universität wien

BACHELORARBEIT

ADDING CONSTRAINTS TO THE VIENNA PHYSICS ENGINE

Verfasserin ODER Verfasser

Julian Schneebaur

angestrebter akademischer Grad

Bachelor of Science (BSc)

Wien, 2023

Studienkennzahl lt. Studienblatt:	A 033 521
Fachrichtung:	Informatik - Allgemein
Betreuerin / Betreuer:	Univ.-Prof. Dipl.-Ing. Dr. Helmut Hlavacs

Contents

Abstract

Constraints are an essential part of real time rigid body simulations needed for applications like video games. A constraint is a condition on the state of the simulation that needs to be upheld by the engine at all times. For example, bodies are usually not supposed to overlap when colliding. This can be implemented by introducing a constraint that enforces that there must be no inter-penetration between bodies. Another important application of constraints are joints, which will be the main topic of this project. Joints can be used to model doors, swings or complex systems like ragdoll physics.

The Vienna Physics Engine is a simple C++ physics engine written by Prof. Helmut Hlavacs for usage in a variety of gaming related courses at the University of Vienna. It supports collision detection and resolution with friction for polytopes.

The aim of this project was to extend the engine, adding more constraints in the form of different joint types. A Sequential Impulses solver with Baumgarte stabilization was implemented. This approach enforces constraints on the velocity level by restricting the motion of the constrained objects. Illegal velocities can be corrected by applying constraint impulses. The solver works iteratively, making the algorithm and implementation straightforward and comparatively easy to understand.

The implemented constraints include a simple distance constraint in addition to four joint types: A ball-and-socket, hinge, slider and fixed joint.

1 Motivation

1.1 The Vienna Physics Engine

The Vienna Physics Engine (in short VPE) is a simple C++ real time physics engine developed by Professor Helmut Hlavacs. Its main purpose is to serve as an educational tool for various gaming related courses taught at the Faculty of Computer Science of the University of Vienna.

Feature-wise, the engine already supported collision detection and resolution for convex polytopes (the default shape is a cube), as well as friction and arbitrarily high cube stacking. For rendering the Vienna Vulkan Engine is used, although the engine can be easily integrated with other game engines.

The engine can be found on `https://github.com/hlavacs/ViennaPhysicsEngine`. All code relevant to this project can be found on the constraints branch, and it has also been integrated into the main branch.

1.2 Constraints

Constraints are necessary to create complex, realistic physics simulations, and are thus a major part of most physics engines. Generally speaking, a constraint is a condition that has to be satisfied by the current state of the simulation at all times. This condition usually involves one or two bodies (objects in the physics simulation) and restricts the involved bodies' relative movement.

> "Constraints are as fundamental to game physics as shaders are to graphics programming" [1]

As an example, one of the arguably most important constraints in any physics engine is the non-interpenetration constraint used for collision resolution. In this case, the condition that the system has to uphold would be that two colliding bodies will not penetrate into each other. Friction is usually implemented alongside this. Both have already been implemented in the VPE before this project was started.

Further important constraints are joints (like ball-and-socket joints or hinge joints) and motors, which are needed to simulate things like doors, swings and complex systems of objects like ragdoll physics.

When talking about constraints, a term that occurs commonly is "degree of freedom" or DOF. An unrestricted simulation entity initially has six degrees of freedom (along the three translation and rotation axes, respectively). Enforcing a constraint on a system of bodies removes degrees of freedom from that system, and hence influences how the bodies can move in respect to each other. For instance, a ball-and-socket joint removes the three translational DOF from the system, and only allows arbitrary relative rotation.

When only one body is involved in a constraint, the second body is usually treated as a static, unmovable object that is part of the world. This can also be achieved by setting the mass of the body that should remain static to be infinitely large. Using this, one can anchor the constraint system to a fixed point in space (otherwise it can move around freely as a whole since only the relative motion of the involved bodies is restricted).

The process of implementing constraints can be roughly split into two parts: Modelling the constraints (i.e., finding equations that represent the constrained geometry and figuring out to use this to ensure constraint satisfaction), and then solving a (potentially very large) system of these equations. The modelling part usually happens “offline” by doing the necessary math, whereas the second part happens in the actual code and is performed by a solver. A solver is an algorithm that must prevent constraint violation in some way. Multiple solvers with different strengths and weaknesses exist and will be described below.

The goal of this project was to extend the VPE so that it supports common joint constraints while not changing the already existing code too much. The constraints should be simple to add to the simulation in the user code and additional constraint types should be easy to add to the engine (by creating a new class, for example).

Initially, only the ball-and-socket and hinge joint types were supposed to be implemented. Adding new constraints only requires doing the required math behind them (see 3.7) and implementing it in a new class, so in the end, the following was implemented:

- A simple distance constraint
- Ball-and-socket joint
- Hinge joint with motor and angle limit
- Slider joint with motor and distance limit
- Fixed joint

2 Related work

The topic of real time physics simulation and constraints has been heavily researched for the past few decades. Many solutions for the problems one might encounter can be found, and additionally, a plethora of proprietary and open source 2D and 3D engines that fully implement a wide range of constraints already exist.

Consequently, the approach that was taken for this project was not to come up with new solutions, but instead to conduct a literature analysis, finding and comparing existing implementation methods, and then working this into the already existing framework of the VPE.

Garstenauer gives a great overview over the most important concepts and approaches in his master thesis [2]:

Following this description, constraints are formulated as a scalar function C that depends on the state of the involved objects. Constraints that specify that $C = 0$ are called equality constraints, and constraints that require $C \geq 0$ are called inequality constraints. Inequality constraints are useful for collision resolution (the penetration depth has to be non-negative) and things like angle limits for joints. Equality constraints on the other hand are mostly used for joints and motors.

When considering multiple equality constraints, the problem that needs to be solved is a system of linear equations. For inequality constraints (especially when handling contacts), the resulting problem is a Linear Complementarity Problem or LCP due to the extra conditions that need to be satisfied [3]. Mixing both results in a mixed LCP (MLCP) [4]. How these systems can be solved is a major aspect of constraint solvers.

Further, three methods for how constraints can be modelled and handled are described: one can either work with positions and orientations (which can be modified directly), with velocities (which can be changed by applying impulses) or with accelerations (which can be changed by applying forces). The corresponding constraint functions are referred to as position-, velocity- or acceleration constraints. Position constraints describe allowed positions, whereas velocity constraints and acceleration constraints describe allowed motion and allowed acting forces, respectively. Mixing these approaches is also possible.

An important approach that works on the position level is Position Based Dynamics [5], which is often used for soft bodies and cloth simulations. For rigid bodies, position-based approaches are mainly used for error correction [2].

The more common method is working with impulses or forces, but acceleration constraints struggle with handling collisions, so working on the velocity level is often preferred [6], [1]. Constraint forces can be calculated via Lagrange multipliers as described in [7] and [8], and can then be used to modify the velocities or accelerations of the involved objects so the constraint does not become violated. These new values are used to compute updated positions by numerical integration via some time stepping algorithm.

[2] further differentiates between sequential and simultaneous solvers. Sequential solvers only solve singular constraints at a time, while simultaneous methods try to find a solution that satisfies all constraints at once.

The advantage of the first approach is its simplicity – solving a simple constraint is comparably straightforward as it only involves a single equation when working on the velocity level, as shown later. However, applying correctional forces or impulses to bodies might invalidate constraints that had already been solved in previous steps. To alleviate this, all constraints are processed several times over multiple iterations [2]. For reaching convergence in a complex system involving many constraints, a possibly large number of iterations can be needed, which might have an impact on the engine's performance. Applications like video games often don't need perfect physical accuracy as long as the result is visually plausible, so the trade-off between speed and correctness is usually not an issue [9]. Errors that remain uncorrected in one time step can also be corrected over the next few time steps, assuming no new error is introduced by external forces.

The Gauss-Seidel method as discussed in [6] is a very popular iterative way of solving systems of linear equations. For solving an (M)LCP, a Projected Gauss-Seidel approach can be used. Given enough iterations, it should converge to an acceptable solution. As mentioned in [2], most sequential solver methods resemble the Gauss-Seidel method and are similar to solving the corresponding MLCP.

Simultaneous solvers, on the other hand, provide much more robust solutions, but involve solving a potentially huge MLCP to find the Lagrange multipliers and subsequently compute the constraint forces. A variety of different solvers exist for this purpose, for example using the Dantzig [3] and Lemke [10] algorithms, both of which are direct pivot based MLCP solvers. According to [11], the Lemke algorithm can deal with larger scale MLCPs than the Dantzig approach, but both are useful for smaller problems.

Iterative and simultaneous solvers can be combined into a block solver. A block solver tries to improve the robustness of sequential solvers by solving constraints in blocks instead of one at a time, and the blocks are then handled iteratively. If the blocks are kept small enough, the performance should not suffer too much. For example, the three translational constraints of a ball-and-socket joint can be solved as a system of 3 linear equations. The mentioned Dantzig or Lemke solvers can be used to solve smaller systems

as a part of an iterative solver [1][11].

A different type of simultaneous solver that does not work with Lagrange multipliers uses the Featherstone Articulated Body Algorithm as described in [12], with a great simplified description given in [13]. This method works with reduced coordinates (instead of taking away DOFs, it adds them to the system) and is suitable for chains or links of equality constraints like joints [11].

Of particular interest for this project were some papers and presentations by Erin Catto (see [9], [14], [1] and [15]), describing the Sequential Impulses solver. A Sequential Impulses solver is an intuitive and fast iterative solver that tries to reach the global solution by looping over and solving all constraints multiple times. In each iteration, impulses are applied to the bodies to ensure that the velocity constraint remains satisfied. This has already been used in the VPE for collision resolution, so it made sense to choose the same approach for other constraints as well.

What solver to use is a major design choice. Each solver has its own strengths and weaknesses, depending on the specific use case. Iterative solvers, for example, struggle with large mass rations and stiffness [9]. To improve robustness of the simulation, different solvers can be implemented and used depending on the situation. [11] gives a great description of how different solvers can be combined when starting with a basic SI solver.

Another issue to solve is constraint drift introduced by imprecise integration methods. The tried and tested (albeit not perfect) solution for this problem is Baumgarte stabilization as described in [16], which feeds the constraint error back into the velocity constraint in the form of a bias term.

A more expensive, but also more stable method of combating constraint drift is post-projection, for example by solving the position constraint after the positions have been integrated as described in [2].

An additional extremely helpful resource was a paper written by Daniel Chappuis for his own engine [17]. It shows the detailed derivation of the math necessary to formulate the constraint equations for the most important joints. This, alongside [15] and [9] was used as a main reference for the implementation.

Looking at existing, fleshed-out engines and their approaches was also very helpful. A common entry point for people wanting to learn more about real time physics simulations is box2d (and box2d-lite). Examples for open-source 3D engines would be the Open Dynamics Engine, Bullet, Project Chrono or ReactPhysics 3D. Bullet in particular implements many different solver types. Proprietary engines like Havok seem to be the most robust and are widely used by big game studios for AAA productions.

3 Algorithm

As mentioned above, the VPE already uses the Sequential Impulses solver for handling collision resolution and friction. This, coupled with the fact that this solver is easier to implement and understand compared to other solvers (which is also helpful given the educational context of the VPE), made the choice of solver easy.

3.1 Sequential Impulses

Sequential Impulses (in short SI) is a sequential solver that works on the velocity level by applying impulses to the constrained objects. It was described by Erin Catto in [9] and [15]. The concept is straightforward and does not necessitate solving big systems of linear equations or MLCPs, which makes it easy to implement as well. Conceptually, it is similar to using a (Projected) Gauss-Seidel method. To speed up convergence to the global solution, warm starting can be used to exploit the temporal coherence of physics simulations by using the impulse used in the previous time step as the initial solution [14], [15].

As a sequential solver, SI loops over all constraints and handles each one separately. Because the solution to one constraint can violate other constraints that have been solved previously, multiple iterations are needed for each time step. The concrete number of iterations can be hard-coded (in the VPE, this value can be changed via the debug GUI), or the solver can stop after errors and corrective impulses become small enough [14].

The solver loop consists out of three main steps:

- Integrate external forces like gravity, resulting in an intermediate velocity
- This velocity might violate the constraint(s), so apply corrective impulses for each constraint for some iteration number
- Use the resulting velocities to compute new positions and orientations

Using this, the basic algorithm looks as follows:

```
while simulation is running do
    integrate external forces to get v1;
    for some iteration number do
        for all constraint do
            solve velocity constraint;
            apply impulses to get v2;
        end
    end
    integrate positions and orientations using v2;
end
```

Algorithm 1: Basic Sequential Impulses Algorithm

The parts handling the external forces as well as the position update were already implemented in the VPE. Changing the order in which the constraints are solved can also have an impact [2]. For instance, engines like Bullet have the option to randomly sort the data structure containing the constraints.

SI can be turned into a block solver by diving constraints into blocks which can then be solved simultaneously [1]. In my implementation, the largest block size is 3, as a 3x3 matrix can be easily inverted using the glm matrix library.

3.2 Formulating constraint equations

Before we can solve constraints using SI, we first need to actually find the constraint functions that the solver needs. Following the approach of [2] and [9], pairwise constraints can be formulated on the position level as a scalar function of the state of the two involved objects. The state of an object at a specific time step is given by a position x_i in world coordinates (this is also the center of mass) and an orientation defined by a quaternion q_i, both of which depend on the current simulation time slot. The state for both objects is collected in the vector s:

$$s(t) = (x_1(t), q_1(t), x_2(t), q_2(t))$$

The scalar position constraint function C for an equality constraint is then given by:

$$C(s) = 0$$

and for inequality constraints:

$$C(s) \geq 0$$

Note that for this implementation, inequality constraints are solved as an equality constraint if they are violated, and otherwise ignored. For example, if $C(s) \leq 0$, then the solver tries to move things around so that C becomes 0. As long as $C(s) \geq 0$, nothing is

done.

If C is equal or very close to 0, the constraint is satisfied, and the evaluation of C at a specific time t is the constraint error. This value is mainly relevant for error correction and will be discussed later.

SI works on the velocity level and thus requires solving the velocity constraint $\dot{C}$. The velocity constraint can be found by taking the first time derivative of the position constraint (the second time derivative would be the acceleration constraint, which is not needed for this solver). If $\dot{C} = 0$ in addition to $C = 0$, the bodies do not move in a way that invalidates the constraint, so it will also be satisfied in the next time step. Ensuring that the value of the velocity constraint remains 0 should prevent the occurrence of constraint violations [2].

Since C is a function of the state s and s is a function of t, the derivative can be computed using the multi-variable chain rule:

$$\dot{C} = \sum_{s_i \in s} \frac{dC}{ds_i} \frac{\partial s_i}{\partial t} = 0$$

This results in the following form:

$$\dot{C} = J_c V_c = 0$$

where J_c is the constraint's Jacobian matrix and V_c is a 12x1 column vector containing the linear and angular velocities v_i and ω_i of the two constrained bodies:

$$V_c = (v_1, \omega_1, v_2, \omega_2)^T$$

The 1x12 Jacobian matrix can be obtained by inspecting the coefficients of the velocity terms in the resulting expression of the time derivate (examples are shown in 3.7).

For a collection of constraints, all constraint functions can be combined into a single sx1 column vector, where s is the number of constraints. Given a system of n bodies, this consequently results in a sx6n system Jacobian J, where each single constraint contributes one row. The velocity vector V then contains the velocities of all bodies and is a 6nx1 column vector [2], [9].

An important distinction to make here is that a constraint in this context always removes one degree of freedom from the system, and in this case $J_c V_c$ is a scalar. While commonly referred to as a constraint as well, most joints remove multiple degrees of freedom and are therefore already a system of constraints. In this case, s is also the number of DOFs removed from the system [17]. For example, a hinge joint removes 5 degrees of freedom (as only one rotation axis is allowed), so it is made up out of 5 constraints (involving two bodies) and will have a 5x12 Jacobian matrix.

In a system with many bodies and constraints where constraints are only defined pairwise (that is, one constraint restricts at most two bodies), the system Jacobian will be a sparse matrix as only the entries with indices that correspond to the two involved bodies will be nonzero. This property can be exploited to construct more efficient solving algorithms [7].

3.3 Constraint forces

Each constraint has its own reaction force f_{c_i} and torque τ_{c_i} . Assuming a system of n bodies again, these can be gathered in a $6n$x1 colum vector F_c:

$$F_c = \begin{pmatrix} f_{c_1} \\ \tau_{c_1} \\ \vdots \\ f_{c_n} \\ \tau_{c_n} \end{pmatrix}$$

From $\dot{C} = J_c V_c = 0$ we can see that the velocity vector that is allowed by the constraint is orthogonal to the rows of the Jacobian [9]. Additionally, an important aspect of constraint forces is that they do no work (they neither add energy to nor remove energy from the system – this is also called the principle of virtual work). If $F_c^T V = 0$, the power of the system is 0 and thus no work is being done. Both conditions can only be satisfied if the constraint forces are of the form:

$$F_c = J^T \lambda$$

where λ is a vector of s unknown Lagrange multipliers that specify the magnitude of the constraint forces. These are the values that the solver needs to find [8].

3.4 Error correction and Baumgarte stabilization

Before looking at how we can compute λ, another problem must be considered. When working with velocity constraints, there is no information about the actual position and orientation of the bodies. Applying the constraint impulses only ensures that velocities that would lead to a violation are cancelled out, but nothing is done to actually fix already existing errors. There is an implicit assumption that the position constrain is met, i.e., $C = 0$ [2].

The VPE, like most engines, uses a time stepping algorithm, meaning the state of the system is not simulated continuously, but rather only at discrete points in time. Because of this, the assumption $C = 0$ is often wrong, as the time at which the constraint becomes violated might fall between two subsequent time steps. In this case, the constraints try to keep the violation from growing, but over time, position errors caused by the mentioned time stepping issue in addition to errors caused by numerical integration imprecision will accumulate and lead to constraint drifting (for example, joints will separate over time).

The following graphs show how the magnitude of the constraint error for the translation constrains of a hinge joint grows over time when no error correction is used:

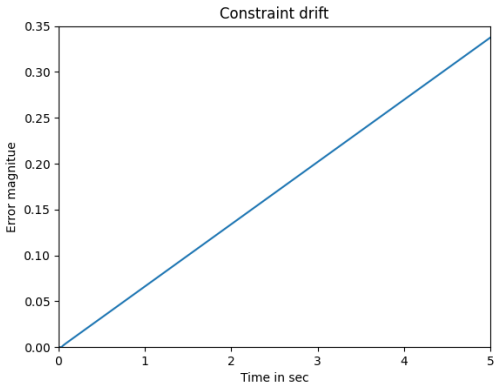


Figure 3.1: Constraint drift with only gravity

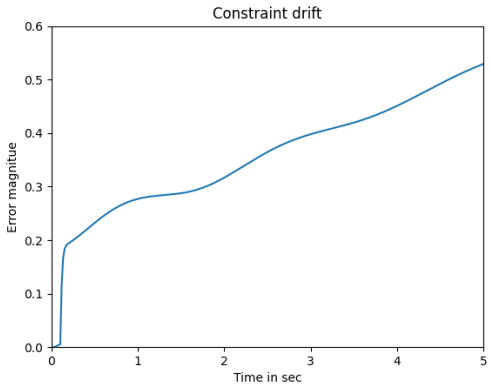


Figure 3.2: Constraint drift with collision

One straightforward and efficient solution for this problem is Baumgarte stabilization [16]. When working on the velocity level, this works by feeding the position error back into the velocity constraint by introducing a bias term b that depends on the error. The velocity constraint is then:

$$\dot{C} = J_c V_c + b = 0$$

The position error at the current time t is given by the evaluation of the position constraint $C(s)$. If we want to fix this error within the next timestep Δt, then b becomes

$$b = \frac{C(s)}{\Delta t}$$

Fine-tuning the time taken until the error is fixed can be done by introducing another factor $\beta \in [0, 1]$:

$$b = \frac{\beta \cdot C(s)}{\Delta t}$$

Treating beta as a percentage value, this means that each time step (so in each frame), the corresponding percentage of the current error will be corrected [17]. Given an initial error C_0 and assuming that no further errors are introduced during this time, the position error decays exponentially as given by

$$Error(t) = C_0 \cdot (1 - \beta)^t$$

The following graphic shows the error decay for $\beta = 0.15$. Note that it takes around 30 time steps for the error to become neglectable small, which translates into roughly half a second when running the simulation at 60 frame per second.

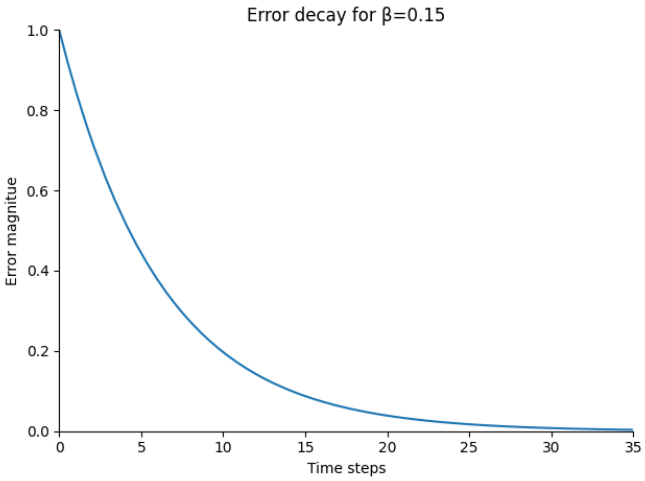


Figure 3.3: Error decay for $\beta = 0.15$

β can be used to adjust the stiffness or softness of joints. If no softness is desired, it might sound plausible to fix errors as fast as possible by choosing a value close to 1. Unfortunately, choosing large values for β is often impractical as the addition of the bias term leads to the constraint forces doing work and consequently introduce energy into the system (the velocity that is used to correct the error remains even after the error is gone), which is a disadvantage of this method [18]. This might be fine for small systems like a single joint, but a more complex one can become very instable (I noticed this with fixed chains of ball-and-socket joints, for example).

Consequently, the value of beta needs to be selected carefully. As there seems to be no straightforward and reliable method to do so, the necessary tweaking is another disadvantage of Baumgarte stabilization [2]. For the VPE, I found that values from 0.1 to 0.2 work well enough, and the joints were implemented in way that allows the user to change β dynamically. This way, one can fine-tune β for each closed system of constraints in the simulation if necessary.

A similar approach to Baumgarte stabilization is also very helpful when trying to implement joint motors. In this case, the motor's velocity constraint should not be equal to 0, but rather equal to some motor speed, which translates to setting the bias term to the wanted speed value.

3.5 Computing Lagrange multipliers

Each body in our simulation has a scalar mass m and an inertia tensor I defined by a 3x3 matrix. According to [9], assuming a force F and a torque τ that act on the body,

the Newton-Euler equations of motion are:

$$m\dot{v} = F$$

$$I\dot{\omega} = \tau$$

When considering constraints, F and τ can be split into external parts (like gravity) and the internal force applied by the constraints:

$$F = F_c + F_{ext}$$

$$\tau = \tau_c + \tau_{ext}$$

For body i, the mass m_i and inertia tensor I_i can be combined into the mass matrix M_i:

$$M_i = \begin{pmatrix} m_i \cdot E_{3\times 3} & 0 \\ 0 & I_i \end{pmatrix}$$

Considering a system of n bodies, we can again combine all these matrices M_i into a single mass matrix M:

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

Just as for the constraint forces F_c, the external forces for the entire system can be combined into one column vector F_{ext}:

$$F_{ext} = \begin{pmatrix} f_{ext_1} \\ \tau_{ext_1} \\ \vdots \\ f_{ext_n} \\ \tau_{ext_n} \end{pmatrix}$$

Using $F_c = J^T\lambda$, for the entire system, the equation becomes:

$$M\dot{V} = F_c + F_{ext} = J^T\lambda + F_{ext}$$

In the SI solver, the external forces are handled and integrated into the velocity separately, before the constraints are solved, and thus removed from the formula. This step results in the initial velocity vector V_1, which usually violates the velocity constraints. We want to find a new velocity vector V_2 that satisfies the constraint (so $JV_2 + b = 0$). The change in velocity needed to achieve this is V_2 – V_1, and we achieve this by applying a constraint impulse P_c [14]:

$$P_c = M(V_2\text{-}V_1) = J^T\lambda \Rightarrow V_2 = V_1 + M^{-1}J^T\lambda$$

By plugging this expression for V_2 into the velocity constraint, we get

$$J \cdot (V_1 + M^{-1}J^T\lambda) + b = 0 \Rightarrow JM^{-1}J^T\lambda = -(JV_1 + b)$$

This is a system of linear equations of the form $A\lambda = c$ where $A = JM^{-1}J^T$ and $c = -(JV_1 + b)$. A is also called the effective mass matrix of the constraint. Solving this system is the main job of simultaneous solvers.

However, when using SI, we consider a single constraint at a time, so we only need to deal with a single linear equation as the effective mass matrix A will be a scalar ($JM^{-1}J^T \to 1 \times 12 \cdot 12 \times 12 \cdot 12 \times 1$). To improve robustness, we group logically related constraints (i.e., constraints that will highly influence each other when solved) into blocks with a maximum size of 3. This requires solving a system of 2 or 3 linear equations, which can be done by inversion and left side multiplication of the effective mass matrix A, which gives us a final equation for λ:

$$\lambda = -A^{-1}(JV_1 + b)$$

Once we have λ, we can compute the constraint impulse and compute V_2, which will now satisfy the constraint. This can then be used to directly modify the bodies' velocities and concludes the solver iteration.

3.6 Summary

Summarized, the following steps are necessary to create a new constraint:

- Create position constraint function
- Take time derivate to get velocity constraint
- Find Jacobian by inspection
- Calculate effective mass matrix $JM^{-1}J^T$

These steps need to be done for each constraint type or joint and are best done offline, as the engine only needs to compute the values of the Jacobian and effective mass matrix using the found expressions [17], [9].

During the simulation, the solver needs to (assuming external forces have already been integrated):

- Compute the entries of the Jacobian
- Compute the entries of the effective mass matrix
- Compute the constraint error and bias term
- Compute the Lagrange multipliers λ
- Use λ to compute and apply impulses

Sequential Impulses loops over all constraints multiple times, but the only thing that changes during these iterations is the velocity of the bodies. Therefore, for better performance, variables that don't depend on the velocity vector can be pre-computed

once per time step before the constraint loop begins. From the list above, this applies to the first 3 steps (the Jacobian, effective mass matrix and bias term). Only steps 4 and 5 need to be done multiple times per frame.

3.7 Constraint derivation

This chapter gives an overview over the implemented joint types and the specific math behind them. For each constraint, we need to find the Jacobian and effective mass matrix. The following content closely follows the math shown in [17] and [2] and is not my own work. I decided to include it since this is a very important part of the concrete constraint implementation and as a personal reference. Some steps are left out for brevity, please see the referenced material for detailed steps.

The derivations assume constraints that involve two bodies. The bodies have scalar masses m_1 and m_2 and 3x3 inertia tensors I_1 and I_2. The world positions of their centres of mass are x_1 and x_2, and their orientations in world space are specified by the unit quaternions q_1 and q_2. Their linear and angular velocities are given by v_i and ω_i, respectively.

Note that for most constraint equations C, the Baumgarte bias term b is given by $b = \frac{\beta \cdot C(s)}{\Delta t}$, where $C(s)$ is the current error of the position constraint and β is the tweaking constant mentioned in 3.4.

3.7.1 Distance constraint

A distance constraint has to ensure that the distance between two bodies is always some scalar d. It removes one translational degree of freedom. Usually, position constraints allow one to fix the distance between arbitrary points on both bodies. This constraint was mainly meant for testing and getting familiar with the subject, so only the distance between the object's centers of mass is considered, which makes the math a lot easier. The position constraint looks as follows:

$$C(s) = \frac{1}{2}\left(|x_2 - x_1|^2 - d^2\right) = 0$$

This equation says that the distance between x_1 and x_2 has to be equal to d. The addition of squares and the factor $\frac{1}{2}$ is only for simplification of the time derivative and does not change the meaning of the constraint [18]. These can be left out when computing the position error for Baumgarte stabilization. Taking the time derivative, the velocity constraint is then:

$$\dot{C}(s) = (x_1 - x_2) \cdot v_1 + (x_2 - x_1) \cdot v_2 = 0$$

Because x_1 and x_2 are the centres of mass, no angular term exists. By inspecting the coefficients of v_1 and v_2, we can find the 1x12 Jacobian matrix:

$$J = \begin{pmatrix} x_1 - x_2 & 0 & x_2 - x_2 & 0 \end{pmatrix}$$

The effective mass matrix A is a scalar:

$$A = JM^{-1}J^T = m_1 + m_2$$

3.7.2 Ball-and-socket joint

A ball-and-socket joint disables relative translation of the constrained bodies while allowing arbitrary rotation, removes three degrees of freedom and is made up out of 3 separate constraints (one for each translation axis). The joint is defined by an anchor point p in world space. When the joint is created, we transform the anchor point to the local space of each body. During the simulation, we move the local anchor points back to world space in each frame again, giving us a world anchor point p_i for body i.

Again, assuming the world space coordinates of the centres of mass are x_1 and x_2, the position constraint equations look like this:

$$C(s) = x_2 + r_2 - x_1 - r_1$$

where r_1 and r_2 are vectors from the bodies' centres to the anchor point in world space (so $p_i = x_i + r_i$). All used variables are 3x1 vectors, so we have 3 total equations which say that the world space anchor point for both bodies must be the same. The velocity constraint is:

$$\dot{C}(s) = v_2 + \omega_2 \times r_2 - v_1 - \omega_1 \times r_1 = 0$$

Using skew symmetric matrices to replace the cross products with matrix multiplications, we get the following 3x12 Jacobian:

$$J = \begin{pmatrix} -E_3 & skew(r_1) & E_3 & -skew(r_2) \end{pmatrix}$$

where $skew(r_i)$ is the 3x3 skew matrix constructed from vector r_i.

The 3x3 effective mass matrix A is given by:

$$A = JM^{-1}J^T = \frac{1}{m_1}E_3 + skew(r_1) \cdot I_1^{-1} \cdot skew(r_1)^T + \frac{1}{m_2}E_3 + skew(r_2) \cdot I_2^{-1} \cdot skew(r_2)^T$$

3.7.3 Hinge joint

A hinge joint removes all relative translation, like a ball-and-socket joint, in addition to only leaving one rotational degree of freedom. It needs an anchor point p and a hinge axis a, both in world space. It only allows rotation around this hinge axis.

The translation constraints are the same as for the ball-and-socket joint, so we only need to figure out the rotation constraints. We again move the (unit) hinge axis to local space for both bodies when the joint is created. During the simulation, we move the local axes back to world space to get a_1 and a_2. In addition, we need to construct two

unit vectors b and c that are orthogonal to a_2. Using this, we can formulate the rotation constraint functions:

$$C_{rot}(s) = \begin{pmatrix} a_1 \cdot b \\ a_1 \cdot c \end{pmatrix}$$

The constraints enforce that a_1 is also orthogonal to b and c, which limits the allowed rotation to the hinge axis. Taking the time derivative, we can find the velocity constraint functions:

$$\dot{C}(s) = \begin{pmatrix} (b \times a_1)(\omega_2 - \omega_1) \\ (c \times a_1)(\omega_2 - \omega_1) \end{pmatrix}$$

The 2x12 Jacobian is then:

$$J_{rot} = \begin{pmatrix} 0 & -(b \times a_1) & 0 & (b \times a_1) \\ 0 & -(c \times a_1) & 0 & (c \times a_1) \end{pmatrix}$$

The 2x2 effective mass matrix A is:

$$A_{rot} = J_{rot} M^{-1} J_{rot}^T =$$

$$\begin{pmatrix} (b \times a_1)^T I_1^{-1} (b \times a_1) + (b \times a_1)^T I_2^{-1} (b \times a_1) & (b \times a_1)^T I_1^{-1} (c \times a_1) + (b \times a_1)^T I_2^{-1} (c \times a_1) \\ (c \times a_1)^T I_1^{-1} (b \times a_1) + (c \times a_1)^T I_2^{-1} (b \times a_1) & (c \times a_1)^T I_1^{-1} (c \times a_1) + (c \times a_1)^T I_2^{-1} (c \times a_1) \end{pmatrix}$$

Angle limits

For the hinge joint, two additional constraints can be added that limit how large the rotation range around the hinge axis can be by enforcing a minimum and maximum relative rotation angle. The minimum and maximum angle $\theta_{min} \in [-2\pi; 0]$ and $\theta_{max} \in [0; 2\pi]$ limit rotation in the negative or positive direction, respectively.

To implement this constraint, the current relative rotation angle between the bodies along the hinge axis needs to be computed so we can compare it with the limits. The initial orientation between the bodies when the joint is created corresponds to an angle of 0. We save this orientation in a quaternion q_{init}. $q_{init} q_1 = q_2$, so

$$q_{init} = q_2 q_1^{-1}$$

Similarly, during the simulation, we need to find the current orientation q_{curr} in each frame:

$$q_{curr} = q_2 q_1^{-1}$$

Using this, we can find a quaternion representing the difference between the current and initial orientation:

$$q_{diff} = q_{curr} q_{init}^{-1}$$

The quaternion q_{diff} encodes the current angle θ_{curr}, which we need to extract as described in [17]. Using this angle, we can now formulate the two position constraint functions.

Note that these are now inequality constraints, and we can ignore them if the current angle is between the limits:

$$C_{min}(s) = \theta_{curr} - \theta_{min} \geq 0$$
$$C_{max}(s) = \theta_{max} - \theta_{curr} \geq 0$$

By taking the time derivative, we can find the velocity constraints:

$$\dot{C}_{min}(s) = (\omega_2 - \omega_1) \cdot a$$
$$\dot{C}_{max}(s) = -(\omega_2 - \omega_1) \cdot a$$

With this, we can find the two 1x12 Jacobian matrices:

$$J_{min} = \begin{pmatrix} 0 & -a^T & 0 & a^T \end{pmatrix}$$
$$J_{max} = \begin{pmatrix} 0 & a^T & 0 & -a^T \end{pmatrix}$$

The effective mass scalar matrices are the same for both cases:

$$A_{min} = A_{max} = a^T I_1^{-1} a + a^T I_2^{-1} a$$

Motor

If we want to simulate things like wheels or automatically opening doors, another feature that can be added to hinge joints is a motor. Using the motor, we can specify a specific angular velocity ω_{motor} along the hinge axis that we want to be maintained between the bodies. In addition, to control how fast the motor speed ramps up, we can set a maximum force F_{max} that can be applied each time step.

The motor constraint involves velocities, and is therefore easy to formulate directly on the velocity level:

$$\dot{C}_{motor}(s) = a \cdot (\omega_2 - \omega_1) + \omega_{motor} = 0$$

Note that we simply use the motor speed for the bias term usually used for Baumgarte stabilization. The constraint makes sure that the relative angular velocity projected onto the hinge axis is equal to w_{motor}. From this, we can find the 1x12 Jacobian matrix:

$$J_{motor} = \begin{pmatrix} 0 & -a^T & 0 & a^T \end{pmatrix}$$

Because $J_{motor} = J_{min}$, we find that the 1x1 effective mass matrix A_{motor} is also equal to A_{min}:

$$A_{motor} = A_{min} = a^T I_1^{-1} a + a^T I_2^{-1} a$$

To make sure that the applied force does not exceed the maximum value, we need to clamp the magnitude of the applied impulse (given by λ):

$$-\|F_{max}\| \leq \lambda \leq \|F_{max}\|$$

By replacing a in the Jacobians of the limit and motor constraints with either a_1 or a_2, we can allow the hinge axis to tilt, which can be important when the joint is not fixed to a static body.

3.7.4 Slider joint

A slider joint, just like a hinge joint, removes 5 degrees of freedom from the system. The difference between the two is that the slider joint removes all relative rotation while keeping one rotational axis (the slider axis) accessible, so it consists out of 3 rotational and 2 translational constraints.

For the translation constraints, just as for the previous joints, given an anchor point p in world space, we convert this to local space for both bodies and move these points back into world space to get p_1 and p_2 during the simulation. Additionally, we also convert the slider axis a into local space for one of the bodies, which is also converted back to world space in each frame, giving us a_w. By using this axis instead of the original slider axis, we can allow the whole system to tilt. We compute two vectors n_1 and n_2 that are orthogonal to a_w to formulate the position constraint for the translations:

$$C_{trans}(s) = \begin{pmatrix} (x_2 + r_2 - x_1 - r_1) \cdot n_1 \\ (x_2 + r_2 - x_1 - r_1) \cdot n_2 \end{pmatrix} = 0$$

r_1 and r_2 are the same vectors used for the ball-and-socket joint. The constraints say that the relative translation projected onto n_1 and n_2 (which are orthogonal to the slider axis) has to be 0, so only relative translation along the slider axis is allowed.

The rotation constraint can be formulated using the angles θ_{ix}, θ_{iy} and θ_{iz}, which describe the rotation of body i around the principal axes. By requiring that the angles need to be the same for all axes, we can make sure that the relative rotation is 0. The constraint functions look like this:

$$C_{rot} = \begin{pmatrix} \theta_{2x} - \theta_{1x} \\ \theta_{2y} - \theta_{1y} \\ \theta_{2z} - \theta_{1z} \end{pmatrix} = 0$$

As usual, we get the velocity constraints by taking the time derivatives:

$$\dot{C}_{trans} = \begin{pmatrix} n_1 \cdot v_2 + \omega_2 \cdot (r_2 \times n_1) - n_1 \cdot v_1 - \omega_1 \cdot ((r_1 + u) \times n_1) \\ n_2 \cdot v_2 + \omega_2 \cdot (r_2 \times n_2) - n_2 \cdot v_1 - \omega_1 \cdot ((r_1 + u) \times n_2) \end{pmatrix} = 0$$

$$\dot{C}_{rot} = \omega_2 - \omega_1 = 0$$

This gives us the 2x12 and 3x12 Jacobian matrices:

$$J_{trans} = \begin{pmatrix} -n_1^T & -((r_1 + u) \times n_1)^T & n_1^T & (r_2 \times n_1)^T \\ -n_2^T & -((r_1 + u) \times n_2)^T & n_2^T & (r_2 \times n_2)^T \end{pmatrix}$$

$$J_{rot} = \begin{pmatrix} 0 & -E_3 & 0 & E_3 \end{pmatrix}$$

Using:

$$a = \left(\frac{1}{m_1} + \frac{1}{m_2} \right) + ((r_1 + u) \times n_1)^T I_1^{-1} ((r_1 + u) \times n_1) + (r_2 \times n_1)^T I_2^{-1} (r_2 \times n_1)$$

$$b = ((r_1 + u) \times n_1)^T I_1^{-1} ((r_1 + u) \times n_2) + (r_2 \times n_1)^T I_2^{-1} (r_2 \times n_2)$$
$$c = ((r_1 + u) \times n_2)^T I_1^{-1} ((r_1 + u) \times n_1) + (r_2 \times n_2)^T I_2^{-1} (r_2 \times n_1)$$
$$d = \left(\frac{1}{m_1} + \frac{1}{m_2}\right) + ((r_1 + u) \times n_2)^T I_1^{-1} ((r_1 + u) \times n_2) + (r_2 \times n_2)^T I_2^{-1} (r_2 \times n_2)$$

we get the 2x2 and 3x3 effective mass matrices:

$$A_{trans} = J_{trans} M^{-1} J_{trans}^T = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$$

$$A_{rot} = J_{rot} M^{-1} J_{rot}^T = I_1^{-1} + I_2^{-1}$$

Limits

Similar to the hinge joint, we can add limits to the slider joint. For this, we can set a minimum (negative) and maximum (positive) relative translation distance along the slider axis, d_{min} and d_{max}.

The relative distance between the bodies is given by the vector u between their respective world anchor points p_1 and p_2. When the joint is created, this vector and thus the initial distance is 0. Projecting this onto the slider axis gives us the current distance d_{curr}:

$$d_{curr} = u \cdot a = (p_2 - p_1) \cdot a$$

This allows us to formulate the limit position constraints:

$$C_{min}(s) = d_{curr} - d_{min} \geq 0$$

$$C_{max}(s) = d_{max} - d_{curr} \geq 0$$

By taking the time derivative, we can find the velocity constraints:

$$\dot{C}_{min}(s) = a \cdot v_2 + \omega_2 \cdot (r_2 \times a) - a \cdot v_1 - \omega_1 \cdot ((r_1 + u) \times a)$$

$$\dot{C}_{max}(s) = -a \cdot v_2 - \omega_2 \cdot (r_2 \times a) + a \cdot v_1 + \omega_1 \cdot ((r_1 + u) \times a)$$

With this, we can again find the two 1x12 Jacobian matrices:

$$J_{min} = \begin{pmatrix} -a^T & -((r_1 + u) \times a)^T & a^T & (r_2 \times a)^T \end{pmatrix}$$

$$J_{max} = \begin{pmatrix} a^T & ((r_1 + u) \times a)^T & -a^T & -(r_2 \times a)^T \end{pmatrix}$$

And we have the same 1x1 effective mass matrix for both cases:

$$A_{min} = A_{max} = \left(\frac{1}{m_1} + \frac{1}{m_2}\right) + ((r_1 + u) \times a)^T I_1^{-1} ((r_1 + u) \times a) + (r_2 \times a)^T I_2^{-1} (r_2 \times a)$$

Motor

The slider joint can also support relative motor forces, which can be used to build pistons, for example. As opposed to the hinge motor, we now specify the linear motor speed v_{motor}, and we limit the motor force with F_{max} again. Following the approach of the hinge motor, we formulate the constraint directly on the velocity level by projecting the relative velocity difference onto the slider axis and introducing the motor speed via the bias term:

$$\dot{C}_{motor}(s) = a \cdot (v_2 - v_1) + v_{motor} = 0$$

From this we can extract the 1x12 Jacobian matrix:

$$J_{motor} = \begin{pmatrix} a^T & 0 & -a^T & 0 \end{pmatrix}$$

The effective mass is a scalar:

$$A_{motor} = \frac{1}{m_1} + \frac{1}{m_2}$$

Just as with the hinge joint, we need to clamp the magnitude of the applied impulse (given by λ):

$$-\|F_{max}\| \leq \lambda \leq \|F_{max}\|$$

3.7.5 Fixed joint

Finally, the fixed joint removes all degrees of freedom and thus all relative motion from the system. The constrained bodies essentially behave as one unit (however, from a design standpoint, it is usually better to combine objects during the modelling process instead of using a fixed joint to glue them together, unless this needs to happen dynamically [19]).

The joint is defined by an anchor point, although the position of the anchor should not really influence how the joint works. In the implementation, it looked like a joint that has one static body loses a lot of stiffness when the anchor point is too far away from the non-static body, however.

A fixed joint is a combination of a ball-and-socket joint and the rotational aspect of a slider joint, so we already have everything we need: For the translation constraints, we can reuse the work done for the ball-and-socket joint, and for the rotation constraints, we can use the rotation part from the slider joint.

4 Implementation

The implementation closely follows the algorithm described above and was very straightforward. The VPE was an already fully functional Visual Studio project, so no work had to be done on the build setup and development environment. The entire project can be found on GitHub, and should be usable out of the box. To see some constraints in action, demos can be spawned into the world by using the constraint GUI window.

An abstract base class `Constraint` was created as a part of the VPE which holds pointers to the two bodies and specifies two abstract methods `setUp()` and `solveVelocity()`. `setUp()` has to be called once per frame before the constraints are solved. Its job is to pre-compute the Jacobian, effective mass matrix and bias term, which are saved in class instance variables. As the name suggests, `solveVelocity()` has to solve the velocity constraint by finding the Lagrange multipliers and applying the constraint impulses. The call for this method has been included in the VPE's collision resolution loop, so it uses the same number of loop iterations. Interleaving the solving of collisions and constraints produces better results. When doing them separately, after one another, collisions with constrained bodies have visible penetration depth as the constraints push the objects back into each other after collisions have already been resolved.

The `Constraint` class also offers the method `containsBody()` which can be used to check whether a body is part of this constraint or not. This is mainly used to erase a constraint when one of the two bodies is removed from the simulation. Additionally, there are two extra fields that can be used to specify whether each body should be affected by the angular constraint impulses. The fields are set to false (meaning the body should not be affected) if the respective mass is very large. The reason for this is that the inverted inertia tensors for objects with infinite mass still contain very small values. If these values are used to compute the impulses, at 1800 times per second for 30 constraint iterations at 60 frames per second, this will add up over time and the infinitely heavy objects will rotate.

A class inheriting from `Constraint` has been created for each constraint type. Each class has instance variables for saving pre-computed values. Constraints can be added to the simulation by creating a shared pointer to the wanted constraint class and passing this pointer to `addConstraint()`, which simply adds it into a vector holding all constraints.

Hinge and fixed joints both use the same translation constraints as the ball-and-socket joint. Instead of duplicating the code for this, these two classes hold a pointer to a ball-and-socket joint that takes care of the translational aspects.

Concerning the grouping of constraints into blocks, this has been done for:

- The 3 translation constraints for the ball-and-socket joint
- The 2 rotational constraints for the hinge joint
- The 3 rotational constraints for the slider and fixed joint
- The 2 translation constraints for the slider joint

The following code example shows how `setUp()` and `solveVelocity()` are implemented for the distance constraint. Note how the code is a pretty much direct implementation of the algorithm and constraint derivations described so far.

```
void setUp(real dt) override {
    // Compute distance between the objects' center, their distance and the difference to the constraint distance
    m_rel_pos = m_body1->m_positionW - m_body2->m_positionW;
    m_body_distance = glm::length(m_rel_pos);
    m_offset = m_distance - m_body_distance;

    // Compute parts of the jacobian matrix; in this case the relative positions
    m_j1 = glm::normalize(m_rel_pos);
    m_j2 = -m_j1;

    // Compute total constraint mass and invert it if possible
    real total_mass = m_body1->m_mass_inv + m_body2->m_mass_inv;
    m_inv_constraint_mass = total_mass < Constraint::epsilon ? 0.0_real : 1.0_real / total_mass;

    // Compute bias for Baumgarte stabilization
    m_bias = (m_bias_factor * m_offset) / dt;
};

void solveVelocity() override {
    if (abs(m_offset) > Constraint::epsilon) {
        // Compute dot product of Jacobian and velocity vector; keep in mind that j2 = -j1, so the original expression can be simplified
        real jv = glm::dot(m_body1->m_linear_velocityW - m_body2->m_linear_velocityW, m_j1);
        real lambda = m_inv_constraint_mass * -(jv - m_bias);

        glmvec3 impulse1 = m_j1 * lambda * m_body1->m_mass_inv;
        glmvec3 impulse2 = m_j2 * lambda * m_body2->m_mass_inv;

        m_body1->m_linear_velocityW += impulse1;
        m_body2->m_linear_velocityW += impulse2;
    }
}
```

Listing 4.1: Distance Constraint

5 Evaluation

The evaluation of physics simulations is crucial in ensuring a stable and correct system. A very common stability test is box stacking, but this was already implemented in the VPE as a part of the collision resolution. Another possible evaluation method would be a comparison to other existing solutions. I used the test demos of ReactPhysics 3D for this, as they are easy to build and that engine also uses Sequential Impulses. Some projects like Project Chrono even go as far as conducting real laboratory experiments that can be compared to the simulation results [20]. For this project, I mainly relied on visual evaluation, i.e. creating joints with different parameters, letting bodies collide with them, and seeing how the simulation behaves. For single joints, this works reasonably well because errors are usually immediately visible (the joint separates, for example). Especially the effectiveness of Baumgarte stabilization can be observed quite well. Complex constraint systems like joint chains are more interesting. Here, individual joint errors are not as easy to spot and might not even matter that much as long as the total system is stable enough, which is the main concern. For this, I created a few structures composed out of multiple joints and tested how they responded to external influences. Some of these are shown in 5.3.

5.1 Effects of Baumgarte stabilization

The effect of Baumgarte stabilization in single joints can be visualized quite well by measuring the absolute constraint error in each time step. An initial error is introduced while the simulation is paused (debug mode) by editing the positon/orientation via the debug GUI. The joints consisted out of one static body with infinite mass and one movable body, so the constraint can only influence the latter. Given the time step $\Delta t = \frac{1}{60}$, the curves behave more or less as one might expect given the nature of the exponential decay.

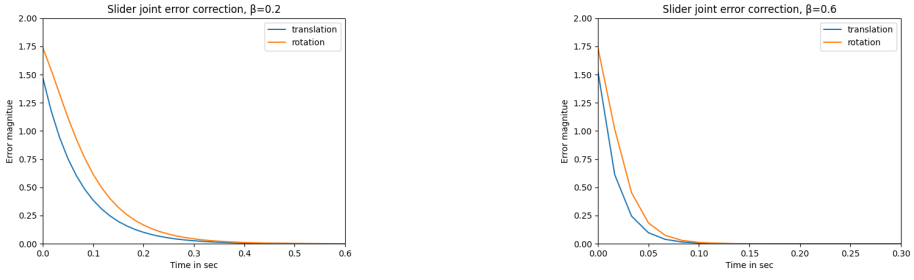


Figure 5.1: Baumgarte stabilization for a slider joint

Here is a similar graph for a hinge joint:

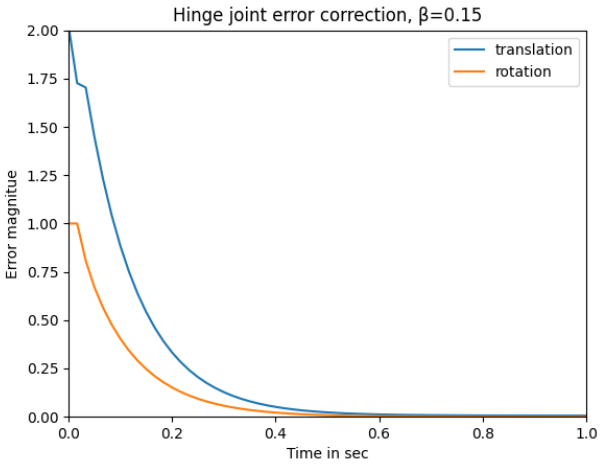


Figure 5.2: Hinge joint error correction

The interesting observation here is that, even though the objects were initally motionless, after the errors were corrected, the non-static body moved with an angular velocity of $\left(8.78009e^{-7}, 1.8155, 5.08804e^{-8}\right)$, which is hardly ignorable. This highlights one of the major disadvantages of Baumgarte stabilization, namely the introduction of energy into the system.

Another test that shows issues with Baumgarte stabilization is a simple ball-and-socket joint chain where both ends are static. The joint anchor point is always the center of the previous link. Even with just 4 elements (2 of which are movable), shooting a few cubes into the chain and especially towards the second element, the system often explodes even with smaller values for β. While the joints don't drastically separate, the second cube starts moving very fast and clearly deviates from the allowed position. Iterative solvers like SI generally struggle with huge mass ratios like in this case, but a quick test-implementation of projection error correction solved this issue, so the reason is likely Baumgarte stabilization.

5.2 Performance

Another important factor for real time engines is performance. For iterative solvers, the crucial thing to look at are the operations that are part of the inner solver loop (which corresponds to `solveVelocity()` for this implementation), as these are executed multiple times per frame. Taking into account that the biggest block size we use is 3, the only steps necessary here, as described in 3.6, consist out of small matrix and vector multiplications and additions:

- Computing JV_1. This is in the worst case the product of a 3x12 matrix and a 12x1 vector which can be split into smaller sub-matrix multiplications.
- Computing $\lambda = -A^{-1}(JV_1 + b)$. Since A^{-1} and b are computed in the setup step, the worst case here is a 3x3 matrix multiplied with a 3x1 vector
- Updating the velocities with $V_2 = V_1 + M^{-1}J^T\lambda$. Again, this can be split in way that results a 3x3 matrix and 3x1 vector multiplication at worst.

Considering this, the performance impact should be moderate if not too many loops are used for the solver. For performance testing, I disabled all collision detection as this is not part of what we want to test here. Frames per second were measured with an external tool. The engine starts off in paused/debug mode. The project was run in full screen on a 4K monitor. For the test, I added hinge joint (consisting out of 5 constraints grouped into a block of 3 and a block of 2) chains with a length of 100 cubes to the scene. After the chains were created and the FPS were somewhat stable again, the engine was turned on. The test was completed with 10 and with 20 chains:

Loops	FPS
Paused	570
10	550
30	520
60	480
90	440
120	410
200	310

Table 5.1: FPS with 10 chains (1000 bodies)

Loops	FPS
Paused	490
10	450
30	380
60	320
90	170
120	60

Table 5.2: FPS with 20 chains (2000 bodies)

When the engine gets turned on, one has to consider that the base loop also starts running at 60 times per second. A cost of around 20 FPS for using 10 loops with almost 1000 joints (each of which is made up out of 5 constraints) seems pretty decent. Game engines often allow even less than 10 loops for their solvers to further reduce the performance impact [11].

5.3 Demos

For easy testing and playing around, I created some constraint demonstrations. These can be spawned into the world by pressing the respective button in the debug GUI and are very useful when trying to see if changes have an impact on more complex systems. One demonstration is included for each joint type, and then a few more complex systems were implemented:

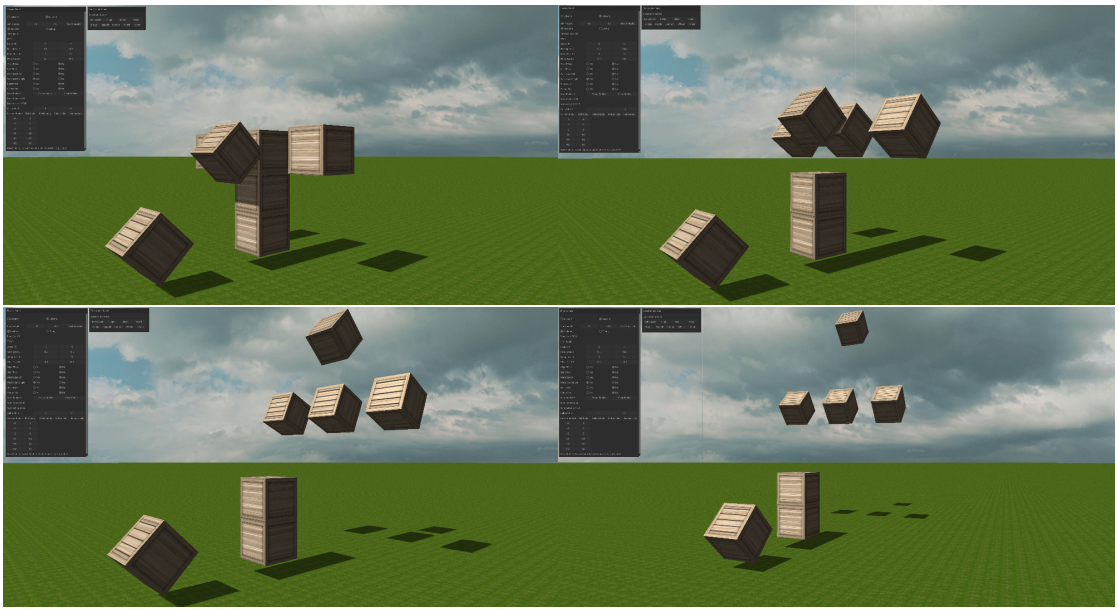


Figure 5.3: A slider joint cannon that uses the motor to shoot away a fixed joint projectile

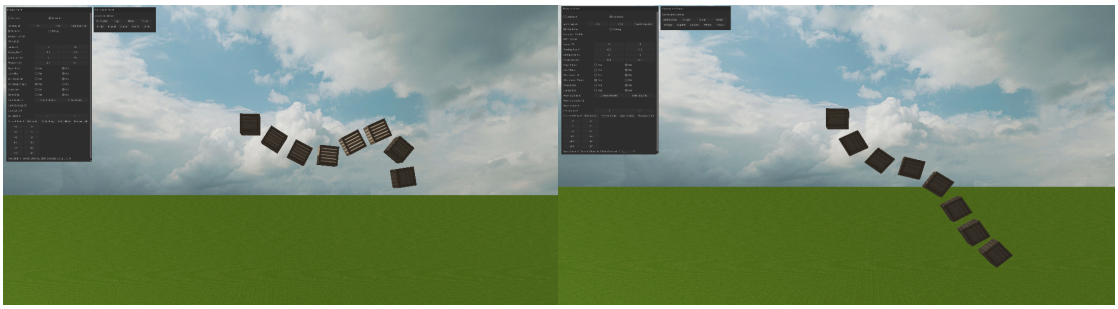
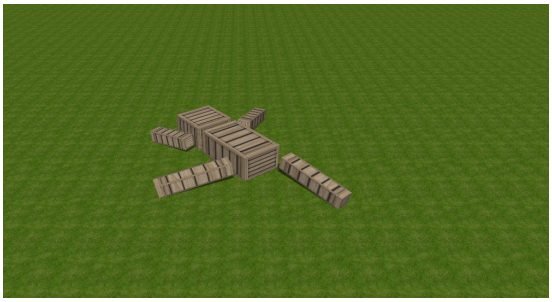


Figure 5.4: A hinge joint chain moved by a motor



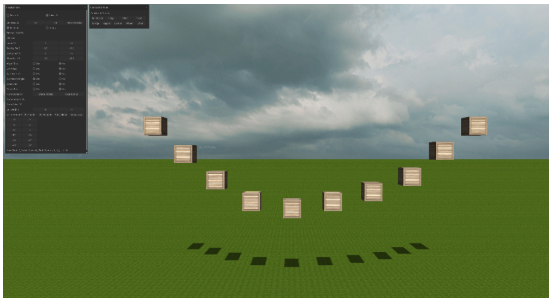


Figure 5.5: Very crude attempt at ragdoll physics

Figure 5.6: A suspension bridge using distance constraints

Figure 5.7: A self driving wheel made up of a motor-driven hinge joint and some fixed joints

6 Conclusion and future work

Concerning future work, there are lots of possibilities for improvement – Sequential Impulses has many advantages, but it is not perfect. The probably most impactful change would be better error correction using position based projection methods such as the ones described in [2]. This should increase stability of more complex systems and should fix errors as described in 3.4, at the cost of some performance. A good approach would be to implement both Baumgarte stabilization and position projection and allow the user to enable either one depending on the situation.

To increase robustness and accuracy, more solvers can be implemented. For example, in this implementation, the translation and rotation constraints of a hinge or slider joint were separated into 2 blocks. Solving them iteratively, they tend to fight against each other, which leads to reduced stiffness. Direct MLCP solvers like the ones mentioned in 2 can be used to solve larger blocks of constraints at once, allowing us to solve the 5 constraints making up a hinge or slider joint simultaneously. Just as with the error correction, allowing the user to dynamically change what solvers are used would allow for the greatest amount of flexibility, like it is done in the Bullet engine.

Another thing that can be done is warmstarting. By using the impulses accumulated during the previous frame as input for the current frame, the solver convergence should be sped up a bit. However, while this is especially important for box stacking, a quick test didn't visibly improve joint chains, so I decided against implementing it.

In conclusion, Sequential Impulses seems to be a great solver to start with when building a physics engine. It is intuitive and not complicated, which makes the implementation quite comfortable. The same is true for the involved math, where only the constraint derivations can be quite tough - this needs to be done for all Lagrange-based solvers though, and is not a property of SI. As an iterative solver, the performance is decent while still delivering good results for simple systems. The fact that the VPE provided an already fully functional framework and collision was already properly handled in the engine was a huge help, as I didn't have to worry about the structure of the project at all. Additionally, the collision part is usually the major challenge, and a lot of literature is focused mainly on this topic.

Bibliography

[1] E. Catto, *Understanding Constraints.* Game Developers Conference 2014 Slides, 2014.

[2] H. Garstenauer, *A Unified Framework for Rigid Body Dynamics.* Master thesis, Johannes Kepler Universität Linz, 2006.

[3] D. Baraff, "Fast contact force computation for nonpenetrating rigid bodies," in *Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '94, (New York, NY, USA), p. 23–34, Association for Computing Machinery, 1994.

[4] S. C. Billups and K. G. Murty, "Complementarity problems," *Journal of Computational and Applied Mathematics*, vol. 124, no. 1, pp. 303–318, 2000.

[5] M. Müller, B. Heidelberger, M. Hennix, and J. Ratcliff, *Position Based Dynamics.* Journal of Visual Communication and Image Representation, Volume 18, Issue 2, Pages 109-118, 2007.

[6] K. Erleben, *Stable, Robust, and Versatile Multibody Dynamics Animation.* PhD thesis, University of Copenhagen, 2004.

[7] D. Baraff, "Linear-time dynamics using lagrange multipliers," *Proc. of ACM SIGGRAPH*, 07 2000.

[8] A. Witkin, *An Introduction to Physically Based Modeling: Constrained Dynamic.* Journal of Visual Communication and Image Representation, Volume 18, Issue 2, Pages 109-118, 2007.

[9] E. Catto, "Iterative dynamics with temporal coherence," *Game Developer Conference*, pp. pages 1–24, 2005.

[10] K. Murty, *Linear Complementarity, Linear and Nonlinear Programming.* Heldermann Verlag, 1988.

[11] E. Coumans, *Exploring MLCP and Featherstone Solvers.* Game Developers Conference 2014 Slides, 2014.

[12] R. Featherstone, "The calculation of robot dynamics using articulated-body inertias," *The International Journal of Robotics Research*, vol. 2, no. 1, pp. 13–30, 1983.

[13] B. V. Mirtich, *Impulse-based Dynamic Simulation of Rigid Body System*. PhD thesis, University of California at Berkeley, 1996.

[14] E. Catto, *Modeling and Solving Constraints*. Game Developers Conference 2009 Slides, 2009.

[15] E. Catto, *Fast and Simple Physics using Sequential Impulses*. Game Developers Conference 2006 Slides, 2006.

[16] J. Baumgarte, "Stabilization of constraints and integrals of motion in dynamical systems," *Computer Methods in Applied Mechanics and Engineering*, vol. 1, no. 1, pp. 1–16, 1972.

[17] D. Chappui, "Constraints derivation for rigid body simulation in 3d." `https://danielchappuis.ch/download/ConstraintsDerivationRigidBody3D.pdf`, 2013.

[18] M. Tamis and G. Maggiore, "Constraint based physics solver." `http://mft-spirit.nl/files/MTamis_ConstraintBasedPhysicsSolver.pdf`, 2015.

[19] R. Smith, "Open dynamics engine." `https://www.ode.org/`, 2004.

[20] D. Negrut and M. Kwarta, "Using the complementarity and penalty methods for solving frictional contact problems in chrono: Validation for the cone penetration test." `https://sbel.wiscweb.wisc.edu/wp-content/uploads/sites/569/2018/05/TR-2016-16.pdf`, 2017.