $\vec{F}_{imp}^{total}$ and the generalized body velocity $\vec{v}$ like in (6.22)), $\vec{F}_{imp}^{total}$ can be seen as a generalized *total external impulse* that consists of the only term — the inertial term $(\vec{P}, \vec{L})$. Equation (6.24) is the "impulsive variant" of (5.23).

Finally if we have a set of n rigid bodies then their impulsive dynamics is described by the following velocity-level analog of (6.6)

$$M \cdot \vec{v} = \vec{F}_{imp}, \tag{6.25}$$

where M is the mass matrix of the rigid body system, $\vec{F}_{imp} = (\vec{F}_1^{imp}, \ldots, \vec{F}_n^{imp})$, $\vec{v} = (\vec{v}_1, \ldots, \vec{v}_n)$ and $\vec{v}_j$ is the velocity of the j-th body due to impulse $\vec{F}_j^{imp}$ exerted on that body. For simplicity $\vec{F}_{imp}$ is called the impulsive force (impulse) exerted on the system and $\vec{v}$ the velocity of the system due to $\vec{F}_{imp}$ or the response of the system to $\vec{F}_{imp}$.

6.3.2 Constraints

We will now transfer acceleration-level formulations and results to the velocity level and present impulsive constraint forces. There is no need to derive the equations once more because the acceleration-level formulation of rigid body dynamics (5.23) exactly corresponds to the velocity-level formulation of the impulsive dynamics (6.24) and the derivations would be the same. This implies that (nearly) the same algorithms can be used to implement velocity-level constraints. The only differences are due to the fact that we will work with velocities $\vec{v}$, impulsive constraint forces $\vec{F}_{imp}^{c}$ and momentums $\vec{F}_{imp}^{total}$ instead of accelerations $\vec{a}$, constraint forces $\vec{F}_c$ and total external forces $\vec{F}_{total}$.

Generalizing (6.2) for arbitrary (non-constant) vector $\vec{k}$ and taking the same approach as we did to generalize (6.3), we can formulate velocity-level unbounded equality, greater-or-equal, less-or-equal and bounded equality constraints.

Velocity-level constraint i affecting bodies A_i and B_i is described by its dimensionality m_i, $m_i \times 6$ Jacobian blocks J_{i,A_i}, J_{i,B_i} due to the constraint and the constrained bodies A_i, B_i, right-hand side m_i-vector $\vec{k}_i$ and one of the following equations

$$J_{i,A_i} \cdot \vec{v}_{A_i} + J_{i,B_i} \cdot \vec{v}_{B_i} = \vec{k}_i \tag{6.26}$$

$$J_{i,A_i} \cdot \vec{v}_{A_i} + J_{i,B_i} \cdot \vec{v}_{B_i} \geq \vec{k}_i \tag{6.27}$$

$$J_{i,A_i} \cdot \vec{v}_{A_i} + J_{i,B_i} \cdot \vec{v}_{B_i} \leq \vec{k}_i, \tag{6.28}$$

where (6.26) describes the *velocity-level unbounded equality constraint* i, (6.27) the *velocity-level greater-or-equal constraint* and (6.28) the *velocity-level less-or-equal constraint.* Moreover if $\vec{\lambda}_i$ limits $\vec{\lambda}_i^{lo} \leq \vec{0}$ and $\vec{\lambda}_i^{hi} \geq \vec{0}$ are provided, then (6.26) describes the *velocity-level bounded equality constraint* i, where $(\vec{F}_c^i)_{imp} = J_i^T \cdot \vec{\lambda}_i$, $\vec{\lambda}_i^{lo} \leq \vec{\lambda}_i \leq \vec{\lambda}_i^{hi}$ is the bounded constraint impulse due to the constraint. Similarly to acceleration-level constraints, all velocity-level constraints of the above types can be abstracted by velocity-level bounded equality constraints.

We have spoken of $(\vec{F}_c^i)_{imp} = J_i^T \cdot \vec{\lambda}_i$ as of the constraint impulse due to constraint i. This deserves a little explanation. When we were deriving equations for acceleration-level equality constraints we were given a velocity-level equality constraint $\vec{C}_v(\vec{r}, \vec{v}) = \vec{0}$, assumed that the constrained bodies lied on the intersection of the hypersurfaces due to the velocity-level constraint DOFs in the *velocity product space* (the velocity-level constraint was maintained at the current time) and constrained the body *accelerations* $\ddot{\vec{r}}$ along the hypersurface normals so that the bodies remained on the hypersurfaces (the velocity-level constraint was thus maintained at the next time step). The hypersurface normals corresponded to the (scaled) rows of $J_i = \frac{\partial \vec{C}_v}{\partial \vec{v}}$ and hence the *constraint force* we were to compute due to the constraint i was of the form $\vec{F}_c^i = J_i^T \cdot \vec{\lambda}_i$, where $\vec{\lambda}_i$ was a vector of unknown multipliers due to the constrained DOFs.

In the case of velocity-level equality constraints, we are given a position-level equality constraint $\vec{C}_p(\vec{r}) = \vec{0}$, assume that the constrained bodies lie on the intersection of the hypersurfaces due to the position-level constraint DOFs in the *position product space* (the position-level constraint is maintained at the current time) and constrain the body *velocities* $\dot{\vec{r}}$ along the hypersurface normals so that the bodies remain on the hypersurfaces (the position-level constraint will thus be maintained at the next time step). The hypersurface normals correspond to the (scaled) rows of $J_i = \frac{\partial C_p}{\partial \vec{r}}$ and hence the *constraint impulse* we are to compute due to the constraint i is of the form $(\vec{F}_c^i)_{imp} = J_i^T \cdot \vec{\lambda}_i$, where $\vec{\lambda}_i$ is a vector of unknown multipliers due to the constrained DOFs.

Now if we have c velocity-level equality constraints we would like to compute the ultimate constraint impulse $\vec{F}_{imp}^c = J^T \cdot \vec{\lambda}$, where $\vec{\lambda} = (\vec{\lambda}_1, \ldots, \vec{\lambda}_c)$ is a block vector of the unknown Lagrange multipliers due to individual constraints. To do so, we would like to formulate certain conditions on $\vec{\lambda}$, solve for $\vec{\lambda}$ and yield $\vec{F}_{imp}^c$. Starting with (6.25), following the derivation of (4.6) and utilizing the constraint formulation (6.26) we come up with the following conditions on $\vec{\lambda}$,

$$A \cdot \vec{\lambda} + \vec{b} = \vec{0}, \tag{6.29}$$

where $A = J \cdot M^{-1} \cdot J^T$ is a $m \times m$ matrix, m is the total number of DOFs removed by the constraints, $m = \sum_{i=1}^{c} m_i$, $\vec{b} = J \cdot M^{-1} \cdot \vec{F}_{imp}^{total} - \vec{k}$ is a $m \times 1$ vector, $\vec{k} = (\vec{k}_1, \ldots, \vec{k}_c)$ and $\vec{F}_{imp}^{total} = ((\vec{F}_{imp}^{total})_1, \ldots, (\vec{F}_{imp}^{total})_n)$ is the total external impulse exerted on the system (a vector of generalized momentums).

Note that conditions (6.29) match (6.8), only $\vec{F}_{imp}^{total}$ is used instead of $\vec{F}_{total}$ while forming the vector $\vec{b}$. Vector $\vec{\lambda}$ can be thus computed analogously to the acceleration-level case. Inequalities and bounded equalities impose different conditions on $\vec{\lambda}$ and are formulated directly in terms of matrix A and vector $\vec{b}$ and so do not depend on whether they are derived with respect to the hypersurfaces in the velocity product space (acceleration-level constraints) or position product space (velocity-level constraints), hence (6.10), (6.12), (6.14) and (6.15) apply to corresponding velocity-level constraints and the problem of computing $\vec{\lambda}$ for the constraint impulse to be exerted on the system is also a LCP.

6.3.3 Handling Constraints

One might wonder why we ever tackle the problem of acceleration-level constraints if we can control the body velocity directly. The answer is simple. Impulses applied to a rigid body change the body state directly and whenever such a change occurs, the ODE solver must be stopped and restarted from the new changed state, which might be a costly operation. More formally if $\vec{y}(t) = (\vec{y}_1(t), \ldots, \vec{y}_n(t))$ is the system state of n bodies, where $\vec{y}_i(t)$ is the state of body i given by (5.17), and an impulse is applied to body j, then the body state $\vec{y}_j(t) = (\vec{x}(t), R(t), \vec{P}(t), \vec{L}(t))$ turns to $\vec{y}_j^{+}(t) = (\vec{x}(t), R(t), \vec{P}^{+}(t), \vec{L}^{+}(t))$ and hence $\vec{y}(t)$ is discontinuously changed to $\vec{y}^{+}(t)$, which must be treated as a new initial state.

The other problem with impulses relates to the way how ODE solvers work. Given a current state $\vec{y}(t)$ and a history of k states from the last k steps, the solver predicts $\vec{y}(t + \Delta t)$ by utilizing the state history and evaluating the system state derivative at one or more points $\vec{y}$. Equation (5.18) shows that $\frac{\partial}{\partial t}\vec{y}(t) = (\dot{\vec{x}}(t), \vec{\omega}(t)^{*} \cdot R(t), \vec{F}_{total}(t), \vec{\tau}_{total}(t))$, hence in order to capture the effects of constraints while estimating $\vec{y}(t + \Delta t)$ (sampling the vector field), the constraints must be formulated on the acceleration level.

Modifications To ODE Solver Interface

In order to handle constraints on the velocity level, modifications to **evaluate_derivative** function from the ODE interface have to be made. The purpose of the function is to compute the system state derivative for the current system state, following (5.18). If the system state is going to be changed directly by the code in the function, then the derivative should be computed with respect to the changed state and the ODE solver asked to restart from the modified state. This can be abstracted as follows. The function will take another argument that will indicate whether the current state is intermediate (see below) and will return another indicator that will tell whether the current state was changed and the derivative computed with respect to the new state,

- If the current state $\vec{y}(t)$ is intermediate, that is, **evaluate_derivative** is invoked just for the purposes of the evaluation of $\vec{f}(t, \vec{y}(t))$, in order to estimate $\vec{y}(t + \Delta t)$, there is no point in applying impulses to bodies, because the state is intermediate and its change will be ignored. Hence velocity-level constraints have no effect at intermediate states and can be ignored at those states.

- If the current state is regular (as opposed to an intermediate state), system state change could be accepted by the ODE solver and if the state is actually changed, the solver must be notified so that it could restart self from the new state.

  In this case, velocity-level constraints are enforced first. This changes the system state and the derivative of the changed state is computed and returned to the ODE solver then.

Velocity-Level or Acceleration-Level

Acceleration-level constraints might implement position-level or velocity-level constraints under an assumption that the current state is consistent with the velocity-level and position-level formulation of the constraint (constraints hold at the current time). If an impulse is applied to a body, this assumption might be violated[8]. To overcome this, all position-level and velocity-level

[8]For example if we have a position-level constraint $\vec{C}_p = \vec{0}$, the constraint can be maintained on the acceleration level as $\ddot{\vec{C}}_p = \vec{0}$ if $\vec{C}_p = \vec{0}$ and $\dot{\vec{C}}_p = \vec{0}$ at the current time. If an impulse is applied to a constrained body then the

constraints must also be enforced on the velocity level when an impulse is going to be applied and/or constraint stabilization (4.8) or (4.9) must be incorporated.

Note that if we use a first order integration method (Euler solver) we can forget about acceleration-level constraints because all states are regular and there is actually no philosophical difference between a force and an impulse (impulses are forces times time step here, that is, knowing the step size, the effects of impulses can be simulated by appropriately scaled forces). However if using a higher order solver, we have to compute constraint forces at least at intermediate states so that constraints are considered when estimating $\vec{y}(t + \Delta t)$.

To conclude, acceleration-level constraints are preferred to velocity-level constraints, *where appropriate*, because velocity-level constraints change system state directly by the application of constraint impulses, independently on the used integration method (ODE solver) and do not play well with higher order methods (velocity-level constraints have no effect when enforced at intermediate states). Moreover, velocity-level constraints can violate the assumptions of acceleration-level constraints and thus break position-level and velocity-level constraints that are implemented on the acceleration level. On the other side, as we saw in the motivational example (a ball striking the wall), *certain constraints can not be enforced on the acceleration-level reliably* and must be implemented on the velocity level. This happens mostly in those cases when constraints to be enforced are not permanent and the system state is not currently consistent with the corresponding acceleration-level formulation of the constraint when the constraint becomes effective.

6.4 Position-Level Constraints

In this section we will summarize our knowledge of position-level equality constraints from section (6.1.2), define position-level equality constraints in the context of other types of constraints, so that it will be more consistent with the material from the previous sections, and tackle the problem of position-level inequality constraints. We will still work with c constraints and n bodies.

6.4.1 Constraint Formulation

Position-level constraint i affecting bodies A_i and B_i is described by its dimensionality m_i and one of the following m_i-vector equations

$$\begin{aligned}
\vec{C}_p^i(\vec{r}) &= \vec{0} \in \mathbf{R}^{m_i} && (6.30)\\
\vec{C}_p^i(\vec{r}) &\geq \vec{0} \in \mathbf{R}^{m_i} && (6.31)\\
\vec{C}_p^i(\vec{r}) &\leq \vec{0} \in \mathbf{R}^{m_i}, && (6.32)
\end{aligned}$$

where $\vec{r} = (\vec{r}_1, \dots, \vec{r}_n)$ is a block-vector of generalized body positions and $\vec{C}_p^i$ is a function of $\vec{r}_{A_i}$ and $\vec{r}_{B_i}$, formally a function of $\vec{r}$. Vectors $\vec{r}_{A_i}, \vec{r}_{B_i}$ that satisfy the constraint equation are the valid generalized positions of the constrained bodies due to the constraint. Equation (6.30) describes the *position-level equality constraint*, (6.31) describes the *position-level greater-or-equal constraint* and (6.32) the *position-level less-or-equal constraint*.

To simplify further discussion we will assume that $m_i = 1$ if the constraint represents a greater-or-equal or a less-or-equal constraint and will simply write C_p^i instead of $\vec{C}_p^i$ in these cases.

assumption $\dot{\vec{C}}_p = \vec{0}$ no longer holds and the velocity-level constraint $\dot{\vec{C}}_p = \vec{0}$ should be imposed.

Unlike acceleration-level or velocity-level constraints we are not be able to maintain position-level constraints directly on the position level and have to resort to corresponding velocity-level or acceleration-level formulations. This implies that whenever a velocity-level or an acceleration-level constraint is imposed, the current system state must be consistent with the original position-level constraint.

We will provide the velocity-level and acceleration-level formulations of the constraints with additional notes indicating when the particular formulation can be applied. Since acceleration-level formulations will be applicable only if the corresponding velocity-level formulation, expressed in terms of $\dot{C}_p^i$, already holds and the value of $\dot{C}_p^i$ is a function of impulses exerted on the system, we can not tell in advance what constraints can be implemented on the acceleration level reliably, without risking the violation of the assumptions and hence the violation of the constraint, *if there are velocity level constraints in effect*[9]. Therefore, as a general rule, we will either

- implicitly enforce all constraints also on the velocity level if there is at least one constraint i that *must be implemented on the velocity level* or
- choose between the acceleration-level or velocity-level formulation according to the value of $\dot{C}_p^i$ before impulses are applied, make up with the fact that the constraint can break if it is implemented on the acceleration level and there is at least one constraint that *must be implemented on the velocity level*, and rely on the constraint stabilization. See notes on page 68.

6.4.2 Equality Constraints

Position-level equality constraint i can be implemented either

- on the acceleration level, by requiring $\ddot{\vec{C}}_p^i = \vec{0}$ and starting from a state where $\vec{C}_p^i = \dot{\vec{C}}_p^i = \vec{0}$. Body states will be changed indirectly by constraint forces.
- on the velocity level, by requiring $\dot{\vec{C}}_p^i = \vec{0}$ and starting from a state where $\vec{C}_p^i = \vec{0}$. Body states will be changed directly by constraint impulses which might break other position-level or velocity-level constraints implemented on the acceleration level.

Following the standard derivation procedure and incorporating position and velocity stabilization terms from section (4.3) we get the formulations of constraint i on the velocity and acceleration levels

$$\begin{aligned} J_i \cdot \vec{v} = J_{i,A_i} \cdot \vec{v}_{A_i} + J_{i,B_i} \cdot \vec{v}_{B_i} &= -\vec{C}_p^i \cdot \alpha \\ J_i \cdot \vec{a} = J_{i,A_i} \cdot \vec{a}_{A_i} + J_{i,B_i} \cdot \vec{a}_{B_i} &= -\vec{C}_p^i \cdot \alpha - J_i \cdot \vec{v} \cdot \beta - \dot{J}_i \cdot \vec{v}, \end{aligned} \tag{6.33}$$

where $\alpha > 0$ and $\beta > 0$ are constants due to the Baumgarte stabilization terms from (4.8) and J_i is the Jacobian matrix due to the constraint, computed either from the position-level constraint $\vec{C}_p^i = \vec{0}$ as $J_i = \frac{\partial \vec{C}_p^i}{\partial \vec{r}}$ or the velocity-level constraint $\dot{\vec{C}}_p^i = \vec{0}$ as $J_i = \frac{\partial \dot{\vec{C}}_p^i}{\partial \vec{v}}$. Note that the acceleration-level and velocity-level formulations use the same Jacobian matrices J_i and only differ in their right-hand side vectors, where the acceleration-level formulation contains the extra $\dot{J}_i \cdot \vec{v}$ and $\dot{\vec{C}}_p^i$ stabilization terms.

[9]There is a sort of cyclic dependency. $\dot{C}_p^i$ determines whether constraint i must be implemented on the velocity level while simultaneously $\dot{C}_p^i$ is a function of the set of constraints to be implemented on the velocity level.

6.4.3 Inequality Constraints

Inequality (greater-or-equal or less-or-equal) constraints are different from equality constraints because no constraints are actually imposed until the constrained bodies reach the boundary of the valid set of body positions in the position product space. That is, unless $C_p^i = 0$ at the current time step, no constraints are in effect.

Let us suppose that we are going to handle greater-or-equal constraint[10] (6.31) which can be (for clarification) interpreted as a constraint between a ball and a wall, like in the motivational examples in sections (6.2.2) and (6.3.1). Vectors $\vec{r}$ satisfying $C_p^i(\vec{r}) \geq 0$ are the valid positions of bodies and parametrize the set of valid positions in the position product space. Vectors $\vec{r}$, where $C_p^i(\vec{r}) = 0$, lie on the boundary of the valid set of body positions in the position product space and vectors $\vec{r}$, where $C_p^i(\vec{r}) < 0$, are the invalid positions that must be avoided.

We will now discuss three cases, depending on whether $C_p^i(\vec{r}(t)) > 0$, $C_p^i(\vec{r}(t)) = 0$ or $C_p^i(\vec{r}(t)) < 0$ at the current time t.

In Interior

If $C_p^i(\vec{r}(t)) > 0$ at the current time t, bodies did not reach the boundary of the set of valid positions in the position product space yet and no constraint will be imposed. In the motivational example, this corresponds to the case when the ball is in front of the wall.

On Boundary

$C_p^i(\vec{r}(t)) = 0$ at the current time t, bodies are at the boundary of the set of valid positions in the position product space and the velocity-level constraint $\dot{C}_p^i(\vec{r}(t)) \geq 0$ must be imposed in order to ensure that the original position-level constraint $C_p^i \geq 0$ will remain obeyed. Depending on the fact whether the velocity-level constraint already holds or not, the constraint can be implemented either

- on the acceleration level, if $\dot{C}_p^i(\vec{r}(t)) = 0$, in the form of $\ddot{C}_p^i \geq 0$, or
- on the acceleration level, if $\dot{C}_p^i(\vec{r}(t)) > 0$, by ignoring the constraint, because the velocity-level constraint will hold regardless of the value of $\ddot{C}_p^i$ if a sufficiently small step size is taken, or
- on the velocity level in the form of $\dot{C}_p^i \geq 0$, with the risk of breaking other position-level or velocity-level constraints implemented on the acceleration level. The constraint must be enforced on the velocity level if $\dot{C}_p^i(\vec{r}(t)) < 0$.

The first case corresponds to the example in section (6.2.2), the second case describes a situation when the ball retreats from the wall and the third case corresponds to the motivational example in section (6.3.1). In all the three cases, the ball is in contact with the wall.

In the practice, due to integration error and the evaluation of $C_p^i(\vec{r}(t))$ at discrete time steps, we might never reach a state where $C_p^i(\vec{r}(t)) = 0$ holds exactly and the above test needs be evaluated with respect to a certain $\epsilon > 0$ precision (the same can be said about the $\dot{C}_p^i < 0$ test). For example if $|C_p^i(\vec{r}(t))| < \epsilon$ then the bodies are considered to lie on the boundary of the valid set of body positions in the position product space and if $\epsilon > \dot{C}_p^i(\vec{r}(t)) \geq -\epsilon$ then the acceleration

[10]If we were going to handle less-or-equal constraint (6.32) we could proceed analogously.

level constraint might still be imposed. Since constraint assumptions are not strictly obeyed, the constraint needs be stabilized.

This can be concluded by the following equation, which shows what constraint type should be imposed at the time t and what the formulation of the constraint is (stabilization terms are included),

$$\begin{aligned} \dot{C}_p^i < -\epsilon \quad &\Rightarrow \\ J_i \cdot \vec{v} = J_{i,A_i} \cdot \vec{v}_{A_i} + J_{i,B_i} \cdot \vec{v}_{B_i} \quad &\geq \quad -C_p^i \cdot \alpha \\ \epsilon > \dot{C}_p^i \geq -\epsilon \quad &\Rightarrow \\ J_i \cdot \vec{a} = J_{i,A_i} \cdot \vec{a}_{A_i} + J_{i,B_i} \cdot \vec{a}_{B_i} \quad &\geq \quad -C_p^i \cdot \alpha - J_i \cdot \vec{v} \cdot \beta - \dot{J}_i \cdot \vec{v}, \end{aligned} \tag{6.34}$$

where $\alpha > 0$ and $\beta > 0$ are the Baumgarte stabilization constants.

In Exterior

$C_p^i(\vec{r}(t)) < 0$ at the current time t, body positions are already invalid and no velocity-level constraint would be able to ensure that the original position-level constraint would be maintained at the next time step. In our motivational example, this corresponds to the case when the ball penetrates the wall.

This is a similar problem to the acceleration-level problem of preventing the ball from penetrating the wall when it struck the wall (motivational example from section (6.3.1)), which was solved by going to the velocity level from the acceleration level, but this time we are already on the velocity level. We would like to go from the velocity level to the position level and directly adjust the body positions so that the position-level constraint will be obeyed. Since this is hard to implement reliably we instead

- do not accept the invalid state and ask the ODE solver to stop, instruct it to rollback to the previous state which is considered still valid and take an alternative approach to locate time t_c, so that $C_p^i(\vec{r}(t_c)) = 0$, and transfer the problem to the previous case.

  This approach ensures that if the solver is always started from a valid state then *invalid states are avoided*. Obviously, such a behavior is essential when maintaining "important" constraints, such as non-penetration constraints when penetration must be strictly avoided.

  According to our motivational interpretation of C_p^i, this approach makes the ODE solver rollback to a state when the ball does not penetrate the wall yet and then locate time t_c when the ball is in contact with the wall.

- accept the invalid state, behave as if $|C_p^i(\vec{r}(t))| < \epsilon$ was true and rely on the constraint stabilization to bring the body positions back to the set of valid body positions in the position product space.

  Clearly, this behavior is suitable for "unimportant" constraints only whose violation does not matter. Despite of this, low-end simulators often implement position-level constraints in this way and are not able to avoid invalid states at all.

We will now pursue the first approach, that avoids invalid states, and look at the estimation of time t_c. If t_{prev} is the time of the previous regular state then $C_p^i(\vec{r}(t_{prev})) \geq 0$ because the state would not be accepted otherwise. Since the current state at time $t_{current} > t_{prev}$ is invalid, we have $C_p^i(\vec{r}(t_{current})) < 0$ and the state can not be accepted. By inspecting the current system state, the previous system state and $\vec{C}_p$ we would like to compute time t_c, $t_{prev} < t_c < t_{current}$ so that all states from t_{prev} to t_c are still valid and $\vec{C}_p(\vec{r}(t_c)) = 0$. If we knew t_c, we could rollback to t_{prev}, instruct the simulator to advance to t_c, handle the position-level constraint on the velocity or acceleration level according to (6.34) and simulate forward from t_c.

Since computing t_c exactly might be difficult, we can employ a *bisection* algorithm that solves for t_c with the above properties numerically, in terms of queries whether the ODE solver can advance to a desired state without encountering an invalid state, [10], [13]. We will denote $t_a < t_b$ the lower and upper bound estimates of t_c, initially set to $t_a = t_{prev}$, $t_b = t_{current}$. The algorithm will incrementally restrict the lower and upper bounds until $|t_a - t_b| < \delta$ and t_c is estimated with a sufficient precision $\delta > 0$. If $C_p^i(\vec{r}(t_a)) \geq 0$ and $C_p^i(\vec{r}(t_b)) < 0$ we can ask the ODE solver to rollback to t_a and advance to $t_m = (t_a + t_b)/2$ or any other $t_a < t_m < t_b$, where t_m is the estimate of t_c. If the solver reaches t_m without getting into an invalid state, we can assume that $C_p^i(\vec{r}(t)) \geq 0$ for $t_a \leq t \leq t_m$ and hence can update the lower bound so that $t_a = t_m$. If the solver gets to an invalid state while advancing from t_a to t_m, we can update the upper bound so that $t_b = t_m$. Bounds t_a and t_b are repeatedly updated until $|t_a - t_b| < \delta$. When the bisection algorithm finishes, $|C_p^i(\vec{r}(t_a))| < \epsilon$, we can let $t_c = t_a$, rollback to t_c, utilize (6.34) and simulate forward from t_c in the usual way.

This approach assumes that we are always able to enforce constraints at time t_c so that the bodies are not driven to an invalid state at time $t_c + \epsilon$. However, due to a limited integration and numerical precision, we might not always be able to do that. When the ODE solver detects that it can not advance from the current (still valid) state to the next state even when the smallest possible step of size ϵ is considered, the solver must rollback to t_c and notify the simulator to stop bodies involved in the constraint (reset their momentums to zero vectors) so that the simulation could advance further.

The value of t_c can be estimated from the time derivatives of C_p^i. This can, however, be problematic for a certain type of constraints. For example non-penetration constraints are formulated in terms of greater-or-equal position-level constraints that measure relative body distance at geometry contacts, but it is hard to define (or compute) what the contacts (formulations of C_p^i) should be when the bodies are already penetrating. Yet, according to results from collision detection routines, we can usually tell pretty easily whether the current state is valid or not.

Modifications to ODE Solver

In order to avoid invalid states and support and incorporate the bisection algorithm to the ODE solver, both the ODE solver interface and the implementation of the solver must be updated

- function **evaluate_derivative** must be extended to return the indicator whether the current state is valid or not. This can be done by evaluating C_p^i for all position-level inequality constraints. For example if i is the index of a position-level greater-or-equal constraint and $C_p^i < 0$ then the constraint is violated and if i is an “important” constraint then the current state should be considered invalid. This fact should be indicated to the ODE solver so that it could rollback to the previous (valid) state and take smaller steps to locate the latest valid state (bisection algorithm).

- the implementation of the ODE solver’s **advance** function must be updated to run the bisection algorithm when **evaluate_derivative**, invoked by the ODE solver, indicates that the current state is invalid. The bisection algorithm can be implemented in terms of **evaluate_derivative**, **get_state** and **set_state**, using the logic of the old **advance**.

6.5 Contact

In this section we will tackle the problem of non-penetration constraints and contact with friction. The problem of non-penetration constraints is to ensure that rigid body shapes (geometries) do not penetrate while the bodies are simulated, yet properly respond to forces and impulses. Non-penetration is enforced by introducing non-penetration constraints into the system, formulated as position-level greater-or-equal constraints and maintained incrementally just like other inequality constraints — assuming that the geometries do not penetrate at the initial state, non-penetration constraints ensure that the penetration will not occur at the next time step. The problem of contacts is to enforce non-penetration while modelling real-world body interaction, such as realistically-looking impacts (collisions) or friction, by imposing additional contact constraints.

The problem of non-penetration constraints with friction is inherently hard because it involves inequality constraints. This is well demonstrated by the efforts of David Baraff, who tackled the problem of non-penetration constraints for many years, [3], [6], [5], [7], [8].

Rigid body shapes will be represented by *collision geometries* that abstract the shapes for the purposes of the simulation. Since rigid bodies can not deform, body shapes can be defined in body spaces. Collision geometries are fixed in body spaces and rotate and translate with the bodies they are attached to as the bodies move in the world space.

6.5.1 Collision Detection

The purpose of the collision detection is to support non-penetration and contact constraints in the simulator. Given a system state $\vec{y}(t)$, collision detection must be able to determine whether there are any penetrating bodies (penetration indicates an invalid state and the simulation state must be rolled back) and, if no penetration is detected, find out what body geometries contact and at which points (contacts). Optionally, if body geometries are penetrating, collision detection might predict the time of collision to improve the performance of the bisection algorithm, [4], [13]. For more complex information on collision detection one might refer to [13] or [6].

Contacts can be imagined (and in fact it is desirable to treat or define them in this way) as the *coinciding points of the first and second geometry where the contact force/impulse should act in order to prevent penetration.* For example if a ball rests on a planar floor, then the set of geometry contacts between the ball and the floor equals the single point where the ball touches the floor.

More precisely, if we define *contact regions* as the set of collision geometry surface patches on the first and second geometry so that points on the first patch coincide with points on the second patch and vice versa, [13], then contacts lie on the contact regions. If the contact region is polygonal (or can be split to polygonal regions) then the contacts correspond to the vertices of the region (the effects of contact forces distributed over the contact region can be replaced by the effects of contact forces exerted at the vertices of the contact region only) — for example if a cube rests on the floor then the set of geometry contacts between the cube and the floor consists of the four vertices of the cube's bottom face. On the other hand, to illustrate the complexity of the problem and justify the reason why we did not present any formal definitions, let us consider a cylinder whose principal axis points upwards and the cylinder is resting on a horizontal desk. In that case the contact region between the cylinder and the floor equals a (portion of a) disc and it might not be trivial to define/compute contacts in this scenario.

Higher-Level and Lower-Level Collision Detection

Collision detection is supposed to determine whether there are any penetrating geometries and, if there are no penetrations, compute all contacts. To reduce the work that needs be done, collision detection is usually performed on two levels. Higher-level collision detection (*broad phase*) keeps track of collision geometry pairs whose geometry hulls (bounding boxes, etc.) overlap while lower-level collision detection (*narrow phase*) processes information from the higher-level collision detection and actually computes contacts and tests for penetration, [13].

Higher-level collision detection serves as a filter that rejects collision geometry pairs that certainly do not contact and prevents such pairs from being processed by the lower-level collision detection, thus decreasing the computational cost. Note that traditional algorithms from computational geometry, such as various space-partitioning trees (oct-trees, k-d trees, BSP trees, etc.), are not suitable for the implementation of higher-level collision detection — such algorithms can be used to query what geometry hulls occupy a specified volume or what geometry hulls overlap a given geometry hull. Instead, we need to determine quickly what geometry hulls overlap each other, which is a different problem, and special algorithms have to be used, such as *sort and sweep*[11], [10], [13], [8], [6], [2].

Lower-level collision detection processes overlapping geometry hulls and invokes geometry type pair specific algorithms to actually perform the collision detection between the two geometries. Collision geometry types handled by the lower-level collision detection should be general enough so that complex shapes can be modelled while at the same time they should be simple so that the determination of contacts and the testing for geometry penetration can be implemented efficiently. Since each geometry type pair requires a special handling (algorithms) the number of supported geometry types should be as small as possible. Obviously, volume-based representations of collision geometries are preferred over boundary representations, triangle meshes & co. because volume-based representations usually allow to test for penetration quickly and/or are able to measure penetration depth, which can be used to stabilize non-penetration constraints or estimate the time of collision. Traditional algorithms from computational geometry can be used to implement lower-level collision detection. Collision detection libraries often support convex polyhedra as the geometry primitives, because contacts can be computed efficiently and spatial and temporal coherence can be utilized, [10], [13], [2], [6].

Collision detection affects both the stability and performance of the whole simulator. Collision detection (and constraint solver) is usually the bottleneck of the simulator and so it must be efficient. At minimum, the used algorithms should utilize the coherence of the collision detection results over successive time steps (results from the previous time step are nearly correct for the current time step). Also, the stability of the simulation highly depends on the precision and correctness of the collision detection results and so it must be reliable.

Contact Representation

If i is the index of the i-th contact then A_i will denote the first body in the contact, B_i the second body, $\vec{p}_i$ the world-space position of the coinciding points $\vec{p}_{A_i}$ and $\vec{p}_{B_i}$ on the first and second body ($\vec{p}_i = \vec{p}_{A_i} = \vec{p}_{B_i}$) and $\vec{n}_i$ the surface normal (with a unit length) at the contact pointing *towards body* B_i. Vector $\dot{\vec{n}}_i$ will denote the time derivative of $\vec{n}_i$ and if $\vec{n}_i$ is attached to the first body, then $\dot{\vec{n}}_i$ can be computed according to (5.5).

If t_{cur} is the current time and $\vec{r}_{A_i} = \vec{p}_i - \vec{x}_{A_i}(t_{cur})$[12] and $\vec{r}_{B_i} = \vec{p}_i - \vec{x}_{B_i}(t_{cur})$ are world-space

[11]Which is implemented in our simulator.

[12]For the purposes of this section $\vec{r}$ will not denote generalized position(s). This is different from previous

vectors attached to the first and second body and $\vec{r}^{\,b}_{A_i}$ and $\vec{r}^{\,b}_{B_i}$ the corresponding body space vectors then, according to (5.1), $\vec{p}_{A_i}(t) = \vec{x}_{A_i}(t) + R_{A_i}(t) \cdot \vec{r}^{\,b}_{A_i}$ and $\vec{p}_{B_i} = \vec{x}_{B_i}(t) + R_{B_i}(t) \cdot \vec{r}^{\,b}_{B_i}$ and $\vec{p}_{A_i}(t)$ and $\vec{p}_{B_i}(t)$ can be treated as functions of time t.

This allows to define the *separation distance* $d_i(t)$ at contact i as a function of time, $d_i(t) = \vec{n}_i(t) \cdot (\vec{p}_{B_i}(t) - \vec{p}_{A_i}(t))$. Separation distance is a measure of the relative displacement of the two coinciding points along $\vec{n}_i(t)$ ("distance" of the two bodies at contact i) so that positive values of $d_i(t)$ indicate separation at i while the negative values indicate penetration at i. Ideally, $d_i(t_{cur}) = 0$ (separation distance is zero when the contact is detected). However since the simulation advances in discrete time steps, $d_i(t)$ might never reach zero and hence we will assume that the bodies are in contact if $|d_i(t_{cur})| < \epsilon$.

Contacts will be represented by the values of A_i, B_i, $\vec{p}_i$, $\vec{n}_i$, $\dot{\vec{n}}_i$, d_i which are supposed to be evaluated by collision detection with respect to the current time t_{cur}.

6.5.2 Non-Penetration

Non-penetration will be enforced as follows. Whenever ODE solver invokes **evaluate_derivative**, collision geometry state is updated (geometries positioned and oriented) according to the system state $\vec{y}(t)$ and collision detection ran to test for penetration. If a penetration is detected, ODE solver is notified that the current state is invalid (non-penetration constraints are violated) so that it could run the bisection algorithm to locate the latest valid state. If there is no penetration, contacts are computed, position-level non-penetration constraints formulated and solved on the velocity or acceleration level. Note that non-penetration is enforced by exerting constraint forces and impulses only, positions and orientations of the affected bodies are *not modified directly.*

Formulation

Each contact i yields a position-level greater-or-equal constraint $C_p^i \geq 0$ affecting bodies A_i and B_i, defined in terms of the separation distance d_i at contact i,

$$C_p^i(t) = d_i(t) = \vec{n}_i(t) \cdot (\vec{p}_{B_i}(t) - \vec{p}_{A_i}(t)) \geq 0. \tag{6.35}$$

Since $|C_p^i(t_{cur})| < \epsilon$ at the current time t_{cur}, the constraint can be implemented either on the velocity or acceleration level according to (6.34).

Computing Jacobians and Equation Right-Hand Side

We will now compute the derivatives required by (6.34),

$$\begin{aligned}
C_p^i &= \vec{n}_i \cdot (\vec{p}_{B_i} - \vec{p}_{A_i}) \\
\dot{C}_p^i &= \dot{\vec{n}}_i \cdot (\vec{p}_{B_i} - \vec{p}_{A_i}) + \vec{n}_i \cdot (\dot{\vec{p}}_{B_i} - \dot{\vec{p}}_{A_i}) \\
\ddot{C}_p^i &= \ddot{\vec{n}}_i \cdot (\vec{p}_{B_i} - \vec{p}_{A_i}) + \dot{\vec{n}}_i \cdot (\dot{\vec{p}}_{B_i} - \dot{\vec{p}}_{A_i}) + \\
&\quad \dot{\vec{n}}_i \cdot (\dot{\vec{p}}_{B_i} - \dot{\vec{p}}_{A_i}) + \vec{n}_i \cdot (\ddot{\vec{p}}_{B_i} - \ddot{\vec{p}}_{A_i}) \\
&= \ddot{\vec{n}}_i \cdot (\vec{p}_{B_i} - \vec{p}_{A_i}) + 2 \cdot \dot{\vec{n}}_i \cdot (\dot{\vec{p}}_{B_i} - \dot{\vec{p}}_{A_i}) + \vec{n}_i \cdot (\ddot{\vec{p}}_{B_i} - \ddot{\vec{p}}_{A_i}).
\end{aligned}$$

We can express $\vec{p}_{A_i}$ and $\vec{p}_{B_i}$ in terms of $\vec{x}_{A_i}, \vec{r}_{A_i}$ and $\vec{x}_{B_i}, \vec{r}_{B_i}$, where $\vec{x}_{A_i}$ and $\vec{x}_{B_i}$ are the centers of masses of the first and second body and $\vec{r}_{A_i}$ and $\vec{r}_{B_i}$ are world space vectors attached

sections and we will have to write $C_p^i(t)$ instead of $C_p^i(\vec{r}(t))$ to prevent the clash when formulating position-level constraints.

to the two bodies, so that

$$\begin{aligned}\vec{p}_{A_i} &= \vec{x}_{A_i} + \vec{r}_{A_i}\\ \vec{p}_{B_i} &= \vec{x}_{B_i} + \vec{r}_{B_i}.\end{aligned}$$

Their time derivatives can be computed according to (5.5) as[13]

$$\begin{aligned}\dot{\vec{p}}_{A_i} &= \dot{\vec{x}}_{A_i} + \vec{\omega}_{A_i} \times \vec{r}_{A_i}\\ \dot{\vec{p}}_{B_i} &= \dot{\vec{x}}_{B_i} + \vec{\omega}_{B_i} \times \vec{r}_{B_i}\\ \ddot{\vec{p}}_{A_i} &= \ddot{\vec{x}}_{A_i} + \vec{\alpha}_{A_i} \times \vec{r}_{A_i} + \vec{\omega}_{A_i} \times (\vec{\omega}_{A_i} \times \vec{r}_{A_i})\\ \ddot{\vec{p}}_{B_i} &= \ddot{\vec{x}}_{B_i} + \vec{\alpha}_{B_i} \times \vec{r}_{B_i} + \vec{\omega}_{B_i} \times (\vec{\omega}_{B_i} \times \vec{r}_{B_i}),\end{aligned}$$

where $\vec{\omega}_{A_i}$ and $\vec{\omega}_{B_i}$ denote the angular velocity of the first and second body and $\vec{\alpha}_{A_i}$ and $\vec{\alpha}_{B_i}$ the angular acceleration.

Assuming that $C_p^i(t_{cur})$ is exactly zero, $\vec{p}_{A_i}(t_{cur}) = \vec{p}_{B_i}(t_{cur}) = p_i(t_{cur})$ we obtain

$$\begin{aligned}\dot{C}_p^i(t_{cur}) &= \vec{n}_i \cdot (\dot{\vec{p}}_{B_i} - \dot{\vec{p}}_{A_i}) = \vec{n}_i \cdot (\dot{\vec{x}}_{B_i} + \vec{\omega}_{B_i} \times \vec{r}_{B_i} - \dot{\vec{x}}_{A_i} - \vec{\omega}_{A_i} \times \vec{r}_{A_i})\\ &= \vec{n}_i \cdot \dot{\vec{x}}_{B_i} + \vec{n}_i \cdot (\vec{\omega}_{B_i} \times \vec{r}_{B_i}) - \vec{n}_i \cdot \dot{\vec{x}}_{A_i} - \vec{n}_i \cdot (\vec{\omega}_{A_i} \times \vec{r}_{A_i})\\ &= \vec{n}_i \cdot \dot{\vec{x}}_{B_i} + (\vec{r}_{B_i} \times \vec{n}_i) \cdot \vec{\omega}_{B_i} - \vec{n}_i \cdot \dot{\vec{x}}_{A_i} - (\vec{r}_{A_i} \times \vec{n}_i) \cdot \vec{\omega}_{A_i}\\ &= J_{i,B_i} \cdot \vec{v}_{B_i} + J_{i,A_i} \cdot \vec{v}_{A_i} = J_i \cdot \vec{v}\\ \ddot{C}_p^i(t_{cur}) &= 2 \cdot \dot{\vec{n}}_i \cdot (\dot{\vec{p}}_{B_i} - \dot{\vec{p}}_{A_i}) + \vec{n}_i \cdot (\ddot{\vec{p}}_{B_i} - \ddot{\vec{p}}_{A_i})\\ &= 2 \cdot \dot{\vec{n}}_i \cdot (\dot{\vec{p}}_{B_i} - \dot{\vec{p}}_{A_i}) +\\ &\quad \vec{n}_i \cdot (\ddot{\vec{x}}_{B_i} + \vec{\alpha}_{B_i} \times \vec{r}_{B_i} + \vec{\omega}_{B_i} \times (\vec{\omega}_{B_i} \times \vec{r}_{B_i})) -\\ &\quad \vec{n}_i \cdot (\ddot{\vec{x}}_{A_i} + \vec{\alpha}_{A_i} \times \vec{r}_{A_i} + \vec{\omega}_{A_i} \times (\vec{\omega}_{A_i} \times \vec{r}_{A_i}))\\ &= \vec{n}_i \cdot \ddot{\vec{x}}_{B_i} + (\vec{r}_{B_i} \times \vec{n}_i) \cdot \vec{\alpha}_{B_i} - \vec{n}_i \cdot \ddot{\vec{x}}_{A_i} - (\vec{r}_{A_i} \times \vec{n}_i) \cdot \vec{\alpha}_{A_i} +\\ &\quad 2 \cdot \dot{\vec{n}}_i \cdot (\dot{\vec{p}}_{B_i} - \dot{\vec{p}}_{A_i}) + \vec{n}_i \cdot (\vec{\omega}_{B_i} \times (\vec{\omega}_{B_i} \times \vec{r}_{B_i})) -\\ &\quad \vec{n}_i \cdot (\vec{\omega}_{A_i} \times (\vec{\omega}_{A_i} \times \vec{r}_{A_i}))\\ &= J_{i,B_i} \cdot \vec{a}_{B_i} + J_{i,A_i} \cdot \vec{a}_{A_i} + \dot{J}_i \cdot \vec{v} = J_i \cdot \vec{a} - c_i,\end{aligned}$$

which can be distilled to the form that can be directly substituted into (6.34) and yield the formulation of the non-penetration constraint due to contact i on the velocity and acceleration level with stabilization,

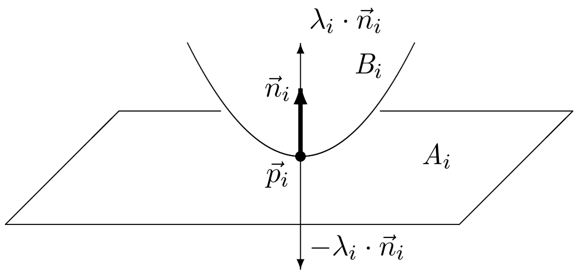
$$\begin{aligned}J_{i,A_i} &= \begin{pmatrix} J_{i,A_i}^{linear} & J_{i,A_i}^{angular}\end{pmatrix} = \begin{pmatrix} -\vec{n}_i^T & -(\vec{r}_{A_i} \times \vec{n}_i)^T\end{pmatrix}\\ J_{i,B_i} &= \begin{pmatrix} J_{i,B_i}^{linear} & J_{i,B_i}^{angular}\end{pmatrix} = \begin{pmatrix} \vec{n}_i^T & (\vec{r}_{B_i} \times \vec{n}_i)^T\end{pmatrix}\\ \dot{J}_i \cdot \vec{v} &= 2 \cdot \dot{\vec{n}}_i \cdot (\dot{\vec{x}}_{B_i} + \vec{\omega}_{B_i} \times \vec{r}_{B_i} - \dot{\vec{x}}_{A_i} - \vec{\omega}_{A_i} \times \vec{r}_{A_i}) +\\ &\quad \vec{n}_i \cdot (\vec{\omega}_{B_i} \times (\vec{\omega}_{B_i} \times \vec{r}_{B_i}) - \vec{\omega}_{A_i} \times (\vec{\omega}_{A_i} \times \vec{r}_{A_i}))\\ C_p^i &= d_i.\end{aligned} \tag{6.36}$$

Constraint Interpretation

C_p^i measures the relative body displacement along the contact normal at the contact, $\dot{C}_p^i$ measures the relative body velocity along the normal at the contact and $\ddot{C}_p^i$ the relative acceleration. The

[13] $\frac{\partial}{\partial t}(\vec{a} \times \vec{b}) = \left(\frac{\partial}{\partial t}\vec{a}\right) \times \vec{b} + \vec{a} \times \left(\frac{\partial}{\partial t}\vec{b}\right)$.

Figure 6.2: Illustration of an acceleration-level non-penetration constraint $\ddot{C}_p^i \geq 0$ for a contact i. Constraint force $-\lambda_i \cdot \vec{n}_i$ acts on A_i at $\vec{p}_i$, $\lambda_i \cdot \vec{n}_i$ acts on B_i at the same point. $\lambda_i \geq 0$ is interpreted as the normal force magnitude at the contact and is complementary to the relative body acceleration $\ddot{C}_p^i \geq 0$ at $\vec{p}_i$ along $\vec{n}_i$ called the normal acceleration.



formulation of the constraint on the velocity or acceleration level says that the relative body velocity or acceleration at the contact along the contact normal should not drop below zero, which is something we would intuitively expect.

If λ_i is the Lagrange multiplier due to the acceleration-level constraint then, following (6.7), $(J_{i,A_i}^{linear})^T \cdot \lambda_i$ and $(J_{i,A_i}^{angular})^T \cdot \lambda_i$ are the constraint force and torque exerted on the first body due to the constraint and $(J_{i,B_i}^{linear})^T \cdot \lambda_i$ and $(J_{i,B_i}^{angular})^T \cdot \lambda_i$ are the constraint force and torque exerted on the second body. By inspecting the Jacobians, we see that the effects of the constraint force correspond to a force $-\vec{n}_i \cdot \lambda_i$ and $\vec{n}_i \cdot \lambda_i$ exerted on the first and second body at $\vec{p}_i$ and hence λ_i can be interpreted as the *normal force magnitude* at contact i. Constraint $\ddot{C}_p^i \geq 0$ is a 1-DOF constraint (6.11) and therefore equation (6.12) holds, $\lambda_i \geq 0$ and λ_i is complementary to $\ddot{C}_p^i = J_i \cdot \vec{a} - c_i$, interpreted as the *normal acceleration* at the contact. See figure (6.2) for the illustration. The same can be said about impulsive forces and torques and normal velocity for the velocity-level constraint $\dot{C}_p^i \geq 0$.

Computational Issues

Although the non-penetration approach we have just presented should theoretically avoid penetration and ensure that bodies do not get stuck, ODE solver might get to a state when it can not advance any further (see notes on page 73). The problems relate to the fact that non-penetration constraints are not permanent and are enforced at contacts only, reported by collision detection if bodies do not penetrate. The ODE solver might fail to advance

- if collision detection is not reliable (buggy) and misses contacts.

  For example if collision detection only reports whether bodies are penetrating or not and never computes any contacts, ODE solver's bisection algorithm will locate the latest valid state but the solver will not be able to advance further from that state.

  If collision detection misses a contact that should have been reported at a previous step, it might be too late to handle non-penetration at the current step.

- if the separation distance at contact i drops below a critical value, but does not exceed a preset limit to signalize penetration, and constraint solver fails to establish $\dot{C}_p^i \geq 0$.

  In that case C_p^i (separation distance) will decrease while taking a step and the penetration depth thus increase with the risk of exceeding the limit. Further penetration will not be avoided at contact i and if the current state is the latest valid state, ODE solver will not be able to advance.

We will call contacts i with penetration depths near the limit the critical contacts. If $\dot{C}_p^i < 0$ after the constraint solver ran and constraint i is critical, bodies involved in the contact must be stopped in order to prevent further penetration.

- if collision detection reports redundant or "bogus" contacts.

  For example when two equal boxes are stacked upon each other so that they are nearly aligned but penetrate a little, collision detection might report contacts involving the top and bottom faces of the two boxes, but also contacts involving the penetrating edges and the top or bottom faces due to various tolerances. As a result, there might be multiple contacts whose constraints "fight" against each other and the constraint solver would fail to compute a viable constraint force that could satisfy all constraints.

  The problem of infeasible or redundant constraints can be handled by not insisting on maintaining all constraints exactly but making the constraints soft instead, see section (6.6).

Since collision detection and simulation are performed in discrete time steps, it might happen that certain collisions remain unspotted. For example a bullet might teleport behind a thin wall without encountering a collision if the bullet moves sufficiently fast and the time step size is big enough. This suggests that there must be an upper bound $\Delta t^{max}(t)$ imposed on the time step size at time t, depending on the system state $\vec{y}(t)$, to ensure that no collision (penetration) is missed if simulation step is taken from time t to $t + \Delta t^{max}(t)$, [13].

In the most simplistic way, $\Delta t^{max}(t)$ can be computed from the upper bounds of body velocity magnitudes $v_i^{max}(t)$. If $\vec{p}_i(t)$ is a point on the surface of a rigid body i, then its velocity is given by (5.4) and $\vec{p}_i(t)$ sweeps a trajectory of an approximate length $\Delta t \cdot \|\dot{\vec{p}}_i(t)\|$ from time t to $t + \Delta t$. If ϵ is the maximum penetration depth we can tolerate, we can conservatively require that $\Delta t^{max}(t) \cdot \|\dot{\vec{p}}_i(t)\| \leq \epsilon$. If r_i is the radius of a sphere that bounds the rigid body's collision geometry, centered at the rigid body's center of mass, then $\|\dot{\vec{p}}_i(t)\| = \|\dot{\vec{x}}_i(t) + \vec{\omega}_i(t) \times (\vec{p}_i(t) - \vec{x}_i(t))\| \leq \|\dot{\vec{x}}_i(t)\| + \|\vec{\omega}_i(t)\| \cdot \|\vec{p}_i(t) - \vec{x}_i(t)\| = \|\dot{\vec{x}}_i(t)\| + \|\vec{\omega}_i(t)\| \cdot r_i = v_i^{max}(t)$ and hence obtain that $\Delta t^{max}(t) \leq \epsilon / v_i^{max}(t)$ for body i. Considering all bodies altogether, we realize that $\Delta t^{max}(t) \leq \epsilon / \max\{v_i^{max}(t)\}$. It is desirable to limit (damp) body velocities so that $\Delta t^{max}(t)$ is kept at a reasonable value. Naturally, $\Delta t^{max}(t)$ computed in this way is often too strict (conservative) and more involved methods exploiting spatial properties of rigid bodies and operating in terms of potentially colliding rigid body pairs should be used in the practice.

6.5.3 Contact Forces

We will augment the formulation of acceleration-level non-penetration constraints from the previous section in order to model friction at contacts. Unlike acceleration-level non-penetration constraints, that only constrain relative body acceleration along the surface normals at contacts, contact constrains will also restrict the acceleration in the directions perpendicular to the surface normals. Note that we will be interested in contacts i where $\dot{C}_p^i = 0$ only. Such contacts are called the *resting contacts*.

Coulomb Friction Law

In the previous section we have interpreted $\ddot{C}_p^i$ and λ_i due to the non-penetration constraint $\ddot{C}_p^i \geq 0$ as the normal acceleration and the normal force magnitude, which we will now denote

a_{n_i} and f_{n_i}, hence $a_{n_i} = \ddot{C}_p^i$ and $f_{n_i} = \lambda_i$, and pointed out that $a_{n_i} \geq 0$, $f_{n_i} \geq 0$ and that a_{n_i} was complementary to f_{n_i} $(a_{n_i} \cdot f_{n_i} = 0)$[14]. *Coulomb friction law* extends these conditions to model friction, [23], [13]. We have to define some additional quantities specific to the contact i and the current time t_{cur} first,

- $\vec{t}_i^x$, the first tangential (friction) direction, a unit vector perpendicular to $\vec{n}_i$,
- $\vec{t}_i^y$, the second tangential (friction) direction, a unit vector perpendicular to $\vec{n}_i$ and $\vec{t}_i^x$,
- $\vec{v}_{n_i} = v_{n_i} \cdot \vec{n}_i = \frac{\partial}{\partial t}(\vec{n}_i \cdot (\vec{p}_{B_i} - \vec{p}_{A_i})) \cdot \vec{n}_i$, the relative body velocity along the surface normal,
- $\vec{v}_{t_i^x} = v_{t_i^x} \cdot \vec{t}_i^x = \frac{\partial}{\partial t}(\vec{t}_i^x \cdot (\vec{p}_{B_i} - \vec{p}_{A_i})) \cdot \vec{t}_i^x$, the relative body velocity along the first tangential direction,
- $\vec{v}_{t_i^y} = v_{t_i^y} \cdot \vec{t}_i^y = \frac{\partial}{\partial t}(\vec{t}_i^y \cdot (\vec{p}_{B_i} - \vec{p}_{A_i})) \cdot \vec{t}_i^y$, the relative body velocity along the second tangential direction,
- $\vec{v}_{t_i} = \vec{v}_{t_i^x} + \vec{v}_{t_i^y}$, the relative tangential velocity
- $\vec{a}_{n_i} = a_{n_i} \cdot \vec{n}_i = \frac{\partial^2}{\partial t^2}(\vec{n}_i \cdot (\vec{p}_{B_i} - \vec{p}_{A_i})) \cdot \vec{n}_i$, the relative body acceleration along the surface normal,
- $\vec{a}_{t_i^x} = a_{t_i^x} \cdot \vec{t}_i^x = \frac{\partial^2}{\partial t^2}(\vec{t}_i^x \cdot (\vec{p}_{B_i} - \vec{p}_{A_i})) \cdot \vec{t}_i^x$, the relative body acceleration along the first tangential direction,
- $\vec{a}_{t_i^y} = a_{t_i^y} \cdot \vec{t}_i^y = \frac{\partial^2}{\partial t^2}(\vec{t}_i^y \cdot (\vec{p}_{B_i} - \vec{p}_{A_i})) \cdot \vec{t}_i^y$, the relative body acceleration along the second tangential direction,
- $\vec{a}_{t_i} = \vec{a}_{t_i^x} + \vec{a}_{t_i^y}$, the relative tangential acceleration,
- $\vec{f}_{n_i} = f_{n_i} \cdot \vec{n}_i$, the normal force,
- $\vec{f}_{t_i^x} = f_{t_i^x} \cdot \vec{t}_i^x$, the tangential force along the first tangential direction,
- $\vec{f}_{t_i^y} = f_{t_i^y} \cdot \vec{t}_i^y$, the tangential force along the second tangential direction,
- $\vec{f}_{t_i} = \vec{f}_{t_i^x} + \vec{f}_{t_i^y}$, the tangential (friction) force, perpendicular to $\vec{n}_i$,
- $\vec{f}_{c_i} = \vec{f}_{n_i} + \vec{f}_{t_i}$, the contact force to be exerted on B_i at $\vec{p}_i$, $-\vec{f}_{c_i}$ exerted on A_i at $\vec{p}_i$,
- μ_i, the friction coefficient, a positive scalar value.

Coulomb friction law postulates that if i is a resting contact $(v_{n_i} = 0)$ then

$$f_{n_i} \geq 0 \qquad a_{n_i} \geq 0 \qquad f_{n_i} \cdot a_{n_i} = 0 \tag{6.37}$$

and

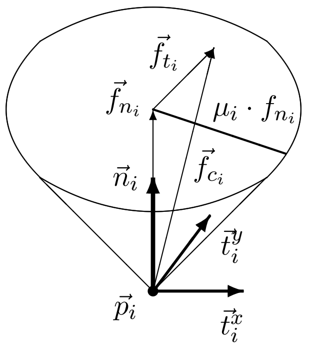
- if $\vec{v}_{t_i} \neq \vec{0}$ then the *dynamic (sliding) friction* is in effect at contact i and[15]

$$\vec{f}_{t_i} = -\mu_i \cdot f_{n_i} \cdot \vec{v}_{t_i} / \|\vec{v}_{t_i}\|. \tag{6.38}$$

[14]These conditions are the well-known conditions (6.12), where $m_i = 1$, $A_i \cdot \vec{\lambda} + b_i = a_{n_i}$.

[15]Note that $\|\vec{f}_{t_i}\| = \mu_i \cdot f_{n_i}$ for a contact i with dynamic friction.

Figure 6.3: Illustration of Coulomb friction for a contact i. Magnitude of the tangential force $\vec{f_{t_i}}$ is restricted so that $\|\vec{f_{t_i}}\| \leq \mu_i \cdot f_{n_i}$ and hence the tip of the contact force $\vec{f_{c_i}} = \vec{f_{n_i}} + \vec{f_{t_i}}$ exerted on B_i at $\vec{p_i}$ is restricted to lie inside or on the boundary of the friction cone. Force $-\vec{f_{c_i}}$ exerted on A_i at $\vec{p_i}$ and bodies A_i and B_i are not drawn. If $\mu_i = 0$ then $\vec{f_{t_i}} = \vec{0}$ and the problem reduces to non-penetration illustrated at figure (6.2).



- if $\vec{v_{t_i}} = \vec{0}$ and $\vec{a_{t_i}} = \vec{0}$ then the *stable static (dry, rolling, sticking) friction* is in effect at contact i and
$$\|\vec{f_{t_i}}\| \leq \mu_i \cdot f_{n_i}. \tag{6.39}$$
- if $\vec{v_{t_i}} = \vec{0}$ and $\vec{a_{t_i}} \neq \vec{0}$ then the *instable static (dry, rolling, sticking) friction* is in effect at contact i and
$$\|\vec{f_{t_i}}\| = \mu_i \cdot f_{n_i} \qquad \vec{f_{t_i}} \cdot \vec{a_{t_i}} \leq 0. \tag{6.40}$$

We would like to express the conditions stipulated by the Coulomb law in the form of acceleration-level constraints (6.17). Unfortunately this is not possible because $\|\vec{f_{t_i}}\|$ is a non-linear function and so Coulomb law has to be approximated.

Contact force $\vec{f_{c_i}}$ can be imagined as if its end-tip was restricted to lie inside an infinite cone centered at the position of the contact, the axis was pointing in the direction of the surface normal and the "width" was determined by μ_i. Coulomb law says that the contact force tip should lie inside or on the boundary of the cone in the case of stable static friction, that the tip should lie on the boundary of the cone and the direction of the contact force should oppose the direction of the tangential acceleration in the case of instable static friction, and that the tip should lie on the boundary of the cone and the contact force should act in the direction opposite to the direction of the tangential velocity in the case of dynamic friction. By setting μ_i to zero, the Coulomb friction law reduces to the known acceleration-level non-penetration constraint $\ddot{C}_p^i \geq 0$. See figure (6.3) for illustration.

Note that stable and instable static friction must be handled altogether because $\vec{f_{t_i}}$ affects $\vec{a_{t_i}}$. Therefore, given a contact i, we can tell whether there is a dynamic or static friction in effect by inspecting $\vec{v_{t_i}} = 0$, but if it is static we can not tell if it is actually a stable or instable static friction until all constraints are solved. Depending on whether static or dynamic friction is in effect at contact i, different constraints will be effective due to i.

We will now present how dynamic friction and static friction can be approximated. The approximation of the static friction we are going to present will replace the friction cone by a four-sided friction pyramid whose sides are parallel to the tangential directions $\vec{t_i^x}$ and $\vec{t_i^y}$ and friction along $\vec{t_i^x}$ will be handled independently on friction along $\vec{t_i^y}$.

We will assume that the contact representation is augmented to hold μ_i and directions $\vec{t_i^x}$ and $\vec{t_i^y}$, henceforward called the friction approximation directions.

Dynamic Friction

The handling of dynamic friction according to the Coulomb law is hard, to say the least, [6]. It is well known that there exist contact configurations where the Coulomb dynamic friction model does not allow any valid solution (inconsistency) or allows multiple solutions with different effects on the simulation in terms of the accelerational response (indeterminacy). It can be shown that computing contact forces (if they exist), when dynamic friction is in effect, is NP-hard as shown in [6], [5].

Suppose that all constraints are due to contacts i with dynamic friction and $\mu_i > 0$. Then $\vec{f}_{c_i} = \vec{f}_{n_i} + \vec{f}_{t_i} = f_{n_i} \cdot \vec{n}_i - \mu_i \cdot f_{n_i} \cdot \vec{v}_{t_i}/\|\vec{v}_{t_i}\| = f_{n_i} \cdot (\vec{n}_i - \mu_i \cdot \vec{v}_{t_i}/\|\vec{v}_{t_i}\|) = f_{n_i} \cdot \vec{u}_i$ and $\vec{u}_i \neq \vec{n}_i$ is the direction of the contact (constraint) force acting at $\vec{p}_i$. We could formulate the constraint due to contact i in the form of non-penetration constraint $\ddot{C}_p^i \geq 0$, but this time the constraint force due to the constraint would have to act in the direction of $\vec{u}_i$, not $\vec{n}_i$ (the relative acceleration would be measured along $\vec{n}_i$ but the constraint force would act in the direction of $\vec{u}_i$). More formally, let us define $J_i(\vec{d})$ as a $1 \times n$ block matrix, where n is the number of bodies, and the non-zero blocks are given by

$$\begin{aligned} J_{i,A_i}(\vec{d}) &= \left(-\vec{d}^T \quad -(\vec{r}_{A_i} \times \vec{d})^T \right) \\ J_{i,B_i}(\vec{d}) &= \left(\vec{d}^T \quad (\vec{r}_{B_i} \times \vec{d})^T \right). \end{aligned} \tag{6.41}$$

Then, using the generalized notation, the non-penetration constraint with dynamic friction at contact i could still be formulated as $J_i(\vec{n}_i) \cdot \vec{a} \geq c_i$ but, this time, the constraint force $\vec{F}_c^i$ due to the constraint would have to equal $\vec{F}_c^i = J_i(\vec{u}_i)^T \cdot \lambda_i$, $\vec{F}_c^i \neq J_i(\vec{n}_i)^T \cdot \lambda_i$ (compare with (6.11) and (6.12)). Let J^n denote a $n \times n$ block matrix whose block rows are given by $J_i(\vec{n}_i)$ and J^u denote a $n \times n$ block matrix with block rows $J_i(\vec{u}_i)$. Then, following the derivation of acceleration-level inequality constraints, we could again reduce the problem to a LCP, but matrix A would no longer be symmetric. In fact, $A = J^n \cdot M^{-1} \cdot (J^u)^T$ and specialized inefficient LCP solvers would have to be used to compute $\vec{\lambda}$. The trouble is that there might not be any valid solution to the LCP and a polynomial time algorithm capable of solving this LCP can not exist unless $P = NP$, [6].

To overcome these problems, dynamic friction is often approximated either by external forces (non-penetration constraints are imposed only) or by impulsive friction on the velocity level, [16]. We will now present the approximation by external forces, called the *Lötstedt's approximation*, [6].

The idea is very simple — instead of imposing friction constraints at the current time step, acceleration-level non-penetration constraint $\ddot{C}_p^i \geq 0$ is imposed only, $\vec{f}_{t_i} = -\mu_i \cdot f_{n_i} \cdot \vec{v}_{t_i}/\|\vec{v}_{t_i}\|$ is approximated using the value of f_{n_i} from the previous time step and $\vec{f}_{t_i}$ exerted as an external force. To simplify the matter and avoid the need for tracking contact correspondence between time steps, we can instead let constraint solver compute $f_{n_i} = \lambda_i$ at the current time step (enforce non-penetration without friction) and compute what $\vec{f}_{t_i}$ should be at the next time step, using the contact configuration and f_{n_i} guess from the current time step, and apply $\vec{f}_{t_i}$ "in advance" (so that it would be automatically added to the force accumulators at the next time step).

Static Friction

Let us assume that we are going to model static friction at a contact i ($\vec{v}_{t_i} = \vec{0}$) and, for illustration, the friction force is restricted to act along the direction of $\vec{t}_i^x$ only[16]. We thus have

[16]This corresponds to the static friction model in $\mathbf{R}^2$.

only a single friction direction $\vec{t}_i^x$, $\vec{f}_{t_i} = \vec{f}_{t_i^x} = f_{t_i^x} \cdot \vec{t}_i^x$, $\|\vec{f}_{t_i}\| = |f_{t_i^x}|$, $\vec{a}_{t_i} = \vec{a}_{t_i^x} = a_{t_i^x} \cdot \vec{t}_i^x$, $\|\vec{a}_{t_i}\| = |a_{t_i^x}|$, $\vec{a}_{t_i} \cdot \vec{f}_{t_i} = f_{t_i^x} \cdot a_{t_i^x}$ and the conditions due to the (stable and instable) static friction can be written as

$$\begin{aligned} -\mu_i \cdot f_{n_i} \leq \quad & f_{t_i^x} \quad \leq \mu_i \cdot f_{n_i} \\ f_{t_i^x} = -\mu_i \cdot f_{n_i} \quad & \Rightarrow \quad a_{t_i^x} \geq 0 \\ f_{t_i^x} = \mu_i \cdot f_{n_i} \quad & \Rightarrow \quad a_{t_i^x} \leq 0 \\ -\mu_i \cdot f_{n_i} < f_{t_i^x} < \mu_i \cdot f_{n_i} \quad & \Rightarrow \quad a_{t_i^x} = 0. \end{aligned} \tag{6.42}$$

We already know that $a_{n_i} = \frac{\partial^2}{\partial t^2}(\vec{n}_i \cdot (\vec{p}_{B_i} - \vec{p}_{A_i}))$ and f_{n_i} is the Lagrange multiplier due to the non-penetration constraint $a_{n_i} \geq 0$. Since $a_{t_i^x} = \frac{\partial^2}{\partial t^2}(\vec{t}_i^x \cdot (\vec{p}_{B_i} - \vec{p}_{A_i}))$ we would like to interpret $f_{t_i^x}$ as a multiplier due to a sort of $a_{t_i^x} = 0$ constraint with appropriate constraint force limits. By comparing (6.42) with equation (6.15), we see that the static friction conditions actually impose an 1-DOF acceleration-level bounded-equality constraint $a_{t_i^x} = 0$ with *non-constant* constraint force limits given by $\mp\mu_i \cdot f_{n_i}$. The conditions thus say that

- points $\vec{p}_{A_i}$ and $\vec{p}_{B_i}$ in the contact should not accelerate with respect to each other in the direction of $\vec{t}_i^x$, that is $a_{t_i^x} = 0$, but
- the constraint might break, if the magnitude of the tangential force required to enforce $a_{t_i^x} = 0$ had to exceed $\mu_i \cdot f_{n_i}$.

Note that we are able to handle bounded-equality constraints with *constant* constraint force limits only. However in the static friction case, the limits are a function of f_{n_i}, the (unknown) magnitudes of the non-penetration forces. To handle static friction constraints in terms of bounded-equality constraints (6.17), we have to guess f_{n_i} so that the friction force limits could be estimated and made constant. In the next section we will show how f_{n_i} can be guessed, apart from taking the values of f_{n_i} from the previous time step. For now, we will assume that we are already provided with the guesses of f_{n_i} which we will denote $\hat{f}_{n_i}$.

To clarify, if i is the index of a contact with static friction acting along $\vec{t}_i^x$ only, $c_i(\vec{d}) = \dot{J}_i(\vec{d}) \cdot \vec{v}$, where $J_i(\vec{d})$ is given by (6.41), and proceeding similarly to the derivation of (6.36), then the contact constraints can be formulated as the following two 1-DOF constraints,

- $a_{n_i} = \frac{\partial^2}{\partial t^2}(\vec{n}_i \cdot (\vec{p}_{B_i} - \vec{p}_{A_i})) = J_{i,A_i}(\vec{n}_i) \cdot \vec{a}_{A_i} + J_{i,B_i}(\vec{n}_i) \cdot \vec{a}_{B_i} - c_i(\vec{n}_i) \geq 0$, where $\vec{f}_{n_i} = f_{n_i} \cdot \vec{n}_i$ is the corresponding constraint force, called the normal force, exerted positively on B_i and negatively on A_i at $\vec{p}_i$ and f_{n_i} is the multiplier due to the normal constraint[17],
- $a_{t_i^x} = \frac{\partial^2}{\partial t^2}(\vec{t}_i^x \cdot (\vec{p}_{B_i} - \vec{p}_{A_i})) = J_{i,A_i}(\vec{t}_i^x) \cdot \vec{a}_{A_i} + J_{i,B_i}(\vec{t}_i^x) \cdot \vec{a}_{B_i} - c_i(\vec{t}_i^x) = 0$, where $\vec{f}_{t_i^x} = f_{t_i^x} \cdot \vec{t}_i^x$ is the corresponding constraint force, called the friction force, exerted positively on B_i and negatively on A_i at $\vec{p}_i$, scalar $f_{t_i^x}$, $-\mu_i \cdot \hat{f}_{n_i} \leq f_{t_i^x} \leq \mu_i \cdot \hat{f}_{n_i}$, is the multiplier due to the friction constraint[18] and $\hat{f}_{n_i}$ is the (constant) guess of f_{n_i}.

To model friction along the tangential plane we incorporate an additional 1-DOF constraint that restricts relative body acceleration along the second friction direction $\vec{t}_i^y$ at $\vec{p}_i$, modelled exactly in the same way we modelled friction along the first friction direction $\vec{t}_i^x$, [22], [6]. Static friction is thus approximated in the directions of $\vec{t}_i^x$ and $\vec{t}_i^y$ and each direction is handled separately.

[17] $J_i(\vec{n}_i)^T \cdot f_{n_i}$ is the generalized constraint force exerted on the system due to the normal force constraint.

[18] $J_i(\vec{t}_i^x)^T \cdot f_{t_i^x}$ is the generalized constraint force exerted on the system due to the friction force constraint.

The following equation defines a 3-DOF bounded-equality constraint (6.17) due to contact i with static friction and illustrates the technique

$$J_{i,A_i} = \begin{pmatrix} -\vec{n}_i^T & -(\vec{r}_{A_i} \times \vec{n}_i)^T \\ -\vec{t}_i^{x\,T} & -(\vec{r}_{A_i} \times \vec{t}_i^x)^T \\ -\vec{t}_i^{y\,T} & -(\vec{r}_{A_i} \times \vec{t}_i^y)^T \end{pmatrix} \qquad J_{i,B_i} = \begin{pmatrix} \vec{n}_i^T & (\vec{r}_{B_i} \times \vec{n}_i)^T \\ \vec{t}_i^{x\,T} & (\vec{r}_{B_i} \times \vec{t}_i^x)^T \\ \vec{t}_i^{y\,T} & (\vec{r}_{B_i} \times \vec{t}_i^y)^T \end{pmatrix}$$

$$\begin{aligned} \vec{c}_i &= \dot{J}_{i,A_i} \cdot \vec{v}_{A_i} + \dot{J}_{i,B_i} \cdot \vec{v}_{B_i} \\ \vec{\lambda}_i^{lo} &= (0, -\mu_i \cdot \hat{f}_{n_i}, -\mu_i \cdot \hat{f}_{n_i}) \\ \vec{\lambda}_i^{hi} &= (+\infty, \mu_i \cdot \hat{f}_{n_i}, \mu_i \cdot \hat{f}_{n_i}), \end{aligned} \tag{6.43}$$

where $\hat{f}_{n_i}$ is a guess of $f_{n_i} = (\vec{\lambda}_i)_1$, $\vec{c}_i$ can be computed like in (6.41), the first constraint DOF is due to the non-penetration and the other two DOFs are due to the static friction along $\vec{t}_i^x$ and $\vec{t}_i^y$.

Handling Static Friction Constraints

We will treat constraint DOFs due to static friction as *special*, the other DOFs will be non-special, and solve for $\vec{\lambda}$ due to *all non-special* DOFs first, ignoring the special DOFs. That way we obtain the estimates of f_{n_i} as the magnitudes of the normal forces valid with respect to the case when there were no friction, which are in turn used to set and *fix* static friction force limits. Finally, all non-special and special DOFs (with fixed limits) are handled and the global $\vec{\lambda}$ obtained, [22].

Luckily, this can be incorporated directly into the Dantzig LCP solver we outlined in section (6.2.5) — all constraints will be conceptually broken down to 1-DOF constraints and ordered so that the non-special DOFs would come before the special DOFs. If there are p non-special DOFs and q special DOFs, we will take the first p iterations to compute $\vec{\lambda}$ valid with respect to the non-special DOFs only, get the estimates of f_{n_i}, fix the limits of the special DOFs and take the last q iterations to compute $\vec{\lambda}$ valid with respect to both the non-special and special DOFs (with fixed limits).

6.5.4 Contact Impulses

The purpose of contact impulses is to model collisions with impulsive friction defined in [16]. The most of the material described in this section can be treated as a velocity-level formulation of the material from the previous section and so we will proceed quickly.

Let us assume that we have a position-level non-penetration constraint $C_p^i \geq 0$ at contact i defined as in (6.35). If $C_p^i = 0$ and $\dot{C}_p^i < 0$, the bodies A_i and B_i hit each other and a collision (impact) occurs. Non-penetration constraint at i merely requires that $\dot{C}_p^i \geq 0$ so that penetration would be avoided. What we would like to do is to define a different velocity-level constraint that would make the bodies bounce off, eventually accounting for friction that would act over the period of the collision in the real-world physics. For the purposes of further discussion we will assume that $\dot{C}_p^i < 0$.

Frictionless Collisions

To model collisions without friction we use a simple empirical law, called the Newton Collision Law, which says that the relative body velocity v_{n_i} at contact i along the surface normal $\vec{n}_i$ should be opposed and its magnitude scaled by $0 \leq \epsilon_i \leq 1$ when the collision occurs, that is,

$$v_{n_i}^+ = -\epsilon_i \cdot v_{n_i}^-,$$

where $v_{n_i}^+$ is the value of v_{n_i} (relative body velocity along $\vec{n}_i$ at $\vec{p}_i$) after the collision is resolved, $v_{n_i}^-$ is the value before the collision is resolved and ϵ_i is called the *coefficient of restitution.* Setting $\epsilon_i = 1$ would model a perfectly elastic collision while $\epsilon_i = 0$ would make the bodies not to bounce at all, [13].

Since $\dot{C}_p^i$ is the relative body velocity at contact i, $\dot{C}_p^i = v_{n_i}$, the collision law can be implemented by the velocity-level constraint

$$\dot{C}_p^i = v_{n_i} = J_{i,A_i}(\vec{n}_i) \cdot \vec{v}_{A_i} + J_{i,B_i}(\vec{n}_i) \cdot \vec{v}_{B_i} \geq -\epsilon_i \cdot v_{n_i}^-. \tag{6.44}$$

Note that we have to use a greater-or-equal formulation of the constraint, because there might be other impulses, say due to other collisions, acting on B_i that should not be cancelled.

Collisions With Impulsive Friction

To model collisions with friction we would like to (in addition to the collision constraint (6.44)) constrain $v_{t_i^x}$ to zero but allow the constraint to break if the magnitude of the impulsive constraint force due to the constraint had to exceed $\mu_i \cdot f_{n_i}$, where f_{n_i} was the magnitude of the impulsive normal force due to the collision constraint, and do the same for $v_{t_i^y}$.

If $f_\bullet$ and $\vec{f}_\bullet$ denote the impulsive quantities analogous to the force quantities $f_\bullet$ and $\vec{f}_\bullet$ from the previous section, then the velocity-level constraints we would like to model are given by

$$\begin{array}{lll} f_{n_i} \geq 0 & v_{n_i} \geq -\epsilon_i \cdot v_{n_i}^- & f_{n_i} \cdot v_{n_i} = 0 \\ \vec{v}_{t_i} = \vec{0} & \Rightarrow \quad \|\vec{f}_{t_i}\| \leq \mu_i \cdot f_{n_i} & \\ \vec{v}_{t_i} \neq \vec{0} & \Rightarrow \quad \|\vec{f}_{t_i}\| = \mu_i \cdot f_{n_i} \ \wedge \ \vec{f}_{t_i} \cdot \vec{v}_{t_i} \leq 0. & \end{array} \tag{6.45}$$

We would like to approximate these equations analogously to the static friction case so that the static friction's algorithm could be reused (this time we would relate constraint impulses to velocities instead of constraint forces to accelerations, though).

By following the derivation of (6.43) we get the formulation of the constraints we are looking for, in the form of a 3-DOF velocity-level bounded-equality constraint (6.26),

$$\begin{array}{rcl} J_{i,A_i} = \begin{pmatrix} -\vec{n}_i^T & -(\vec{r}_{A_i} \times \vec{n}_i)^T \\ -\vec{t}_i^{x\,T} & -(\vec{r}_{A_i} \times \vec{t}_i^x)^T \\ -\vec{t}_i^{y\,T} & -(\vec{r}_{A_i} \times \vec{t}_i^y)^T \end{pmatrix} & & J_{i,B_i} = \begin{pmatrix} \vec{n}_i^T & (\vec{r}_{B_i} \times \vec{n}_i)^T \\ \vec{t}_i^{x\,T} & (\vec{r}_{B_i} \times \vec{t}_i^x)^T \\ \vec{t}_i^{y\,T} & (\vec{r}_{B_i} \times \vec{t}_i^y)^T \end{pmatrix} \\ \vec{c}_i & = & (-\epsilon_i \cdot v_{n_i}^-, 0, 0) \\ \vec{\lambda}_i^{lo} & = & (0, -\mu_i \cdot \hat{f}_{n_i}, -\mu_i \cdot \hat{f}_{n_i}) \\ \vec{\lambda}_i^{hi} & = & (+\infty, \mu_i \cdot \hat{f}_{n_i}, \mu_i \cdot \hat{f}_{n_i}), \end{array} \tag{6.46}$$

where $\hat{f}_{n_i}$ is a guess of $f_{n_i} = (\vec{\lambda}_i)_1$, the first constraint DOF is due to the collision (impact) and the other two DOFs are due to the impulsive friction along $\vec{t}_i^x$ and $\vec{t}_i^y$ directions and are *special.*

Impulsive friction with other velocity-level constraints is handled analogously to the acceleration-level case of static friction with other acceleration-level constraints. All constraints are again reduced to a LCP, the constraint rows due to the non-special DOFs are solved first, the values of $\hat{f}_{n_i}$ are obtained and impulsive friction limits are set. Finally, all constraint rows due to the non-special and special DOFs are solved.

6.6 Soft Constraints

Constraints that we handled so far were all considered "hard" and it was not desirable to let them break. Constraint stabilization was incorporated in order to ensure that constraint error, introduced into the simulation mostly as a result of integration error, could be corrected. On the other side, one might want to implement "soft" constraints designed not to be maintained exactly. For example articulated figure's joints can be "programmed" to follow desired angles (kinematic animation), but it might be desirable to violate the joint angle constraints when the figure's arm hits an obstacle in the environment or the arm is being pulled. The other reason for incorporating soft constraints is the need for handling singular systems, where hard constraints might be infeasible or redundant. We will now describe how soft constraints, defined according to [22], can be handled by the simulator. The problem will be discussed for the case of acceleration-level equality constraints, but the exactly same approach can be used to implement soft inequality or bounded-equality constraints on the acceleration or velocity levels.

Let us assume that we have an m_i-DOF acceleration-level equality constraint i defined as in equation (6.4), $J_i \cdot \vec{a} = \vec{c}_i$. Equation $(J_i \cdot \vec{a} - \vec{c}_i)_j$ measures the current constraint error at the j-th constraint DOF ($1 \leq j \leq m_i$) and if the constraint is meant to be hard, $(J_i \cdot \vec{a} - \vec{c}_i)_j$ must be kept at zero, which is expressed by conditions (6.10) on $\vec{\lambda}_i$ multipliers due to the constraint. The corresponding soft constraint will be defined and implemented as follows.

Each constraint DOF j will have associated a "constraint softness" constant, a positive scalar value, that will indicate how much soft the given DOF is (zero softness will indicate that the constraint DOF is absolutely hard). Constraint softness constants will be concatenated to m_i-vector $\vec{s}_i$ so that $(\vec{s}_i)_j$ will define the softness of the j-th DOF. The value of $(\vec{s}_i)_j$ will *instruct the constraint solver to gain an error* at the j-th DOF that equals exactly $(\vec{s}_i)_j \cdot (\vec{\lambda}_i^*)_j$, where $\vec{\lambda}_i^*$ is the vector of Lagrange multipliers due to the original $J_i \cdot \vec{a} = \vec{c}_i$ constraint. Therefore, it is requested that $(J_i \cdot \vec{a} - \vec{c}_i)_j = (\vec{s}_i)_j \cdot (\vec{\lambda}_i^*)_j$, the original hard constraint is replaced by another hard constraint $J_i \cdot \vec{a} = \vec{c}_i + \vec{s}_i \cdot E \cdot \vec{\lambda}_i^*$ and so the original constraint is enforced with an error $\vec{s}_i \cdot E \cdot \vec{\lambda}_i^*$ ("proportional to the softness constant times the restoring force magnitude due to the original constraint", if m_i was 1).

In the practice, however, the soft constraint due to (6.4) with softness vector $\vec{s}_i$ is implemented by the following hard constraint

$$J_{i,A_i} \cdot \vec{a}_{A_i} + J_{i,B_i} \cdot \vec{a}_{B_i} = \vec{c}_i + \vec{s}_i \cdot E \cdot \vec{\lambda}_i, \tag{6.47}$$

where $\vec{\lambda}_i$ is the vector of Lagrange multipliers due to this constraint. Vector $\vec{\lambda}_i^*$ is thus approximated by $\vec{\lambda}_i$.

We are now interested in how softness constants affect the matrix A and vector $\vec{b}$ from (6.8) so that we could compute $\vec{\lambda}$ and obtain the constraint force to be exerted on the system. Further on, we will use the notation from section (6.1.2).

Formulation

If we have c equality constraints with softness constants $\vec{s}_i$, $1 \leq i \leq c$, concatenated to a block vector $\vec{s}$, $\vec{s} = (\vec{s}_1, \dots, \vec{s}_c)$, $m = \sum_{i=1}^{c} m_i$ is the total number of DOFs removed by the constraints, S is an $m \times m$ matrix with the values of the vector $\vec{s}$ components on the diagonal and zeroes elsewhere, $\vec{c} = (\vec{c}_1, \dots, \vec{c}_c)$ and $\vec{\lambda} = (\vec{\lambda}_1, \dots, \vec{\lambda}_c)$, then we can express the constraints as $J \cdot \vec{a} = \vec{c} + \vec{s} \cdot E \cdot \vec{\lambda}$. Following the derivation scheme of (6.8) we get

$$A = J \cdot M^{-1} \cdot J^T + S \qquad \vec{b} = J \cdot M^{-1} \cdot \vec{F}_{total} - \vec{c}, \tag{6.48}$$

which should be considered as an equivalent of (6.8) — the only difference is that $\vec{s}$ is added to the diagonal of the original matrix A. Obviously the same derivation procedure could be repeated for bounded-equality acceleration-level or velocity-level constraints and hence all we know about hard constraints transfers to the area of soft constraints. To implement soft constraints, we only have to add $\vec{s}$ to the diagonal of the original matrix A.

Therefore, if we have a hard m_i-DOF constraint i formulated as (6.4), (6.11), (6.13), (6.16) or (6.26) and $\vec{s}_i$ is a vector of constraint DOF softness constants, then the corresponding soft constraint is defined and implemented by the same hard constraint whose right-hand side vector $\vec{c}_i$ or $\vec{k}_i$ is extended to include the $\vec{s}_i \cdot E \cdot \vec{\lambda}_i$ term, which has the effect of adding $\vec{s}_i$ to the diagonal of $A_{i,i}$ due to the original hard constraint. Hard constraints i can be seen as soft constraints i with $\vec{s}_i = \vec{0}$.

By rearranging (6.48), soft constraints can also be formulated in terms of an always sparse matrix H from (6.9), in the form of

$$H = \begin{pmatrix} M & -J^T \\ -J & S \end{pmatrix} \qquad H \cdot \begin{pmatrix} \vec{y} \\ \vec{\lambda} \end{pmatrix} + \begin{pmatrix} \vec{0} \\ -\vec{b} \end{pmatrix} = \vec{0}, \tag{6.49}$$

which should be considered as an equivalent of (6.9).

Discussion

For an illustration let us assume that we have a position-level constraint $\vec{C}_p = \vec{0}$ that constrains points $\vec{p}_{A_i}$ and $\vec{p}_{B_i}$ on body A_i and B_i to occupy the same world-space position ($\vec{C}_p = \vec{p}_{A_i} - \vec{p}_{B_i} = \vec{0}$) that we are going to implement on the velocity level. Velocity-level constraint softness constants $\vec{s}_i = (s_i, s_i, s_i)$ together with the Baumgarte stabilization constant α_i can be used to implement springy behavior. The softness constants determine how much the velocity-level constraint can be violated (which results in the gain of position-level error), while the stabilization constant determines how strongly will position-level error be reduced, α_i can be interpreted as a first-order spring gain constant and $1/s_i$ as a spring damping constant, [22]. It is advantageous to treat the stabilization and softness constants as proportional to the frame rate.

Note that soft constraints should not be confused with penalty forces. Penalty forces, presented in section (3.4), use the techniques of local optimization to attract system state towards a state with less energy (error), the computed forces compete with other external forces and their effects are hard to predict. Contrary, soft constraints are implemented by "modified" hard constraints, the results are predictable and guaranteed. Let us stress that the problem of handling soft constraints defined in this way is not an optimization problem.

It follows from (6.48) that soft constraints can stabilize the simulation when the original matrix A is nearly singular (or singular when the original constraints are infeasible or redundant), because non-zero elements are added to the diagonal. This can help to factorize A (or get rid of the singularity). Therefore, as a rule, *no constraints should be absolutely hard, if only to improve the simulation stability and avoid infeasible constraints.* This is important to keep in mind when imposing non-penetration and friction constraints due to contacts, because collision detection might report redundant or "bogus" contacts.

Chapter 7

Simulator

This chapter is devoted to a brief description of concept abstractions in the simulator and an overview of the simulator implementation — the goal is to provide the reader with a global view of the simulator but not to pursue the implementation details. The details are nicely documented in the Reference Guide, that comes with this thesis. We will also attempt to avoid the repetition of various topics already discussed in the previous chapters where appropriate (such as notes on computational/numerical issues), because the discussions can be found there, as well as in the Reference Guide.

The simulator implementation is more or less based on the notes from the previous chapters, but as one could expect, many concepts have to be implemented in a more sophisticated way and smart algorithms have to be used in order to achieve an interactive performance (yet the notes should be sufficient to implement a functional simulator). Although these aspects can be characterized as "implementation details", insignificant from the point of the rigid body dynamics theory, this is what makes the implementation quite involved and distinguishes a practical simulator from a toy. Certain implementation tricks were adopted from [22].

For example, conditions on $\vec{\lambda}$ multipliers due to constraints were formulated in terms of dense matrix A from equation (6.8), but equation (6.9) needs be exploited in the practice. As another example, all bodies were simulated in terms of a single equation of motion, but it might be desirable to partition bodies to distinct simulation groups so that each group could be handled separately (so that when a particular group was being advanced the other groups could remain idle and would not have to be updated). For example, its ODE might be integrated with a different time step, using a different integration method or precision. If all bodies in the group are "at rest", the simulation group could be skipped while advancing the simulation state, which could significantly decrease the processing time when handling complex scenes, etc.

We will now present the main simulator components, outline the way the components collaborate and describe the improvements over the notes from previous chapters, noting that details can be found in the Reference Guide.

7.1 Overview

The rigid body simulation is represented by **Simulator** class that maintains a list of registered bodies, their separation to *disjoint simulation groups* simulated in terms of their own equations of motion and a list of force objects representing explicit interactions among bodies. The idea is to keep bodies that might influence each other in a common simulation group while bodies that can not influence each other in distinct groups so that the groups could be simulated independently on each other.

As a matter of fact, simulation groups consisting of idle bodies need not be simulated at all, and most of the simulation work can be performed locally within a simulation group, exploiting spatial coherence, and thus decreasing the computational cost. For example when a penetration is detected and the ODE solver has to rollback the simulation state, it can do so without affecting bodies from other simulation groups. Also, since many algorithms do not run in the linear time with respect to the number of bodies or constraints to be handled, the separation to simulation groups is really advantageous (the total cost is smaller than the cost of the same work performed straight on all bodies). On the other side, bodies that did not influence each other in the past can begin to influence each other at a later time and the simulator must be able to detect when this is going to happen and merge the corresponding simulation groups (ie. two cars can be simulated independently until they crash into each other — the collision must be handled in the context of a common simulation group). Therefore the separation to simulation groups must be dynamic and under a condition that the simulation effects are (more or less) the same as if the bodies were all simulated within a single simulation group.

Since the only way to influence a rigid body state is to exert a force or an impulse on the corresponding body, the simulator keeps track of what forces (impulses) might be exerted so that it could partition bodies to simulation groups appropriately. The contributions are represented by **ForceObject** instances that are queried by the simulator to describe what rigid bodies the force object might act or depend on and provide functions to exert the represented force (impulse) on the affected bodies, updating their force and torque accumulators. There are *global forces* (such as gravity) that can potentially act on all bodies, where the force exerted on a body does not depend on the state of another body, and general *n-ary* forces, that can act or depend on an a priori known set of bodies only. A particular n-ary force is always assigned to a certain simulation group and the simulator ensures that all bodies the force object acts or depends on belong to the same simulation group, which the force object is assigned to. Rigid body constraints pursued in the previous chapter, in the simulator jargon called the *joints*, are a special kind of binary forces, abstracted by **Joint** class. Non-penetration and contact constraints are handled by the simulator automatically, with the help of collision detection. For performance reasons, bodies with overlapping geometry hulls are always simulated together, regardless of whether they already contact or not.

Rigid bodies are simulated in terms of (5.18) and are represented by **RigidBody** instances. The instances store the rigid body states (5.17), their mass properties, collision geometries attached to the bodies, force and torque accumulators (the force objects contribute to) and provide various functions to be used by the simulation group's ODE solver (**get_state**, **set_state**, **evaluate_derivative**) and force objects (**apply_force**). In order to support complex scenes, the simulator distinguishes among *enabled* bodies, that respond to forces and impulses, *disabled* bodies, that currently do not respond to forces nor impulses but can be enabled upon a request, and *fixed* bodies, that are permanently disabled and serve the purpose of the static geometry. In addition, bodies that have been at rest[1] for a certain amount of time are classified *idle* (an idle indicator is turned on) and the simulation group is no longer simulated if it consists of idle bodies until the group is merged to another group. This is an aggressive optimization based on the assumption that if all bodies (within the group) are idle then the forces that act upon them must be at an equilibrium which can not break until another force begins to act[2].

The actual simulation work is performed by the instances of **Simulator::SimulationGroup**

[1]Their linear and angular velocities and force and torque accumulators did not exceed a preset limit.

[2]This can not happen until a different simulation group is merged to this group.

that represent and simulate individual simulation groups. Simulation groups maintain lists of bodies and force objects belonging to the corresponding groups and have their own ODE solvers that govern the motion of the bodies in the group. **Simulator** instance "merely" registers rigid bodies, force objects and simulation groups and coordinates the simulation — partitions bodies to simulation groups, advances individual groups, merges groups as needed, etc.

7.2 Characteristics

We have implemented an extensible general purpose constrained rigid body dynamics simulator (the simulator library), designed according to the approach discussed by this thesis and capable of handling very different kinds of simulations, ranging from the simulation of mere non-penetrating bodies to the simulation of various kinds of vehicles, highly articulated structures or human-like figures.

Apart from the simulator library we have also implemented the figure library, suitable for the animation of articulated human-like figures, the test library, and the demonstration application. These libraries will be described later.

We chose `C++` as the implementation language due to the high availability of optimizing compilers capable of producing tight and efficient code, yet allowing for a high-level of code abstraction. The simulator is written literally from the ground up and does not use any third-party library, except for the standard `C++` library (the test library and the demo application, separated from the simulator, require `Microsoft Platform SDK` to compile and `Windows 95` or better to run). As a development environment we chose `Microsoft Visual C++`, version 6.0[3] or later, but port to a different compiler should be trivial.

Although similar systems exists, they are often commercial, the source code is not publicly available and little is known of the technology used. Open source systems often suffer from the quality of implementation or the source code is not readable, maintainable or easy to extend. Contrary, in our implementation, an extra care is paid to the abstraction of concepts, extensibility, readability and maintainability of the source code as well as the quality of implementation and the source code documentation.

The simulator currently implements the following major features:

- Constrained rigid body dynamics, equations of motion expressed in terms of "maximal coordinates" approach.
- Global constraint solver. Support for equality, inequality and bounded equality constraints. Lagrange multiplier based analytical solvers. Baraff's linear time dynamics method. Reduction to LCP.
- Generic infrastructure for solving ODEs. Support for higher order integration methods.
- Simulation groups.
- Support for multiple contacts with friction. Various friction models.
- Adaptive time stepping, rollback on body inter-penetration.
- Collision geometries defined as volumes.
- Self-tests.

[3]`STLPort`, an alternative implementation of the standard `C++` library, has to be used in the case of `MSVC 6.0`.

To run the demo application, the following minimum hardware and software requirements have to be met. However, as seen from the present day perspective, the requirements seem to be somewhat insignificant. The most of the demo and test scenes provided with the simulator run moderately on the following configuration:

- A `Pentium` class processor, 24 Mb of main computer memory.
- An `OpenGL` 1.1 accelerated graphics card, 2 Mb of video memory.
- `Windows 95`[4] or `Windows NT 4.0`.

7.3 Components

The source code of the rigid body simulator is organized into several functional, mostly independent, libraries. Each library performs a specific piece of work, which we are going to characterize now.

7.3.1 Algebra

Algebra library provides classes that abstract floating point math, represent matrices, vectors, quaternions, orientations and affine transformations, define standard operations on these structures and represent and implement other linear algebra concepts and algorithms. The used theory is mostly based on the work of [14], [1] and [9].

One of the simulator goals is to be able to operate in various floating point math precision modes. Algebra classes are therefore implemented as template classes, parametrized by the floating point type to be used to represent real numbers. Template class **MathLib**, specialized for the supported floating point types, implement the elementary math functions in terms of the generic interface that is used by other classes to perform the math.

Template class **Matrix** implements matrices with static or dynamic dimensions and provides classes that represent views to bound matrices (matrix slices). Matrix views allow to formulate many algorithms in a readable way (elements of the matrix view/slice are indexed instead of the elements of the original matrix) and provide an optimization opportunity for the compiler. The implementation of the matrix and the corresponding view classes is specialized and optimized for the most commonly used cases, which allows to use the same generic interface in different contexts without risking a performance loss[5].

Counter-clockwise orientations (rotations) in three dimensional space are abstracted by **Orientation** class that internally uses rotation quaternions to represent orientations but also accepts and provides functions to convert between the following orientation representations — Euler angles in the order $Z-X-Y$, rotation matrix, axis & angle pair, rotation quaternion. Coordinate axes are oriented so that the X axis points to the right, Y axis points forward, Z axis upwards. In the case of world space axes, the X axis points to the east, Y axis points to the north, Z axis points upwards.

Factorization classes **LDLTFactorization** and **CholeskyFactorization** are also an important part of the **Algebra** library. They implement efficient algorithms for the factorization of symmetric (**LDLTFactorization**) and positive-definite matrices (**CholeskyFactorization**) and provide modified versions of the standard algorithms that allow to refactor only a portion

[4]`Windows NT`-compatible command line shell is required for the parsing of the demo batch files.

[5]When the simulator is compiled under release mode, certain classes are implemented even more aggressively which allows for improved performance.

of the corresponding matrix, when a row and a column is added to or removed from the matrix. The factorization classes are used to implement the internals of constraint and LCP solvers.

7.3.2 Geometry

Geometry library abstracts collision geometries and supports and implements collision detection according to [10] and [22]. Collision geometries are represented by **CollisionGeometry** superclass providing access to the collision geometry's geometry hull (an axes aligned bounding box), an infrastructure for attaching geometries to parent geometries and rigid bodies and mapping of the local collision geometry spaces, the geometries are defined at, to the *parent* spaces (local spaces of the parent geometries or the world space). This allows to create geometry hierarchies and compose complex collision geometries from more primitive geometries. The library handles the following geometry types and implements classes to perform low-level collision detection among these types,

- Convex polyhedra

  Convex polyhedra are volumes defined as the intersection of a set of half-spaces determined by their boundary planes. Convex polyhedra are represented by the instances of **Polytope** class, which store the boundary planes and other related information distilled from the polytope definitions, such as lists of polytope faces, edges and vertices together with the topology information, utilized by the collision detection code.

- Spheres

  Spheres are defined as volumes with centers $\vec{c}$ and radii r that are parametrized by points $\vec{p}$, where $\|\vec{p} - \vec{r}\| \leq r$. They are represented by the instances of **Sphere** class.

- Capped cylinders

  Capped cylinders are volumes defined by two end points $\vec{a}$ and $\vec{b}$ and radii r, parametrized by points $\vec{p}$ so that $dist(\vec{p}, \overrightarrow{ab}) \leq r$, where $dist(\vec{p}, \overrightarrow{ab})$ denotes the distance of point $\vec{p}$ from line $\overrightarrow{ab}$. Capped cylinders are usual cylinders with radii r and their principal axes pointing along the directions of $\overrightarrow{ab}$ with the addition of two sphere "caps" at their end points. Capped cylinders can be seen as volumes swept by spheres with radii r when they are translated from $\vec{a}$ to $\vec{b}$ along straight lines. Capped cylinders are represented by instances of **CappedCylinder** class.

- Spaces

  Spaces are containers for other geometry types. They represent volumes corresponding to the unions of the (appropriately transformed) owned geometries and are implemented by **Space** class.

CollisionDetection class manages collision geometries and implements higher-level and lower-level collision detection. Higher-level collision detection operates upon geometry hulls of collision geometries attached to rigid bodies and utilizes the *sort and sweep* algorithm to determine all overlapping geometry hull pairs. Geometry hulls are represented by bounding boxes with their sides aligned with the world space axes.

Lower-level collision detection invokes geometry type pair specific algorithms, encapsulated by the subclasses of **CollisionDetection**::**Collider** class, that test for penetration and compute contacts if the tested geometries do not penetrate. Collider classes often rely on the fact that contacts need not be computed on penetrations (in fact it would be hard to define what the contacts should be if geometries already penetrated), which allows to use more efficient algorithms.

7.3.3 Differential Equation Solver

Differential Equation Solver library implements ODE solvers according to the outline from section (2.3) and provides some extensions required by the simulator to handle multiple simulation groups that are governed by their own ODEs. ODE solvers are abstracted by **DifferentialEquationSolver** superclass that provides the interface towards the ODE solver user, which is the **Simulator**. The ODE problems to be solved are represented by the instances of **DifferentialEquationSolver::Problem** class providing the interface towards the ODE solver.

The implementation of ODE solvers is logically separated to the solver method specific parts that implement the logic of taking steps (**make_step** function implemented by **DifferentialEquationSolver** subclasses) and the solver neutral parts that implement the logic of advancing system states to desired times in terms of **make_step** function, incorporating the bisection algorithm (**advance** function implemented by **DifferentialEquationSolver** superclass). Fixed time stepping as well as adaptive time stepping variants of Euler, Midpoint and Runge-Kutta solvers are currently implemented.

As we said many times, the simulation loop is "hidden" by the logic of the ODE solver that is used to solve the equations of motion. If all bodies were simulated in terms of a single ODE (single simulation group), one would simply invoke **advance** and the ODE solver would attempt to advance the state to the desired time, avoiding invalid states (penetrations) by rolling system states back when necessary and running the bisection algorithm to locate latest valid states.

In the case of multiple simulation groups we have to proceed a little differently. Since each group has its own equations of motion and an own ODE solver, we have to advance multiple ODEs, updating one ODE at a time[6], while ensuring that all ODEs will end up at the same time when done. For example if we have n simulation groups denoted G_i, $1 \leq i \leq n$, update the groups in the order determined by their indices, starting with G_1, and group G_i fails to advance from time t to $t_{desired}$ for a certain reason, then groups G_j, $1 \leq j < i$, already advanced to $t_{desired}$ and groups G_j, $i < j$, have not advanced yet and reside at time t, then we have to roll groups G_j, $1 \leq j \leq i$ back to t so that all groups will end up at the same time t. This shows that the advancing of individual simulation groups must be coordinated.

To support the handling of multiple ODEs, **DifferentialEquationSolver** allows to advance the ODE state within the context of a transaction (function **transacted_advance**). The transaction aborts and the system state rolls back to the state prior the issuing of the transaction if the ODE solver fails to advance to the desired time or the transaction is aborted explicitly. If the solver advances to the desired time, the ODE solver asks the problem instance for a permission to commit, which can be either granted or dismissed. The transaction then either commits or aborts. This mechanism can be used to implement the processing of chained transactions, where the previous transaction waits for the completion of the next transaction in order to commit. We will show in the next section how transactions are used to advance the whole simulation state.

There is nothing special about the implementation of ODE solvers except for the management of system states and their derivatives which is optimized to minimize system state transfers between the ODE solver and the problem instance and the evaluations of state derivatives (functions **get_state**, **set_state**, **evaluate_derivative**).

[6]An attempt to update all simulation groups altogether, synchronizing the time step, would compromise all benefits of multiple simulation groups.

7.3.4 Simulation

Simulation library abstracts rigid bodies, mass properties of rigid bodies, forces, joints, simulation groups and uses other components to implement the “simulator”. Section (7.1) presents the most important classes and the simulator implementation overview. We already know that simulation groups have their own ODE solvers that simulate the particular groups and that the individual solvers must be coordinated in order to produce correct results. We will now focus at the description of this coordination.

Simulator must ensure that bodies that might be influenced by a force or an impulse depending on the state of another body belong to the simulation group of that body. Since forces and impulses are abstracted by force objects, simulator can query these objects to correctly separate bodies to simulation groups. Whenever a new force object is registered or the set of bodies the force object acts or depends on changes, the simulator is notified so that its simulation groups can be merged as required. Global forces, affecting bodies independently on each other, are stored in an auxiliary simulation group of global forces while n-ary forces acting or depending on (affecting) an a priori known set of bodies are assigned to the simulation groups of the affected bodies, called the force object targets.

Advancing Simulator

Simulator’s **advance** function is responsible for advancing the whole simulation state to a desired time, advancing and coordinating the individual ODE solvers due to simulation groups. Its work consists of the following steps, which we will describe momentarily,

- Distribute rigid bodies to simulation groups, restart the simulation group’s ODE solvers.

  - Ensure that collision geometries are up-to-date with respect to the current rigid body states, update collision geometries where needed and update collision detection structures to reflect the current geometry state.

    The simulator requires rigid bodies with overlapping geometry hulls to be simulated within a common simulation group and so the geometries must be updated before simulation groups are advanced in order to obey this requirement. If collision geometries are actually updated at this step, new overlaps might be detected, producing requests to merge simulation groups.

    Unless there are simulation groups whose ODE solvers must be restarted, collision geometries are up-to-date already and no work has to be done.

  - Merge simulation groups due to group membership constraints.

    These constraints already hold and simulation groups need not be merged unless there are pending requests to merge groups due to geometry hull overlaps detected at the previous step or due to the change of force object targets.

  - Restart individual ODE solvers where requested.

    A simulation group’s ODE solver must be restarted (function **restart** called) whenever the group’s ODE state changes explicitly (which happens when a new body is added to or removed from the group or a body’s state is modified, for example) so that the solver can start from the new initial state.

- Advance individual simulation groups.

Simulation groups are advanced in contexts of chained transactions, where each transaction advances a corresponding simulation group to a desired time. If we have n simulation groups denoted $G_1, \ldots, G_n$ then there are n chained transactions, where the i-th transaction advances G_i, the first transaction is triggered by the simulator and the j-th transaction ($j \geq 2$) by the ($j-1$)-th transaction.

The chaining of transactions works as follows. The simulator triggers the first transaction to advance G_1 (while advancing G_1 the other simulation groups are not updated). If G_1 reaches the desired time, then the second transaction is triggered to advance G_2 and the first transaction waits for the second transaction to complete. If the second transaction commits then the first transaction must commit too and if the second transaction aborts then the first transaction must abort. The same rule applies to the i-th and the ($i+1$)-th transactions (the ($i+1$)-th transaction is triggered by the i-th transaction when G_i reaches the desired time, the i-th transaction commits or aborts depending on the completion status of the ($i+1$)-th transaction). As a result, either all transactions $1, \ldots, n$ commit if all G_i reach the desired time or transactions $1 \leq j \leq i$ abort if G_i fails to advance.

The transactional processing is required in order to ensure consistency and do proper individual rollbacks when certain ODE solver (due to G_i) can not advance to the desired time. It remains to show why transaction i can ever fail (besides the obvious reasons, such as numerical problems when penetration can not be avoided, for example). The reason is the need for simulation group merging. Although force object targets can not change while advancing the simulation, new geometry hull overlaps (and contacts) might emerge which requires the simulator to simulate the corresponding bodies within a common group in order to produce consistent results.

To illustrate the problem, let us assume that we have two bodies A and B, so that the first body belongs to G_1 and the second body to G_2 and the corresponding geometry hulls do not overlap at time t. Now let us advance the simulation state. G_1 is advanced first and while advancing G_1, it can happen that the geometry hull of A will overlap the hull of B. The trouble is that when the overlap is detected, at time $t_{cur} \neq t$, the geometry hull of A is valid with respect to time t_{cur} but the geometry hull of B is valid with respect to t, because G_2 was not updated yet. Even if we could merge the corresponding simulation groups straight in the middle of the transaction so that we could handle the overlap (which we can not), potential contacts would be inconsistent because they would be computed with respect to geometry states at different times — hence the inconsistency. The (chained) transaction is thus aborted, the simulation state restored to the state at time t, bodies merged to a common simulation group and the transaction rerun.

Advancing Simulation Group

Simulation groups maintain lists of rigid bodies and force objects assigned to them so that they could build the ODE that determines the evolution of the group in time and compute the equation right-hand side (evaluate ODE state derivative) upon the request of the ODE solver that is specific to the simulation group. A particular simulation group is then advanced in terms of its ODE. We will now outline the steps taken by **evaluate_derivative**,

- Collision detection is performed

  Higher-level collision detection runs and the list of overlapping geometry hulls is updated. New geometry hull overlaps among bodies from different simulation groups indicate that the transaction should abort and bodies involved in the overlaps merged to common groups (the evaluation of the ODE state derivative ends with a request to abort the transaction).

Lower-level collision detection involving bodies in the simulation group is performed. Penetrations indicate that the current state is invalid and the ODE solver should rollback and locate the latest valid state (the evaluation of the ODE state derivative ends with an indication that the current state is invalid). Otherwise, contacts among bodies in the simulation group are computed and temporary contact joints, represented by the instances of classContactJoint, created to handle contact.

- External forces and torques are computed

  Force and torque accumulators of the bodies in the group are cleared, the list of force objects assigned to the simulation group and the list of global forces traversed to accumulate the force and torque contributions.

  If we are in the middle of an ODE step (the derivative is evaluated for an intermediate state), impulses are ignored.

- Constraints are processed

  Dynamics constraints represented by the instances of **Joint** class assigned to this simulation group (including the implicit joints due to contacts) are traversed to determine what constraints are to be maintained on the velocity and acceleration levels and constraint impulses and forces are computed and applied.

  Non-penetration is enforced by stopping bodies at critical contacts. Contact joints are finally removed.

- The equation right-hand side is returned

  Force and torque accumulators and the rigid body states are used to fill the right-hand sides of (5.18) due to individual bodies in the group.

  If impulses were applied at this invocation of the function, the ODE solver is notified that the ODE state has changed.

7.3.5 Constraint Solver

Constraints on the acceleration and velocity levels are represented by a special kind of binary forces called joints (instances of **Joint** class). Joints provide functions to describe the constraints in the form of (6.17) and (6.26) and their corresponding soft variants. Constraint solver traverses the list of joints and invokes these functions to retrieve the velocity level or acceleration level formulations of the constraints, described in terms of the instances of **Joint*::*BasicInformation** (providing the information on the constraint DOFs, the dimensionality m_i of the constraint) and **Joint*::*ExtendedInformation** (providing the values of the Jacobian blocks J_{i,A_i} and J_{i,B_i}, constraint right-hand side vector $\vec{c}_i$ or $\vec{k}_i$, Lagrange multiplier limits $\vec{\lambda}_i^{lo}$ and $\vec{\lambda}_i^{hi}$ and softness constants $\vec{s}_i$). The formulations are in turn used to develop conditions on the multipliers $\vec{\lambda}$, the multipliers are computed and constraint force (impulse) exerted on the bodies in the group. Velocity level constraints are solved first, yielding a change of the ODE state. Acceleration level constraints are then computed with respect to the new ODE state. We will now describe this process more thoroughly.

There are some differences from the outline we have just presented. First, bodies in the simulation group are further partitioned to so called *islands* defined as the groups of bodies (subsets of the bodies in the simulation group) that are currently connected by a joint, [22]. If we have two islands A and B, then the constraint force due to constraints in the island A can not depend on the constraints in the island B, because the islands are not connected by a joint. Hence constraints can be processed by islands, handling each island independently on each other (this reduces the computational cost). Each simulation group maintains a *body-joint graph* (bodies are the vertices and joints are the edges connecting the bodies affected by the particular joint) whose components correspond to the islands. Second, the instances of **Joint::BasicInformation** provide information on the types of Lagrange multiplier limits, which allows to determine in advance whether all constraint rows are unbounded equalities, for example.

Constraints in the simulation group are handled as follows. Body-joint graph is traversed, its components (islands) identified and bodies and constraints indexed, yielding the indices of bodies and constraints in the corresponding islands (these indices determine the layout of matrix J due to the bodies and constraints in the island, the layout of $\vec{F}_{total}$, etc). Whenever an island is found, velocity-level constraint solver is invoked to solve velocity-level constraints in the island (if the current state is not intermediate), followed by the invocation of acceleration-level constraint solver to solve acceleration-level constraints.

Currently, we implement two *analytical* constraint solvers — **LagrangeMultiplierSolver** and **FastLagrangeMultiplierSolver**. Since the implementation of acceleration-level and velocity-level constraint solvers is similar (ie. forces vs. impulses and total external forces vs. momentums), we abstract the differences by appropriate *traits* classes and implement the relevant code as templates.

LagrangeMultiplierSolver solves constraints in terms of dense matrix A from (6.8) and its implementation strictly follows the notes from chapter 6 to reduce the problem to LCP. Dantzig LCP solver outlined in section (6.2.5) is used to solve the LCP, sparsity of matrix J is exploited to build A and $\vec{b}$ efficiently. The algorithm requires $O(n^2)$ time to formulate the LCP for n constraints and an extra time to solve the LCP.

FastLagrangeMultiplierSolver is the implementation of the algorithm from [9]. The solver distinguishes between two kinds of constraints. Primary constraints are those pure equality constraints[7] that do not cause cycles in the body-joint graph. Auxiliary constraints are the remaining constraints (equality constraints, that yield a cycle in the graph, bounded equality constraints and inequality constraints). The goal is to solve primary constraints in the linear time and combine the results with auxiliary constraints. Auxiliary constraints are solved by a reduction to LCP that "anticipates" the response of the primary constraints to constraint force due to auxiliary constraints (auxiliary constraint force is computed so that when it is combined with the constraint force due to primary constraints, both primary and *also auxiliary constraints* will hold). The algorithm is described in the Reference Guide and consumes $O(n)$ time to handle n primary constraints, $O(n \cdot k)$ time to formulate the LCP for k auxiliary constraints (typically $k \ll n$) and an extra time to solve the LCP. LCP matrix A is no longer positive definite, but it is still symmetric and so **LDLTFactorization** is employed by the LCP solver.

The algorithm is based on the fact that matrix H from (6.9) for primary constraints is a $(n+c) \times (n+c)$ block matrix whose block-rows and block-columns can be associated with constrained bodies and primary constraints. The goal is to reorder the block columns/rows so that the permuted H, denoted H', can be factored in the linear time. Each permutation of matrix H corresponds to a certain ordering of bodies and primary joints. The coveted permutation is retrieved by traversing the body-joint graph using a depth-first-search algorithm and ordering the blocks of H' according to the postorder order in which the bodies and primary joints are visited, [9].

If J is the Jacobian matrix due to primary joints and $\vec{F}$ is a generalized force acting on the system then the system's accelerational response in reaction to $\vec{F}$, considering the primary constraints and ignoring the auxiliary constraints, equals $\vec{a} = M^{-1} \cdot (\vec{F} + J^T \cdot \vec{\lambda}_{\vec{F}})$, where $J^T \cdot \vec{\lambda}_{\vec{F}}$ is the constraint force due to primary constraints exerted in reaction to $\vec{F}$. Since we can factor H' and solve for $\vec{\lambda}_F$ in the linear time, we can compute also $\vec{a}$ in the linear time. We will write that the acceleration $\vec{a}$ of the system in response to $\vec{F}$, accounting for the effects of the

[7] All constraint rows are unbounded equalities.

primary constraints, equals

$$\vec{a} = \hat{M}^{-1} \cdot \vec{F} = M^{-1} \cdot (\vec{F} + J^T \cdot \vec{\lambda}_{\vec{F}}), \tag{7.1}$$

which can be seen as a modified version of the equation of motion (5.23) that directly incorporates the effects of the primary constraints.

If we have a set of auxiliary constraints formulated in the matrix form as $J_{aux} \cdot \vec{a} = \vec{c}_{aux}$, where $\vec{a}$ is the total acceleration of the system after all forces are applied, then the constraint force to maintain the auxiliary constraints will again equal $J_{aux}^T \cdot \vec{\mu}$, where $\vec{\mu}$ is the vector of Lagrange multipliers due to these constraints. By utilizing (7.1), we realize that it is required that

$$J_{aux} \cdot (\hat{M}^{-1} \cdot (\vec{F}_{total} + J_{aux}^T \cdot \vec{\mu})) = \vec{c}_{aux}, \tag{7.2}$$

which says that auxiliary constraints and primary constraints must hold (the conditions on the primary constraints are "hidden" in $\hat{M}^{-1}$). This can be rewritten as $J_{aux} \cdot \hat{M}^{-1} \cdot J_{aux}^T \cdot \vec{\mu} = \vec{c}_{aux} - J_{aux} \cdot \hat{M}^{-1} \cdot \vec{F}_{total}$. Considering that auxiliary constraints might impose limits on $\vec{\mu}$, we obtain a LCP in the form of $A \cdot \vec{\mu} + \vec{b} = \vec{0}$ with $\vec{\mu}_{lo} \leq \vec{\mu}_{hi}$ limits, where $A = J_{aux} \cdot \hat{M}^{-1} \cdot J_{aux}^T$ and $\vec{b} = J_{aux} \cdot \hat{M}^{-1} \cdot \vec{F}_{total} - \vec{c}_{aux}$. If we had access to $\hat{M}^{-1}$ we could build A and $\vec{b}$ exactly in the same way like we did it at (6.8). The good news is that matrix $\hat{M}^{-1}$ need not actually be computed at all and the coefficients of A and $\vec{b}$ can be computed in terms of $\vec{F}$, M^{-1} and $J^T \cdot \vec{\lambda}_{\vec{F}}$ from (7.1). Indeed, the i-th column of matrix A, denoted A_i, can be computed as $A_i = (J_{aux} \cdot \hat{M}^{-1} \cdot J_{aux}^T)_i = J_{aux} \cdot (\hat{M}^{-1} \cdot J_{aux}^T)_i = J_{aux} \cdot (\hat{M}^{-1} \cdot (J_{aux}^T)_i) = J_{aux} \cdot (M^{-1} \cdot ((J_{aux}^T)_i + J^T \cdot \vec{\lambda}_{(J_{aux}^T)_i}))$ and $\vec{b} = J_{aux} \cdot (M^{-1} \cdot (\vec{F}_{total} + J^T \cdot \vec{\lambda}_{\vec{F}_{total}})) - \vec{c}_{aux}$.

The solver identifies the primary and auxiliary constraints, orders the block columns (rows) of matrix H', builds and solves the LCP to obtain the auxiliary constraint force $J_{aux}^T \cdot \vec{\mu}$ (that anticipates the response of the primary constraints to $\vec{F}_{total} + J_{aux}^T \cdot \vec{\mu}$) and exerts it on the system, yielding the new total external force $\vec{F}_{total} + J_{aux}^T \cdot \vec{\mu}$. Primary constraint force $J^T \cdot \vec{\lambda}_{\vec{F}_{total} + J_{aux}^T \cdot \vec{\mu}}$ is then computed with respect to $\vec{F}_{total} + J_{aux}^T \cdot \vec{\mu}$ and exerted. Primary constraints will obviously hold then and auxiliary constraints will hold as well, because the effects of the primary constraint force were anticipated when $\vec{\mu}$ was computed, see (7.2).

7.3.6 LCP Solver

Linear Complementarity Problem Solver library abstracts LCP problems and implements LCP solvers. Class **BasicLCP** abstracts pure and mixed LCP problems, class **LoHiLCP** lo-hi LCP problems. There are several solver implementations — **DebugBasicLCPSolver** solves basic LCP problems and proceeds directly as described by Baraff in [7], **BasicLCPSolver** is the optimized version of the basic LCP solver and **LoHiLCPSolver** is the generalized basic solver outlined in section (6.2.5) capable of solving lo-hi LCP problems. **BasicLCPSolver** and **LoHiLCPSolver** solvers are implemented as template classes parametrized by *traits* classes that hide the manipulation with the slices of matrix A and vectors $\vec{b}$, $\vec{\lambda}$, $\vec{\lambda}^{lo}$, $\vec{\lambda}^{hi}$. That way, we are able to provide optimized and unoptimized variants of the solvers, where the optimized versions permute rows and columns of A (and the rows of vectors $\vec{b}$, $\vec{\lambda}$, $\vec{\lambda}^{lo}$, $\vec{\lambda}^{hi}$) during the course of solving the LCP in order to minimize memory transfers while updating A_{CC} and allow to access A_{CC} slices efficiently. These optimization tricks were adopted from [22].

7.4 Joints

Simulator library provides the implementation of certain joint classes that can be used to connect bodies by virtual hinges or nails, restrict and control rotations about specified axes, implement various motors, etc. Joints allow to build articulated structures and simulate other types of objects than mere rigid bodies.

Built-in joint support can be separated to two categories, the category of anchors, used to anchor bodies, and the category of motors, allowing to control joint angles, displacements along axes, rotation speeds, etc. Since joints are (basically) formulated in terms of their Jacobian blocks, right-hand side vectors and softness parameters, one can implement new joint types by providing functions to evaluate these quantities upon the simulator's request, that is, by inheriting from **Joint** class and implementing functions returning the basic and extended joint informations, as prescribed by the **Joint** interface.

We will now present the two joint type categories and note that the list of all joint types, their properties and formulations (including the derivations) can be found in the Reference Guide.

7.4.1 Anchors

Anchors are formally defined as position-level equality constraints (yet implemented either on the velocity or acceleration levels) that restrict relative body positions and orientations. They implement virtual nails, pins, hinges and are used to anchor bodies together. Anchor joints are often combined with motor joints to allow the control of the other DOFs, otherwise unconstrained by the joint.

PointToPoint is a base class for anchor joints providing an infrastructure for defining the positions of anchors (points) in the body spaces of the constrained bodies and functions to formulate the "point-to-point" constraint that requires the anchors to occupy the same world space position. Subclasses can augment the constraint formulation to impose a more "strict" constraint such as that bodies can not rotate with respect to each other along a specified axis, etc.

For an illustration, we will show how the constraint formulation for **PointToPoint** class can be derived, utilizing the equations and notation from chapters 5 and 6. If i is the index of the constraint, A_i and B_i are the indices of the constrained bodies and $\vec{r}^{\,b}_{A_i}$ and $\vec{r}^{\,b}_{B_i}$ are the positions of the two anchors on the first and second body, expressed in the corresponding body spaces, then $\vec{p}_{A_i} = \vec{x}_{A_i} + R_{A_i} \cdot \vec{r}^{\,b}_{A_i}$ and $\vec{p}_{B_i} = \vec{x}_{B_i} + R_{B_i} \cdot \vec{r}^{\,b}_{B_i}$ are the world space positions of the anchors and the goal is to ensure that $\vec{p}_{A_i} = \vec{p}_{B_i}$. Let us denote $\vec{r}_{A_i} = \vec{p}_{A_i} - \vec{x}_{A_i}$ and $\vec{r}_{B_i} = \vec{p}_{B_i} - \vec{x}_{B_i}$. These vectors are attached to the first and second body (so $\dot{\vec{r}}_{A_i} = \vec{\omega}_{A_i} \times \vec{r}_{A_i}$ and $\dot{\vec{r}}_{B_i} = \vec{\omega}_{B_i} \times \vec{r}_{B_i}$, (5.5)) and the constraint can be then formulated as

$$
\begin{aligned}
\vec{C}^{i}_{p} &= \vec{p}_{B_i} - \vec{p}_{A_i} = \vec{x}_{B_i} + \vec{r}_{B_i} - \vec{x}_{A_i} - \vec{r}_{A_i} = \vec{0} \\
\dot{\vec{C}}^{i}_{p} &= \dot{\vec{x}}_{B_i} + \vec{\omega}_{B_i} \times \vec{r}_{B_i} - \dot{\vec{x}}_{A_i} - \vec{\omega}_{A_i} \times \vec{r}_{A_i} = \vec{0} \\
\ddot{\vec{C}}^{i}_{p} &= \ddot{\vec{x}}_{B_i} + \vec{\alpha}_{B_i} \times \vec{r}_{B_i} + \vec{\omega}_{B_i} \times (\vec{\omega}_{B_i} \times \vec{r}_{B_i}) \\
&\quad - \ddot{\vec{x}}_{A_i} - \vec{\alpha}_{A_i} \times \vec{r}_{A_i} - \vec{\omega}_{A_i} \times (\vec{\omega}_{A_i} \times \vec{r}_{A_i}) \\
&= E \cdot \ddot{\vec{x}}_{B_i} - \vec{r}^{\,*}_{B_i} \cdot \vec{\alpha}_{B_i} + \vec{\omega}_{B_i} \times (\vec{\omega}_{B_i} \times \vec{r}_{B_i}) \\
&\quad - E \cdot \ddot{\vec{x}}_{A_i} + \vec{r}^{\,*}_{A_i} \cdot \vec{\alpha}_{A_i} - \vec{\omega}_{A_i} \times (\vec{\omega}_{A_i} \times \vec{r}_{A_i}) = \vec{0},
\end{aligned}
\tag{7.3}
$$

obtaining

$$
\begin{aligned}
J_{i,A_i} &= \begin{pmatrix} J^{linear}_{i,A_i} & J^{angular}_{i,A_i} \end{pmatrix} = \begin{pmatrix} -E & \vec{r}^{\,*}_{A_i} \end{pmatrix} \\
J_{i,B_i} &= \begin{pmatrix} J^{linear}_{i,B_i} & J^{angular}_{i,B_i} \end{pmatrix} = \begin{pmatrix} E & -\vec{r}^{\,*}_{B_i} \end{pmatrix} \\
\dot{J}_i \cdot \vec{v} &= \vec{\omega}_{B_i} \times (\vec{\omega}_{B_i} \times \vec{r}_{B_i}) - \vec{\omega}_{A_i} \times (\vec{\omega}_{A_i} \times \vec{r}_{A_i}) \\
\vec{C}^{i}_{p} &= \vec{p}_{B_i} - \vec{p}_{A_i} \\
m_i &= 3,
\end{aligned}
\tag{7.4}
$$

which can be directly substituted into (6.33) to yield the final acceleration-level and velocity-level formulations with the stabilization terms.

7.4.2 Motors

Motors are usually defined as equality or inequality constraints formulated directly on the velocity level[8] that restrict relative body angular or linear velocity along specified axes. They implement joint angle or displacement limits and motors at joints. Although the constraints are formulated directly on the velocity level and finally implemented either on the velocity or acceleration level, they often represent position-level constraints (such as joint angle/displacement limits) and so the notes from section (6.4) are considered.

PoweredAxis provides basic support for the implementation of angular (rotational) or linear (translational) joint limits and motors. Given an angular axis $\vec{a}_i$ attached to the first body and expressed in the world coordinates and an angle α_i the second body is rotated about $\vec{a}_i$ (with respect to the initial relative orientation), the class imposes an 1 DOF constraint on the relative angular velocity so that specified angle limits $\alpha_i^{lo} \leq \alpha_i^{hi}$ will not be exceeded[9] or a certain angle or an angular velocity will be attained. If there are joint angle limits associated with the axis, we say that the axis is *limited*. If a certain joint angle or angular velocity should be followed, we say that the axis is *powered*.

We will now present the formulation of an 1-DOF velocity-level constraint (with a Baumgarte stabilization term due to constant $\alpha > 0$) that ensures that angle limits are obeyed[10] and a desired angle or angular velocity followed if the angle is not at a limit,

- if the angle is at the lower limit α_i^{lo} then equation (6.34) is in effect and $\dot{C}_p^i = \vec{a}_i \cdot (\vec{\omega}_{B_i} - \vec{\omega}_{A_i}) - k_i \geq 0$ constraint imposed, where k_i is chosen to stabilize the position-level error $\alpha_i^{lo} - \alpha_i$, $k_i = (\alpha_i^{lo} - \alpha_i) \cdot \alpha$,
- if the angle is at the upper limit α_i^{hi} then equation (6.34) is in effect and $\dot{C}_p^i = \vec{a}_i \cdot (\vec{\omega}_{B_i} - \vec{\omega}_{A_i}) - k_i \leq 0$ constraint imposed, where k_i is chosen to stabilize the position-level error $\alpha_i^{hi} - \alpha_i$, $k_i = (\alpha_i^{hi} - \alpha_i) \cdot \alpha$,
- if the angle is not at a limit and a desired angle $\alpha_i^{desired}$ should be followed then equation (6.33) is in effect and $\dot{C}_p^i = \vec{a}_i \cdot (\vec{\omega}_{B_i} - \vec{\omega}_{A_i}) - k_i = 0$ constraint imposed, where k_i is chosen to stabilize the position-level error $\alpha_i^{desired} - \alpha_i$, $k_i = (\alpha_i^{desired} - \alpha_i) \cdot \alpha$,
- if the angle is not at a limit and a desired velocity $\dot{\alpha}_i^{desired}$ should be followed then $\dot{C}_p^i = \vec{a}_i \cdot (\vec{\omega}_{B_i} - \vec{\omega}_{A_i}) - \dot{\alpha}_i^{desired} = 0$ constraint is imposed, see equation (6.26).

If the angle is at a limit and the axis is powered, we impose a constraint due to the angle limit only and approximate the axis motor by an external force.

Obeying the rules from section (6.4), we can implement the constraint on the acceleration level. In that case we will work with $\ddot{C}_p^i = \vec{a}_i \cdot (\dot{\vec{\omega}}_{B_i} - \dot{\vec{\omega}}_{A_i}) + (\vec{\omega}_{A_i} \times \vec{a}_i) \cdot (\vec{\omega}_{B_i} - \vec{\omega}_{A_i})$ and will have to stabilize both the position-level and velocity-level errors. We obtain

$$\begin{aligned} J_{i,A_i} &= \begin{pmatrix} J_{i,A_i}^{linear} & J_{i,A_i}^{angular} \end{pmatrix} = \begin{pmatrix} 0 & -\vec{a}_i \end{pmatrix} \\ J_{i,B_i} &= \begin{pmatrix} J_{i,B_i}^{linear} & J_{i,B_i}^{angular} \end{pmatrix} = \begin{pmatrix} 0 & \vec{a}_i \end{pmatrix} \\ \dot{J}_i \cdot \vec{v} &= (\vec{\omega}_{A_i} \times \vec{a}_i) \cdot (\vec{\omega}_{B_i} - \vec{\omega}_{A_i}) \\ m_i &= 1. \end{aligned} \tag{7.5}$$

[8]To avoid problems associated with the constraint parametrization in terms of Euler angles $\vec{\theta}$.

[9]With respect to a certain precision.

[10]It is relied upon the constraint stabilization. States where angle limits are violated are not flagged as invalid.

Analogously is proceeded in the case of linear axis, displacements along the axis and linear velocities when the axis is attached to the center of mass of the first body and the displacement of the second body's center of mass from the first body's center of mass along the axis direction (relative to the initial displacement) is measured. The only difference is that linear velocities are constrained instead of angular velocities and displacement errors are measured instead of angle errors. We obtain

$$\begin{aligned} J_{i,A_i} &= \left(J_{i,A_i}^{linear} \quad J_{i,A_i}^{angular} \right) = \left(-\vec{a}_i \quad 0 \right) \\ J_{i,B_i} &= \left(J_{i,B_i}^{linear} \quad J_{i,B_i}^{angular} \right) = \left(\vec{a}_i \quad 0 \right) \\ \dot{J}_i \cdot \vec{v} &= (\vec{\omega}_{A_i} \times \vec{a}_i) \cdot (\dot{\vec{x}}_{B_i} - \dot{\vec{x}}_{A_i}) \\ m_i &= 1. \end{aligned} \tag{7.6}$$

MotorJoint, based on the concepts of **PoweredAxis**, allows to directly control desired relative linear and angular velocities, displacements and orientations of the constrained bodies — the kinematic properties of the rigid bodies — and hence allows to control bodies *kinematically in terms of dynamics.* Constraint solver computes appropriate forces to obey the requested behavior and constrained bodies still properly respond to impacts and other forces. Moreover, imposed constraints can be made soft (by setting appropriate softness constants) and the resulting motion thus affected by other bodies and forces, which is often desirable and produces interesting results.

Chapter 8

Figure Library

The implementation of a figure library that would allow to animate human-like figures in terms of constrained rigid body dynamics and demonstrate the capabilities of the simulator was our another goal. The library is implemented as an extension to the simulator library, built on top of the simulator. It allows to create human-like articulated figures from their skeletal descriptions and control them on the level of both dynamics and kinematics, where constraint forces are computed to attain the desired kinematic behavior.

The dynamics approach allows to adapt the figure motion to external influences and easily solve many problems that are hard to deal with on the kinematics level. Experiments described in the next chapter present how kinematics-based animation (such as the animation taken from motion capture) can be replayed, edited, modified and retargetted to other figures with different body proportions and mass properties using the dynamics-based model.

Since the library supports the animation of loop-free articulated figures only, there are no cycles in the body-joint graph (except for contact joints) and **FastLagrangeMultiplierSolver** computes the dynamics of articulated figures in the linear time[1] (with an extra time to handle contact and joint angle limits). There are alternative methods to simulate articulated figures with the same asymptotic computational cost. These methods are based on the reduced coordinates paradigm, where constraints abstracting the figure joints (corresponding to our primary joints) are incorporated directly to the figure's equations of motion, formulated in terms of the unconstrained figure DOFs and solved in the linear time, such as in [17]. Extra constraints (corresponding to our auxiliary joints) due to contact and joint angle limits are handled in terms of maximal coordinates, analogously to the method pursued by **FastLagrangeMultiplierSolver**.

In this chapter we will informally describe the features of the figure library and outline its implementation. The library exploits services of our simulator, but could be easily ported to other rigid body simulators that offer a similar set of features. As usually, a more detailed description of the library can be found in the Reference Guide.

8.1 Overview

The library's concern is to animate articulated human-like figures in terms of constrained rigid body dynamics. For the purposes of these notes we will assume that *articulated figures* consist of several *segments*, represented by rigid bodies (and their collision geometries), connected to each other by joints in such a way so that segment loops are avoided — the figure structure and segment branchings are represented by joint & bone hierarchy trees, called the figure *skeletons.*

[1]With respect to the number of figure segments.

Skeletons consist of vertices called the joints (corresponding to simulator joints) and edges called the bones (corresponding to figure segments). The library models the motion of figure segments ("skeletal motion", animation of skeletons), considering the associated collision geometries, their mass properties and forces acting upon them. Traditionally, the animation of figure skin ("skeletal animation", the visualization of figures according to postures of their skeletons) is separated from the animation of skeletons and is not of our concern[2].

Since we are not primarily interested in the simulation of articulated figures whose behavior depends on the external influences only (passive motion), articulated figures are often controlled to express a certain kind of active motion. The difference is much like the difference between a real-world passive *rag-doll* and a *puppet* whose limbs are attached to springs controlled by an actor. The idea of puppets is abstracted by the concept of figure *controllers* that are attached to figures to control their joint angles, joint trajectories, etc.

The library can be characterized by the following set of basic features:

- Articulated figures are defined by skeletons. Skeleton bones correspond to (parts of) figure segments and have volumes and collision geometries associated with them. Skeleton joints attach (anchor) figure segments together and determine the types of anchors used to connect them.

- Figure segments are handled as general rigid bodies responding to contact (impacts, friction) and other constraints. Other bodies can attach to figure segments[3].

- Figure motion is controlled by a hierarchy of controllers. Lower level controllers are associated with figure segments and control the limits, angles and angular velocities about axes attached to joints or the trajectories of "markers" attached to segments. Higher level controllers control figures globally, using lower level controllers to perform the specific task.

8.2 Skeletons

The branching structures, initial postures, collision geometries and mass properties of the segments of articulated figures are abstracted by the instances of **Skeleton** class, called the skeletons. Skeletons describe joint & bone hierarchy trees, whose vertices are called the joints, oriented edges the bones and the root vertices the root joints, and an associated supplementary information — anchor types used to attach figure segments, collision geometries (and their mass properties) attached to bones or figure segments and bone orientations and lengths determining initial postures. Actual instantiated figures managed by the figure library are represented by the instances of **ArticulatedFigure** class.

Each joint (except for joints with no outgoing bones) defines a rigid body segment associated with that joint and *all bones outgoing from that joint* — the bones and the joint are rigidly attached to the associated segment. End points of the outgoing bones and the position of the joint itself are called the *sites*, which define the places where other segments can connect to this segment, through corresponding joints. If there is no segment attached to a site at an outgoing bone then it is called the *end site* or zero DOF joint. Root joint is attached to a built-in "world" body (with no collision geometry), which allows to translate and rotate the whole figure by controlling the root joint. The position and orientation of the root segment (given by the root

[2]The demo application simply draws corresponding collision geometries to visualize figures.

[3]Such as an umbrella or a basket attached to the figure's hand in the demo.

joint) is called the position and orientation of the corresponding articulated figure[4].

To make things more clear, let us suppose that we have a skeleton whose joints are denoted by $UPPER\_LETTERS$, $HIPS$ is the root joint and $A \rightarrow B$ denotes bones outgoing from A and ingoing to B, eg.

$$HIPS \rightarrow SPINE \rightarrow CLAVICLE \rightarrow \ldots$$
$$HIPS \rightarrow LEFT\_UPPER\_LEG \rightarrow \ldots$$
$$HIPS \rightarrow RIGHT\_UPPER\_LEG \rightarrow \ldots$$

This portion of the skeleton defines two figure segments A and B consisting of the following bones, $A = \{\ \underline{HIPS} \rightarrow SPINE, \underline{HIPS} \rightarrow LEFT\_UPPER\_LEG, \underline{HIPS} \rightarrow RIGHT\_UPPER\_LEG\}$ and $B = \{\ \underline{SPINE} \rightarrow CLAVICLE\}$, where A is associated with the $\underline{HIPS}$ joint and defines $HIPS$, $SPINE$, $LEFT\_UPPER\_LEG$, $RIGHT\_UPPER\_LEG$ sites and B is associated with the $\underline{SPINE}$ joint and defines $SPINE$, $CLAVICLE$ sites.

Collision geometries of figure segments are defined either as unions of collision geometries associated with the individual bones, outgoing from the corresponding joints the segments corresponds to, or explicitly by attaching explicit collision geometries (with their mass properties) to joints. Segments at initial postures are by default oriented so that body space "up" vectors point in the directions of the incoming bones.

Joints impose various constraints on the attached segments and thus remove their DOFs. While the root joint (implemented by **MotorJoint**) removes from 0 to 6 DOFs, regular joints (implemented by the subclasses of **PointToPoint**) always remove at least 3 DOFs and end sites remove exactly 0 DOFs and have no direct representation in the simulator.

The instances of **Skeleton** class represent joint & bone hierarchies, whose joint structures (**Skeleton::Joint**) store the information related to the skeleton joints and the associated segments, their anchoring to parent segments and the definitions of the incoming bones. The instances of **ArticulatedFigure** class represent articulated figures created from their skeletons.

8.3 Controllers

One of the design goals of the figure library is to allow to control the motion of articulated figures hierarchically so that lower level controllers would implement "muscles" and directly control figure joints or associated segments while higher level controllers would implement the "brain" and control the figure motion globally, using lower level controllers to attain the desired motion expression. There should be an option to control each joint by a different type of lower level controller or do not control a particular joint at all and let the corresponding segment react to external influences only. These ideas are abstracted by **RawController** and **TaskController** base classes.

TaskController is a base class for a special kind of n-ary force object that actuates segments of an attached articulated figure. Given a particular task (figure behavior, desired motion), the figure's degrees of freedom are affected by the controller to fulfill the motion requirements. This is done by applying appropriate external forces and/or imposing various constraints on the figure segments. Task controllers usually do not control the figure segments directly but delegate the burden upon lower level controllers attached to figure joints (segments). This approach supports modularity of the library and allows to replace lower level controllers without the need for affecting the task controllers.

[4] According to this interpretation of the root joint and the root segment, it is advantageous to build skeletons in such a way so that root segments represent figure "hips".

The instances of **TaskController** are registered into simulation as force objects acting or depending on the segments of the linked articulated figures, provide the infrastructure for binding (attaching) lower level controllers to figure joints and implement support for controller serialization. We say that the figure joint (segment) is *actuated* if there is a lower level controller attached to it.

The concept of lower level (raw) controllers is abstracted by **RawController** base class. Lower level controllers affect individual joints or segments and implement various joint "motors" or "springs" pulling figure segments to requested positions and orientations and provide other means to control the figure at the lower level. Lower level controllers are used, managed and owned by task controllers linked to the corresponding figures. There are several kinds of lower level controllers providing different ways to control figure joints and segments,

- Controllers that track angles about axes attached to actuated joints.

  AngleTrackingController is a base class for angle tracking (angular) controllers that control angular velocities, angles and their limits about axes of actuated joints. Given a desired angular velocity or an angle about a joint's axis, the controller generates a motor torque to reach the velocity or angle.

- Controllers that control figure positions and orientations.

  RootJointController is a base class for root joint controllers that allow to control both linear and angular velocities, position and orientation of the root joint's segment or displacements and angles (together with the corresponding limits) along/about the joint axes. Given a desired linear velocity or displacement of the root segment's center of mass along a specified axis, the controller generates a motor force to reach the velocity or displacement. Similarly, given a desired angular velocity or an angle at the root joint about its axis, the controller generates a motor torque to reach the velocity or angle.

- Controllers that control trajectories of selected points on actuated segments.

  TrajectoryController is a base class for a special kind of controllers that control desired trajectories (world space positions) of specified sites (called the *markers*) attached to rigid body segments associated with actuated joints. Markers attached to a rigid body segment together with their desired positions in the world space (called the *virtual markers*) define the desired segment transformation (= position and orientation).

  Each segment can have at most 3 markers (more markers would be redundant). The desired marker's trajectories should be defined so that they yield a general rigid transformation in the case of the root segment and a rotation in the case of other segments.

  Trajectory controllers can be used to control figures by raw (translational) optical motion capture data and to transform the data to skeletal motion data (to be replayed by **AngleTrackingController** and **RootJointController**) by controlling the figure by the raw data and sampling its posture over time ("motion capture data to skeletal motion mapping"). An approach based on penalty forces is pursued by [25] to map the raw data to skeletal motion.

ConstraintBasedAngleController implements an angle tracking controller by imposing constraints on the actuated segments and their parents, utilizing the joint angle limit & motor support built into the simulator (**PoweredAxis**). Similarly, **ConstraintBasedRootController** implements a root joint controller in terms of **MotorJoint**. **ConstraintBasedTrajectoryController** implements a trajectory controller by imposing **PointToPoint** constraints to align the positions of markers on the rigid body segments to the positions of virtual markers. Obviously these constraints must be made soft, because the desired trajectories might suffer from noise and hard constraints would likely be unfeasible.

RagDollController is a task controller that attempts to keep angular velocities or angles at zeroes at actuated joints to simulate joint stiffness or attraction to an initial (comfortable) posture. The actual work is performed by attached angle tracking controllers, programmed by the task controller. **AnimationController** is an interpolation-driven task controller that sets desired joint angles, positions of the root joint site or segment markers according to rotational and translational data that is read and interpolated from associated animation sequence(s). There can be multiple animations bound to the controller. The controller blends the ending and starting portions of the animations that come after each other so that the transitions would be smooth. The actual work is performed by attached angle, root or trajectory controllers. When the control error is exceeded (for example when a walking figure is shot and the impact causes the figure to deviate from the desired posture), joint segments can be unpowered and controller stopped.

The controlling strategies taken by all implemented lower level controllers are based on rigid body constraints. This means that the controllers impose additional (soft) constraints to control the figure joints and segments. Such an approach has a great advantage over penalty-based methods (which are nevertheless widely used in the practice, such as in [18], [24], [25]) because the controller results are predictable, guaranteed, the parameter tuning is trivial (one can get along with the same set of parameters for very different types of scenes and figure interactions[5]) and yet external influences can affect the controller results (again, in a predictable way). On the other side, constraint based approach requires a greater computational power, because the constraints are handled globally and constraint force obeying the constraints must be computed at each time step. Moreover, all figure DOFs might actually be constrained (for example when the figure is told to follow a kinematic animation) and the problems to be solved to compute the motor forces might be really *large*[6]. Luckily, **FastLagrangeMultiplierSolver** is at service and, assuming that all constraints are equalities, its running time is still linear and the extra computational cost required to control the figure by constraints negligible (we would still need at least a linear time to simulate the same figure with no controls). Kokkevis [17] controls the motion of articulated figures also by constraints (with the same asymptotic computational cost) but the figures are simulated in terms of reduced coordinates (the equations of motion incorporate the effects of joints due to segment branching).

8.4 External Influences

Probably the most important reason for animating articulated figures on the dynamics level is the need for adapting the figure motion to external influences (incorporate the influences) or to simplify the animator's work by providing approximate inputs to the figure controllers and let the dynamics system control the subtleties of the resulting animation, considering joint stiffness or attraction to comfortable postures. For example, one might want to animate human arms explicitly and let the dynamics system generate the animation of loose hands automatically. When human-like figures are animated, the animator usually follows a “top-to-bottom” paradigm, so that those parts of the body, that are important for the overall impression of the resulting motion and that characterize the motion globally, are animated first (such as the torso) and the animation

[5]See sections (3.4) and (6.6) for more information on the comparison of penalty forces and soft constraints.

[6]The simulator's test scenes exercise an articulated structure consisting of 127 segments whose all DOFs are constrained.

subtleties are considered the last (such as the animation of the hands and head). Systems working on the dynamics level allow to automatically handle most bottom level animation tasks, that are laborious to handle on the kinematics level, such as fixing the head to prevent it from tilting and leaning or make the hands swing naturally in accordance to the animation of arms, prevent feet from sliding, etc. All these tasks can be handled by imposing *soft unbounded* or *soft bounded equality constraints*, abstracted by lower level controllers or the simulator library[7].

While unbounded equality constraints do not limit the magnitudes of the restoring constraint forces due to the constraints, bounded equality constraints impose limits on the constraint DOF multipliers (limiting the constraint force magnitude) and instruct the constraint solver to "work as hard as possible to enforce the constraint, but allow the constraint to be violated when it can not be satisfied". This is a key distinction. For example if we position an articulated figure to the front of a wall and tell it to walk against the wall, following a kinematic animation by imposing unbounded equality constraints and actuating the figure's root site against the wall, then the non-penetration constraints due to the wall and the contacting figure segments will "fight" with the animation constraints and there will be no feasible solution — no constraint force will be computed and all constraints will break (non-penetration will not be enforced, figure limbs will be torn off the torso, etc).

The remedy is to impose bounded equality constraints with finite $\vec{\lambda}^{lo}$ and $\vec{\lambda}^{hi}$ limits instead of unbounded constraints *where appropriate*. In the case of motion control, where lower level controllers formulate constraints to affect the figure motion, our controllers provide an option to impose bounded equality constraints instead of unbounded equality constraints, allowing the constraint solver to limit the constraint force (motor force) magnitude. Recalling the example with a walking figure, if the root joint's motor force was limited, the non-penetration constraints due to the wall would "win" and so the solution would exist[8]. However there is one thing that must be kept in mind — bounded equality constraints can not be primary and so are expensive to handle.

[7]These constraints are described in sections (6.6), (7.4.2) and (6.5.3).

[8]The mechanism of dragging rigid bodies by the mouse cursor in the demo application is implemented by imposing bounded equality constraints as well. This ensures that constraints are feasible even when the body is dragged against a fixed body.

Chapter 9

Results

In this chapter we will present results of the thesis, capabilities of the implemented libraries and their application to animation of human-like figures.

One of the goals of this thesis was to get the reader acquainted with the parts of the constrained rigid body dynamics theory necessary to implement a functional dynamics simulator. Although there are many worthwhile materials and papers devoted to certain parts of the theory publicly available, it is relatively hard to find a complete, yet deep enough, material that would cover all that is needed to implement a practical dynamics simulator incorporating the usual set of features demanded in the practice. As a result, one either comes upon introductory, simplified and incomplete materials or has to break through many different (often involved) papers that use mutually inconsistent notations and pursue certain aspects of the theory too thoroughly for one's specific needs. To address this issue, we have put the available materials together, with the effort to allow readers, who are totally unaware of the topic, to understand the theory without the need for studying other materials. No prior knowledge of the topic is required to read this thesis, however basic knowledge of calculus and linear algebra are needed.

Our effort was mostly concentrated at the implementation of an efficient constrained rigid body simulator that would utilize the most recent algorithms published by authorities in this area of computer graphics, allowing it to simulate highly constrained rigid body systems at interactive rates and making it competitive with other challenging systems at the market. We have focused at the design, modularity, extensibility, the level of concept abstraction and the quality of implementation of the simulator. The simulator implementation and concept abstraction are mostly based on the notes presented in the previous chapters, but certain concepts had to be implemented differently and more sophisticated algorithms used to achieve an interactive performance. Despite of these facts, which can be classified as implementation details, these notes should be sufficient to comprehend the theory and implement a functional simulator. The implementation details and simulator internals are described in the Reference Guide and should be easy to understand after having finished reading these notes.

An extra attention was paid to the abstraction of equations of motion formulated as ODEs. The simulator provides a generic infrastructure for solving ODEs and implements various ODE solvers that are used to solve ("integrate") the equations of motions. Besides the fact that the integration method is not hard-coded into the simulator and can be replaced without the need for affecting the rest of the system, the ODE infrastructure allows to incorporate various adaptive time-stepping strategies to control the integration precision or avoid invalid states (enforce body non-penetration, rollback on body penetration) for free. This approach was pursued even further. The simulator separates rigid bodies to multiple simulation groups so that each group would have its own equations of motion, handled independently on the other simulation groups, which results in the reduction of the total computational cost. Each simulation group's equations of motion are integrated using an own ODE solver with a different time step, adapted to the states of bodies in the simulation group. The simulator ensures that simulation groups are merged whenever required, ensuring the simulation consistency (bodies that might interact are always simulated in terms of a common simulation group, individual ODE solvers due to simulation groups are advanced in a controlled way, making sure that all simulation groups end up at the same time and allowing to rollback the whole simulation state when a certain ODE solver can not advance to a desired time or a request to merge simulation groups is detected while advancing the simulation). As far as we know, this is a unique feature, not implemented in other publicly available (free) simulators, yet very useful as it allows to "skip" simulation groups consisting of "idle" bodies when advancing the simulation state.

The other simulator component we value the most is the global constraint solver. Simulator library user specifies various constraints on rigid bodies that have to be met (geometric constraints that implement virtual joints, hinges, pins, joint angle limits, velocity constraints, etc) and the constraint solver computes appropriate constraint forces or impulses ensuring that the constrained bodies will obey both the constraints and the Newton laws. Apart from explicit (usually permanent) constraints, the simulator imposes implicit short-lived constraints to enforce body non-penetration and implement friction at contacts. Constraints are formulated as position-level, velocity-level or acceleration-level equations, either as equalities or inequalities, and implemented on the velocity or acceleration levels. The implementation of a Lagrange multiplier based solver capable of handling highly constrained articulated structures, whose equations of motion are solved in the linear time, is provided, offering an efficient alternative to reduced coordinates approaches used to simulate loop-free articulated structures.

Various test scenes exercising certain simulator features and testing the performance capabilities, precision and robustness of the simulator and its components are implemented. Moreover, library self-tests are provided to assure that the library works as expected and does not break during its development. The tests can be run by the user to see how well the simulator performs, which we find more valuable than presenting the results of stark measurements.

To manifest the performance capabilities and the fitness of the simulator to handle "serious" problems, the demo application presenting the features of the figure library and serving as a "front end" to the test library is implemented. Demonstration scenes show how figure motion, driven by translational or rotational motion capture data, can be adapted to external influences, edited or retargetted to different characters in terms of rigid body dynamics and how easily certain classes of problems, that are hard to deal with on the kinematics level, can be solved using a physical model.

9.1 Tests

The purpose of the test library is to provide appealing demonstration scenes that exhibit certain simulator features, realize various performance, stability or simulation/integration precision experiments or simply test the integrity of the simulator components. Depending on the type of the functionality to be tested and whether the tests are batch or interactive, the tests are separated to several test categories.

Self-tests ("regression tests", "unit tests") are batch tests that invoke simulator services and verify that the service responses match the expected results. Due to the nature of these tests, self-tests mostly test various numerical algorithms or those services where the verification of the response was easy to implement. The most important self-tests (that proved to be extremely useful during the development of the simulator library) are the tests that exercise linear algebra components, collision geometry detection and LCP solvers.

Interactive tests examine the simulator features by providing test scenes that are visualized by the scene visualizer. The simulation state is presented to the user by drawing the collision geometries of the rigid bodies, the scene is made of, and it is relied upon the user's perception to validate the test. Interactive tests and demonstration scenes from the demo application are visualized by a generic scene visualizer, implemented by the **Drawing** library[1], that allows to observe the scene from various positions and angles, inspect rigid body states and *affect the bodies* in the scene, either by dragging/rotating existing bodies by mouse or firing/dropping new bodies from the "cannon" attached to the observer. Although this way of specifying external influences is very simple, it is extremely powerful and capable of providing interesting results, demonstrating the features of the simulator and the figure library quite nicely. Some tests actually "do nothing" by themselves and rely on the user input to exercise the features to be tested. We will now describe the implemented tests[2].

9.1.1 Contact Tests

These tests exercise the simulator behavior with respect to body separation to simulation groups, the contact and friction model and the enforcement of non-penetration,

- **TestGroupMerging**

  The test verifies that simulation groups are merged properly when bodies from different simulation groups approach each other and that the simulation transaction is aborted correctly when the request to merge simulation groups is submitted.

- **TestSimultaneousImpacts**

  This is the most elementary non-penetration test involving multiple contacts that verifies that aligned cubes will not begin to spin when they strike each other (ie. when a cube is dropped flat on a table, the impact should not produce any torque).

- **TestContactStabilization**

  The purpose of this test is to demonstrate how non-penetration constraints are stabilized. A stack of boxes is created and the boxes are positioned so that they penetrate a little at the initial state but the state is still valid. Constraint forces exerted at the contacts will repulse the bodies and increase local separation distances at the contacts, thus preventing further penetration and eventually separating the bodies. Note that non-penetration is enforced by exerting constraint forces and impulses only that are computed globally with respect to all contacts and other constraints. This guarantees that the separation distance at contacts will actually increase if requested. Positions and orientations of the affected bodies are *never modified directly.*

- **TestCriticalContacts**

  Contacts whose separation distance drops below a preset limit are called the critical contacts. In order to ensure that penetration will be avoided, constraint solver stops bodies involved in critical contacts if it fails to establish non-penetration constraints at those contacts. This feature is exercised by this test, where a cube is being crushed by two fixed

[1] `OpenGL` library is used to do the rendering.

[2] The tests are integrated by the Euler solver unless stated otherwise.

desks and non-penetration constraints due to the contacts between the cube and the desks can not be satisfied.

- **TestFriction**

  Static, dynamic and impulsive friction approximation at contacts is evaluated by this test. The scene consists of a table, desk and a stack of boxes. The boxes are stacked on the desk that lies on the table and there is no friction between the desk and the table. A ball strikes the desk and the impulsive friction between the ball and the desk makes the desk (as well as the stacked boxes) move. The whole desk can be dragged by just pushing onto the top-most box.

9.1.2 Joint Tests

Simulator library implements various geometric joints that can be used to attach bodies together and provides a way to control angles or velocities about joint axes, impose joint angle limits, control displacements along joint axes, etc. This functionality is presented by **TestHingeJoint**, **TestUniversalJoint**, **TestBallAndSocketJoint**, **TestSliderJoint**, **TestMotorJoint** and **TestLimitedSpringJoint** tests that evaluate the corresponding joint classes. It is assumed here that the user provides appropriate external influences by dragging the attached bodies to validate the functionality of the joints and their motors.

9.1.3 Performance and Robustness Tests

To test the performance of the constraint solver and its fitness to handle highly constrained articulated structures whose segments can collide, the following two tests exercising certain types of articulated structures were implemented.

- **TestChain**

  A chain consisting of 32 segments connected by various joint types, where each joint removes at least 3 DOFs, corresponding to a linear articulated structure is presented by this test.

- **TestArticulatedStructure**

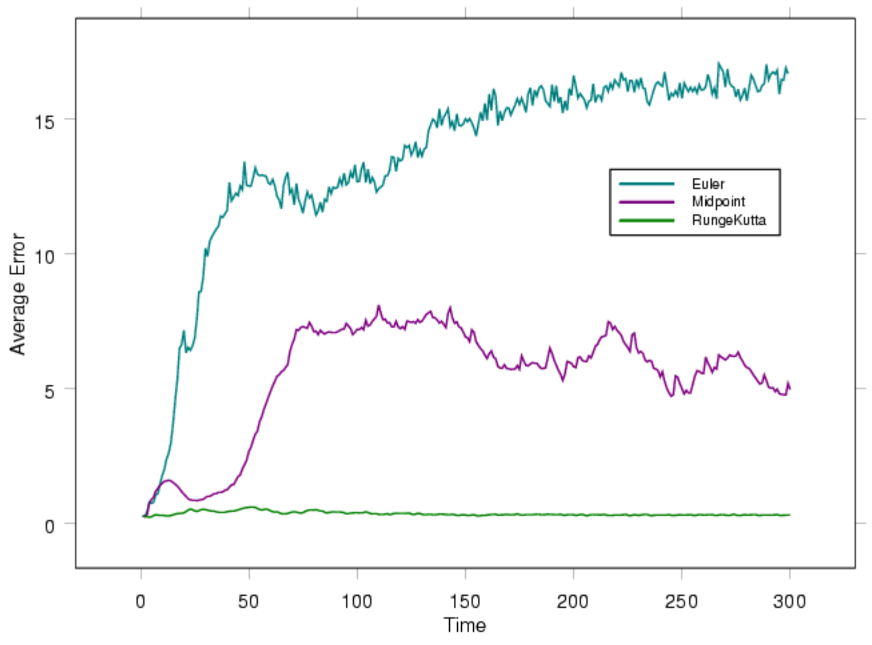
  This test implements and evaluates a non-linear highly articulated structure consisting of 127 segments whose all DOFs are constrained and presents the use of soft constraints to attract articulated structures to initial states — soft constraints are used to constrain joint angles to zeroes. Unless the structure's segments contact, the constraint forces are computed in the linear time.

TestJointCycles test verifies that articulated structures with loops (rigid body systems with cycles in the body-joint graph and other auxiliary constraints) are handled correctly by the linear time constraint solver and that the LCP solver can solve problems whose matrices A are not positive-definite. The test also demonstrates the handling of redundant constraints, because more constraints are imposed than is actually needed to achieve the effect.

9.1.4 Integration Precision Tests

These tests simulate the same test scene by integrating the equations of motion using a set of distinct ODE solvers so that the simulation effects (integration precision of the individual solvers) could be compared. An articulated structure consisting of a rotating carousel with four chains

Figure 9.1: The illustration of the average constraint error at the carousel links. Different ODE solvers were used to integrate the equations of motion, the values of $\|\vec{C}_p^i\|$ due to **PointToPoint** constraints at the links measured each 0.1 seconds (= the time axis unit) and the average values plotted.



attached to its construction is chosen to demonstrate the effects, constraint errors at the chain links measure the integration precision (the precision can be compared visually quite easily — the chain links either hold or the chain is broken). To stress the precision differences attained by the solvers, all position-level constraints $\vec{C}_p = \vec{0}$ are implemented on the acceleration level only[3], an integration error creeps into both $\dot{\vec{C}}_p$ and $\vec{C}_p$ and it is nice to see how well the particular ODE solver performs to keep $\vec{C}_p$ and $\dot{\vec{C}}_p$ at zero vectors.

The tests are implemented by **TestIntegrationPrecisionEuler**, **TestIntegrationPrecisionMidpoint** and **TestIntegrationPrecisionRungeKutta** that use the corresponding ODE solvers (Euler, Midpoint, Runge-Kutta) to integrate the equations of motion. As can be seen by running the tests or looking at figure (9.1), Euler solver fails miserably. However when the constraints are implemented directly on the *velocity level* (contrary to the other tests), the method performs quite well — yet it is no wonder, because velocity-level constraints affect the states of constrained bodies directly, independently on the used integration method. In that case the only external force integrated by the Euler solver is the force due to gravity and so there is nothing to mess up. This is demonstrated by **TestIntegrationPrecisionEulerVelocityLevel**.

Since the maximum time step sizes are currently determined by the upper velocity bounds of the bodies in the corresponding simulation groups (see page 79), the step sizes are often too limited, higher-order integration methods can not manifest their potential (the ability to take relatively large time steps with relatively small integration error) and the extra computational cost required to take a step is not balanced by the step size. That said, Euler solver is currently

[3]By default, these constraints would be maintained on the velocity level at regular states (to stabilize the velocity level errors) and on the acceleration level otherwise.

Figure 9.2: For an illustration, the time spent in various simulator components or performing specific tasks was measured. The results provide an overview of the overall simulator performance, but the measured data that should be interpreted from a relaxed detached view, because the running times mostly depend on what is actually happing in the scene, whether the user interacts with the bodies, etc. The "Contact / Constraints" column measures the time spent performing collision detection relative to the time spent solving constraints and "Restarts / Advances" (restart overhead) shows how much time relative to the total time spent advancing the simulation was required to restart ODE solvers during the course of simulation. Obviously, the restart overhead is greater at those tests where simulation groups merge and dissolve quite "often", as opposed to other tests, where the body separation to simulation groups is nearly static.

Test	Contact / Constraints	Restarts / Advances
TestChain	23.3506 %	0.937988 %
TestFallingObjects	102.511 %	3.04259 %
TestFriction	23.1678 %	0.747662 %
TestVehicle	20.3607 %	0.587703 %

the fastest but the least accurate integration method implemented in the simulator. Despite of its inaccuracy, it is used as the default method.

9.1.5 Global Tests

To test the simulator features globally, exercise the cooperation of the simulator components, rigid body interaction, collision detection reliability and the validity of the friction model more thoroughly, several tests were implemented and the running times summarized in the figure 9.2.

- **TestFallingObjects**

  A bunch of objects of various shapes is dropped to a pool and the collision detection reliability and constraint solver precision are tested.

- **TestVehicle**

  This tests presents a model of a simple terrain vehicle that can be controlled by the user. The vehicle consists of a chassis, body and wheels attached to the chassis by constraints implementing the wheel suspension. Motor constraints are imposed to implement steering and actuate the vehicle. A carriage is attached to the vehicle, to make the test more interesting. The test focuses at the evaluation of the friction model, collision detection and the constraint solver performance.

9.1.6 Figure Library Tests

These tests briefly validate the functionality of the figure library — skeletons, lower level controllers (**TestAngleTrackingController**, **TestTrajectoryController**), higher level controllers (**TestAnimationController**, **TestRagDollController**) and various helper/auxiliary classes (**TestBVHLoader**). To support the demo application, articulated figures and controllers are serializeable and can be externalized. This is tested by **TestExternalizedFigure**.

To avoid the creation of a ton of specialized hard-coded tests, whose parameters can not be tuned without recompiling the relevant tests, the demo application was created. The application implements a powerful scene loader that allows to create “scenes” (consisting of static geometry, rigid bodies, force objects, articulated figures and figure controllers) from their textual descriptions stored in files. The demo scenes, presented in the next section, test and demonstrate the features of the figure library more thoroughly.

9.2 Demo Application

The demo application is the simulator and the figure library “front end”. It allows to run tests implemented by the test library (described in the previous section), load and visualize demonstration scenes stored in external files and run the figure tool.

The purpose of the figure tool is to help the animator set up figures by allowing to “import” skeletons and animations from popular file formats used by common animation tools and output files that can be read by the figure library. These files are actually the serializations of the corresponding figure library objects and their editing corresponds to setting the figures up. The figure tool was used to extract the skeletons and animations from motion capture data stored in standard `BVH` files[4] and preset figure controllers in the demonstration scenes.

Since the parsers used by the demo and the figure library suggest the lists of valid tokens whenever an invalid token is encountered while reading a file and the figure library files “mirror” the structures of the library objects (documented in the Reference Guide), we will not bother describing the file structures. In case the demonstration files had to be modified in a non-trivial way, a deeper understanding of the figure library would have been required anyway.

Demonstration scenes present the animation of articulated human-like figures. The same visualizer, used by the test library, is used to visualize these scenes. This is good because users can interact with the figures (generate external influences) and observe how the figure motion adapts to the influence. For example one might pull or rotate figure segments by mouse, strike figures by firing new bodies from the attached “cannon”, create obstacles that prevent figures from moving in desired directions, affect other bodies attached to figures (such as a basket or an umbrella), etc. We will now describe the individual scenes and point out the features of the simulator and the figure library that the particular scenes rely upon.

9.2.1 Kinematic Animation

Scene `scenes/animation.txt` presents a figure that is controlled pure kinematically, by a data from motion capture. The figure consists of 18 segments, where legs are represented by 2×4 segments, arms by 2×3 segments, the torso by 3 segments and the head by 1 segment. The branching structure of the figure, collision geometries associated with the figure segments and their mass properties are stored in the relevant *skeleton* file.

Animation controller attached to the figure, and whose state is stored in the corresponding *controller* file, controls the figure joint DOFs according to angles and displacements stored in the associated animation sequence, read from an associated *animation sequence file*. Angle tracking and root controllers are used to drive the figure joints by the data from the animation sequence and certain joint axes are made soft (soft constraints are imposed by angle tracking controllers) so that the figure segments can be pulled by mouse.

[4]The format is described in [20].

The produced animation should resemble the original kinematic animation as much as possible, except for the animation of the torso and arms that are set up so that they can be dragged by mouse and affected by other external influences.

The figure skeleton and controller files are stored in `controllers/joint_dof_animation` directory and the animation to be replayed read from `animations/joints_dofs`, as requested by the controller. To compare the actual performance with the original `BVH` animation stored in `animations/bvh`, one might run the *BVH visualizer* that is a part of the demo. To replay a different animation, `controllers/joint_dof_animation/controller.txt` must be edited appropriately.

Another simulation effects can be achieved by adjusting the controller parameters, modifying the figure's skeleton and collision geometries or attaching other bodies to the segments. The other examples we are going to present were mostly prepared using this technique.

9.2.2 Animation With Loose Arms

The most simplistic modification of the previous example is to make the arms loose and let the physical system animate the arms by itself — figure DOFs due to the torso and legs are controlled by motion capture data while DOFs due to the arms are made stiff.

This is presented in `scenes/animation_with_loose_arms.txt`. The corresponding controller controls joint DOFs according to the angles and displacements stored in the animation sequence, but, unlike `controllers/joint_dof_animation` controller, the arms are loose and do not follow the angles stored in the animation sequence. Angle tracking controller is set up so that shoulders are limited and tend to get into the initial posture (soft constraints to make joint angles zero are imposed), elbows are limited and stiff (soft constraints to prevent rotation about joint axes are imposed) and wrists follow the initial posture. The skeleton and controller files are stored in `controllers/joint_dof_animation_loose_arms`.

9.2.3 Animation With Limited Forces

Another trivial modification of the original `controllers/joint_dof_animation` controller is to limit the magnitudes of motor forces so that the figure could interact with the scene and infeasible constraints would be avoided. Since the motor forces are limited, joint angle limits have to be provided to prevent limbs from dislocating.

Scene `scenes/animation_with_limited_forces.txt` presents a figure whose DOFs are controlled by an animation controller capable of producing limited forces (ie. arm won't bend if there is an obstacle). The figure switches to a simplified "rag-doll mode" when it is hit to the chest (joints will no longer be powered but will remain limited) and the figure feet properly bend when they are in contact with the surface.

To demonstrate more complex interactions with the obstacles in the scene, we have prepared another scene `scenes/animation_with_obstacles.txt` that puts the figure into a pool of moveable bodies. Limited motor forces ensure that the animation constraints can be maintained reliably.

9.2.4 Editing Animation

Kinematic animation can be easily edited on the dynamics level by making certain constraints soft and imposing additional constraints to adapt the motion as requested. For example one might easily fix head to look towards a desired point, prevent the head from tilting, animate hands by making the joints at the wrists loose, etc.

Scene `scenes/animation_with_pointing_arm.txt` demonstrates how the animation of a human figure can be edited so that the figure's right arm would point in the specified direction. This is implemented by attaching a **MotorJoint** to the figure's hand and imposing a bounded equality constraint that constrains the hand to occupy the specified world space position corresponding to the point the hand should point at.

Another example, implemented by `scenes/animation_with_basket.txt`, attaches a basket to the figure's right hand and lets the animation system adapt the motion. The basket is represented by a general rigid body with mass properties and is attached to the figure's hand by a joint that simulates the friction between the basket and the hand. The figure's controller is adjusted so that the right arm is made more stiff (animation constraints are less soft).

Finally, `scenes/animation_with_umbrella.txt` demonstrates how the animation can be mapped to a figure that holds an umbrella in its right hand. Please notice that the umbrella is always held upright, regardless of the figure posture (ie. even when the figure bows, for example), that the umbrella is held in front of the head and the hand is always correctly oriented with respect to the umbrella handle. This is implemented by three additional constraints. An umbrella is created as a rigid body attached to the figure's hand by an appropriate anchor joint that enforces the proper umbrella versus hand orientation. Another constraint pulls the figure's hand in front of its head and the third constraint demands that the umbrella is held upright.

9.2.5 Retargetting and Combining Animations

The problem of transferring animations that fit a certain figure to another figure (motion retargetting) and the problem of combining motion segments (so that the transition between the segments would be smooth and realistic) are often hard to solve on the kinematics level. With the help of dynamics, the problems can be handled much easily. The reason for this to work is the fact, that the kinematic data (controller inputs) that might be inconsistent with the dynamics of the animated figure are automatically "corrected" by the dynamics system, if the figure controllers are programmed not to follow the (inconsistent) inputs exactly.

Scene `scenes/fat_man_animation.txt` retargets the animation of the default figure to a fat figure with very different body proportions — limbs are much shorter than those of the original figure and the torso is bigger and hunched. When we were preparing this test, we modified the original skeleton, the collision geometries and their mass properties to represent the fat body, but did not have to modify the animation data. The rest of the work is due to the rigid body dynamics.

Scene `scenes/combined_animation.txt` presents how two animation sequences can be combined. The animation controller blends (interpolates) inputs from the two animation sequences while making a transition from one animation to the other one and the dynamics system "corrects" the blended animation if it is inconsistent with the dynamics of the figure. Standard linear interpolation is used to interpolate the translational data and the spherical linear interpolation to interpolate the rotational data.

9.2.6 Optical Motion Capture To Skeletal Motion

Motion capture is used to capture the motion of actors so that it could be replayed on the computer. An actor, equipped with motion capture sensors, performs the desired motion that is captured by the motion capture device. The captured data are then analysed and mapped to skeletal motion so that it could be replayed using kinematics or dynamics based techniques, [19]. In the case of optical motion capture, the motion capture device provides the trajectories of optical motion capture sensors ("markers") that are rigidly attached to the actor's limbs and body. Having known the figure's skeleton, the assignment of markers to the figure segments and their positions on the segments, the task is to map the marker trajectories (translational data) to skeletal motion (angles at joints and displacements of the root joint over time).

The figure posture can be characterized by the state vector $\vec{\theta}$ which contains the position and orientation of the figure and the angles at the figure joints concatenated in a certain order. If we have n markers attached to the figure segments and $\vec{\theta}$ determines the figure posture then $\vec{p}_i(\vec{\theta})$ will denote the world space position of the i-th marker ($1 \leq i \leq n$), which is fully determined by $\vec{\theta}$. If $\vec{q}_i(t)$ is the desired world-space position of the i-th marker at time t, determined by the motion capture device, then the task is to find the figure posture $\vec{\theta}(t)$ so that the mapping error $\sum_{i=1}^{n} \|\vec{p}_i(\vec{\theta}(t)) - \vec{q}_i(t)\|$ is minimal.

There are many approaches to perform the mapping. One can for example employ classical global or local optimization techniques to minimize the mapping error and obtain $\vec{\theta}(t)$, such as in [21]. We take a different approach instead and impose soft **PointToPoint** constraints that demand $\vec{p}_i$ to occupy the same world space position as $\vec{q}_i$ — for each marker i we impose a position-level constraint $\vec{C}_p^i = \vec{p}_i - \vec{q}_i = \vec{0}$ and let the constraint solver compute appropriate constraint forces to drive the markers along the desired trajectories. Since the constraint solver is efficient enough, the mapping of the desired trajectories to skeletal motion can be performed online. The whole process is abstracted and implemented by the **TrajectoryController** and presented by `scenes/animation_by_markers.txt` scene[5]. Although some authors attempt to employ penalty forces to attract markers to their desired positions, as in [25], penalty forces are hard to tune and do not guarantee a result, as opposed to constraints[6].

The advantage of the physical based approach over optimization methods is clear — it can be used to directly incorporate certain external influences, such as friction (which can be used to reduce foot sliding), aid in the process of classifying markers (determine whether the given marker is the marker i, predict the marker positions at the next motion capture frame) and continue in the mapping process even when certain markers are currently "hidden" and their positions can not be tracked, by leaving the markers temporarily unconstrained.

9.2.7 Rag Dolls

Figures in the previous scenes were all controlled by kinematic animations, possibly affected by external influences. To conclude and present something more entertaining, we have prepared a certain class of figures called the rag-dolls. Their controllers impose joint angle limits only and make the joints stiff or follow the initial postures, but do not control the figures in any other way and so are technically "less advanced" than the animation controllers used in the previous scenes. Scene `scenes/hanged_man.txt` presents a rag-doll hanging on a rope and `scenes/staircase.txt` shows a rag-doll falling off the staircase.

[5]Since we did not have an access to raw optical motion capture data, the trajectories were generated from the demonstration `BVH` files and our purposefully thought up marker set, by using the services offered by the figure tool.

[6]The constrained based approach allowed us to use the same constraint parameters (= 2 constants) for all markers without a glitch. More information on constraint-based motion control can be found in section (8.3).

Chapter 10

Conclusions

Rigid body theory necessary for the implementation of a functional constrained rigid body dynamics simulator has been presented. The implementation of our simulator has been outlined, describing the concept abstraction and the implementation overview. The simulator enforces constraints by introducing constraint forces and is capable of solving both equality and inequality constraints, which are used to implement non-penetration constraints with friction. An efficient constraint solver was implemented to simulate loop-free articulated figures in the linear time.

The total computational cost required to simulate rigid body systems has been reduced by separating bodies to multiple simulation groups whose equations of motion are abstracted by generic ODEs and solved independently on each other. Simulation groups are merged whenever required so that the simulation consistency is ensured.

The test library, the figure library and the demo application have been implemented to validate the offered features and show that the libraries can be used to animate articulated human-like figures. An infrastructure for constrained-based motion control has been implemented in the simulator and abstracted by the figure library. Demonstration scenes present how kinematic animation can be replayed, edited, retargetted and combined using a physical model. A constrained-based method to map raw optical motion capture data to skeletal motion in terms of motion control has been shown.

10.1 Future Work

As it is with most programs and libraries, directions of the further development and improvements exist. We would like to outline what features are currently missing and would be nice to have incorporated and what could be implemented or abstracted differently or more efficiently. We keep the following things in mind,

- New & improved collision detection

  New collision detection abstraction layer should be implemented so that different collision detection engines (external libraries) could be used by the simulator, the implementation of collision detection should be more separated from the rest of the simulator. Built-in collision detection should be rewritten to be more efficient, match the new interface and respect the other new requirements described below.

  Sort and sweep algorithm used by the higher-level collision detection does not integrate well with multiple simulation groups. Hierarchical hash spaces might be more appropriate, [13].

The collision detection's collider objects are able to determine whether two geometries penetrate and compute contacts if there is no penetration only. This is okay when all bodies are simulated, but it might cause problems in computer games, when certain bodies are not simulated and need be "activated" upon a request or new bodies need be spawned (the new bodies might penetrate other bodies at the time of activation and the simulator would not be able to advance further). Collision detection should be able to compute "surrogate" contacts, defined as the sites where constraint forces should act, in order to separate already penetrating bodies.

- Aggressive body enabling & disabling

  Simulation groups consisting of "idle" bodies are not simulated. It would be nice to implement a more aggressive method that would allow to automatically disable and enable bodies within a simulation group and ignore the effects of the constraints at the current time step if they involve enabled and automatically disabled bodies. These constraints would request the involved bodies to enable since the next time step and the handling of the constraints would be postponed until then. This is based on the idea implemented by [22].

  In order to support this feature, ODE solver interface would have to be augmented so that it would be able to ignore the ODE components due to the disabled bodies. Collision detection would have to be able to compute "surrogate" contacts in order to separate penetrating bodies, because contacts between automatically disabled bodies and enabled bodies would be ignored until the next time step and it might be too late to avoid penetration then.

- Iterative LCP and constraint solvers

  Currently, all LCP and constraint solvers are analytical. New iterative LCP or constraint solvers should be implemented so that one could trade the simulation precision for the improved performance.

- Estimation of the time of collision

  Simulation time step is limited so that no point on the surface of a rigid body would sweep a trajectory whose length would exceed a preset limit during the duration of the time step. This is too limiting, higher-order integration methods can not manifest their potential. The maximum step size should be estimated by maintaining the pairs of potentially colliding bodies and estimating the time of collisions, such as in [13] or [4].

- More higher level motion control, muscular model

  Additional task controllers should be implemented by the figure library to provide new ways of motion control, such as the ability to generate certain types of parametrized motion (walking, running, balancing). These controllers would be based on the algorithms used by robotics.

  Lower level controllers might want to simulate muscles and tendons so that joint angle limits could be specified as a function of the figure posture and would not be fixed.

Bibliography

[1] Martin Baker. 3D World Simulation. `http://www.euclideanspace.com/`.

[2] David Baraff. Collision Detection That Really Works. Lecture notes slides. `http://www.cs.cmu.edu/~baraff/sigcourse98/cd-slides.pdf`.

[3] David Baraff. Analytical Methods for Dynamic Simulation of Non-penetrating Rigid Bodies. *SIGGRAPH '89, Boston*, pages 223 – 232, July 1989.

[4] David Baraff. Curved Surfaces and Coherence for Non-penetrating Rigid Body Simulation. *SIGGRAPH '90, Dallas*, pages 19 – 28, Auguest 1990.

[5] David Baraff. Coping with Friction for Non-penetrating Rigid Body Simulation. *SIGGRAPH '91, Las Vegas*, pages 31 – 40, July 1991.

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

[7] David Baraff. Fast Contact Force Computation for Nonpenetrating Rigid Bodies. *SIGGRAPH '94, Orlando*, pages 23 – 34, July 1994.

[8] David Baraff. Interactive Simulation of Solid Rigid Bodies. *IEEE Computer Graphics and Applications*, pages 63 – 75, 1995.

[9] David Baraff. Linear-Time Dynamics using Lagrange Multipliers. *SIGGRAPH '96, New Orleans*, pages 137 – 146, August 1996.

[10] David Baraff, Andrew Witkin, and Michael Kass. An Introduction to Physically-Based Modelling. *SIGGRAPH '97*, 1997. Course notes.

[11] David M. Bourg. *Physics for Game Developers.* O'Reilly & Associates, Inc., 2002.

[12] Michael Bradley Cline. Rigid Body Simulation with Contact and Constraints. Master's thesis, The University of British Columbia, 2002.

[13] Kenny Erleben. Rigid Body Simulation, 2002. Lecture notes. `http://www.diku.dk/forskning/image/teaching/Courses/e02/RigidBodySimulation/`.

[14] Michael T. Heath. Scientific Computing: An Introductory Survey, 1997. Lecture Notes. `http://www.cse.uiuc.edu/heath/scicomp/notes/`.

[15] Ladislav Kavan. Simulation of Fencing in Virtual Reality. Master's thesis, Charles University, Prague, 2003.

[16] Katsuaki Kawachi, Hiromasa Suzuki, and Fumihiko Kimura. Simulation of Rigid Body Motion with Impulsive Friction Force. *Proceedings of IEEE International Symposium on Assembly and Task Planning*, pages 182 – 187, August 1997.

[17] Evangelos Kokkevis. Practical Physics for Articulated Characters, 2004.

[18] Evangelos Kokkevis, Dimitri Metaxas, and Norman I. Badler. User-Controlled Physics-Based Animation for Articulated Figures. *IEEE Proceedings of Computer Animation'96 (CA)*, 1996.

[19] Alberto Menache. *Understanding Motion Capture for Computer Animation and Video Games*. Academic Press, 2000.

[20] M. Meredith and S. Maddock. Motion Capture File Formats Explained. `http://www.dcs.shef.ac.uk/~mikem/`.

[21] Marius-Calin Silaghi, Ralf Plänkers, Ronan Boulic, Pascal Fua, and Daniel Thalmann. Local and Global Skeleton Fitting Techniques for Optical Motion Capture. *IFIP CapTech '98*, November 1998.

[22] Russell Smith. Open Dynamics Engine, 2004. `http://www.ode.org/`.

[23] Jeff Trinkle, Jong-Shi Pang, Sandra Sudarsky, and Grace Lo. On Dynamic Multi-Rigid-Body Contact Problems with Coulomb Friction. *Zeitschrift für Angewandte Mathematik und Mechanik*, pages 267 – 279, 1997.

[24] Victor B. Zordan and Jessica K. Hodgins. Tracking and Modifying Upper-body Human Motion Data with Dynamic Simulation. *Computer Animation and Simulation*, 1999.

[25] Victor B. Zordan and Nicholas C. Van Der Horst. Mapping Optical Motion Capture Data to Skeletal Motion Using a Physical Model. *The Eurographics Association*, 2003.