Notes on Discrete Mechanics

M. Servin

April 19, 2011

Contents

Chapter 1

Introduction

This text is a collection of notes intended for students taking the courses *The Physics of Virtual Environments* and *Visual interactive simulation* at UmeåUniversity.

The long-term ambition is to provide a coherent knowledge base for the art of physics-based numerical simulation, with emphasis on the techniques for modeling, computation, algorithms and how to produce and utilize well-designed software to realize powerfull simulations and applications involving time-evolving complex mechanical systems. The existing literature is either domain specific, e.g., CAD, scientific computing, computer games, simulators, with very different formalisms and what appears to be very different methods while in reality there is a continuos exchange of techniques and a common underlying structure closely related to the fundamental structure in the laws of nature and mathematics. In this text, we use a general framework as much as possible and highlight the underlying common structure when presenting specialized methods. This is also a key to find good methods for multiphysics – combination of several distinct physics models, e.g., rigid bodies and fluids, in the very same simulation in a consistent and computationally efficient way. The readers are expected to have a wide and variable range of prior knowledge and skills in the fields of mathematics, computing science, physics and mechanics. Therefore, there is a mix of introductory informal texts with only basic physics and mathematics and more advanced texts.

1.1 Domains of applications

Physics-based numerical simulation has many uses. They all have the common purpose of replacing a physical reality. The reason for this can be to study a phenomenon or process in order to get an increased understanding of it, to test different ideas on how to design or modify a device before realizing it, to make prognoses of how a physical system will evolve in time for a given initial configuration or to produce an application for training or pure entertainment.

We make the following division in domains of applications although it should be pointed out that there is a clear trend in convergence between these domains and increasing use in common software libraries, models and computational techniques. The domains are already today somewhat overlapping.

1.1.1 Computational science

Computational science is the field of mathematical modeling and numerical techniques and the systematic use of computing and simulation in order to gain deeper understanding and resolve questions in various scientific disciplines. Computer simulations complements the use experiments in the development of new theory and testing of predictions. Examples of computational science is climate simulation, molecular dynamics simulation to find stable protein structures, forecasting of tsunamis and earthquakes, analysis of the human genome. Computational science also includes the mathematics and computer science that enables simulations and advanced analysis, e.g., numerical analysis, optimization, Monte-Carlo simulation, linear and quadratic programming, parallel computing. It is not uncommon that software is made accessible and free to use for non-commercial use (open source). It is also common with simulations are run on large clusters or supercomputers, sometimes for several months.

1.1.2 Computational engineering

Computational engineering is the field of applied and numerical mathematics for solving engineering problems or engineering tools for design and analysis tasks. The software is often a commercial product and designed as set of flexible tools well fitted for the engineering tasks and work-pipeline with good functionality for importing and exporting data to and from databases and other software. These tools are referred to as computer-aided design (CAD) software. Other software is more specialized code libraries for particular problems, e.g., optimization problems, signal analysis or even adapted for a particular machine. Examples of computational engineering tasks are finding the optimal design for a component that minimizes production coast but maximizes durability for a given set of deformation forces, computing the trajectory of a satellite, simulation of walking robots for for testing of control algorithms, analysis of the structural integrity of a ship, simulating the effect of installing a new control system in a chemical plant.

1.1.3 Design, visualization and animations

The software tools for artists and designers have increasingly more elements of computing and simulation integrated, especially software intended to produce interactive applications or animations. To produce an animation by specifying the motion frame-by-frame is extremely time-consuming, especially if the scenes are complex and involves many moving objects that should behave realistically. By incorporating physics-based simulations in the tools the result automatically behaves more realistic according to the implemented laws of nature. But, even more importantly, it suffices to specify the geometry, physical properties, initial state once and let the simulation software compute the future configurations frame by frame – with some interaction from the animators or from real actors using motion capture technology. This way also the work required to produce complex visualizations and animations is reduced from large teams of artists and animators to a few artist using powerful computers. There are now many software tools and companies specialized in physics-based animation and computer generated special effects for movies. Some specializations are character animation and clothing, crowds of people and animals, fluid simulation, explosions and destruction.

1.1.4 Virtual environments and games

One particular set of software and applications are those designed to run in real-time and with humans and other hardware or software in-the-loop interacting with the simulation. We refer to these applications as *virtual environments*. Computer games is one special case that dominates commercially and user-wise, and thus strongly impacts the technology – both software and hardware – used in other types of virtual environments. Other examples of virtual environments are simulators for vehicle operator training, practicing surgical procedures, simulators for research and development of new machine designs and interfaces through virtual experiments with humans in-the-loop. Further more examples are educational software, recreational virtual environments, tools in neurological and psychological studies on perception and learning, and for medical treatments and rehabilitation. Key requirements on virtual environments include

Real-time. Interactive systems with visual feedback usually runs at a frame rate of 60 Hz in order to be perceived as smooth in time. This means that for each frame there is at maximum 0.016 s of computing time. If the computations for producing the data for each frame takes more than 0.016 s the application will run slower than real-time. If the computations are faster, the system is designed to take a short breaks to match real-time.

3D-computer graphics. The simulation must produce data in a form that the visualization engine accepts, e.g., position and orientation of the geometrical shapes involved in the application. There are data standardization for 3D graphics software and hardware, and usually a middle-ware (scene graphs) is used that simplifies this data stream.

Robustness. All users appreciate software that doesn't crash or misbehave. In simulation of dynamical systems there is high risk for numerical instabilities to develop. Small errors may accumulate to large errors. This often leads unnatural, jittery behavior of the objects or even uncontrolled explosive motion which completely ruins the simulation and results in a system crash at worst. Avoiding and suppressing numerical instabilities by making good choice in mathematical models and numerical methods is of utmost importance.

Faithful dynamics. The very purpose of using physics-based simulation to construct applications is that the virtual environment should behave according to the very laws of nature that has been implemented, e.g., Newton's laws of motion. Since modeling and numerical simulation involves both simplifying assumptions, approximations and small numerical errors the correspondence cannot be total. In every event of the simulation the dynamics should be faithful to the laws of nature, e.g., preservation of total energy and momentum, force-balance etc.

Complexity. The software must allow modeling of sufficiently complex environments to be relevant. Complexity is often measured by the number of degrees of freedom in the system, i.e., the number of elements and the number of ways the state of each element may change. The number of possible ways of interaction between the elements and the geometrical shapes also contributes to complexity.

User interaction and flexible design. The application must allow for user interaction, e.g., from signals via joysticks, mouse, keys, optical sensors etc. These signals are usually translated into forces or target velocities for particular objects in the simulation, e.g., for increasing the throttle on a vehicle engine, turning the steering wheel or pushing on the side of a box. The software needs a flexible design that enables modeling and interfacing with many types of simulated systems, e.g., vehicles, robots, characters, biomechanical systems, and with mechanical or electrical hardware.

A code library dedicated to physics based-simulation for virtual environments are often referred to as a *physics engine* or a *physics API* (Application Programming Interface). A physics engine is combined together with other code libraries for visualization, audio, haptics, and hardware and with data models to form applications – virtual environments. A list of physics engines is given below. Some of them are profiled for game physics and others for professional simulators. Some are commercial some are free (open source).

- AgX Multihphysics Toolkit – www.algoryx.se
- Bullet Physics – bulletphysics.org
- Box2D – box2d.org
- Chrono::Engine – www.deltaknowledge.com
- Havok – www.havok.com
- Newton Game Dynamics – newtondynamics.com
- ODE (Open Dynamic Engine) – www.ode.org
- Open Tissue - www.opentissue.org
- PhysX – www.nvidia.com/object/nvidia_physx.html
- Simple Physics Engine - www.spehome.com
- Tokamak – www.tokamakphysics.com
- Vortex - www.vxsim.com

1.2 The Fundamentals

1.3 History

1.4 Outlook

Chapter 2

Modeling and Simulation

2.1 Types of models and simulations

Following the book [1] we distinguish between the following type of computer simulations: *discrete event models and simulation*, *discrete time models and simulations*, *continuous time models and simulations*. We focus primarily on continuous time models and simulations for models based on ordinary differential equations, partial differential equations or differential algebraic equations. These models are the common ones in the realm of mechanics and multibody systems. There are, however, many natural phenomena that can be reproduced and studied using discrete event and discrete time models and simulations. There also exist hybrids of these models and simulations.

Computer simulation of multibody systems require representation of data and computational operations on the data. For efficient simulations the data representation and computational operations must be well adapted to each other and to the other software and hardware components of the simulation, e.g., processor architecture, memory storage and visualization and interaction tools. When the simulation code has many users and developers or is to be a general software component for producing numerous applications it is also important that the code is distributed as as an accessible software library with a flexible yet intuitive interface – an *API* (*Application Programming Interface*).

2.2 Data representation

There are several types of data that needs representation. We distinguish between *state variables*, *computed quantities*, *derived quantities*, *constants*. The state variables are those that define the system state a particular point of the simulation, e.g., position and velocity. The simulation is equipped with some rule for updating the state variables, e.g., equations of motion. Computed quantities are quantities that are not state variables but are necessary in-data for updating the state variables to the next point in the simulation, e.g., force, torque, impulse, contact points. The derived quantities are quantities are simulation output or by-products that depends directly on the state variables but is not a state variable itself, e.g., energy, momentum, oriented geometrical shape, moment of inertia tensor. Constants are quantities that define the model or simulation, e.g., mass, friction coefficient, geometry, gravity constant, time-step size.

In physics-based simulation there is a number of ways for the elements (particle, rigid body, deformable body, field, etc.) to interact with each others or with the user or other hardware or software components in the simulation loop. In object-oriented code the element representations may also include methods for interaction. In other code these have distinct representation.

Many continuous time models are formulated as a set of *ordinary differential equations*, which may be written as

$$\hat{D}\ y(t) = f(y(t), t) \tag{2.1}$$

where $y(t)$ represents the vector of ¡state variables $y(t) = [y_1(t), y_2(t), ...]^T$ at time t, $\hat{D}$ is a *differential operator*, e.g., the time derivative $\hat{D} = d/dt$ and $f(y(t), t)$ is some given function that may depend on $y(t)$ or t or both. The state vector $y(t)$ may represent the positions and velocities of a set of bodies and $f(y(t), t)$ may be the force acting on the bodies. The differential equation may describe the rate of change of velocity due to the force acting on it. There are several ways for obtaining numerical approximations to the differential equations. These lead to

some rule for computing $y(t_1)$ given sufficient information of $y(t)$ at previous times, e.g., $y(t_0)$. This rule may be formulated as

$$y(t_0) \rightarrow y(t_1) = \texttt{solveODE}(\hat{D}, y(t), t_0, t_1) \tag{2.2}$$

where `solveODE()` represents some numerical computation, e.g., a simple multiplication or solving a linear or nonlinear system of equations. In a simulation this update rule is applied repeatedly to advance the system in time. As we shall see in this text, there are many generalizations to this continuous time model. The problem may be a set of *partial* differential equations where the state variable also depends on the position, i.e., $y(x,t)$ and the differential operator may depend on space and time and. It is not uncommon that the right hand side is a discontinuous function (in the case of contact impulse exchange it is a discontinuous and infinite). The computational solution method may require more data points besides for t_0. The differential equation model may be supplemented with constraint equations, of equality or inequality type, e.g., $g(y(t),t) = 0$ for some constraint function g. This continuous time model is then a system of *differential algebraic equations*, which require special solution techniques. These generalizations will, however, not entirely change the way data needs to be represented but rather call for extensions.

2.2.1 MATLAB/Octave examples

There are several forms of representations of the same data. The choice is often a choice between code readability and computational efficiency. Next we give example of representations in MATLAB/Octave programming environment. As an example system we chose a system of N_p particles governed by Newton's equations of motion, see Section 3, and coupled together by a set of N_s spring forces, see Section 3. Each particle is labeled by i ranging from 1 to N_p. Each particle has a position vector $\mathbf{x}_{(i)} = [x_1^{(i)}, x_2^{(i)}, x_3^{(i)}]$ and velocity vector $\mathbf{v}_{(i)} = [v_1^{(i)}, v_2^{(i)}, v_3^{(i)}]$, where the subscript $1, 2, 3$ denotes the three spatial directions (x,y,z-coordinates). On each particle there act a total force $\mathbf{f}_{(i)}$ and the particle energy is denoted $E_{(i)}$. Each particle has mass $m_{(i)}$. Each spring is labeled by s ranging from 1 to N_s. For each spring connecting particle i_a to i_b there is a spring force $\mathbf{f}_s^{\text{spring}}$ on particle i_a and a counteracting spring force $-\mathbf{f}_s^{\text{spring}}$ on particle i_b. The continuous time model in this case have the form

$$\frac{d}{dt} y(t) = f(y(t), t) \tag{2.3}$$

with

$$y(t) = \begin{bmatrix} \mathbf{x}_{(1)} \\ \mathbf{x}_{(2)} \\ \vdots \\ \mathbf{v}_{(1)} \\ \mathbf{v}_{(2)} \\ \vdots \end{bmatrix}, \qquad f(t) = \begin{bmatrix} \mathbf{v}_{(1)} \\ \mathbf{v}_{(2)} \\ \vdots \\ m_{(1)}^{-1}\mathbf{f}_{(1)} \\ m_{(2)}^{-1}\mathbf{f}_{(2)} \\ \vdots \end{bmatrix} \tag{2.4}$$

A simulation requires systematic book-keeping of the state variables as well as how the particles are connected by the springs and the spring forces. Each particle has a total spring force $\mathbf{f}_{(i)} = \sum_s \mathbf{f}_s^{\text{spring}}$, with summation over all springs particle (i) is connected to. It is then a very bad idea to refer to the particles by explicit variable names, e.g., `particle_1`, `particle_2`, ... and masses as `m_1`, `m_2`, ... This would make it complicated to change the number of particles in the simulation and there would be no way for systematic mapping between particles and forces.

Representation using structured arrays

Structured arrays, or *structs* for short, offers a representation that is similar to the mathematical notation for each body of the simulation. It gives high readability to the code and is flexible to use. On the down-side is that it is not as computational efficient as using only vectors and matrices.

The particle system can be represented as

```
particle(i).x                    % state variable
particle(i).v                    % state variable
particle(i).f_tot                % computed quantity
```

```
particle(i).E_tot                                % derived quantity
particle(i).m                                    % constant
```

Setting the position of particle 4 to $\mathbf{v}_{(4)} = [1, -1, 0]^T$, its mass to 0.5 and copying its total energy to `E_tmp` is then accomplished by

```
particle(4).x = [1, -1, 0]';
particle(4).m = 0.5;
E_tmp = particle(4).E_tot
```

The spring forces can be represented as

```
spring(s).a                                      % spring particle a
spring(s).b                                      % spring particle b
spring(s).length                                 % spring rest length
spring(s).ks                                     % spring coefficient
spring(s).kd                                     % spring damping coefficient
```

Creating a spring, named spring number 2, between particle 1 and 9 with rest length $l = 0.3$, spring coefficient 0.1 and damping coefficient 0.01 can be done as

```
s = 2;
spring(s).a = 1;
spring(s).b = 9;
spring(s).length = 0.3;
spring(s).ks = 0.1;
spring(s).kd = 0.01;
```

and computing the force of spring 5, with length l_5 on particle i_a from particle i_b

$$\mathbf{f}_5^{\text{spring}} = -k_5 \left(|\mathbf{x}_{(i_a)} - \mathbf{x}_{(i_b)}| - l_5 \right) \frac{\mathbf{x}_{(i_a)} - \mathbf{x}_{(i_b)}}{|\mathbf{x}_{(i_a)} - \mathbf{x}_{(i_b)}|}$$

and storing it as `f_tmp` as

```
s = 5;
i_a = spring(s).a;
i_b = spring(s).b;
length = spring(s).length;
ks = spring(s).ks;
dx = particle(i_a).x - particle(i_b).x;
f_tmp = - ks * ( norm( dx ) - length ) * dx / norm( dx );
```

Representation using matrices

MATLAB/Octave is optimized for working with matrices and vectors. The simulation code will therefore run faster if it is uses matrix and vector operations as much as possible rather than loops of operations on struct elements. The effect is especially significant when the update rule, Eq. (2.2), is in form of a system of linear equation $Ay = b$, for some matrix A and vector b. It is then practical to collect the variables either by the state vector y or by a global position vector X, velocity vector V, force vector F and mass matrix M as

$$X = \begin{bmatrix} \mathbf{x}_{(1)} \\ \mathbf{x}_{(2)} \\ \vdots \end{bmatrix} = \begin{bmatrix} x_1^{(1)} \\ x_2^{(1)} \\ x_3^{(1)} \\ x_1^{(2)} \\ x_2^{(2)} \\ x_3^{(2)} \\ \vdots \end{bmatrix} \quad , \qquad V = \begin{bmatrix} \mathbf{v}_{(1)} \\ \mathbf{v}_{(2)} \\ \vdots \end{bmatrix} = \begin{bmatrix} v_1^{(1)} \\ v_2^{(1)} \\ v_3^{(1)} \\ v_1^{(2)} \\ v_2^{(2)} \\ v_3^{(2)} \\ \vdots \end{bmatrix} \tag{2.5}$$

$$F = \begin{bmatrix} \mathbf{f}_{(1)} \\ \mathbf{f}_{(2)} \\ \vdots \end{bmatrix} = \begin{bmatrix} f_1^{(1)} \\ f_2^{(1)} \\ f_3^{(1)} \\ f_1^{(2)} \\ f_2^{(2)} \\ f_3^{(2)} \\ \vdots \end{bmatrix} \quad , \qquad M = \begin{pmatrix} m_{(1)} & 0 & 0 & 0 & 0 & 0 & \dots \\ 0 & m_{(1)} & 0 & 0 & 0 & 0 & \dots \\ 0 & 0 & m_{(1)} & 0 & 0 & 0 & \dots \\ 0 & 0 & 0 & m_{(2)} & 0 & 0 & \dots \\ 0 & 0 & 0 & 0 & m_{2)} & 0 & \dots \\ 0 & 0 & 0 & 0 & 0 & m_{(2)} & \dots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots \end{pmatrix} \tag{2.6}$$

Observe that the vectors X, V, F have dimension $3N_p$ and the mass matrix M have dimension $3N_p \times 3N_p$. With these notations the equation of motion, Eq. (2.3), reads

$$\frac{d}{dt}X = V \tag{2.7}$$

$$\frac{d}{dt}V = M^{-1}F \tag{2.8}$$

A MATLAB/Octave representation in matrix form of a particle system with all $N_p = 10$ particle positions and velocities equal to $[0, 0, 0]^T$ and mass equal to $m = 5$ is thus given by

```
N_p = 10;
X = zeros( 3*N_p , 1 );
V = zeros( 3*N_p , 1 );
M = 5 * eye( 3*N_p , 3*N_p);
```

Setting the position of particle 4 to $\mathbf{v}_{(4)} = [1, -1, 0]^T$ and its mass to 0.5 is accomplished by

```
i = 4;
X( 3*(i-1)+1 : 3*(i-1)+3, 1) = [1,-1,0]';
M( 3*(i-1)+1 : 3*(i-1)+3 , 3*(i-1)+1 : 3*(i-1)+3 ) = 0.5 * eye( 3 , 3 );
```

Reading the value of $\mathbf{x}_{(4)}$ and storing it as a 3-vector `x_tmp` is done by

```
i = 4;
x_tmp = X( 3*(i-1)+1 : 3*(i-1)+3, 1);
```

2.3 The simulation algorithm

A generic overall algorithm for a simulation of a mechanical system is given by

```
definitions
initialization
while (running)
collision detection and collision response
     compute forces and constraints
     stepforward
         solve to update state variables
         udate derived quantities
     simulation I/O
end
post-processing
```

These steps typically includes

definitions Setting the properties that defines the mathematical model and simulation constants, e.g., gravity, friction, mass and geometry of the objects, timestep size, length of the simulation.

initialization Creating the data structures for the simulation and assigning them initial data. Often read from a data initialization file that may contain definitions as well.

collision detection and collision response Searching for colliding geometries. Collision response impulse transfer, projections that resolve overlapping geometries or just storing of contact data for later use.

compute forces and constraints Compute external forces on the objects, internal forces for interacting objects. Constraint forces to maintain joints and contacts. Forces are accumulated to a total force for each object.

stepforward Advances the simulation data from the current point of time to the next.

solve to update state variables Computing the new state variables often requires solving a system of equations.

update derived quantities Once the state variables are updated all derived quantities can be updated.

simulation I/O Handle input and output of the simulation. Input can be user interaction (signals from joystick, mouse, keyboard etc) or data streaming from another simulation or hardware in the loop. Output can be data storage for post-processing, realtime graphics rendering, signals to haptic force-feedback unit.

post-processing Processing of data stored during the simulation into quantities required by the user, e.g., energy, temperature, velocity fields, time-averaged force. Data is stored in file or made into graphs, tables or animation. The system state variables may be saved and used for initialization data for another simulation.

In practice, however, the order of these algorithmic steps may vary and may be iterative rather than sequential, e.g., the solve and update may include steps of collision detection and collision response as well as steps of I/O data exchange with other simulation threads.

2.4 Software architecture

Packaging of simulation code into software library. From API to application.

Chapter 3

Dynamics

Dynamics is the study of the relation between force and motion. A *dynamical system* is a mathematical description of how a system develop in time. A dynamical system may or may not be a mechanical system. Other examples are biological systems (population ecology) or an economical system.

3.1 Equations of motion

3.1.1 Newton's laws of motion

Let $\mathbf{x}$ denote the position vector of the center of mass of a body of mass m. The velocity is $\mathbf{v} = \dot{\mathbf{x}} \equiv d\mathbf{x}/dt$, where we introduce the dot-notation for the time derivative. Newton's laws of motion state that

1. Every body remains at rest or in uniform motion unless acted on by a force $\mathbf{f}$. The condition $\mathbf{f} = 0$ thus implies a constant velocity $\mathbf{v}$ and constant momentum $\mathbf{p} = m\mathbf{v}$.
2. Application of a force alters the momentum according to the relation
$$\mathbf{f} = \dot{\mathbf{p}} \tag{3.1}$$
3. To each action, there is an equal and opposite reaction. Thus if $\mathbf{f}$ is a force exerted on body 1 by body 2, then body 1 exerts force $-\mathbf{f}$ on body 2 that is equal in magnitude and opposite in direction.

When mass is conserved, it is also common to write the second law as $\mathbf{f} = m\mathbf{a}$, with the acceleration $\mathbf{a} = \dot{\mathbf{v}}$. It should be observed that Newton's laws of motion holds not only for point masses but also for extended bodies, rigid, deformable or fluid. These bodies have more dynamics than only the motion of the center of mass, e.g., rotation and vibrations. But the Newton's law of motion for the center of mass holds just as well.

3.2 Trajectories

A given trajectory can be visualized as a curve in space as done in Figure 3.1.

The trajectories have different geometrical shapes, e.g., linear, circular or irregular. At any point the curve has some unit tangent $\mathbf{t}$ and unit normal $\mathbf{n}$ that is orthogonal to the normal such that $\mathbf{t}^T\mathbf{n} = 0$. The velocity vector is always tangential to the curve, $\mathbf{v} = v\mathbf{t}$. The acceleration can be split into the part that is tangential to the curve $a_\mathbf{t}$ and the part that is normal to the curve $a_\mathbf{n}$. Such that
$$\mathbf{a} = a_\mathbf{t}\mathbf{t} + a_\mathbf{n}\mathbf{n} \tag{3.2}$$
The normal acceleration is closely related to the curvature of the trajectory while the tangential acceleration is a measure of how the velocity the increase or decrease along the curve. From the Newton's law of motion, $\mathbf{f} = m\mathbf{a}$, there is a direct relation between force and acceleration and it can actually be deduced from the trajectory what force produced that motion. In simulations we are, however, interested in the inverse problem: to deduce what is the resulting trajectory for a given force.

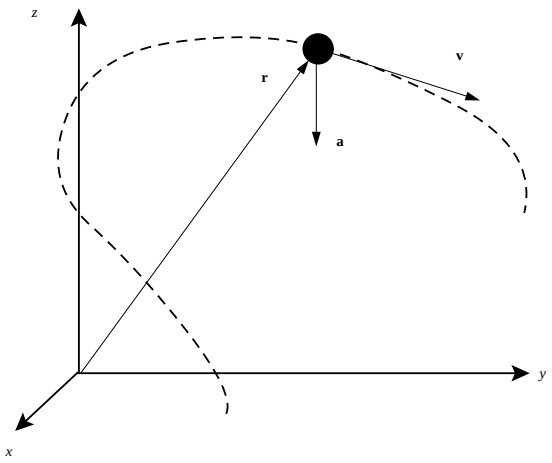

Figure 3.1: Position, velocity and acceleration.

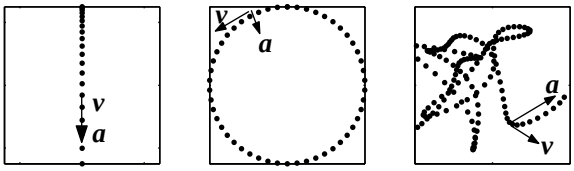

Figure 3.2: Exempel of linear, circular and irregular motion. Each trajectory point are equally spaced in time, i.e., larger velocity means larger distance between the points.

3.3 Forces

We distinguish between *external forces* and *internal forces*. An internal force act between two bodies that are part of the simulation. We also distinguish between *conservative forces* and *dissipative forces*.

3.3.1 Conservative forces

A conservative force does not affect the total energy of the system. The change in kinetic energy from change in velocity is balanced by a change in potential energy. The amount mechanical work exerted by a conservative force on a body is independent of the path of the body through the force field between two given points $\mathbf{x}_{(a)}$ to $\mathbf{x}_{(b)}$.

For every conservative force $\mathbf{f}$ there exist an energy potential $U(\mathbf{x})$ which is a scalar function with the property that

$$\mathbf{f} = -\nabla U \equiv \left[\frac{\partial U}{\partial x_1}, \frac{\partial U}{\partial x_2}, \frac{\partial U}{\partial x_3}\right] \tag{3.3}$$

where we have introduced the gradient operator $\nabla = [\partial/\partial x_1, \partial/\partial x_2, \partial/\partial x_3]$.

3.3.2 Dissipative forces

A dissipative force decreases the total energy of the system. Dissipative forces typically depends on the path along which the object moves, i.e., moving along different paths between two points $\mathbf{x}_{(a)}$ to $\mathbf{x}_{(b)}$ dissipates different amount of energy. Dissipative forces often depends on the velocity but not necessarily.

3.3.3 A Zoo of forces

Here we list some of the common force models in classical mechanics.

Newton's law of gravitation

The gravitational force from body i acting on body j is

$$\mathbf{f}^{\mathrm{G}}_{(ij)} = -G\frac{m_{(i)}m_{(j)}}{x^2_{(ij)}}\mathbf{n}_{(ij)} \tag{3.4}$$

where $\mathbf{x}_{(ij)} = \mathbf{x}_{(i)} - \mathbf{x}_{(j)}$ is the vector displacement vector from body j to i, the distance $x_{(ij)} = |\mathbf{x}_{(ij)}|$ and the unit direction vector $\mathbf{n}_{(ij)} = \mathbf{x}_{(ij)}/x_{(ij)}$. The gravitational constant is $6.673 \times 10^{11} Nm^2kg^2$. The Newton's law of gravitation is used for computing trajectories of bodies in space moving over large distances, e.g., planets and satellites. The potential for body i in the gravitational field of body j is

$$U^{\mathrm{G}} = -G\frac{m_{(i)}m_{(j)}}{x_{(ij)}} \tag{3.5}$$

Gravity force close to the surface of Earth

For motion on shorter distances (less than a kilometer) and close to the surface of Earth it suffices to use an approximation of the gravitational force.

$$\mathbf{f}^{\mathrm{g}} = m\mathbf{g} \tag{3.6}$$

$$U^{\mathrm{g}} = -m\mathbf{x}^T\mathbf{g} \tag{3.7}$$

with the gravity acceleration vector $\mathbf{g} = [0, 0, -g]$, and $g = 9.8\ m/s^2$, using a cartesian coordinate system oriented with the x_3-direction anti-parallel with the direction of gravity (directed towards the center of Earth).

Spring force

In scalar form the undamped spring force between two bodies i and j reads

$$f^{\mathrm{spring}} = -k(x_{(ij)} - L) \tag{3.8}$$

which says that if the distance between the particles is larger than the spring length L there is a force acting to restore the elongation. The size of the force is proportional to the elongation. Similarly if the there is a compression the spring force acts the other way to counteract this. The stiffness of the spring force is regulated by the spring coefficient k. The spring potential is

$$U^{\mathrm{spring}} = \tfrac{1}{2}k(x_{(ij)} - L)^2 \tag{3.9}$$

The generalization to vector form and including dissipation is

$$\mathbf{f}^{\mathrm{spring}}_{(ij)} = -\left[k_s(x_{(ij)} - L) + k_d\frac{\dot{\mathbf{x}}_{(ij)} \cdot \mathbf{x}_{(ij)}}{x_{(ij)}}\right]\frac{\mathbf{x}_{(ij)}}{x_{(ij)}} \tag{3.10}$$

with spring coefficient k_s and damping coefficient k_d.

Viscous damping

Viscous damping, or viscous drag, is the force on a body from moving through a surrounding gas or fluid

$$\mathbf{f}^{\mathrm{visc}} = -\kappa_L|\mathbf{v}|^\gamma\frac{\mathbf{v}}{|\mathbf{v}|} \tag{3.11}$$

where the viscous damping coefficient is introduced, $\kappa_L = \frac{1}{2}\rho A C_L$, where ρ is the mass density of the medium ($\rho = 1.3\ kg/m^3$ for air at Earth surface), A is the cross section area of the object and c is an empirical constant that is different for different geometrical shapes and surface materials, usually between 0 and 1. The γ-factor depends on the type of flow (laminar or turbulent) and $\gamma = 1$ for "low" velocities and $\gamma = 2$ for "high" velocities. Viscous damping is a dissipative force.

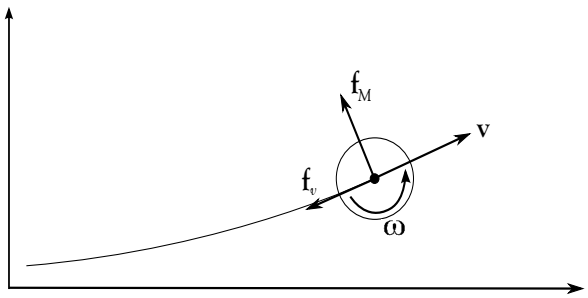

Figure 3.3: Illustration of the Magnus force.

Magnus force

Related to viscous damping (viscous drag) is the Magnus force on rotating bodies. For a moving and rotating sphere, the relative velocity of the body surface relative to the air varies over the body surface, see Figure 3.3. As a consequence also the pressure force varies over the surface. The net effect is a force perpendicular to the linear velocity ($\mathbf{v}$) and the angular velocity (ω)

$$\mathbf{f}^{\mathrm{M}} = \kappa_M |\mathbf{v}|(\omega \times \mathbf{v}) \tag{3.12}$$

where $\kappa_M = 0.5\rho A C_M$, with ρ is the mass density of the gas/fluid, A is the cross-section area ($A = \pi r^2$ for a sphere) and C_M is a numeric factor specific for the surface property. The Magnus force is responsible for the curved trajectories of e.g., footballs, tennis balls, golf balls. The lift force dependence depends nonlinearly on the velocity $\mathbf{v}$. Equation 3.12 is a representative simple model. You can find models in litterature that seems linear but a velocity dependency is then usually hidden in C_M.

Dry friction

Dry friction has two modes, *static friction* and *kinetic friction*. It is very simple to formulate friction models but surprisingly difficult to simulate dry friction. In static friction, the friction force equals the value required for the body to remain at rest relative to the contacting surface unless it exceeds a critical threshold and slips. In both modes the friction force is proportional proportional to the normal force, $\mathbf{f_n}$, between the body and the contacting surface.

$$|\mathbf{f}^{\mathrm{fric}}| \leq \mu |\mathbf{f_n}| \tag{3.13}$$

$$\mathbf{f}^{\mathrm{fric}} = \pm f^{\mathrm{fric}} \mathbf{t} \tag{3.14}$$

The friction force is tangential to the contact plan and directed oppositely to the sliding velocity or the opposite to any force . The friction coefficient μ is in general different for the two friction modes.

Leonard-Jones potential

The Leonard-Jones potential is repulsive at short distances and attractive at longer distances. The attractive part is responsible for gas condensation into liquid droplets at low temperatures.

3.3.4 Constraint forces

Constraint forces are different from other forces. They do not have an explicit dependency on the bodies positions or velocities. Constraint forces come from geometric constraints on the motion of bodies, e.g., two bodies can be constrained to stay at a fixed distance from each others but otherwise free to move. A constraint force is assumed to exist and have the magnitude and direction to maintain this geometric constraint, making it *implicitly* depend on position and velocity. Constraints is a very powerful tool of modeling systems and interactions without the need of knowing the precise nature of the interaction forces but knowing the observable geometric effect.

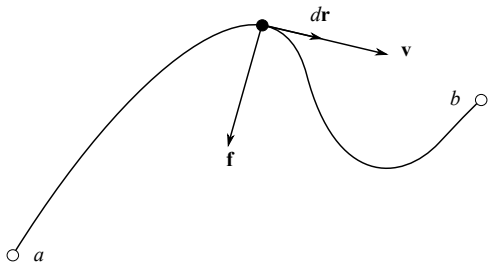

Figure 3.4: A particle moving from a to b with a force $\mathbf{f}$ acting on it.

3.4 Impulses

The effect on the motion by a force does not only depend on the size of the force. It also depends very much on the duration of the force. A strong force with very short duration may have insignificant effect while a weak force may sum up to a large effect if it is given long enough time. Therefore it is reasonable to study the time integrated force, the *impulse*,

$$\mathbf{j} = \int_{t_0}^{t_1} \mathbf{f}(t)dt \tag{3.15}$$

Newton's equations reveals that an impulse can be interpreted as the change in linear momentum

$$\mathbf{j} = \int_{t_0}^{t_1} \mathbf{f}(t)dt = \int_{t_0}^{t_1} \dot{\mathbf{p}}dt = \mathbf{p}(t_1) - \mathbf{p}(t_0) \equiv \Delta\mathbf{p} \tag{3.16}$$

3.5 Invariants

3.5.1 Energy and work

Consider a particle of mass m moving from point a at time t_a to a point b at time t_b along a trajectory $\mathbf{x}(t)$, see Fig. 3.4. The force on the particle is $\mathbf{f}$. At any point in time t the instantaneous particle velocity is $\mathbf{v}(t)$. The force may either work with or against the particle's motion and thus either add energy or remove energy from the system. The amount of *work* exerted by the force on the particle along the path a to b is defined by the line integral

$$\mathbf{W}_{ab} \equiv \int_b^a \mathbf{f}^T d\mathbf{x} = \int_{t_b}^{t_a} \mathbf{f}^T \frac{d\mathbf{x}}{dt}dt = m\int_{t_b}^{t_a} \frac{d\mathbf{v}}{dt}^T \mathbf{v}dt = \frac{m}{2}\int_{t_b}^{t_a} \frac{d}{dt}|\mathbf{v}|^2 dt = \frac{m}{2}v_b^2 - \frac{m}{2}v_a^2 \tag{3.17}$$

In this we have used the Newton equation of motion $\mathbf{f} = m\frac{d\mathbf{v}}{dt}$, the chain rule and the fundamental theorem of calculus. We recognize kinetic energy as $K = \frac{1}{2}mv^2$. This means that the work equals the change in kinetic energy

$$\mathbf{W}_{ab} = K_b - K_a \tag{3.18}$$

A particular type of forces are those that may be represented as a gradient of a potential U, i.e., $\mathbf{f} = -\nabla U(\mathbf{x})$. Applying this yields

$$\mathbf{W}_a b \equiv \int_b^a \mathbf{f}^T d\mathbf{x} = -\int_{t_b}^{t_a} \nabla U(\mathbf{x})^T d\mathbf{x} = -U(\mathbf{x}_b) + U(\mathbf{x}_a) \tag{3.19}$$

This means that

$$K_a + U_a = K_b + U_b \tag{3.20}$$

which we recognize as the conservation of total energy. For an arbitrary set of N_U potentials U_i and momentarily in time we write this as

$$\frac{dE_{\text{tot}}}{dt} = 0 \tag{3.21}$$

where the total energy is

$$E_{\text{tot}}(t) = K + \sum_{i=1}^{N_U} U_i \tag{3.22}$$

A dissipative force, e.g., viscous friction, cannot be written as a gradient of a potential $U(\mathbf{x})$ and will result in a decrease in total energy $\frac{dE_{\text{tot}}}{dt} \leq 0$. However, there exist forces that preserve the total energy but cannot be expressed as the gradient of a potential. One example of this is the magnetic Lorentz force $\mathbf{f} = q\mathbf{v} \times \mathbf{B}$.

3.5.2 Momentum

The momentum of a particle i is the weighted velocity $\mathbf{p}_i = m_i \mathbf{v}_i$. The total momentum of a system of N particles is

$$\mathbf{p} = \sum_{i=1}^{N} \mathbf{p}_i \tag{3.23}$$

Assume that there are no external systems in the system. All forces are inter-particle forces i to j that obey Newton's third law: $\mathbf{f}_{ij} = -\mathbf{f}_{ji}$. The total momentum is then a preserved quantity. To prove this consider the rate of change of total momentum

$$\frac{d\mathbf{p}}{dt} = \sum_{i=1}^{N} \frac{d\mathbf{p}_i}{dt} = \sum_{i=1}^{N} \sum_{j=1}^{N} \mathbf{f}_{ij} = 0 \tag{3.24}$$

where we have noted that for each ij force there is a ji force that cancel it exactly.

Observe that for bodies in a gravity field $\mathbf{g} = -g\hat{z}$ the total momentum is not preserved because it is an approximation that neglects both the nonuniformity of the earths gravitational field as well as the change in momentum of the earth as interacts with a small light body (like an apple). The Newton's gravitational law, on the other hand, preserves total momentum as it is pairwise force consistent with the third law.

3.5.3 Angular momentum

3.5.4 Other invariants

Chapter 4

Numerical time-integration

The time-evolution of a mechanical system usually have the form either as a system of *ordinary differential equations* (ODE), like Newton's equation for particles or the Newton-Euler equations for rigid bodies, or as *partial differential equations* (PDE), such as Navier-Stokes fluid eqautions or the elastic Cosserat model for thin cables. The process of stepping the dynamical system from one time point t_n to another later time point $t_{n+1} = t_n + h$ is referred to as *time integration*, or *time-stepping* and $h > 0$ denotes the time-step size (sometimes denoted by Δt). We simplify the notation of functions at discrete time by $f_n = f(t_n)$, $f_{n+1} = f(t_{n+1})$ etc.

There is a number of different time-stepping algorithms with different properties. The important properties to consider is the *accuracy*, the *numerical stability* and the *computational performance* which the different time-steppers are associated with. Some algorithms are fast to compute but numerically unstable unless small time-steps are used. SFome are computational expensive (requires many mathematical operations and CPU time) but stable and accurate even for large time-steps.

4.1 Equations of motion that are ODEs

4.1.1 Explicit, implicit and semi-implicit Euler

Consider Newton's law of motion $\mathbf{f} = m\mathbf{a}$. We can write this as a set of first order ordinary differential equations

$$\frac{d\mathbf{v}(t)}{dt} = m^{-1}\mathbf{f}(t) \tag{4.1}$$

$$\frac{d\mathbf{x}(t)}{dt} = \mathbf{v}(t) \tag{4.2}$$

We approximate

$$\frac{\mathbf{v}(t+h) - \mathbf{v}(t)}{h} \approx m^{-1}\mathbf{f}(t + \alpha h) \tag{4.3}$$

$$\frac{\mathbf{x}(t+h) - \mathbf{x}(t)}{h} \approx \mathbf{v}(t + \beta h) \tag{4.4}$$

where $0 \leq \alpha, \beta \leq 1$ are real constants. Observe that in the limit of $h \to 0$ we recover precisely the definition of the derivative on the left hand side of the equations and Eq. (4.3)-(4.3) become identical to Eq. (4.1)-(4.1). But for a finite value of h these are a numerical approximation to the original ODE. Depending on the choice of the parameters α and α we generate different time-steppers that will have different properties. We consider three special cases.

Explicit Euler: $\alpha = \beta = 0$

Setting $\alpha = \beta = 0$ gives

$$\frac{\mathbf{v}(t+h) - \mathbf{v}(t)}{h} \approx m^{-1}\mathbf{f}(t) \tag{4.5}$$

$$\frac{\mathbf{x}(t+h) - \mathbf{x}(t)}{h} \approx \mathbf{v}(t) \tag{4.6}$$

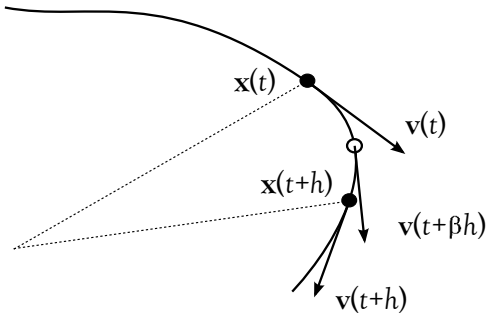

Figure 4.1: Illustration of positions and velocities in the stepping algorithm.

After isolating $\mathbf{v}(t+h)$ and $\mathbf{x}(t+h)$ in each equation and setting $t = t_n$ and $t + h = t_{n+1}$ we obtain the stepping algorithm

$$\mathbf{v}_{n+1} = \mathbf{v}_n + hm^{-1}\mathbf{f}_n \tag{4.7}$$
$$\mathbf{x}_{n+1} = \mathbf{x}_n + h\mathbf{v}_n \tag{4.8}$$

We refer to this as the *Explicit Euler method* (or *Forward Euler*). Explicit Euler uses exclusively the positions, velocities and forces from the last time-step to compute the new values. The advantage of this method is that it is fast to compute. It requires few mathematical operations, mostly just multiplication and addition. The disadvantage is that it is numerically unstable. The numerical errors from each time-step accumulate and grow indefinitely with time. This is very notable when simulating a mass-spring system. The oscillation amplitude grow and grow as well as the total energy of the system. This may be compensated for by using a smaller time-step but this slows down the simulation dreadfully. Figure 4.2 and 4.3 shows the result of applying the explicit Euler algorithm to a particle of mass $m = 20kg$ suspended in a spring of length $L = 1\ m$, spring constant $k_s = 3000\ N/m$, gravity acceleration $9.81\ m/s^2$ and time step $h = 0.01\ s$. The oscillation amplitude as well as the total energy E_{tot} increase in amplitude indefinitely with time. This is a clear sign of numerical instability. The kinetic energy is denoted K and U^{g} and U^{p} are the gravitational and spring potential energy, respecively.

Implicit Euler: $\alpha = \beta = 1$

Setting $\alpha = \beta = 1$ gives

$$\frac{\mathbf{v}(t+h) - \mathbf{v}(t)}{h} \approx m^{-1}\mathbf{f}(t+h) \tag{4.9}$$
$$\frac{\mathbf{x}(t+h) - \mathbf{x}(t)}{h} \approx \mathbf{v}(t+h) \tag{4.10}$$

and we obtain the following stepping algorithm

$$\mathbf{v}_{n+1} = \mathbf{v}_n + hm^{-1}\mathbf{f}_{n+1} \tag{4.11}$$
$$\mathbf{x}_{n+1} = \mathbf{x}_n + h\mathbf{v}_{n+1} \tag{4.12}$$

This is called *Implicit Euler* and it uses the future values of velocity, position and force to compute the new values. At first glance the equations may not look very troublesome. But in almost all cases the force depends on the position of the bodies. It may also depend on the velocities and explicitly on time. This means that $f_{n+1} = f(x_{n+1}, v_{n+1}, t_{n+1})$. Altogether, this make Eqs. (4.11)-(4.11) to a set of hairy entangled equations that may not be possible to solve by analytic means. The alternatives are then to use a Newton-Raphson-solver to find $\mathbf{v}_{n+1}$ and $\mathbf{x}_{n+1}$ based on some guessed seed-solution $\tilde{\mathbf{v}}_{n+1}$ and $\tilde{\mathbf{x}}_{n+1}$, or to make a linearization of the right-hand side in terms of $\mathbf{v}_{n+1}$ and $\mathbf{v}_{n+1}$. This results then in a linear system of equations to be solved. For large systems this can be quite computationally expensive and somewhat tricky to implement all derivatives correctly. The advantage of using implicit integration is that the numerical stability properties are very good - a benefit from looking into the future. Linearized implicit Euler is stable in the sense that the solution is bounded and do not grow indefinitely. But it has numerical dissipation, which means that the system is damped and energy decreases

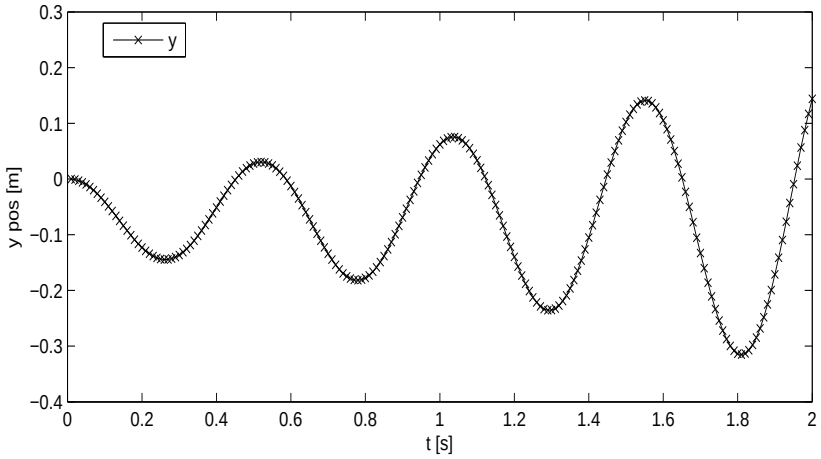

Figure 4.2: The explicit Euler method lead to unstable simulation of a spring-mass system. The position oscillates with increasing amplitude.

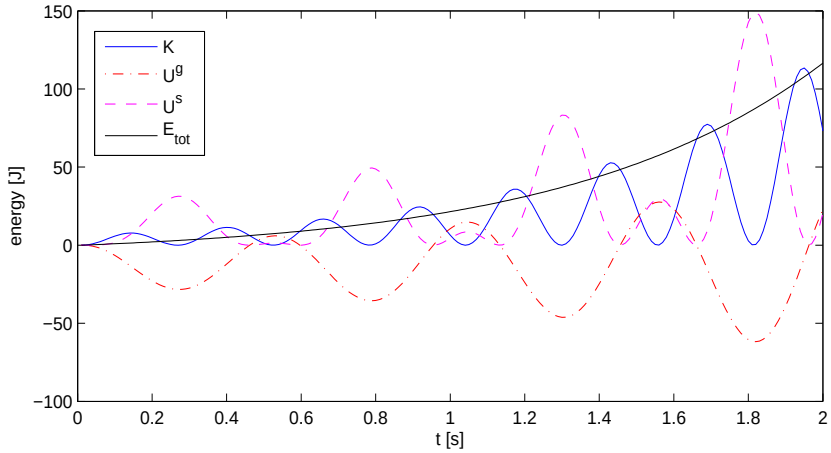

Figure 4.3: The total energy, E_{tot} increases indefinitely with the explicit Euler method. This is a clear sign of numerical instability. increasing amplitude.

with time and motion stops eventually. When the dissipation is small or moderate this is not a big problem because most real systems are dissipative. But for stiff forces and large time-steps the artificial numerical damping may become so big that it exceeds the real-world damping and destroys the physics.

Semi-implicit Euler: $\alpha = 0, \beta = 1$

Setting $\alpha = 0$ and $\beta = 1$ gives

$$\frac{\mathbf{v}(t+h) - \mathbf{v}(t)}{h} \approx m^{-1}\mathbf{f}(t) \tag{4.13}$$

$$\frac{\mathbf{x}(t+h) - \mathbf{x}(t)}{h} \approx \mathbf{v}(t+h) \tag{4.14}$$

which translates to the following stepping algorithm

$$\mathbf{v}_{n+1} = \mathbf{v}_n + hm^{-1}\mathbf{f}_n \tag{4.15}$$

$$\mathbf{x}_{n+1} = \mathbf{x}_n + h\mathbf{v}_{n+1} \tag{4.16}$$

Observe that Eq. (4.15) must be computed first to obtain $\mathbf{v}_{n+1}$ which is then used to compute $\mathbf{x}_{n+1}$ by Eq. (4.16). This time-stepper is a hybrid between explicit and implicit Euler and is called *semi-implicit Euler* or *symplectic Euler* or *Stoermer-Verlet*. It also gives equivalent results to the *Leap Frog* and *Velocity Verlet* algorithms, but there are subtle differences in how these are computed and on how to compute the energy in the system.

The semi-implicit Euler method is just as fast to compute as the explicit Euler method but is numerically stable as long as the time-step is below the highest time-scale τ of oscillations that are inherent in the system, i.e., $h < \tau$. For instance a particle of mass m attached in a spring of stiffness k has oscillation frequency $\omega = \sqrt{k/m}$. The semi-implicit Euler is then stable for time-steps $h < \omega^{-1}$. A similar analysis can be made for other forces as well. When the system has more stiff forces or lighter masses the semi-implicit Euler method fails just like explicit Euler. The choice the stands between using a different time-stepper (implicit Euler or other higher order methods, such as Runge-Kutta) or to model the forces differently, e.g., to replace the stiff forces by kinematic constraints. Figure 4.4 and 4.5 shows the result of applying the semi-implicit Euler algorithm to a particle of mass $m = 20kg$ suspended in a spring of length $L = 1\ m$, spring constant $k_s = 3000\ N/m$, gravity acceleration $9.81\ m/s^2$ and time step $h = 0.01\ s$, which is smaller than $\omega^{-1} = \sqrt{m/k} \sim 0.08\ s$. The oscillation amplitude remains roughly constant as well as the total energy, E_{tot}. There are small oscillations in the total energy that indicates the presence of numerical errors but the error remain bounded. For smaller time steps the error decreases. The kinetic energy is denoted K and U^{g} and U^{p} are the gravitational and spring potential energy, respecively.

4.1.2 Numerical error

Let $y(t)$ denote the exact solution of an ODE. Let the y_n denote the numerical solution at time $t_n = nh$, where $n = 0, 1, 2, ...$, for a given time-integration method and fixed time-step h. The *global numerical error* at time n and step-size h is $\epsilon_n^h = |y(t_n) - y_n|$. This measure include the possible accumulation of errors in the solution over time. The *local numerical error* is the error in one single time-step. In most cases the exact solution is not known and therefore neither is the numerical error. The local numerical error can often be estimated by theoretical considerations. The global error is usually computed using a reference solution obtained using an alternative time-integrator with high accuracy or using very small time-step.

4.1.3 Convergence

The numerical solution is said to *converge* uniformly to the exact solution with decreasing time-step h, i.e., if $\epsilon^{h'} < \epsilon^h$ for $h' < h$. Different numerical methods also have different convergence rate with respect to h.

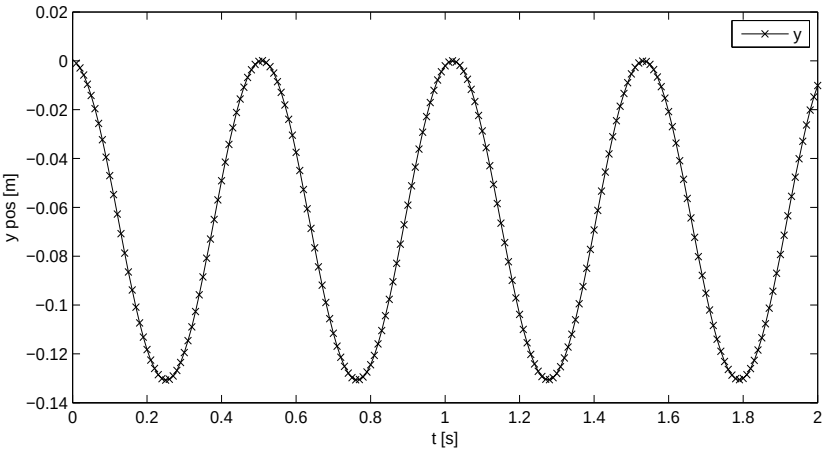

Figure 4.4: The semi-implicit Euler method applied for simulating a spring-mass system. The position oscillates with preserved amplitude.

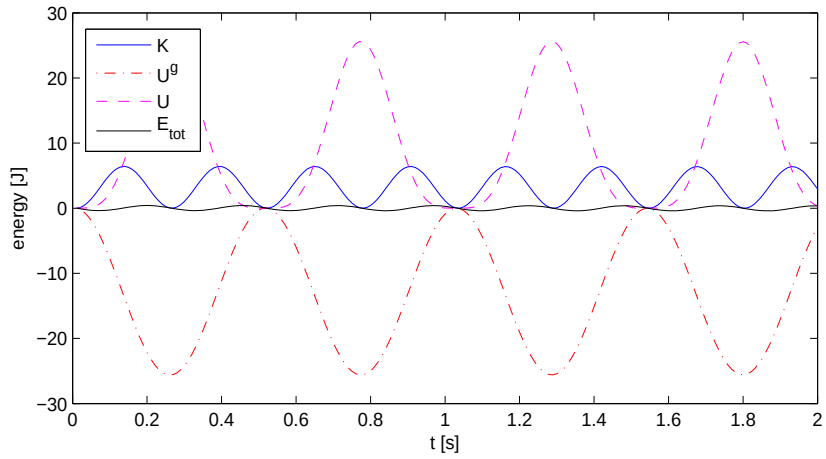

Figure 4.5: The total energy varies somewhat due to numerical errors. The variations remain bounded in the stable regime.

4.1.4 Order and accuracy

The accuracy of different time-integrators depend on time-step size h in different ways. The error is usually denoted by $O(h^n)$ and the method is said to be of order h. High order means high accuracy.

Taylor's theorem can be used to determine the order of an integration method, e.g.,

$$y(t+h) = y(t) + hy'(t) + \frac{1}{2}h^2y''(t) + ... \tag{4.17}$$

This means that the error in one Euler step (the local error) is

$$|y(t+n) - y_{n+1}| = 0 + O(h^2) \tag{4.18}$$

and the method is referred to as being a *first order* method. Observe that the actual size of the error depends not only on h but also on y'. This is why stiff systems are difficult to handle. Stiff systems results sudden changes in y, which means large values of y' and y''. The errors thus become large unless the time-step is kept small. Adaptive time-step control (depending on y') is a common strategy for stiff systems. For applications to realtime visual interactive simulation this is undesired as the computational time become unpredictable.

4.1.5 Numerical stability

Some numerical time-integration methods are *numerically unstable* for some type of systems. In this case the numerical errors accumulate and grow indefinitely with time, often at exponential rate: $\epsilon_n = Ce^{\kappa t}$, where κ is the exponential growth rate. This is observed in Fig. 4.2.

4.1.6 Computational complexity

4.2 Equations of motion that are PDEs

4.3 Variational integration

Observe that high accuracy (high order) is a *local* property. High accuracy is no guarantee that the method preserve system invariants globally, e.g., preservation of momenta and energy over time. In fact, the semi-implicit Euler method, which is a low order method, preserve these invariants very well. This is in contrast to many Runge-Kutta methods that provides trajectories with high accuracy but only for a finite amount of time. The drift from the true solution can be observed by gradual drift in the system invariants. Which solution is the most best depends on the purpose of use of the numerical solution.

Variational integration is a technique for constructing integrators that preserve the physical symmetries related to the system invariants (total energy, momentum, angular momentum).

Chapter 5

Particle systems

Classical particles are very simple objects. Their fundamental properties are mass, position and velocity. Some more properties may be added to give more features, e.g., charge, radius, color, lifetime, elasticity, restitution etc. They distinguish from rigid bodies in that they do not have a sense of orientation and thus do not rotate. But coupled together by interaction forces they form a system of particles that me be very complex and feature rich. Some examples of objects that are often modeled as particle systems include loosely coupled systems such as gas, smoke, plasmas, stars and galaxies, etc., but also strongly coupled systems like fluids, sand, cloth, tissue.

We consider a system of N_p particles governed by Newton's equations of motion. Each particle is labeled by integer i ranging from 1 to N_p. Each particle has a position vector $\mathbf{x}_{(i)} = [x_1^{(i)}, x_2^{(i)}, x_3^{(i)}]$ and velocity vector $\mathbf{v}_{(i)} = [v_1^{(i)}, v_2^{(i)}, v_3^{(i)}]$, where the subscript $1, 2, 3$ denotes the three spatial directions (x,y,z-coordinates). On each particle there act a total force $\mathbf{f}_{(i)}$ and the particle energy is denoted $E_{(i)}$. Each particle has mass $m_{(i)}$. The Newton's equations of motion provide us with the ODEs upon which we base our simulation

$$\frac{d\mathbf{v}_{(i)}}{dt} = m_{(i)}^{-1}\mathbf{f}_{(i)} \tag{5.1}$$

$$\frac{d\mathbf{x}_{(i)}}{dt} = \mathbf{v}_{(i)} \tag{5.2}$$

Simulations of particle systems does, however, not reduce to simple time-integration of these equations. Efficient collision detection become very important and the interactions between the particles may be variable in time or fixed. And the interactions may be soft or stiff. In the case of very stiff interactions where the characteristic collision duration time is less than any acceptable time-step the interactions are rather treated by impulse transitions and/or kinematic constraints.

5.1 Initialization - source and emitter

A particle system can be initialized in different ways. All the particles may be created at start of the simulation with a particular start distribution of positions and velocities etc. The particle initial distribution is many times stored from a previous simulation. An alternative is to create a source, or emitter, that streams particles into the simulation at a given rate and with given particle properties. An emitter typically has a specific cross section area with a normal direction. It is also conventional to place sinks in the simulation where particles are absorbed and removed from the simulation.

5.2 Variable connectivity

Each particle has a set of partners it interact with. We call these neighbors and we say that the system has a connectivity, which may be thought of as a network (or graph). In general, the set of neighbors varies with time – the system has variable connectivity. This is typical for contacting systems.

Let each particle have a radius of interaction $r_{(i)}$. This may be the physical radius of the particles hard core or the range of some long distance interaction force with to others. The radius defines a sphere of interaction (or circle of interaction in 2D).

5.2.1 Elementary contact detection

Two particles a and b are separated by the *relative position vector* $\mathbf{x}_{(ab)} = \mathbf{x}_{(a)} - \mathbf{x}_{(b)}$. The distance between the particles is $x_{(ab)} = |\mathbf{x}_{(ab)}|$. The condition if they are in contact (or interact) is if the spheres of interaction are overlapping

$$|\mathbf{x}_{(ab)}| \leq r_{(a)} + r_{(b)} \tag{5.3}$$

and we compute the *overlap* (or *penetration depth*) $\Delta x_{(ab)}$ as

$$\Delta x_{(ab)} = r_{(a)} + r_{(b)} - |\mathbf{x}_{(ab)}| \tag{5.4}$$

As contact normal we may chose the unit direction from b to a, i.e., $\mathbf{n}_{(ab)} = \mathbf{x}_{(ab)}/x_{(ab)}$.

We may also distinguish wether the contact is *impacting contact*, *resting contact* or *separating contact* with the conditions

$$\mathbf{v}_{(ab)}^T \mathbf{n}_{(ab)} < 0 \tag{5.5}$$

$$\mathbf{v}_{(ab)}^T \mathbf{n}_{(ab)} = 0 \tag{5.6}$$

$$\mathbf{v}_{(ab)}^T \mathbf{n}_{(ab)} > 0 \tag{5.7}$$

where the relative velocity is $\mathbf{v}_{(ab)} = \dot{\mathbf{x}_{(ab)}} = \mathbf{v}_{(a)} - \mathbf{v}_{(b)}$

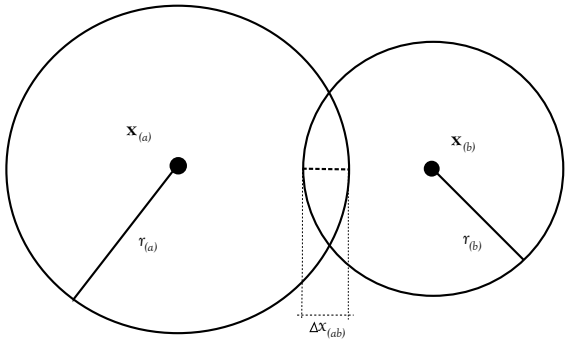

Figure 5.1: Two contacting spherical particles.

5.2.2 Contact with a flat surface

Say that $\mathbf{r}$ is a point on a surface and $\mathbf{n}$ is the normal to the surface. The distance vector from the closest point on the surface to the particle is $[(\mathbf{x}_{(a)} - \mathbf{r})^T \mathbf{n}]\mathbf{n}$. The condition for contact thus become

$$|(\mathbf{x}_{(a)} - \mathbf{r})^T \mathbf{n}| \leq r_{(a)} \tag{5.8}$$

and the overlap is

$$\Delta x_{(a)} = r_{(a)} - |(\mathbf{x}_{(a)} - \mathbf{r})^T \mathbf{n}| \tag{5.9}$$

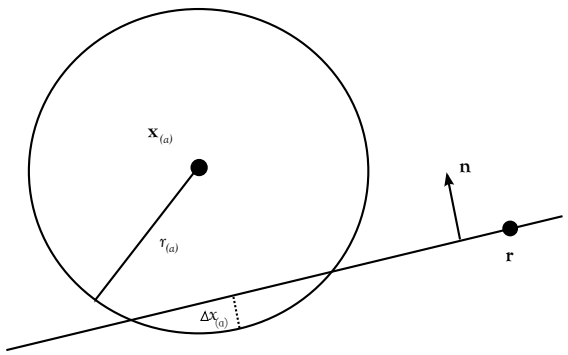

Figure 5.2: A spherical particle contacting with a surface.

5.2.3 Softcore interactions

In softcore interactions there is an interaction force that depends on the overlap $|\mathbf{x}_{(ab)}|$. These are often referred to as *penalty forces*. The most simple version is the linear spring model. The forces are simple to compute but stiff springs require small integration time steps or use of implicit integrators.

Linear spring model

The spring force on particle a from particle b is

$$\mathbf{f}_{(ab)} = \left[k_s \Delta x_{(ab)} - k_d \dot{\mathbf{x}}_{(ab)}^T \mathbf{n}_{(ab)}\right] \mathbf{n}_{(ab)} \tag{5.10}$$

with spring coefficient k_s and damping coefficient k_d. The force on particle b is $\mathbf{f}_{(ba)} = -\mathbf{f}_{(ab)}$. The modification to contact with a static plane surface is straight forward.

Hertz models

Hertz contact models are based on continuum elasticity and considers the finite spherical contact area between the particles. Hertz models take the form of a nonlinear springs.

5.2.4 Hardcore interaction

In hardcore interactions the particles are considered entirely rigid. This can also be thought of the limit of the softcore interactions when the stiffness goes to infinity. The hardcore model is suitable also in simulations where the time step h is larger than the characteristic interaction time of impacting particles. In this case the interactions are conveniently treated using *impulses*, see Sec. **??**.

Two-body collision impulses

The collision impulse model is irrespective of the underlying interaction force. Different forces can give rise to the same impulse transfer. The effect of an impulse transfer is an instantaneous change in velocity.

For simplicity we restrict ourself to two-body collisions. If there are simultaneous contacts with more bodies the impulse exchange is more complex. We denote the incoming velocities of two spherical particles a and b by $\mathbf{v}_{(a)}$ and $\mathbf{v}_{(b)}$, and the outgoing by $\mathbf{v}'_{(a)}$ and $\mathbf{v}'_{(b)}$. The collision results in a net transfer of momentum, impulse $\mathbf{j}_{(ab)}$, between the particles. If the mass of each particle is preserved this new velocities become

$$\mathbf{v}'_{(a)} = \mathbf{v}_{(a)} + m_{(a)}^{-1}\mathbf{j}_{(ab)} \tag{5.11}$$

$$\mathbf{v}'_{(b)} = \mathbf{v}_{(b)} - m_{(b)}^{-1}\mathbf{j}_{(ab)} \tag{5.12}$$

Observe that this process preserves the total momentum in the system

$$\mathbf{p}'_{(a)} + \mathbf{p}'_{(b)} = \mathbf{p}_{(a)} + \mathbf{j}_{(ab)} + \mathbf{p}'_{(b)} - \mathbf{j}_{(ab)} = \mathbf{p}_{(a)} + \mathbf{p}_{(b)} \tag{5.13}$$

Recall that the direction of the impulse is given by $\mathbf{n}_{(ab)}$ which is given from the relative position vector. That means that only the size j of the impulse remains to be computed such that

$$\mathbf{j} = j\mathbf{n}_{(ab)} \tag{5.14}$$

If the collision dissipates no energy and can be considered instantaneous the magnitude can be derived from the requirement that the total energy is preserved.

$$\sum_{i=a,b} \frac{1}{2} m_{(i)} |\mathbf{v}_{(i)}|^2 + \sum_{i=a,b} U_{(i)} = \sum_{i=a,b} \frac{1}{2} m_{(i)} |\mathbf{v}'_{(i)}|^2 + \sum_{i=a,b} U'_{(i)} \tag{5.15}$$

If the collision is instantaneous and does not alter the position, then $U_{(i)} = U'_{(i)}$ and after substituting Eq. (5.11)-(5.12) into it j can be isolated to

$$j = -\frac{2\mathbf{v}_{(ab)}^T \mathbf{n}_{(ab)}}{m_{(a)}^{-1} + m_{(b)}^{-1}} \tag{5.16}$$

There are several models for including dissipative effects to the impulse transfer, e.g., Newton's impact law, Poisson's hypothesis (in terms of normal impulse) and Stronge's hypothesis (in terms of the work done by the normal impulse). The difference between these approaches are important only in the presence of friction.

In the absence of friction we can assume that the dissipative force only acts in the normal direction $\mathbf{n}_{(ab)}$. As a result, the dissipation can only affect the velocity along this direction. Newton's impact law states that

$$\mathbf{v}'^T_{(ab)}\mathbf{n}_{(ab)} = -e\mathbf{v}^T_{(ab)}\mathbf{n}_{(ab)} \tag{5.17}$$

with coefficient of restitution $e \in [0, 1]$. Zero dissipation corresponds to $e = 1$ and maximum dissipation to $e = 0$. The restitution coefficient can be determined empirically by experiments. Combining Newton's contact law (5.17) with Eq. (5.11)-(5.12) we find that impulse magnitude including the effects of dissipation become

$$j = -\frac{(1+e)\mathbf{v}^T_{(ab)}\mathbf{n}_{(ab)}}{m^{-1}_{(a)} + m^{-1}_{(b)}} \tag{5.18}$$

Collision impulses with static surface

The case of a particle a colliding with a static surface b with unit normal $\mathbf{n}$ is found by simply setting $\mathbf{v}_{(a)} = \mathbf{0}$ and taking the limit of $m_{(b)} \to \infty$, such that $m^{-1}_{(b)} \to 0$. The result is

$$\mathbf{v}'_{(a)} = \mathbf{v}_{(a)} + m^{-1}_{(a)}\mathbf{j} \tag{5.19}$$

$$j = -(1+e)m_{(a)}\mathbf{v}^T_{(a)}\mathbf{n} \tag{5.20}$$

Observe that this simplification breaks the conservation of total momentum. In reality the heavy static surface is not static. There is a impulse transfer which alters its velocity slightly. In the limit $m_{(b)} \to \infty$ this change goes to zero and the change in momentum becomes untraceable.

The surface may be represented by a heightfield function $h(x, y)$. The heightfield may be given as an analytic function or as a discrete set of grid points, e.g., in a rectangular grid (x_i, y_j) and corresponding height value z_{ij}. To find wether a point (x, y, z) contacts or penetrates the height field is to test wether $z \leq h(x, y)$. The contact normal can be constructed by finding two vectors $\mathbf{t}_1$ and $\mathbf{t}_2$ that are tangential to the height field at the contact point and then constructing

$$\mathbf{n} = \frac{\mathbf{t}_1 \times \mathbf{t}_2}{|\mathbf{t}_1 \times \mathbf{t}_2|} \tag{5.21}$$

The tangent vectors may be computed

$$\mathbf{t}_1 = [x + \Delta, y, h(x + \Delta, y)]^T - [x, y, h(x, y)]^T \tag{5.22}$$

$$\mathbf{t}_2 = [x, y + \Delta, h(x, y + \Delta)]^T - [x, y, h(x, y)]^T \tag{5.23}$$

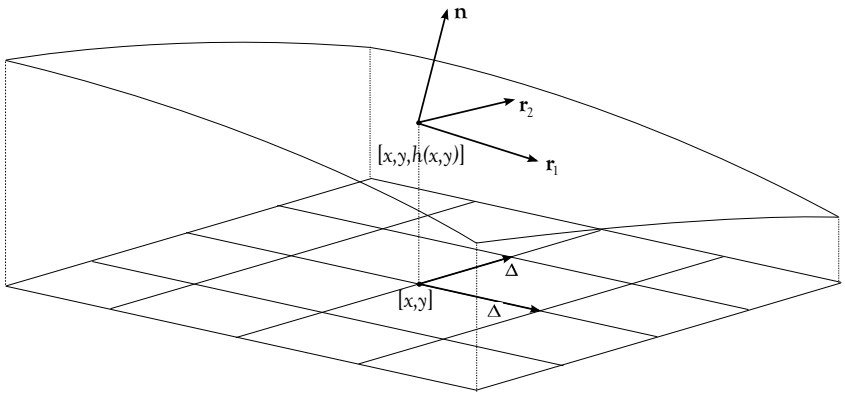

Figure 5.3: A spherical particle contacting with a surface.

Simplified frictional collision impulses

Recall the friction force in Eq. (3.13) and (3.14). A rough simplification is to set

$$\mathbf{f}^{\text{fric}}_{(a)} = \mu|\mathbf{f}_{(ab)}|\mathbf{t}_{(ab)} \tag{5.24}$$

$$\mathbf{t}_{(ab)} = \mathbf{n}_{(ab)} \times (\mathbf{n}_{(ab)} \times \mathbf{v}_{(ab)}) \tag{5.25}$$

The contact impulse thus modifies to

$$\mathbf{j} = j\mathbf{n}_{(ab)} - \mu j\mathbf{t}_{(ab)} \tag{5.26}$$

with j still computed as in Eq. (5.18) or as in Eq. (5.20) for contacts with surfaces.

Observe that this friction model is only well defined when $\mathbf{v}_{(ab)}$ is finite. Therefore this model can only produce slide friction and not stick friction. But worse is, the friction may become larger than the condition $|\mathbf{f}^{\text{fric}}| \leq \mu|\mathbf{f_n}|$. This is unphysical. This may produce change in sign of the relative velocity at the contact point. To avoid this, the fictional force should be clamped to a value that prevents this. More accurate models treats the friction as applying sequential impulses by iterating until the friction condition is fulfilled up to some tolerance level. Even more accurate methods is to treat both the non-penetration contact and frictional forces by introducing constraints. More about this later.

Projections

The impulse method may be combined with a projection of the positions along the contact normal to instantly remove the overlap. The drawback is that this may introduce extra energy in the system. The projections of two particles may introduce overlaps with other particles in the system. The use of projections is not recommended unless it is very important for the application not to have any overlaps. The projected positions are

$$\mathbf{x}'_{(a)} = \mathbf{x}_{(a)} + \delta_{(a)}\mathbf{n}_{(ab)} \tag{5.27}$$

$$\mathbf{x}'_{(b)} = \mathbf{x}_{(b)} - \delta_{(b)}\mathbf{n}_{(ab)} \tag{5.28}$$

and

$$\delta_{(a)} = \frac{m^{-1}_{(a)}}{m^{-1}_{(a)} + m^{-1}_{(b)}}\Delta x_{(ab)} \tag{5.29}$$

$$\delta_{(b)} = \frac{m^{-1}_{(b)}}{m^{-1}_{(a)} + m^{-1}_{(b)}}\Delta x_{(ab)} \tag{5.30}$$

N-body impulses

In general there are more than two bodies involved in a contact. There is no guaranty that applying 2-body impulses to each pair contact of the system. In fact if three or more particles are involved in a collision, one sweep of pair impulses will in general not solve the system. Typically some body pairs will remain in an impacting state. This problem can be remedied by several sweeps of iterations over all contact pairs. This is in fact equivalent to Gauss-Seidel iteations. We will return to this in Section. **??**.

Constraints

Unilateral constraints are suitable for resting contacts.

5.2.5 Kinetic theory

Liouville equation, Boltzmann equation. Ergodicity, ensembles, temperature.

5.3 Fixed connectivity

Particle-spring networks with fixed connectivity. Lattice models. Cloth and volumetrics.

5.4 Miscellaneous

Chapter 6

Solids

6.1 Overview of solids and simulation approaches

Chapter 7

Rigid bodies

7.1 Rigidity

An object is absolutely rigid if the *distance between all parts of the object are fixed*. Observe that this does not say anything of the shape of rigid objects. A rigid object can be a solid 3D geometry (box, sphere, cylinder, arbitrary shape), a 2D surface element, a 1D line element and even a system of point particles – given that all internal distances really are fixed. Many objects can be considered more or less rigid when computing its motion, e.g., rocks, steel, wood, etc. Many other objects can be considered as a collection of rigid bodies connected with various joints, e.g., vehicles, robots, biomechanical systems etc.

The mental picture of particle system is actually quite helpful to understand and derive the dynamic properties of rigid bodies. A particle system has a center of mass position $\mathbf{x}$. The center of mass position has a translation velocity $\mathbf{v}$. Since all internal distances are fixed the only remaining degree of freedom is co-rotation about the center of mass with an *angular velocity* that we denote ω. Newton's law of motion applies also to *the motion of the center of mass*. This is what makes Newton's law of motion so useful. It applies not only to point particles but to extended objects, planets, satellites, cars, kitchen chair etc. But it only speaks of the motion of the center of mass. It must be complemented with equation of motion for the rotational degrees of freedom. This extension is referred to as *Newton-Euler's equations of motion* and relates the rate of change of the angular velocity to the torque produced by any force acting on the body.

The big advantage of modeling an object as a rigid body compared to modeling it as a system of particle is the mathematical simplicity of the rigid body model. The motion of a geometrically complex object can be simulated by numeric integration of only 6 equations (3 for translation and 3 for rotation in 3D – in 2D it is only 3 equations). A system of N particles has $3N$ equations. Let $\mathbf{r}$ be a body-fix position relative to the body center of mass in the frame moving with the body. In world coordinates it is given by

$$\mathbf{x}_\mathrm{p} = \mathbf{x} + \mathbf{r} \tag{7.1}$$

The velocity of the point is the sum of the center of mass velocity and the velocity of the point relative to the center of mass $\dot{\mathbf{r}} = \omega \times \mathbf{r}$. This is given by

$$\dot{\mathbf{x}}_\mathrm{p} = \mathbf{v} + \omega \times \mathbf{r} \tag{7.2}$$

The cross-product notations are very handy in 3D but does not translate to an equally simple notation in 2D. In fact, it is often more simple to consider the 2D-case as 3D but with no motion in, say, the z-direction. For practical reasons we sometimes make use of the following notation where we again use we notation

$$[\mathbf{u} \times \mathbf{v}]_\mathrm{2D} = u_x v_y - u_y v_x. \tag{7.3}$$

This is shorter to write and translates more directly to the 3D case.

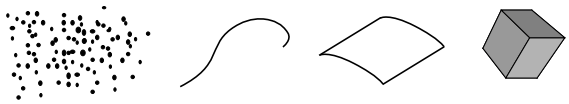

Figure 7.1: A rigid body can be a system of particles, a 1D line object, a 2D surface element or a 3D body. Rigidity means that all internal distances are fixed.

7.2 Rigid body motion in 2D

We represent a 2D rigid body by the state variables (center of mass) position $\mathbf{x}$, translational velocity $\mathbf{v}$, angle of rotation θ (counter clockwise), rotational velocity ω and the constants mass m and moment of inertia I. Any force $\mathbf{f}$ also give rise to a torque $\tau = \pm|\mathbf{r}||\mathbf{f}_\perp|$ that acts to change the rotational velocity, where $\mathbf{r}$ is the position where the force acts relative to the body center of mass and $\mathbf{f}_\perp$ is the component of the force that acts perpendicular to $\mathbf{r}$. The perpendicular component acts to change both the angular velocity and the linear velocity. The rest of the force affects only the linear velocity. The moment of inertia is a rotational inertia. The $\pm$ sign corresponds to wether if the torque acts clockwise or counter-clockwise. A body with large moment of inertia requires a large torque to change its angular velocity. The moment of inertia depends on the amount of mass but also wether it is collected close to the center point (small moment of inertia) or spread over a larger distances form the center of mass. The moment of inertia for various standard geometries can be found pre-computed in tables. For a circular disk of radius r it is $I = (1/2)mr^2$. For a rectangular plate with height h and width w it is $I = (1/12)m(h^2 + w^2)$. The effect by the torque on the rate of angular velocity is given by $I(d\omega/dt) = \tau$.

The *angular momentum* for a rigid body is the sum of the angular momentum of the center of mass motion about the origin plus the angular momentum from the rotation about the bodies own center of mass

$$L = [\mathbf{r} \times \mathbf{p}]_{2\mathrm{D}} = m(xv_y - yv_x) + I\omega. \tag{7.4}$$

Mathematically, the rotational equations of motion have essentially the same structure as the translational equations of motion. If we introduce the *angular acceleration* $\alpha = d\omega/dt$ the similarity is apparent. The center of mass (translational) equations of motion are

$$m_{(i)}\mathbf{a}_{(i)} = \mathbf{f}_{(i)} \tag{7.5}$$

$$\mathbf{a}_{(i)} = \frac{d\mathbf{v}_{(i)}}{dt} \tag{7.6}$$

$$\mathbf{v}_{(i)} = \frac{d\mathbf{x}_{(i)}}{dt} \tag{7.7}$$

with the conservation of the total linear momentum

$$\frac{d\mathbf{p}_{\mathrm{tot}}}{dt} = 0 \tag{7.8}$$

$$\mathbf{p}_{\mathrm{tot}} = \sum_i \mathbf{p}_{(i)} \tag{7.9}$$

The rotational equations of motion in 2D have the similar form

$$I_{(i)}\alpha_{(i)} = \tau_{(i)} \tag{7.10}$$

$$\alpha_{(i)} = \frac{d\omega_{(i)}}{dt} \tag{7.11}$$

$$\omega_{(i)} = \frac{d\theta_{(i)}}{dt} \tag{7.12}$$

with the conservation of the total angular momentum

$$\frac{dL_{\mathrm{tot}}}{dt} = 0 \tag{7.13}$$

$$L_{\mathrm{tot}} = \sum_i m_{(i)}(x_{(i)}v_y^{(i)} - y_{(i)}v_x^{(i)}) + I_{(i)}\omega_{(i)} \tag{7.14}$$

The torque is most practically computed as[1]

$$\tau = r_x f_y - r_y f_x \tag{7.15}$$

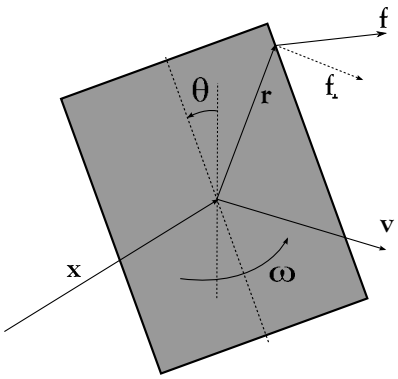

Figure 7.2: A rigid body has translational and rotational degrees of freedom.

How is position $\mathbf{x}_\mathrm{p}$ fixed to a body tracked? Say that the point is given in the rest-frame of the body (for $\theta = 0$) by the relative vector $\mathbf{r}' = [r'_x, r'_y]^T$ from the body center of mass position $\mathbf{x}$. At any later point when the body (and its frame) is rotated by an angle θ the position in world coordinates is given by

$$\mathbf{x}_\mathrm{p} = \mathbf{x} + \mathbf{r} = [x, y]^T + [r_x, r_y]^T \tag{7.16}$$
$$r_x = r'_x \cos(\theta) - r'_y \sin(\theta) \tag{7.17}$$
$$r_y = r'_x \sin(\theta) + r'_y \cos(\theta) \tag{7.18}$$

or in terms of the 2D rotation matrix $R(\theta)$

$$\mathbf{x}_\mathrm{p} = \mathbf{x} + \mathbf{r} = \mathbf{x} + R(\theta)\mathbf{r}' \ , \quad R(\theta) = \begin{pmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{pmatrix} \tag{7.19}$$

The velocity of the co-moving point $\mathbf{x}_\mathrm{p}$ is

$$\dot{\mathbf{x}_\mathrm{p}} = \mathbf{v} + \omega \times \mathbf{r} \tag{7.20}$$

7.2.1 Collision detection in 2D

The following is a simple collision detection algorithm for two convex 2D geometries A and B. The algorithm is based on checking wether the corners overlap all the edges. If so, the shortest overlap gives on which edge the collision occur.

Algorithm 1 Simple 2D collision detection

for all bodies a and b **do**
  for all corners $n = 1, 2, 3, 4$ in a and sides $m = 1, 2, 3, 4$ in b **do**
    compute the side normal $\mathbf{n}_\mathrm{m}$
    compute the distance vector $\mathbf{r} = \mathbf{x}_a^n - \mathbf{x}_b^m$ from corner m to n
    compute the penetration depth $\Delta x_{nm} = \mathbf{r}^T \mathbf{n}_m$
    abort if $\Delta x_{nm} > 0$, i.e., if the corner is found to be outside the object
  end for
  if all Δx_{nm} were negative we have collision
  register the collision, store contact point n, contact normal $\mathbf{n}_m$ corresponding to minimal $|\Delta x_{nm}|$
end for

[1]This can be deduced from the 3D case where $\tau = \mathbf{r} \times \mathbf{f}$

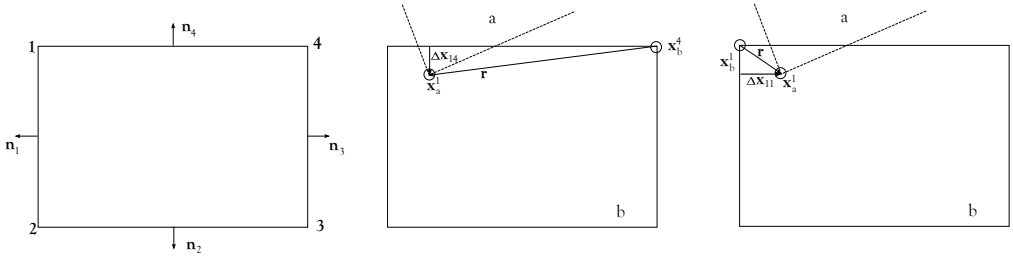

Figure 7.3: A simple collision detection algorithm for convex polygon 2D geometries.

7.2.2 Contact impulses in 2D

Just like a force produces a torque an impulse produce a torque impulse. And just like an impulse corresponds to a change in momentum, a torque impulse corresponds to a change in angular momentum. By the relation $L = I\omega$ this translates directly to a change in angular velocity. For two colliding rigid bodies in 2D Eqs. (5.11)-(5.12) generalizes to

$$\mathbf{v}'_{(a)} = \mathbf{v}_{(a)} + m^{-1}_{(a)}\mathbf{j}_{(ab)} \tag{7.21}$$

$$\omega_{(a)} = \omega_{(a)} + I^{-1}_{(a)}[\mathbf{r}_{(a)} \times \mathbf{j}_{(ab)}]_{\text{2D}} \tag{7.22}$$

$$\mathbf{v}'_{(b)} = \mathbf{v}_{(b)} - m^{-1}_{(b)}\mathbf{j}_{(ab)} \tag{7.23}$$

$$\omega_{(b)} = \omega_{(b)} - I^{-1}_{(b)}[\mathbf{r}_{(b)} \times \mathbf{j}_{(ab)}]_{\text{2D}} \tag{7.24}$$

The impulse including (simplified) contact friction still reads, compare with Sec. 5.2.4,

$$\mathbf{j} = j\mathbf{n}_{(ab)} - \mu j\mathbf{t}_{(ab)} \tag{7.25}$$

but for rigid bodies the magnitude of the impulse modifies to

$$j = -\frac{(1+e)\mathbf{v}^T_{(ab)}\mathbf{n}_{(ab)}}{m^{-1}_{(a)} + m^{-1}_{(b)} + I^{-1}_{(a)}[\mathbf{r}_{(a)} \times \mathbf{n}_{(ab)}]^2_{\text{2D}} + I^{-1}_{(b)}[\mathbf{r}_{(b)} \times \mathbf{n}_{(ab)}]^2_{\text{2D}}} \tag{7.26}$$

where we again we notation $[\mathbf{u} \times \mathbf{v}]_{\text{2D}} = u_x v_y - u_y v_x$. The relative velocity is computed

$$\mathbf{v}_{(ab)} = \mathbf{v}_{(a)} + \omega_{(a)}[-r^{(a)}_y, r^{(a)}_x]^T - \mathbf{v}_{(b)} - \omega_{(b)}[-r^{(b)}_y, r^{(b)}_x]^T \tag{7.27}$$

Observe that by construction the impulse transfer law preserves both linear momentum and angular momentum exactly. Recall also from Sec. 5.2.4 that the friction model is a simplification that has artifacts, e.g., does not produce static friction and may produce friction that is too large such that the relative velocity at the contact point changes sign (this is unphysical).

7.2.3 Time-integration in 2D

The equations of motion for rigid bodies have the same structure as for particles. Applying the semi-implicit Euler discretization produces the stepping algorithm

$$\mathbf{v}_{n+1} = \mathbf{v}_n + hm^{-1}\mathbf{f}_n \tag{7.28}$$

$$\omega_{n+1} = \omega_n + hI^{-1}\tau_n \tag{7.29}$$

$$\mathbf{x}_{n+1} = \mathbf{x}_n + h\mathbf{v}_{n+1} \tag{7.30}$$

$$\theta_{n+1} = \theta_n + h\omega_{n+1} \tag{7.31}$$

7.3 Rigid body motion in 3D

In 3D we chose to represent the motion of a rigid body by the center of mass position $\mathbf{x}$, center of mass linear velocity $\mathbf{v}$, orientation variable $\mathbf{q}$ angular velocity $\omega = [\omega_x, \omega_y, \omega_z]^T$ of the body's

rotation about its center of mass. The Newton-Euler equations of motion for rigid bodies in 3D are

$$m\frac{d\mathbf{v}}{dt} = \mathbf{f} \tag{7.32}$$

$$\frac{d\mathbf{x}}{dt} = \mathbf{v} \tag{7.33}$$

$$I(\mathbf{q})\frac{d\omega}{dt} = \tau + \tau_{\mathrm{g}} \tag{7.34}$$

$$\frac{d\mathbf{q}}{dt} = T(\mathbf{q})\omega \tag{7.35}$$

where $I(\mathbf{q})$ is the 3×3 inertia tensor, $\tau = \sum_i \mathbf{r}_i \times \mathbf{f}_i$ is the torque from the forces $\mathbf{f}_i$ acting on the points $\mathbf{r}_i$ relative to the body's center of mass, and $\tau_{\mathrm{g}} = -\omega \times (I\omega)$ is the *gyroscopic force* responsible for gyrating motion of bodies spinning about non-symmetry axis. The quaternion transform matrix $T(\mathbf{q})$ is given below. Observe that the rotational equations of motion have almost the same structure as the translational equations. All equations are first order ordinary differential equations. The main differences lies in that the inertia tensor is not constant unless the rigid body does not rotate about a symmetry axis. In general, a rotation means that the mass has been re-distributed and the inertia tensor $I(\mathbf{q})$ must be updated. Also related to rotation of asymmetrically distributed mass is the gyroscopic force $\tau_{\mathrm{g}} = \omega \times (I\omega)$. A free body spinning about a non-symmetry axis wobbles due to the gyroscopic force, i.e., the angular velocity vector ω varies both in amplitude and direction. The gyroscopic force is not a real force, it is an effect of having rotating coordinate systems. The gyroscopic force may cause numerical instabilities to low-order integrators. It is not uncommon that it is neglected in large time-step simulations, e.g., virtual environments and computer games.

WARNING: there are many different conventions for 3D rotations, e.g., the Euler angles may be defined in different orders, in different directions and relative to the world frame or to the co-rotating body frame. Be care full not to mix expressions from different conventions.

7.3.1 Representing orientation

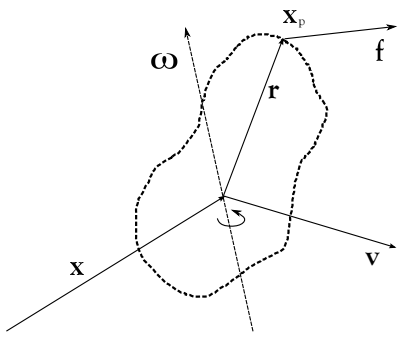

Figure 7.4: Notations for 3D rotations.

In 3D there are many ways to represent orientation, e.g., using rotation matrices, Euler angles and quaternions. Rotation matrices are natural for making transformations, Euler angles are intuitively clear while quaternions are advantageous for computational efficiency and stability. We choose quaternions but sometimes use both rotation matrices and Euler angles for notations and computations when this is simpler. By quaternion it is generally understood that we mean *unit quaternions*.

7.3.2 Quaternion algebra

A quaternion $\mathbf{q} = [w, \mathbf{a}]$ is a 4D complex number with real scalar part w and real 3D vector part $\mathbf{a}$.

The product between quaternion $\mathbf{q}_1 = [w, \mathbf{a}]$ and $\mathbf{q}_2 = [v, \mathbf{b}]$ is defined to by

$$\mathbf{q}_1 \cdot \mathbf{q}_2 = [wv - \mathbf{a}^T\mathbf{b}, w\mathbf{b} + v\mathbf{a}, \mathbf{a} \times \mathbf{b}] \tag{7.36}$$

The conjugate is defined

$$\mathbf{q}^* = [w, -\mathbf{a}] \tag{7.37}$$

The norm is defined

$$|\mathbf{q}| = \sqrt{w^2 + \mathbf{a}^T\mathbf{a}} \tag{7.38}$$

7.3.3 Quaternion representation of 3D rotations

The algebra for unit quaternions, $|\mathbf{q}| = 1$, forms a mathematical group that is strongly related to the geometry of 3D space.

A unit quaternion $\mathbf{q} = [w, \mathbf{a}]$ has geometrical interpretation as a counter-clockwise rotation of angle $\theta = 2\arccos(w) \in [-\pi, \pi]$ about a unit vector $\mathbf{n} = \mathbf{a}/|\mathbf{a}|$, i.e,

$$\mathbf{q} = [\cos(\theta/2), \sin(\theta/2)\mathbf{n}] \tag{7.39}$$

A vector $\mathbf{r}'$ is rotated about the quaternion normal $\mathbf{n}$ by the transformation

$$\mathbf{r} = \mathbf{q} \cdot \mathbf{r}' \cdot \mathbf{q}^* \tag{7.40}$$

where it is understood that $\mathbf{r}' \cdot \mathbf{q}^* = [0, \mathbf{r}'] \cdot \mathbf{q}^*$. The compound of two rotations $\mathbf{q} = \mathbf{q}_1\mathbf{q}_2$ is

$$\mathbf{r} = \mathbf{q} \cdot \mathbf{r}' \cdot \mathbf{q}^* = \mathbf{q}_2 \cdot (\mathbf{q}_1 \cdot \mathbf{r}' \cdot \mathbf{q}_1^*) \cdot \mathbf{q}_2^* \tag{7.41}$$

The conversion from Euler angles, Ψ, Θ, Φ about x, y and z axes, respectively, to quaternions is

$$\mathbf{q} = \mathbf{q}_\Phi \cdot (\mathbf{q}_\Theta \cdot \mathbf{q}_\Psi) \tag{7.42}$$

where

$$\mathbf{q}_\Psi = [\cos(\Psi/2), \sin(\Psi/2), 0, 0] \tag{7.43}$$
$$\mathbf{q}_\Theta = [\cos(\Theta/2), 0, \sin(\Theta/2), 0] \tag{7.44}$$
$$\mathbf{q}_\Phi = [\cos(\Phi/2), 0, 0\sin(\Phi/2)] \tag{7.45}$$

The conversion from quaternion $\mathbf{q} = [w, \mathbf{a}]$ to rotation matrix is

$$R = \begin{pmatrix} 1 - 2(a_y^2 + a_z^2) & 2a_xa_y - 2wa_z & 2a_xa_z + 2wa_y \\ 2a_xa_y + 2wa_z & 1 - 2(a_x^2 + a_z^2) & 2a_ya_z - 2wa_x \\ 2a_xa_z - 2wa_y & 2a_ya_z + 2wa_x & 1 - 2(a_x^2 + a_y^2) \end{pmatrix} \tag{7.46}$$

It can be shown that the time derivative of a quaternion is

$$\frac{d\mathbf{q}}{dt} = T(\mathbf{q})\omega \tag{7.47}$$

with the transformation matrix

$$T(\mathbf{q}) = \frac{1}{2}\begin{pmatrix} -a_1 & -a_2 & -a_3 \\ w & a_3 & -a_2 \\ -a_3 & w & a_1 \\ a_2 & -a_1 & w \end{pmatrix} \tag{7.48}$$

Consider again the tracking of a position $\mathbf{x}_\mathrm{p}$ fixed to a body. Say that the point is given in the rest-frame of the body by the relative vector $\mathbf{r}' = [r'_x, r'_y, r'_z]^T$ from the body center of mass position $\mathbf{x}$. At any later point, when the body (and its frame) is rotated by an amount given by the quaternion $\mathbf{q}$, the world position and velocity of the point is

$$\mathbf{x}_\mathrm{p} = \mathbf{x} + \mathbf{r} = \mathbf{x} + \mathbf{q} \cdot \mathbf{r}' \cdot \mathbf{q}^* \tag{7.49}$$
$$\dot{\mathbf{x}}_\mathrm{p} = \mathbf{v} + \omega \times \mathbf{r} \tag{7.50}$$

Angular momentum and inertia tensor

Recall that the angular momentum for a particle is $\mathbf{L} = m\mathbf{x}\times\mathbf{v}$. Considering a rigid as a collection of particles at fix relative distances this generalizes to the sum of the angular momentum of the center of mass motion and the angular momentum of rotation reltive to the body's center of mass

$$\mathbf{L}_{\text{tot}} = \mathbf{L}_{\text{CM}} + \mathbf{L} = m\mathbf{x} \times \mathbf{v} + \sum_{(i)} m_{(i)}\mathbf{r}_{(i)} \times (\omega \times \mathbf{r}_{(i)}) \tag{7.51}$$

More compactly we write the angular momentum of the rotation about body center of mass

$$\mathbf{L} = I\omega \tag{7.52}$$

where the inertia tensor I is a 3×3 matrix

$$I = [I_{jk}] = \sum_{(i)} m_{(i)}(|\mathbf{r}^{(i)}|^2\delta_{jk} - r_j^{(i)}r_k^{(i)}) \to \int \rho(\mathbf{r})(|\mathbf{r}|^2\delta_{jk} - r_j r_k)dV \tag{7.53}$$

where we introduced the continuous mass density distribution function $\rho(\mathbf{r})$ in the continuum limit.

The inertia tensor of common geometrical shapes can be found in tables. These are typically given in frame of reference aligned with the symmetry axes. For a solid homogenous block with sides a, b and c in the inertia tensor is

$$I' = \frac{m}{12}\begin{pmatrix} b^2+c^2 & 0 & 0 \\ 0 & a^2+c^2 & 0 \\ 0 & 0 & b^2+c^2 \end{pmatrix} \tag{7.54}$$

In a general orientation the inertia tensor is given by

$$I = R(\mathbf{q})I'R(\mathbf{q})^T \tag{7.55}$$

The angular momentum is a fundamentally conserved quantity in the absence of external forces, i.e., $\dot{\mathbf{L}} = 0$. Using $I\omega$ it can be shown that $0 = \dot{\mathbf{L}} = I\dot{\omega} + \omega \times (I\omega)$. Where the second term is the source of the gyroscopic force. Observe that although it give rise to an apparently wobbly motion the angular momentum remains constant. Observe also that the angular momentum and angular velocity are in general not parallel to each other. Only if the rotation is about a symmetry axis.

An external force $\mathbf{f}$ acting on the body at a point $\mathbf{r}$ accelerates the center of mass and give rise to a torque $\tau = \mathbf{r} \times \mathbf{f}$ such that

$$\dot{\mathbf{L}} = I\dot{\omega} + \omega \times (I\omega) = \tau \tag{7.56}$$

Time-integration in 3D

3D rigid body motion has many terms that depends implicitly on orientation and it is possible to produce several time-integration schemes that are very similar. The following we refer to as the semi-implicit Euler stepping algorithm for rigid bodies

$$\mathbf{v}_{n+1} = \mathbf{v}_n + hm^{-1}\mathbf{f}_n \tag{7.57}$$
$$\omega_{n+1} = \omega_n + hI_n^{-1}(\tau_n + \tau_n^{\text{g}}) \tag{7.58}$$
$$\mathbf{x}_{n+1} = \mathbf{x}_n + h\mathbf{v}_{n+1} \tag{7.59}$$
$$\mathbf{q}_{n+1} = \mathbf{q}_n + hT_n\omega_{n+1} \tag{7.60}$$

where

$$\mathbf{f}_n = \sum_i \mathbf{f}_n^i \tag{7.61}$$
$$\tau_n = \sum_i \tau_n^i = \sum_i \mathbf{r}_n^i \times \mathbf{f}_n^i \tag{7.62}$$
$$\tau_n^{\text{g}} = \omega_n \times (I_n\omega_{\mathbf{n}}) \tag{7.63}$$
$$I_n^{-1} = I(\mathbf{q}_{\mathbf{n}})^{-1} = R(\mathbf{q}_{\mathbf{n}})I'^{-1}R(\mathbf{q}_{\mathbf{n}})^T \tag{7.64}$$
$$T_n = T(\mathbf{q}_n) \tag{7.65}$$
$$\mathbf{q}_n = \mathbf{q}_n/|\mathbf{q}_n| \tag{7.66}$$

Chapter 8

Fluids

8.1 Fluid overview

Simulation approaches - pro and cons.

8.2 SPH

Chapter 9

Forces, impulses and constraints

Chapter 10

Collision detection

Chapter 11

Contacts and friction

11.1 The physics of contacts and friction

11.2 Frictionless contact

11.3 Frictional contact

Chapter 12

Constrained multibodies

12.1 General theory

A particle has three degrees of freedom, corresponding to translation in the three directions of three dimensional space. A rigid body has six degrees of freedom, translation plus freedom to rotate about three orthogonal axes. A constraint is a restriction to this freedom, e.g., that removes the freedom to move in one direction in space, not to penetrate a given surface, to stay at fix distance relative to another body or to rotate only about a given axis at a given velocity. Each constraint is associated with a constraint force that is not given an explicit force law like spring forces or gravity. A constraint force is assumed to exist and assumed to attain the precise value to keep the constraint satisfied but not inflict otherwise with the dynamics and not to add or remove any energy from the system.

Constraints is a powerful modelling tool when the actual underlying physics is not important or not well understood but the effect of it can be captured in geometrical relations. An example is the hinge that holds a door to the frame. The contact mechanics is very complicated but the function of swinging about the hinge axis is easily described mathematically.

In this section we will use x to denote the the generalized position vector of our system. The system may be a single particle, a rigid body or a system of bodies with all the position vectors collected in the vector $x = [x_1, x_2, ..., x_i, ...x_N]^T$. We denote $\dim(x) = N$. Correspondingly, v and f will denote the generalized velocity and force vectors that may contain angular velocities and torque if there are any rigid bodies. Masses and inertia tensors are collected on the diagonal of the system matrix M.

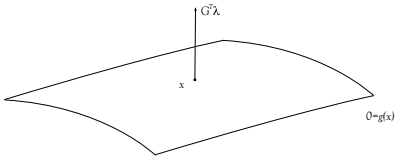

Figure 12.1: Mathematically, a constraint means that the system is confined to move in a hypersurface that comprice a supspace of the original space. The constraint force makes the system stay on the constraint surface but does not affect the motion inside the surface.

A *position constraint* is a hypersurface in the space where x lives. A *holonomic constraint* has the form

$$0 = g(x) \tag{12.1}$$

All positions x that fulfill $g(x) = 0$ are valid positions. The constraint may be of higher dimension than one, i.e., $g = [g_1, g_2, ..., g_i, ...g_M]^T$ and $M < N$. The constraint force should keep the system on the surface but do nothing else. That means that the constraint force should be orthogonal to the surface. Recall that the gradient of the surface function, $G = \frac{\partial g}{\partial x^T}$, is normal to the surface. We call G the *constraint Jacobian*. The constraint force thus have the form

$$f_c = G^T\lambda = G_{ji}\lambda_j \tag{12.2}$$

where λ is called the *Lagrange multiplier* and remains to be computed. Loosely, one may think of the Jacobian as the direction of the constraint force and the Lagrange multiplier as its magnitude.

Consider now the time derivative of the constraint. By the chain rule this produces a condition for valid velocities

$$0 = \dot{g}(x) = \frac{\partial g}{\partial x^T}\frac{dx}{dt} = Gv \tag{12.3}$$

This says that for the constraint to be maintained the system velocity should be tangential to the surface. Observe also that if we v remains tangential then

$$f_c^T v = (G^T\lambda)^T v = G_{ji}\lambda_j v_i = \lambda_j^T(Gv) = 0 \tag{12.4}$$

Physically this means that the constraint force exert no work – it does not add or remove any energy from the system. It only acts to keep the system on the constraint surface. One may also go on with yet another time differentiation to obtain the valid acceleration. Holonomic constraints may be used for linking bodies together into more complex flexible bodies, e.g., robots, vehicles and bio-mechanical systems.

There are also *unilateral* (one-sided) constraints

$$0 \geq g(x) \tag{12.5}$$

and *nonholonomic* constraints

$$0 = g(x, v) \tag{12.6}$$

if it cannot be reduced to a holonomic constraint $0 = g(x)$ by integration. These may be pfaffian velocity constraints $0 = g(v) \equiv Gv$. Unilateraal and nonholonomic constraints are used for contacts and friction.

The equations of motion are extended from ordinary differential equations (ODE) to so called *differential algebraic equations* (DAE), e.g.,

$$M\dot{v} = f + G^T\lambda \tag{12.7}$$

$$0 = g(x) \tag{12.8}$$

Observe that the Lagrange multiplier is still unknown to us. There are several ways to solve such equations numerically. Typically, this is much harder to do than the corresponding ODE. One solution method, referred to as the *acceleration method* is based on differentiating Eq. (12.8) twice with respect to time to make it $0 = \dot{G}v + G\dot{v}$ and substituting it into Eq. (12.7) after first having multiplied it by GM^{-1}. This produces

$$[GM^{-1}G^T]\lambda = -GM^{-1}f - \dot{G}v \tag{12.9}$$

Having computed the Lagrange multiplier from Eq. (12.9) the constraint force $G^T\lambda$ can be computed and the equation of motion integrated by Eq. (12.7). This approach, however, has severe problems with drifts and instabilities. The matrix $GM^{-1}G^T$ may be very ill-conditioned. Furthermore, there is nothing that recognizes and compensates for drift from the constraint surface $0 = g(x)$ due to numerical errors. Adding stabilization terms $\lambda \to \lambda + \alpha g(x) + \beta Gv$ improves this but there is no good way to pick stabilization coefficients α and β that are will give a stable result.

12.2 Stable time-integration of constrained multibody systems

An alternative way is not to solve the hard problem of integrating Eq. (12.7)-(12.9) but instead to solve a problem that is a slight perturbation from the original one. The perturbation is to replace $0 = g(x)$ by $-\varepsilon\lambda = g(x)$, where ε is a small number. This approach is called *constraint regularization* and addresses the problem of the matrix $GM^{-1}G^T$ sometimes being ill-conditioned. But instead of differentiating the constraint function twice we consider the system

$$M\dot{v} = f + G^T\lambda \tag{12.10}$$

$$-\lambda = \varepsilon^{-1}g(x) \tag{12.11}$$

and discretize it directly. The second equation we approximate using Taylor expansion $-\varepsilon\lambda_{n+1} = g(x_{n+1}) \approx g(x_n)+hG_nv_{n+1}$. With a semi-implicit Euler type of discretization of the first equation we end up with

$$Mv_{n+1} - G_n^T\lambda_{n+1} = Mv_n + hf_n \tag{12.12}$$

$$G_nv_{n+1}h^{-1}\varepsilon\lambda_{n+1} = -h^{-1}g_n \tag{12.13}$$

In matrix form this reads

$$\begin{pmatrix} M & -G_n^T \\ G_n & \varepsilon/h \end{pmatrix} \begin{bmatrix} v_{n+1} \\ \lambda_{n+1} \end{bmatrix} = \begin{bmatrix} Mv_n + hf_n \\ -h^{-1}g_n \end{bmatrix} \tag{12.14}$$

This is a linear system $Au = b$ and may be solved using direct or iterative methods. For large systems it is important to exploit the fact that the matrix A is typically very sparse with relatively few non-zero elements often ordered in particular band structure. In a simulation the matrix A must be build each time-step to solve $Au = b$. The system may also be re-written on Schur complement form which reads

$$\left[GM^{-1}G^T + \varepsilon/h\right]\lambda_{n+1} = -hG_nM^{-1}f_n - G_nv_n - h^{-1}g_n \tag{12.15}$$

Observe the source terms for the lagrangian multiplier in the right hand side of Eq. (12.15). The lagrange multiplier creates a force that counteracts any external force component that is orthogonal to the constraint surface (the first term), any invalid velocity pointing outwards from the surface (second term) and any deviation of the system position from the surface (third term). The regularization term $h^{-1}\varepsilon$ in the Schur complement on the left hand side relaxes the constraint from being entirely stiff. A higher value of ε makes the constraint more elastic. Solving Eq. (12.15) for the Lagrange multiplier, the constraint force can be computed as $f_n^{\mathrm{c}} = G_n^T\lambda_{n+1}$ and then the velocity is updated by

$$v_{n+1} = v_n + hM^{-1}(f_n + f_n^{\mathrm{c}}) \tag{12.16}$$

12.3 Constraint library

Some of the commonly used constraints are listed below. These are found in most code libraries for simulation of rigid multibody systems. They are useful for realizing various type of robots, machines and bio mechanical systems. One constraint may be formulated in several ways although the effect on the dynamical system is the same.

Many constraints are two-body constraints restraining the relative motion of two bodies a and b with center of mass position $\mathbf{x}_{(a)}, \mathbf{x}_{(b)}$, orientation $q_{(a)}, q_{(b)}$, translational velocity $\mathbf{v}_{(a)}, \mathbf{v}_{(b)}$ and angular velocity $\omega_{(a)}, \omega_{(b)}$. Co-moving body-fix vectors from body center to joint center are denoted $\mathbf{d}_{(a)}$ and $\mathbf{d}_{(b)}$, respectively. Co-moving body-fix vectors for expressing joint axes are denoted by $\mathbf{u}_{(a)}, \mathbf{w}_{(a)}, \mathbf{z}_{(a)}$ and $\mathbf{u}_{(b)}, \mathbf{w}_{(b)}.\mathbf{z}_{(b)}$, respectively.

In this section we use bold notation for 3-vectors. We sometimes collect both linear and angular velocity of a body in 6-vectors, e.g., $V_{(a)} = (\mathbf{v}_{(a)}^T, \omega_{(a)}^T)$.

The Jacobian of a constraint is most easily identified by noting that the time derivative of zero is zero and that the chain rule thus implies

$$0 = \dot{g}(x) = \frac{\partial g}{\partial x}\dot{x}(t) = Gv \tag{12.17}$$

Recall also that and vector $\mathbf{d}$ that is body-fix to a body a has time derivative

$$\dot{\mathbf{d}} = \omega \times \mathbf{d} = -\mathbf{d} \times \omega = -d^{\times}\omega \tag{12.18}$$

where $d^{\times}$ is the skew matrix of the vector $\mathbf{d} = (d_x, d_y, d_z)^T$

$$d^{\times} = \begin{pmatrix} 0 & -d_z & d_y \\ d_x & 0 & -d_x \\ -d_y & d_x & 0 \end{pmatrix} \tag{12.19}$$

The full set of constraints are stored in a "global" constraint vector

$$g = \begin{bmatrix} g_1 \\ g_2 \\ \vdots \\ g_i \\ \vdots \\ g_M \end{bmatrix} \tag{12.20}$$

It is convenient to let each constraint g_i be an "elementary" constraint (the ones listed below) that is typically of dimension between one and six. A two-body constraint, for body a and b will have the structure

$$G_i = \begin{pmatrix} 0 & \dots & G_{i(a)} & 0 & \dots & G_{i(b)} & 0 & \dots \end{pmatrix} \tag{12.21}$$

where the column positions of $G_{i(a)}$ and $G_{i(b)}$ in G_i are those of $V_{(a)}$ and $V_{(b)}$ in $v = [V_1, V_2, \dots, V_N]$. The full Jacobian matrix is

$$G = \begin{pmatrix} G_1 \\ G_2 \\ \vdots \\ G_i \\ \vdots \\ G_M \end{pmatrix} \tag{12.22}$$

It is sometimes also convenient when we work with a two-body constraint to use the notation of "local" Jacobian where we collect

$$\mathbf{G}_i^{\text{local}} = \begin{pmatrix} G_{i(a)} & G_{i(b)} \end{pmatrix} \tag{12.23}$$

12.3.1 Point-to-point constraint

With a *point-to-point constraint* (p2p) the bodies are free to move such that two body-fix positions $\mathbf{p}_{(a)} = \mathbf{x}_{(a)} + \mathbf{d}_{(a)}$ and $\mathbf{p}_{(b)} = \mathbf{x}_{(b)} + \mathbf{d}_{(b)}$ must coincide at the joint center. This results in a joint much like a shoulder joint. This is often referred to as *spherical joint* or *ball-and-socket joint*. The constraint function is

$$0 = \mathbf{g}^{\text{p2p}} = \mathbf{p}_{(a)} - \mathbf{p}_{(b)} = \mathbf{x}_{(a)} + \mathbf{d}_{(a)} - \mathbf{x}_{(b)} - \mathbf{d}_{(a)} \tag{12.24}$$

The time derivative of the constraint is

$$\begin{aligned} 0 = \dot{\mathbf{g}}^{\text{p2p}} &= \dot{\mathbf{p}}_{(a)} - \dot{\mathbf{p}}_{(a)} = \dot{\mathbf{x}}_{(a)} + \dot{\mathbf{d}}_{(a)} - \dot{\mathbf{x}}_{(b)} - \dot{\mathbf{d}}_{(a)} && (12.25)\\ &= \mathbf{v}_{(a)} - d_{(a)}^{\times}\omega_{(a)} - \mathbf{v}_{(b)} + d_{(b)}^{\times}\omega_{(b)} && (12.26)\\ &= \begin{pmatrix} 1_{3\times3} & -d_{(a)}^{\times} & -1_{3\times3} & d_{(b)}^{\times} \end{pmatrix} \begin{bmatrix} \mathbf{v}_{(a)} \\ \omega_{(a)} \\ \mathbf{v}_{(b)} \\ \omega_{(b)} \end{bmatrix} && (12.27) \end{aligned}$$

where $1_{3\times3}$ is the identity matrix of dimension 3×3. Since $0 = Gv$ we identify the "local" Jacobian matrix as

$$\mathbf{G}_{\text{p2p}} = \begin{pmatrix} 1_{3\times3} & -d_{(a)}^{\times} & -1_{3\times3} & d_{(b)}^{\times} \end{pmatrix} \tag{12.28}$$

or in other words

$$\mathbf{G}_{(a)}^{\text{p2p}} = \begin{pmatrix} 1_{3\times3} & -d_{(a)}^{\times} \end{pmatrix} \tag{12.29}$$

$$\mathbf{G}_{(b)}^{\text{p2p}} = \begin{pmatrix} -1_{3\times3} & d_{(b)}^{\times} \end{pmatrix} \tag{12.30}$$

We observe that the Jacobian has dimension 3×12.

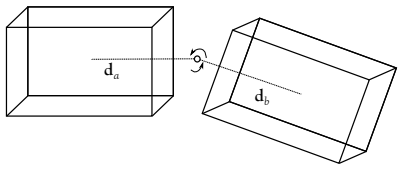

Figure 12.2: Illustration of a point-to-point constraint.

12.3.2 Distance constraint

A distance constraint maintain the distance between the body-fix positions $\mathbf{p}_{(a)} = \mathbf{x}_{(a)} + \mathbf{d}_{(a)}$ and $\mathbf{p}_{(b)} = \mathbf{x}_{(b)} + \mathbf{d}_{(b)}$ at constant length $l^0_{(ab)}$. We denote the separation vector by $\mathbf{l}_{(ab)} = \mathbf{p}_{(a)} - \Delta\mathbf{p}_{(b)}$ and its length $l_{(ab)} = |\mathbf{l}_{(ab)}|$ and the normalized separation vector by $\hat{\mathbf{l}}_{(ab)} = \mathbf{l}_{(ab)}/l_{(ab)}$ The constraint is

$$0 = \mathbf{g}^{\text{dist}} = l_{(ab)} - l^0_{(ab)} \tag{12.31}$$

The Jacobian is identified from the time derivative

$$\begin{aligned} 0 = \dot{\mathbf{g}}^{\text{dist}} &= l^{-1}_{(ab)} \left[\mathbf{l}^T_{(ab)} \dot{\mathbf{p}}_{(a)} - \mathbf{l}^T_{(ab)} \dot{\mathbf{p}}_{(b)} \right] && (12.32)\\ &= \hat{\mathbf{l}}^T_{(ab)} \left(\mathbf{v}_{(a)} + \omega_{(a)} \times \mathbf{d}_{(a)} - \mathbf{v}_{(b)} - \omega_{(b)} \times \mathbf{d}_{(b)} \right) && (12.33)\\ &= \begin{pmatrix} 1_{1\times 3} & -(\hat{\mathbf{l}}_{(ab)} \times \mathbf{d}_{(a)})^T & -1_{1\times 3} & (\hat{\mathbf{l}}_{(ab)} \times \mathbf{d}_{(b)})^T \end{pmatrix} \begin{bmatrix} \mathbf{v}_{(a)} \\ \omega_{(a)} \\ \mathbf{v}_{(b)} \\ \omega_{(b)} \end{bmatrix} && (12.34) \end{aligned}$$

which means

$$\begin{aligned} G^{\text{dist}} &= \begin{pmatrix} 1_{1\times 3} & -(\hat{\mathbf{l}}_{(ab)} \times \mathbf{d}_{(a)})^T & -1_{1\times 3} & (\hat{\mathbf{l}}_{(ab)} \times \mathbf{d}_{(b)})^T \end{pmatrix} && (12.35)\\ G^{\text{dist}}_{(a)} &= \begin{pmatrix} 1_{1\times 3} & -(\hat{\mathbf{l}}_{(ab)} \times \mathbf{d}_{(a)})^T \end{pmatrix} && (12.36)\\ G^{\text{dist}}_{(b)} &= \begin{pmatrix} -1_{1\times 3} & (\hat{\mathbf{l}}_{(ab)} \times \mathbf{d}_{(b)})^T \end{pmatrix} && (12.37) \end{aligned}$$

and the dimensions are $\dim(G^{\text{dist}}) = 1 \times 12$, $\dim(G^{\text{dist}}_{(a)}) = \dim(G^{\text{dist}}_{(b)}) = 1 \times 6$. In deriving this we used the identity $\mathbf{a} \cdot (\mathbf{b} \times \mathbf{c}) = \mathbf{b} \cdot (\mathbf{c} \times \mathbf{a})$ and $\mathbf{a} \times \mathbf{b} = -\mathbf{b} \times \mathbf{a}$.

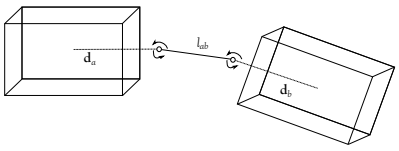

Figure 12.3: Illustration of a distance constraint between two rigid bodies.

If the bodies are point particles there are no rotational terms, i.e.,

$$\begin{aligned} 0 = \dot{\mathbf{g}}^{\text{dist}} &= l^{-1}_{(ab)} \left[\mathbf{l}^T_{(ab)} \dot{\mathbf{p}}_{(a)} - \mathbf{l}^T_{(ab)} \dot{\mathbf{p}}_{(b)} \right] && (12.38)\\ &= \hat{\mathbf{l}}^T_{(ab)} \left(\mathbf{v}_{(a)} - \mathbf{v}_{(b)} \right) && (12.39)\\ &= \begin{pmatrix} 1_{1\times 3} & -1_{1\times 3} \end{pmatrix} \begin{bmatrix} \mathbf{v}_{(a)} \\ \mathbf{v}_{(b)} \end{bmatrix} && (12.40) \end{aligned}$$

which means that the Jacobian of distance constraints for particles is

$$\begin{aligned} G^{\text{dist}} &= \begin{pmatrix} 1_{1\times 3} & -1_{1\times 3} \end{pmatrix} && (12.41)\\ G^{\text{dist}}_{(a)} &= \begin{pmatrix} 1_{1\times 3} \end{pmatrix} && (12.42)\\ G^{\text{dist}}_{(b)} &= \begin{pmatrix} -1_{1\times 3} \end{pmatrix} && (12.43) \end{aligned}$$

with the dimensions $\dim(G^{\text{dist}}) = 1 \times 6$, $\dim(G^{\text{dist}}_{(a)}) = \dim(G^{\text{dist}}_{(b)}) = 1 \times 3$.

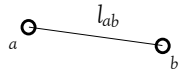

Figure 12.4: Illustration of a distance constraint between two particles.

12.3.3 Hinge constraint

With a hinge two rigid bodies are restricted to revolve about a common axis through the joint center. We use the body fix vectors $\mathbf{u}_{(a)}, \mathbf{w}_{(a)}, \mathbf{z}_{(a)}$ and $\mathbf{u}_{(b)}, \mathbf{w}_{(b)}.\mathbf{z}_{(b)}$. In figure 12.5 the axis of rotation is parallel to both $\mathbf{u}_{(a)}$ and $\mathbf{u}_{(b)}$. We may choose $\mathbf{u}_{(a)}$ to represent the hinge axis. Observe that $\mathbf{v}_{(b)}$ and $\mathbf{w}_{(b)}$ are orthogonal to $\mathbf{u}_{(a)}$ and should remain so for any rotation of a and b about the hinge axis. This means that $\mathbf{u}^T_{(a)}\mathbf{v}_{(b)} = 0$ and $\mathbf{u}^T_{(a)}\mathbf{w}_{(b)} = 0$. The relative joint position vectors $\mathbf{p}_{(a)} = \mathbf{x}_{(a)} + \mathbf{d}_{(a)}$ and $\mathbf{p}_{(b)} = \mathbf{x}_{(b)} + \mathbf{d}_{(b)}$ should also coincide at the joint center. The hinge constraint is thus an extension of the point-to-point constraint that removes two more degrees of freedom – such that the only remaining degree of freedom is rotation about the hinge axis plus rigid motion of the total system. The hinge constraint is also known as a *revolute joint*.

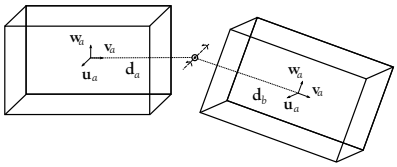

Figure 12.5: Illustration of a hinge constraint.

The hinge constraint can be written as

$$0 = \mathbf{g}^{\text{hinge}} = \begin{bmatrix} \mathbf{p}_{(a)} - \mathbf{p}_{(b)} \\ \mathbf{u}^T_{(a)}\mathbf{v}_{(b)} \\ \mathbf{u}^T_{(a)}\mathbf{w}_{(b)} \end{bmatrix} \tag{12.44}$$

Note that

$$\begin{aligned} \frac{d}{dt}\left(\mathbf{u}^T_{(a)}\mathbf{v}_{(b)}\right) &= \dot{\mathbf{u}}^T_{(a)}\mathbf{v}_{(b)} + \mathbf{u}^T_{(a)}\dot{\mathbf{v}}_{(b)} && (12.45)\\ &= (\omega_{(a)} \times \mathbf{u}_{(a)})^T\mathbf{v}_{(b)} + \mathbf{u}^T_{(a)}(\omega_{(b)} \times \mathbf{v}_{(b)}) && (12.46)\\ &= (\mathbf{u}_{(a)} \times \mathbf{v}_{(b)})^T\omega_{(a)} - (\mathbf{u}_{(a)} \times \mathbf{v}_{(b)})^T\omega_{(b)} && (12.47)\\ &= \begin{pmatrix} 0 & (\mathbf{u}_{(a)} \times \mathbf{v}_{(b)})^T & 0 & -(\mathbf{u}_{(a)} \times \mathbf{v}_{(b)})^T \end{pmatrix} \begin{bmatrix} \mathbf{v}_{(a)} \\ \omega_{(a)} \\ \mathbf{v}_{(b)} \\ \omega_{(b)} \end{bmatrix} && (12.48) \end{aligned}$$

and thus

$$0 = \dot{\mathbf{g}}^{\text{hinge}} = \frac{d}{dt}\begin{bmatrix} \mathbf{p}_{(a)} - \mathbf{p}_{(b)} \\ \mathbf{u}^T_{(a)}\mathbf{v}_{(b)} \\ \mathbf{u}^T_{(a)}\mathbf{w}_{(b)} \end{bmatrix} \tag{12.49}$$

$$= \begin{pmatrix} 1_{3\times3} & -d^{\times}_{(a)} & -1_{3\times3} & d^{\times}_{(b)} \\ 0 & (\mathbf{u}_{(a)} \times \mathbf{v}_{(b)})^T & 0 & -(\mathbf{u}_{(a)} \times \mathbf{v}_{(b)})^T \\ 0 & (\mathbf{u}_{(a)} \times \mathbf{w}_{(b)})^T & 0 & -(\mathbf{u}_{(a)} \times \mathbf{w}_{(b)})^T \end{pmatrix} \begin{bmatrix} \mathbf{v}_{(a)} \\ \omega_{(a)} \\ \mathbf{v}_{(b)} \\ \omega_{(b)} \end{bmatrix} \tag{12.50}$$

The hinge Jacobian is thus

$$G^{\text{hinge}} = \begin{pmatrix} 1_{3\times3} & -d^{\times}_{(a)} & -1_{3\times3} & d^{\times}_{(b)} \\ 0 & (\mathbf{u}_{(a)} \times \mathbf{v}_{(b)})^T & 0 & -(\mathbf{u}_{(a)} \times \mathbf{v}_{(b)})^T \\ 0 & (\mathbf{u}_{(a)} \times \mathbf{w}_{(b)})^T & 0 & -(\mathbf{u}_{(a)} \times \mathbf{w}_{(b)})^T \end{pmatrix} \tag{12.51}$$

$$G^{\text{hinge}}_{(a)} = \begin{pmatrix} 1_{3\times3} & -d^{\times}_{(a)} \\ 0 & (\mathbf{u}_{(a)} \times \mathbf{v}_{(b)})^T \\ 0 & (\mathbf{u}_{(a)} \times \mathbf{w}_{(b)})^T \end{pmatrix} \tag{12.52}$$

$$G^{\text{hinge}}_{(b)} = \begin{pmatrix} -1_{3\times3} & d^{\times}_{(b)} \\ 0 & -(\mathbf{u}_{(a)} \times \mathbf{v}_{(b)})^T \\ 0 & -(\mathbf{u}_{(a)} \times \mathbf{w}_{(b)})^T \end{pmatrix} \tag{12.53}$$

The dimensions are $\dim(G^{\text{hinge}}) = 5 \times 12$, $\dim(G^{\text{hinge}}_{(a)}) = \dim(G^{\text{hinge}}_{(b)}) = 5 \times 6$.

12.3.4 Prismatic constraint

12.3.5 Lock constraint

12.3.6 Point-to-line constraint

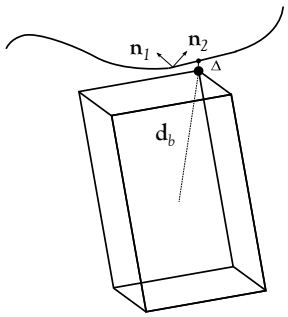

Figure 12.6: Illustration of a hinge constraint.

A particle or a point $\mathbf{p}$ on a rigid body follows a line if the distance between the point and the line is zero. The distance vector $\mathbf{\Delta} = \mathbf{p} - \mathbf{p}_l$ is decomposed in components in the directions of two orthogonal normals to the line $\mathbf{n}_1$ and $\mathbf{n}_2$.

$$0 = \mathbf{g}^{\text{p2line}} = \begin{bmatrix} \mathbf{n}_1^T \mathbf{\Delta} \\ \mathbf{n}_2^T \mathbf{\Delta} \end{bmatrix} \tag{12.54}$$

Note that

$$0 = \dot{\mathbf{g}}^{\text{p2line}} = \begin{bmatrix} \mathbf{n}_1^T (\mathbf{v} + \omega \times \mathbf{d}) \\ \mathbf{n}_2^T (\mathbf{v} + \omega \times \mathbf{d}) \end{bmatrix} \tag{12.55}$$

and the Jacobian is

$$G^{\text{p2line}} = \begin{pmatrix} \mathbf{n}_1^T & -\mathbf{n}_1^T \times \mathbf{d} \\ \mathbf{n}_2^T & -\mathbf{n}_2^T \times \mathbf{d} \end{pmatrix} \tag{12.56}$$

and has dimension $\dim(G^{\text{p2line}}) = 2 \times 6$.

The pont-to-line constraint may be applied to curves by treating them as many short straight line segments. In each transition between two line segments the velocity will violate the constraint condition $Gv = 0$. Many solvers resolve this violation by numerical damping in the direction orthogonal to the the constraint. To remove the effect of dissipation the constraint should rather be expressed by analytic functions or splines. The effect of line curvature will then show up in the Jacobian.

12.4 Constraint solver

By a solver we mean a numerical method for solving the linear set of equations 12.14 or 12.15 with or without complementarity conditions on the solution, e.g., $\alpha < \lambda < \beta$. For the latter case we refer to the solver as a LCP (linear complementarity problem) or MLCP (mixed linear complementarity problem) solver.

The solver may be either direct or iterative. A direct solver computes the solution in a finite number of steps and may be very accurate (exact up to machine precision). It is a common misconception that direct solvers per se are slow and scales poorly with system size. Most mechanical systems give rise to sparse systems of particular structures, i.e., most bodies in the system are coupled only to a small fraction of the total set of bodies and most elements in the matrix are thus zero. If the sparseness is exploited using sparse data representation and if libraries with hardware optimized operations (factorization, vector-vector and vector-matrix multiplication) are used the computations become very efficient and may scale linearly with the number of mechanical components.

Iterative solvers are approximate solvers that approaches the solution with increased number of iterations. The convergence rate varies as well as the computational time per iterations for different iterative solvers. Iterative solvers do not necessarily treat the problem as a system of linear equations. Some solvers just iterate over the set of constraints and compute new constraint forces and velocities that satisfy each constraint at the time. Often these forces are applied as impulses and the method may be referred to as an impulse solver. Each impulse transfer aims to resolve one conflicting constraint but may produce conflicts in the other constraints. At best the constraint violations decrease monotonically with each iteration cycle but it may be difficult to prove this and to predict the convergence rate of a given system at a given state. Some iterative solvers also applies projections of positions and velocities to force the system back to the constraints surface. The projections should preserve linear and angular momentum. Otherwise it is no longer physics consistent with Newton's laws of motion and real world phenomena.

It can be shown that many of the iterative impulse solvers are equivalent to an iterative matrix solver.

12.4.1 Iterative solvers by pair-wise interaction

Naive iterative solver

The idea behind the naive iterative solver is simple. Iterate over each constraint, e.g., pairwise coupling of two bodies. Solve the small two-body problem and store the obtained constraint force as an external force. Single body constraints works the same way. Go on to the next constraint, compute the constraint forces and accumulate them to the body force. The naive solver is easy to understand, easy to implement, fast to compute but has poor convergence rate if there are many constraints and large forces and big mass difference between the involved objects. And most severely, it does not solve the original problems as there are some off-diagonal terms in the Schur complement that are never accounted for, e.g., $G_{k(ab)}M^{-1}_{(ab)}G^T_{k'(bc)}$ which arise if body b is involved in both constraint k to a and in k' to body c. The converging solution may be close to the exact one, but there is no guarantee and the convergence rate may be slow.

Assume the constraints $g = (g_1, g_2, ..., g_k, ..., g_{NC})$ and Jacobians G are given. Each constraint is assumed to connect two bodies a and b together. The "local" Jacobians $G_{k(a)}$ and $G_{k(b)}$ are collected in $G_{k(ab)} = (G_{k(a)}, G_{k(b)})$. We also introduce "local" mass matrix $M_{(ab)} = \mathrm{diag}(M_{(a)}, M_{(b)})$ and force and velocity $f_{n(ab)} = (f^T_{n(a)}, f^T_{n(b)})$ and $v_{n(ab)} = (v^T_{n(a)}, v^T_{n(b)})$.

The algorithm for the naive solver is given in Algorithm 3.

The threshold $|\max(G^T_k\lambda^{[i]})| < f_{\min}$ is set desired accuracy of the constraint force. One may also use condition on how well the velocity lies in the tangent space of the constraint surface as condition, e.g., $|\max(G^T_k v_{n+1}| < \delta$.

The order of integration may affect the convergence rate, e.g., a stack of objects typically converge faster if the iteration order is swapped between top-to-bottom and bottom-to-top. The Schur complement S_k in Eq. (12.64) is typically of dimension $1-3$ in which case there are well-known compact formulas for computing the solution. If S_k is of bigger dimension $4-6$ the inverse may be computed by any dense linear solver library – or a suitable factorization of the matrix that makes the solve of $S_k\lambda^{[i]}_k = b^{[i]}_k$ fast. For bigger matrices sparse matrix solvers should be used.

Add convergence results here.

Algorithm 2 Naive iterative solver by pairwise interaction

1: compute forces other than constraint forces f_n
2: **for** all constraints $k = 1, 2, ..., NC$ **do**
3: get constraint data g_k, $G_{k(a)}$, $G_{k(a)}$ for constraint k connecting body a and b
4: compute the inverse "local" Schur complement for each constraint k

$$S_k^{-1} = \left[G_{k(ab)} M_{(ab)}^{-1} G_{k(ab)}^T + \varepsilon_k / h \right]^{-1} \tag{12.57}$$

5: **end for**
6: **for** $i = 1, 2, ...,$ `MAX_ITERATIONS` **do**
7: **for** all constraints $k = 1, 2, ..., NC$ **do**
8: compute the right hand side for each constraint k

$$b_k^{[i]} = -h G_{k(ab)} M_{(ab)}^{-1} f_{n(ab)} - G_{k(ab)} v_{n(ab)} - h^{-1} g_k \tag{12.58}$$

9: solve the linear system to find the lagrange multipliers

$$\lambda_k^{[i]} = S_k^{-1} b_k^{[i]} \tag{12.59}$$

10: accumulate the constraint force

$$f_{n(a)} = f_{n(a)} + G_{k(a)}^T \lambda_k^{[i]} \tag{12.60}$$

$$f_{n(b)} = f_{n(b)} + G_{k(b)}^T \lambda_k^{[i]} \tag{12.61}$$

11: abort iterations if the maximum constraint force is small: $|\max(G_k^T \lambda^{[i]})| < f_{\min}$
12: **end for**
13: **end for**
14: update the velocity $v_{n+1} = v_n + hM^{-1} \left[f_{n(ab)} + \tilde{f}_{n(ab)} \right]$

Gauss-Seidel solver by pair-wise interaction

The Gauss-Seidel algorithm is almost as simple to implement as the naive solver but has better convergence and solves the original problem when it converges. The iteration updates are constructed to account for off diagonal blocks in the Schur complement from bodies that are involved in several constraints.

The algorithm for the Gauss-Seidel iterative solver by two-body interactions is given in Algorithm 3.

Add convergence results here.

12.4.2 Iterative block-matrix solvers

Jacobi solver

Gauss-Seidel solver

– almost as simple but better convergence

Conjugate Gradient solver

– excellent convergence especially if preconditioned

Algorithm 3 Naive iterative solver by pairwise interaction

1: compute forces other than constraint forces f_n
2: **for** all constraints $k = 1, 2, ..., NC$ **do**
3: get constraint data g_k, $G_{k(a)}$, $G_{k(a)}$ for constraint k connecting body a and b
4: compute the inverse "local" Schur complement for each constraint k

$$S_k^{-1} = \left[G_{k(ab)} M_{(ab)}^{-1} G_{k(ab)}^T + \varepsilon_k / h \right]^{-1} \tag{12.62}$$

5: **end for**
6: **for** $i = 1, 2, ...,$ `MAX_ITERATIONS` **do**
7: **for** all constraints $k = 1, 2, ..., NC$ **do**
8: compute the right hand side for each constraint k

$$b_k^{[i]} = -h G_{k(ab)} M_{(ab)}^{-1} f_{n(ab)} - G_{k(ab)} v_{n(ab)} - h^{-1} g_k \tag{12.63}$$

9: solve the linear system to find the lagrange multipliers

$$\lambda_k^{[i]} = S_k^{-1} b_k^{[i]} \tag{12.64}$$

10: accumulate the constraint force

$$f_{n(a)} = f_{n(a)} + G_{k(a)}^T \lambda_k^{[i]} \tag{12.65}$$

$$f_{n(b)} = f_{n(b)} + G_{k(b)}^T \lambda_k^{[i]} \tag{12.66}$$

11: abort iterations if the maximum constraint force is small: $|\max(G_k^T \lambda^{[i]})| < f_{\min}$
12: **end for**
13: **end for**
14: update the velocity $v_{n+1} = v_n + h M^{-1} \left[f_{n(ab)} + \tilde{f}_{n(ab)} \right]$

12.4.3 Direct solvers

12.5 Effort constraints

12.6 Unilateral constraints and complementarity conditions

12.6.1 LCP solvers

12.7 Joint limits

12.8 Breakable joints

Chapter 13

Vehicles

Chapter 14

Robots

Chapter 15

Complex materials

Chapter 16

Biomechanics

Appendix A

Numerical techniques

A.1 Numerical linear algebra

A.2 Linear programming

Appendix B

Parallel computing and GPGPU

Bibliography

[1] Bernard P. Zeigler, Herbert Praehofer, and Tag G. Kim. *Theory of Modeling and Simulation, Second Edition.* Academic Press, 2 edition, January 2000.