# Comparison between Projected Gauss–Seidel and Sequential Impulse Solvers for Real-Time Physics Simulations

Marijn Tamis

July 1, 2015 v1.01

**Abstract**

The increase in popularity of rigid body dynamics for games has lead to the appearance of various physics solvers, many with a lot of similarities. This has made it difficult to choose the best fitting solver for a particular game or engine. This paper aims to give an overview of two methods, projected Gauss-Seidel and sequential impulse, and compares them to motivate the replacement of the projected Gauss-Seidel solver in Crystal Dynamic's Foundation engine with a sequential impulse solver.

## 1 Introduction

Rigid body simulation has become increasingly important for games, and game engines in general. It has moved from simulating simple environmental effects and props in the game world, to a feature required for game play.

Iterative solvers have become the method of choice for games and real-time simulations to solve constrained rigid body systems with contact, due to their speed and ability to exploit temporal coherence. [1] describes a popular algorithm that utilizes the above mentioned properties. The algorithm solves for the constraint forces needed to keep the constraints satisfied each simulation step using the projected Gauss-Seidel method. This algorithm was used for numerous titles produced with the Foundation engine including Tomb Raider and Deus Ex 3 [3].

The author of that paper has continued to work in the field of rigid body dynamics and presented the sequential impulse algorithm to iteratively solve rigid body systems in [2].

The goal of this document is to compare the presented algorithms from [1] and [2] to decide if it is beneficial to implement the sequential impulse algorithm for the Foundation engine. While comparing the two similar algorithms we will focus on the following aspects:

1. Maintainability.
2. Features and flexibility.
3. Performance.
4. Stability and correctness.

- In Section 2 we give a brief overview of the projected Gauss-Seidel (PGS) solver.
- In Section 3 we give a brief overview of the sequential impulse (SI) solver.
- In Section 4 compare the two algorithms.

It is assumed that the reader is already familiar with the subject as we will not go in to detail about the solvers.

## 2 Overview of the Projected Gauss-Seidel Solver

In this section we give a brief overview of how a PGS solver is setup. We refer to [1] or our previous work [4] for a more in in depth review of this solver.

We need to express the dynamics problem in a single equation of the form $\mathbf{A}\boldsymbol{\lambda} = \mathbf{b}$ so the PGS solver can find a solution. To do this we will take the kinematics equations for all the bodies in the system and the constraint equations linking these bodies, and put them together in one large system equation.

- In Section 2.1 we show how we combine the properties of individual bodies and constraints to express the whole system.
- In Section 2.2 we use the equations of motion to solve the system.
- In Section 2.3 we discuss some optimizations specific to this solver.

### 2.1 System Variable Definitions

The system velocity vector combines both the linear velocities ($\mathbf{v}_i$) and the angular velocities ($\boldsymbol{\omega}_i$) of all the bodies in the simulation in a column vector:

$$\mathbf{v} = \begin{bmatrix} \mathbf{v}_1 \\ \boldsymbol{\omega}_1 \\ \vdots \\ \mathbf{v}_n \\ \boldsymbol{\omega}_n \end{bmatrix} \tag{1}$$

We combine the inverse masses ($m_i^{-1}$) and the inverse inertia tensors ($\mathbf{I}_i^{-1}$) similarly to form the system mass matrix:

$$\mathbf{M}^{-1} = \begin{bmatrix} [m_1^{-1}\mathbf{D}] & \mathbf{0} & \dots & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{I}_1^{-1} & \dots & \mathbf{0} & \mathbf{0} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ \mathbf{0} & \mathbf{0} & \dots & [m_n^{-1}\mathbf{D}] & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & \mathbf{I}_n^{-1} \end{bmatrix} \tag{2}$$

where $\mathbf{D}$ is the $3 \times 3$ identity matrix.

We express a constraint limiting a single degree of freedom between bodies $a$ and $b$ as a function of their positions and orientations:

$$\mathrm{c}\left(\mathbf{x}_a, \mathbf{q}_a, \mathbf{x}_b, \mathbf{q}_b\right) = 0 \tag{3}$$

The solver only supports pairwise constraints for performance and memory optimization reasons.

We solve at velocity level so we use the derivative of c(...) to express the velocity constraint:

$$\dot{\mathrm{c}}\,(\ldots) = \frac{d}{dt}\,\mathrm{c}\,(\ldots) = \frac{-\beta\,\mathrm{c}\,(\ldots)}{\Delta t} \tag{4}$$

Baumgarte stabilization $-\beta\,\mathrm{c}\,(\ldots)\,\frac{1}{\Delta t}$ is added to account for penetration and numerical drift.

We split $\mathbf{v}$ away from the velocity constraint to obtain the Jacobian:

$$\dot{\mathrm{c}}\,(\ldots) = \mathbf{j}\mathbf{v} \tag{5}$$

We combine the constraint Jacobians by stacking them on top of each other, giving us the system Jacobian matrix:

$$\mathbf{J} = \begin{bmatrix} \cdots & \mathbf{j}_1^{1\ldots6} & \cdots & \mathbf{j}_1^{7\ldots12} & \cdots \\ & & \vdots & & \\ \cdots & \mathbf{j}_c^{1\ldots6} & \cdots & \mathbf{j}_c^{7\ldots12} & \cdots \end{bmatrix} \tag{6}$$

Note that the parts of $\mathbf{j}$ should be aligned horizontally to match the masses of the affected body in the mass matrix.

We also combine the Baumgarte stabilization terms in a vector:

$$\begin{aligned} \boldsymbol{\zeta} &= \begin{bmatrix} \zeta_1 & \ldots & \zeta_c \end{bmatrix}^{\mathsf{T}} \\ \zeta_c &= \frac{-\beta_c\,\mathrm{c}_c\,(\ldots)}{\Delta t} \end{aligned} \tag{7}$$

Now we can write (4) for the entire system as:

$$\mathbf{J}\mathbf{v} = \boldsymbol{\zeta} \tag{8}$$

The constraint reaction forces

$$\mathbf{f}_c = \begin{bmatrix} \mathbf{f}_{c1} \\ \boldsymbol{\tau}_{c1} \\ \vdots \\ \mathbf{f}_{cn} \\ \boldsymbol{\tau}_{cn} \end{bmatrix} \tag{9}$$

can be calculated using the Jacobian matrix:

$$\mathbf{f}_c = \mathbf{J}^{\mathsf{T}}\boldsymbol{\lambda} \tag{10}$$

Where $\boldsymbol{\lambda}$ is a vector containing the unknown signed magnitudes of the constraint forces.

## 2.2 Equations of Motion

Now we will use the equations of motion and the definitions from the previous section to show how we can solve for $\boldsymbol{\lambda}$.

The current velocities of the bodies may violate the constraints. We want to calculate a constraint force that, after applying it for the duration of a time step, corrects the the velocities so that:

$$\mathbf{J}\mathbf{v}_2 = \boldsymbol{\zeta} \tag{11}$$

Where $\mathbf{v}_2$ is the system velocity of the next frame.

We approximate $\mathbf{v}_2$ using an Euler step:

$$\mathbf{v}_2 = \mathbf{v}_1 + \dot{\mathbf{v}}\Delta t = \mathbf{v}_1 + \mathbf{M}^{-1}\mathbf{J}^{\mathsf{T}}\boldsymbol{\lambda}\Delta t \tag{12}$$

where $\mathbf{v}_1$ is the current system velocity. Substituting (12) in to (11) gives us:

$$\mathbf{J}\mathbf{M}^{-1}\mathbf{J}^{\mathsf{T}}\boldsymbol{\lambda} = \frac{1}{\Delta t}\boldsymbol{\zeta} - \mathbf{J}\frac{1}{\Delta t}\mathbf{v}_1 \tag{13}$$

We rewrite this equation to use it with the Gauss-Seidel method[5]:

$$\mathbf{A}\boldsymbol{\lambda} = \mathbf{b} \tag{14}$$

with

$$\begin{aligned} \mathbf{A} &= \mathbf{J}\mathbf{M}^{-1}\mathbf{J}^{\mathsf{T}} \\ \mathbf{b} &= \frac{1}{\Delta t}\boldsymbol{\zeta} - \mathbf{J}\frac{1}{\Delta t}\mathbf{v}_1 \end{aligned} \tag{15}$$

We call $\mathbf{A}$ the effective mass.

We can specify an interval of valid values for each element in $\boldsymbol{\lambda}$, using projected Gauss-Seidel, to support inequality constraints.

### 2.3 Optimizations

Inspecting the system mass matrix (2) and the system Jacobian matrix (6) it is clear that these matrices are sparse. This opens possibilities for optimization.

Every row of $\mathbf{J}$ contains up to 12 non-zero values. $\mathbf{M}^{-1}$ contains only one non-zero value for rows $\in \{6i + j \mid i \in \mathbb{Z}_{\geq 0}, j \in \{1, 2, 3\}\}$ and 3 non-zero values for rows $\in \{6i + j \mid i \in \mathbb{Z}_{\geq 0}, j \in \{4, 5, 6\}\}$

Using this sparsity we can significantly optimize the calculation of the effective mass $\mathbf{A}$ and the matrix vector products within the Gauss-Seidel iteration.

The convergence of the Gauss-Seidel method can be improved with an initial guess for $\boldsymbol{\lambda}$. We can exploit the temporal coherence of the system by reusing the calculated $\boldsymbol{\lambda}$ from the previous frame. Care should be taken that the order of the elements in $\boldsymbol{\lambda}$ is kept with respect to the order of the other system variables, as constraints might be present in a different order, or vanish, between frames.

## 3 Overview of the Sequential Impulse Solver

In this section we give a brief overview of the sequential impulse algorithm.

- In Section 3.1 we show how to calculate the impulse to solve a single constraint.
- In Section 3.2 we show how to iteratively solve the entire system using those impulses.

## 3.1 Constraint Impulse

We have a constraint between bodies $a$ and $b$, using the same formulation as in Equations (3)–(5). We combine the velocities and angular velocities of bodies $a$ and $b$ in a single vector:

$$\mathbf{v} = \begin{bmatrix} \mathbf{v}_a \\ \boldsymbol{\omega}_a \\ \mathbf{v}_b \\ \boldsymbol{\omega}_b \end{bmatrix} \tag{16}$$

And we combine the inverse mass and inverse moments of inertia of bodies $a$ and $b$ in a matrix:

$$\mathbf{M}^{-1} = \begin{bmatrix} [m_a^{-1}\mathbf{D}] & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{I}_a^{-1} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & [m_b^{-1}\mathbf{D}] & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{I}_b^{-1} \end{bmatrix} \tag{17}$$

The constraint impulses affecting bodies $a$ and $b$ is combined in a vector:

$$\mathbf{p}_c = \begin{bmatrix} \mathbf{p}_a \\ \boldsymbol{\ell}_a \\ \mathbf{p}_b \\ \boldsymbol{\ell}_b \end{bmatrix} \tag{18}$$

where $\mathbf{p}$ is the linear impulse and $\boldsymbol{\ell}$ is the angular impulse.

We will now setup a system of equations that will let us solve for $\mathbf{p}_c$:

$$\mathbf{v}_2 = \mathbf{v}_1 + \mathbf{M}^{-1}\mathbf{p}_c \tag{19}$$

$$0 = \zeta - \mathbf{j}\mathbf{v}_2 \tag{20}$$

$$\mathbf{p}_c = \mathbf{j}^\mathsf{T}\lambda\Delta t \tag{21}$$

(19) is a simple Euler step where $\mathbf{v}_1$ contains the pre-impulse velocities and $\mathbf{v}_2$ the post-impulse velocities. (20) is the velocity constraint with $\zeta = \frac{-\beta\, c(\ldots)}{\Delta t}$. And (21) defines the relation between the signed magnitude of the constraint force and the constraint impulse.

Solving for $\lambda$ gives us:

$$\lambda = \frac{-\mathbf{j}\mathbf{v}_1 + \zeta}{\mathbf{j}\mathbf{M}^{-1}\mathbf{j}^\mathsf{T}\Delta t} \tag{22}$$

which can be substituted again into (21) to get $\mathbf{p}_c$:

$$\mathbf{p}_c = \mathbf{j}^\mathsf{T}\left(\frac{-\mathbf{j}\mathbf{v}_1 + \zeta}{\mathbf{j}\mathbf{M}^{-1}\mathbf{j}^\mathsf{T}}\right) \tag{23}$$

## 3.2 Solving for a Global Solution

The previous section showed how to solve a single constraint. Solving all the constraints individually will not yield a correct result when multiple constraints act on a single body. We need to iterate over all constraints multiple times, so that the impulses can propagate, to arrive at a global solution.

Special care should be taken with clamping the impulse for inequality constants. The solver will be unable to recover from impulses that are too big if $\lambda$

is clamped to $0 < \lambda < +\infty$. Instead of clamping the corrective $\lambda$ each time it is calculated, the accumulated $\lambda$ should be clamped, as described in Algorithm 1, so that the corrective $\lambda$ can be negative.

We can also use the $\lambda$s computed in the previous time step to speed up convergence similarly to warm starting the Gauss-Seidel algorithm. To do this we initialize the accumulated $\lambda$ to the value from the previous step, and apply the impulse computed from these $\lambda$s to the bodies before the first iteration. The sequential impulse algorithm with warm start is shown in Algorithm 1.

```
foreach Constraint do
    // $\lambda_{\text{accumulated}}$ is tracked separately for each constraint
    if Previous frame $\lambda$ is available then
        $\lambda_{\text{accumulated}}$ = previous frame $\lambda$;
        Update body velocities using the impulse calculated from
        $\lambda_{\text{accumulated}}$;
    else
        $\lambda_{\text{accumulated}} = 0$;
    end
end
while Not satisfied with global solution do
    foreach Constraint do
        Calculate $\lambda_{\text{corrective}}$;
        $\lambda_{\text{old}} = \lambda_{\text{accumulated}}$;
        $\lambda_{\text{accumulated}} = \lambda_{\text{accumulated}} + \lambda_{\text{corrective}}$;
        Clamp $\lambda_{\text{accumulated}}$ to allowed interval;
        Update body velocities using the impulse calculated from the
        difference $\lambda_{\text{accumulated}} - \lambda_{\text{old}}$;
    end
end
foreach Constraint do
    Store $\lambda_{\text{accumulated}}$ so it can be retrieved next frame.
end
```

**Algorithm 1:** Sequential impulse algorithm with warm start.

## 4 Comparison

In this section we will compare the solvers described in Sections 2 and 3. Both solvers have been implemented, in a roughly equal amount of time, in a test application to give this comparison a practical basis.

We have divided the discussion in several categories:

- In Section 4.1 we compare the algorithms to show that they can be directly related.
- In Section 4.2 we discuss the architecture and maintainability of the solvers.
- In Section 4.3 we discuss the features supported by the solvers, and the flexibility to extend them.
- In Section 4.4 we discuss the performance of the solvers.

- In Section 4.5 we discuss the stability and correctness of the solvers.

## 4.1 Similarity

We can show the similarity between the two solvers if we inspect the inner loop of the Gauss-Seidel method and compare it with (22).

We begin with some definitions to deal with the fact that sequential impulses updates the velocity of the rigid bodies within the iterations. We have $n$ rigid bodies and $m$ constraints in our system.

The vector $\boldsymbol{\eta}^{(l)}$ consists of the accumulated $\lambda$s for the system at step $l \leq m$ of the current iteration. With step $l$ we mean the step in which we are going to calculate the impulse for constraint $l$.

$$\boldsymbol{\eta}^{(l)} = \begin{bmatrix}\lambda_1, & \ldots, & \lambda_{l-1}, & \overline{\lambda}_l, & \ldots, & \overline{\lambda}_m\end{bmatrix} \tag{24}$$

Where we indicate the $\lambda$s that are untouched in step $l$ of the iteration with $\overline{\lambda}$.

From (24) we can deduce this relationship:

$$\boldsymbol{\eta}_i^{(l+1)} = \boldsymbol{\eta}_i^{(l)} \leftrightarrow i \neq l \tag{25}$$

$\boldsymbol{\upsilon}^{(l)}$ contains the system velocity at step $l$ of the current iteration:

$$\boldsymbol{\upsilon}^{(l)} = \overline{\mathbf{v}} + \mathbf{M}^{-1}\mathbf{J}^{\mathsf{T}}\boldsymbol{\eta}^{(l)}\Delta t \tag{26}$$

where $\overline{\mathbf{v}}$ is the system velocity from before the first iteration.

Using these definitions we now rewrite (22):

$$\boldsymbol{\eta}_i^{(i+1)} = \boldsymbol{\eta}_i^{(i)} + \frac{-\mathbf{j}\boldsymbol{\upsilon}^{(i)} + \zeta}{\mathbf{j}\mathbf{M}^{-1}\mathbf{j}^{\mathsf{T}}\Delta t} \tag{27}$$

Where $\mathbf{j}$ is extended to multiply with the other system variables similar to a single row of $\mathbf{J}$ as described by (6) and $\mathbf{M}^{-1}$ is the system mass as described by (2).

Now we write the Gauss-Seidel iteration for $\mathbf{A}\boldsymbol{\lambda} = \mathbf{b}$ using $\boldsymbol{\eta}$[5]:

$$\boldsymbol{\eta}_i^{(i+1)} = \left(\mathbf{b}_i - \sum_{0<j<l}\left(\mathbf{A}_{ij}\boldsymbol{\eta}_j^{(i)}\right) - \sum_{l<j<n}\left(\mathbf{A}_{ij}\boldsymbol{\eta}_j^{(i)}\right)\right)\mathbf{A}_{ii}^{-1} \tag{28}$$

We can rewrite this with some algebraic manipulation and show that this is equivalent to (27) if we fill in $\mathbf{A}$ and $\mathbf{b}$ from (15):

$$\begin{aligned}\boldsymbol{\eta}_i^{(i+1)} &= \left(-\mathbf{A}_i \cdot \boldsymbol{\eta}^{(i)} + \mathbf{A}_{ii} \cdot \boldsymbol{\eta}_i^{(i)}\right)\mathbf{A}_{ii}^{-1} + \mathbf{b}_i\mathbf{A}_{ii}^{-1} \\ &= \boldsymbol{\eta}_i^{(i)} + \frac{\boldsymbol{\zeta}_i - \mathbf{J}_i\boldsymbol{\upsilon}^{(i)}}{\mathbf{J}_i\mathbf{M}^{-1}\left(\mathbf{J}_i\right)^{\mathsf{T}}\Delta t}\end{aligned} \tag{29}$$

## 4.2 Maintainability

Both algorithms are simple, in their basic form, and not much code is needed to implement them. Implementations that follow the basic algorithm naively are generally easy to maintain as they are easier to understand. They will become more complex once the solver is adjusted for performance. We will look at the differences between SI and PGS solvers assuming the implementation is naive. This is a good starting point for the analysis, but it is still a subjective matter.

It is logical to have an object class for each constraint type for the SI solver, so that the iteration as described in Algorithm 1 can be written without constraint specific code in it, keeping the constraints and the solver separated. This is harder to do for a PGS solver as all the information has to be present in a similarly formatted structure within the vectors and matrices. It can still be achieved with objects that set their respective parts of the data structures, but it brings some bookkeeping code with it. It can also be achieved by referencing to the objects from within the projected Gauss-Seidel iterations, but this will start to look more like an SI solver. The ability to have a good view of the constraint while debugging is another advantage of using objects for constraints, which is not possible when using bigger structures containing multiple constraints.

A solver is hard to maintain when it is not understood well. So it is important that the basic algorithm is understood by all programmers who maintain the solver. Not all programmers might have the mathematical background to understand the solvers completely. For these programmers it is generally hard to think about the rigid body simulation in the form of abstract mathematical constructs. Presenting them the PGS solver, which uses big matrices to describe the whole system at once, may not be as effective as presenting them the SI solver, which looks at the constraints individually and explicitly describes the iterations. It is even possible to describe the iterations within the SI solver without equations but with abstraction, as illustrated by the pseudo-code "Calculate $\lambda_{\text{corrective}}$;" in Algorithm 1, keeping the algorithm and the math separated.

The SI solver describes the solution to the problem of interacting constraints in a more intuitive way. Once it is understood how to solve a single constraint, the next intuitive step would be to solve all constraints multiple times to propagate forces, not to build a matrix equation and apply projected Gauss-Seidel.

Some optimization are also easier to discover with the SI algorithm. If the PGS solver is implemented using the big matrix approach the matrix multiplications have to be inspected to see the sparseness interact. Taking advantage of this sparseness can introduce complicated code (e.g. specialized sparse matrices) which can make it hard to maintain, or it will create a divergence between the basic algorithm and the implementation. The sparseness optimizations (for the effective mass for example) are implicitly described in the SI solver.

The properties of the constraints, like the effective mass $\mathbf{j}\mathbf{M}^{-1}\mathbf{j}^{\mathsf{T}}$ and $\zeta$, have to be calculated explicitly for each constraint for (22) with the SI solver. This might be a costly operation, and these values do not change while iterating. The PGS algorithm expects you to calculate all these properties before iterating, while an extra constraint initialization step has to be added to Algorithm 1 to utilize this optimization. Also, the effective mass will be implicitly calculated from the $\mathbf{J}$ and $\mathbf{M}^{-1}$ matrix in the PGS solver without extra code. While you get the chance of optimizing these calculations for specific constraints in the SI solver, it also introduces code duplication.

A good implementation of both solvers will be closely competing for good maintainability. However the fact that the SI solver is easier to understand and that an optimized implementation can be closely related to the original algorithm makes it more elegant.

## 4.3 Features and Flexibility

The supported features and flexibility of the solvers are directly related to their implementations. We continue with the assumptions from the previous section about the implementations.

The PGS solver limits the constraints to act on only two bodies so that the space needed to store a row of the **J** matrix does not depend on the constraint type it contains. This makes it easy to store **J** sparsely in a continuous array. The SI solver, which represents each constraint type as a class, is not constrained by these requirements as the constraint object can store its own **j**. This makes it possible to use constraints that act on more than two bodies without modifying the solver. This opens up new possibilities for constraints that can be used for game mechanics, like pulleys with bodies on both ends of the rope.

Representing the constraints using classes has more advantages. For instance, because each constraint type can execute its own routine, it is possible to solve more complex constraints in a single iteration step. A constraint keeping two anchor points overlapped, for example, needs to constrain 3 degrees of freedom. In a PGS solver it is only possible to use 3 separate rows in **J** to achieve this. While using an SI solver we can solve the whole constraint at once in the constraint routine. This requires to calculate the inverse of a $3 \times 3$ matrix (once per time step, not per solver iteration), but this improves convergence, it removes other duplicated calculations, and it will save 2 velocity updates per iteration.

The flexibility of using classes can also be used to implement other features specific to a constraint, without modifying the rest of the solver. Adding softness to constraints, for example, requires to change the calculation of the systems effective mass. The PGS solver has to be modified as it calculates the effective mass for the entire system in one place, while the changes to the SI solver are local to the constraint class.

## 4.4 Performance

A lot of the differences in optimization possibilities have already been covered in the previous sections, which we will not repeat. This section will discuss the remaining differences between the solvers regarding performance.

If implemented naively, the entire updated system velocity is recalculated for each update of $\boldsymbol{\lambda}$ within the Gauss-Seidel iteration. This is wasteful as the velocity of only two bodies can change in each iteration step. This can be optimized, but the SI algorithm implicitly includes this optimization as the impulses are directly applied to the affected bodies, calculating the new system velocity automatically.

The effective mass calculated in the naive PGS solver is stored in a $m \times m$ matrix for $m$ constraints. This can take up a lot of memory, and does not scale well. Exploiting the sparseness of the matrix will reduce the size to:

$$\text{number of non-zero elements in } \mathbf{A} = \sum_{i}^{m} \sum_{j}^{m} \delta\left(\mathbf{J}_i \cdot \mathbf{J}_j\right) \tag{30}$$

$$\delta(x) = \begin{cases} 1, & x \neq 0 \\ 0, & x = 0 \end{cases}$$

excluding the memory needed for bookkeeping. The SI solver only stores one effective mass per constraint (equivalent to the diagonal of the PGS effective mass matrix), making it scale linearly with $m$.

Each constraint can execute its own routine in the SI solver, as mentioned in section 4.3. This is usually implemented with virtual function calls, which can bring overhead with them. In most situations there are a magnitude more contact constraints than other joints present in the simulation. It is trivial to split the loop that iterates over all constraints into separate loops for contacts and other joints. The contact constraints do not need to use virtual functions, as they all execute the same routine. The overhead of the virtual functions solving the remaining constraints is a small cost for the extra flexibility they provide.

The contact constraints can be optimized similar to the discussion in Section 4.3 about the position constraint. This reduces computation, especially when using two friction constraints per contact, since calculations can be reused between the contact and friction constraints.

A good side effect of storing the system information in big vectors and matrices is data locality for the cache. This is not a big benefit for systems that are small enough to fit in the cache entirely and it introduces the overhead of creating these structures. However it is not as trivial to improve data locality in the SI solver as it is with the PGS solver when it does become a problem.

A downside to the extra flexibility of the SI solver is that the algorithm is harder to map to a parallel architecture like a GPU. The PGS solver algorithm maps well to such an architecture if the Gauss-Seidel method is replaced with the Jacobi method. It is not an issue to consider for simulations where game play mechanics need access to the physics objects, as synchronization overhead between the CPU and GPU will reduce the performance gain from using the GPU. However this might be an issue for bigger simulations that can run without significant synchronization overhead.

### 4.5 Stability and Correctness

We compared the stability and visible correctness of various simulations between the two solvers without noticeable difference. This is not surprising given that the basic algorithms are mathematically equivalent. However there are cases where the SI solver allows for a formulation of constraints that results in better convergence and stability (eg. matrix constraints solving multiple rows of $\mathbf{J}$ at once).

Solving the friction along both axes perpendicular to the normal at once, in the SI solver, makes it possible to clamp the resulting impulse as a vector. This is a more natural way to clamp the impulse, resulting in a 'friction cone' instead of a 'friction pyramid'. No noticeable difference has been seen when comparing it to separate friction constraints, where one axis is chosen in the direction of the relative velocity.

Both solvers are virtually equal in terms of stability with contact and stacking.

## 5 Conclusion

We have given a brief overview of two constraint based physics solvers in Sections 2–3. Then we showed that the two algorithms are mathematically equivalent in Section 4.1. Finally we discussed their differences in Sections 4.2–4.5.

We argued that the SI solver is easier to maintain because an optimized implementation is closer to the original algorithm than a PGS solver and also because the SI solver is easier to understand for people less familiar with the math. We also discussed that a naive implementation of SI is flexible in its definition of constraints, allowing more than two bodies to be affected by a single constraint and allowing for per-constraint flexibility in behavior and optimization. We also briefly discussed the advantages of the PGS solver for a GPU implementation.

So the SI solver is the better choice in most situations, even though it is shown in Section 4.1 that the solvers are mathematically equivalent. This concludes that the Foundation engine will benefit from moving to an SI solver.

## References

[1] E. Catto. Iterative Dynamics with Temporal Coherence. *Game Developer Conference*, January 2005.

[2] Erin Catto. Fast and simple physics using sequential impulses. In *Games Developer Conference*, 2006.

[3] Erin Catto. Soft constraints. In *Game Developers Conference*, 2011.

[4] Marijn Tamis and Giuseppe Maggiore. Constraint based physics solver. Published on August 8, 2014.

[5] Eric W. Weisstein. Gauss-seidel method. From MathWorld—A Wolfram Web Resource. Last visited on 31-2-2015.