Algorithmic and Architectural Gaming Design:
Implementation and Development
ISBN 978-1-4666-1634-9
p159-202

Chapter 8

Practical Introduction to Rigid Body Linear Complementary Problem (LCP) Constraint Solvers

Ben Kenwright
Newcastle University, UK
b.kenwright@ncl.ac.uk

Graham Morgan
Newcastle University, UK
graham.morgan@ncl.ac.uk

ABSTRACT

This chapter introduces Linear Complementary Problem (LCP) Solvers as a method for implementing real-time physics for games. We explain principles and algorithms with practical examples and reasoning. When first investigating and writing a solver, one can easily become overwhelmed by the number of different methods and lack of implementation details, so this chapter will demonstrate the various methods from a practical point of view rather than a theoretical one; using code samples and real test cases to help understanding.

INTRODUCTION

With the ever-increasing visual realism in today's computer-generated scenes, it should come as no shock that people also expect the scene to move and react no less realistically. With the computational power available today, the ability to run physically accurate real-time simulations is required to hold a players attention.

Simulating scenes, using physics-based methods, is important because it enables us to produce environments that respond to unpredictable actions and simulate situations that are indistinguishable from real life, e.g. buildings collapsing. However, it is very difficult to simulate reliably, large number of objects and complex articulated structures as shown in Figure 1.

DOI: 10.4018/978-1-4666-1634-9.ch008

Information Science REFERENCE

Copyright © 2012, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

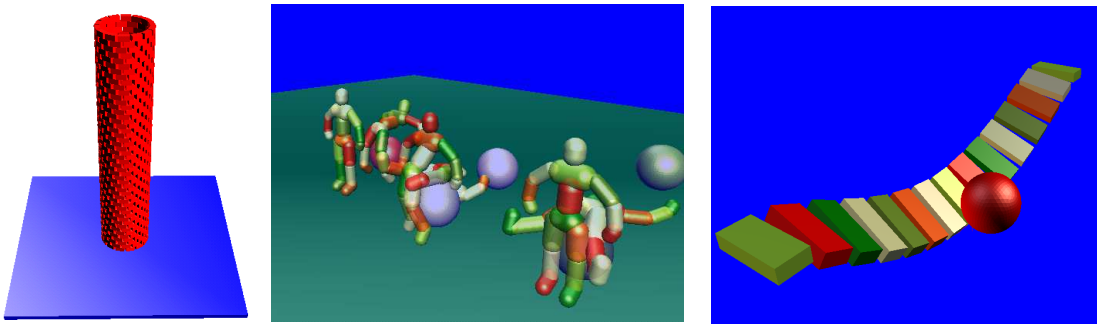

Figure 1. Simulation screenshots demonstrate stable stacking (left), articulated joints for characters (middle) and chains of objects (right).

Writing a flexible, scalable rigid body simulator is a challenging task because you need strong background knowledge in programming and Newtonian mechanics. While there are several approaches (i.e. penalty methods, impulse methods), solvers offer numerous advantages, such as requiring less user tuning and the ability to handle highly coupled configurations (e.g. large stacks or chains).

The reader, after being introduced to how solvers operate and how they are constructed, is introduced to dependent techniques; such as sparse matrices. And while we try and explain everything from the bottom-up, it is still required that the reader is at least familiar with basic Newton's laws and calculus techniques. After completing this chapter, the reader should have a basic understanding of what a solver is, how to implement one, and how to use it.

BACKGROUND

Rigid body dynamics

Rigid body dynamics is a well understood and documented field, and as such, will not be covered here. For background information, we recommend reading (Baraff, 1999; David H. Eberly, 2004; Hecker, 1998), which gives details on unconstrained dynamics and concepts such as body mass, acceleration, velocity and the equations of motion, which we use throughout this chapter.

While we introduce the reader to writing a practical LCP solver, there are also commercial and open source engines, which can be taken advantage of, and we would recommend them for the purposes of background knowledge. Some well-known LCP simulation engines are:

- Open Dynamics Engine (ODE), (Smith, 2004)
- PhysX (NVIDIA, 2011)
- Newton Game Dynamics (Jerez & Suero, 2003)
- Havok (Havok, 1998)
- Crisis Physics Engine (Vondrak, 2006).

Block matrix methods

The equations that make up our dynamic system and constraints can be large, cumbersome, error-prone and difficult to manage, and to help alleviate this problem, we represent them in matrix form. This gives us a more manageable high-level view of the system and its components, which is more intuitive to work with.

In our simulator, a large majority of the computation cost is in calculating an inverse matrix for our solution. As with real numbers, when you have 'ax=b' you can solve for 'x' by dividing both sides by 'a'

to get ‘x=b/a’, which is acceptable, as long as ‘a’ is not zero. Similarly, when working with matrices, we formulate the equation ‘AX=B’, and divide both sides by ‘A’ to get ‘X=B/A’. But instead of dividing both sides by ‘A’, we calculate the inverse of ‘A’ (i.e. A^{-1}) and multiply both sides, to achieve the same result.

This chapter uses numerical methods to calculate the inverse of the matrix, which is central to achieving a usable solver. However, on occasions, we are unable to calculate the inverse, i.e. we get a *singular* matrix, similar to a divide by zero with real numbers. When this occurs, usually some ill-conditioned configuration has developed or perhaps some numerical problem has occurred, which we detect as a singular matrix (i.e. determinant is zero) and we determine why it has arisen so that we can fix it and prevent it happening again.

Lagrange Multiplier Formulation

We have the unconstrained dynamic equations of motion from classical mechanics, which describes how the rigid bodies move, and a set of constraint conditions – which describe how they cannot move. We then combine these two equations and solve the unknowns by using a powerful technique of multivariable calculus, known as ‘Lagrange multipliers’.

We create the equations for our system using Newton’s second law ‘f = ma’, in combination with constraint equations which we form through differentiation and substitution to establish a combined problem, which is solved using Lagrange’s multiplier methods.

Equations of Motion

Each unconstrained body has six *Degrees of Freedom* (DOF), three for translation and three rotation. For a group of rigid bodies (m), the total DOFs is 6m. Since constraints restrict the relative motion, the total number of DOFs of a group of rigid bodies, with constraints, is less than 6m. The rigid body dynamics in collaboration with the constraint configurations form the *Equations of Motion* (EOM), that describe how the group of rigid bodies will move as time changes.

We can categorise the EOM into two types: *Maximal* and *Reduced* coordinates.

Maximal coordinates use Cartesian space, so each body has 6m state variables, and requires 6m-n constraint equations (where n represents the number of constraints). These constraints *explicitly* remove extraneous DOF through their formulation. For further reading on explicit constraint methods see (Shabana, 1994), and further details on maximal coordinate methods are available in (Baraff, 1999; David H. Eberly, 2004; Hecker, 1998).

Reduced coordinates use an *implicit* incorporation of the constraints to formulate the equations of motion. The system uses n state variables to represent the various DOF, so for example, if we have a single object which can only rotate around the ‘y’ axis (no translation or x-z rotation), then the system state only needs a single state variable to represent the system (i.e. the object’s angle). It has a major drawback, whereby for each unique configuration we need to derive by hand a set of equations for that particular arrangement.

Both *Maximal* and *Reduced* coordinate methods are able to run in ‘O(n)’ time. Maximal coordinates are more popular because of their modularity and straightforwardness to understand and implement. Although maximal coordinates operate in Cartesian space, we still sometimes need to use awkward conversions, to convert between constraint and Cartesian space. Maximal coordinates can drift due to

numerical errors and integration inaccuracies, in addition they need a minimum of 6m state variables to represent the system, so optimised methods need to be used to reduce memory and bandwidth impacts (sparse matrices). Due to the modular flexible nature of maximal coordinates, we use this method in this chapter, so we can formulate constraints once, and use them again and again for various configurations.

Simulation Approaches

Three main types of constraint methods exist: *Penalty methods*, *Impulse methods* and *Global methods*.

Penalty methods (springs) – are the easiest technique for formulating constraints, whereby the violated constraint error is fed back into the system as restoring forces to correct the error. They have the ability to be simple, fast and intuitive, and can be combined with other methods to add controllability. Their downside is that the constraints rely on error feedback forces, and suffer from stability issues, which require small time-steps (offline) or computationally expensive integration techniques to remain stable. The reader can refer to (Kaˇ, Nordenstam, & Bullock, 2003) for a practical example of penalty methods being used to create constraints.

Impulse methods (velocity impulses) – use instantaneous force changes, known as velocity impulses to implement constraints. These velocity impulses are repeatedly applied to each constraint, where you solve one constraint after the other, re-evolving the system to satisfy all constraint conditions, until the system converges, or you reach a maximum update limit. Further reading can be found in (Guendelman, Bridson, & Fedkiw, 2003) which presents a realistic impulse-based simulator with stacking.

Global methods (analytical methods) – which we use and apply in this chapter, involve computing the exact magnitude of force that will satisfy the constraints at every step of the simulation. It is accurate and requires minor parameter tuning by the user and can maintain stability for relatively large integration steps. It works, fundamentally, by constructing a linear system of the form.

SOLVER (LCP)

The Linear Complementarily Problem (LCP) is a special kind of problem that aims to find a solution to a set of equations, subject to constraint limits. The type of LCP we focus on in this chapter is the box-constrained LCP, which aims to find a solution to the form ‘y = Ax+b’, subject-to limits on ‘y’:

$$\begin{aligned} &\textit{where} && y = Ax + b \\ &\textit{subject to} && y \geq 0 \;\rightarrow\; x = x_{lower} \\ & && y \leq 0 \;\rightarrow\; x = x_{upper} \\ & && y = 0 \;\rightarrow\; x_{lower} < x < x_{upper} \end{aligned} \tag{1}$$

We can broadly classify LCP solvers into two method types, *iterative* methods and *pivoting* methods.

Pivoting methods use recursion; where the solution to the problem depends on solutions to smaller instances of the same problem, which can be solved in a finite number of steps. While pivoting methods can be fast, it is our experience that for a large number of constraints, they can produce erroneous results for perfectly valid systems due to floating point errors. Further reading and examples of pivoting methods are Lemke’s algorithm used in (David H. Eberly, 2004), and Dantzig’s algorithm, used in (Baraff, 1994).

Iterative methods alternatively do not terminate but converge on a solution finitely, where convergence depends on a number of factors such as the initial starting value. In addition, because iterative methods move closer to the solution with every update, if they were interrupted early, the current result can be

good enough for the simulation to continue. Furthermore, we can take advantage of the starting guess and coherency between frame updates to accelerate convergence, by feeding the previous frame's solution to the next. This property makes them ideal for real-time applications, where we can break out at varied times, trading accuracy for speed. Finally, iterative methods in practice are able to find acceptable results in ill-conditioned or singular configurations; due to contacts or overly constrained systems that allow the simulation to continue and recover.

Further reading about LCP methods for solving can be found in (Cottle, Pang, & Stone, 1992).

To restate, we use iterative methods because:

- The stopping criteria can be adjusted to trade accuracy for performance.
- In practice, they are more stable and reliable.
- They are simple to implement compared to other direct methods.
- There is a greater potential for optimisation (exploiting matrix sparsity for speedups).

Two popular iterative methods for solving *systems of linear equations* are '*Gauss-Seidel*' and the '*Jacobi method*', which can be modified to handle equality constraints for our complementary problems, such as, contacts. For this chapter, we use a modified '*Gauss-Seidel*' algorithm, called the '*Projected Gauss-Seidel*', which offers a simple and intuitive implementation with good convergence rates. An in-depth explanation on how systems of linear and complementary equations are solved can be obtained by reading (Hagger, 1988), (Cottle et al., 1992) and (Erleben, 2004), also for a more detailed explanation of the Gauss-Seidel and its differences to projected Gauss-Seidel, see (Catto & Park, 2005).

The *Gauss-Seidel* equation is shown below, followed by its implementation in code.

$$x_i^{(k+i)} = \frac{1}{a_{ii}}\left(b_i - \sum_{j=1}^{i-1} x_j^{(k+1)} - \sum_{j=i+1}^{n} x_j^k \right) \tag{2}$$

where the subscript i and j indicates the column and row of the matrix elements from our linear equation Ax=b. It works by starting from some initial value (e.g. 0), then iteratively updating the answer repeatedly using the result from the previous step to converge on a solution. In the equation above, we have x^k as our current result, and x^{k+1} as the next.

The convergence on an acceptable answer depends upon the size and complexity of the configuration, where it can take anywhere from two or three iterations to hundreds, depending on the topology and initial starting value. For example, highly coupled configurations such as stacks of objects or long chains take longer to converge than less densely coupled ones.

We add an extra step to our vital *Gauss-Seidel* method to incorporate boundary conditions and enforce the complementary constraints with an additional projection step:

$$x_j = \max(\min(x_j, x_j^{upper}), x_j^{lower}) \tag{3}$$

Stop Condition

The sample code uses a fixed iteration count, but a check can be added to determine if the solution is within a certain tolerance and provide an early breakout. We can calculate this value using the equation below:

$$\frac{\| b - Ax^{(k)} \|}{\| b \|} < \varepsilon \tag{4}$$

Acceleration or Velocity Level

We can broadly classify LCP solvers into two main types, *acceleration-based* (Baraff, 1989; 1994; Lötstedt, 1984) or *velocity-based* (Anitescu & Potra, 1996; Stewart & Trinkle, 1996), where the solver is classified according to the level it operates on to solve its constraints. In the next few sections, we review both the acceleration and velocity level solvers, outlining their similarities and differences, but towards the end of this chapter, we will focus on *velocity-based* solvers due to their added simplicity and practicality.

We give a brief comparison of both methods and review their advantages and disadvantages. To begin with, we demonstrate the two main sets of equations and the steps for formulating constraints at the various levels to illustrate their differences.

Velocity
$JM^{-1}J^T\lambda + J\dot{q} + JM^{-1}F_{ext} \geq 0$ using $\dot{q}_{n+1} = \dot{q}_n + M^{-1}(F_{ext} + F_c)\Delta t$ $F_c = J^T\lambda$

Acceleration
$JM^{-1}J^T\lambda + \dot{J}\dot{q} + JM^{-1}F_{ext} \geq 0$ using $\ddot{q}_{n+1} = M^{-1}\left(F_{ext} + F_c\right)$ $F_c = J^T\lambda$

The constraint formulation steps are as follows:

Velocity
1. Create positional constraint C. 2. Differentiate C with respect to time to obtain the velocity constraint $\dot{C}$. 3. Isolate and extract the Jacobian from $\dot{C}$.

Acceleration
1. Create positional constraint C. 2. Differentiate C with respect to time to obtain the velocity constraint $\dot{C}$. 3. Isolate and extract the Jacobian from $\dot{C}$. 4. Solve for the derivative of the Jacobian ($\dot{J}$) w.r.t. time.

From the constraint formulation steps above, the reader may notice that the initial steps are very similar, but the acceleration level requires additional work to calculate the Jacobian derivative. The velocity approach is basically a subset of the acceleration level, where it moves the acceleration problem into the LCP integration step in order to obtain a discrete problem, having velocities as unknowns rather than accelerations.

The velocity-based method has the slight advantage of only having to compute the first differential for the constraint equation due to the velocity method not needing the Jacobian derivative to calculate the unknowns. Earlier solvers were formulated at the acceleration level, but had problems where friction

constraints could produce unstable systems. Later, the problem was modified so it could be solved at the velocity level where this problem was solved.

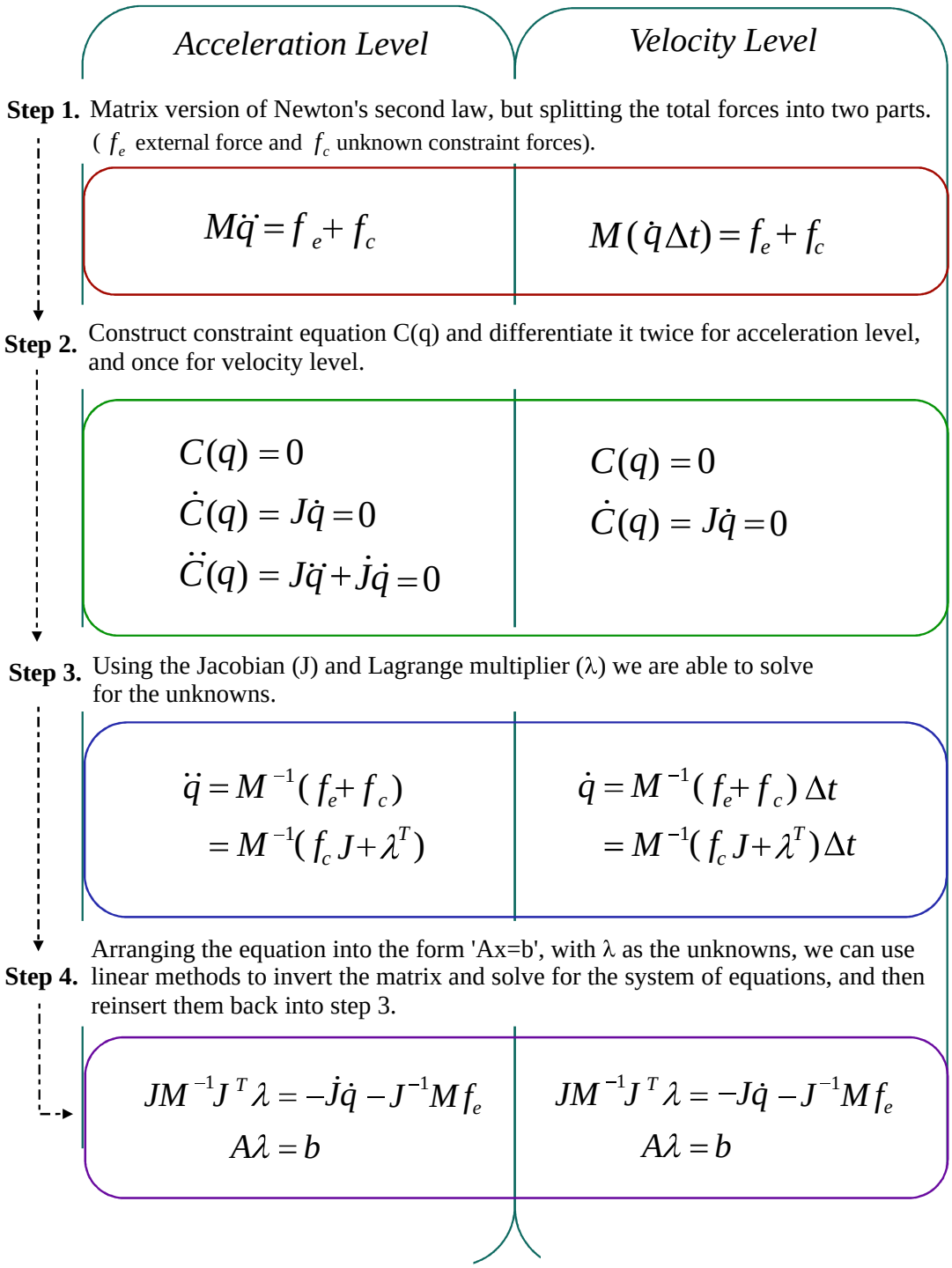

Figure 2. Basic four-step breakdown of acceleration and velocity based methods.

If principles of how to resolve constraints at the velocity level are understood, then the knowledge is there to implement it for the acceleration level (and vice versa). For the remainder of this chapter, we will focus on deriving and implementing methods for the velocity level, and where appropriate, mention any extraneous details that might be relevant to the acceleration level.

CONSTRAINTS

Overview

The formulation of the equations we use for our simulator can be broken down into four easy steps as shown in (Hecker, 1998) and reproduced in Figure 2. We introduce the steps briefly here, showing how the solver equations fit together, to attain a physics-based simulator. Even though they may not be completely understood initially, we reintroduce them again with further details and applied examples as we advance through the chapter.

Equality and Non-Equality Constraints

There are two types of constraints, *Equality* "==" and *Inequality* ">,>=,<,<=". Examples of *equality* constraints are rigid-ropes and ball-joints, while *inequality* constraints would be contacts and collisions.

Equality constraints have a fixed solution and *cannot* be varied along the constraint direction. They're formed by setting the constraint condition 'c' to zero, using the equality sign.

Inequality constraints can have numerous solutions and are formed by using a greater than or equal operator in constraint equations.

In the steps mentioned earlier for solving the unknown constraint values, we used an equal sign for $A\lambda = b$. For inequality conditions, we modify our linear solver to handle linear complementary problems. The modification adds the condition $'\geq'$, and only accepts values greater or equal to zero, essentially clamping λ so it is always positive:

Equality	*Inequality*
$A\lambda + b = 0$	$A\lambda + b \geq 0$ $\lambda \geq 0$ $(A\lambda + b)\, x = 0$
$c = 0$ $\dot{c} = 0$	$c \geq 0$ $\dot{c} \geq 0$

Due to inequality constraints having multiple results, we can check if we need them before we add them to our solver, i.e.:

```
if c ≤ 0 :
    add ċ ≥ 0 to constraint solver
else
    ignore
```

Jacobian

Forward kinematics enables us to define functions that convert between Cartesian space (i.e. positions and orientations) to constraint space, and the Jacobian represents how this Cartesian-constraint space relationship changes with respect to time, (e.g. Cartesian-constraint space *velocity* relationship).

The Jacobian gives the instantaneous transformation between the constraint velocities and the rigid body velocities. Where, for objects in 3D, each rigid body has six DOF, which represents the three linear and three angular velocities, for (m) rigid bodies we are able to form a (6m) column vector containing all the rigid body velocities, in addition to our (n) constraints, to form a (6m x n) Jacobian matrix, describing how all the objects in our system are *allowed* to move.

Jacobian is a matrix function of the form:

$$\dot{c} = J\,\dot{q} \tag{5}$$

where $\dot{q}$ (linear and angular) Cartesian space velocities and $\dot{c}$ the constraint velocities.

We can also express $\dot{q}$ as its separate angular and linear velocity components:

$$\dot{q} = [\,v,\ \omega\,] \tag{6}$$

for each body, $\dot{q}$ is a 6x1 Cartesian velocity vector, (3x1) linear vector and (3x1) rotational vector stacked together.

We can also write:

$$J = [\,J_v,\ J_\omega\,] \tag{7}$$

i.e.

$$\begin{aligned} v &= J_v\,\dot{q} \\ \omega &= J_\omega \dot{q} \end{aligned} \tag{8}$$

As we said earlier, a single unlinked object moving in 3D, can move in six possible ways, three linear and three angular. If you wanted to constrain the object from moving in a certain direction, the ‘z’ direction, for example, then you would set the ‘z’ velocity to zero, effectively limiting translation movement to the x-z plane. This is a simplified example of how a Jacobian works, and describes how constraints operate to remove DOF to achieve a desired motion trajectory. While the Jacobian is at the heart of our analytical method, it allows us to relate the classical equations of motion (f=ma) with Lagrange’s multiplier to solve systems of constraints. Later in the chapter we give numerous examples on how to describe and derive a mixture of common constraints and their Jacobian.

Because the Jacobian is central to our constraint formulation, we will spend a bit more time explaining its mechanics in detail with examples to give a rock-solid understanding. The Jacobian, sometimes called the *constraint Jacobian*, is a matrix which allows us to specify which motions are *not allowed*. The number of rows of the matrix determines the order of the constraint or the number of degrees of freedom removed from the system. For most constraints we calculate J on a frame by frame basis as it depends on the body’s positions and orientations.

$$J = \frac{Change\,Input}{Change\,Output} = \frac{\partial In}{\partial Out} \tag{9}$$

We can combine Jacobian matrices to formulate more complex ones, by which we mean, we can construct simple constraints and combine them to build more difficult ones.

To repeat the most important equation of this section, the *Jacobian velocity constraint* is shown again below in Equation (10), where $\dot{q}$ is the system body velocities, and $\dot{c}$ is the derivative of the positional constraints c with respect to time, and produces constraints by *removing* degrees of freedom from the system.

$$J\,\dot{q} = \dot{c} \tag{10}$$

To derive the Jacobian, we need to obtain $\dot{c}$, which we attain by constructing a positional constraint 'c' and differentiating it. The positional constraint is constructed by representing how the body is allowed to move with a kinematic equation such that when it is satisfied, it evaluates to zero:

$$c = 0 \tag{11}$$

If we differentiate our positional constraint equation (c) with respect to time, it will describe the constraint velocity properties. Also as 'c' is equal to zero, the derivative should be zero:

$$\dot{c} = 0 \tag{12}$$

Since we have said that c and $\dot{c}$ will be zero and have explained that the Jacobian has six elements for each body which constitute how the six velocity components change with respect to time, we can conclude that any non-zero values in J will affect the corresponding body velocities. Remember that these are relative to the body's point of reference (its position and rotation at that instance in time and can need re-calculating as the object moves).

Using this knowledge, the Jacobian can be calculated in three straightforward steps; firstly, by constructing an equation for the positional constraints, secondly by differentiating it with respect to time to get the velocity properties, and finally by isolating and extracting the Jacobian.

In some cases, you can compose the Jacobian constraint matrix by visually looking at the system, but for more complicated constraint schemes, you will need to follow the steps above, whereby you'll find that the ability to construct the Jacobian gets easier with practice. We give examples of simple Jacobian constraint matrices in the next section.

Note

When a constraint only affects a single object, and does not affect or rely upon any other objects, they are referred to as 'unary constraints', whilst a constraint involving another point-mass or rigid body is termed a 'binary constraint'. In addition, the Jacobian for binary constraints is usually the same but reversed.

Constraint Equations

We now focus on explaining how we resolve the constraints at the velocity level and the formulation of our system of equations that we use to represent our procedure. Even though we focus on the velocity level, we can apply the same principles and practice to the acceleration level without too much effort.

During the simulation update, we calculated a set of constraint forces, which we combined with the external forces, to keep the constraints valid. The main equation, which we construct and use to determine these cancelling constraint forces, is shown in Figure 3:

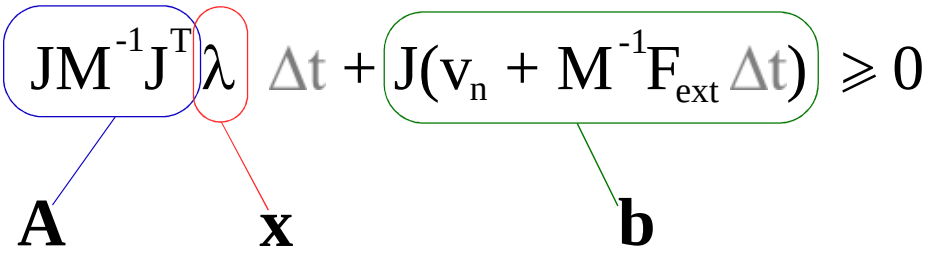

Figure 3. Lagrange multiplier modified to fit a system of linear equation to solve our constraints.

The equation can be initially overwhelming, but once the reader understands what it does and how it works, it is very satisfying and rewarding. The LCP equations are usually presented in the form:

$$\begin{aligned} b &= Ax + q \\ b, x &\geq 0, \quad b^T x = 0 \end{aligned} \tag{13}$$

This may seem straightforward, but notice the complementary conditions ('b' is 0 when 'x' is not and vice versa) and the non-negativity conditions ('b' and 'x' are >= 0), which is what makes the equations so useful and powerful.

But where does this equation come from, and what does it mean? We will start at the beginning and introduce the problem and how this solves it for us.

It might not be obvious, but if a snapshot of a rigid body simulator taken at any moment in time, the system can be represented, using a set of equations that illustrates how it behaves. If the objects are connected (e.g. through contacts or joints), then their individual EOM will be connected, and so we build up a single large equation (i.e. using matrices). This large equation follows a set of rules, which represent how the system of objects can move, allowing us to predict how the system will change with time.

Our formulation of a solver relies heavily on the principles of linear algebra. To restate, a basic system of linear equations in matrix form, would be:

$$Ax = b \tag{14}$$

where we know 'A' and 'b', and we are trying to find 'x'. 'A' is a symmetric, positive definite sparse matrix. The linear system can have three possible outcomes:

- A single solution which converges
- No solution
- Infinite solutions.

We use the linear equation as the starting foundation from which we build our solver, expanding the basic linear equation above so that it can be used for equality and inequality constraints, i.e.:

$$Ax \geq b \, and \, x \geq 0 \tag{15}$$

(From our ‘Ax = b’ linear equation above, if ‘Ax’ is not equal to ‘b’, then we have to reformulate it into something more complex, i.e. ‘Ax-b = s’, and if we make some assumptions, such as mentioned above, ‘Ax>=b’ and ‘x>0’, then we have introduced some constraints.)

The large equation used earlier actually comes from basic algebraic principles. Since we are focusing on the velocity level, we start with the simple velocity integration equation and convert it through substitution and common sense into the equation above.

At each time-step, we can use the Euler integration method to predict how the velocity will change, based on the applied acceleration and time-step. We use this updated velocity to predict how our position will change during our time-step, e.g.

$$\begin{aligned} \mathrm{v}_{n+1} &= \mathrm{v}_n + a\Delta t \\ x_{n+1} &= \mathrm{v}_{n+1} + a\Delta t \end{aligned} \tag{16}$$

According to Newton’s second law (i.e. f=ma), we are able to modify our velocity update to:

$$\mathrm{v}_{n+1} = \mathrm{v}_n + \mathrm{M}^{-1}\mathrm{F}\Delta\mathrm{t} \tag{17}$$

With the velocity integration scheme, we extrapolate it to solve for the applied acceleration changes, which are proportional to the applied forces.

It is important to comprehend how we go from basic velocity integration to the constraint equation. Whereas a majority of the texts skim over the subject, we will give a step-by-step justification in Figure 4.

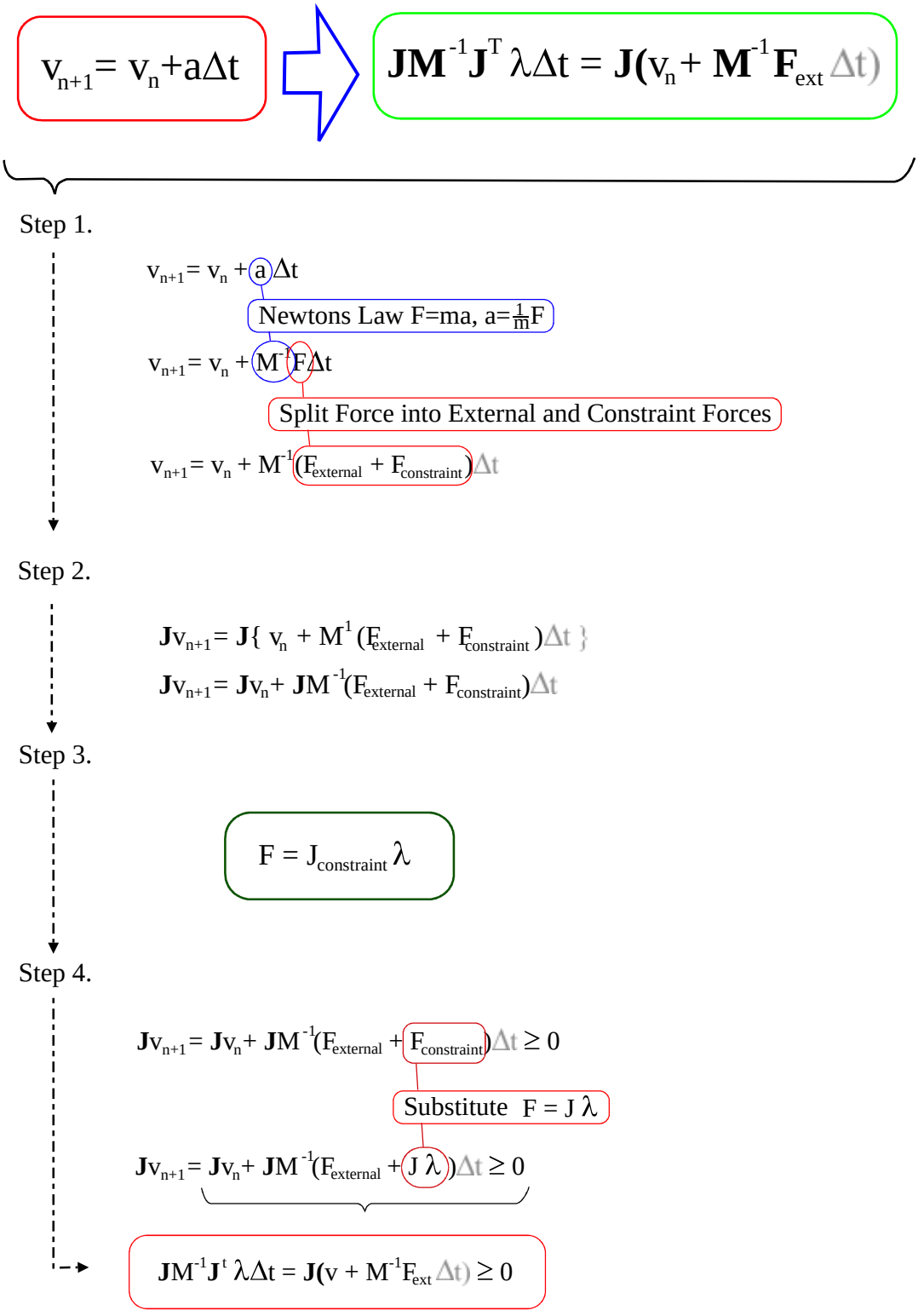

Figure 4. Four steps for deriving the constraint equation from the velocity integration step.

The two main steps in Figure 4, are *2* and *4*, which we describe further.

In Step 2, we integrate our Jacobian into the velocity equation by multiplying it across both sides. Recalling that the Jacobian defines how our objects will move, adding the Jacobian vector enables us to define the movement of our objects.

Step 4, is one of the biggest steps, as it brings together the equations for classical mechanics and the Lagrange multiplier through the use of the Jacobian. We arrive at our final equation by splitting the constraint force into its magnitude and direction. (This is a very noteworthy step, and should be remembered for later when we reuse it to reinsert our solved values).

$JM^{-1}J^T$ is referred to as the effect mass matrix, and we give it the symbol 'K'. For basic cases we can pre-compute the effective mass matrix and simplify it to show additional properties in the way the constraint behaves.

Note

When a solution for lambda (λ) is obtained, it can be used to give additional information about the system, e.g. the stress or strain between constraints.

Additionally, during implementation, the $JM^{-1}J^T$ matrix should have no zeros in the diagonal.

Solving the equations gives us lambda, which when multiplied by our Jacobian, gives us the constraint forces. We add the computed constraint forces to the applied external forces before integrating. This ensures that the constraints will constantly stay valid, even when large external forces are applied.

The rigid body matrix representations for our simulation are shown in Figure 5; where we collect together similar attributes, such as position and orientation, linear and angular velocities, into groups of matrices.

We use quaternions to represent our rigid body's orientation, and hence our incremental update to angular velocity, using:

$$\dot{q}_{n+1} = \frac{1}{2}\omega_n \dot{q}_n \tag{18}$$

So to achieve this same result with a matrix multiplication, we need to use a special matrix to represent our quaternion orientation. We refer to this matrix as the '*Q*' matrix, which is inside the 'S' matrix and is of the form:

$$Q = \frac{1}{2}\begin{bmatrix} -x & -y & -z \\ w & z & -y \\ -z & w & x \\ y & -x & w \end{bmatrix}$$

where 'w, x, y, z' are scalar and vector components of the quaternion. To phrase it another way, the Q matrix is essentially a sub-matrix representing the rotation using quaternions. (We highlight the 'Q' matrix above inside the 'S' matrix formulation).

6 D.O.F (3 Linear, 3 Angular) *3 D.O.F (3 Linear)*

$$\mathbf{M} = \begin{bmatrix} m & 0 & 0 & 0 & 0 & 0 \\ 0 & m & 0 & 0 & 0 & 0 \\ 0 & 0 & m & 0 & 0 & 0 \\ 0 & 0 & 0 & I_{xx} & I_{xy} & I_{xz} \\ 0 & 0 & 0 & I_{yx} & I_{yy} & I_{yz} \\ 0 & 0 & 0 & I_{zx} & I_{zy} & I_{zz} \end{bmatrix} \quad \mathbf{M} = \begin{bmatrix} m & 0 & 0 \\ 0 & m & 0 \\ 0 & 0 & m \end{bmatrix}$$

Rotation Inertia Matrix

Mass Matrix (Angular and linear components).

6 D.O.F (3 Linear, 3 Angular) *3 D.O.F (3 Linear)*

$$\mathbf{F} = \begin{bmatrix} F_x & 0 & 0 & 0 & 0 & 0 \\ 0 & F_y & 0 & 0 & 0 & 0 \\ 0 & 0 & F_z & 0 & 0 & 0 \\ 0 & 0 & 0 & \tau_x & 0 & 0 \\ 0 & 0 & 0 & 0 & \tau_y & 0 \\ 0 & 0 & 0 & 0 & 0 & \tau_z \end{bmatrix} \quad \mathbf{F} = \begin{bmatrix} F_x & 0 & 0 \\ 0 & F_y & 0 \\ 0 & 0 & F_z \end{bmatrix}$$

Force Matrix (Linear Force and Angular Force, aka Torque).

6 D.O.F (3 Linear, 3 Angular) *3 D.O.F (3 Linear)*

$$\mathbf{V} = \begin{bmatrix} v_x & 0 & 0 & 0 & 0 & 0 \\ 0 & v_y & 0 & 0 & 0 & 0 \\ 0 & 0 & v_z & 0 & 0 & 0 \\ 0 & 0 & 0 & \omega_x & 0 & 0 \\ 0 & 0 & 0 & 0 & \omega_y & 0 \\ 0 & 0 & 0 & 0 & 0 & \omega_z \end{bmatrix} \quad \mathbf{V} = \begin{bmatrix} v_x & 0 & 0 \\ 0 & v_y & 0 \\ 0 & 0 & v_z \end{bmatrix}$$

Velocity Matrix (Linear and Angular Velocitys)

$$\mathbf{S} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & q_x & q_y & q_z \\ 0 & 0 & 0 & q_w & q_z & q_y \\ 0 & 0 & 0 & q_z & q_w & q_x \\ 0 & 0 & 0 & q_y & q_x & q_w \end{bmatrix}$$

Q Matrix

$$q_{n+1} = \frac{1}{2}\omega q_n$$

S Matrix (combine quaternion matrix and linear components).

6 D.O.F (3 Linear, 3 Angular) *3 D.O.F (3 Linear)*

$$\mathbf{X} = \begin{bmatrix} p_x & 0 & 0 & 0 & 0 & 0 \\ 0 & p_y & 0 & 0 & 0 & 0 \\ 0 & 0 & p_z & 0 & 0 & 0 \\ 0 & 0 & 0 & q_x & 0 & 0 \\ 0 & 0 & 0 & 0 & q_y & 0 \\ 0 & 0 & 0 & 0 & 0 & q_z \end{bmatrix} \quad \mathbf{X} = \begin{bmatrix} p_x & 0 & 0 \\ 0 & p_y & 0 \\ 0 & 0 & p_z \end{bmatrix}$$

$$X_{n+1} = X + \Delta t S V$$

Exclude S if no Rotation

Position Matrix (Angular component is a quaternion).

Figure 5. Matrix Configuration and Contents (Linear and Angular Components).

Traditionally, using the Euler integration method, we would have integrated each component (linear and angular) for each rigid body using Equations (19) and (20).

$$\begin{aligned} v_{n+1} &= v_n + \frac{1}{m} F_{ext} \Delta t \\ x_{n+1} &= x_n + v_{n+1} \Delta t \end{aligned} \tag{19}$$

$$\begin{aligned} \omega_{n+1} &= \omega_n + \omega_n I_{InvWorld} \ \Delta t \\ q_{n+1} &= q_n + \frac{1}{2} \omega_{n+1} q_n \ \Delta t \end{aligned} \tag{20}$$

We move our rigid body into matrices and perform the integration update using Equations (21) and (22) below.

$$u_{n+1} = u_n + \Delta t \, M^{-1} F_{ext} \tag{21}$$

$$s_{n+1} = s_n + \Delta t \, S \, u_{n+1} \tag{22}$$

This is the basic integration without any intervention from our constraint forces, which we add into the modified block matrix integration:

$$u_{n+1} = u_n + \Delta t \, M^{-1} F_{ext} - (\Delta t \, M^{-1} \, J^T \, x) \tag{23}$$

$$s_{n+1} = s_n + \Delta t \, S \, u_{n+1} \tag{24}$$

where $(\Delta t M^{-1} J^T x)$ uses the cancelling force magnitude in 'x' to feedback into the integrated update and keep the constraints valid. The following code snippets show the method implemented in code.

```
    // Basic Integration without constraints or collisions
    u_next = u + dt*MInverse*Fext;
    s_next = s + dt*S*u_next;
```

Code 1. Basic Euler integration step using matrices.

```
     // Basic Integration without constraints or collisions
     u_next = u + dt*MInverse*Fext - MInverse*Jt*x;
     s_next = s + dt*S*u_next;
```

Code 2. Modified integration step with our added constraint forces.

Block matrix methods make it possible to group together system state variables, such as forces, velocities and positions, into intuitive manageable pieces and reduces the amount of code disarray. In the sample above, we use a matrix to represent the position and rotation of each body; alternatively, you can exclude this fragment and merely solve for the constraint forces, then apply them as you would to the simulator. The reasons for this may be because the constraint solver is part of a larger system where objects are

unable to be merged into matrix formation, or because it is easier to manage data some other way, due to optimisation.

Whilst we use methods to combine angular and linear components into single larger matrices, it can sometimes be more efficient to separate them into individual matrices so as to disable and enable rotational components for debugging, or alternatively develop a point mass simulator which contains no rotational components. For example, the mass matrix is a combination of the linear and angular mass components (centre of mass and inertia tensors), which could be broken up into two separate matrix functions.

Initially, matrix methods in combination with analytical solutions can seem like more work compared to the penalty and impulse methods, but the results give us a reliable, flexible simulator, which requires little or no user tweaking.

Our solver uses classical mechanics in combination with Lagrangian multiplier techniques, so we are able to formulate systems of equations to calculate the necessary forces to apply at each frame and enable our constraints to remain legal. For example, a ball resting on the floor has a downward force applied to it, known as gravity. We cannot allow this downward force to update the velocity and let the ball move translate downwards. Hence, we add a contact constraint to the system to prevent this penetration violation. This constraint would produce a cancelling upward force, keeping the ball resting on the ground. The complementary part of the solution is when the ball is falling with velocity, and we need to add an impact force, causing the ball to bounce upwards and not stick to the ground (it can move up but not down).

Algorithm Steps

1. Apply forces and torques (i.e. from springs, gravity, etc.).
2. Build matrices for rigid body masses/velocities/positions.
3. Build Jacobian constraint matrices representing various constraints.
4. Evaluate your system by solving for lambda.
5. Build constraint forces using lambda and Jacobian matrices.
6. Calculate and apply constraint forces.
7. Integrate and update as usual.

```
void Simulation_c::Update(float dt)
{
    // Add gravity, forces from springs etc.
    ComputeForces(dt);

    // Calculate constraint forces and apply them to our system
    ComputeJointConstraints(dt);

    // Step forwards in time - integration done inside ComputeJointConstraints(..)
    // using block matrices, but you can disable it and integrate using
    // an alternative integrator
    //Integrate(dt);
}
```

Code 3. Steps for updating code.

Note

Making objects immovable by setting their mass to zero or making their mass near infinite (e.g. 10000000) can crash the simulator. This is because the solver needs to invert the matrix, so large or zero

values can result in a non-invertible matrix. Large values may not crash the simulation, but it can require a very large number of iterations to find its inverse solution.

'1D' Numerical Example

The best way to get familiar with solvers is practice, which we now do using simplified straightforward test cases that we analyse with common sense and understanding. Common mistakes to keep an eye out for initially, are errors due to wrong number sign (+ instead of a -) or invalid divide by zero troubles.

We start with simple 1D examples, where we do not need to worry about rotation, just the translation, and can focus on the force magnitudes and directions. Whereby, our matrix formulations will be greatly simplified (e.g. the mass matrix in 1D is cut down to a single variable for each object), so we can evaluate the equations by hand. It is also recommended that this simplified approach is used to work through additional problems which will demonstrate first-rate principles of how constraint solvers achieve reliable simulations, such as jitter free stacking.

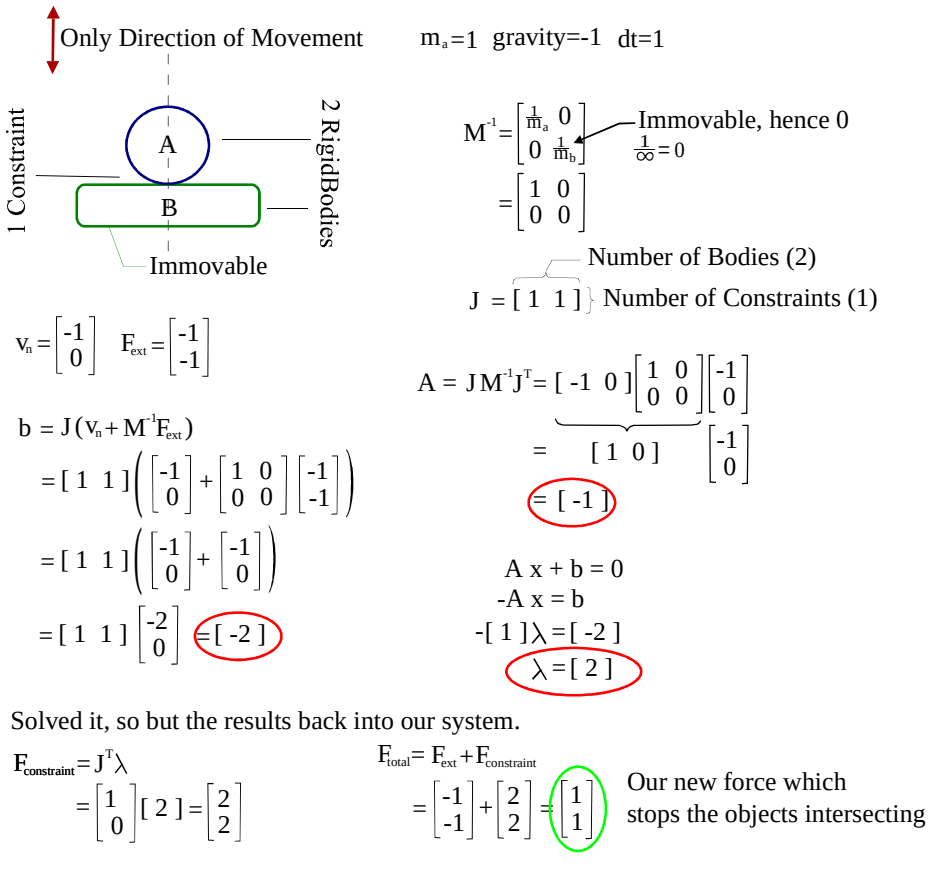

Figure 6. Numerical Example

The example shown in Figure 6 is constructed by taking a single object and letting it rest on an immovable surface. Then when we apply downward force due to gravity, the top object will continue to stay fixed. Since it is only a straightforward example, we use a very simple uncomplicated Jacobian for

our constraint, which prevents any movement in the single-axis direction. Of course, to make it a little more interesting, the top object also has an internal velocity, so the calculated resultant force, when integrated into the velocity, will cancel it out. Hence, our constraint solver keeps our resting object on the surface as the simple constraint demands.

If the question; why do we get [1,1] for the resulting force and not [0,0], is asked, this is as we mentioned earlier. Our object has an initial velocity of [-1,0], so a positive force is necessary to cancel out this downward velocity and keep our constraint valid. If we have had zero velocities, the downward force would have cancelled out, leaving us with [0,0].

It should be well-known by now that the solver needs to run each frame before the integration step, which updates the velocities and positions. This pre-step evaluation calculates the correcting constraint forces which, when applied to the input forces, (e.g. from gravity, wind, springs) keeps the constraints legal.

1D Example – Multiple Bodies

Continuing to keep the examples as simple as possible, we can build on the previous example to illustrate stacking. The configuration shown in Figure 7 consists of three objects stacked upon one another; each having equal mass.

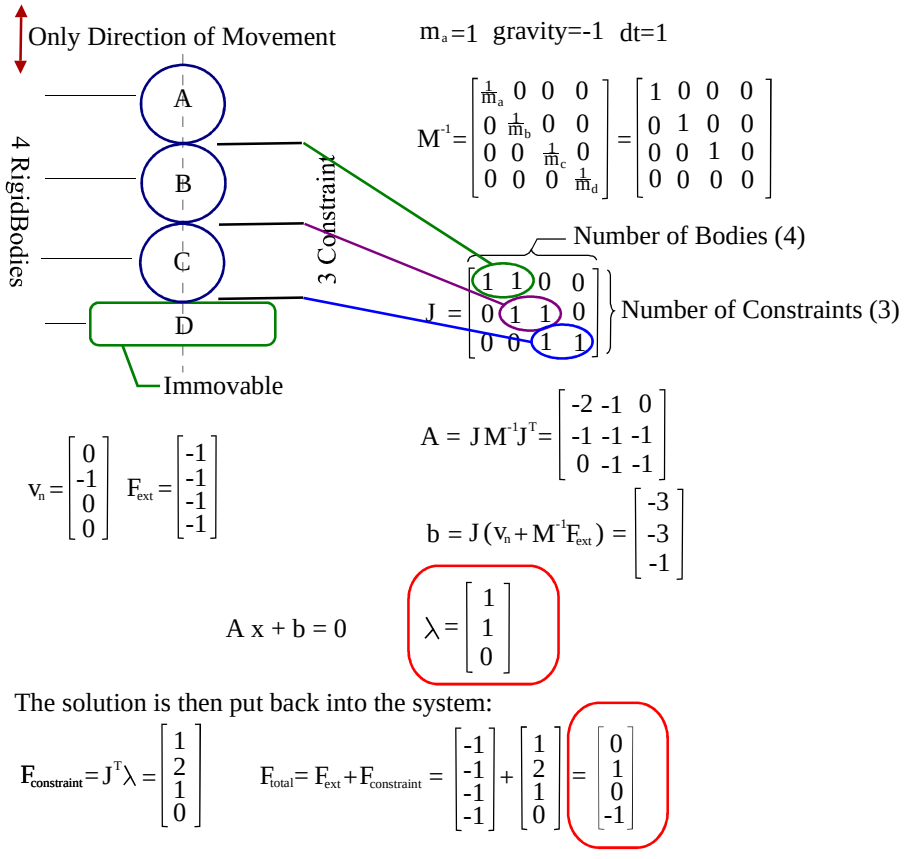

Figure 7. Numerical Example.

We set all the objects to have zero velocities initially, except the second to top one, which we set to having an internal downward velocity. As above we use a simplified constraint formulation to calculate

the constraint force to prevent movement of the objects. The solved example calculates the modified force to stop the objects moving downwards, including an additional force to cancel the second to top body's velocity.

Drifting (Baumgarte Stabilization)

The constraint calculation keeps the system of constraints valid by cancelling out any violating forces/velocities. However, due to numerical inaccuracies in calculations, or bad starting values, the constraints can become invalid, and so we need to modify our calculation slightly to correct this.

To solve for this, we modify our solver equation by adding a feedback term, which gradually corrects drifting errors. We achieve this correction by using the position constraint 'c', to feed back into the velocity constraint, causing the simulation constraints to remain valid at all times.

Modifying our original velocity constraint from Equation (10) to give feedback, we get:

$$\dot{c} = J\dot{q} + \beta c = 0 \tag{25}$$

where β is the bias factor $(0 < \beta < 1)$.

This corrective term is effectively the same as adding a spring, due to its remedial nature being proportional to the positional error.

```
float beta = 0.1f;                        // Drifting correction factor

DMatrix Jt = Transpose(J);                // Formulation of the solver equations
DMatrix A  = J*MInverse*Jt;

DMatrix b  = J*(u + dt*MInverse*Fext) +  //
             beta*c;                      // Drifting (Baumgarte) appended on end

DMatrix x(A.GetNumRows(), b.GetNumCols());

// Solve for x
LCPSolver(A, b, &x);                      // Solver, equality constraint case
```

Code 4. Drifting correction – achieved by appending a drift correction term to the base solver equation.

Note

If the simulation begins with invalid constraint configurations, the Baumgarte feedback term will cause the system to try and correct itself. Additionally, we can use this effect to modify our contact constraints, so that if they penetrate, the constraint will apply a correcting penalty force to push the shapes out of penetration.

Constraint Examples

Formulating the constraint condition 'C', we can calculate how to differentiate and extrapolate the Jacobian to determine how our system moves and to restrict its movement to create our constraint.

Example: Fixed with no translational or rotational movement

Formulate the positional constraint:

$$c = \left[q_x - p_x,\ q_y - p_y,\ q_z - p_z,\ q_{\theta x} - p_{\theta x},\ q_{\theta y} - p_{\theta y},\ q_{\theta z} - p_{\theta z} \right] = 0 \tag{26}$$

where $q_x, q_y, q_z,\ q_{\theta x}, q_{\theta y}, q_{\theta z}$ are the position and orientation of the body, and $p_x, p_y, p_z,\ p_{\theta x}, p_{\theta y}, p_{\theta z}$ are the global translation and angular *constants*. Noting that '0' represents a zero vector [0,0,0,0,0,0], if we then differentiate this constraint with respect to time, we obtain the velocity constraint:

$$\dot{c} = \left[\dot{q}_x, \dot{q}_y, \dot{q}_z, \dot{q}_{\theta x}, \dot{q}_{\theta y}, \dot{q}_{\theta z} \right] = 0 \tag{27}$$

Since our constraint position and orientation are constant, they simply cancel out during differentiating, hence enabling us to compare and isolate the Jacobian:

$$J\dot{q} = \dot{c} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \dot{q} = 0 \tag{28}$$

We make the constraint from six rows and takes away six degrees of freedom, hence the rigid body cannot move in any direction or orientation.

Example: No movement along the X-axis

From the previous example, we can then build upon the fact that if we use only a single row, we can eliminate a single degree of freedom.

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \dot{q} = 0 \tag{29}$$

Through visual analysis, we can make out that the velocity in the x-axis is cancelled out by having a single 1 in the first column, hence no velocity in this direction results in no movement. Since the other velocity components are not affected, we can deduce that they will be unaffected. This constraint formulation only affects the changing velocity to prevent movement in a single direction, and not the actual position at each moment.

Example: No rotation around up vector (y-direction)

Whereas we removed translation in the x-axis previously, we extend this further by removing a single axis of rotation. It should be visually clear that the first three columns are the positional constraints, and the final three columns make up the rotational movement.

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \dot{q} = 0 \tag{30}$$

As shown above, adding a one to the fifth column prevents any angular velocity in the y-axis and hence any rotational movement similarly.

Example: Ball and socket constraint

When dealing with pairs of bodies, we need to use two Jacobian matrices to represent the constraint condition. Below, we show the formulation of a ball-joint, by constructing its positional constraint then differentiating it, so we can extract the Jacobian for each object. Notice that the rows are a combination of translational and rotational information; making the constrained movement more complex.

$$c = \left[(p_1 + r_1) - (p_0 + r_0) \right] = 0 \tag{31}$$

where p represents the object centre, and r the relative offset from it.

$$\dot{c} = \left[(v_1 + \omega_1 \times r_1) - (v_0 + \omega_0 \times r_0) \right] = 0 \tag{32}$$

$$J_o = [I, r_1^*], \quad J_1 = -[I, r_0^*] \tag{33}$$

$$\begin{bmatrix} 1 & 0 & 0 & 0 & r_{0z} & -r_{0y} \\ 0 & 1 & 0 & -r_{0z} & 0 & r_{0x} \\ 0 & 0 & 1 & r_{0y} & -r_{0x} & 0 \end{bmatrix} \dot{q}_0 - \begin{bmatrix} 1 & 0 & 0 & 0 & r_{1z} & -r_{1y} \\ 0 & 1 & 0 & -r_{1z} & 0 & r_{1x} \\ 0 & 0 & 1 & r_{1y} & -r_{1x} & 0 \end{bmatrix} \dot{q}_1 = 0 \tag{34}$$

where r_1^*, r_0^* represents a 3x3 skew matrix which is equivalent to the cross product.

Example :Fixed Point (Nail)

If we took our object and just stuck a nail through any point, it would stay locked at that point while still able to rotate. We effectively use the same method as above (ball and socket), except, we only have a single body. The Jacobian formulation is shown below, and shows how the object cannot move in the ‘x’, ‘y’ or ‘z’ direction but can still rotate relative to a specific point.

$$\begin{bmatrix} 1 & 0 & 0 & 0 & r_z & -r_y \\ 0 & 1 & 0 & -r_z & 0 & r_x \\ 0 & 0 & 1 & r_y & -r_x & 0 \end{bmatrix} \dot{q} = 0 \tag{35}$$

where r_x, r_y, r_z represent the offset from the object’s centre to the point we are rotating around.

Example: Distance constraint (No Rotation)

A simple and useful constraint, especially when working with point-masses is a distance constraint whereby the distance between any two points remains fixed.

Setting up the conditions under which a constraint would be valid, we can say that the distance must always be equal to some length (l):

$$\| p0 - p1 \| = l \tag{36}$$

where ||p0-p1|| is the length between the two points, ‘p0’ and ‘p1’.

Constructing the position constraint ‘c’, we have:

$$c = \| p0 - p1 \| - l = 0 \tag{37}$$

Knowing 'c', we can differentiate it to get $\dot{c}$, and using the definition for the Jacobian velocity constraint, extrapolate the Jacobian component:

$$J\dot{q} = \dot{c} \quad and \quad \dot{q} = \begin{bmatrix} \mathrm{v0} \\ \mathrm{v1} \end{bmatrix} \tag{38}$$

$$J = \begin{bmatrix} -\hat{n}^T & \hat{n}^T \end{bmatrix} \tag{39}$$

e.g.

$$\begin{bmatrix} \hat{n}_x & \hat{n}_y & \hat{n}_z \end{bmatrix} \dot{q} = 0 \tag{40}$$

where 'n' is the unit vector between the two points:

$$\hat{n} = \frac{(p1 - p0)}{\| p1 - p0 \|} \tag{41}$$

Example: Contact Constraint

We formulate a contact constraint which uses the contact normal 'n' to prevent objects intersecting. The added difference between this constraint and the previous constraint examples is the added boundary condition. Whereas we used the equality condition for previous examples, we now apply an inequality condition for the solution λ to be greater than or equal to zero, warranting objects to move away from each other. For the calculations below we take for granted that the normal vector is pointing from body0 to body1.

$$c = \left[(p_1 + r_1) - (p_0 + r_0) \right] \cdot \hat{n}_1 \geq 0 \tag{42}$$

$$\dot{c} = \left[(v_1 + \omega_1 \times r_1) - (v_0 + \omega_0 \times r_0) \right] \cdot \hat{n}_1 + \left[(p_1 + r_1) - (p_0 + r_0) \right] \cdot (\omega \times \hat{n}_1) \geq 0 \tag{43}$$

$$J_o = [\hat{n}_1, r_0 \times \hat{n}_1], \quad J_1 = -[\hat{n}_1, r_1 \times \hat{n}_1] \tag{44}$$

$$\begin{bmatrix} n_{1x} & 0 & 0 & 0 & a_{0z} & -a_{0y} \\ 0 & n_{1y} & 0 & -a_{0z} & 0 & a_{0x} \\ 0 & 0 & n_{1z} & a_{0y} & -a_{0x} & 0 \end{bmatrix} \dot{q}_0 - \begin{bmatrix} n_{1x} & 0 & 0 & 0 & a_{1z} & -a_{1y} \\ 0 & n_{1y} & 0 & -a_{1z} & 0 & a_{1x} \\ 0 & 0 & n_{1z} & a_{1y} & -a_{1x} & 0 \end{bmatrix} \dot{q}_1 = 0 \tag{45}$$

where 'r' is the relative offset from the object centre to the contact point and $a_0 = r_0 \times \hat{n}_1$, $a_1 = r_1 \times \hat{n}_1$.

When implementing this constraint, it is crucial to remember $\lambda \geq 0$ for the boundary conditions. We can additionally add friction by modifying our constraint to add a cancelling force along the tangential direction of movement.

IMPLEMENTATION

Writing the code

Below is the code for the rigid body class, which encapsulates the properties of each object. We separate out the angular and linear components, so that we can easily modify it to work with full 3D or point-mass simulations.

```
class RigidBody_c // DirectX Implementation
{
    //<---------LINEAR---------------><-------------ANGULAR-------------->
    D3DXVECTOR3 m_position;             D3DXQUATERNION  m_orientation;
    D3DXVECTOR3 m_linearVelocity;       D3DXVECTOR3     m_angularVelocity;
    D3DXVECTOR3 m_force;                D3DXVECTOR3     m_torque;
    float       m_invMass;              D3DXMATRIX      m_invInertia;


    void AddForce(const D3DXVECTOR3& worldPosForce,
                  const D3DXVECTOR3& directionMagnitude)
    {
        DBG_VALID(worldPosForce);
        DBG_VALID(directionMagnitude);

        //<---------LINEAR----------------->
        m_force         += directionMagnitude;

        //<-----------ANGULAR-------------->
        D3DXVECTOR3 distance = worldPosForce - m_position;
        D3DXVECTOR3 torque   = Cross(distance, directionMagnitude);
        AddTorque(torque);
        DBG_VALID(m_force);
        DBG_VALID(m_torque);
    }

    void AddTorque(D3DXVECTOR3 worldAxisAndMagnitudeTorque)
    {
        DBG_VALID(worldAxisAndMagnitudeTorque);
        m_torque += worldAxisAndMagnitudeTorque;
    }

    D3DXMATRIX CreateWorldII()
    {
        D3DXMATRIX orientationMatrix = CreateMatrixFromQuaternion( m_orientation );
        D3DXMATRIX inverseOrientationMatrix  = Transpose( orientationMatrix );
        D3DXMATRIX inverseWorldInertiaMatrix = inverseOrientationMatrix * m_invInertia *
                                                                          orientationMatrix;
        return inverseWorldInertiaMatrix;
    }
};
```

Code 5. Basic Euler Integrator.

Note

Disable Rotation – If the rigid body simulator contains bugs, it is best to go back to basics, if angular problems are suspected, remove the rotation, then various degrees of freedom and add them back when it is known that they are working.

Iterative LCP Solver

From the code below, you can see that the LCP solver consists of nested loops, within which we modify our starting approximation iteratively and converge gradually towards a solution. Where the sample code uses a preset maximum number of iterations (maxIterations) for straightforwardness, we can alternatively, add an early breakout condition to accelerate the simulation, which we introduce later.

```
void GaussSeidelLCP(DMatrix& a, DMatrix& b, DMatrix* x, const DMatrix* lo, const DMatrix* hi)
{
    int maxIterations = 10; // Test Max value

    x->SetToZero();  // Clear our matrix to start with (slow, only for debug)
    const int n = x->GetNumRows();

    float sum = 0.0f;
    while(maxIterations--)
    {
        for(int i = 0; i < n; i++)
        {
            sum = b.Get(i);
            for(int j = 0; j < n; j++)
            {
                if( i != j )
                {
                          sum = sum - (a.Get(i,j) * x->Get(j));
                }
            }
            // If a.Get(i,i) is zero – you have a bad matrix!
            DBG_ASSERT(a.Get(i,i)!=0.0f);
            x->Set(i) = sum/a.Get(i,i);

          // Only do condition to check if we have them
        }

       // If we have boundary conditions, e.g. >= or <=, then we modify our basic Ax=b,
       // solver to apply constraint conditions
       // Optional - only if bounds
       if ( lo || hi )
       for (int i=0; i<n; i++)
       {
         if (lo)
         {
            DBG_ASSERT(lo->GetNumCols()==1); // Sanity Checks
            DBG_ASSERT(lo->GetNumRows()==n);
            if (x->Get(i) < lo->Get(i)) x->Set(i) = lo->Get(i);
         }

         if (hi)
         {
            DBG_ASSERT(hi->GetNumCols()==1); // Sanity Checks
            DBG_ASSERT(hi->GetNumRows()==n);
            if (x->Get(i) > hi->Get(i)) x->Set(i) = hi->Get(i);
         }
       }
    }
    // We've solved x!
}
```

Code 6. Iterative Solver – showing how simple it can be to invert a matrix.

Note

Fewer Iterations – If you notice in the example code for the 'Gauss-Seidel' iterative solver, it clears the solution matrix to zero at the start. This makes the assumption that our initial guess is zero. A faster method is not to clear the matrix, instead just using the values that are still in it from the previous frame. This will give us a good starting guess, enabling us to find a correct result with fewer iterations due to very small changes happening between frames.

Constraints that 'Snap-Together'

One important aspect when writing a solver is to write the code in such a way that it can easily be adapted and the code expanded to handle new constraints and configurations. We do this by constructing a

common virtual base class, which all constraints inherit from and implement, so that we are able to 'plug-in' any constraint into our simulator as and when we need it.

```
class Constraint_c
{
public:
    virtual ~Constraint_c(){}

    virtual DMatrix   GetPenalty()                       {   DBG_HALT;   return DMatrix(0,0);};

    virtual DMatrix   GetJacobian(const RigidBody_c* rb) {   DBG_HALT;   return DMatrix(0,0);};

    virtual int       GetDimension() const               {   DBG_HALT;   return 0;           };

};
```

If equality and inequality constraints are to be implemented, a base method should be added to differentiate them (i.e. as shown below). These are so the inequality conditions can incorporate additional boundary checks when solving the system of equations.

```
    virtual bool  IsEquality () const                    {   return false;                   };
```

'Heart' of the Simulator

Below we show the function '`ComputeJointConstraints`', which is responsible for asking every constraint about its dimensions, then constructing large sparse matrices, which correspond to the system configuration and calculate the corresponding constraint forces:

```
void Simulation:: ComputeJointConstraints ()
{
    // Magic Formula
    //
    // J * M^-1 * J^t  * lamba = -1.0 * J * (1/dt*V + M^-1 * Fext)
    //
    // A x = b
    //
    // where
    //
    // A = J * M^-1 * J^t
    // x = lambda
    // b = -J * (1/dt*V + M^-1 * Fext)
    //

    const int numBodies         = m_rigidBodies.Size();
    const int numConstraints    = m_constraints.Size();

    if (numBodies==0 || numConstraints==0) return;

    //-----------------------------------------------------------------------
    // 1st - build our matrices - very bad to build them each frame, but
    //-----------------------------------------------------------------------
    // simpler to explain and implement this way
    DMatrix s(numBodies*7, 1);          // pos & qrot
    DMatrix u(numBodies*6, 1);          // vel & rotvel
    DMatrix s_next(numBodies*7, 1);     // pos & qrot after timestep
    DMatrix u_next(numBodies*6, 1);     // vel & rotvel after timestep
    DMatrix S(numBodies*7, numBodies*6);
    DMatrix MInverse(numBodies*6, numBodies*6);
    DMatrix Fext(numBodies*6, 1);

    for (int i=0; i<numBodies; i++)
    {
```

```
        const RigidBody_c* rb = m_rigidBodies[i];

        s.Set(i*7+0) = rb->m_position.x;
        s.Set(i*7+1) = rb->m_position.y;
        s.Set(i*7+2) = rb->m_position.z;
        s.Set(i*7+3) = rb->m_orientation.w;
        s.Set(i*7+4) = rb->m_orientation.x;
        s.Set(i*7+5) = rb->m_orientation.y;
        s.Set(i*7+6) = rb->m_orientation.z;

        u.Set(i*6+0) = rb->m_linearVelocity.x;
        u.Set(i*6+1) = rb->m_linearVelocity.y;
        u.Set(i*6+2) = rb->m_linearVelocity.z;
        u.Set(i*6+3) = rb->m_angularVelocity.x;
        u.Set(i*6+4) = rb->m_angularVelocity.y;
        u.Set(i*6+5) = rb->m_angularVelocity.z;

        const D3DXQUATERNION& q = rb->m_orientation;
        DMatrix Q(4,3);
        Q.Set(0,0)=-q.x;        Q.Set(0,1)=-q.y;        Q.Set(0,2)=-q.z;
        Q.Set(1,0)= q.w;        Q.Set(1,1)= q.z;        Q.Set(1,2)=-q.y;
        Q.Set(2,0)=-q.z;        Q.Set(2,1)= q.w;        Q.Set(2,2)= q.x;
        Q.Set(3,0)= q.y;        Q.Set(3,1)=-q.x;        Q.Set(3,2)= q.w;
        Q = 0.5f * Q;
        DMatrix Idenity(3,3);
        Idenity.SetToZero();
        Idenity.Set(0,0) = Idenity.Set(1,1) = Idenity.Set(2,2) = 1.0f;
        S.SetSubMatrix(i*7+0,   i*6+0,   Idenity);
        S.SetSubMatrix(i*7+3,   i*6+3,   Q);

        DMatrix M(3,3);
        M.Set(0,0) = M.Set(1,1) = M.Set(2,2) = rb->m_invMass;

        const D3DXMATRIX& dxm = rb->CreateWorldII();

        DMatrix I(3,3);
        I.Set(0,0)=dxm._11;  I.Set(1,0)=dxm._12;  I.Set(2,0)=dxm._13;
        I.Set(0,1)=dxm._21;  I.Set(1,1)=dxm._22;  I.Set(2,1)=dxm._23;
        I.Set(0,2)=dxm._31;  I.Set(1,2)=dxm._32;  I.Set(2,2)=dxm._33;
        MInverse.SetSubMatrix(i*6,   i*6,   M);
        MInverse.SetSubMatrix(i*6+3, i*6+3, I);

        DMatrix F(3,1);
        F.Set(0,0) = rb->m_force.x;
        F.Set(1,0) = rb->m_force.y;
        F.Set(2,0) = rb->m_force.z;
        D3DXVECTOR3 rF = rb->m_torque;
        DMatrix T(3,1);
        T.Set(0,0) = rF.x;
        T.Set(1,0) = rF.y;
        T.Set(2,0) = rF.z;
        Fext.SetSubMatrix(i*6,   0,  F);
        Fext.SetSubMatrix(i*6+3, 0,  T);
    }

    //------------------------------------------------------------------------
    // 2nd - apply constraints
    //------------------------------------------------------------------------
    // Determine the size of our jacobian matrix
    int numRows = 0;
    for (int i=0; i<numConstraints; i++)
    {
        const Constraint_c* constraint = m_constraints[i];
        DBG_ASSERT(constraint);
        numRows += constraint->GetDimension();
    }

    // Allocate it, and fill it
    DMatrix J(numRows, 6*numBodies);
    DMatrix e(numRows, 1);              // Error Correction
```

```
    int constraintRow = 0;
    for (int c=0; c<numConstraints; c++)
    {
        Constraint_c* constraint = m_constraints[c];
        DBG_ASSERT(constraint);

        for (int r=0; r<numBodies; r++)
        {
            const RigidBody_c* rigidBody = m_rigidBodies[r];
            DBG_ASSERT(rigidBody);

            DMatrix JMat = constraint->GetJacobian(rigidBody);
            if (JMat.GetNumCols()==0 && JMat.GetNumRows()==0)
                continue;
            DBG_ASSERT(JMat.GetNumCols()!=0);
            DBG_ASSERT(JMat.GetNumRows()!=0);

            J.SetSubMatrix(constraintRow, r*6, JMat);

            DMatrix errMat = constraint->GetPenalty();
            e.AddSubMatrix(constraintRow, 0, errMat);
        }
        constraintRow += constraint->GetDimension();
    }

    float beta = 0.1f; // Error correction term

    DMatrix Jt = Transpose(J);
    DMatrix A  = J*MInverse*Jt;
    DMatrix b  = J*(u + dt*MInverse*Fext) + beta*e;
    DMatrix x(A.GetNumRows(), b.GetNumCols());

    DMatrix* lo = NULL; // Don't set any min/max boundaries for this demo/sample
    DMatrix* hi = NULL;

    // Solve for x
    LCPSolver(A, b, &x, lo, hi);

    //dprintf( A.Print() );

    u_next = u - MInverse*Jt*x + dt*MInverse*Fext;
    s_next = s + dt*S*u_next;

    // Basic integration without - euler integration standalone
    // u_next = u + dt*MInverse*Fext;
    // s_next = s + dt*S*u_next;

    //-----------------------------------------------------------------------
    // 3rd - re-inject solved values back into the simulator
    //-----------------------------------------------------------------------
    for (int i=0; i<numBodies; i++)
    {
        RigidBody_c* rb = m_rigidBodies[i];
        rb->m_position.x            = s_next.Get(i*7+0);
        rb->m_position.y            = s_next.Get(i*7+1);
        rb->m_position.z            = s_next.Get(i*7+2);
        rb->m_orientation.w         = s_next.Get(i*7+3);
        rb->m_orientation.x         = s_next.Get(i*7+4);
        rb->m_orientation.y         = s_next.Get(i*7+5);
        rb->m_orientation.z         = s_next.Get(i*7+6);

        rb->m_linearVelocity.x      = u_next.Get(i*6+0);
        rb->m_linearVelocity.y      = u_next.Get(i*6+1);
        rb->m_linearVelocity.z      = u_next.Get(i*6+2);
        rb->m_angularVelocity.x     = u_next.Get(i*6+3);
        rb->m_angularVelocity.y     = u_next.Get(i*6+4);
        rb->m_angularVelocity.z     = u_next.Get(i*6+5);

        rb->m_force                 = D3DXVECTOR3(0,0,0);
        rb->m_torque                = D3DXVECTOR3(0,0,0);
```

```
        // Just incase we get drifting in our quaternion orientation
        D3DXQuaternionNormalize(&rb->m_orientation, &rb->m_orientation);
    }

}
```

Code 7. Section of code that actually does the magic, by building the large sparse matrices, passing them to the solver which returns the corrected the forces and torques which we re-inject back into the simulator.

Of course, the solver is the one that is going to consume most of the time, as it has to determine the actual solution to the matrix. The code of our iterative solver, however, is really quite simple. As mentioned earlier, if one is dealing with real-time applications, an iterative solver should always be used.

Why do solvers break?

When implementing a solver, there are a few things to keep an eye out for:

- Overly constrained systems.
- Numerical error (drifting, floating point accuracy).
- Divide by zero errors.
- Impossible constraints.
- Invalid coordinates (e.g. placing rigid bodies at identical locations).
- Bad code (e.g. memory leaks, memory corruption, and incorrect implementation of algorithm).

Note

Diagonal Zeros – If there are zeros on the diagonal, something is wrong. It can cause the LCP solver to break, giving solutions that go to infinity.

Bad Constraints

There are times when the simulation constraints cannot be solved, usually because of human error or some unforeseen circumstances. This commonly occurs when constraints fight or violate each other, so that no one single solution exists to keep all the constraints valid. An example is illustrated below, whereby two objects are placed out of reach of one another using nail constraint, plus an additional joint constraint connecting them. These constraints contradict each other because the objects cannot be kept out of reach and also connected. For instance, in the example shown in Figure 8, illustrates an impossible constraint situation.

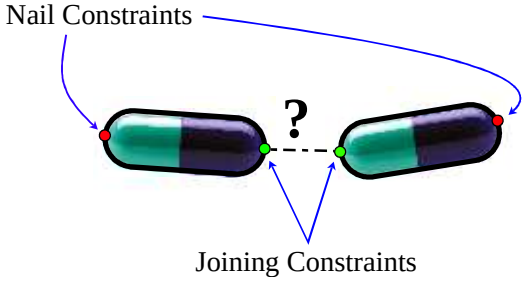

Figure 8. Impossible constraint situations.

These ill-conditioned constraint configurations can introduce erratic behavior into the simulation because there is no real solution. However our iterative solver, more often than not, aims to find a best-fit approximation, which will allow us to move forward in the hope that the configuration fixes itself.

Point-Mass Constraints (2D, No Rotation)

Using a simplified constraint solver, rotations can be cut out, just working with position constraints, enabling complex structures to be constructed in 2D or 3D just using point-mass objects. Simple constraints (e.g. nail and distance) let the solver be tested before moving onto rigid bodies and adding in rotations. Figures 9 and 10 illustrates examples of point-mass constraints for constructing articulated structures.

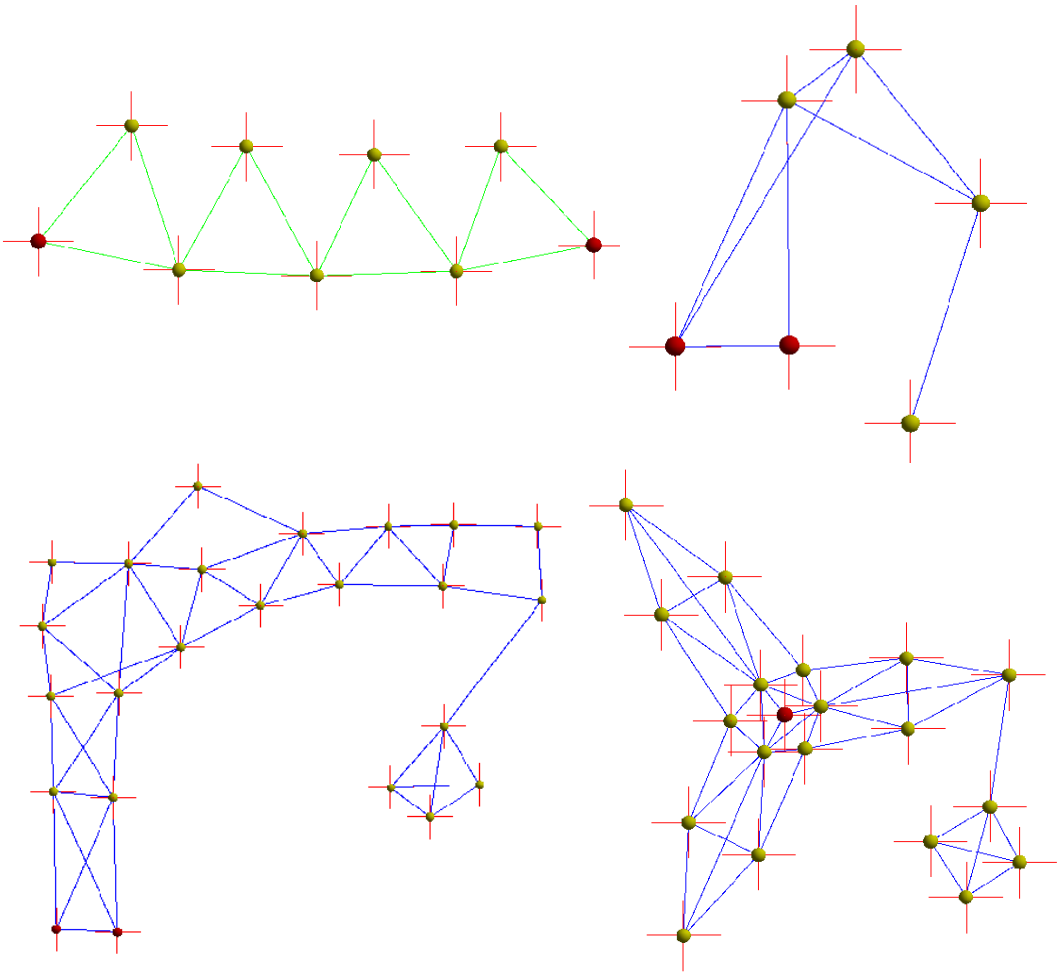

Figure 9. Simulation screenshots for Point-Mass solver. (Spheres represent connecting joint constraints, and the lines represent point-mass distance constraints).

The code, written for simplicity, has no optimisations, is inefficient and slow, but has been written to introduce a working solver that can be played with to gain understanding. All the particles have a mass of 1, but they can easily be set to different values – maybe draw larger spheres for larger mass? We used only two constraint types, fixed (nail) and distance (rods), but other constraints and springs can be mixed in. The simulation runs in 3D, so all the points and constraints actually solve for ‘x’, ‘y’ and ‘z’, but we

clamp the 'z' to zero and render in orthogonal to give the appearance of 2D; so elaborate 2D or 3D structures can be constructed.

We set a fixed number of iterations for the solver demo, so the constraints will stretch slightly for large forces. We thought this was acceptable, but the maximum iterations can be increased and the error can be monitored.

The mass-point solver in 2D/3D environments can be used to create dynamic structures, from which orientation and rotation information can be extracted, by using positions of point masses to build up a reference orientation.

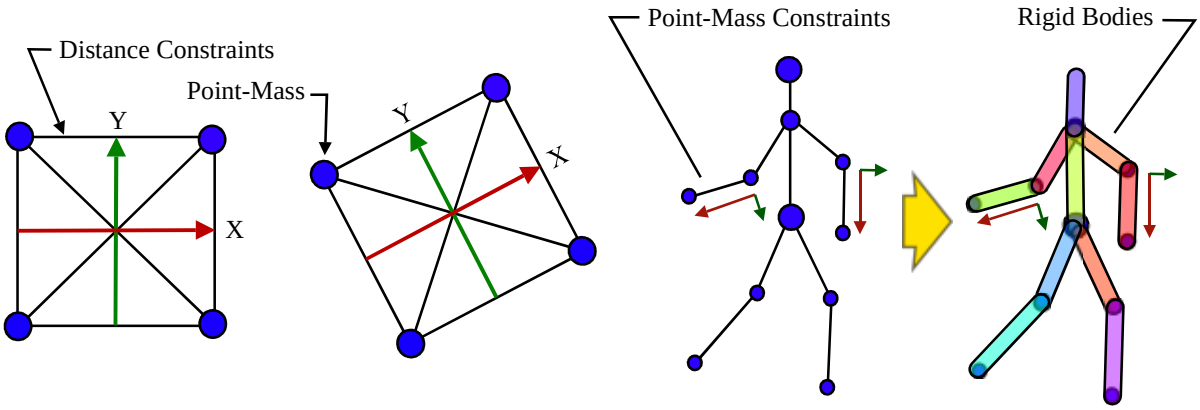

Figure 10. Use point-mass configurations to extract orientation (rotation) information for graphical models or rigid bodies.

Point Mass Demo (3D, No Rotation)

Expanding the simple point-mass demo to 3D, with a sphere dropping onto a mesh surface, we added constraints at each frame between the rigid body spheres to keep their distance to the sum of their radius, as shown in Figure 11.

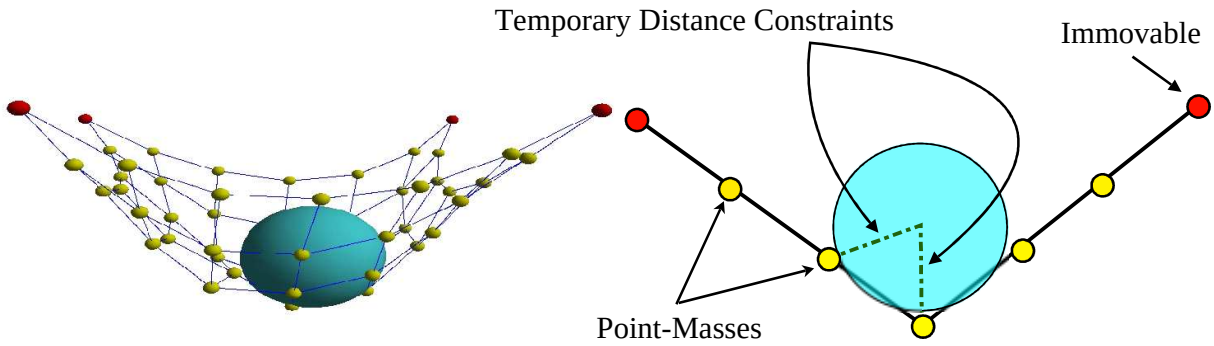

Figure 11. (Left) Demo screenshot – large sphere falling onto the distance constrained net. (Right) Simplified diagram showing how we handled collisions between spheres and mass-points.

Stable stacking demo (3D, With Rotation)

A good test for any physics simulator is the ability to deal with large stacks of objects. Creating a stable stacking configuration is in part down to how reliable the contact information is, and how much it changes between frames. To create reliable contact information, reliable contact manifolds need to be built, which are in contact across multiple frames. When we stack objects that are not moving, the collision information between frames should remain the same, and not jitter around due to floating point errors and re-calculation of collision points.

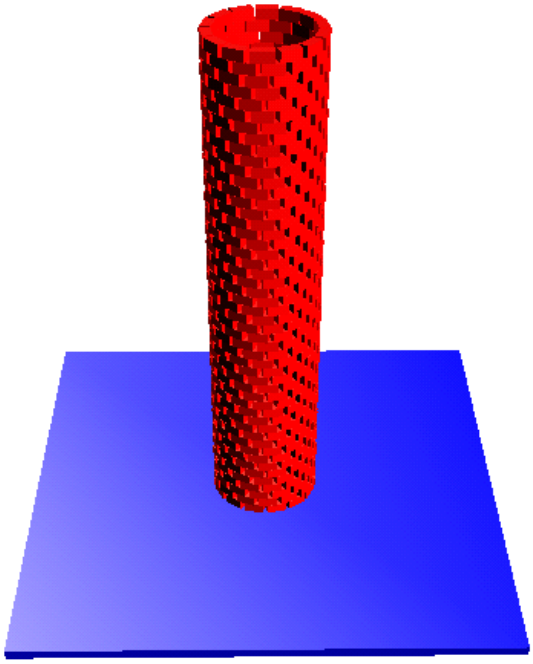

Figure 12. Stacking demo, 700 rigid body cubes in a circular tower.

SPARSE MATRICES

Applying conventional *dense matrix* techniques to solve systems of constraints presented in this chapter is an exceedingly ill-advised thing to do. As mentioned earlier, the matrices used in our solver are sparse, and we need to use this sparsity to our advantage. You can see a simple demonstration of the sparsity of a matrix illustrated in Figure 13, where we plot the non-zero matrix values to the right of the simulation screenshot in the figure.

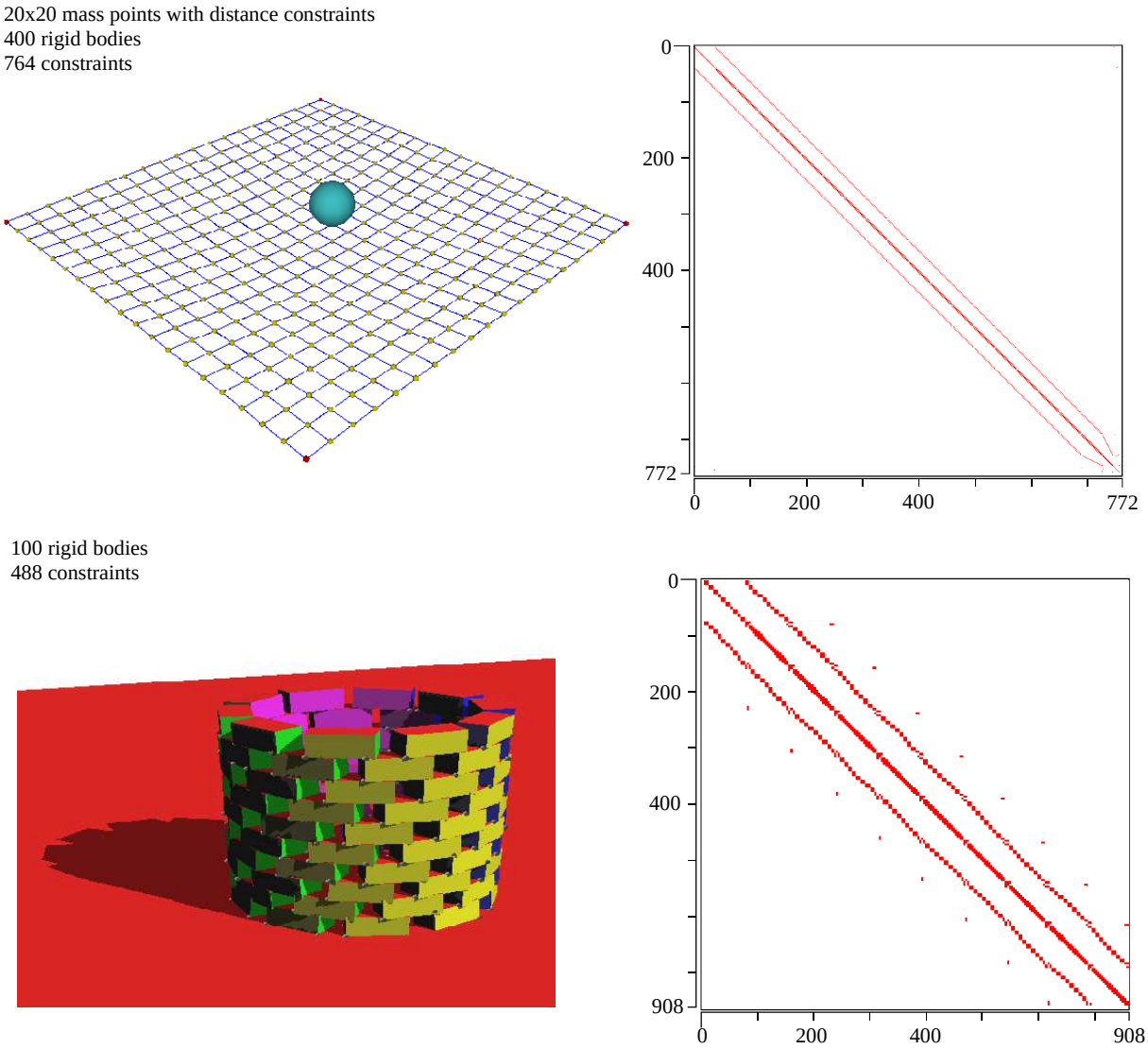

Figure 13. Sparsity pattern of the $JM^{-1}J^T$ matrix for constraint simulations. Top, mass-point distance constraints constructing a net (matrix is less than 1% filled). Bottom, circular stack of rigid bodies. The graphs to the right illustrate the matrix density.

Taking advantage of operator overloading, especially the multiplication operator '*', allows us to create more robust code with the ability to track down problems early on. Asserts and halts enable runtime problems to be detected, both by other people using the code, or just silly typing mistakes. These good coding techniques will help prevent memory corruption and random unpredictable crashes in unrelated code. Reliable code will 'stop' and tell us *why*. If possible, code that ignores a situation without telling us and attempts to cope with the problem and continue, should be avoided.

When implementing the matrix code, a wrapper class should be used so that various implementations for comparison and profiling can be swapped out. There are numerous matrix solutions freely available (Dongarra, 2009), each having their own advantages and disadvantages, and trading between various features such as memory and access times. We introduce a few methods here, giving the source for an STL mapping method.

Initially, the aim should be to get the LCP constraint solver working, without worrying too much about optimisation. Later, when things are working as expected, everything can be sped up. The first implementation will use a dense matrix method, and will initially be the biggest slowdown, because 90% of the time it will be managing the matrix data, where massive chunks of memory will be created, destroyed and copied in each frame.

Sparse matrices enable us to gain optimisation speedups while saving memory and bandwidth. In addition, our linear iterative systems are also able to exploit sparsity, by skipping unnecessary calculations, to improve performance, where even trivial operations like filling a $n \times m$ matrix with '0s' would take 'O(nm)' time, for a dense matrix can be avoided.

Note

A matrix wrapper class will give safety checks, an array of out of bound checks, and enable various internal optimisations (e.g. memory speedups) to be added without modifying the working solver code.

If multiple platforms are worked on, a lot of platform specific code inside the matrix class can be embedded without complicating the solver.

The matrix code can be expanded to use templates, so it can work with more data types, other than just floats, e.g. doubles, or any other objects, so it can be expanded to other situations, such as image processing or fluid dynamics.

One of the biggest slowdowns that will be encountered in solvers, other than actually solving the system of equations by inverting the matrix, is the dynamic nature of the matrices. Dynamic constraints, such as collisions and contacts can appear and disappear from frame to frame, which means resizing and updating the large matrix. Modifying the matrix by each frame, means introducing additional slowdowns, as the matrix class is allocating and reallocating large chunks of memory each frame, and if using the assignment operator, it is also being copied. For static constraints, there is the added benefit of coherency between frames, which we can exploit for speedups.

```
// Build ...
DMatrix MInverse(numBodies, numBodies);
DMatrix MFe(numBodies, 6);
DMatrix MVel(numBodies,6);

DMatrix JMatrix(constraints, numBodies);
DMatrix JTransposeMatrix = Transpose(JMatrix);

// .. fill with data

DMatrix A =  JMatrix * MInverse * JTransposeMatrix;
DMatrix b = JMatrix * ( MVel + MInverse * MFe );

// .. solve etc
```

Code 8. Showing a slice of code, where working with matrices is as easy as working with base variables like ints and floats.

In a poorly implemented solver with little consideration for matrix management, the biggest slowdown will be the matrices, where a large number of rigid bodies and constraints can produce very large matrices which are difficult to manage and manipulate and trigger large memory allocations and copies.

We solve our slow matrix implementation by taking advantage of our custom matrix wrapper class, where we are able to optimise things under the hood, using methods such as custom caching and memory pools. Internally, the matrix class should have a MatrixManager, which we shared across matrix instances, and stores common sized matrices, so instead of creating them, the manager would have them loaded and ready for use.

In those cases when the matrix has not actually been destroyed and created, but exists as a member variable, or a single instance, you can have it keep a large chunk of memory internally, so when the matrix is resized to make it smaller or bigger, it will keep enough memory internally for a larger matrix. So if the matrix through experimentation is no larger than 100, it could be set to allocate space internally to 100x100, but to the outside world it would appear to be whatever size it was resized to, e.g. 2x3, 30x60 etc., but it will have allocated 100x100 internally and avoid memory resize slowdowns.

Overloading the assignment operator is another good way to achieve speedups, where we can copy large chunks at a time internally instead of byte by byte, and if we are using sparse data representations for our matrix data, the amount of data that we will copy is a noticeably less, as we are only copying the non-zero data instead of the massive chunk of memory.

Dense Matrix (Brute Force)

So to begin with, we start with a basic dense matrix class, which we will call the DMatrix (i.e. Dynamic Matrix). Three essential variables are needed to begin with: the pointer to the data, the number of columns and the number of rows, as shown below:

```
class DMatrix
{
public:
    float*  m_data;     // rows x cols array of data
    int     m_numRows;
    int     m_numCols;

}; // End DMatrix Class
```

Code 9. 'Raw' starting point – the essentials only.

A few helper functions need to be added, so that all the data operations inside the class are managed. For example, when the DMatrix class is created, we need to be able to specify a default size and maybe the starting data, to initialize the array. We add some helper functions below, including a constructor, and add numerous assert checks.

```
class DMatrix
{
public:
    DMatrix(const int numRows, const int numCols, const float* data=NULL)
    {
        Init(numRows, numCols, data);
    }

    ~DMatrix()
    {
        delete[] m_data;
    }

    inline int GetNumRows() const   { return m_numRows; }
    inline int GetNumCols() const   { return m_numCols; }

    inline float Get(int row, int col=0) const
    {
      DBG_ASSERT(row>=0 && row<m_numRows);
      DBG_ASSERT(col>=0 && col<m_numCols);
      const int indx = GetDataIndex(row,col);
```

```
      return m_data[indx];
    }

    inline float& Set(int row, int col=0) const
    {
      DBG_ASSERT(row>=0 && row<m_numRows);
      DBG_ASSERT(col>=0 && col<m_numCols);
      const int indx = GetDataIndex(row,col);
      return m_data[indx];
    }

    inline int GetDataIndex(int row, int col) const
    {
      const int indx = col + m_numCols*row;
      DBG_ASSERT(indx>=0 && indx<m_numRows*m_numCols);
      return indx;
    }

private:
    void Init(const int numRows, const int numCols, const float* data=NULL)
    {
        DBG_ASSERT(numRows>=0 && numRows<500); // Sanity checks - temp, do we really have size
        DBG_ASSERT(numCols>=0 && numCols<500); // 0 or >500 array or did something go wrong?
        if (numRows==0) { DBG_ASSERT(numRows==0 && numCols==0); }
        if (numCols==0) { DBG_ASSERT(numRows==0 && numCols==0); }

        m_data      = NULL;
        m_numRows   = numRows;
        m_numCols   = numCols;


        if (numRows>0 && numCols>0)
        {
            m_data = new float[numRows * numCols];
            DBG_ASSERT(m_data);

            if (data)
            {
                memcpy(m_data, data, sizeof(float)*numRows*numCols);
            }
            else
            {
                memset(m_data, 0, sizeof(float)*numRows*numCols);
            }
        }
    }


    float*  m_data;
    int     m_numRows;
    int     m_numCols;

}; // End DMatrix Class
```

Code 10. Constructor and sanity checks – starting pieces.

If you look at the code above, you will notice that a large majority of the code is for checks. This is important, as it is far too easy for a mistaken value to propagate between functions and variables, to a point where it is impossible to track down the cause, leaving the system in an unrecoverable state. To reiterate, it is best to assert and stop as soon as a problem appears, so that the situation can be analysed and fixed.

There are various ‘sanity’ checks in the code, which are not vital, but give us an early warning that something might be incorrect. One such example of an early warning assert, is if the matrix size is greater than 500x500, we know something might have gone wrong, before attempting to allocate memory for some unexpected size matrix (i.e. 10000x10000, which can really be harmful). An additional check would be that zero size arrays are not created, and hence assert if the code attempts to, so we can track it down

and repair it. These checks give us a 'robust' matrix wrapper class that defines a set of common access functions from which we can swap in a better sparse matrix implementation at a later time.

Notice as well that the data and size variables have been made private. This is deliberate, so that all our internal workings for the matrix class remain hidden from the outside world, and use access functions to set or get the information within. Even simple functions such as 'Get', 'Set' and 'GetNumRows', can be used to give us extra checks, and enable us to change the code internally for any reason (i.e. data management optimisations). So to reiterate, this enables us to modify and improve our implementation without actually changing our simulator code, only the matrix code needs be changed. The dense matrix class gives a solid test bed from which results can be compared and checks made to ensure optimisations and improvements work correctly.

STL Map Matrix

Here, we present a practical, exploitable sparse matrix implementation that uses the Standard Template Library (STL). The code gives a skeleton implementation which can built upon, and uses a technique based on Compressed Row Storage (CRS) (i.e. similar to the method MatLab uses). The STL map matrix saves space by using more pointers to represent the data.

Matrix using Brute Force	Matrix using STL Map
(0%, 1%, 10% Random Fill) Size: 1 x 1, Bytes Used: 4 Size: 10 x 10, Bytes Used: 400 Size: 100 x 100, Bytes Used: 40000 Size: 1000 x 1000, Bytes Used: 4000000 Size: 10000 x 10000, Bytes Used: 400000000	(0% Fill) Size: 1 x 1, Bytes Used: 36 Size: 10 x 10, Bytes Used: 36 Size: 100 x 100, Bytes Used: 36 Size: 1000 x 1000, Bytes Used: 36 Size: 10000 x 10000, Bytes Used: 36 (1% Random Fill) Size: 1 x 1, Bytes Used: 36 Size: 10 x 10, Bytes Used: 120 Size: 100 x 100, Bytes Used: 6456 Size: 500 x 500, Bytes Used: 89520 Size: 1000 x 1000, Bytes Used: 299028 Size: 10000 x 10000, Bytes Used: 24481620 (10% Random Fill) Size: 1 x 1, Bytes Used: 36 Size: 10 x 10, Bytes Used: 756 Size: 100 x 100, Bytes Used: 28932 Size: 500 x 500, Bytes Used: 600924 Size: 1000 x 1000, Bytes Used: 2343684 Size: 10000 x 10000, Bytes Used: 229075404

Table 1. Comparing standard brute force matrix memory usage with that of an STL map one.

```
//  Sparse Matrix Implementation using STL Maps

#include <cstdlib>
#include <map>
#include <vector>

#define STL_CONST const

class DMatrixSTL
{
public:
    typedef std::map<size_t, std::map<size_t , float> >     mat_t;
    typedef mat_t::const_iterator                           row_iter;
```

```
    typedef std::map<size_t, float>                         col_t;
    typedef col_t::const_iterator                           col_iter;

    DMatrixSTL(size_t numRows, size_t numCols, const float* data=NULL)
    {
        m_numRows=numRows;
        m_numCols=numCols;

        DBG_ASSERT(numRows>=0 && numRows<1501);
        DBG_ASSERT(numCols>=0 && numCols<1501);
        if (numRows==0) { DBG_ASSERT(numRows==0 && numCols==0); }
        if (numCols==0) { DBG_ASSERT(numRows==0 && numCols==0); }

        if (data)
        {
            m_mat.empty();
            for (int i=0; i<(int)numRows; ++i)
            {
                for (int k=0; k<(int)numCols; ++k)
                {
                    const float val = data[ i + k*numRows ];
                    Set(i,k) = val;
                }
            }
        }
    }

    inline float Get(size_t row, size_t col=0) const
    {
        if(row<0 || col<0 || row>=m_numRows || col>=m_numCols) { DBG_HALT; }
        #if 0
        // Destroys sparsity and prevents the use of const
        // If the element doesn't exist, it inserts one.
        return m_mat[row][col];
        #else
        mat_t::const_iterator it;
        it = m_mat.find( row );
        float val = 0.0f;
        if ( it != m_mat.end() )
        {
            col_t::const_iterator itc = (*it).second.find( col );
            if ( itc != (*it).second.end() )
            {
                val = (*itc).second;
            }
        }

        return val;
        #endif
    }

    float& Set(size_t row, size_t col=0)
    {
        if(row<0 || col<0 || row>=m_numRows || col>=m_numCols) {  DBG_HALT  };
        return m_mat[row][col];
    }

    void SetToZero()
    {
        m_mat.empty();
    }

    inline int GetNumRows() const   { return m_numRows; }
    inline int GetNumCols() const   { return m_numCols; }

protected:
    DMatrixSTL(){}

public:
    mat_t   m_mat;
```

```
    size_t  m_numRows;
    size_t  m_numCols;
};

// Declarations for essential helper functions and operators overloading. e.g:
DMatrixSTL operator*( DMatrixSTL &a, DMatrixSTL &b );
DMatrixSTL Transpose(STL_CONST DMatrixSTL& m);
DMatrixSTL operator+ (STL_CONST  DMatrixSTL &a, STL_CONST DMatrixSTL &b);
inline DMatrixSTL operator- (STL_CONST DMatrixSTL  &a, STL_CONST DMatrixSTL &b);
inline DMatrixSTL operator*( float a, STL_CONST DMatrixSTL &b );
```

Code 11. STL Map Matrix.

Linear Linked-List Sparse Matrix

Linked-list matrices are an alternative way of building a sparse matrix class, and while probably not one of the fastest, they do offer large memory savings. One way the linked-list matrix would take advantage of sparsity, is by storing only non-zero elements in a list, and access them by it. It offers large memory savings but access times can be slow, since we need to find the index in the list, which can mean iterating over the list to find it.

This is a good solution for extremely sparse matrices, but as the matrix becomes less sparse, its memory overheads and access times grow exponentially. For example, the memory overhead of an empty linked-list class is can be as little as '12 bytes', no matter how many rows or columns, and whereas a dense matrix class of size 100x100 usually uses around 40,000 bytes, for a linked-list version, if every element contained a non-zero value, would use approximately 280,000 bytes. This is due to the linked-list overheads (i.e. a ratio of 28:1).

So what if 50% is used? Approximately 70,000 bytes would be used then; so as you would expect, the sparser the matrix, the better, not just with linked-list matrices but any sparse matrix implementation.

In conclusion, linked-lists offer a flexible data-storage solution that can be combined with other methods to achieve hybrid configurations, which offer both speed and memory savings. For example, speeding up element access times, by using nested linked-lists, or fast search algorithms, are just some ideas to explore.

Binary Tree (Search Speedup)

The binary tree version is basically a way of speeding up the linked-list version by storing the data in a binary tree formation. Whereby, when we search for a particular index, we search in a binary tree methodology, hence drastically reducing our search time.

Hashing

In order to achieve an alternative way of building a sparse matrix class to achieve exceptionally good access times, 'hashing' can be used, whereby indices are mapped to specific memory addresses using a hash lookup table and results are achieved in almost near instant access times. A typical implementing allocates a large number of elements, which are mapped to a hash lookup table, so each time an element is accessed, the hash algorithm generates the index offset within the large array. If there is no element at that address (i.e. if it does not exist), then it is added to the hash lookup list. Furthermore, because our matrix is sparse, the large chunk of elements we set aside should only be a fraction of the size of what a dense matrix allocates.

Note

Debugging – When working with solvers and matrices it is worth investing some time in learning about the various matrix types, e.g. adding code to check for singular matrices, and how 'bad' matrices can be

tested for. One such example is a matrix where one of its rows is all zeros, and understanding what this means.

Thinking out of the box

It is worth experimenting and profiling to aim for the lowest possible memory usage and fastest achievable access times. Simple things like caching and coherent memory accesses are good things to keep an eye out for. If you access an element at 'x', you might cache it so itself and its neighbours can be accessed immediately if the next access is itself or its neighbour.

Memory allocations are another factor, which can cause a major slowdown, so adding in a memory pool to the array creation/destruction helps with memory delays and fragmentation. Furthermore, using a memory manager in conjunction with the sparse matrix would enable partitioning of the memory and keeping track of usage; improving bandwidth and giving further control.

Cross platform computability is always a side thought, for example, working with matrices on the GPU, in which case the focus needs to be on parallelizing the data and modifying it to take advantage of the high number of cores. Further reading on exploiting LCP solvers on the GPU is available in (Nguyen, 2007).

STABILITY AND RELIABILITY

One of the biggest problems with writing physics simulators is reliable stable contact/collision information. One such example is an object resting on the ground. If the object moves within a minimum threshold, the contact information from previous frames should be reused so that it remains constant, and the objects converge and settle down. Some collision detection algorithms only return a single contact point between frames. Hence, this contact point needs to be stored and a contact manifold over numerous frames needs to be built up, which will keep the object from jittering and give a stable rigid body stacking.

When writing a simulator, asserts and sanity checks should be used whenever possible, especially checks for NaNs, which can arise, more often than not, due to the large number of mathematical operations. This will allow the simulation to be halted at the point where it went wrong, so an investigation of what went wrong at that moment in time can be made, instead of looking at debug prints or, even worse, randomly re-running the code and hoping for it to happen again and guessing what could have gone wrong.

CONCLUSION

In the scope of this chapter, we introduced LCP solvers as a practical solution for rigid body constraint simulations, with an emphasis on clarity and simplicity. It introduced the basic algorithms and their implementation so that the reader, having grasped the principles, can go on to more esoteric constructs.

We discussed and compared the two main solver types, acceleration and velocity, upon which we outlined their differences and finally went on to focus on the velocity method in the remainder of the chapter, due to its effectiveness and simplicity.

The chapter gave special emphasis towards the implementation design, whereby the reader could go away and construct a modular flexible simulator with the ability to extend it over time and add original joint types effortlessly; where, the crucial steps necessary for constructing new constraints types using numerous examples and their Jacobian formulation were presented in a simplified clear-cut approach.

FUTURE RESEARCH DIRECTIONS

This chapter has laid the practical foundations for using solvers to represent a robust constraint simulator. Further work entails numerous enhancements to increase performance, either by investigating and implementing more sophisticated algorithms (e.g. Successive Over Relaxation Method), or taking advantage of today's highly parallel processors (i.e. CPU or even better the GPU cores).
While we only introduced a few basic constraints, it is worthwhile experimenting and creating a wider range of unique constraints, both equality and inequality. When formulating constraint types, alternative derivatives of the equations should be researched and how they perform (e.g. length squared instead of length for distance constraints) should be compared.

This chapter has only been a springboard into the world of solvers and it is highly recommend that the reader look at the latest literature and explore the subject.

Possible side projects to continue with might include:

- Profiling and optimising code.
- Displaying kinetic/potential energy of objects.
- Creating complex constraint structures (bridges, ragdolls, catapult, ropes).
- Adding further distinctive constraint types.
- Displaying external and internal constraint forces.
- Using varying masses for point-mass distant constraint simulations.
- Analysing and optimising sparse matrix operations, and explore bandwidth speedups.

REFERENCES

Anitescu, M., & Potra, F. A. (1996). Formulating Dynamic Multi-rigid-body Contact Problems with Friction as Solvable Linear Complementarity Problems. *Computer*, (93), 1-21.

Baraff, D. (1989). Analytical Methods for Dynamic Simulation of Non-penetrating Rigid Bodies. *Computer*, *23*(3), 223-232.

Baraff, D. (1994). Fast contact force computation for non-penetrating rigid bodies. *Proceedings of the 21st annual conference on Computer graphics and interactive techniques - SIGGRAPH '94*, (May), 23-34. New York, New York, USA: ACM Press. doi: 10.1145/192161.192168.

Baraff, D. (1999). Physically based modeling course notes. *ACM SIGGRAPH*, (2-3).

Catto, E., & Park, M. (2005). Iterative Dynamics with Temporal Coherence, 1-24.

Cottle, R. W., Pang, J.-S., & Stone, R. E. (1992). *The Linear Complementarity Problem.*

David H. Eberly. (2004). *Game Physics. Morgan Kaufmann.*

Dongarra, J. (2009). Free Matrix Library List. Retrieved from http://www.netlib.org/utk/people/JackDongarra/la-sw.html.

Erleben, K. (2004). Stable, robust, and versatile multibody dynamics animation. *Unpublished Ph. D. Thesis, University of Copenhagen, Copenhagen*, *2004*(April). Retrieved July 23, 2011, from http://www2.imm.dtu.dk/visiondag/VD05/graphical/slides/kenny.pdf.

Guendelman, E., Bridson, R., & Fedkiw, R. (2003). Nonconvex rigid bodies with stacking. *ACM Transactions on Graphics*, *22*(3), 871. doi: 10.1145/882262.882358.

Hagger, W. W. (1988). *Applied Numerical Linear Algebra* (p. 528).

Havok. (1998). Havok Inc. Retrieved from http://www.havok.com.

Hecker, C. (1998). Chris Hecker. Retrieved from http://chrishecker.com/Rigid_Body_Dynamics.

Jerez, J., & Suero, A. (2003). Newton. Retrieved from http://www.newtondynamics.com.

Kaˇ, Z., Nordenstam, M., & Bullock, D. (2003). A Practical Dynamics System. *Work*, 7-17.

Lötstedt, P. (1984). Numerical simulation of time-dependent contact friction problems in rigid body mechanics. *SIAM J. on Scientific Computing*, *5*(2), 24.
Nguyen, H. (2007). *GPU Gems 3* (p. 1008). Addison-Wesley.
NVIDIA. (2011). PhysX. Retrieved from http://www.nvidia.com/object/physx_new.html.
Shabana, A. (1994). *Computational Dynamics*. John Wiley and Sons, Inc.
Smith, R. (2004). Open Dynamics Engine. Retrieved from http://www.ode.org.
Stewart, D., & Trinkle, J. C. (1996). An Implicit Time-Stepping Scheme for Rigid Body Dynamics with Coulomb Friction. *International Journal of Numerical Methods in Engineering*, *39*, 2673-2691. Ieee.
Vondrak, M. (2006). Crisis Physics Library. Retrieved from http://crisis.sourceforge.net/.

RECOMMENDED READING

Bourg, D. (2002). Physics for Game Developers, O'Reilly & Associates, Inc.
Cottle, R. W., and Dantzig, G (1968). "Complementarity Pivot Theory of Mathematical Programming." Linear Algebra and Its Applications 1, pp. 103–125.
Dantzig, G. B. (1963). Linear Programming and Extensions. Princeton University Press.
Eberly, D. (2003). Game Physics (Interactive 3D Technology Series), Morgan Kaufman
Erleben, K. (2005). Physics-Based Animation, Charles River Media
Hecker C. (2000). How to Simulate a Ponytail, In: http://chrishecker.com/How_to_Simulate_a_Ponytail/

Kokkevis, E. (2004). Practical Physics for Articulated Characters.
Kipfer P. (2007). GPU Gems 3, Chapter 33. LCP Algorithms for Collision Detection using CUDA
Murty, K. G. (1988). Linear Complementarity, Linear and Nonlinear Programming. Heldermann Verlag.
Smith, R. (2004). Open Dynamics Engine, In: http://www.ode.org/
Vondrak, M. (2006). Crisis Physics Engine, In: http://crisis.sourceforge.net/

KEY TERMS AND DEFINITIONS

Degrees of Freedom (DOF): The number of independent ways an object may move, and consequently the number of measurements necessary to document the kinematics of the object.

Equations of Motion (EOM): Is a set of equations, which describe how the objects of the system will move as time changes.

Constraint: Is a limitation or restriction on a Degree of Freedom (DOF) of the system.

Inertia: The tendency of an object in motion to remain in motion, and of an object at rest to remain at rest.

Mass: A measure of inertia, indicating the resistance of an object to a change in its motion; including a change in velocity. A kilogram is a unit of mass.

Centre of Mass (COM): Also known as the centre of gravity, and represents a centroid point where the mass of an object is balanced in all directions.

Weight: A measure of the gravitational force on an object; the product of mass multiplied by the acceleration due to gravity (equal to 9.8 m per second per second).

Moment of Inertia: The resistance of a body to change its state when rotating.

Acceleration: The rate of change of velocity.

Velocity: The rate of change of position.

Newton's Laws of Motion: The principles, formulated by Sir Isaac Newton (1642-1727), which state how objects move.

Linear System: A linear system is a collection of linear equations.