Havok Constraints

*This **User Guide** describes the constraint system in Havok.*

Contents

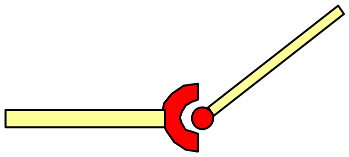
1. Introduction

Constraints are essentially restrictions on the freedom of movement of objects. They may be artificial, such as restricting an entity to have a velocity given by some complex function of the system, or may result from the inclusion of a connecting joint, such as a hinge, between several entities. Whatever the type of constraint, it is very difficult to create dynamic physical environments without these elements.

The following sections describe the different types of constraint that are supported by Havok. The creation of constraints and the tweaking of their strengths and limits is discussed. Finally, some guidelines are provided for the use and limitations of constraints.

2. Constraints and Dashpots

Constraints and dashpots restrict the movement of objects. They may restrict the position/orientation of a single body in world space or they may restrict the relative position/orientation of pairs of bodies:



Example of a ball socket constraint (red)
constraining to boxes (yellow)
(another name is point to point constraint)

Havok Hardcore provides support for constraints of varying degrees of complexity and accuracy:

Constraints:	Constraints are solved in groups, each managed by a FastConstraintSolver. The FastConstraintSolver calculates the forces needed to satisfy all its constraints. Based on the objects masses and the simulation frequency the FastConstraintSolver calculates the maximum allowed forces automatically. As a result constraints are relatively stable and pretty stiff, however constraints get slightly softer with lower simulation frequency or low object masses.
Dashpots:	Dashpots are a simpler version of constraints. The user has to specify the final forces and parameters initially. As a result, the stability of a dashpot is really sensitive to the simulation frequency and user parameters. Dashpots are standalone actions, resulting in a relatively springy behavior.

Types of constraints:

In rigid body dynamics each body has 6 degrees of freedom:

- 3 translation axis
- 3 rotation axis

As a consequence each constraints can limit one or more degrees of freedom. Depending on the number and type of these degrees of freedoms removed, we get different basic types of constraint like point-to-point constraint or rag doll constraint.

2.1. Notes on Dashpots

A **Dashpot** is like a heavily damped spring. It applies forces to objects when the velocities of their attached points begin to differ. Dashpots can be made much stiffer than regular springs because velocities are taken into account. Dashpots have two parameters that you can set:

Strength	The strength of the restoring force between the two points. A good initial range is 0.5 - 1 of the mass of the attached objects.
Damping	Low values create spring-like behavior, while high create a smoother behavior. A good initial value is 0.1 of the strength.

Strong forces (such as a Mouse Spring) or impulses may momentarily break this constraint, but the bodies will automatically snap back to reinstate it.

2.2. Notes on Constraint

Constraints differ from dashpots (in terms of simulation) in one important respect. They are treated as components of entire constraint systems that must be solved simultaneously. This means that they are far more accurately maintained. While dashpots above may be stretched quite freely, constraints are relatively stiff.

Tau	Large impulses or forces can stretch all constraints. The speed of recovery in physical time step is determined by the constraint's **tau** value. Higher values recover the constraint more quickly. Therefore the tau value influences the strength of the restoring force between the two points. Values range from 0.0f to 1.0f. A good choice is 1/sqrt(numberOfConstraintsInSystem)
Damping (Strength)	A factor how strong the velocities of the objects is taken into account. Normally a value of 1.0 is perfect, only when very many objects are constraint together, this parameter might be set to slightly lower values to archive higher stability. **Note: In the Havok SDK 1.6 the Damping parameter is actually named "Strength". This inconsistency between documentation and SDK will be fixed till the next release.**

Creating constraints between two bodies, one of which is fixed in space, effectively constrains the unfixed body to a point.

To use the above constraints you must first create a constraint solver to which you will add the constraints individually as you create them. For example:

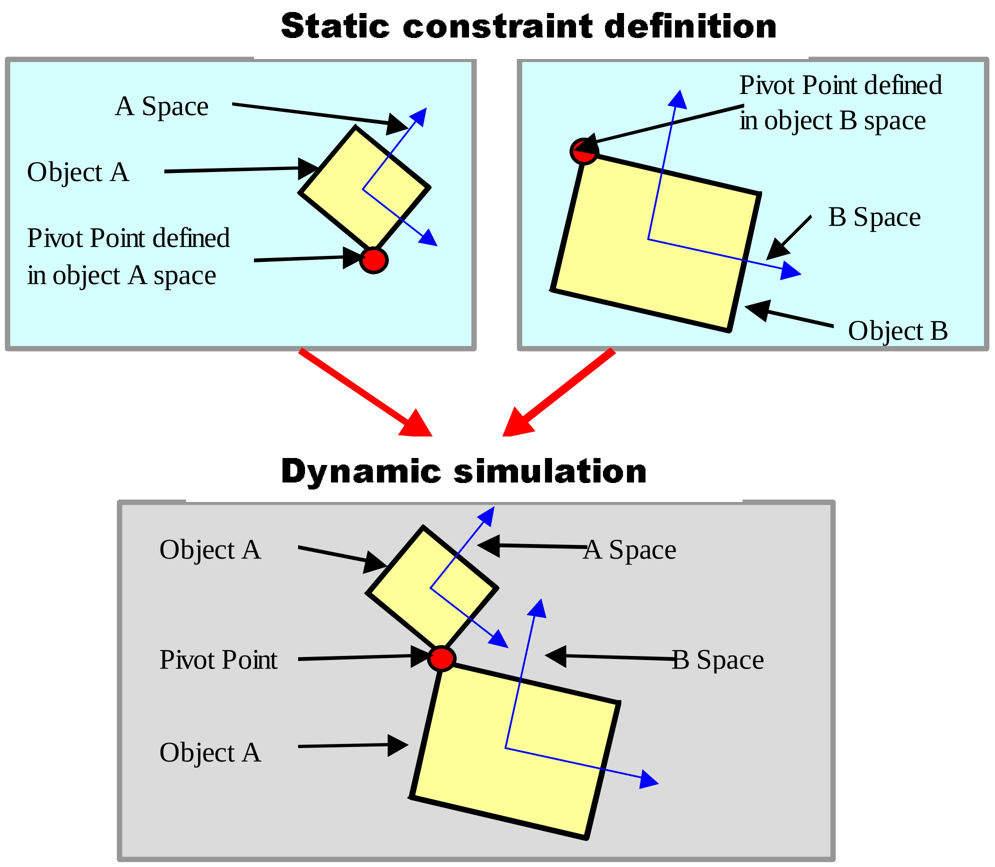
```
FastConstraintSolver* solver = new FastConstraintSolver();

toolkit->m_defaultRigidCollection->addAction(solver);
```

Now we can add constraints to this solver using the method addConstraint() and during simulation they will be maintained by this solver.

Note: Once added to a solver, if any parameters of the constraint are updated during simulation (by direct access to the constraint's member variables) a call to Constraint::applyChanges() is necessary to effect this update properly.

2.3. Point to point constraints

A point-to-point constraint (another name: ball-socket constraints) forces objects to try to share a common point in space. The objects can move freely about this space, but always have this point in common. This point is called pivot point:



Static constraint definition
A Space
Object A
Pivot Point defined in object A space
Pivot Point defined in object B space
B Space
Object B
Dynamic simulation
Object A
A Space
Pivot Point
B Space
Object A

At creation of the constraint, the pivot point has to be defined in the object space of each object involved. During runtime the constraint tries to apply forces to the object, so that the two pivot points defined by the two objects match.

For convenience, some constraints allow you to define the pivot point in world space. In this case the constraint transforms this position to the local space of both objects. Note: Setting this pivot point using world space only works if both objects are placed at a position where the constraint is already satisfied.

The following examples shows how to add two bodies to a scene and how to create different types of constraints connecting the top right corner of body1 to the bottom left corner of body2.
When the simulation is run the bodies spring together in order to try and share this common point in space, and try and maintain this common point throughout simulation.

2.3.1. Linear Dashpot

```
#include <hkdynamics/action/dashpot.h>

RigidBody* body1 = … // get my body at position Vector3(1,2,3)
RigidBody* body2 = … // get my body at position Vector3(4,5,6)

Dashpot *d = new Dashpot(
             body1,
             Vector3(.6,.6,.6),      // pivot point defined in body 1 space
             body2,
             Vector3(-.6,-.6,-.6));  // pivot point defined in body 2 space

toolkit->m_defaultRigidCollection->addAction(d);
```

2.3.2. StrongJoint

The strong joint is an old implementation of a point-to-point dashpot.

```
#include <hkdynamics/action/strongjoint.h>

RigidBody* body1 = … // get my body at position Vector3(1,2,3)
RigidBody* body2 = … // get my body at position Vector3(4,5,6)

StrongJoint *d = new StrongJoint(
             body1,
             Vector3(.6,.6,.6), // pivot point defined in body 1 space
             body2,
             Vector3(-.6,-.6,-.6));  // pivot point define in body 2 space

toolkit->m defaultRigidCollection->addAction(d);
```

2.3.3. PointToPointConstraint

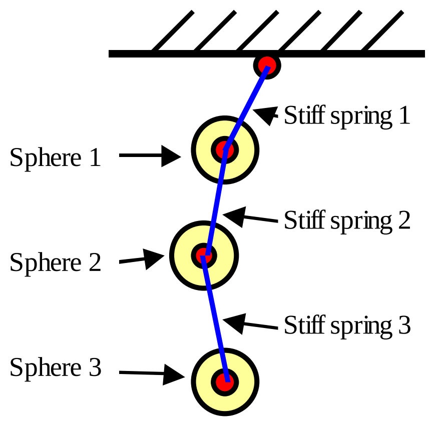
```
#include <hkdynamics/action/pointtopointconstraint.h>

FastConstraintSolver* solver = new FastConstraintSolver ();
toolkit->m_defaultRigidCollection->addAction(solver);


RigidBody* body1 = … // get my body at position Vector3(0,0,0)
RigidBody* body2 = … // get my body at position Vector3(2,2,2)

    // Ignore collisions for now
toolkit->m_defaultCollisionDetector->disableCollision(body1, body2);

    // Create constraint
PointToPointConstraint* c = new PointToPointConstraint(body1, body2);

if (1)
{   // use world space helper function
c->setPivotWorldSpace(Vector3(1, 1 ,1));
}
else
{   // use low level parameters
    c-> m_points_os[0] = Vector3(  1, 1, 1 ); // pivot point in body1 space
    c-> m_points_os[1] = Vector3( -1,-1,-1 ); // pivot point in body2 space
}
solver->addConstraint(c);
```

Note: You can use a series of such constraints between adjacent bodies to form a chain.

2.4. StiffSpring

The stiff spring constraint is similar to the point-to-point constraint except for one important detail: It holds the constrained bodies apart at a specified distance. As with the point-to-point constraint, the stiff spring operates from a point within each body. The bodies are free to move and rotate around the point, but the points are fixed at a constant relative distance to each other.

So chains of objects can easily created using stiff springs:



In the following example, we constrain two bodies to have a “stiff-spring” joining their centers. The constraint is fully specified by a point in the body space of each of two bodies, and a constant distance by which these two points must be kept separated. The default body space point is the origin, which we use below, and since these points are 4 units apart initially, we set the length of the StiffSpring to be 4 as well.

Note: do not use stiff springs with small lengths use point-to-point constraints instead.

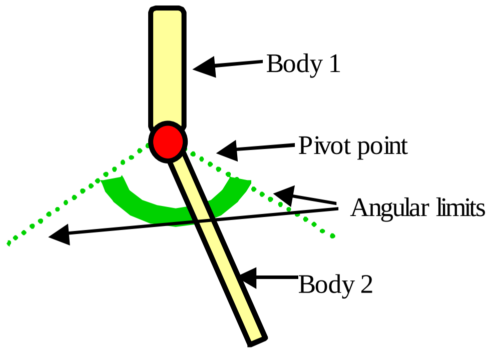
```
#include <hkdynamics/action/stiffspringconstraint.h>

// Create two bars
RigidBody* box1 = … // get a box at position (-2,0,0) and size (4,1,1)
RigidBody* box2 = … // get a box at position ( 2,0,0) and size (4,1,1

// Ignore collisions for now
toolkit->m_defaultCollisionDetector->disableCollision(box1, box2);

// Create constraint
StiffSpringConstraint* c = new StiffSpringConstraint(box1, box2);
c->setLength(4);

// Add to constraint solver
solver->addConstraint(c);
```

Note: Stiff springs can be used to replace normal springs: Just set tau to very low values (0.001 to 0.5) to archive the desired spring constant and the dampening accordingly (rule: damping ~ [0.5 * tau … 1.0]. Stiff springs are more stable than real springs, especially when used with the Euler integrator.

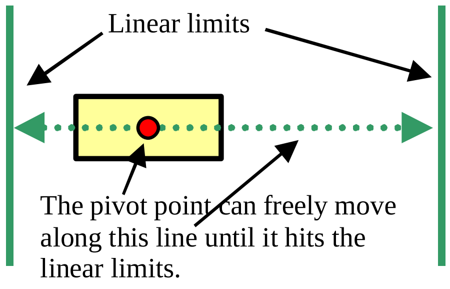
2.5. Constraint Limits and Constraint Friction

For each non-restricted axis of some constraints, you can use:

Limits	to limit the movement to be in a certain range Here is an example of an angular limit:



And another example of a linear limit:


Linear limits
The pivot point can freely move along this line until it hits the linear limits.

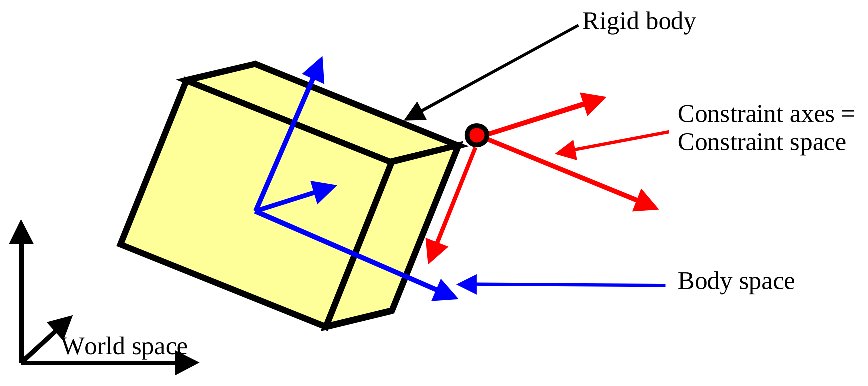
Friction	to simulate friction. The forces/torques a constraint can apply without sliding are defined in [N] / [N/m].
Motor	to create a motor like behavior. Just specify the angular velocity and the maximum force/torque. (The implementation of the motor is using the friction algorithm with a small modification: the resting position of the constraints simply gets a velocity).

Note: only use the motor for small angular velocities, not for fast vehicles.

2.6. Angular Constraints Introduction

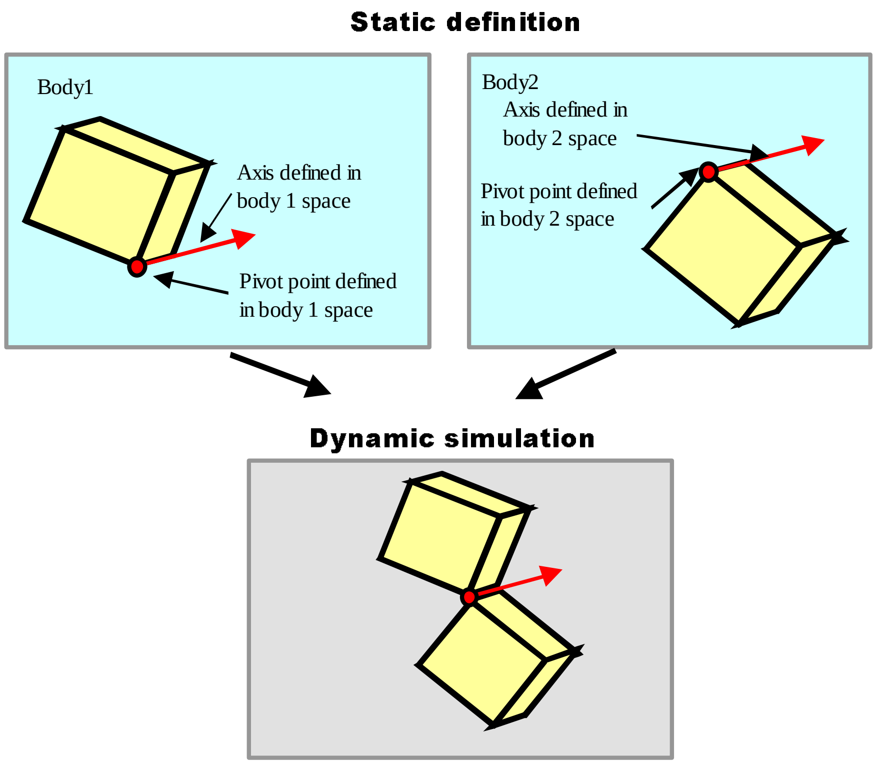
An angular constraint restricts the relative angular movement of two objects.

Similar to the pivot point for point-to-point constraints, you have to define one to three axes per object in the corresponding object space:



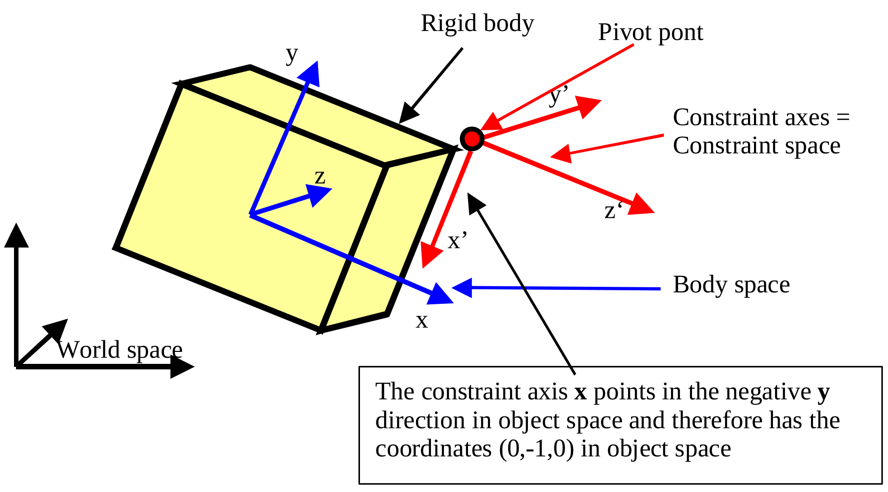
Hinge example:

The hinge constraint tries to force two objects to share a common pivot point and an axis in world space:



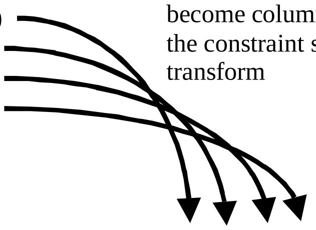
2.7. *Constraint Matrix: m_transform_os_ks*

For some constraints three axes and the pivot point are required. Three axes and a point actually do define a coordinate system, the constraint space **(ks)**. You can store them in 4x4 **transformation matrix** by storing the first three axis as the first three **columns** and the pivot point as the forth column. Doing this, you get a matrix which transforms points from constraint space to object space (**m_transform_os_ks**).



The constraint axis **x** points in the negative **y** direction in object space and therefore has the coordinates (0,-1,0) in object space

Constraint axis **X'** in object space: **Vector3(0, -1, 0)**
Constraint axis **Y'** in object space: **Vector3(0, 0, 1)**
Constraint axis **Z'** in object space: **Vector3(1, 0, 0)**
Pivot point in object space: **Vector3(1,1,1)**


Constraint axis become columns in the constraint space transform

m_transform_os_ks

$$\begin{matrix} 0 & 0 & 1 & 1 \\ -1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{matrix}$$

Given the constraint axes in world space (**m_transform_ws_ks**) and the body transform (**m_transform_ws_os**), you can calculate the local constraint matrix by:

```
Transform m_transform_os_ws = m_transform_ws_os.getInverse();
Transform m_transform_os_ks = m_transform_os_ws * m_transform_ws_ks;
```

2.8. Angular Dashpots

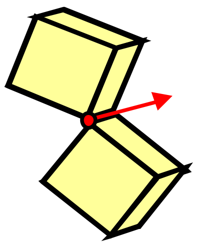
An **Angular Dashpot** is the rotational equivalent of a linear dashpot. An angular dashpot tries to align two objects so that they have the same orientation (with an optional offset).

```
#include <hkdynamics/action/angulardashpot.h>

RigidBody* box1 = … // get a box at position ( 1,2,3)
RigidBody* box2 = … // get a box at position ( 4,5,6)

//optional angular offset can be passed into as parameter 3 here
AngularDashpot* d = new AngularDashpot(box1, box2);

toolkit->m defaultRigidCollection->addAction(d);
```

The optional angular offset is a quaternion defining the relative orientation of the second body relative to the first.

2.9. HingeConstraint

This constraint allows you to simulate a hinge-like action between two bodies. You specify a line segment in body space for each body, with a position and a direction. The two lines then attempt to match position and direction, thereby creating an axis around which the two bodies can rotate. Hinges also have limit and friction parameters. Limits specify the maximum angle to which hinged bodies can rotate, which is useful for preventing interpenetration. Friction allows you to specify the velocity toward which the rotating bodies tend. Usually, friction has a value of zero, however specifying a non-zero value allows you to build cogs or similarly styled motors using the hinge constraint:



In the following example, we constrain two slabs to be hinged around their end points. We specify both the common axis of rotation (here aligned with the Z-axis), and a point through which this.

```
#include <hkdynamics/action/hingeconstraint.h>

RigidBody* box1 = … // get a box at position (-2, 0, 0)  size (4,1,2)
RigidBody* box2 = … // get a box at position ( 2, 0, 0)  size (4,1,2)

// Ignore collisions for now
toolkit->m defaultCollisionDetector->disableCollision(box1, box2);

// Create constraint
HingeConstraint* c = new HingeConstraint(box1, box2);
Vector3 midpoint = 0.5f * (box1->getPosition()+box2->getPosition());

// Specify rotation allowed through common point at ends and around Z-axis
c->setAxisWorldSpace(midpoint, Vector3(0,0,1));
solver->addConstraint(c);
```

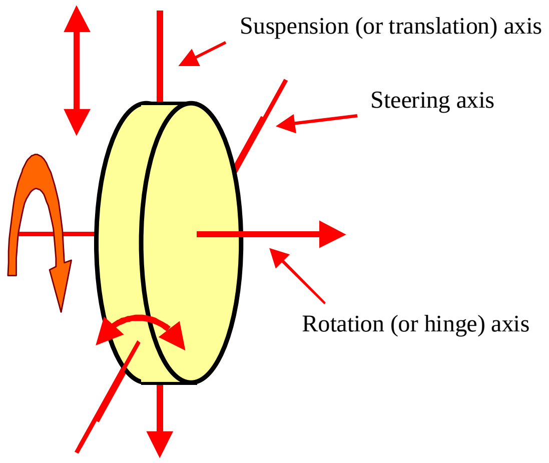
The constructor of the hinge constraint creates the **m_transform_os_ks** transforms the following way:

1st column	is set to the hinge axis.
2nd column	is set perpendicular to the hinge axis in world space and then transformed to both object spaces.
3rd column	is set perpendicular to the hinge axis and the 2nd axis.

That means that after construction, the angle used for the limits between the two objects is zero.

2.10. WheelConstraint

The car wheel is in essence a hinge with one extra free translation/suspension axis. This joint allows you to attach a wheel to a car. The rotation axis provides the axle of the wheel, allowing it to spin freely without limits. For convenience, you can also specify motor parameters from turning the wheel. The suspension axis works relative to the body of the car and allows vertical suspension movement. You can define limits and friction for the suspension part:



For convenience the car wheel constraints allows for dynamic positioning of the rotation axis to allow for steering using the *setSteeringAngle*() function. To do this, the car constraint needs an additional steering axis. In most cases, the suspension and the steering axis will be the same axis.

Note: See Vehicle SDK for an alternate method of creating vehicles.

Note: At construction of the car, the wheel constraint assumes a steering angle of 0.0.

A simple example of how it might be used to build a car using a chassis and four wheels follows:

```
#include <hkdynamics/action/Wheelconstraint.h>


// Create chassis
RigidBody* chassis = … // get a box position (0,1,0)  size (4,.5,1.5) mass 10

Vector3 wheel_pos[4];
wheel_pos[0].set(1,1,1);
wheel_pos[1].set(1,1,-1);
wheel_pos[2].set(-1,1,1);
wheel_pos[3].set(-1,1,-1);

for(int i=0; i<4; ++i)
{
    // Create a (square!) wheel
    RigidBody* wheel  = … // get a box pos wheel_pos[i]  size (1,1,1) mass 1

    // Turn off collision detection
    toolkit->m_defaultCollisionDetector->disableCollision(chassis, wheel);

    WheelConstraint* c = new WheelConstraint(chassis, wheel);

            // Specify where the wheel center should be,
            // its spin axis and its suspension axis

    c->attachWheel(
        wheel_pos[i],     // the position in world space
        Vector3(0,0,1),   // spin axis in world space
        Vector3(0,1,0) ); // suspension and steering axes in world space

            // Set limits of vertical wheel movement
            // (along suspension direction)
    c->m_suspension_limit.setLimits(-.1f,.1f);

    // Let's add a motor as well
    c->setWheelMotor(2,20);  //

    solver->addConstraint(c);
}
```

During runtime you just have to:

```
Real angle = joystick->getX();
WheelConstraint->setSteeringAngle(angle);
```

2.11. Spin Dashpots

A spin dashpot is an old constraint, which was used to build cars. It is not recommended to use them, use the wheel constraints, or even better the vehicle SDK instead.

Basically a spin dashpot is an angular constraint, which tries to keep directions defined by two objects being parallel.

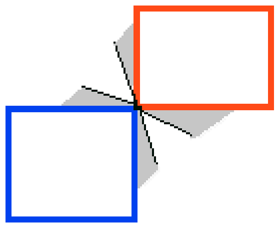
2.12. Suspension Dashpots

Suspension dashpots are an old constraint, which was used to build cars. It is not recommended to use them, use the wheel constraints, or even better the vehicle SDK instead.

Basically a suspension dashpot restricts the movement of the pivot point of a second object to move only on a ray defined in a reference object space.

2.13. LimitedPointToPointConstraint

The application for the limited point-to-point constraint is when two objects should maintain a fixed relative **position and orientation**, with some **small** degree of freedom in the orientation part. While the point-to-point constraint allows its constrained bodies to rotate freely about their contact point, the limited point-to-point constraint uses an oriented coordinate system on each body to adjust the orientations and positions of the constrained bodies.



LimitedPointToPoint constraints

You specify three axes and a pivot point (= forming the m_transform_os_ks transform) for each body, expressed in body space coordinates. In addition, you **must** specify three angular limits (one for each axis), which represent the maximum degree of flexibility in rotation and twist between these two coordinate systems. The constraint then holds both bodies within angles you have specified in relation to the offset.

Note: The constraint finds the current angle for one axis by trying to map the other two perpendicular constraint axis of the two bodies.

Important note: The LimitedPointToPointConstraint only works if all angular limits are set, and the angular range is within [–pi/2 … pi/2]. For higher ranges the constraint might get unstable. If some angles do exceed pi/2 use the **RagdollConstraint** instead.

In the following example, we constrain two bar-like bodies to be joined together at their ends. In addition, we specify restrictions on the relative orientation of the bodies at this join. Since the offset Transforms are initially aligned (we specified no rotational component using setRotation(), hence internally they both default to the identity), the three rotational limits corresponding to the difference in the two coordinate systems are aligned with the X, Y and Z axes respectively.

Thus, setting m_angular_limits[0] corresponds to restricting the amount of “twist”. This is the rotation around the first axis of the constraint (in our example the X axes). Remember this is the first column in the **m_transform_os_ks** transform, transforming from constraint space into object space).

Setting m_angular_limits[2] corresponds to restricting the amount of “bend” around the third axis (in our example the Z-axis).

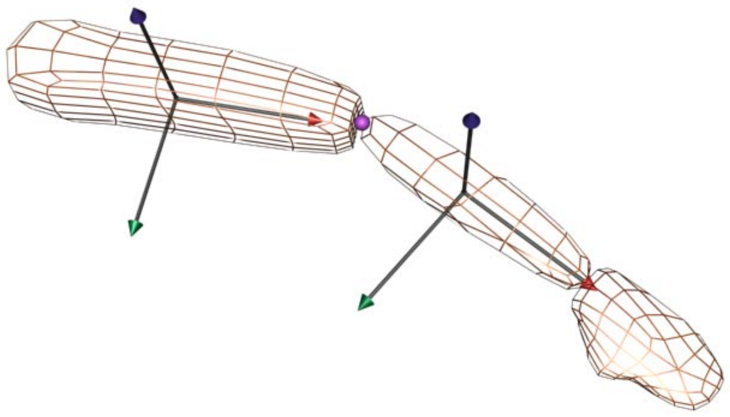
The second axis (in our example the Y-axis) has default limits of [0,0], hence permits no rotation.

```
#include <hkdynamics/action/limitedpointtopointconstraint.h>


RigidBody* box1 = … // get a box position (-2,0,0)  size (4,1,1)
RigidBody* box2 = … // get a box position ( 2,0,0)  size (4,1,1)

    // Ignore collisions for now
toolkit->m_defaultCollisionDetector->disableCollision(box1, box2);

    // Create constraint
LimitedPointToPointConstraint* c =
                           new LimitedPointToPointConstraint(box1, box2);

    // Move the constrained points to the ends of the bars
c->m_transform_os_ks[0].setTranslation(-2,0,0); // pivot point in box1 space
c->m_transform_os_ks[1].setTranslation(2,0,0);  // pivot point in box2 space

    // Expand angle limit around X to create a (restricted) twist
c->m_angular_limits[0].set(-.7f,.7f);

    // Expand angle limit around Z to create a (restricted) hinge
c->m_angular_limits[2].set(-.7f,.7f);


solver->addConstraint(c);
```

2.14. RagDollConstraint

Location: <hkdynamics/action/ragdollconstraint.h>

This is a rather sophisticated joint, which restricts the angular motion of an attached object (body 2) relative to a reference object (body 1), useful for the construction of limb joints for articulated figures such as people and animals:



Ragdoll Constraint with Pivot Point in World Space

In principle the rag doll constraint is a **modified LimitedPointToPoint-Constraint**. The main difference is that it allows for high ranges of angular limits. However, to achieve this stable constraint, certain algorithms are used, which uses the three coordinate axes in a **non-symmetrical** way.

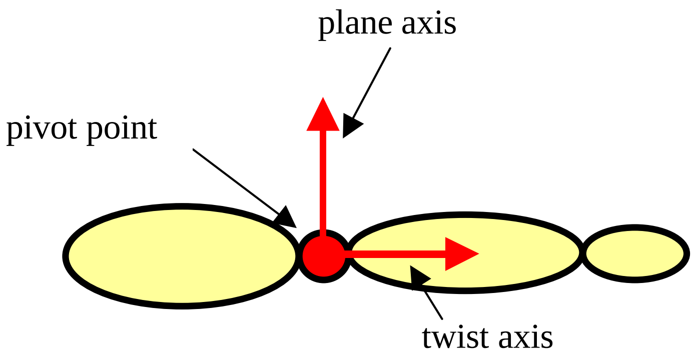
To restrict angular movements, the constraint solver:

- Takes each object's three axes, which defines the constraint space,
- Transforms them into world space,
- Compares them to calculate some angles
- Uses these angles to restrict the angular movements.

The most important axis for the rag doll constraint is called the **twist** axis. As this constraint allows some angular freedom, there are actually two twist axes, one defined per object.

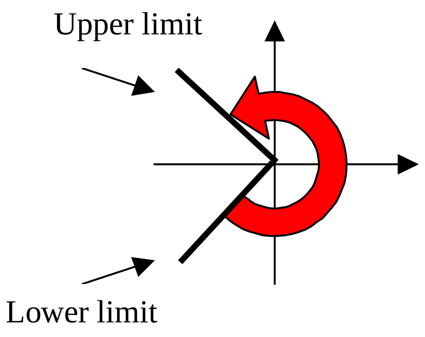
2.15. *The fast track to building rag doll constraints.*

As detailed earlier, you can define a coordinate system using a point and three axes. In order to build the constraint space coordinate system for a rag doll constraint, take the point to be the pivot point between the reference and attached bodies. The columns of the transform matrix, i.e. the axes of the constraint space, for this constraint are first twist, then cone, and finally plane. We now define the axes, beginning with the twist axis. Consider a n outstretched human arm; looking at it from the front:



2.15.1. What is the twist axis?

Consider the twist axis to be the “main” axis of the rag doll constraint. It is typically aligned with the “longest” axis of the body; a more rigorous interpretation of “longest” is to consider the twist axis to be aligned, either exactly or very closely, to the axis around which the rotational inertia is lowest. The stock example we use is an outstretched arm (as depicted diagrammatically, above). The twist axis runs the length of the arm. The angular limitations around the twist axis should be set between [-pi … pi]; diagrammatically, you should consider it as follows:



With the twist axis pointing up, and out of the page.

2.15.2. What is the plane axis?

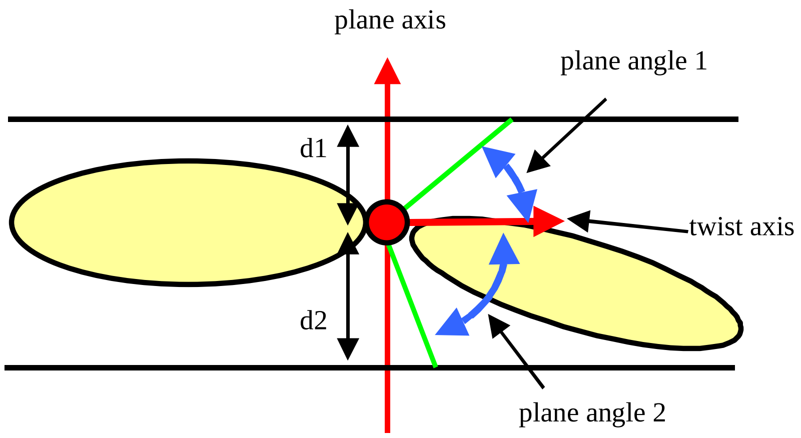
We now skip forward to the third axis, the plane axis. To decide which axis this is, search for the smaller range of angular freedom of the other two axes. Now, even though this limit restricts movement of the twist axis to lie between two planes, the limits on this axis are actually angular limits, as shown below. The planes are offset by a distance of

d1 = cos(plane angle 1)

and

d2 = cos(plane angle 2)

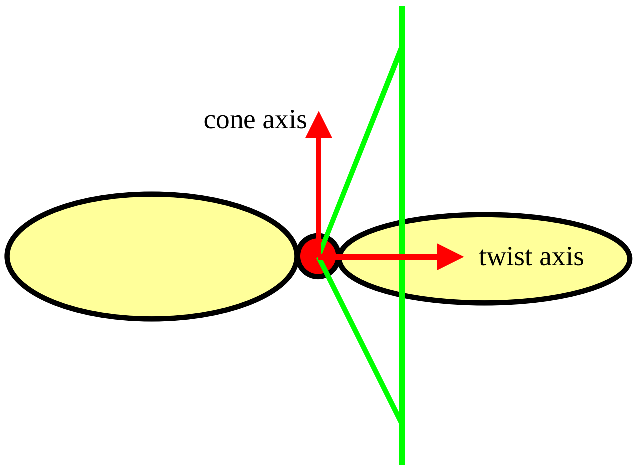
from the twist axis, with the direction of the offset being along the plane axis.



The plane angles should be in the range [-0.4 * pi … 0.4*pi].

2.15.3. What is the cone axis?

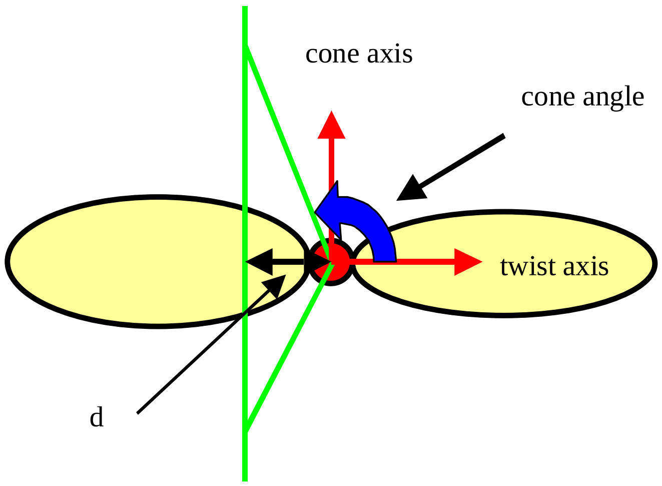
The second axis restricts the twist axis axis to movement within a cone. We are now looking top down at the arm along the plane axis, rather than at its front:



The distance of the cone plane, d, below, is calculated as

d = 1.0 – cos(cone angle)

The cone angle can also be greater than pi / 2, as shown below.



As the cone plane only restricts the angular movement of the bodies, the cones angular limits must be symmetric;

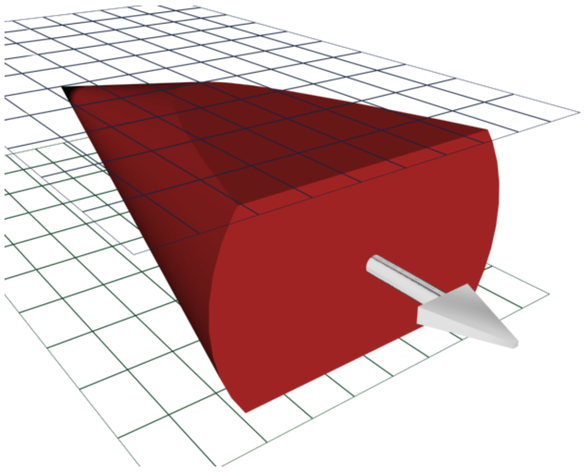
However, for convenience, you can specify angular limits that are not symmetric, and the constraint will rotate the twist axis of the first body around the plane axis so that the cone limits are symmetric.

2.15.4. Some restrictions

- The difference between the cone limits should be at least 0.1 * pi
- The sum of both cone limits should be in the range [-pi / 2 .. pi / 2]. The closer the sum is to zero, the better.

2.15.5. What it should look like!

The twist axis is constrained to lie within a clipped cone defined by two cone angles and clipping planes:



Ragdoll Constraint Cone with Clipping Planes

2.15.6. Easy steps to setup a RagdollConstraint

With all that in mind, here's a quick list of all of the things you need to be aware of when setting up a ragdoll constraint: if you just follow some easy steps, it's not as difficult as it might appear.

- Place all objects of the rag doll in your world in a reasonable state (E.g. a human figure rag doll with limbs outstretched).
- For each joint, the heavier or more central object will be object 1, the other object 2.
- Set the twist axis parallel to the longest extends of the thinner object.
- Set the twist limits as you wish.
- Search for the smaller range of angular freedom of the other two axes. This axis will become the plane axis. Set the angular plane limits accordingly.

  Note: the plane axis is perpendicular to the rotation axis you want to limit.
- Set the remaining cone axis to twistAxis.cross(planeAxis). Set the angular cone limits accordingly.

  Note: the cone axis is perpendicular to the rotation axis you want to limit.
  If you do not want to limit the movement using the cone limits, just set the limits to (-10 * pi, 10 * pi).

2.15.7. Ragdoll Source code example

Outstretch your left arm in such a way that if you turn your head 90 degrees to the left, you are looking straight along it with the palm facing downwards (assuming, of course, you are looking straight ahead at this document!). Imagine a pivot point at your elbow. How would you model the freedom that this joint has to move using a rag doll constraint?

First of all, you need to find the twist axis. This is easy in this case, as it is the axis running the length of your arm. If you stood facing forward at the origin of a coordinate system, it would extend along the x-axis. Hence, the twist axis is the x-axis. There is very little play in this joint for twisting, hence we'll set a tight limit of –0.1 .. 0.1 rads (it's debatable whether you can actually twist about it at all; maybe I just have strange arms!).

Now, you can rotate your forearm around the y-axis at your elbow in a clockwise direction almost a full half circle (it is stopped by your upper arm) in one direction, and not at all in the other. One of the features of the rag doll constraint is that the cone limits can be greater than PI/2 rads (unlike a limited point to point constraint). Hence we would like to allow the cone limits to restrict movement around the y-axis; and because the twist axis is the x-axis, this means that the cone axis will be the z-axis. Note also that the limits specified for this axis in the example below are not symmetric (see 2.4); the constraint self-orients to compensate for this. We restrict it to -3.0 .. 0.0 rads (almost a half circle, specified anticlockwise) This leaves the y-axis for the plane axis, which we will fully restrict.

We now construct this joint using a rag doll constraint.

```
#include <hkdynamics/action/ragdollconstraint.h>

RigidBody* box1 = .. // get a box position (-2,0,0) size (4,1,1)
RigidBody* box2 = .. // get a box position ( 2,0,0) size (4,1,1)

// Ignore collisions for now
toolkit->m_defaultCollisionDetector->disableCollision(box1, box2);

// Create constraint
RagdollConstraint* c = new RagdollConstraint(box1, box2);

// We want twist (0) around the x-axis, plane (2) deviated from the y-axis
// and cone (1) deviated from the z-axis. So the transforms look like this:
// (transforms are column major, so the columns read axis-axis-axis-trans,
// and the two objects are aligned along the twist axis in world space, so
// the constraint transforms are the same for both)

c->m_transform_os_ks[0].setCol(0, 1.0f, 0.0f, 0.0f);
c->m_transform_os_ks[0].setCol(1, 0.0f, 0.0f, 1.0f);
c->m_transform_os_ks[0].setCol(2, 0.0f, 1.0f, 0.0f);

c->m_transform_os_ks[1].setCol(0, 1.0f, 0.0f, 0.0f);
c->m_transform_os_ks[1].setCol(1, 0.0f, 0.0f, 1.0f);
c->m_transform_os_ks[1].setCol(2, 0.0f, 1.0f, 0.0f);

// Move the constrained points to the ends of the bars (pp – pivot point)
c->m_transform_os_ks[0].setTranslation(-2.0f, 0.0f, 0.0f); // pp in box1 space
c->m_transform_os_ks[1].setTranslation( 2.0f, 0.0f, 0.0f); // pp in box2 space

// Restrict limit around X (angular limit of twist axis)
// Allow only a small amount of rotation around the twist
// axis
c->m_limits[0].setLimits(-0.1f, 0.1f);

// Expand limit around Y (angular limit of cone axis)
// Note angular limit greater then PI/2 (feature of the
// ragdoll constraint as against limited point to point)
c->m_limits[1].setLimits(-3.0f, 0.0f);

// Restrict limit around z (angular limit of plane axis)
c->m_limits[2].setLimits( 0.0f, 0.0f);

solver->addConstraint(c);
```

3. Behind the Curtain of Constraints

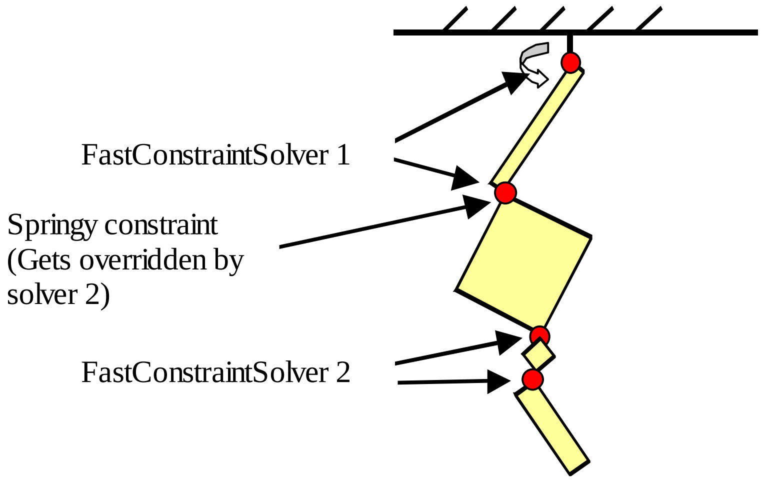
3.1. Constraint Solvers

To use constraints in a system, you need two things:

- One or more constraints, supplied with all its parameters. The constraint classes store the data required for the corresponding constraint type. Each inherits from the class **Constraint**.
- A constraint solver, which ensures that the constraints are met. Each inherits from the **ConstraintSolver** class.

A constraint solver simultaneously solves and maintains a set of constraints for a set of rigid bodies. It is important that all constraints involving these bodies are solved at the same time, so they should all be added to the same solver.

You can use several solvers for one multi body system, however some constraint might get partially overridden by the second solver and therefore get very springy:



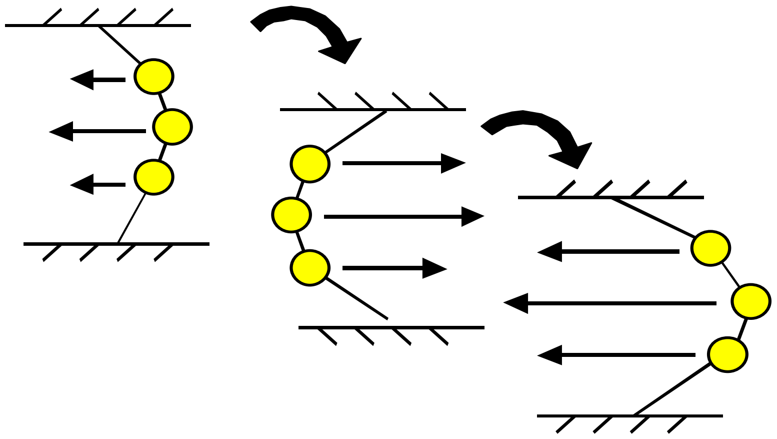
For example, if constraints are used to create hinge joints for an articulated character, then they should all be added to the same solver. If in addition you then constrain the character to the ground using another constraint, you must also add this to the same solver.

The set of all constraints that affect a set of associated rigid bodies is referred to as a constraint system.

3.2. Reasons for Instability of Constraints

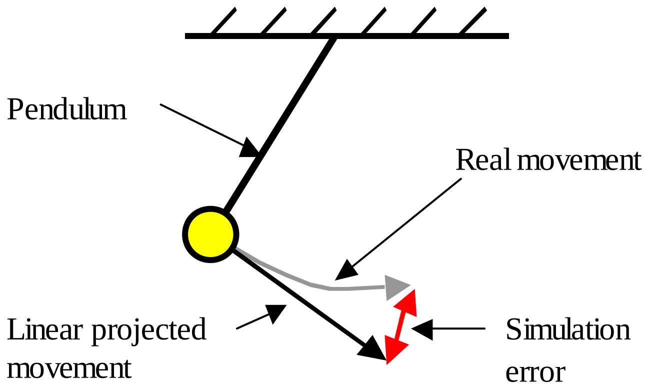
Simulating constraints is difficult and a hard simulation problem to solve. For practical reasons a constraint solver has certain stability limitations:

I. The forces involved in solving a constraint system are two high and lead to frequency of object movements higher than the simulation frequency. E.g. imagine simulating a swinging short rope, which gets pulled harder and harder. In real life you will hear a higher and higher pitched sound, in a physical simulation with 30 physical steps per second pulling the simulated rope leads to strange artefacts like instability or soft constraints:



Swinging exploding rope (with three balls)

II. The constraint solver solves a set of linear equation. In the physical simulation however, angular movement leads to non-linear behaviour. So if the time step is relatively big compared to the angular velocities of your object, the error introduced may lead to instability.



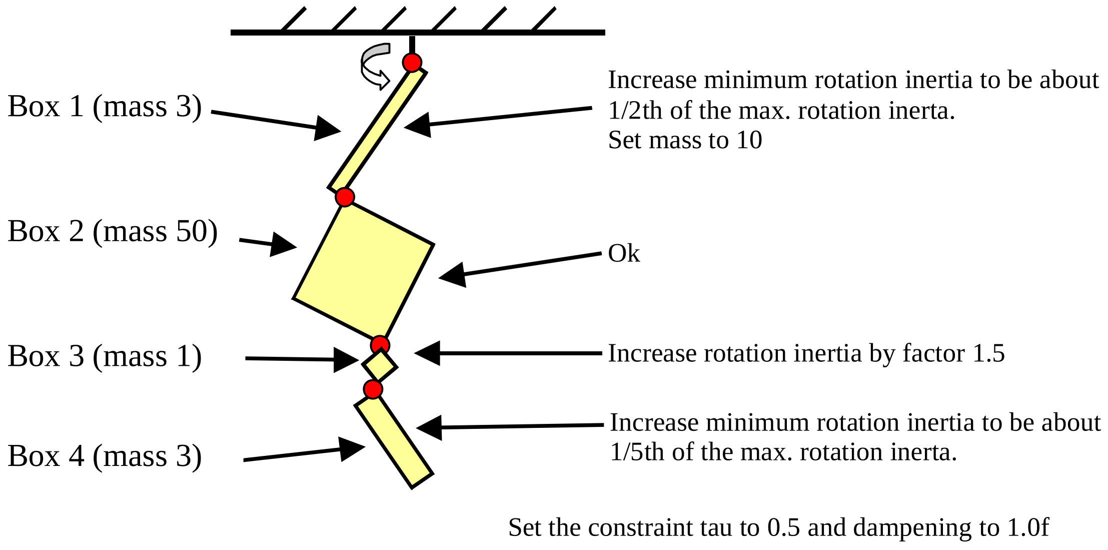
3.3. Getting stable constraints

If you want to make your constraints more stable just follow the next simple rules (start with the top rule and only if you do not get your expected results, try the next):

- Use constraints instead of dashpots, because
  constraints automatically calculate the maximum allowed forces (see Reasons for Instability of Constraints I).
- Use **Euler** integrator. All constraints are optimised for Euler integration; other integrators need more CPU with less good results.
- If you use dashpots, decrease the strength and dampening values.
- **Avoid low rotation inertia** (!!!!) because
  low inertia means high angular velocity, which leads to high simulation errors (See Reasons for Instability of Constraints II).
  Note: Long and thin objects have one axis with a very low inertia. Try to artificially increase the inertia around this axis:
  Rule:
  - In a multi body system, the minimum rotation inertia of one object should not be less then 1/10th of the maximum rotation inertia of the same object.
  - If a thin objects is constraint between two other big objects, the minimum rotation inertia of one object should not be less then 1/4th of the maximum rotation inertia of the same object
- If your constraints appear to be very soft, try to avoid constraining light objects between two heavy objects. Just make the light object at least as heavy as a 1/10th of the mass of the heavier objects.
- If you use create a big constraint system with many objects constraint together, decrease the tau value of all constraints involved:
  Rule: tau ~ 1.0/sqrt(number of constraints in a system).
- Increase rotation inertia.
- Decrease the dampening of the constraints slightly (to 0.6f).
- Increase simulation frequency by increasing the number of substeps.
- Contact havok

Note: these rules are rather strict. If you care more about physical accuracy, than you can either experiment with your system to find the real limits or if possible just increase the number of substeps.

Example of a chain of objects with some hints to get a very stable solution:


Box 1 (mass 3)
Increase minimum rotation inertia to be about 1/2th of the max. rotation inerta.
Set mass to 10
Box 2 (mass 50)
Ok
Box 3 (mass 1)
Increase rotation inertia by factor 1.5
Box 4 (mass 3)
Increase minimum rotation inertia to be about 1/5th of the max. rotation inerta.
Set the constraint tau to 0.5 and dampening to 1.0f

How to get a very stable chain of objects

3.4. CPU requirements

All dashpots and constraints in Havok are O(n) constraints. That means that 100 constraints do need 10 times more CPU than 10 constraints. As a result, there is no worse case CPU behaviour if many objects are constraint together. There are simple rules about the CPU costs of constraints:

- In principle, the more degrees of freedom a constraint restricts, the more expensive it gets. A hinge with one degree of freedom restricts the remaining 5 degrees and is about as twice as expensive as point-to-point constraint.
- Using limits and friction instead of a fixed axis is slightly more expensive. So using a rag doll constraint to simulate a hinge leads to an about 25% slower solution.
- If only one constraint is added to a FastConstraintSolver, this solver can actually perform some shortcuts and gets about twice as fast compared to solving a set of constraints.
- Dashpots are about 4 times faster than constraints.

3.5. Constraint Limitations

The number of constraints in one FastConstraintSolver is limited.

In general, the vehicle SDK is a better way to construct four wheeled vehicles, especially racing or fast moving vehicles.   Take a look at the Vehicle SDK User Guide document for more information.