http://goo.gl/84N71q

Exploring MLCP solvers and Featherstone

Erwin Coumans

Iterative constraint solvers work great, but there are cases where we could use better convergence. This presentation explores various Mixed Linear Complementarity Problem (MLCP) solvers and the Featherstone articulated body algorithm and how to mix them.

The intention is to augment/extend the popular iterative constraint solver we all use and love, sequential impulse or variants of projected gauss seidel. In other words, we can use a better solution for part of the problem that needs it, but use our existing solver for the parts that we are happy with.

Erwin Coumans

2003-

2010-
GPU/OpenCL
VFX

2003-2010
PS2, PS3
Games

2001-2003
Havok 2.x

Thanks to Sony and AMD, I was able to work on the Bullet Physics SDK, 100% of my time, for over 10 years now. Most of the time I spend on a pure GPU rigid body dynamics engine, written entirely in OpenCL. Aside from that, I have been looking into improved constraint solvers and Featherstone, and that's what today's talk is about.

Before that, I would like to briefly show you a video of the OpenCL work. It is also explained in a new book, Multithreading in Visual Effects, which will be released around July this year.
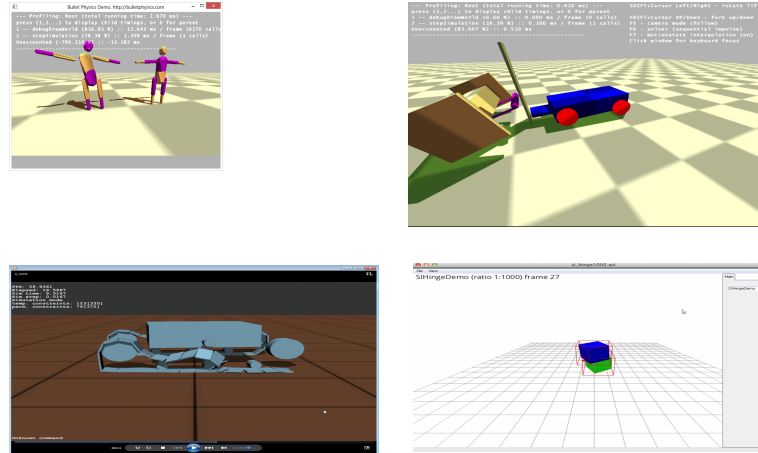
Last year at the GDC 2013 Physics Tutorial, Havok physics simulation architect Oliver Strunk discussed some of the issues with iterative constraint solvers in his 'Stop my Constraints from Blowing Up!' talk https://code.google.com/p/box2d/downloads/list

It is common for game physics developers to use a very low amount of iterations for an iterative constraint solver. When only using 4 to 10 solver iterations, the projected Gauss Seidel numerical method often fails to converge to a proper solution.

Oliver suggested avoiding large mass ratios, decreasing the time step and/or increase the inertia tensor.

There is plenty of evidence that game physics programmers are very creative dealing with large mass ratios.
"I feel ashamed for admitting this, but we sidestepped the whole problem of mass ratios in Halo 2. Havok allows you to mess with the collision response directly, so we simply pin the relative masses of objects so that there is never more than a 16:1 ratio between them."
http://www.bulletphysics.org/Bullet/phpBB3/viewtopic.php?f=4&t=54

Clamping the mass ratios can certainly help avoiding a lot of issues, but can we do better without cheating with the masses? Let's first have a closer look at some of the problems.

Show a few demos that demonstrate some issues with iterative constraint solvers, such as sequential impulse (SI)/projected Gauss Seidel (PGS) numerical method.

1) Stack heavy objects on light objects
2) Door hinge hitting heavy object
3) Ragdoll joint gap
4) Forklift joint gap
5) Heavy tank on tracks

How to improve Sequential Impulse

- **Overview of Rigid Body Simulation Loop**

- Add improved MLCP block solvers
  - Moving from SI to generic PGS MLCP solver
  - Try out NNCG, Dantzig, Lemke MLCP solvers

- Add O(n) solver for tree/chain equality joints
  - Featherstone Articulated Body Algorithm
  - Mixing ABA with Sequential Impulse

In this talk I want to share my experiences during my exploration into MLCP solvers and Featherstone articulated body method. This work is part of the Bullet Physics SDK and it started last year and it is on-going. I was using an iterative constraint solver, similar to Sequential Impulse, as Erin Catto described.

Although Sequential Impulse is an MLCP solver, its implementation and interface with the rest of the physics engine was not generic enough to allow easy plug and play of other MLCP solvers. For example we don't build an actual 'A' matrix. So I want to discuss how to move from a Sequential Impulse solver to a more generic format MLCP solver. A generic form MLCP of Sequential Impulse is basically projected Gauss Seidel.

Once we have an interface that can interface with generic MLCP solvers, we will have a look at some solvers, such as Non-linear Non-smooth Conjugent Gradient, Dantzig and Lemke.

For a chain or tree of equality constraints the Featherstone Articulated Body Algorithm is very suitable, and O(n). We will look at the basic parts of this algorithm, and how to integrate it with an existing constraint solver, such as Sequential Impulse.

In my first draft of this presentation, I realized that it would be useful to give an overview of some important parts of the rigid body simulation loop. This will make it easier to put things into context. So first, let's have a look at the overview.

Maximal versus Reduced Coordinates

Single 6DOF Rigid Body
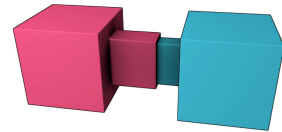and pair-wise constraints

Multibody with child links
and mobilizers

Hinge constraint
removing 5DOF
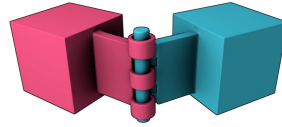
Revolute joint
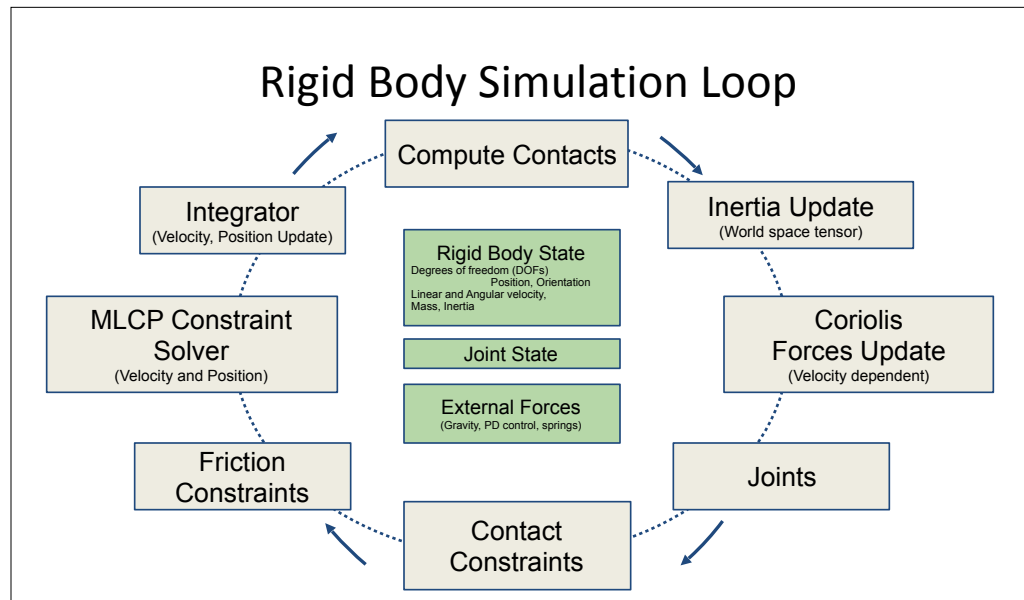adding 1 DOF
(mobilizer)

Slider constraint
removing 5DOF

Prismatic joint
adding 1 DOF
(mobilizer)

It is important to know the difference between maximal coordinates single rigid bodies attached by constraints and a reduced coordinate multibody. Single rigid bodies are represented with 6 degree of freedom and constraints are used to control those degrees of freedom. One of the issues of using maximal coordinates is that if the constraints are not fully satisfied, there can be a visible gap.
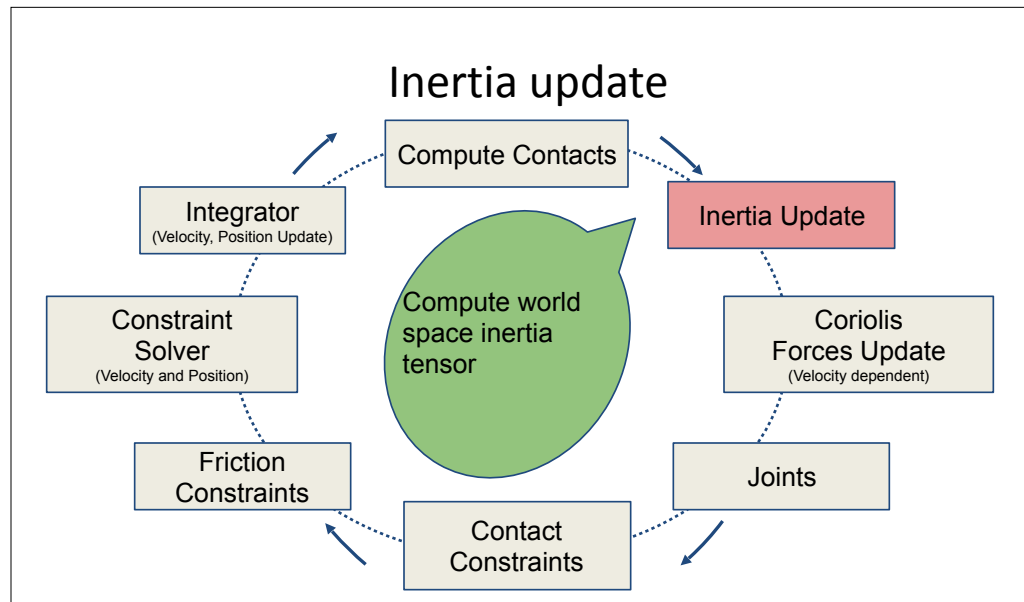
Reduced coordinate, or generalized coordinates methods only express the degrees of freedom. For example a hinge is expressed as a 1-DOF revolute joint angle. We can call such multibody joint a mobilizer to distinguish it from a joint constraint. Using such representation means that there can be no gap.

Rigid Body Simulation Loop

Compute Contacts

Integrator
(Velocity, Position Update)

Inertia Update
(World space tensor)

Rigid Body State
Degrees of freedom (DOFs)
Position, Orientation
Linear and Angular velocity,
Mass, Inertia

Coriolis
Forces Update
(Velocity dependent)

MLCP Constraint
Solver
(Velocity and Position)

Joint State

External Forces
(Gravity, PD control, springs)

Friction
Constraints

Joints

Contact
Constraints

Before going into detail, let's have a look at a simplified rigid body simulator loop. To keep it simple, we assume discrete simulation using a fixed small time step in the range of 30 to 1000 Hertz. So in the case of 60 Hertz, this loop is performed 60 times per second.

The state of the system includes the rigid body state variables. This includes the 6 degrees of freedom for translation (position) and rotation (orientation). In addition, bodies can be connected by joints. Those joints can be represented as constraints, or as mobilers of a multibody.

Furthermore there can be external forces such as gravity or user-defined forces. For example the user could simulate a spring or a controller by applying external forces. Let's have a closer look at the steps in this simulation loop, and what choices we made that affect simulation quality and stability.

Contact and Collision detection

Compute Contacts

Collision detection and contact computation

Inertia Update

Coriolis Forces Update
(Velocity dependent)

Joints

Contact Constraints

Friction Constraints

Constraint Solver
(Velocity and Position)

Integrator
(Velocity, Position Update)

Starting at the top, we perform collision detection and compute contact between rigid bodies. As we are concerned about discrete simulation, we are using discrete collision detection. It is common to compute one or more contact points between touching objects. A contact point has a distance, which is positive if the objects are separate and negative if there is penetration. A contact point also has a contact normal and two closest (witness) points, one on each object.

When dealing with a collision that involves one or more multibody, we need to determine the 'child' body, or link that is involved. Also, we might want to control if a multibody has self-collision, and if jointed links/bodies are allowed to collide. Self-collision might introduce some issue, as we briefly discuss later.

Inertia update

Compute Contacts

Integrator
(Velocity, Position Update)

Inertia Update

Compute world space inertia tensor

Constraint Solver
(Velocity and Position)

Coriolis Forces Update
(Velocity dependent)

Friction Constraints

Joints

Contact Constraints

To update a rigid body and multibody, we will need to have access to its mass and inertia, in world space or joint space. Usually we pre-compute the local space inertia tensor for a rigid body by applying the body world transform. As we will see, there is a little bit more work involved for multibodies to compute the articulated inertia tensor.

Velocity-dependent forces
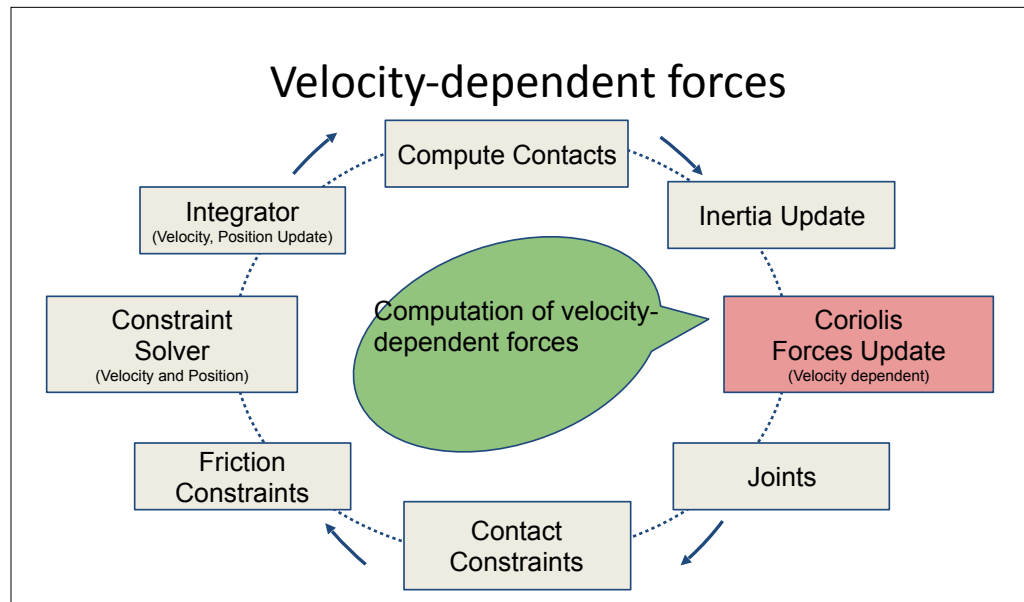
There are velocity-dependent forces, such as Coriolis forces. Those forces are necessary to maintain angular momentum. Think of a rotating skater on ice that rotates faster if he moves his limbs closer to the rotation axis. This is a quadratic term and this can cause a feedback loop that adds a lot of energy, causing instability. When using single rigid bodies, this term is often left out.
It is also possible to include the term using explicit integration, and adding some safety mechanism (hack) that adds artificial damping and clamp the maximum velocities.

It is also possible to use implicit integration, which adds damping as well. We will briefly show some way to derive the function for implicit integration.

Joints as Constraints or as Mobilizers
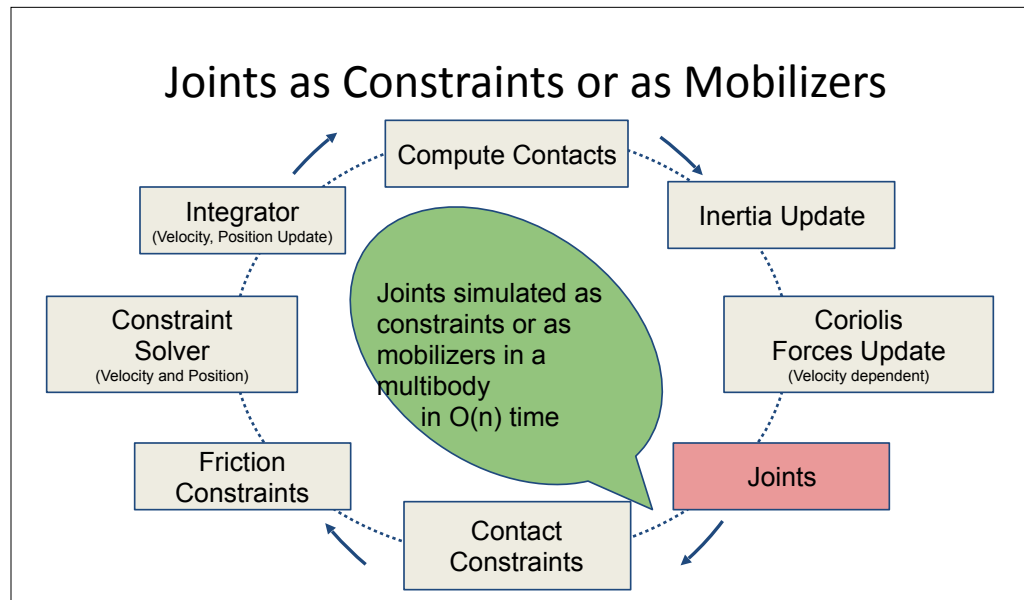
It is common to simulate joints, such as a hinge joint, using constraints that are solved using the SI or PGS constraint solver. Each rigid body has 6 degrees of freedom, and such joint constraints remove degrees of freedom. If the constraints are not properly solved, this will result in joint gap.

As an alternative, we can simulate joints as mobilizers of a multibody. In other words, a multibody consists of a chain or tree of bodies that are connected by mobilizers. Each mobiler adds one or more degrees of freedom. This is a reduced or general coordinate method, as we will discuss more in detail later.

In the context of reduced coordinate multibody simulation, this presentation I focus on propagation methods that have O(n) complexity. There are also multibody methods that explicitly build and invert the full matrix, such as the Composite Rigid Body Algorithm (CRBA) as discussed by Roy Featherstone. Due to the large matrix inversion, the time complexity is O(n^3). Nevertheless, for smaller n, such as a ragdoll or vehicle, the CRBA algorithm is attractive and has some performance benefits over the ABA. You can see comparisons and details in Jakub Stępień PhD thesis.

Roy Featherstone & RBDL
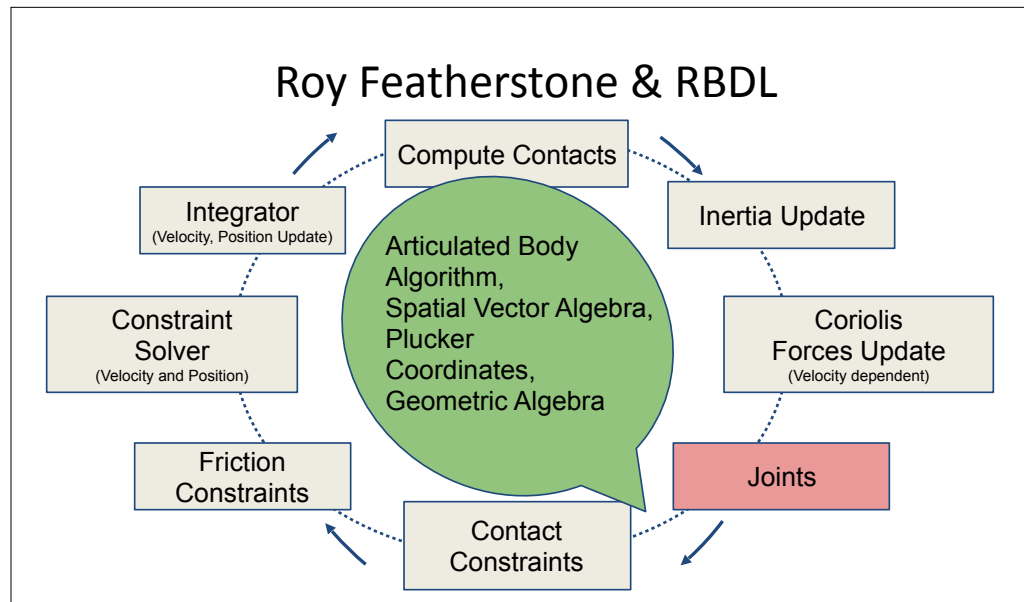
There are several ways to explain a reduced coordinate method to deal with joints in O(n) time, for n mobilizers. Roy Featherstone has written several papers and books about this topic, and introduce the Articulated Body Method. The presentation by Roy is very math-heavy and requires knowledge of geometric algebra with spatial vector algebra relying on concepts such as plucker coordinates, dual spaces. Gino van den Bergen introduced those concepts yesterday, in the context of inverse kinematics, in the Math Tutorial. If you are interested in an implementation that closely follows Roy Featherstone's book, I recommend looking at the open source RBDL library at github.
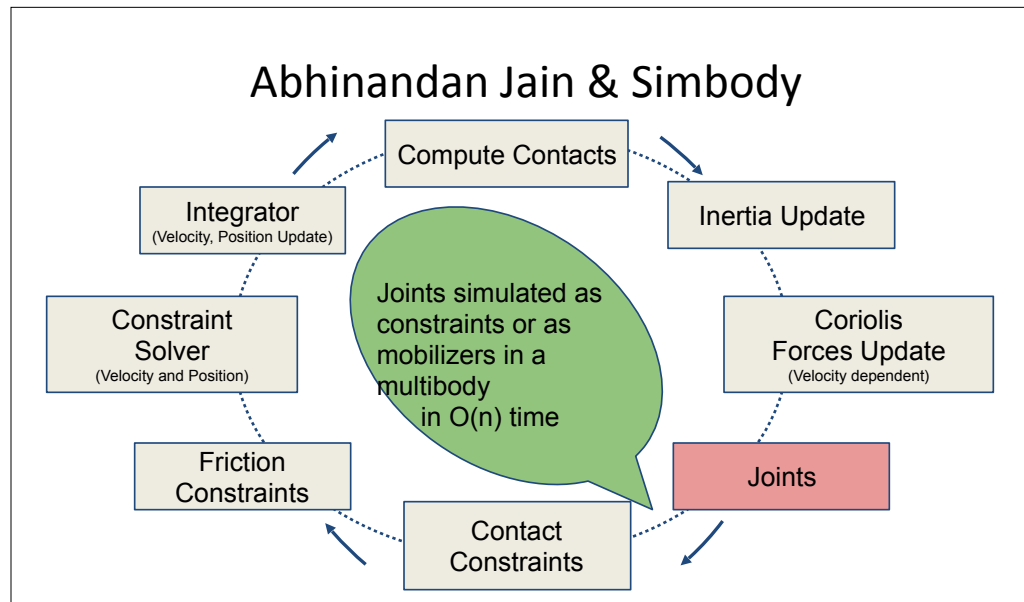
Abhinandan Jain from the Nasa Jet Propulsion Laboratory has written an excellent book about multibody dynamics. He uses a different way to derive the equations of motion that lead to a O(n) algorithm, similar to the Articulated Body Method. He uses spatial operators, not to be confused with spatial algebra. The open source Simbody project lead by Michael Sherman at Stanford University follows the notation and conventions of Jain's book.
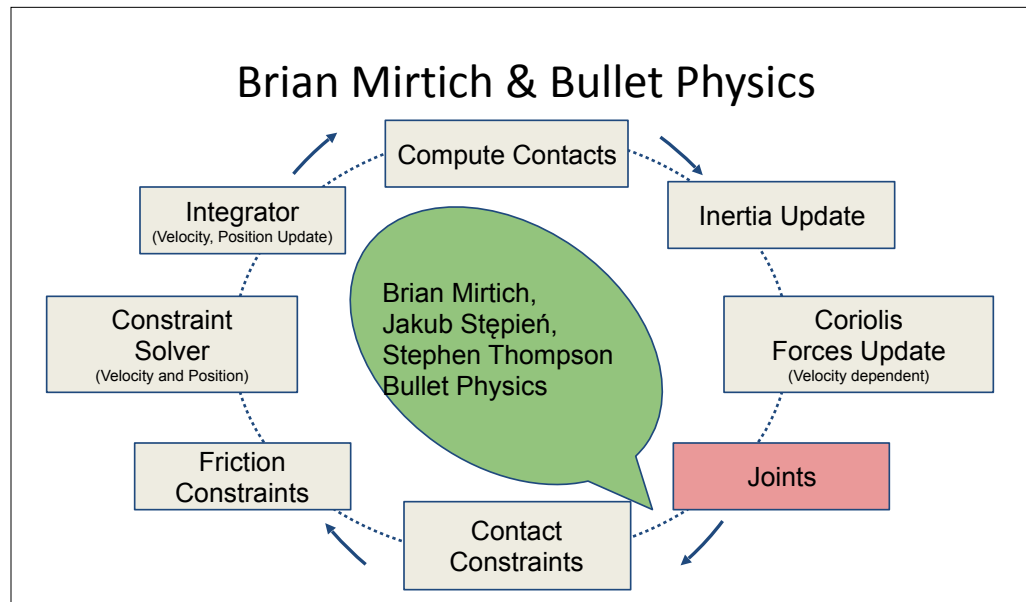
Brian Mirtich has written an excellent introduction to the Articulated Body Algorithm in his PhD thesis, chapter 4. We will use his spatial vector algebra notation, which doesn't require knowledge about plucker coordinates and such, so its presentation is a bit less alienating to mere mortals. The latest Bullet physics engine follows Brian Mirtich's notation and conventions.

Stephen Thompson implemented the first version of the reduced coordinate algorithm and I adapted it for proper integration into Bullet. Also, I integrated it tightly with the collision detection and existing constraint solver, and I added constraints limits and motors for btMultiBody.

Jakub Stępień extended the btMultiBody implementation in Bullet with multi-DOF joint mobilizers that are not described in Mirtich's thesis, but you can find the explanation in Roy Featherstone's book.

ABA+SI = Articulated Island Algorithm

Compute Contacts

Integrator
(Velocity, Position Update)

Inertia Update

Jakub Stępień
PhD thesis and
MultiDOF support
for Bullet btMultiBody

Constraint
Solver
(Velocity and Position)

Coriolis
Forces Update
(Velocity dependent)

Friction
Constraints

Joints

Contact
Constraints
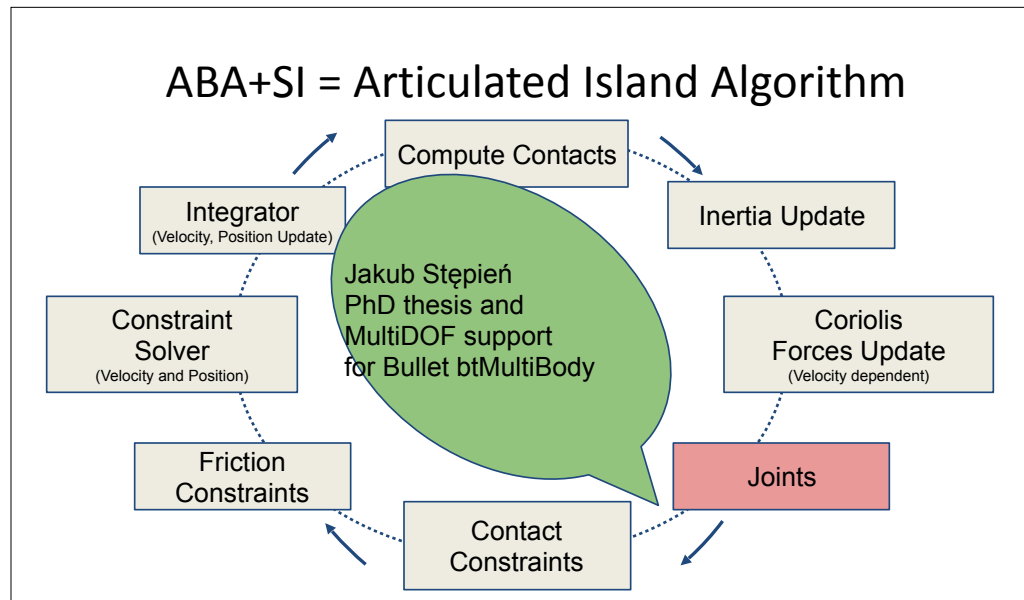
Jakub Stępień extended the btMultiBody implementation in Bullet with multi-DOF joint mobilizers that are not described in Mirtich's thesis, but you can find the explanation in Roy Featherstone's book.

Jakub also just finished his PhD thesis on the topic of integrating the Articulated Body Algorithm with Sequential Impulse. So if you are interested in mixing Articulated Body Algorithm and Sequential Impulse, I can highly recommend reading his thesis for much more details.

Contact constraints

Compute Contacts

Integrator
(Velocity, Position Update)

Inertia Update

Constraint Solver
(Velocity and Position)

Setup of contact constraints with variable degrees of freedom, and self collision between child bodies of a multibody.

Coriolis Forces Update
(Velocity dependent)

Friction Constraints

Joints

Contact Constraints
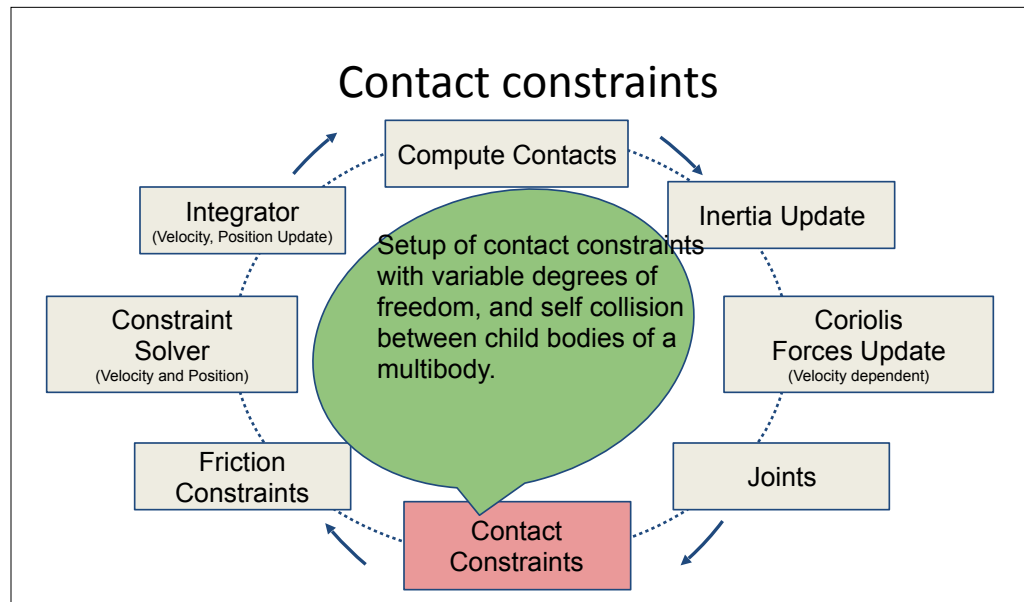
When formulating contact constraints, we need to take into account if the bodies involved are single rigid bodies or multibody. For a multibody we need to know the actual child bodies (links) involved. It is possible to have self-collision between different child bodies (links) of a multibody.

The size of the jacobian is the number of degrees of freedom, which is 6 for single rigid bodies. For multibody the number of degrees of freedom depends on the mobilizers.

Friction Constraints
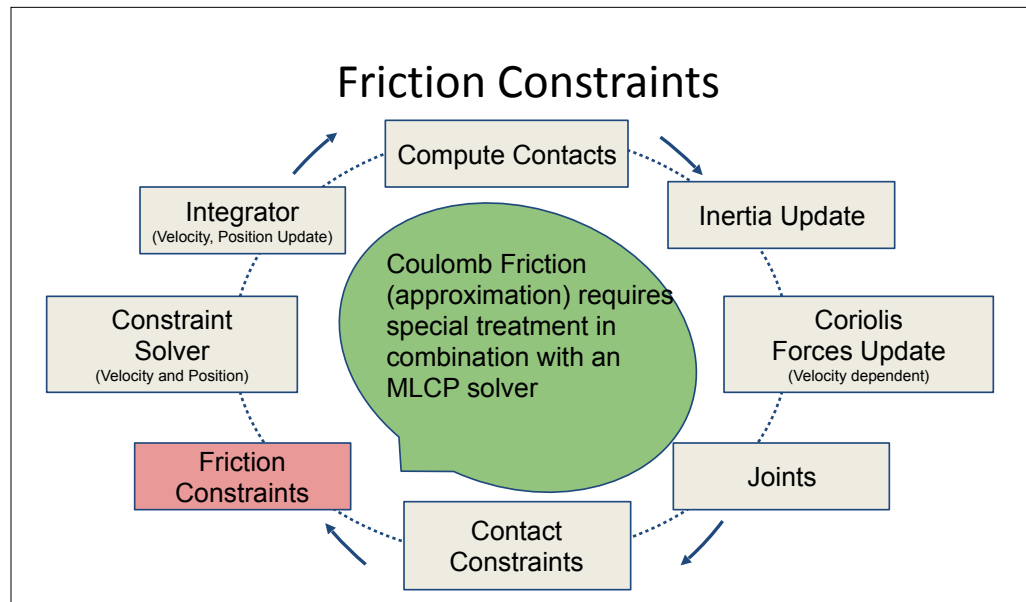
The magnitude of Coulomb friction is dependent on the normal contact force. This dependency is usually not expressed as an LCP condition. Usually there is a special treatment for the friction force limits, not part of the usual numerical method used to solve the constraints. It is important to realize what the impact is of the friction model and how it can be integrated into the numerical method solving an MLCP. Alternatively there is a cone approximation that explicitly encodes the dependency as an LCP condition.

From Sequential Impulse to PGS MLCP

- Compute Contacts
- Inertia Update
- Coriolis Forces Update (Velocity dependent)
- Joints
- Contact Constraints
- Friction Constraints
- Constraint Solver (Velocity and Position)
- Integrator (Velocity, Position Update)

Comparing Sequential Impulse with projected Gauss Seidel and preparation for direct MLCP solvers
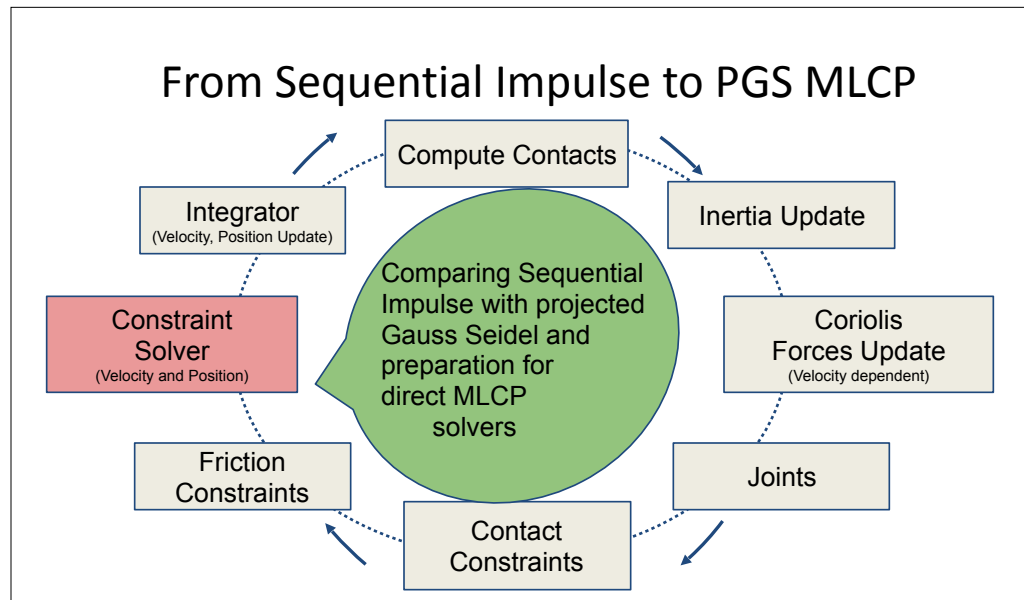
The normal contact, friction and other constraints are solved together using a constraint solver. As Erin Catto already mentioned, constraints can be formulated and solved on different levels: on the acceleration, velocity and/or positional level. In our context, we assume we deal with velocity constraints (make sure objects don't move towards each other when in contact) and positional constraints (move objects out of penetration).

In addition, we may solve those levels combined (coupled) as in Baumgarte stabilization, or we may solve the velocity and position constraints as two independent problems.

In gaming we usually use a variation of a Sequential Impulse (SI) solver or equivalent projected Gauss Seidel (PGS) solver, Erin Catto talked about this previously.

If your solver produces always perfect results that you are happy with, you could leave this presentation now, because the purpose of my talk is to explore how we can improve or augment this loop to get better results.

NNCG, Dantzig, Lemke MLCP solvers

Compute Contacts

Inertia Update

Coriolis
Forces Update
(Velocity dependent)

Joints

Contact
Constraints

Friction
Constraints

Constraint
Solver
(Velocity and Position)

Integrator
(Velocity, Position Update)
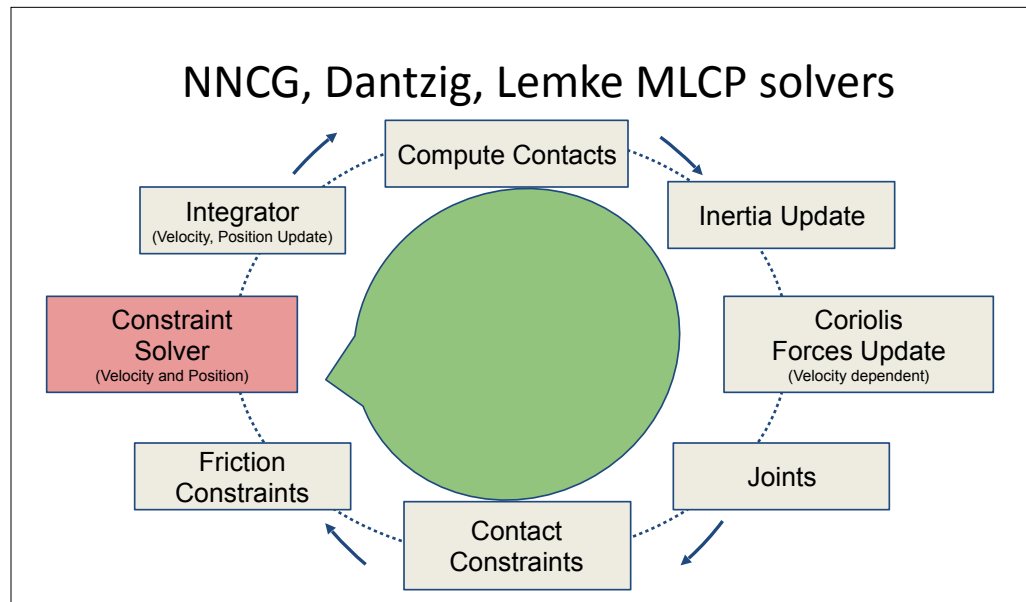
Kenny Erleben and his students published a paper describing the non-smooth conjugate gradient method. If you already have a sequential impulse solver implementation, it is pretty easy to extend it to support this method. Its convergence is better than plain PGS, but we need to let the solver use enough solver iterations to show benefits. I'll show some graphics with convergence comparison between PGS and NNCG in Bullet.

Position/Velocity Integration
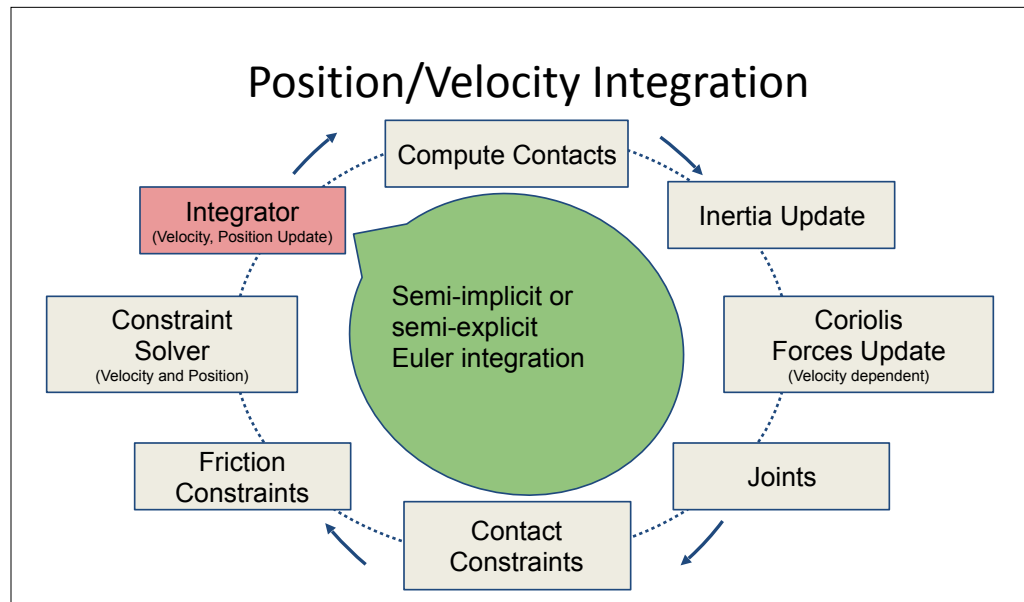
The semi-explicit or semi-implicit Euler integrator is usually used in the game physics context to update the position and orientation. When dealing with multibody the integrator will update all links in a multibody by propagating the transform from the base to the leaf links.

How to improve Sequential Impulse

- Overview of Rigid Body Simulation Loop

- Add improved MLCP block solvers
  - Moving from SI to generic PGS MLCP solver
  - Try out NNCG, Dantzig, Lemke MLCP solvers

- Add O(n) solver for tree/chain equality joints
  - Featherstone Articulated Body Algorithm
  - Mixing ABA with Sequential Impulse

Equations of motion

$$F = ma$$

$$\tau = I\dot{\omega} + \textcolor{red}{\omega \times I\omega}$$

$$v_{t+\Delta t} = v_t + a\Delta t = v_t + \frac{F_{ext} + F_c}{m}\Delta t = v_t + \frac{F_{ext}}{m}\Delta t + \frac{Impulse_c}{m}$$

$$x_{t+\Delta t} = x_t + v_{t+\Delta t}\Delta t$$

We use the Newton laws to govern the motion of rigid bodies. In the absence of interaction the motion of rigid bodies would be simple: we take a time step of delta t and perform an (semi) explicit Euler integration step. The term in red is called the Coriolis force, a non-linear term. The combination of explicit Euler integration, a large time step and non-linearity can cause energy gain. For this reason, the term is often left out in game physics libraries. It is possible to use implicit integration to deal with this Coriolis term in a better way. I added some slide at the end, thanks to Dirk Gregorius. Also, Joseph Cooper made a derivation, used in the Open Dynamics Engine.
http://lost-found-wandering.blogspot.com/2013/01/gyroscopic-forces-in-ode.html

When we add physical interaction between objects, we can express interactions using constraints resulting to constraint forces or impulses. While the external forces are given, we need to compute the constraint forces or impulses.

Question

The ultimate question we want to answer is - how big of an
impulse do we need to apply to make the objects stop moving
towards each other?

Let's assume we're applying an impulse of magnitude 1.0 (unit
impulse) and see how big of a velocity change that would
cause.

Dennis Gustavson has an interesting blog about game physics bits, and in one posting he describes the working of sequential impulse in layman's terms.
http://tuxedolabs.blogspot.com/2010/08/explaining-rigid-body-solver.html When we want to compute the magnitude of an impulse in a certain direction, we can
do this empirically: measure the effect of an impulse on the velocity. Let's look into this a bit closer and relate this empirical method to other methods that
compute the relationship between impulse and velocity.

Effect of an impulse on velocity

$$Impulse = F\Delta t = m\Delta v$$

$$Impulse_{torque} = \tau\Delta t = I\Delta\omega$$

$$\Delta v = \frac{Impulse}{m}$$

$$\Delta\omega = \frac{Impulse_{torque}}{I} = \frac{r \times Impulse}{I}$$
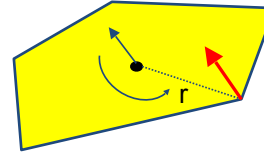
When we apply an impulse to a rigid body, it can have an effect on the linear velocity and the angular velocity of the center of mass. Using Newton's second law and the relationship between impulse and force, the change in linear velocity v, or delta v is Impulse over mass. In a similar way we can compute the change in angular velocity, delta omega, using Newton's second law for rotation. Note that we need to know the effect of an impulse on the velocity at a point on the body, not the effect on the center of mass. The torque impulse can be computed given an impulse and a point at an object using the cross product of the relative position r with the impulse.

Single body Effective Mass (inverse)

- Effective mass is change in velocity per impulse
  - projected on the normal

$$u_a = v_a + \omega_a \times r_a$$

$$\Delta u_a = \frac{Impulse}{m_a} + (\frac{r_a \times Impulse}{I_a}) \times r_a$$

$$effective\_mass_{inv} = \frac{dot(\Delta u_a, normal)}{|Impulse|}$$
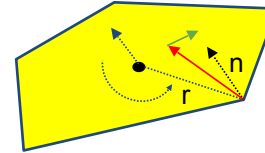
The velocity in a point on a body is related to the linear and angular velocity of the center of mass. The velocity at a point on the body adds the linear velocity v and the cross product of omega and the arm vector r. Let's call the point velocity u, to distinguish it from the linear velocity v.

When we substitute the effect of the impulse on linear and angular velocity of the center of mass, into the point velocity formula from previous slide, we get our desired effect, delta u for body a. The effect of an impulse is a change in velocity, a vector.

The scalar number, expressing the linear relationship between the application of a (unit) impulse and the change in velocity projected on the impulse direction (normal) can be viewed as the "inverse effective mass". Note that the impulse magnitude is 1 of course, so the division by the impulse magnitude has no effect.

So far we only looked at a single body a. If we collide with a static, non-moving object, we are done.

Effective Mass (inverse)

$$u_{ab} = u_a - u_b$$

$$= v_a - v_b + \omega_a \times r_a - \omega_b \times r_b$$

$$\Delta u_{ab} = \Delta v_a - \Delta v_b + \Delta \omega_a \times r_a - \Delta \omega_b \times r_b$$

$$\Delta u_{ab} = \frac{n}{m_a} - \frac{-n}{m_b} + (\frac{r_a \times n}{I_a}) \times r_a - (\frac{r_b \times -n}{I_b}) \times r_b$$

$$effective\_mass_{inv} = \frac{dot(\Delta u_{ab}, normal)}{|Impulse|}$$
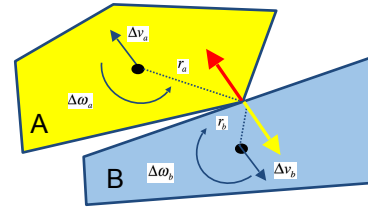
For a collision between two dynamic moving rigid bodies, where body a and body b have positive mass, we compute the relative velocity by subtraction. The same applies for the change in relative point velocity between two colliding bodies. Again we need to substitute the impulse into the formula of the previous slide, and we get the actual effect. To get the scalar inverse effective mass ratio we project the change in relative velocity onto the normal, using a dot product, just as we did with a single body.

Resolving a single collision
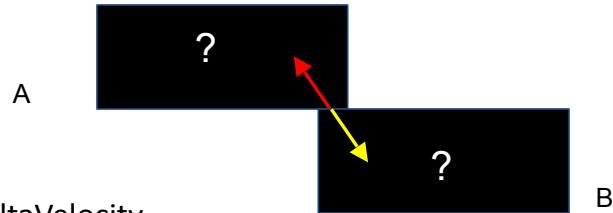
- Black box interface
- computeDesiredDeltaVelocity
- computeEffectiveMass(unitImpulse)
- applyImpulse

- Black box could be particle, rigid body, multibody

Now we know the effect of an impulse, and we know that there is a linear relationship between the impulse and the relative point velocity, we can go back to the original problem: how to make the objects stop moving towards each other.

Resolving a single collision
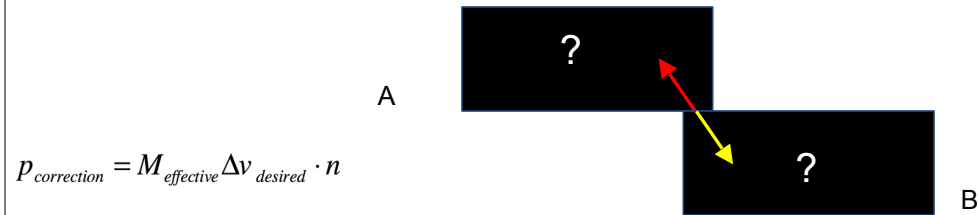
$$p_{correction} = M_{effective} \Delta v_{desired} \cdot n$$

$$p_{correction} = \frac{-\Delta v_{ab} \cdot n}{n \cdot n \left(\dfrac{1}{M_a} + \dfrac{1}{M_a} \right) + \left(\dfrac{r_a \times n}{I_a} \right) \times r_a + \left(\dfrac{r_b \times n}{I_b} \right) \times r_b}$$

Given the desired velocity change, and inverse effective mass, we can compute the correction impulse p.
If we substitute the delta v and the effective mass we showed previous, we get the collision impulse formula, also described by Chris Hecker long time ago.

Mixed Linear Complementarity Problem

$$A\lambda + b \geq 0$$
$$\lambda \geq 0$$
$$\lambda(A\lambda + b) = 0$$

$v = Ap + b$

p

v
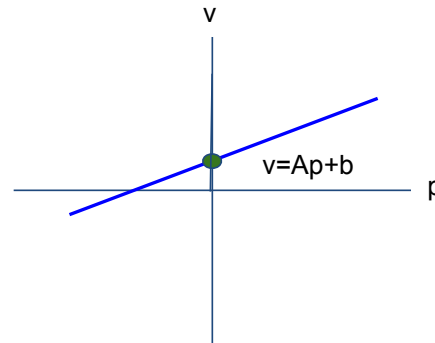
$$M^{-1}_{effective}p - \Delta v_{desired} = 0$$

We can substitute A for the effective mass inverse, and lambda for impulse p, and b for the change in relative velocity.
Then we add two conditions, that the impulse has to be larger than zero (don't attract) and either the relative velocity is zero, or the impulse is zero. Now we end up with the Linear Complementarity Problem, or LCP.
In the case of a collision, we have an inequality: the constraint condition has to be larger or equal than zero.

There are a few other ways of formulating this problem, I can recommend Kenny Erleben's SIGGRAPH 2013 tutorial on this topic. For a single constraint row, the problem is not very interesting. Once we get more constraint rows, the problem becomes more interesting and harder to solve. We call the problem Mixed when there is a mix of inequality and equality constraints. If all constraints rows are equality constraints, we can solve the problem using a linear equation solver, which is a different topic. Lets have a look at the most popular way of solving the MLCP: projected Gauss Seidel aka Sequential Impulse.

3 variations of building the A matrix

$$A = \frac{dot(\Delta u_{ab}, normal)}{|Impulse|}$$

$$= n \quad K \quad n^T$$

$$= J \quad M^{-1} \quad J^T$$

There are several variations to compute the A matrix for the MLCP. We already discussed the unit impulse method. A very similar collision matrix K method uses the cross product matrix notation. We can also compute the A matrix by explicitly building the Jacobian matrix and the inverse mass matrix and multiplying them.

Unit Impulse method

$$\Delta u_{ab} = \frac{n}{m_a} - \frac{-n}{m_b} + (\frac{r_a \times n}{I_a}) \times r_a - (\frac{r_b \times -n}{I_b}) \times r_b$$

$$A = inverse\_effective\_mass_{ab} = \frac{dot(\Delta u_{ab}, normal)}{|Impulse|}$$

There are various ways to compute this (inverse) effective mass, or A matrix. For a single contact the A 'matrix' is just a 1x1 scalar value. In the previous slides there is an intuitive way to get to this matrix. You could think of this as a unit-impulse method: you apply a unit impulse and read the effect on the relative velocities.

Cross product matrix

$$a \times b = \begin{bmatrix} a_y b_z - a_z b_y \\ a_z b_x - a_x b_z \\ a_x b_y - a_y b_x \end{bmatrix}$$

$$a \times b = \begin{bmatrix} 0 & -a_z & a_y \\ a_z & 0 & -a_x \\ -a_y & a_x & 0 \end{bmatrix} b$$

$a^\times b$

Another way of computing the A matrix for a contact is using the 3x3 collision matrix K. The collision matrix K can be computed using cross product matrices. This cross product matrix is also known as a skew symmetric matrix, and it can be also used for the derivative of a rotation matrix. In a nutshell, the cross product matrix of a vector 'a' can be multiplied with another vector to get the cross product with 'a'. The 3x3 cross product matrix is often written as a small superscript cross symbol above the vector.

Collision Matrix K method

$$K = \begin{bmatrix} \dfrac{1}{m_a} + \dfrac{1}{m_b} & 0 & 0 \\[2mm] 0 & \dfrac{1}{m_a} + \dfrac{1}{m_b} & 0 \\[2mm] 0 & 0 & \dfrac{1}{m_a} + \dfrac{1}{m_b} \end{bmatrix} - (r_a^{\times} I_a^{-1} r_a^{\times} + r_b^{\times} I_b^{-1} r_b^{\times})$$

$$A = dot(\Delta u_{ab}, n) = nKn^T$$

Using this cross product matrix, we can formulate the 3x3 K matrix. The derivation steps to go from the top formula to K is described in Brian Mirtich's PhD thesis in Appendix A2. Using the collision matrix K we get to the A matrix by pre/post multiplying with the n vector. We will see the this collision matrix again in the interaction between Featherstone and MLCP constraints. One of the benefits of the K matrix is that it can be re-used between several constraint rows that use the same bodies and local positions. For example the normal constraint and one or two friction constraints can re-use the same K matrix, saving some computation (double-check this).

Jacobian and Mass Matrix method

$$J = \left[n, r_a \times n, -n, -r_b \times n \right]$$

$$M = \begin{bmatrix}
m_a & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & m_a & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & m_a & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & I_{a_{xx}} & I_{a_{xy}} & I_{a_{xz}} & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & I_{a_{yx}} & I_{a_{yy}} & I_{a_{yz}} & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & I_{a_{zx}} & I_{a_{zy}} & I_{a_{zz}} & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & m_b & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & m_b & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & m_b & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & I_{b_{xx}} & I_{b_{xy}} & I_{b_{xz}} \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & I_{b_{yx}} & I_{b_{yy}} & I_{b_{yz}} \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & I_{b_{zx}} & I_{b_{zy}} & I_{b_{zz}}
\end{bmatrix}$$

$$A = JM^{-1}J^{T}$$

A third way of computing the same A matrix is by multiplying with a mass matrix. For a contact constraint, we stack the linear (normal) and angular (arm cross normal) in a 1x6 vector for a single object, or 1x12 vector for two objects. This vector is also called Jacobian, see also Erin Catto's talk slides. We get the A matrix by pre/post multiplying the inverse mass matrix with the jacobian. The 3 methods look a bit difference, but the end result is the same A matrix, or in the case of a single constraint row, a scalar a.

Creating the MLCP input

- Sparse large vector and large matrix
  - 'Eigen' math library or custom
    btMatrixX<...>,btVectorX<...>

- Convert constraint info into A and b (rhs)
  btMLCPSolver::createMLCP or
  btMLCPSolver::createMLCPFast

MLCP Interface

```cpp
class btMLCPSolverInterface
{
public:
  virtual ~btMLCPSolverInterface()
  {
  }

  //return true is it solves the problem successfully
  virtual bool solveMLCP(const btMatrixXu & A, const btVectorXu & b, btVectorXu& x,
              const btVectorXu & lo,const btVectorXu & hi,
              const btAlignedObjectArray<int>& limitDependency,
              int numIterations, bool useSparsity = true)=0;
};
```

$$A\lambda + b \geq 0$$
$$\lambda \geq 0$$
$$\lambda(A\lambda + b) = 0$$

We use an abstract interface to try out various MLCP solvers. As you can see, the interface takes a few more parameters, next to the A, b and x (lambda). There are low and hi limits for each variable, and limit dependencies for friction. Furthermore there are some additional parameters specific to iterative constraint solvers, the number of iterations, and if the A matrix is sparse or not.

Gauss Seidel MLCP solver

```
for (int k = 0; k <numIterations; k++)
{
        for (i = 0; i <numRows; i++)
        {
                delta = 0.0f;
                {
                        for (j = 0; j <i; j++)
                                delta += A(i,j) * x[j];
                        for (j = i+1; j<numRows; j++)
                                delta += A(i,j) * x[j];
                }
                x [i] = (b [i] - delta) / A(i,i);
        }
}
```

In its original form, Gauss Seidel is a very simple iterative numerical method to solve the MLCP. See also the Cottle and Dantzig book for much more in-depth information. In this slide you can see an actual working implementation in a handful of lines. The x vector is the same as the lambda vector in our previous slide. Note that the A matrix is sparse: typically we only have 12 non-zero elements in each row, so with some minor modification we make it a very efficient implementation using sparse operations.
You can notice the division by the diagonal element A(i,i). This is one of the reasons of numerical issues: once the diagonal elements go close to zero we can introduce all kind of errors. Some popular workaround for this is to add a small epsilon to the main diagonal, also called Constraint Force Mixing (CFM). Adding small values to the main diagonal can help avoiding numerical explosions, but you changed the problem so you don't get the actual result you need. Adding CFM can make the simulation look less stiff, more 'spongy' or 'springy'. The pure Gauss Seidel method doesn't deal with inequality so we need to amend this.

Projected Gauss Seidel

```
for (int k = 0; k <numIterations; k++)
{
        for (i = 0; i <numRows; i++)
        {
                delta = 0.0f;
                {
                        for (j = 0; j <i; j++)
                                delta += A(i,j) * x[j];
                        for (j = i+1; j<numRows; j++)
                                delta += A(i,j) * x[j];
                }
                float aDiag = A(i,i);
                x [i] = (b [i] - delta) / A(i,i);
                if (x[i]<lo[i])
                        x[i]=lo[i];
                if (x[i]>hi[i])
                        x[i]=hi[i];
        }
}
```

In order to deal with inequality constraint rows, we can project the solution vector x within upper and/or lower bounds, as is shown in the blue lines.

Cheating with Friction

```
for (int k = 0; k <numIterations; k++)
{
  for (i = 0; i <numRows; i++)
  {
    delta = 0.0f;
    {
            for (j = 0; j <i; j++)
              delta += A(i,j) * x[j];
            for (j = i+1; j<numRows; j++)
            delta += A(i,j) * x[j];
    }
    float aDiag = A(i,i);
    x [i] = (b [i] - delta) / A(i,i);
    float s = 1.f;
    if (limitDependency[i]>=0)
    {
      s = x[limitDependency[i]];
    }
//in case of friction constraint rows, the lo/hi is the friction coefficient
    if (x[i]<lo[i]*s)
      x[i]=lo[i]*s;
    if (x[i]>hi[i]*s)
      x[i]=hi[i]*s;
  }
}
```

Now we want to add friction constraint rows, using an approximation of Coulomb friction. We want the friction force to be linearly related to the normal force of the same contact. This introduces a dependency that is not part of the original MLCP problem. The easiest way is to cheat and adjust the lower and upper limits based on the most up-to-date value of the related normal constraint force (or impulse) within the iteration loop. The blue lines show the minor changes to support this friction hack.

One issue is that adding such dependency makes the problem non-linear and we cannot prove convergence. The hack seems to work pretty well and for gaming purposes we are often happy with the result. There is a non-hacky way of introducing the friction constraint row dependencies into the MLCP problem formulation, but there will be zero's on the main diagonal. In other words, we cannot use PGS (and related SI) to solver this problem.

The projected Gauss Seidel implementation was first published in the Open Dynamics Engine around 2004, known as the Quickstep solver. Kenny Erleben documented the approach in his PhD thesis and his book 'Physics Based Animation'.

Cone Friction
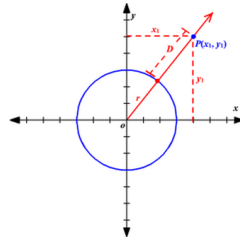
$$\sqrt{\lambda^2_x + \lambda^2_y} - r$$

In many cases, the friction cone is approximated by a pyramid. When using projected Gauss Seidel, it is possible to use cone friction by using two orthogonal friction directions and clamping the two friction directions simultaneously into the cone.

Sequential Impulse (SI) versus PGS

```
for (int k = 0; k <numIterations; k++)
{
  for (i = 0; i <numRows; i++)
  {
  delta =c.m_rhs- c.m_appliedImpulse;
  dV1Dotn=         c.m_n.dot(body1.m_deltaV)+
c.m_relpos1XNormal.dot(body1.m_angVel);
  dV2Dotn=//just like dV1Dotn
  delta-= dV1Dotn*c.m_jacDiagABInv;
  delta-=dV2Dotn*c.m_jacDiagABInv;
  sum =c.m_appliedImp +delta;
  if (sum < c.m_lowerLimit){
    delta = c.m_lo - c.m_appliedImp;
  c.m_appliedImp = c.m_lowerLimit;
  } else if (sum > c.m_upperLimit) {
    delta = c.m_hi-c.m_appliedImp;
    c.m_appliedImp = c.m_upperLimit;
  }else
    c.m_appliedImpulse = sum;
    bodyA->applyImpulse(n*delta);
    bodyB->applyImpulse(-n*delta);
  }
}
```

```
for (int k = 0; k <numIterations; k++)
{
  for (i = 0; i <numRows; i++)
  {
    delta = 0.0f;
    {
      for (j = 0; j <i; j++)
        delta += A(i,j) * x[j];
      for (j = i+1; j<numRows; j++)
        delta += A(i,j) * x[j];
      }
    x [i] = (b [i] - delta) / A(i,i);
    float s = 1.f;
    if (limitDependency[i]>=0)
    {
      s = x[limitDependency[i]];
    }
    if (x[i]<lo[i]*s)
      x[i]=lo[i]*s;
    if (x[i]>hi[i]*s)
      x[i]=hi[i]*s;
  }
}
```

Erin Catto published the Sequential Impulse method, which is closely related to the projected Gauss Seidel method in previous slide. The SI method is more intuitive and applies its operations directly using the rigid body and constraint objects, rather than creating the A, x and b vectors explicitly. A benefit of SI over PGS is that we can tune and optimize specific parts of the solver, because we can make use of the specific constraint types and rigid body information. Plain PGS has not such knowledge: the type information is lost during the conversion into the A matrix and b vector. As you can see, the x/lambda vector of PGS is the accumulated applied impulse, stored in the constraint (row). In addition, SI applies the impulse directly to the rigid body velocity, while PGS uses only the the x/lambda vector. The innerloop of PGS that accumulates the delta using the x vector is basically the dot product in SI on the body linear/angular velocity. The dot product applies to linear and angular velocity for each body, so in total there are 12 values: the total number of degrees of freedom (DOF) for both bodies.We will see later that when integrating Featherstone with PGS/SI, we will use the DOFs of the Featherstone multibody instead.

In other words, the difference between SI and PGS is not in the algorithm but in the implementation.

Gauss Seidel Hinge Stress Test
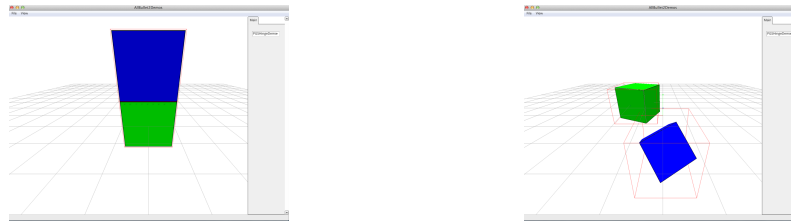

(video demo of PGS Hinge Stress Test)
In this stress test we place a blue 1000kg cube on top of a green cube of mass 1. The green cube is attached to the world using a hinge with the world up-axis as hinge axis. The PGS solver obviously doesn't converge to a stable solution, and the hinge constraint is visibly violated.

Chain with various mass ratios

1:1   1:10   1:100   1:1000
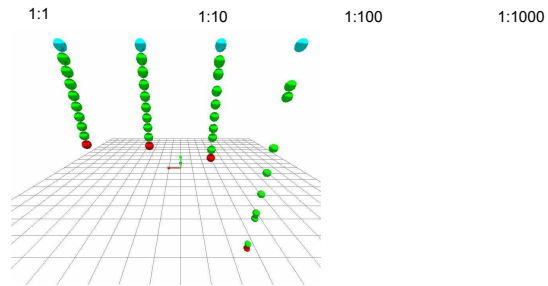
In a different stress test we can see a problem if we attach a large mass to a chain of objects with smaller masses. The blue ball is fixed with mass 0. All the green balls have mass 1 and the red ball has mass 1, 10, 100 or 1000. All objects are attached using a point to point (also known as spherical or ball socket) constraint. In this test we are using default earth gravity and a timestep of 60 Hertz and 10 solver iterations using a sequential impulse solver aka PGS.

Chain: SI/PGS at 1:1000, 60Hz
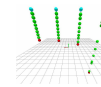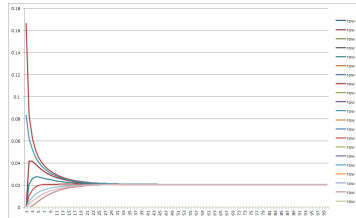
Here is a plot for the constraint rows with a mass ratio 1:1000. The velocity error is not reducing, even after adding many iterations: it stays around 0.02. Only after 10000 iterations or so, it goes to zero, obviously impractical for game or real-time purposes with current hardware.

How to improve Sequential Impulse

- Overview of Rigid Body Simulation Loop

- Add improved MLCP block solvers
  - Moving from SI to generic PGS MLCP solver
  - Try out NNCG, Dantzig, Lemke MLCP solvers

- Add O(n) solver for tree/chain equality joints
  - Featherstone Articulated Body Algorithm
  - Mixing ABA with Sequential Impulse

Nonlinear Nonsmooth
Conjugate Gradient solver

- Extension to a SI/PGS solver
- Projected Conjugate Gradient
- Uses PGS to compute residual

Before looking at direct MLCP solvers and Featherstone, there is an interesting algorithm called Nonlinear Nonsmooth Conjugate Gradient or NNCG. This is also an iterative constraint solver, where we interleave the PGS iterations with a Conjugate Gradient step. When we solve the PGS constraint rows, we can keep track of the constraint error, using the 'delta'. This can be used as the residual for the Conjugate Gradient solver. More details are in the paper. In our experience, the NNCG convergence is better than PGS, but this is usually visible at a high number of solver iterations.

NNCG Residual at 1:1000, 60Hz
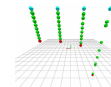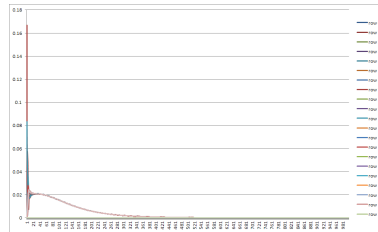
Now using 1000 iterations the benefit of NNCG becomes clear: at around 400 iterations we approach convergence, while PGS doesn't converge until 10000 iterations or higher.

Direct solver: Dantzig

- Original version has no friction
  - Added in Baraff 1994
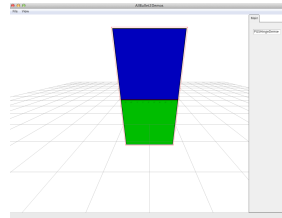
Video of the Hinge stress test, using the Dantzig MLCP solver.
Todo: present basic steps of Dantzig without friction, and with friction and show the convergence curves.

The Dantzig algorithm to solve an LCP problem shows very good convergence for problems with large mass ratios. It is common for direct LCP solvers to be more sensitive for degenerate cases, such as elements on the main diagonal of the A matrix close to zero. We can add some small epsilon (also called constraint force mixing, CFM) to the main diagonal to reduce the numerical issues. In addition, it is possible to detect a failure in the Dantzig algorithm, and use a backup algorithm such as Lemke or PGS.

Baraff extended the Dantzig algorithm to deal with friction, in his SIGGRAPH 1994 paper "Fast Contact Force Computation for Nonpenetrating Rigid Bodies".

For an excellent introduction to the Dantzig algorithm and the friction extension, see chapter 14 of the book "Guide to Dynamic Simulations of Rigid Bodies and Particle Systems" by Murilo Coutinho, Springer Verlag 2013.

Let's have a look at the performance.

Dantzig solver performance
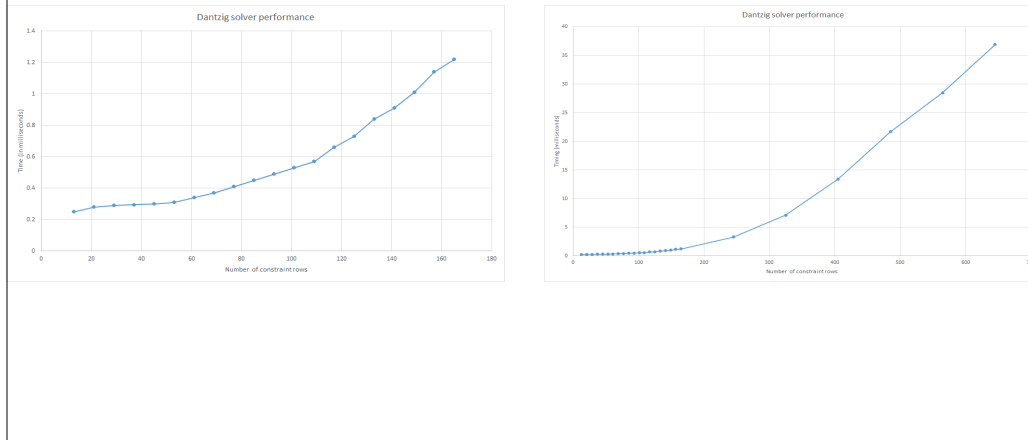
Obviously, the Dantzig algorithm doesn't scale well for larger problems. It takes about 1 millisecond to solve 200 constraint rows.

You can see the scaling problem in the right graph: at 600 constraint rows it takes about 30 milliseconds just for the solver. It is possible to use the Dantzig algorithm as a block solver as part of a PGS solver.

Remarks on Dantzig solver

- Can be used as a block solver with PGS
- Occasionally fails, needs fallback (PGS)

The projected Gauss Seidel method solves one constraint row at a time. As Erin Catto already mentioned in his presentation, it is possible to solve multiple constraint rows at a time, using a block solver. Dantzig has good convergence and pretty good performance for reasonably small constraint sizes so it can be a good candidate for solving blocks.

Direct solver: Lemke
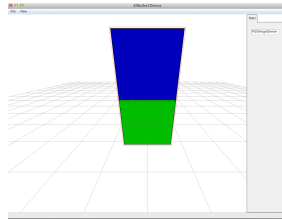
Another MLCP solver is the Lemke algorithm. Like the Dantzig algorithm, Lemke is a direct pivoting solver. Lemke can deal with a larger class of MLCP problems, compared to PGS and it converges well. In the worst case, the Lemke algorithm takes an exponential number of steps, and in some cases it fails to find a solution. Like Dantzig, it is a good idea to detect failure and limit computation time and use a fallback algorithm such as PGS.

Reformulating a bounded LCP to regular LCP

$$Ax + b, lo \leq x \leq hi$$

$$\begin{bmatrix} A^{-1} & -A^{-1} \\ -A^{-1} & A^{-1} \end{bmatrix} \begin{bmatrix} x^+ \\ x^- \end{bmatrix} + \begin{bmatrix} -A^{-1}b - lo \\ hi + A^{-1}b \end{bmatrix}$$

$$x = A^{-1}(x^+ - x^- - lo)$$

The Lemke algorithm takes a LCP problem, without lower or upper bounds. This means that we have to reformulate a bounded LCP problem (BLCP) with lower and/or upper bounds into a different LCP problem without those bounds. Given a LCP of n variables and lower and upper bounds for each variable, we can convert the problem into an LCP with 2n variables.

See Appendix A1 of this paper for a derivation of BLCP to LCP http://www.cs.duke.edu/~parr/nips10.pdf and the Matlab program https://github.com/erwincoumans/num4lcp/blob/master/matlab/test_lcp_bounds.m

One more way to add friction
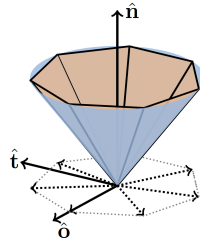
$$\begin{bmatrix} M & J & 0 & 0 \\ J^t & 0 & 0 & 0 \\ D^t & 0 & 0 & E \\ 0 & \mu & -E^t & 0 \end{bmatrix}$$

John Lloyd 'Fast Implementation of Lemke's Algorithm for Rigid Body Contact Simulation'.

There exists some methods to integrate the friction constraints as part of the LCP matrix, see the mu and E sub matrices. One issue is that this matrix is not positive semi-definite, with zeros on the main diagonal. This means we cannot use PGS/SI to solve the problem, and we need a more advanced LCP solver such as Lemke.

See also John Lloyd 'Fast Implementation of Lemke's Algorithm for Rigid Body Contact Simulation'.

PATH, COMPASS and so on

- PATH is a closed source MLCP solver
  - High quality but still solver failures

- COMPASS is an MLCP solver written in Matlab. Pretents to be PATH clone.
  - Can use a C++ to Matlab bridge
  - We also created a C++ conversion
  - Turned out to be not robust

Another MLCP solver is the Lemke algorithm. Like the Dantzig algorithm, Lemke is a direct pivoting solver. Lemke can deal with a larger class of MLCP problems, compared to PGS and it converges well. In the worst case, the Lemke algorithm takes an exponential number of steps, and in some cases it fails to find a solution. Like Dantzig, it is a good idea to detect failure and limit computation time and use a fallback algorithm such as PGS.

How to improve Sequential Impulse

- Overview of Rigid Body Simulation Loop

- Add improved MLCP block solvers
  - Moving from SI to generic PGS MLCP solver
  - Try out NNCG, Dantzig, Lemke MLCP solvers

- Add O(n) solver for tree/chain equality joints
  - Featherstone Articulated Body Algorithm
  - Mixing ABA with Sequential Impulse

Articulated Body (Multibody)
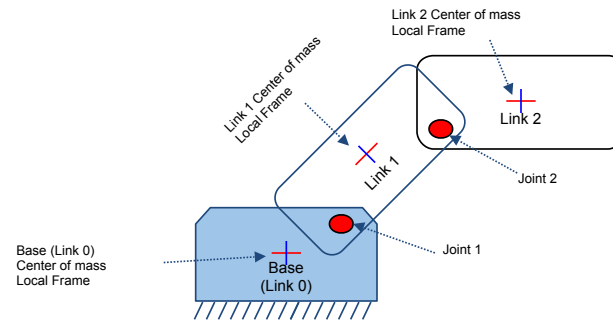
When dealing with bodies using reduced or generalized coordinates there are some conventions that are different from maximal coordinate methods. Bodies are called links, and links are connected by joints. In a similar way to a compound rigid body we view the hierarchy of bodies/links connected by joints as a single entity, often called Multibody or a 'Featherstone Hierarchy'. We follow the naming conventions of Brian Mirtich in his PhD thesis, chapter 4 gives an excellent introduction of the Featherstone Articulated Body Algorithm (ABM).

When dealing with maximal coordinates rigid body and pairwise constraints, we solve the constraints, get the change in velocity for each body and then we compute the velocity and position update individually. In the generalized coordinates approach the position, velocity and acceleration of links are expressed relative to a parent. So in order to compute those states, we start from the base and recurse into the child links. Another difference between the maximal and generalized coordinate approaches is the way we deal with mass and inertia: in maximal coordinates pairwise constraints, we only consider the mass and inertia of the individual two bodies involved. In the generalized coordinates approach, we compute the inertia of the entire hierarchy. Let's first see how we can propagate the position and velocity for a Featherstone hierarchy.

Joint Coordinates q (Revolute)
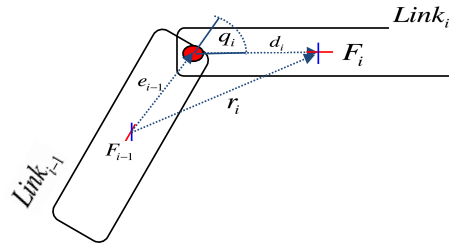
$$F_i = F_{i-1} + e_{i-1} + R(u_i, q_i)d_i$$

The pose of an articulated hierarchy is defined by its joint coordinates. In the case of a hinge, or revolute joint, such coordinate q is the joint angle, a scalar value. In case of a slider, or prismatic joint, the joint coordinate is the joint translation, again a scalar value. Each link has a local frame F. The child frame F_i can be computed by starting at the parent frame F_{i-1} and adding the transform from parent frame to joint frame, and from joint frame to child frame. The transform from parent frame to joint frame is usually constant. The transform from joint frame to child frame depends on the joint coordinate, joint location and joint type. Here we show an example computation for a revolute joint, with a joint angle q_i and a joint axis u_i. The R is the rotation matrix for the joint axis u_i at an angle q_i.

Relative Link Velocities (Revolute)
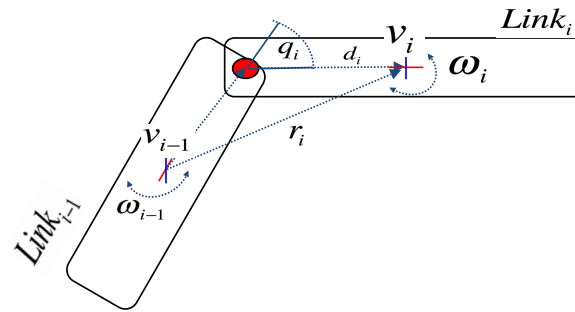
$$\omega_{rel} = q'_i u_i$$
$$v_{rel} = (q'_i u_i) \times d_i$$

The joint velocities q' is also a scalar value, for a revolute and prismatic joint. We can convert the joint velocities into relative link velocities of the center of mass frame F_i, relative to the parent frame F_i-1. This is the velocity due to the motion of joint i. For a revolute joint, the relative angular velocity is simply multiplying the joint velocity q' times the joint axis u_i.

todo: perhaps add prismatic joint example too

Propagating Link Velocities
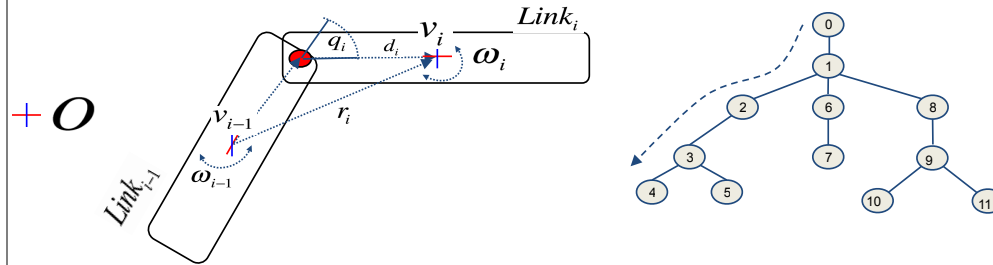
$$\omega_i = \omega_{i-1} + \omega_{rel}$$

$$v_i = v_{i-1} + w_{i-1} \times r_i + v_{rel}$$

We can propagate the relative link velocities from parent to child to get the link velocities. We can start at the base, using a link velocity relative to some coordinate frame, say O, and propagate the velocity outward to the child links.

Propagating Link Velocities

$$\omega_i = \omega_{i-1} + \omega_{rel}$$

$$v_i = v_{i-1} + w_{i-1} \times r_i + v_{rel}$$

```
omega[0]=baseAngularVelocity;
vel[0] = baseLinearVelocity;

// Calculates the velocities of each link (and the base) in its local frame
for (int i = 1; i <numLinks; i++)
{
    int parent = m_links[i].m_parent;

    // Transform parent vel into this frame, store in omega[i], vel[i]
    // SpatialTransform
    omega[i] = m_links[i].m_cachedRotParentToThis * omega[parent];
    vel[i] = m_links[i].m_cachedRVector * vel[parent];

    // Add relative link velocities due to joint i
    omega[i] += getJointVel(i) * m_links[i].m_angularJointAxis;
    vel[i] += getJointVel(i) * m_links[i].m_linearJointAxis;
}
```

Now the same propagation of link velocities in basic C++ code. Note that the links are sorted so that all links that appear closer to the base come before child links. This means we can also support tree structures, not just a serial chain.

Propagating Link Accelerations
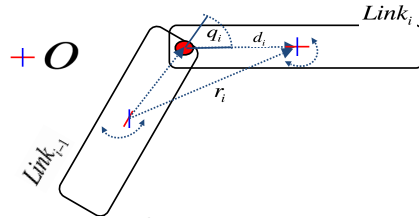
$$\omega_i = \omega_{i-1} + \omega_{rel}$$

$$v_i = v_{i-1} + w_{i-1} \times r_i + v_{rel}$$

$$\alpha_i = \alpha_{i-1} + \dot{\omega}_{rel}$$

$$a_i = a_{i-1} + \alpha_{i-1} \times r_i + \omega_{i-1} \times (\omega_{i-1} \times r_i) + \omega_{i-1} \times v_{rel} + \dot{v}_{rel}$$

$$\dot{v}_{rel} = \frac{d}{dt}(q'_i u_i) \times d_i = \omega_{i-1} \times ((q'_i u_i) \times d_i) +$$

$$(\ddot{q}_i u_i) \times d_i + (q'_i u_i) \times ((q'_i u_i) \times d_i))$$

In a similar way we can propagate angular accelerations. We can derive the accelerations by differentiating the velocity propagation formulas. A tricky part is figuring out the derivative of the relative linear and angular velocity. The red terms in the formulas are Coriolis and centrifugal terms. There is still a part missing, the joint acceleration term q".

6D Spatial algebra notation

$$\hat{v} = \begin{bmatrix} \omega_x \\ \omega_y \\ \omega_z \\ v_x \\ v_y \\ v_z \end{bmatrix} \qquad \hat{a} = \begin{bmatrix} \alpha_x \\ \alpha_y \\ \alpha_z \\ a_x \\ a_y \\ a_z \end{bmatrix} \qquad \hat{f} = \begin{bmatrix} \tau_x \\ \tau_y \\ \tau_z \\ f_x \\ f_y \\ f_z \end{bmatrix}$$

As we saw in last slide, the expressions are getting very long and tedious to deal with. In order to shorten those expressions we can use spatial algebra notation. There is not a single standard convention, and there are subtle differences between authors. In fact, the spatial algebra notation by Featherstone in his earlier work is slightly different in newer work. Here we use the notation as presented in Brian Mirtich's PhD thesis.

Spatial transform

$$_{to}\hat{X}_{from} = \begin{bmatrix} 1 & 0 \\ -\hat{r} & 1 \end{bmatrix}\begin{bmatrix} R & 0 \\ 0 & R \end{bmatrix} = \begin{bmatrix} R & 0 \\ -\hat{r}R & R \end{bmatrix}$$

$$\hat{v}_g = {}_g\hat{X}_f\hat{v}_f$$

Spatial acceleration

$$\alpha_i = \alpha_{i-1} + \dot{\omega}_i$$

$$a_i = a_{i-1} + \alpha_{i-1} \times r_i + \omega_{i-1} \times (\omega_{i-1} \times r_i) + \omega_{i-1} \times v_{rel}$$

$$+ \omega_{i-1} \times ((q'_i u_i) \times d_i) + (\ddot{q}_i u_i) \times d_i + (q'_i u_i) \times ((q'_i u_i) \times d_i))$$

$$\hat{a}_i = {}_i\hat{X}_{i-1}\hat{a}_{i-1} + \hat{c}_i + \ddot{q}_i \hat{s}_i$$

$$\hat{s}_{revolute} = \begin{bmatrix} u_i \\ u_i \times d_i \end{bmatrix}$$

$$\hat{s}_{prismatic} = \begin{bmatrix} 0 \\ u_i \end{bmatrix}$$

Using spatial vector notation the propagation of acceleration looks much simpler. The spatial acceleration 'a' of link i is adds the spatial coriolis acceleration 'c' and the joint acceleration q''s. Note that the spatial joins axis 's' differs for each joint type. The dimension of s is the number of degrees of freedom x 6.

Articulated Equation of Motion

$$\hat{f}_i = \hat{M}_i \hat{a}_i + \hat{f}_i^z$$

$$\ddot{q} = \frac{\hat{s}_i \hat{f}_i - \hat{s} \hat{I} \hat{a}_{i-1} - \hat{s}_i (\hat{f}_i^z + \hat{I}_i \hat{c}_i)}{\hat{s}_i \hat{I}_i \hat{s}_i}$$

$$\hat{a}_i = \hat{a}_{i-1} + \hat{c}_i + \ddot{q}_i \hat{s}_i$$

The equation of motion for articulated bodies is very similar to the usual way in maximal coordinates. Note that the mass matrix M is usually called I in the context of articulated bodies.

Init Spatial Inertia and ZA forces

```
F_za[0] = [- m_baseTorque,- m_baseForce];
I[0] = base_inertia;//this is a 6x6 matrix
// Initialize the isolated spatial inertia and za force for each link
for (int i = 1; i <numLinks; i++)
{
    F_za[i]=[-rot_from_world[i]*m_links[i].m_appliedTorque,
             rot_from_world[i]*m_links[i].m_appliedForce];
    I[i] = m_links[i].m_inertia;
}
for (int i = numLinks-1; i >=1; i--)
{
    int parent = m_links[i].m_parent;

    I[parent] += pXi * (I[i] - h[i] h[i]' / D[i]) * iXp
    F_za[parent] += pXi * (F_za[i] + I[i]*c[i] + h[i]*Y[i]/D[i])
}
```

The initialization of the isolated spatial inertia and za forces is straightforward.

Compute Link Accelerations

$$\ddot{q} = \frac{\hat{s}_i \hat{f}_i - \hat{s} \hat{I} \hat{a}_{i-1} - \hat{s}_i (\hat{f}_i^z + \hat{I}_i \hat{c}_i)}{\hat{s}_i \hat{I}_i \hat{s}_i}$$

$$\hat{a}_i = \hat{a}_{i-1} + \hat{c}_i + \ddot{q}_i \hat{s}_i$$

```
• accel[0] = I_inverse[0]*F_za[0];

for (int i = 1; i <numLinks; i++)
{
        int parent = m_links[i].m_parent;
        SpatialTransform(rot_from_parent[i], m_links[i].m_cachedRVector,
                        accel_top[parent], accel_bottom[parent],
                        accel_top[i], accel_bottom[i]);
                joint_accel[i] = (Y[i] - SpatialDotProduct(h_top[i],
    h_bottom[i], accel_top[i+1], accel_bottom[i+1])) / D[i];
                accel_top[i] += coriolis_angular[i]
    + joint_accel[i] * m_links[i].getAxisTop(0);
        accel_bottom[i] += coriolis_linear[i] +
     • joint_accel[i] * m_links[i].getAxisBottom(0);

    • }
```

Todo: cleanup this code and explain a bit

Articulated Body Algorithm

1. Propagate Link Velocities and isolated terms

$$\omega_i = \omega_{i-1} + \omega_{rel}$$

$$v_i = v_{i-1} + w_{i-1} \times r_i + v_{rel}$$

1. Propage Spatial Inertia and ZA forces

$$I_p = I_p X_{p2i} [I_i - \frac{I_i s_i s_i I_i}{s_i I_i s_i}] X_{p2i}$$

$$F_p^{za} = \ldots$$

1. Compute Link Accelerations

$$\ddot{q} = \frac{\hat{s}_i \hat{f}_i - \hat{s} \hat{I} \hat{a}_{i-1} - \hat{s}_i (\hat{f}_i^z + \hat{I}_i \hat{c}_i)}{\hat{s}_i \hat{I}_i \hat{s}_i}$$

$$\hat{a}_i = \hat{a}_{i-1} + \hat{c}_i + \ddot{q}_i \hat{s}_i$$

By combining the propagation methods to compute link velocities, spatial articulated inertias, bias forces and acceleration computation we have Featherstone's Articulated Body Algorithm in its basic form. From here we can add support for contact and other constraints and multi-DOF joints.

How to improve Sequential Impulse

- Overview of Rigid Body Simulation Loop

- Add improved MLCP block solvers
  - Moving from SI to generic PGS MLCP solver
  - Try out NNCG, Dantzig, Lemke MLCP solvers

- Add O(n) solver for tree/chain equality joints
  - Featherstone Articulated Body Algorithm
  - Mixing ABA with Sequential Impulse
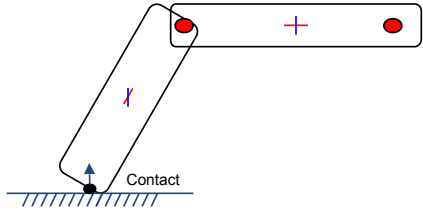
Featherstone Joints versus Constraints
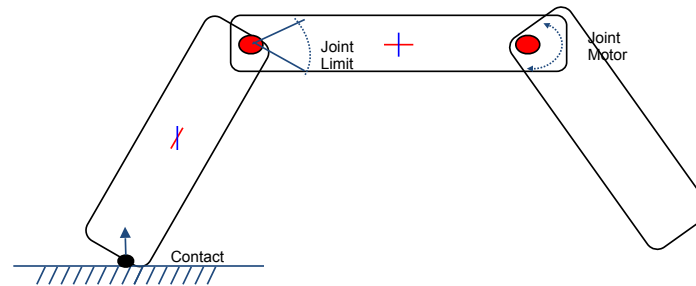
The Articulated Body Method deals with the forward dynamics of chain or tree structures. The ABM cannot deal with closed loops, contacts, joint limits or motors by itself. It is possible to add such constraints using an MLCP solver or Sequential Impulse.

Featherstone with Contact Constraints

What is the effect of an impulse on the velocity?

$$A = J \, M^{-1} \, J^T$$

$$= n \, K \, n^T$$

$$= (\Delta u_{ab}) dot(n)$$

Contact

We can use any of the three methods to build the A matrix and solve the MLCP to add support for contact constraints with Featherstone multibodies. In Brian Mirtich's PhD thesis the K matrix is computed for multibodies. In Jakub Stępień's PhD thesis the big matrix approach is used using the Composite Rigid Body Algorithm. As I mentioned before, for some smaller problems such as ragdolls and vehicles the O(n^3) CRBA approach can be faster than the O(n) ABA.

Here we will measure the effect of an impulse by re-using the Articulated Body Method forward dynamics algorithms. This is similar to the description by Vangelis Kokkevis in his 'articulated characters' GDC 2004 talk, although Vangelis used forces and accelerations, while we use impulses and velocities.

Effect of an impulse on velocity

- We could use an unmodified ABM algorithm

$$a_{before} = ABM(f_0)$$
$$a_{after} = ABM(f_1)$$
$$a = a_{after} - a_{before}$$
$$\Delta v = a\Delta t$$

A

B
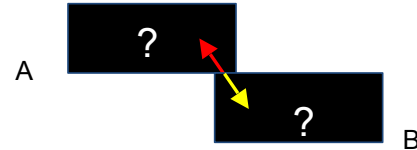
We could use a forward dynamics algorithm, such as the Articulated Body Method and measure the effect of a unit force on the acceleration. The acceleration before applying the force might not be zero, so we need to measure the acceleration when we apply zero force (f_0). Then we can measure the acceleration after applying a unit force and the acceleration due to the unit force is the difference. If we assume the delta time is one, we get to the delta velocity by taking this acceleration difference.

Simplified ABA

Spatial Z.A. impulse

Test Impulse
(Unit Normal)

$$\Delta Y_p = \hat{X}\left[1 - \frac{Iss'}{sIs}\right]\hat{Y}_i$$

$$\Delta \hat{q}_i = -\frac{\hat{s}_i}{\hat{s}_i \hat{I}_i \hat{s}_i}\left[\hat{I}_i \hat{X} \Delta \hat{v}_p + \hat{Y}_i\right]$$

$$\Delta \hat{v}_i = \hat{X}\Delta \hat{v}_p + \Delta \dot{q}_i \hat{s}_i$$
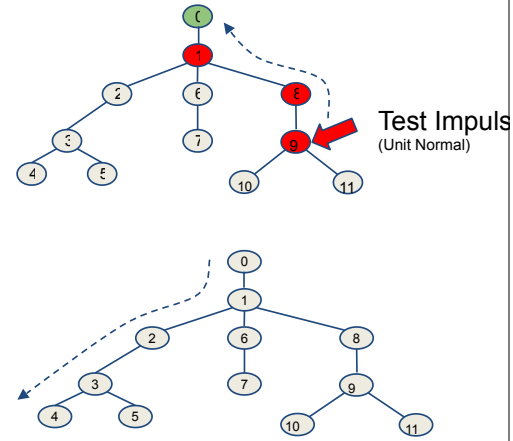
Although calling the unmodified ABM twice will work, we can do better. When applying an impulse, the Coriolis and gravity forces can be ignored because the time duration is nearly zero so the effect of forces approaches zero. This means that the ABM can be simplified. In addition, we don't need to call the ABM twice using this formulation. You can find the derivation and the pseudo code in Brian Mirtich's PhD thesis around page 141.

Multibody Contact Jacobian

Test Impulse
(Unit Normal)
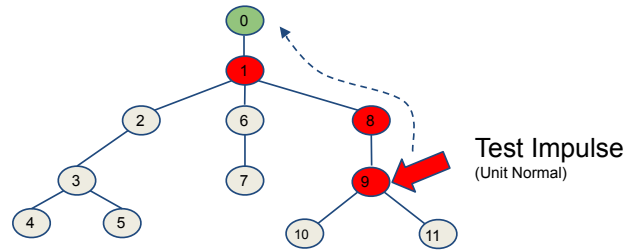
6-DOF Base

Multibody Contact Jacobian

```cpp
for (int i = 0; i < num_links; ++i)
{
  //compute world to local transform matrix
  const int parent = m_links[i].m_parent;
  const btMatrix3x3 mtx(m_links[i].m_cachedRotParentToThis);
  rot_from_world[i+1] = mtx * rot_from_world[parent+1];

  // transform normal and relative position p to local frame
  n_local[i+1] = mtx * n_local[parent+1];
  p_minus_com[i+1] = mtx * p_minus_com[parent+1] - m_links[i].m_cachedRVector;

  // calculate the jacobian entry
  if (m_links[i].m_jointType == btMultibodyLink::ePrismatic) {
    results[i] = n_local[i+1].dot( m_links[i].getAxisBottom(0) );
  }
}
//store jacobian for this link and traverse branch to base
while (link != -1)
{
  jac[6 + link] = results[link];
  link = m_links[link].m_parent;
}
```

Adapting SI/PGS for multi body?

```
for (int k = 0; k <numIterations; k++)
{
  for (i = 0; i <numRows; i++)
  {
    delta =c.m_rhs- c.m_appliedImpulse;
    dV1Dotn=      c.m_n.dot(body1.m_deltaV)+
c.m_relpos1XNormal.dot(body1.m_angVel);
    dV2Dotn=c.m_n.dot(body2.m_deltaV)+
c.m_relpos2XNormal.dot(body2.m_angVel);

    delta-= dV1Dotn*c.m_jacDiagABInv;
    delta-=dV2Dotn*c.m_jacDiagABInv;

    sum =c.m_appliedImp +delta;

    bodyA->applyImpulse(n*delta);
    bodyB->applyImpulse(-n*delta);
  }
}
```

```
for (int k = 0; k <numIterations; k++)
{
  for (i = 0; i <numRows; i++)
  {
    delta = 0.0f;
    {
      for (j = 0; j <i; j++)
        delta += A(i,j) * x[j];
      for (j = i+1; j<numRows; j++)
        delta += A(i,j) * x[j];
    }
    x [i] = (b [i] - delta) / A(i,i);
  }
}
```

Lets have a look at SI/PGS again and look at the parts that need some change. The elements on the main diagonal of A, colored in blue, will need to take into account the multibody. In addition, the dot product of the velocity on the jacobian (normal) needs some change.

In case we would use PGS, the changes just happen in building the A matrix. For a regular 3D rigid body, the size of the Jacobian is 6, the number of degrees of freedom. In the case of a multibody, the size of the Jacobian is the total degrees of freedom of the Multibody.

In case of when we use Sequential Impulse, the 'applyImpulse' method, here in green, will have to apply the impulse on the multibody and propagate it through all links. And finally the right hand side, or b, needs to take into account the multi body as well.

Multibody SI

```
for (int k = 0; k <numIterations; k++)
{
  for (i = 0; i <numRows; i++)
  {
    delta =c.m_rhs- c.m_appliedImpulse;
//        dV1Dotn= c.m_n.dot(body1.m_deltaV)+ c.m_relpos1XNormal.dot(body1.m_angVel);
    for (int i = 0; i < ndofA; ++i)
      dV1Dotn += jacobians[c.m_jacAindex+i] * deltaV[c.m_dVelAindex+i];

    dV2Dotn=c.m_n.dot(body2.m_deltaV)+ c.m_relpos2XNormal.dot(body2.m_angVel);

    delta-= dV1Dotn*c.m_jacDiagABInv;
    delta-=dV2Dotn*c.m_jacDiagABInv;

    sum =c.m_appliedImp +delta;

    bodyA->applyImpulse(n*delta);
    bodyB->applyImpulse(-n*delta);
  }
}
```

The main change in the innerloop of Sequential Impulse for a Featherstone multibody is that we have a dot that has the full width of the degrees of freedom of the multi body,
instead of 6 (3 linear and 3 angular) for a regular rigid body. The two lines in red show the new version, with the original version commented out in green.

Constraints for Joint Limits, Motors

Joint Limit

Joint Motor
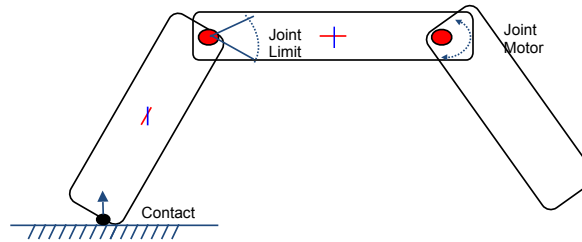
Contact

Once we have a way to relate the applied impulse to the change in velocity, or effective mass, we can use the same mechanism we used for contact constraints also to simulate joint limits and joint motors using constraints.
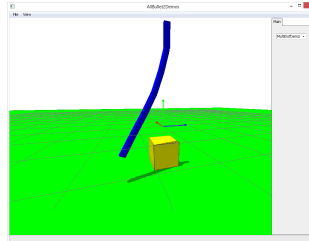
Featherstone ABA Demos
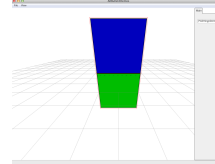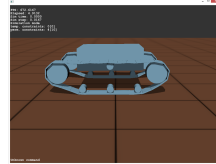
Show a few demos that demonstrate some issues with sequential impulse (SI)/projected gauss seidel (PGS). We will re-use the same demos while we explore other solvers.

1) Stack heavy on light objects
2) Door hinge hitting heavy object
3) Ragdoll joint gap
4) Forklift joint gap
5) Heavy tank on tracks

3DOF Spherical Joint

We can simulate a multi degree of freedom joint such as a 3-DOF spherical joint by stacking several 1-DOF joints together. Similar to using Euler angles we can run into singularities, causing instability. One solution is extending the system to support multi-DOF joints. @todo explain basic changes that have to be made going from 1-DOF to multi-DOF and point to references with mode details.
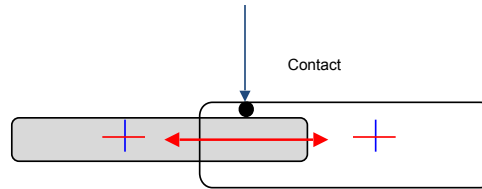
Once I had the simulator up and running, for a few days I experienced instabilities, 'explosions' etc.
So I went through the implementation, verified values, added damping and joint friction (a motor with target velocity zero),
but still the instabilities sometimes happened.

It turned out that the contact normal between to links connected by a prismatic joint,
was pointing in an orthogonal direction to the joint axis. This means that no matter how much impulse you apply,
you will not reduce the constraint error along the prismatic axis, when applying an impulse in the contact normal direction.

The collision detection needs to be aware of the joint axis. When using the SAT to determine the separating axis, we can only use the prismatic axis for
example. In addition, we can project the separating vector onto the joint axis,
to make sure to only add valid components to the Baumgarte stabilization.

Challenges with Featherstone
and some workarounds

- Stability, energy gain
  - damping, clamping, implicit integrator

- Collision Performance issue
  - O(m * n) for m constraints and n joint dofs

- API to setup a multibody needs more work
  - Simbody has auto converter

Semi implicit (or semi explicit) Euler integrator in combination with a fixed time step and the non-linearity of the Coriolis/velocity dependent forces can cause a feedback loop between those forces and angular acceleration. This can quickly add a lot of kinetic energy. High-quality simulators often use a variable time step in combination with sophisticated methods to measure the error and deal with it.
For real-time applications using a fixed time step, we can avoid huge kinetic energy gain by using an implicit integrator for the velocity-dependent forces. Alternatively, we can clamp the maximum angular velocity and apply enough velocity damping to suppress the unwanted effect.

Another drawback of reduced coordinate methods is that each contact constraint requires O(n) time to evaluate the Jacobian and effective mass, and application of the applied impulse. In some cases, such as a ragdoll lying on the floor, each limb (link) has one of more contact constraints with the floor, resulting in a O(n^n) performance, rather than O(n).
Some caching of the computation could bring it back to O(n) on average, but there could be still a spike if all those contact constraint caches become invalid at the same time.

Conclusion

- Featherstone ABA can be easily mixed with Sequential Impulse and is useful for games

- Big matrix direct MLCP solvers can be useful as a block solver within a PGS/SI loop

- It is fun and rewarding to explore better constraint solving methods

Thank you!

References
- Physics-Based Animation of Articulated Rigid Body Systems for Virtual Environments, Jakub Stępień, PhD thesis, 2013
- Robotics and Multibody Dynamics, 2010, Abhinandan Jain
- Rigid Body Dynamics Algorithms, 2008, Roy Featherstone
- Practical Physics for Articulated Characters, GDC 2004, Vangelis Kokkevis
- Impulse-based Dynamic Simulation of Rigid Body Systems, Brian Mirtich, PhD thesis, 1996

Some C++ Software that include reduced coordinate multibody methods
- Bullet Physics, http://bulletphysics.org, http://github.com/bulletphysics
- RBDL, http://bitbucket.org/rbdl
- SimBody, https://github.com/simbody
- Moby, https://github.com/edrumwri/Moby
- Dart (was RTQL8), https://github.com/dartsim/dart

Extra Slides

Implicit Integration Coriolis Forces

$$L = I\omega$$

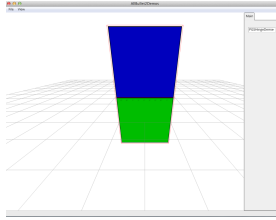$$\tau = \frac{dL}{dt} = I\dot{\omega} + \omega \times I\omega$$

$$\dot{\omega} \approx (\omega_2 - \omega_1)/h$$

$$I(\omega_2 - \omega_1)/h + \omega_2 \times O\omega_2 = 0$$

$$I_{local}$$

$$q' = 0.5 * q * \omega_{local}$$

Removing all degrees of freedom
using zero inverse mass/inertia

By default, a 3D rigid body with positive mass and inertia has 6 degrees of freedom. If we want to create a static object we could create a 3D rigid body with positive mass and inertia and lock all degrees of freedom using a fixed constraint. A rigid body locked using a fixed constraint that is solved using projected Gauss Seidel will show similar issues to the hinge stress test in previous slide. Instead of using a fixed constraint to reduce all degrees of freedom for a rigid body, we usually set the inverse mass to zero. This effectively disables the motion for the object and makes its mass 'infinitely large'. There are a few benefits using this approach: there are no convergence issues and there are no constraints to be solved.

Creating a hinge
using zero inverse mass/inertia
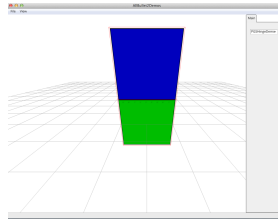
(video demo of Hinge Stress Test by setting 5 components of the inverse mass/inertia to zero)

In a similar way of creating a fixed rigid body, we can also create a hinged rigid body with 5 degrees of freedom removed by setting the related components of inverse mass and inverse inertia tensor to zero. This can be useful to simulate hinged objects such as doors. A benefit is that the door will not have any error in the hinge axis, no matter how large the mass of objects colliding against the door. Aside from a hinge, we can use the same mechanism to reduce the degrees of freedom of bodies to emulate 2D planar objects. Obviously this is not a generic solution, but it can be useful in some cases.

Another way of looking at this is: by setting components of the inverse mass and/or inverse local inertia to zero, we can control the effective mass experienced at a point on the object in a certain direction. We can create a similar effect by using Featherstone, as we will see soon. When interacting with a hierarchical body simulated using the Featherstone Articulated Body Method (ABM) or Articulated Body Algorithm (ABA), we can experience a similar effect: in some directions the effective inverse mass or inverse inertia can be zero.

Chain: SI/PGS Residual at 1:1
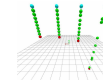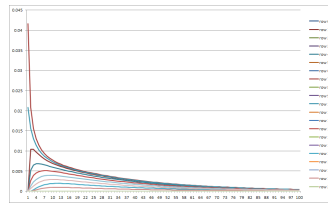
When solving a constraint row, the SI/PGS solver computes the change in accumulated impulse, needed to drive the error for this constraint row to zero. This change, or delta impulse, can be used as a measure of current constraint error, or residual. Here is a plot for the constraint rows.

Chain: SI/PGS at 1:1000, 240Hz
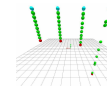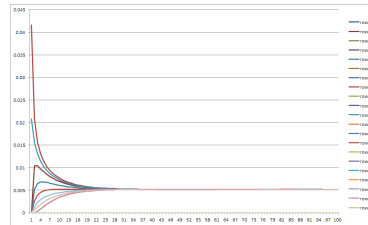
Here is a plot for the constraint rows with a mass ratio 1:1000 at a higher frequency. The constraint error doesn't converge either, but due to the smaller timestep, the error is much smaller, around 0.005. The error is basically related to the impulse that the gravity applies in one timestep.

Chain: NNCG Residual at 1:1
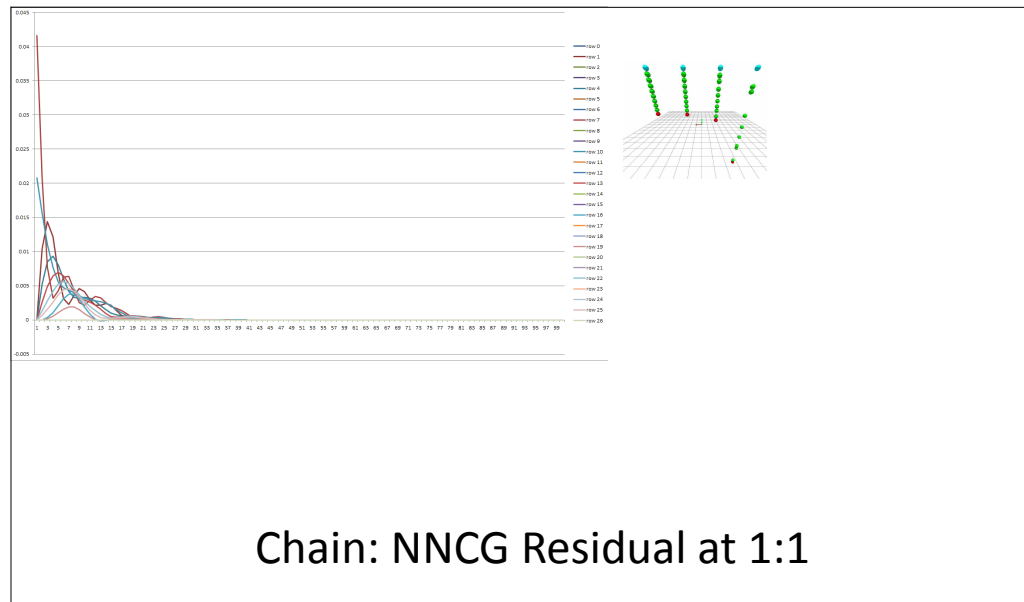
Looking at the curves of the residual, the remaining constraint error at each iteration, we see a similar convergence to PGS. In this test we don't see the same 'spikes' in the residual as mentioned in the NNCG paper: at some stage the Conjugate Gradient algorithm can get stuck in a local minimum, and we need to restart the search with a different start direction.

Chain: NNCG at 1:1000, 60Hz
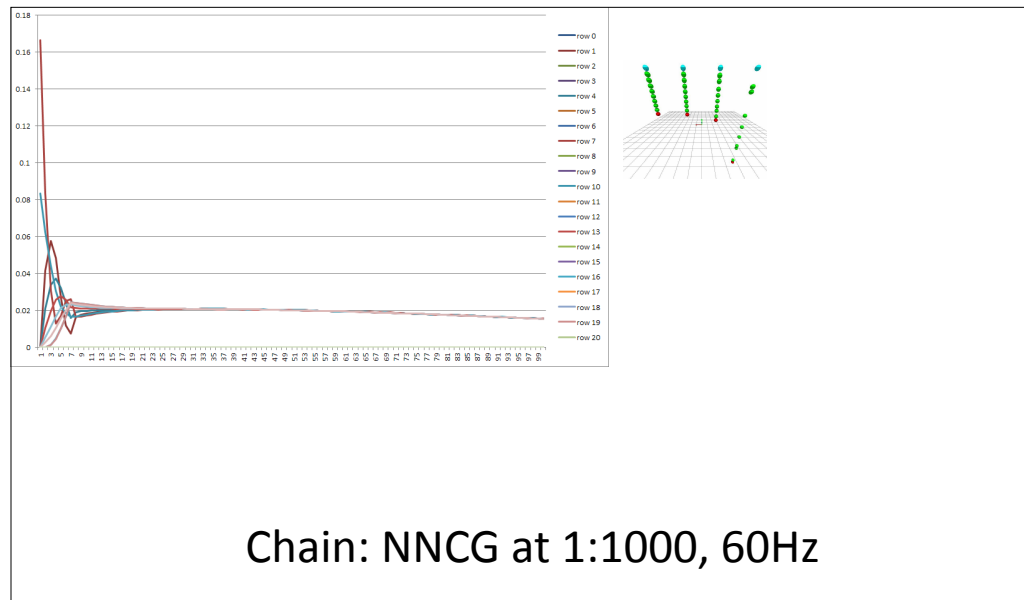
Here is a plot for the constraint rows with a mass ratio 1:1000 at 60 hertz. The graph looks similar to PGS, although at around 80 iterations the curve seems to start converging.