Physically Based Simulation

Alexander Schlögl

February 5, 2018

Contents

This is **my interpretation** of the lecture slides. I tried to be very verbose and explain everything, all while removing irrelevant parts from the lecture. Using this you should be able to pass the lecture easily. **However, I do not take responsibility for any bad results and will not be blamed from anyone. This was a lot of work and I did it to save others (especially students of following semesters) from having to do this themselves. Use this summary at your own responsibility.** If you have any feedback, feel free to create an issue on the git. I don't promise I will fix anything, but I will try.

1 Introduction

Physically based simulation is exactly what it sounds like. Creating accurate visualizations of real world objects by simulating the underlying physical equations. While doing so, a number of shortcuts is taken to speed up computation time. We do not need very accurate results, we need results that **look accurate**.

2 Mass Spring Systems

Mass Spring Systems are a simple way of simulating deformation, and they are often used for hair, clothes etc. The main steps of employing mass spring systems are:

1. Discretizing the object into mass points and connecting them with springs
2. Setting up the equations of motion
3. Discretizing the equations of motion in time
4. Determining internal and external forces acting on mass points
5. Solving the equations numerically

The location and mass distribution of the discretized points has a large influence on the accuracy of the simulation. The simplest approximation is just giving all points equal mass, but better methods exist. Each point has a position $\boldsymbol{x_i}$, a velocity $\boldsymbol{v_i}$ and a mass $\boldsymbol{m_i}$. The points are then connected with springs. The spring topology also heaviliy influences the accuracy, and certain heuristics exist. Each spring stores a stiffness k_q, rest length L_q and current length l_q.

2.1 External Forces f^{Ext}

External forces include gravity ($f_i^G = m_i \begin{bmatrix} 0 & 0 & 9.81 \end{bmatrix}^T m/s^2$), friction and collisions.

2.2 Internal Forces f^{Int}

The internal forces are the forces exerted by the springs, as well as (viscous) damping. Spring force is given by Hooke's law:

$$f = k(L - l) \tag{1}$$

or in 3D

$$\boldsymbol{f_{ij}} = k(L - \|\boldsymbol{x_i} - \boldsymbol{x_j}\|)\frac{\boldsymbol{x_i} - \boldsymbol{x_j}}{\|\boldsymbol{x_i} - \boldsymbol{x_j}\|} \tag{2}$$

for multiple springs connected to a single mass point, just sum up all the individual spring forces.

2.3 Spring Topologies and Parameters

In general springs are connected in three different ways. Structural springs connect neighboring points and oppose stretching; diagonal strings oppose shearing and interleaving springs oppose bending. For cloth or other objects that need to be bent, springs that skip multiple points can be used.

The stiffness for the springs can be calculated with existing heuristics or be set in such a way that they accurately represent known deformations. Usually manual tuning is required.

2.4 Equations of Motion

As the mass of the points, as well as the forces acting on them is known, we can simulate them using Newton's laws of motion. From Newton's second law follows

$$m_i \frac{d^2 \boldsymbol{x_i}(t)}{dt^2} = \boldsymbol{f_i^{Int}}(t) + \boldsymbol{f_i^{Ext}}(t) \tag{3}$$

with which we can calculate the acceleration and update the velocity of the mass point. Equation (3) is a first order ordinary differential equation (ODE), which we can solve and the numerically integrate. For the integration step we differentiate between explicit methods (where all required quantities are known) and implicit methods (where we need to solve a system of equations).

2.5 Numerical Integration Methods

As we actually need two integrations for our systems (one for $\boldsymbol{v}$ and one for $\boldsymbol{x}$) we need two seperate integration steps. For this we simply introduce a variable for velociy and integrate it to get the new position.

All integration methods are derived from Taylor series expansion, and some use multiple Taylor series or more parts of it to reach higher accuracy. As a reminder, the Taylor series for $y(x)$ about development point t is

$$y(x) = \sum_{n=0}^{\infty} \frac{y^{(n)}(t)}{n!}(x-t)^n \tag{4}$$

and if we evaluate this at point $t+h$ we get

$$y(t+h) = y(t) + y'(t)h + \frac{y''(t)}{2!}h^2 + \frac{y'''(t)}{3!}h^3 + ... \tag{5}$$

From that we can derive our numerical integration methods and their accuracy.

Euler Method

Forward Euler is the simplest integration method. It is derived by using the first two summands of the Taylor series, giving us the following equation

$$y(t+h) = y(t) + y'(t)h + \mathcal{O}(h^2) \tag{6}$$

with our error being a function in the realm of $\mathcal{O}(h^2)$.

The following calculations are done for the Forward Euler:

- $\boldsymbol{x}(t+h) = \boldsymbol{x}(t) + h \cdot \boldsymbol{v}(t)$
- $\boldsymbol{f}(t) = \boldsymbol{f}^{Int}(t) + \boldsymbol{f}^{Ext}(t)$
- $\boldsymbol{a}(t) = \frac{1}{m}(\boldsymbol{f}(t) - \gamma \boldsymbol{v}(t))$ (where γ is the damping factor)
- $\boldsymbol{v}(t+h) = \boldsymbol{v}(t) + h \cdot \boldsymbol{a}(t)$

In order to improve behaviour of the Euler method, we can either directly reuse new positions

$$\boldsymbol{x}(t+h) = \boldsymbol{x}(t) + h\boldsymbol{v}(t) \tag{7}$$
$$\boldsymbol{v}(t+h) = \boldsymbol{v}(t) + h\boldsymbol{a}(\boldsymbol{x}(t+h), \boldsymbol{v}(t)) \tag{8}$$

or directly reuse the new velocities

$$\boldsymbol{v}(t+h) = \boldsymbol{v}(t) + h\boldsymbol{a}(\boldsymbol{x}(t), \boldsymbol{v}(t)) \tag{9}$$
$$\boldsymbol{x}(t+h) = \boldsymbol{x}(t) + h\boldsymbol{v}(t+h) \tag{10}$$

Using these improvements is known as the Symplectic Euler.

Heun's Method

This integrator uses more summands in its approximation:

$$y(t+h) = y(t) + hy'(t) + \frac{h^2}{2}y''(t) + \mathcal{O}(h^3) \tag{11}$$

As we don't know the second derivative, we approximate it with forward differences:

$$y''(t) = \frac{y'(t+h) - y'(t)}{h} + \mathcal{O}(h) \tag{12}$$

By plugging Equation (12) into Equation (11) we get

$$y(t+h) \approx y(t) + hy'(t) + \frac{h^2}{2}(\frac{y'(t+h) - y'(t)}{h}) \tag{13}$$
$$\approx y(t) + hy'(t) + \frac{h}{2}y'(t+h) - \frac{h}{2}y'(t) \tag{14}$$
$$\approx y(t) + \frac{h}{2}(y'(t) + y'(t+h)) \tag{15}$$

As we don't know $y'(t+h)$ yet when we evaluate this, we approximate it with a regular Euler step.

Heun's method uses the average of the current velocity and the next velocity for the update step

The implementation steps for Heun's method are the following

- $\boldsymbol{a}(t) = \frac{1}{m}(\boldsymbol{f}(t) - \gamma \boldsymbol{v}(t))$
- $\tilde{\boldsymbol{v}}(t+h) = \boldsymbol{v}(t) + h\boldsymbol{a}(t)$
- $\tilde{\boldsymbol{x}}(t+h) = \boldsymbol{x}(t) + h\tilde{\boldsymbol{v}}(t)$
- $\tilde{\boldsymbol{a}}(t+h) = \frac{1}{m}(\tilde{\boldsymbol{f}}(t+h) - \gamma \tilde{\boldsymbol{v}}(t+h))$
- $\boldsymbol{x}(t+h) = \boldsymbol{x}(t) + h\frac{\boldsymbol{v}(t)+\tilde{\boldsymbol{v}}(t+h)}{2}$
- $\boldsymbol{v}(t+h) = \boldsymbol{v}(t) + h\frac{\boldsymbol{a}(t)+\tilde{\boldsymbol{v}}(t+h)}{2}$

Midpoint Method

Heun's method averages $y'(t)$ and $y'(t+h)$. The midpoint method only determinse y' at $t+h/2$, giving us the same accuracy with less work. We approximate $y'(t+h/2)$ with an Euler step.

For our mass spring use case we need to approximate both $\tilde{\boldsymbol{v}}$ and $\tilde{\boldsymbol{a}}$ (and thus $\tilde{\boldsymbol{x}}$) for $t+h/2$.

Runge-Kutta Methods

These are more general cases of the Midpoint method. They increase accuracy by performing intermediate steps. The most common variant is RK4. Due to the increased accuracy, larger timesteps can be used, but this increases computational overhead.

RK4 works as follows:

$$k_1 = f(t, y(t)) \tag{16}$$
$$k_2 = f(t+h/2, y(t) + h/2k_1) \tag{17}$$
$$k_3 = f(t+h/2, y(t) + h/2k_2) \tag{18}$$
$$k_4 = f(t, y(t) + hk_3) \tag{19}$$

The value for $y(t+h)$ is the calculated using a weighted average

$$y(t+h) = y(t) + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4) \tag{20}$$

Implementation of RK4 (or any RK method) should be pretty clear.

Verlet Method

For this method we start with two Taylor series expansions:

$$y(t+h) = y(t) + y'(t)h + \frac{y''(t)}{2!}h^2 + \frac{y'''(t)}{3!}h^3 + \mathcal{O}(h^4) \tag{21}$$

$$y(t-h) = y(t) - y'(t)h + \frac{y''(t)}{2!}h^2 - \frac{y'''(t)}{3!}h^3 + \mathcal{O}(h^4) \tag{22}$$

then, by summing them up we get

$$y(t+h) = 2y(t) - y(t-h) + h^2 y''(t) + \mathcal{O}(h^4) \tag{23}$$

As you can see the velocity terms cancels out, leading to the typical assumption that the force term is independent of velocity (no damping). The Verlet method is a symplectic integrator (it has nice energy conservation properties in Hamiltonian systems) and is also time reversible.

The velocity term here is never explicitly calculated. If we do need it (e.g. for kinetic energy) we need to approximate it either via backward or central differences.

Leapfrog Method

The main goal of this method is to improve the velocity estimate. It is similar to the Verlet method but from the Taylor series for velocity:

$$\boldsymbol{v}(t+h) = \boldsymbol{v}(t) + \boldsymbol{v}'(t)h + \frac{\boldsymbol{v}''(t)}{2!}h^2 + \frac{\boldsymbol{v}'''(t)}{3!}h^3 + \mathcal{O}(h^4) \tag{24}$$

$$\boldsymbol{v}(t-h) = \boldsymbol{v}(t) - \boldsymbol{v}'(t)h + \frac{\boldsymbol{v}''(t)}{2!}h^2 - \frac{\boldsymbol{v}'''(t)}{3!}h^3 + \mathcal{O}(h^4) \tag{25}$$

and by subtracting we get

$$\boldsymbol{v}(t+h) = \boldsymbol{v}(t-h) + 2\boldsymbol{v}'(t)h + \mathcal{O}(h^3) \tag{26}$$

The Leapfrog method then employs shifted updates, with $\boldsymbol{v}$ being updated for $t + h/2$ and $\boldsymbol{x}$ and $\boldsymbol{a}$ being updated for t. This gives us an error of only $\mathcal{O}(h^3)$ with only one force evaluation. We need to initialize $\boldsymbol{v}(t_{1/2})$ with e.g. an Euler step.

Implicit Euler

The Implicit Euler formulates the update equation like as follows

$$y_{n+1} = y_n + h \cdot f(t_{n+1}, y_{n+1}) \tag{27}$$

Due to this, we have to solve for an unknown in every update step. One option is the **fixed-point iteration**

$$y_{n+1}^{(k+1)} = y_n + h \cdot f(t_{n+1}, y_{n+1}^{(k)}) \tag{28}$$

which requires initialization such as $y_{n+1}^{(0)} = y_n$ or $y_{n+1}^{(0)} = y_n + h \cdot f(t_n, y_n)$.

Another option would be **Newton's Method**.

2.6 Test Equation

We can study stability behaviour of a solver via a linear test equation (Dahlquist's equation):

$$y'(t) = \lambda \cdot y(t) \tag{29}$$

which gives us the following solution for $y(t)$ (assuming $\lambda \in \mathbb{R}$)

$$y(t) = e^{\lambda t} y_0 \tag{30}$$

An example for the forward Euler: The update step is

$$y_{n+1} = y_n + h \cdot y'_n \tag{31}$$

by inserting the test equation we get

$$y_{n+1} = (1 + \lambda h) y_n \tag{32}$$

which with recursive evaluation gives us

$$y_{n+1} = (1 + \lambda h)^{n+1} y_0 \tag{33}$$

This shows that the forward Euler is unconditionally unstabel if $\lambda > 0$ and stable iff $-2 < h\lambda < 0$.

2.7 Collision Handling

When using mass spring systems, a simple form of collision detection and handling can be required. For this, penalty forces are used. In the penalty forces approach any penetration of an object into another creates a spring force separating both objects. This approach is only an approximation, but can be very useful due to its simplicity.

2.8 Problems of Mass Spring Systems

The behaviour of the system is topology dependent. Also, we do not have an explicit notion of volume preservation, which can result in systems collapsing on themselves. This is again dependent on their topology.

3 Solving PDEs with Finite Elements

Differential equations relate an unknown functions with its derivatives. They are typically employed to describe physical laws and phenomena, e.g. growth or decay.

3.1 Types of Differential Equations

Ordinary Differential Equations (ODEs)

These describe an unknown function only based on derivatives with respect to a single varible.

Partial Differential Equations (PDEs)

Describe an unknown function based on its derivatives with respect to multiple variables. The **order** n of a PDE is given by its highest derivative. PDEs are called linear if the coefficients of the function and the derivatives are independent of the latter (e.g. no multiplication between them).

Classification of Second Order PDEs

Physical phenomena are often modeled as second order linear PDEs of the form

$$a \cdot u_{xx} + b \cdot u_{xy} + c \cdot u_x + e \cdot u_y + k \cdot u = g(x, y) \tag{34}$$

Then, based on the coefficients we differentiate three classes:

- **Hyperbolic:** $b^2 - 4ac > 0$
- **Parabolic:** $b^2 - 4ac = 0$
- **Elliptic:** $b^2 - 4ac < 0$

3.2 Initial and Boundary Conditions

In order to find a specific solution to the PDE we need to have some value to start from. This can be either done by **initial conditions**, which specify a certain physical state at start time, or **boundary conditions**, which impose a state on the boundary $\delta\Omega$ of the solution domain Ω.

3.3 Solving PDEs

Unfortunately, closed form analytical solutions exist only for very simple problems. This makes numerical analysis in order to get an approximate solution necessary. Typical solution methods are **finite differences** and **finite elements**. Only the finite element method is described in the lecture slides.

3.4 Finite Element Method

The key idea is to divide the domain into many small elements and then approximate the true solution in each element via a set of simple equations. After the individual equations are solved, they are combined into a global system for the final solution for the given boundary conditions.

Piecewise Approximation

In order to make the numerical solving easier, we approximate the target function via a linear combination of basis functions. Given a set of discrete evaluations of the function f, we want to find a way to get the values between those points for a continuous evaluation. As linear interpolation is a very intuitive form of doing so, we want a simple way of calculating this. This is done via "hat" functions, which can be seen in Figure 1.

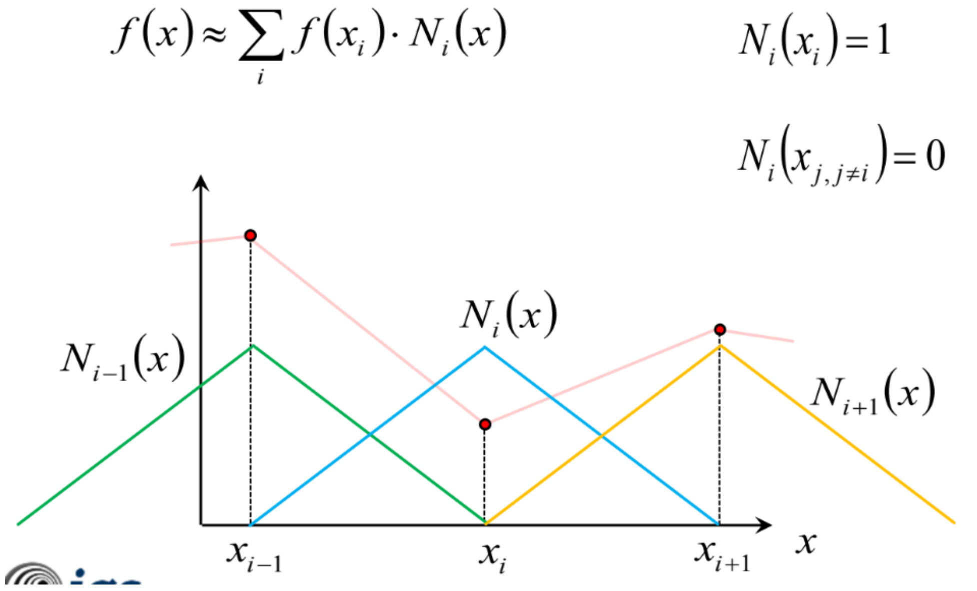


Figure 1: An example of interpolation via hat functions

For the finite element method we utilize the Ritz-Galerkin approach, which approximates the true solution with a linear combination of basis functions:

$$u \approx \sum_i u_i \cdot N_i \tag{35}$$

with unknowns u_i. We then select an arbitrary test function as linear combination of the basis functions

$$v \approx \sum_j N_j \tag{36}$$

which gives us the following approximations of the derivatives

$$\nabla u \approx \sum_i u_i \cdot \nabla N_i \qquad \nabla v \approx \sum_j \nabla N_j \tag{37}$$

Derivation of the Element Equation

We start from the 2D Poisson equation on a domain Ω with Dirichlet boundary conditions (a fixed value on the domain boundary)

$$\Delta u(x,y) = f(x,y), \qquad u(x,y) = 0 \text{ for } (x,y) \in \delta\Omega \tag{38}$$

Then we multiply with a sufficiently smooth (differentiable twice) test function $v(x,y)$ with $v(x,y) = 0$ on boundary $\delta\Omega$.

$$\Delta u(x,y)v(x,y) = f(x,y)v(x,y) \tag{39}$$

This we integrate over the domain

$$\int_\Omega \Delta u(x,y)v(x,y)d\Omega = \int_\Omega f(x,y)v(x,y)d\Omega \tag{40}$$

Then we derive the **weak form** of the problem by applying the Green-Gauss theorem (similar to integrating by parts)

$$\int_\Omega \nabla u(x,y)\nabla v(x,y)d\Omega = \int_\Omega f(x,y)v(x,y)d\Omega \tag{41}$$

After inserting the Ritz-Galerkin approxmation into the weak form we get

$$\int_\Omega \sum_i u_i \cdot \nabla N_i \cdot \sum_j \nabla N_j d\Omega = \int_\Omega f \cdot \sum_j N_j d\Omega \tag{42}$$

$$\sum_i u_i \int_\Omega \nabla N_i \cdot \sum_j \nabla N_j d\Omega = \int_\Omega f \cdot \sum_j N_j d\Omega \tag{43}$$

and by splitting this into a set of single equations we get

$$\sum_i u_i \int_\Omega \nabla N_i \cdot \nabla N_j d\Omega = \int_\Omega f \cdot N_j d\Omega \qquad \forall j = 1, ..., n \tag{44}$$

We can write this system of equations as a matrix, which is symmetric, positive definite and banded. The so called **stiffness matrix** K can be solved with an iterative solver, e.g. conjugate gradient method. Due to the sparsity of the stiffness matrix it is easiest assembled element-wise.

Linear Triangulation

In 2D we want our linear tringular elements to be composed of three nodes with three linear basis functions:

$$N_i(x,y) = c_{i,0} + c_{i,1}x + c_{i,2}y \qquad i = 1, 2, 3 \tag{45}$$

It is important to note that these basis functions have local support, ie. they are only non-zero for adjacent points.

Assembling the Stiffness Matrix

We want to evaluate the following equations

$$K_{ij} = \int_\Omega \nabla N_i \cdot \nabla N_j d\Omega \tag{46}$$

$$K_{ij} = \int_\Omega (\frac{\partial N_i}{\partial x}, \frac{\partial N_i}{\partial y}) \cdot (\frac{\partial N_j}{\partial x}, \frac{\partial N_j}{\partial y}) dxdy \tag{47}$$

$$K_{ij} = \int_\Omega \frac{\partial N_i}{\partial x} \cdot \frac{\partial N_j}{\partial x} + \frac{\partial N_i}{\partial y} \cdot \frac{\partial N_j}{\partial y} dxdy \tag{48}$$

The basis function N_i extends over all elements adjacent to node i and is zero outside of Ω_i. $K_ij \neq 0$ if there exists an element (triangle) containing nodes i and j.

Right-Hand Side

We assemble the right hand side with similar assumptions as before:

$$f_j = \sum_{\text{adjacent}} \int_{\Omega_e} f \cdot N_j d\Omega_e \tag{49}$$

We can approximate the integral with a one point quadrature

$$\int_{\Omega_e} f \cdot N_j d\Omega_e \approx A_e \cdot f(x_c, y_c) \cdot N_j(x_c, y_c) \tag{50}$$

at barycenter (x_c, y_c) in order to save computation time.

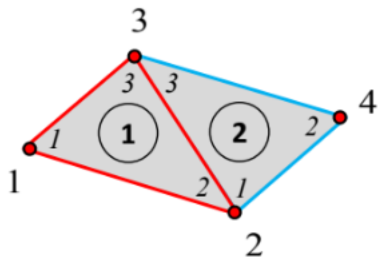
Figure 2 shows an example of a system setup.

4 Solving Systems of Equations

For calculating numerical solutions for our FEM systems we need an efficient way of solving systems of equations. These equations can be written as $\boldsymbol{Ax} = \boldsymbol{b}$. We could solve this by inverting the coefficient matrix $\boldsymbol{A}$, but that takes too long ($\mathcal{O}(n^3)$). To this end we use iterative solvers, which work more efficiently.

4.1 Jacobi Method

The Jacobi method only converges for diagonally dominant system matrices, but this is usually a given in physical applications. The basic approach of it is rearranging the



$$\begin{bmatrix} K_{11}^{(1)} & K_{12}^{(1)} & K_{13}^{(1)} & 0 \\ K_{21}^{(1)} & K_{22}^{(1)} + K_{11}^{(2)} & K_{23}^{(1)} + K_{13}^{(2)} & K_{12}^{(2)} \\ K_{31}^{(1)} & K_{32}^{(1)} + K_{31}^{(2)} & K_{33}^{(1)} + K_{33}^{(2)} & K_{32}^{(2)} \\ 0 & K_{21}^{(2)} & K_{23}^{(2)} & K_{22}^{(2)} \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \end{bmatrix} = \begin{bmatrix} f_1^{(1)} \\ f_2^{(1)} + f_1^{(2)} \\ f_3^{(1)} + f_3^{(2)} \\ f_2^{(2)} \end{bmatrix}$$

$$\mathbf{K} \cdot \mathbf{u} = \mathbf{f}$$

Figure 2: Example setup

individual equations:

$$\sum_{j=1}^{n} a_{ij} x_j = b_i \tag{51}$$

$$x_i = \frac{1}{a_{ii}} (b_i - \sum_{j=1, j \neq i}^{n} a_{ij} x_j) \tag{52}$$

From this we derive our update rule

$$x_i^{(k+1)} = \frac{1}{a_{ii}} (b_i - \sum_{j=1, j \neq i}^{n} a_{ij} x_j^{(k)}) \tag{53}$$

or in matrix notation

$$\boldsymbol{x}^{(k+1)} = \boldsymbol{D}^{-1} (\boldsymbol{b} - (\boldsymbol{A} - \boldsymbol{D}) \boldsymbol{x}^{(k)}) \tag{54}$$

4.2 Solving an Equivalent Problem

This method is especially effective for sparse matrices, and requires that the matrix $\boldsymbol{A}$ be symmetric fulfill

$$\boldsymbol{x}^T \boldsymbol{A} \boldsymbol{x} > 0 \qquad \forall \boldsymbol{x} \neq \boldsymbol{0} \tag{55}$$

The method is then based on solving an equivalent problem, minimizing the function

$$f(\boldsymbol{x}) = \frac{1}{2}\boldsymbol{x}^T\boldsymbol{A}\boldsymbol{x} - \boldsymbol{x}^T\boldsymbol{b} \tag{56}$$

The first and second derivatives of the function are then

$$\nabla f(\boldsymbol{x}) = \boldsymbol{A}\boldsymbol{x} - \boldsymbol{b}, \qquad \nabla^2 f(\boldsymbol{x}) = \boldsymbol{A} \tag{57}$$

The function f is a parabola with a single global minimum. This global minimum contains our solution, because there $\boldsymbol{A}\boldsymbol{x}^* - \boldsymbol{b} = 0$, so finding x^* equals finding our solution. We try to find the solution via line searching.

Our iteration step is

$$\boldsymbol{x}^{(k+1)} = \boldsymbol{x}^{(k)} + \alpha^{(k)}\boldsymbol{p}^{(k)} \tag{58}$$

with initial guess $\boldsymbol{x}^{(0)}$, search direction $\boldsymbol{p}^{(k)}$ and step length $\alpha^{(k)}$. The question now is how to find the most efficient search direction and step length. We approximate the error in each step $\boldsymbol{e}^{(k)} = \boldsymbol{x}^* - \boldsymbol{x}^{(k)}$ with the residual of our approximation of the right-hand side

$$\boldsymbol{r}^{(k)} = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}^{(k)} \qquad \to \boldsymbol{r}^{(k)} = -\nabla f(\boldsymbol{x}^{(k)}) \tag{59}$$

One can also note that $\boldsymbol{r}^{(k)} = \boldsymbol{A}\boldsymbol{e}^{(k)}$.

Steepest Gradient Method

The gradient determines the direction of greatest increase, so we step into the direction of the negative gradient

$$\boldsymbol{x}^{(k+1)} = \boldsymbol{x}^{(k)} + \alpha^{(k)}\boldsymbol{p}^{(k)} \tag{60}$$

$$\boldsymbol{p}^{(k)} = -\nabla f(\boldsymbol{x}^{(k)}) = \boldsymbol{r}^{(k)} = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}^{(k)} \tag{61}$$

We find the step size α by minimzing the next function value, which gives us the following formula for α

$$\alpha^{(k)} = \frac{(\boldsymbol{r}^{(k)})^T\boldsymbol{r}^{(k)}}{(\boldsymbol{r}^{(k)})^T\boldsymbol{A}\boldsymbol{r}^{(k)}} \tag{62}$$

If the matrix $\boldsymbol{A}$ is ill-conditioned, this method is slow converging. A better way of finding a search direction is needed.

Conjugate Gradient Method

The key idea here is to select a number of $\boldsymbol{A}$-conjugate directions in which to go. Two non-zero vectors are conjugate with respect to a matrix $\boldsymbol{A}$ if $\boldsymbol{p}_i^T\boldsymbol{A}\boldsymbol{p}_j = 0$ with $\boldsymbol{p}_i \neq \boldsymbol{p}_j$. Instead of picking a set of vectors in the beginning, we pick a new one each iteration.

$$\boldsymbol{p}^{(k+1)} = \boldsymbol{r}^{(k+1)} + \beta^{(k+1)}\boldsymbol{p}^{(k)}, \qquad \boldsymbol{p}^{(0)} = \boldsymbol{r}^{(0)} \tag{63}$$

Obtaining β works as follows:

$$(\boldsymbol{p}^{(k+1)})^T \boldsymbol{A}\boldsymbol{p}^{(k)} = 0 \tag{64}$$

$$(\boldsymbol{r}^{(k+1)} + \beta^{(k+1)}\boldsymbol{p}^{(k)})^T \boldsymbol{A}\boldsymbol{p}^{(k)} = 0 \tag{65}$$

$$(\boldsymbol{r}^{(k+1)})^T \boldsymbol{A}\boldsymbol{p}^{(k)} + \beta^{(k+1)}(\boldsymbol{p}^{(k)})^T \boldsymbol{A}\boldsymbol{p}^{(k)} = 0 \tag{66}$$

$$\beta^{(k+1)} = -\frac{(\boldsymbol{r}^{(k+1)})^T \boldsymbol{A}\boldsymbol{p}^{(k)}}{(\boldsymbol{p}^{(k)})^T \boldsymbol{A}\boldsymbol{p}^{(k)}} \tag{67}$$

From the update method

$$\boldsymbol{x}^{(k+1)} = \boldsymbol{x}^{(k)} + \alpha^{(k)}\boldsymbol{p}^{(k)} \tag{68}$$

we get the equation for updating the residual

$$\boldsymbol{A}\boldsymbol{x}^{(k+1)} - \boldsymbol{b} = \boldsymbol{A}\boldsymbol{x}^{(k)} - \boldsymbol{b} + \alpha^{(k)}\boldsymbol{A}\boldsymbol{p}^{(k)} \tag{69}$$

$$\boldsymbol{r}^{(k+1)} = \boldsymbol{r}^{(k)} - \alpha^{(k)}\boldsymbol{A}\boldsymbol{p}^{(k)} \tag{70}$$

$$\boldsymbol{A}\boldsymbol{p}^{(k)} = \frac{\boldsymbol{r}^{(k)} - \boldsymbol{r}^{(k+1)}}{\alpha^{(k)}} \tag{71}$$

Equation (71) gives us another way of calculating β

$$\beta^{(k+1)} = -\frac{(\boldsymbol{r}^{(k+1)})^T \boldsymbol{A}\boldsymbol{p}^{(k)}}{(\boldsymbol{p}^{(k)})^T \boldsymbol{A}\boldsymbol{p}^{(k)}} = \frac{(\boldsymbol{r}^{(k+1)})^T \boldsymbol{r}^{(k+1)}}{(\boldsymbol{r}^{(k)})^T \boldsymbol{r}^{(k)}} \tag{72}$$

The step length α is determined by

$$(\boldsymbol{r}^{(k+1)})^T \boldsymbol{r}^{(k)} = 0 \tag{73}$$

$$(\boldsymbol{r}^{(k)} - \alpha^{(k)}\boldsymbol{A}\boldsymbol{p}^{(k)})^T \boldsymbol{r}^{(k)} = 0 \tag{74}$$

$$(\boldsymbol{r}^{(k)})^T \boldsymbol{r}^{(k)} - \alpha^{(k)}(\boldsymbol{A}\boldsymbol{p}^{(k)})^T \boldsymbol{r}^{(k)} = 0 \tag{75}$$

$$\alpha^{(k)} = \frac{(\boldsymbol{r}^{(k)})^T \boldsymbol{r}^{(k)}}{(\boldsymbol{p}^{(k)})^T \boldsymbol{A}^T \boldsymbol{p}^{(k)}} \tag{76}$$

Summary The iteration variables are calculated as follows:

$$\alpha^{(k)} = \frac{(\boldsymbol{r}^{(k)})^T \boldsymbol{r}^{(k)}}{(\boldsymbol{p}^{(k)})^T \boldsymbol{A}\boldsymbol{p}^{(k)}} \tag{77}$$

$$\boldsymbol{x}^{(k+1)} = \boldsymbol{x}^{(k)} + \alpha^{(k)}\boldsymbol{p}^{(k)} \tag{78}$$

$$\boldsymbol{r}^{(k+1)} = \boldsymbol{r}^{(k)} - \alpha^{(k)}\boldsymbol{A}\boldsymbol{p}^{(k)} \tag{79}$$

$$\beta^{(k+1)} = \frac{(\boldsymbol{r}^{(k+1)})^T \boldsymbol{r}^{(k+1)}}{(\boldsymbol{r}^{(k)})^T \boldsymbol{r}^{(k)}} \tag{80}$$

$$\boldsymbol{p}^{(k+1)} = \boldsymbol{r}^{(k+1)} + \beta^{(k+1)}\boldsymbol{p}^{(k)} \tag{81}$$

We are guaranteed to find a solution after n steps for an $n \times n$ matrix, but can get reasonable approximations before that. In order to increase performance, one can use a precondition matrix $\boldsymbol{M}^{-1}\boldsymbol{A}\boldsymbol{x} = \boldsymbol{M}^{-1}\boldsymbol{b}$. The simplest choice is using the diagonal entries of $\boldsymbol{A}$ (Jacobi preconditioner).

4.3 FEM summary

The required steps for FEM are

1. Discretization of the domain
2. Construction of approximate solution over elements
3. Assembling elements into a global system
4. Imposing boundary conditions
5. Numerical solution of resulting equation system
6. Post-processing and visualization of results

FEM is widely applicable, easy to use for complex geometries, easy to adapt for better approximation and it is easy to impose boundary conditions. However, the solution is only an approximate, careful setup is required, the amount of pre-processing required can become large, and parallelization is not straight forward (due to the interdependencies between elements).

5 Fluid Simulation

When trying to simulate fluids and flow, the two main approaches are particle based (where you simulate individual particles) and grid based (where you simulate what happens at fixed positions in the grid and interpolate). While there are others, the most important properties in the simulation are fluid velocity and fluid pressure.

Usually we simplify by assuming constant density (e.g. water is nearly incompressible) and constant viscosity.

5.1 Navier-Stokes Equation

The equation is based on the conservation of momentum

$$\frac{\partial \boldsymbol{v}}{\partial t} + (\boldsymbol{v} \cdot \nabla)\boldsymbol{v} = -\frac{1}{\rho}\nabla p + \upsilon \Delta \boldsymbol{v} + \boldsymbol{f} \tag{82}$$

with a supporting equation derived from the conservation of mass

$$\nabla \cdot \boldsymbol{v} = 0 \tag{83}$$

The symbols are as follows:

- Velocity $\boldsymbol{v}$
- Time t
- Density ρ
- Pressure p
- Kinematic viscosity υ
- "Force" term (e.g. gravity) $\boldsymbol{f}$

On the left-hand side we have advection (self advection in this case), the normal advection "operator" is

$$\frac{d}{dt} = \frac{\partial}{\partial t} + \boldsymbol{v} \cdot \nabla \tag{84}$$

and if applied to any quantity it is just multiplied in (regular distribution over multiplication). Advection is the rate of change for a property of a fluid due to the particles moving. An example would be if you measure the temperature in a faucet and move the temperature control, the temperature at a given point will change because the "new" water coming in has a different temperature than the old.

The right-hand side consists of pressure induced flow $-\frac{1}{\rho}\nabla p$. The direction and magnitude is given by the gradient of the pressure field, and because fluid flows from higher pressures to lower pressures, we need to reverse the direction.

The next summand is viscosity based diffusion. It is given by the formula $\boldsymbol{f}_{viscosity} = \mu\nabla\boldsymbol{v}V$. The volume term is due to the force being an integration over a fluid parcel. Basically we assume constant diffusion over a given volume, and instead of integrating we multiply with the volume. The symbol μ stands for the dynamic viscosity parameter, and if we divide that by the mass m we get the kinematic viscosity υ (which is dynamic viscosity combined with density).

The $\boldsymbol{f}$ term is a bit confusingly named, as it stands for acceleration and not a force. This is because the entire equation is divided by the mass m (this is also where $\frac{1}{\rho}$ comes from in the pressure term).

Our support equation $\nabla \cdot \boldsymbol{v}$ just means that we have neither sources nor sinks in our flow, i.e. no flow is lost. This only holds for Newtonian fluids.

5.2 Solving the Navier-Stokes Equation

The Navier-Stokes equation gives us a set of PDEs, which in general cannot be solved analytically. For 3D flow we have four equations (momentum and continuity) for four unknowns (3D velocity vector & pressure). In order to solve this system we use a numerical approach, by discretizing time and space and solving for each time step. As the Navier-Stokes equation is fairly complex, we split it into multiple steps to solve it more efficiently:

$$\frac{\partial \boldsymbol{v}}{\partial t} + (\boldsymbol{v} \cdot \nabla)\boldsymbol{v} = -\frac{1}{\rho}\nabla p + v\Delta \boldsymbol{v} + \boldsymbol{f} \tag{85}$$

$$\frac{\partial \boldsymbol{v}}{\partial t} = -(\boldsymbol{v} \cdot \nabla)\boldsymbol{v} \tag{86}$$

$$\frac{\partial \boldsymbol{v}}{\partial t} = \boldsymbol{f} \tag{87}$$

$$\frac{\partial \boldsymbol{v}}{\partial t} = v\Delta \boldsymbol{v} \tag{88}$$

$$\frac{\partial \boldsymbol{v}}{\partial t} = -\frac{1}{\rho}\nabla p \tag{89}$$

After splitting, we solve each step individually and add them all up. This reduces accuracy but **greatly** increases simulation speed. As all the individual summands are differentials, and we are finally interested in a velocity field, we numerically integrate the results by multiplying with Δt.

As we want a suitable solution to be divergence free, we need to solve Poisson's equation in the last step. Also, as the numeric approach is already introducing some errors, diffusion is usually dropped. It is generally very low, and the numerical solver introduces a larger error anyways.

5.3 Spatial Discretization

As a scheme for solving the differential equations we use the finite differences method. This means we create a grid and store all relevant properties at the grid points. The derivatives are then calculated by taking the difference between neighboring grid points and dividing the difference by their distance. When using the forward difference we reach $\mathcal{O}(h)$ and using central differences (approximating the derivative in a point using the two neighbors) gives us $\mathcal{O}(h^2)$.

Due to the fact that using central differences results in an odd-even-decoupling and instabilities, staggered grids are often used. This means that we store a certain quantity at the centers of the cell (e.g. pressure), and others at the borders (e.g. velocities). This means that a recalculation of indices (might) be necessary.

We can express higher order derivatives through derivatives of lower order, by repeatedly calculating finite differences.

Derivatives on a Staggered Grid

$$\boldsymbol{v}_{i,j} \approx \begin{bmatrix} \frac{v_x(i,j) + v_x(i+1,j)}{2} & \frac{v_y(i,j) + v_y(i,j+1)}{2} \end{bmatrix}^T \tag{90}$$

Now if we want to express divergence $\nabla \cdot \boldsymbol{v} = \frac{\partial v_x}{\partial x} + \frac{\partial v_y}{\partial y}$ we employ finite differences and get

$$(\frac{\partial v_x}{\partial x} + \frac{\partial v_y}{\partial y})_{(i,j)} \approx \frac{v_x(i+1,j) - v_x(i,j)}{h} + \frac{v_y(i,j+1) - v_y(i,j)}{h} \tag{91}$$

for the derivation at the cell center (due to the staggered grid)

For the Laplace operator (in 2D), which is used for solving Poisson's equation for the pressure field we have

$$\begin{aligned}
(\frac{\partial^2 p}{\partial x^2})_{(i,j)} &= (\frac{\partial}{\partial x}(\frac{\partial p}{\partial x})_{(i,j)} && (92)\\
&\approx \frac{\frac{\partial}{\partial x}p(i+1,j) - \frac{\partial}{\partial x}p(i,j)}{h} && (93)\\
&\approx \frac{\frac{p(i+1,j)-p(i,j)}{h} - \frac{p(i,j)-p(i-1,j)}{h}}{h} && (94)\\
&\approx \frac{p(i+1,j) - 2p(i,j) + p(i-1,j)}{h^2} && (95)
\end{aligned}$$

for the x component, and something very similar for the y component (with j moving instead of i). Combining the two gives us

$$\Delta p_{(i,j)} = (\frac{\partial^2 p}{\partial x^2} + \frac{\partial^2 p}{\partial y^2})_{(i,j)} \approx \frac{p(i+1,j) + p(i-1,j) + p(i,j+1) + p(i,j-1) - 4p(i,j)}{h^2} \tag{96}$$

This can be calculated efficiently using convolution in frequency space and Fast Fourier Transformation (FFT).

5.4 The Individual Solutions

We could solve the self advection via a simple forward Euler step, but that method is unstable. Instead we switch to a Semi-Lagrangian viewpoint and go in the reverse direction of the velocity in order to find the advected velocity. We calculate the location where the current particle comes from by going "back" in time, and inerpolating the wanted property (velocity in our case) from the neighboring cells of the result location. When doing this we have to make sure we do not leave our grid.

External forces are simply added, as they are known.

The diffusion term would require solving the heat equation $\frac{\partial \boldsymbol{v}}{\partial t} = v\Delta\boldsymbol{v}$. We could do this by using the (forward) Euler method, but again this would be unstable. Instead we could use the Implicit (backward) Euler and solving the system of equations arising from discretization with an iterative method. Due to numerical dissipation the diffusion term is often dropped.

Solving for pressure is a bit complicated, as the new velocities have to be divergence free. We start from performing a forward Euler update

$$\boldsymbol{v}^{n+1} = \tilde{\boldsymbol{v}}_{diff} - \Delta t(\frac{1}{\rho}\nabla p^n) \tag{97}$$

Then we plug this into the mass conservation equation for the next step and get

$$\nabla \cdot \boldsymbol{v}^{n+1} = 0 \tag{98}$$

$$\nabla \cdot (\tilde{\boldsymbol{v}}_{diff} - \Delta t(\frac{1}{\rho}\nabla p^n)) = 0 \tag{99}$$

$$\nabla \cdot \tilde{\boldsymbol{v}}_{diff} = \nabla \cdot (\Delta t(\frac{1}{\rho}\nabla p^n)) \tag{100}$$

$$\nabla \cdot \tilde{\boldsymbol{v}}_{diff} = \frac{\Delta t}{\rho}\nabla \cdot (\nabla p^n) \tag{101}$$

$$\nabla \cdot \nabla p^n = \Delta p^n = \frac{\rho}{\Delta t}\nabla \cdot \tilde{\boldsymbol{v}}_{diff} \tag{102}$$

We then approximet the derivatives with finite differences (on both sides). This results in a system of linear equations $\boldsymbol{Ap} = \boldsymbol{d}$, with a coefficient matrix $\boldsymbol{A}$ (which is symmetric and sparse), our vector $\boldsymbol{p}$ containing the unknown pressures and the divergence terms $\boldsymbol{d}$ (multiplied with a constant). Each row in this equation system represents a single cell in our grid. We can solve this system using a component wise iterative solver, the main variants are the **Jacobi method** and the **Gauss-Seidel method**. These solvers use the same update formula

$$p^{n+1}(i,j) = \frac{(h^2 d_{i,j} + p^n(i+1,j) + p^n(i-1,j) + p^n(i,j+1) + p^n(i,j-1))}{4} \tag{103}$$

but the Gauss-Seidel method already uses the newly computed values if they are available. We could also use the conjugate gradient method or other solvers (multigrid solvers, which work hierarchically are often used in larger realistic problems).

In order to find a suitable stopping point for these iterative solvers, we chan check the **residual** and compare it to an accuracy threshold:

$$\boldsymbol{Ap} = \boldsymbol{d} \qquad \boldsymbol{r}^n = \boldsymbol{d} - \boldsymbol{Ap}^n \tag{104}$$

The solution is exact if $\|\boldsymbol{r}^n\| = 0$, and if $\|\boldsymbol{r}^n\|$ is below our threshold, we accept the solution. As the matrix $\boldsymbol{A}$ is sparse we try to avoid explicitly computing it. It is defined by a fixed set of bands, which arise from the second derivative of the pressure.

After finding a good enough approximation for the pressures we need to correct our velocities.

5.5 Boundary Conditions

Often we want to impose conditions on the velocity at the boundaries of our domain (e.g. for obstacles or hard boundaries). This can be done in different ways.

Dirichlet Boundary Condition

Dirichlet conditions directly prescribe the value of a function on the domain boundary (e.g. constant inflow at velocity inlet). For staggered grids, some values are not at the interface, and thus the next neighboring value has to be adjusted so the value at the interface is according to the boundary condition.

Neumann Boundary Conditions

These prescribe the value of the derivative on the domain boundary. For example setting the pressure at the boundary to the pressure *right next to the boundary* should make flow through the obstacle impossible.

5.6 Vorticity Confinement

For this we need the definition of curl, which goes as follows (in 3D):

$$\nabla \times \boldsymbol{v} = \text{curl } \boldsymbol{v} = \left(\tfrac{\partial}{\partial x}, \tfrac{\partial}{\partial y}, \tfrac{\partial}{\partial z}\right) \times \begin{pmatrix} v_x \\ v_y \\ v_z \end{pmatrix} \tag{105}$$

and for a 2D vector field

$$\nabla \times \boldsymbol{v} = \left(\tfrac{\partial}{\partial x}, \tfrac{\partial}{\partial y}\right) \times \begin{pmatrix} v_x \\ v_y \end{pmatrix} = \frac{\partial v_y}{\partial x} - \frac{\partial v_x}{\partial y} \tag{106}$$

Curl characterizes rotation at a point in a 3D field (vorticity), with the vectors pointing along the axis of rotation, with the magnitude being represented via the length.

For vorticity confinement we then determine unit vectors pointing towards the center of rotation (via the gradient of the curl field magnitude). The cross product of this unit vector with the curl gives a force amplifying rotation. This is needed as an unfortunate side effect of the numerical approach is the suppression of vorticity.

6 Rigid Bodies

We sometimes are only interested in non-deformable (or rigid) objects and their behaviour. These are a lot easier to simulate than deformable objects, as their shape does not change. This section details how to deal with them.

6.1 Rotations

Rotations of objects can be represented in multiple ways. We can use Euler angles, which are rotations given by three consecutive rotations around the elementary axes, Quaternions, which are rotations around an arbitrary vector, and rotation matrices. They each have their advantages and disadvantages.

Rotation Matrices

Rotation matrices are real, orthogonal 3×3 matrices, where each column represents a direction of the transformed basis. They have the following properties:

$$\boldsymbol{A}^{-1} = A^T \tag{107}$$

$$\boldsymbol{A}^T\boldsymbol{A} = \boldsymbol{A}\boldsymbol{A}^T = \boldsymbol{I} \tag{108}$$

$$\text{Det}(\boldsymbol{A}) = 1 \tag{109}$$

and concatenating (multiplying) two rotations yields a rotation. The length of vectors is also not changed due to rotation.

Quaternions

As rotation matrices need to be normalized (in order for their determinant to stay 1), they can create quite some computational overhead. We also need to ensure that the components are orthonormal, which also causes overhead. As an alternative representation of rotations we can use quaternions.

$$q = a + b \cdot i + c \cdot j + d \cdot k \qquad a, b, c, d \in \mathbb{R} \tag{110}$$

$$i^2 = j^2 = k^2 = i \cdot j \cdot k = -1 \tag{111}$$

The units have a few relations between them (e.g. $i \cdot j = k$) but I won't list them here. They should be easy to find however.

Quaternions can also be represented with vectors

$$\boldsymbol{q} = (a, b, c, d) = (w, x, y, z) = (w, \boldsymbol{v}) \tag{112}$$

The basic operations for quaternions are

$$\boldsymbol{q}_1 + \boldsymbol{q}_2 = (w_1, \boldsymbol{v}) + (w_2, \boldsymbol{v}_2) \tag{113}$$

$$\boldsymbol{q}_1 \cdot \boldsymbol{q}_2 = w_1 \cdot w_2 + \boldsymbol{v}_1 \cdot \boldsymbol{v}_2 \qquad \text{(dot product)} \tag{114}$$

$$\boldsymbol{q}^* = (w, -\boldsymbol{v}) \qquad \text{(conjugate)} \tag{115}$$

$$\|\boldsymbol{q}\| = \sqrt{\boldsymbol{q}^* \circ \boldsymbol{q}} = \sqrt{w^2 + \boldsymbol{v} \cdot \boldsymbol{v}} \tag{116}$$

$$\boldsymbol{q}^{-1} = \boldsymbol{q}^* / \|\boldsymbol{q}\|^2 \tag{117}$$

$$\boldsymbol{q}_1 \circ \boldsymbol{q}_2 = (w_1 + \boldsymbol{v}_1)(w_2 + \boldsymbol{v}_2) \qquad \text{(Hamilton product)} \tag{118}$$

There exist efficient conversions between rotation matrices and quaternions. This makes normalization and concatenation of rotations a lot easier.

6.2 Rigid Bodies

As the objects are undeformable, we can represent all their vertices by the center of mass x_c and a displacement vector r_i:

$$\boldsymbol{x}_i^t = \boldsymbol{x}^t + \boldsymbol{r}_i^t = (\boldsymbol{x}^0 + \boldsymbol{t}^t) + \boldsymbol{A}^t \cdot \boldsymbol{r}_0^t \tag{119}$$

where $\boldsymbol{A}^t$ is the current rotation matrix.

The velocity is then the derivative of the position

$$\boldsymbol{v}_i^t = \frac{d\boldsymbol{x}_i^t}{dt} = \dot{\boldsymbol{x}}_i^t = \dot{\boldsymbol{x}}_{CM}^t + \dot{\boldsymbol{A}}^t \cdot \boldsymbol{r}_i^0 + \boldsymbol{A}^t \cdot \dot{\boldsymbol{r}}_i^0 \tag{120}$$

And because the displacement vector is constant, the last summand is zero.

Momentum

Linear momentum $\boldsymbol{p} = m\boldsymbol{v}$ is a vector quantitiy pointing in the direction of velocity. With no external influences momentum is conserved.

Angular momentum is the rotational analog $\boldsymbol{L}_i = \boldsymbol{r}_i \times \boldsymbol{p}_i$.

Inertia

Similar to the mass for translation, we have the inertia tensor working against a torque applied. Analogously to linear momentum, the equation for angular momentum is $\boldsymbol{L} = \boldsymbol{I} \cdot \boldsymbol{\omega}$, with $\boldsymbol{I}$ being the inertia tensor and $\boldsymbol{\omega}$ being the angular velocity.

The inertia tensor is not easy to calculate. Fortunately, we only have to calculate it once (for rigid objects), and can rotate it along with the object

$$\boldsymbol{I} = \boldsymbol{A} \cdot \boldsymbol{I}^0 \cdot A^T \tag{121}$$

There also exists an orientation in body space, which causes off-diagonal components of the inertia tensor to vanish. The inverse is just as easily derived

$$\boldsymbol{I}^{-1} = (\boldsymbol{A} \cdot \boldsymbol{I}^0 \cdot \boldsymbol{A}^T)^{-1} = (\boldsymbol{A}^T)^{-1}(\boldsymbol{I}^0)^{-1}\boldsymbol{A}^{-1} = \boldsymbol{A} \cdot (\boldsymbol{I}^0)^{-1} \cdot \boldsymbol{A}^T \tag{122}$$

which means we only have to invert $\boldsymbol{I}$ once. We need the inverse inertia tensor as $\boldsymbol{\omega} = \boldsymbol{I}^{-1} \cdot \boldsymbol{L}$.

6.3 Collisions

In rigid body collisions we want the objects to change direction instantenously. For this we use the impulse

$$\boldsymbol{J} = \int \boldsymbol{f} dt \qquad \boldsymbol{J} = \boldsymbol{f}_{avg} \cdot \Delta t \tag{123}$$

For collisions we have the so called restitution coefficient, which tells us how much of the impulse is lost (without friction) during the collision. It's the ration of velocities before and after

$$\epsilon = -\frac{\boldsymbol{v}_{rel}^{t+}\boldsymbol{n}}{\boldsymbol{v}_{rel}^{t-}\boldsymbol{n}} \tag{124}$$

for "fully" elastic collisions this coefficient is close to 1.

From this we can calculate the impulse magnitude after the collision.

$$-\epsilon \cdot \boldsymbol{v}_{rel}^{t-}\boldsymbol{n} = \boldsymbol{v}_{rel}^{t+}\boldsymbol{n} \tag{125}$$
$$-\epsilon \cdot \boldsymbol{v}_{rel}^{t-}\boldsymbol{n} = (\boldsymbol{v}_{A}^{t+} - \boldsymbol{v}_{B}^{t+})\boldsymbol{n} \tag{126}$$
$$-\epsilon \cdot \boldsymbol{v}_{rel}^{t-}\boldsymbol{n} = (\boldsymbol{v}_{A}^{t-} + j\frac{\boldsymbol{n}}{m_a} - \boldsymbol{v}_{B}^{t+} + j\frac{\boldsymbol{n}}{m_b})\boldsymbol{n} \tag{127}$$
$$-\epsilon \cdot \boldsymbol{v}_{rel}^{t-}\boldsymbol{n} = \boldsymbol{v}_{A}^{t-}\boldsymbol{n} + j\boldsymbol{n} \cdot \frac{\boldsymbol{n}}{m_a} - \boldsymbol{v}_{B}^{t+}\boldsymbol{n} + j\boldsymbol{n} \cdot \frac{\boldsymbol{n}}{m_b} \tag{128}$$
$$-\epsilon \cdot \boldsymbol{v}_{rel}^{t-}\boldsymbol{n} = \boldsymbol{v}_{rel}^{t-}\boldsymbol{n} + j(\boldsymbol{n} \cdot \boldsymbol{n}(m_A^{-1} + m_B^{-1})) \tag{129}$$
$$j = \frac{-(1+\epsilon) \cdot \boldsymbol{v}_{rel}^{t-}\boldsymbol{n}}{\boldsymbol{n} \cdot \boldsymbol{n}(m_A^{-1} + m_B^{-1})} \tag{130}$$

This is only for the linear component of the impulse. We can do the same thing for the angular component as well, but I am too lazy to write those formulas now. The final velocity is a combination of the linear and angular components.

6.4 Resting Contacts

Resting contacts usually have to be handled separately to avoid instabilities. We require all forces to be determined at the same time, unless we want "wobbling" to happen. Resting contacts have the following constraints

- Contact forces ensure that the contact point does not accalerate toward the other body
- Contact force in normal direction must be repulsive to prevent bodies from sticking together
- Contact forces become zero when the objects move apart

Resting contacts is a hard problem.