Nape Constraints

Luca Deltodesco

November 25, 2012

Contents

1 Notations

Types

SCALAR values will be denoted as simple variables s
VECTOR values will be denoted with an arrow decoration like $\vec{v}$
MATRIX values will be denoted with bold, capitals like $\mathbf{M}$

Operator Notation

DOT PRODUCTS $\cdot : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}$ with $\vec{u} \cdot \vec{v} = \vec{u}^T \vec{v} = u_x v_x + u_y v_y$ in the case of $n = 2$
NORM $\|\cdot\| : \mathbb{R}^n \to \mathbb{R}$ with $\|\vec{u}\| = \sqrt{\vec{u} \cdot \vec{u}}$
CROSS PRODUCTS $\times : \mathbb{R}^2 \times \mathbb{R}^2 \to \mathbb{R}$ with $\vec{u} \times \vec{v} = u_x v_y - u_y v_x$ (Perhaps more commonly denoted the perp-dot product)
$[\cdot]_\times : \mathbb{R}^2 \to \mathbb{R}^2$ with $[\vec{u}]_\times = \begin{pmatrix} -u_y \\ u_x \end{pmatrix}$ so that $\vec{u} \times \vec{v} = [\vec{u}]_\times \cdot \vec{v}$.
Overloading the $\times$ operator; let $s \times \vec{u} = s\,[\vec{u}]_\times$ and $\vec{u} \times s = -s \times \vec{u}$ as hoped.

OUTER PRODUCTS:
$\vec{u}\vec{v}^T = \vec{u} \otimes \vec{v} = \begin{pmatrix} u_x v_x & u_x v_y \\ u_y v_x & u_y v_y \end{pmatrix}$ and in general for non-vectors too.
$[\vec{u}]_\times [\vec{v}]_\times^T = \vec{u} \odot \vec{v} = \begin{pmatrix} u_y v_y & -u_y v_x \\ -u_x v_y & u_x v_x \end{pmatrix}$ and in general for non-vectors too.

Useful results

$\vec{u} \cdot \vec{v} = \vec{v} \cdot \vec{u}$ (symmetric)
$(s\vec{u} + t\vec{v}) \cdot \vec{w} = s(\vec{u} \cdot \vec{w}) + t(\vec{v} \cdot \vec{w})$ (linear)
$\vec{u} \times \vec{v} = -\vec{v} \times \vec{u}$ and so $[\vec{u}]_\times \cdot \vec{v} = -\vec{u} \cdot [\vec{v}]_\times$
$(s\vec{u} + t\vec{v}) \times \vec{w} = s(\vec{u} \times \vec{w}) + t(\vec{v} \times \vec{w})$ (linear)
$[\vec{v}]_\times \cdot [\vec{u}]_\times = \vec{v} \cdot \vec{u}$
$\left[[\vec{v}]_\times\right]_\times = -\vec{v}$
$\vec{u} \otimes \vec{v} = (\vec{v} \otimes \vec{u})^T$ (similarly for $\odot$)
$(s\vec{u} + t\vec{v}) \otimes \vec{w} = s(\vec{u} \otimes \vec{w}) + t(\vec{v} \otimes \vec{w})$ (linear) (similarly for $\odot$)
and in general for non-vectors too with regards to outer products.
Specifically $\begin{bmatrix} \mathbf{A} \\ \mathbf{B} \end{bmatrix} \otimes \begin{bmatrix} \mathbf{C} \\ \mathbf{D} \end{bmatrix} = \begin{bmatrix} \mathbf{A} \\ \mathbf{B} \end{bmatrix} \begin{bmatrix} \mathbf{C}^T & \mathbf{D}^T \end{bmatrix} = \begin{bmatrix} \mathbf{A} \otimes \mathbf{C} & \mathbf{A} \otimes \mathbf{D} \\ \mathbf{B} \otimes \mathbf{C} & \mathbf{B} \otimes \mathbf{D} \end{bmatrix}$
TRIPLE PRODUCTS:
$\vec{v} \times (\vec{u} \times \vec{w}) = -(\vec{v} \times \vec{w})\,[\vec{u}]_\times \leftarrow$ a vector
$(\vec{u} \times \vec{v}) \times \vec{w} = (\vec{u} \times \vec{v})\,[\vec{w}]_\times \leftarrow$a vector
$\vec{u} \times (s \times \vec{v}) = s(\vec{u} \cdot \vec{v}) \leftarrow$a scalar
$(s \times \vec{u}) \times \vec{v} = -s(\vec{u} \cdot \vec{v}) \leftarrow$a scalar

Useful derivatives

$\frac{d}{dt}(\vec{u} \cdot \vec{v}) = \left(\frac{d\vec{u}}{dt} \cdot \vec{v}\right) + \left(\vec{u} \cdot \frac{d\vec{v}}{dt}\right)$
$\frac{d}{dt}(\vec{u} \times \vec{v}) = \left(\frac{d\vec{u}}{dt} \times \vec{v}\right) + \left(\vec{u} \times \frac{d\vec{v}}{dt}\right)$ (also true when permitting one of $\vec{u}, \vec{v}$ to be scalar)
$\frac{d}{dt}\|\vec{u}\| = \frac{1}{\|\vec{u}\|}\left(\vec{u} \cdot \frac{d\vec{u}}{dt}\right)$

Reserved variables

POSITION	$\vec{x}$	VELOCITY	$\vec{v}$	MASS	m
ROTATION	θ	ANGULAR VELOCITY	ω	MOMENT OF INERTIA	i

Related derivatives

$\frac{d}{dt}\vec{x} = \vec{v}$
$\frac{d}{dt}\theta = \omega$
$\frac{d\vec{u}}{dt} = \omega \times \vec{u}$ for $\vec{u}$ defined local to the coordinate system of the body (an anchor)

2 Constraint Deriviations

A (positional) constraint is defined by a linear function of all the bodies' positions and rotations collectively grouped into block-vector like:
$\vec{x} = \begin{bmatrix} \vec{x}_1 \\ \theta_1 \\ \vdots \end{bmatrix}$ and velocity block-vector $\vec{v} = \frac{d}{dt}\vec{x} = \begin{bmatrix} \vec{v}_1 \\ \omega_1 \\ \vdots \end{bmatrix}$

The positional constraint is then the function $C : \mathbb{R}^{3n} \to \mathbb{R}^m$ for an m-dimensional constraint satisfying $C(\vec{x}) = \vec{0}$ exactly when the constraint is satisfied, any additional variables used in the positional constraint should be constant whenever the positions are constant.

The related velocity constraint is then determined by $V(\vec{v}) = \frac{d}{dt}C(\vec{x}) + \vec{\beta}$ for a velocity-bias $\vec{\beta}$ (Normally $\vec{0}$) so that $V(\vec{v}) = \vec{0}$ exactly when the constraint is satisfied, the bias should be constant.

From the velocity constraint we derive the constraint JACOBIAN as the block-row-vector formed by the partial derivatives of V with respect to the velocity components.
$\mathbf{J} = \begin{bmatrix} \frac{\partial}{\partial \vec{v}_1}V & \frac{\partial}{\partial \omega_1}V & \cdots \end{bmatrix}$ this is in-fact a function of the positions $\vec{x}$ and remains fixed whenever the position vector is unchanged. The jacobian is such that $V(\vec{v}) = \mathbf{J}\vec{v} + \vec{\beta}$

Finally we determine the EFFECTIVE MASS MATRIX defined as:
$\mathbf{K} = \mathbf{J}\mathbf{M}^{-1}\mathbf{J}^T$ for the MASS MATRIX $\mathbf{M}$; defined like:
$\mathbf{M} = \begin{bmatrix} m_1\mathbf{E}_2 & & 0 \\ & i_1 & \\ 0 & & \ddots \end{bmatrix}$ for 2×2 identity matrix $\mathbf{E}_2$

The effective mass matrix computation is simplified as $\mathbf{M}$ is diagonal, giving:
$\mathbf{K} = \frac{1}{m_1}\left(\frac{\partial V}{\partial \vec{v}_1}\right) \otimes \left(\frac{\partial V}{\partial \vec{v}_1}\right) + \frac{1}{i_1}\left(\frac{\partial V}{\partial \omega_1}\right) \otimes \left(\frac{\partial V}{\partial \omega_1}\right)$
The effective mass matrix is *always* both symmetric, and positive-definite (Positive definiteness is a very nice property for matrices, akin to positive scalars so that $\vec{u}\mathbf{K}\vec{u}^T > 0$ for all $\vec{u} \neq \vec{0}$)

The constraint jacobian $\mathbf{J}$ determines the impulses we apply to the bodies; give the constraint-space impulse $\vec{\lambda} \in \mathbb{R}^m$ we get the world-space impulse $\in \mathbb{R}^{3n}$ via $\mathbf{J}^T\vec{\lambda}$.

For inequality constraints $C(\vec{x}) \leq 0$ and more generaly permitting $C(\vec{x}) \cdot \vec{e_i}$ to be either 0 or ≤ 0 for standard basis vectors $\vec{e_i}$ (Intuitively, some coordinates of constraint space should be fixed at 0 whilst others are only limited-above at 0). We can model these constraints by selectively *zeroing* appropriate coordinates of $C(\vec{x}), V(\vec{v}), \mathbf{J}, \mathbf{K}$ (permitting a more general interpretation of matrix inverse when it comes to inverting $\mathbf{K}$ internally)

For inequality constraints $a \leq C(\vec{x}) \leq b$ (and more generally on coordinate basis) introduce a weight vector $\vec{w}$ with $w_i \in \{-1, 0, 1\}$ so that we can (on a coordinate basis) transform the positional constraint into a one-sided inequality (which we know how to handle) $C_i(\vec{x}) - b \leq 0$ and $(-1) \cdot (C_i(\vec{x}) - a) \leq 0$ or simply disable the coordinate. The purpose of using the weight being that the constraint space does not become mirrored when the inequality side is swapped.

The more general interpretation of a matrix inverse, is to note that when we disable a constraint row, we end up with a matrix $\mathbf{K}$ which has a zero-cross intersecting in the diagonal like: $\mathbf{K} = \begin{pmatrix} k_{11} & 0 & k_{13} \\ 0 & 0 & 0 \\ k_{31} & 0 & k_{33} \end{pmatrix}$ and its "inverse" is given by the matrix $\mathbf{K}^{-1}$ having the same zero-crosses such that $\mathbf{K}\mathbf{K}^{-1} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$; the identity matrix with the same zero-cross applied. This always exists and is unique no matter how many zero-crosses through the diagonal we add and is equivalent to removing the row/columns of all zero-crosses, performing the inverse, then re-adding those rows and columns.

3 Predefined Constraints

3.1 PivotJoint

The PivotJoint is the simplest (non-trivial) constraint in Nape.
This joint defines two local anchor positions (hereafter labeled $\vec{a}_1$ and $\vec{a}_2$) whose world-coordinates are locked together.

We can define the 2-dimensional position constraint as:
$C(\vec{x}) = (\vec{x}_2 + \mathbf{R}_{\theta_2}\vec{a}_2) - (\vec{x}_1 + \mathbf{R}_{\theta_1}\vec{a}_1) = \vec{0}$ for rotation matrix $\mathbf{R}_\theta$
Clearly when position vector is unchanged, $\mathbf{R}_\theta\vec{a} = \vec{r}$ is constant.

$$\boxed{\textsc{Positional constraint:}\ C(\vec{x}) = (\vec{x}_2 + \vec{r}_2) - (\vec{x}_1 + \vec{r}_1) = \vec{0}}$$

The 2-dimensional velocity constraint is:
$V(\vec{v}) = \frac{d}{dt}C(\vec{x})$ and we define no velocity bias (since we want velocity at anchor points to be equal).

$$\boxed{\textsc{Velocity constraint:}\ V(\vec{v}) = (\vec{v}_2 + \omega_2 \times \vec{r}_2) - (\vec{v}_1 + \omega_1 \times \vec{r}_1) = \vec{0}}$$

For the jacobian, it will be helpful to write the velocity constraint like:
$V(\vec{v}) = -\mathbf{E}_2\vec{v}_1 - [\vec{r}_1]_\times\,\omega_1 + \mathbf{E}_2\vec{v}_2 + [\vec{r}_2]_\times\,\omega_2$
from which we can identify the Jacobian easily.

$$\boxed{\textsc{Jacobian:}\ \mathbf{J} = \begin{bmatrix} -\mathbf{E}_2 & -[\vec{r}_1]_\times & \mathbf{E}_2 & [\vec{r}_2]_\times \end{bmatrix} = \begin{pmatrix} -1 & 0 & r_{1y} & 1 & 0 & -r_{2y} \\ 0 & -1 & -r_{1x} & 0 & 1 & r_{2x} \end{pmatrix}}$$

The effective mass matrix is then given by:
$\mathbf{K} = \frac{1}{m_1}(-\mathbf{E}_2) \otimes (-\mathbf{E}_2) + \frac{1}{i_1}\left(-[\vec{r}_1]_\times\right) \otimes \left(-[\vec{r}_1]_\times\right) \ldots$

$$\boxed{\textsc{Eff-Mass:}\ \mathbf{K} = \left(\frac{1}{m_1} + \frac{1}{m_2}\right)\mathbf{E}_2 + \frac{1}{i_1}(\vec{r}_1 \odot \vec{r}_1) + \frac{1}{i_2}(\vec{r}_2 \odot \vec{r}_2) = \begin{pmatrix} \frac{1}{m_1} + \frac{1}{m_2} + \frac{r_{1y}^2}{i_1} + \frac{r_{2y}^2}{i_2} & -\frac{r_{1x}r_{1y}}{i_2} - \frac{r_{2x}r_{2y}}{i_2} \\ \# & \frac{1}{m_1} + \frac{1}{m_2} + \frac{r_{1x}^2}{i_1} + \frac{r_{2x}^2}{i_2} \end{pmatrix}}$$

The impulse applied to bodies in world space is then given by the coordinates of

$$\mathbf{J}^T\vec{\lambda} = \begin{bmatrix} -\mathbf{E}_2 \\ -[\vec{r}_1]_\times^T \\ \mathbf{E}_2 \\ [\vec{r}_2]_\times^T \end{bmatrix} \begin{pmatrix} \lambda_x \\ \lambda_y \end{pmatrix} = \begin{pmatrix} -\mathbf{E}_2\vec{\lambda} \\ -[\vec{r}_1]_\times^T\vec{\lambda} \\ \mathbf{E}_2\vec{\lambda} \\ [\vec{r}_2]_\times^T\vec{\lambda} \end{pmatrix}$$

$$\boxed{\textsc{Impulses:}\ \mathbf{J}^T\vec{\lambda} = \begin{pmatrix} -\vec{\lambda} \\ -\vec{r}_1 \times \vec{\lambda} \\ \vec{\lambda} \\ \vec{r}_2 \times \vec{\lambda} \end{pmatrix} \Rightarrow \begin{cases} \vec{v}_1' = \vec{v}_1 - \frac{1}{m_1}\vec{\lambda} \\ \omega_1' = \omega_1 - \frac{1}{i_1}\left(\vec{r}_1 \times \vec{\lambda}\right) \\ \vec{v}_2' = \vec{v}_2 + \frac{1}{m_2}\vec{\lambda} \\ \omega_2' = \omega_2 + \frac{1}{i_2}\left(\vec{r}_2 \times \vec{\lambda}\right) \end{cases}}$$

3.2 WeldJoint

The WeldJoint is an extension of the PivotJoint to lock the rotations of the bodies to a fixed phase, this is; like the PivotJoint fairly trivial as we have no inequalities, this constraint is 3-dimensional.

Like the PivotJoint we have anchors $\vec{a}$ and world-space anchors $\vec{r}$. We also have an angular phase ϕ

$$\textsc{Positional constraint:}\ C(\vec{x}) = \left[\begin{array}{c} (\vec{x}_2 + \vec{r}_2) - (\vec{x}_1 + \vec{r}_1) \\ \theta_2 - \theta_1 - \phi \end{array} \right] = \vec{0}$$

Again a velocity bias of $\vec{0}$

$$\textsc{Velocity constraint:}\ V(\vec{v}) = \left[\begin{array}{c} (\vec{v}_2 + \omega_2 \times \vec{r}_2) - (\vec{v}_1 + \omega_1 \times \vec{r}_1) \\ \omega_2 - \omega_1 \end{array} \right] = \vec{0}$$

For the jacobian, we can use a trick of taking the jacobian of each row in the velocity constraint and composing the results:

$$\textsc{Jacobian:}\ \mathbf{J} = \left[\begin{array}{cccc} -\mathbf{E}_2 & -[\vec{r}_1]_\times & \mathbf{E}_2 & [\vec{r}_2]_\times \\ \vec{0}^T & -1 & \vec{0}^T & 1 \end{array} \right] = \begin{pmatrix} -1 & 0 & r_{1y} & 1 & 0 & -r_{2y} \\ 0 & -1 & -r_{1x} & 0 & 1 & r_{2x} \\ 0 & 0 & -1 & 0 & 0 & 1 \end{pmatrix}$$

Using our results to partially compute effective mass, we see:

$$\left[\begin{array}{c} \mathbf{E}_2 \\ \vec{0}^T \end{array} \right] \otimes \left[\begin{array}{c} \mathbf{E}_2 \\ \vec{0}^T \end{array} \right] = \left[\begin{array}{cc} \mathbf{E}_2 & \vec{0} \\ \vec{0}^T & 0 \end{array} \right] = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

$$\left[\begin{array}{c} [\vec{r}]_\times \\ 1 \end{array} \right] \otimes \left[\begin{array}{c} [\vec{r}]_\times \\ 1 \end{array} \right] = \left[\begin{array}{cc} \vec{r} \odot \vec{r} & [r]_\times \\ [r]_\times^T & 1 \end{array} \right] = \begin{pmatrix} r_y^2 & -r_x r_y & -r_y \\ -r_x r_y & r_x^2 & r_x \\ -r_y & r_x & 1 \end{pmatrix}$$

$$\textsc{Eff-Mass:}\ \mathbf{K} = \begin{pmatrix} \frac{1}{m_1} + \frac{1}{m_2} + \frac{r_{1y}^2}{i_1} + \frac{r_{2y}^2}{i_2} & -\frac{r_{1x} r_{1y}}{i_2} - \frac{r_{2x} r_{2y}}{i_2} & -\frac{r_{1y}}{i_1} - \frac{r_{2y}}{i_2} \\ & \frac{1}{m_1} + \frac{1}{m_2} + \frac{r_{1x}^2}{i_1} + \frac{r_{2x}^2}{i_2} & \frac{r_{1x}}{i_1} + \frac{r_{2x}}{i_2} \\ \# & & \frac{1}{i_1} + \frac{1}{i_2} \end{pmatrix}$$

$$\mathbf{J}^T \vec{\lambda} = \left[\begin{array}{cc} -\mathbf{E}_2 & \vec{0} \\ -[\vec{r}_1]_\times^T & -1 \\ \mathbf{E}_2 & \vec{0} \\ [\vec{r}_2]_\times^T & 1 \end{array} \right] \begin{pmatrix} \lambda_x \\ \lambda_y \\ \lambda_z \end{pmatrix}$$

$$\textsc{Impulses:}\ \mathbf{J}^T \vec{\lambda} = \begin{pmatrix} -\vec{\lambda}_{xy} \\ -\left(\vec{r}_1 \times \vec{\lambda}_{xy}\right) - \lambda_z \\ \vec{\lambda}_{xy} \\ \left(\vec{r}_2 \times \vec{\lambda}_{xy}\right) + \lambda_z \end{pmatrix} \text{ where } \vec{\lambda}_{xy} = \begin{pmatrix} \lambda_x \\ \lambda_y \end{pmatrix}$$

3.3 AngleJoint

The AngleJoint is a double inequality constraint, which means we now introduce the weight vector $\vec{w}$. The AngleJoint is 1 dimensional, so in this case we just have a scalar weight w.

The AngleJoint constrains the weighted sum of body rotations by a ratio, to be in a given range. I introduce variable ρ for the ratio, and x_0, x_1 with $x_0 \leq x_1$ for the joint limits.

Positional constraint: $x_0 \leq (c(\vec{x}) = \rho\theta_2 - \theta_1) \leq x_1$
transform:
$C(\vec{x}) = c(\vec{x}) - x_0 = 0$ when $x_0 = x_1$ (standard constraint with $w = 1$)

$C(\vec{x}) = x_0 - c(\vec{x}) \leq 0$ when $c(\vec{x}) \leq x_0$ (inequality constraint with $w = -1$)
$C(\vec{x}) = 0$ when $x_0 < c(\vec{x}) < x_1$ (disabled inequality constraint with $w = 0$)
$C(\vec{x}) = c(\vec{x}) - x_1 \leq 0$ when $c(\vec{x}) \geq x_1$ (inequality constraint with $w = 1$)

Using the weight w we can leave the rest of the mathematics untouched by complication.

Velocity constraint: $V(\vec{v}) = w\left(\rho\omega_2 - \omega_1\right) \leq 0$. Can see this is 0 when $w = 0$ as required; in the case that we have a standard constraint we replace $\leq$ with $=$

Jacobian: $\mathbf{J} = \left[\begin{array}{cccc} \vec{0}^T & -w & \vec{0}^T & w\rho \end{array}\right] = \left(\begin{array}{cccccc} 0 & 0 & -w & 0 & 0 & w\rho \end{array}\right)$. Can see this is 0 when $w = 0$ as required.

Eff-Mass: $\mathbf{K} = w^2\left(\frac{1}{i_1} + \frac{\rho^2}{i_2}\right)$. Can see this is 0 when $w = 0$ as required.

Impulses: $\mathbf{J}^T\vec{\lambda} = w\lambda\begin{pmatrix} \vec{0} \\ -1 \\ \vec{0} \\ \rho \end{pmatrix}$. Which is 0 when $w = 0$ as required, when we have a true inequality constraint we will always have $\lambda \leq 0$.

In the case of the AngleJoint, as it's 1 dimensional. When $w = 0$ we simply skip any subsequent calculations.

3.4 DistanceJoint

The DistanceJoint constrains the distance between two anchors $\vec{a_1}, \vec{a}_2$ of the bodies to be in a given positive range of values. This like the AngleJoint is a double inequality constraint, and is also a 1-dimensional constraint.

POSITIONAL CONSTRAINT: $x_0 \leq (c(\vec{x}) = \|(\vec{x}_2 + \vec{r}_2) - (\vec{x}_1 + \vec{r}_1)\|) \leq x_1$ WITH $x_0 \geq 0$
TRANSFORM *into* $C(\vec{x})$ as previously introducing weight w

We then have the velocity constraint given by
$V(\vec{v}) = w\frac{d}{dt}c(\vec{x}) = \frac{w}{\|(\vec{x}_2+\vec{r}_2)-(\vec{x}_1+\vec{r}_1)\|}\left((\vec{x}_2 + \vec{r}_2) - (\vec{x}_1 + \vec{r}_1)\right) \cdot \left((v_2 + \omega_2 \times \vec{r}_2) - (\vec{v}_1 + \omega_1 \times \vec{r}_1)\right)$
For the purposes of simplifying the mathematics (and code) we introduce the vector
$\vec{n} = w \cdot \text{unit}\left((\vec{x}_2 + \vec{r}_2) - (\vec{x}_1 + \vec{r}_1)\right)$. It is clear that should the distance between anchors be exactly 0 we have a problem; the constraint degenerates; to solve this we cache the previously used $\vec{n}$ and recommend that the lower limit be strictly greater than 0.

VELOCITY CONSTRAINT: $V(\vec{v}) = \vec{n} \cdot \left((v_2 + \omega_2 \times \vec{r}_2) - (\vec{v}_1 + \omega_1 \times \vec{r}_1)\right)$

To extract the jacobian, we can rewrite $V(\vec{v})$ like:
$V(\vec{v}) = -\vec{n} \cdot \vec{v}_1 - \omega_1 \vec{n} \cdot [\vec{r}_1]_\times + \vec{n} \cdot \vec{v}_2 + \omega_2 \vec{n} \cdot [\vec{r}_2]_\times$
and noting that $\vec{n} \cdot [\vec{r}]_\times = \vec{r} \times \vec{n}$

Simplifying the mathematics and code again, let $c_1 = \vec{r}_1 \times \vec{n}$ and similarly for c_2

JACOBIAN: $\mathbf{J} = \begin{bmatrix} -\vec{n}^T & -c_1 & \vec{n}^T & c_2 \end{bmatrix}$

Noting that $\vec{n}^T \otimes \vec{n}^T = \vec{n}^T \vec{n} = \vec{n} \cdot \vec{n} = \|\vec{n}\| = w^2$ by definition.

EFF-MASS: $\mathbf{K} = w^2 \left(\frac{1}{m_1} + \frac{1}{m_2} + \frac{c_1^2}{i_1} + \frac{c_2^2}{i_2}\right)$

IMPULSES: $\mathbf{J}^T \vec{\lambda} = \lambda \begin{pmatrix} -\vec{n} \\ -c_1 \\ \vec{n} \\ c_2 \end{pmatrix}$

3.5 LineJoint

The LineJoint constrains the anchor of the second body $\vec{a}_2$ to be restricted to a possibly infinite line-segment defined by the anchor of the first body $\vec{a}_1$ and a local direction to which we will assign the world-space vector $\vec{n}$, normalised.

The LineJoint is a two-dimensional constraint which is in one dimension, a standard constraint, and in the other dimension, a two-way inequality.
To simplify the mathematics and code, define $\vec{d} = (\vec{x}_2 + \vec{r}_2) - (\vec{x}_1 + \vec{r}_1)$

POSITION CONSTRAINT: $c_1(\vec{x}) = \vec{n} \times \vec{d} = 0,\ x_0 \leq \left(c_2(\vec{x}) = \vec{n} \cdot \vec{d}\right) \leq x_1$
TRANSFORM into $C(\vec{x})$ with a weight vector $\vec{w} = \begin{pmatrix} 1 \\ w \end{pmatrix}$

The velocity constraint is then:

$$V(\vec{v}) = \vec{w} \begin{bmatrix} \frac{d}{dt}\vec{n} \times \vec{d} + \vec{n} \times \frac{d}{dt}\vec{d} \\ \frac{d}{dt}\vec{n} \cdot \vec{d} + \vec{n} \cdot \frac{d}{dt}\vec{d} \end{bmatrix} \text{ where } \frac{d}{dt}\vec{n} = \omega_1 \times \vec{n} \text{ and } \frac{d}{dt}\vec{d} = (\vec{v}_2 + \omega_2 \times \vec{r}_2) - (\vec{v}_1 + \omega_1 \times \vec{r}_1)$$

$$V(\vec{v}) = \vec{w} \begin{bmatrix} (\omega_1 \times \vec{n}) \times \vec{d} + \vec{n} \times v_2 - \vec{n} \times v_1 + \vec{n} \times (\omega_2 \times \vec{r}_2) - \vec{n} \times (\omega_1 \times \vec{r}_1) \\ (\omega_1 \times \vec{n}) \cdot \vec{d} + \vec{n} \cdot v_2 - \vec{n} \cdot v_1 + \vec{n} \cdot (\omega_2 \times \vec{r}_2) - \vec{n} \cdot (\omega_1 \times \vec{r}_1) \end{bmatrix}$$

We introduce the following to simplifiy the maths and code.
$d_1 = \vec{n} \cdot (\vec{d} - \vec{r}_1)$
$d_2 = \vec{n} \cdot \vec{r}_2$
$c_1 = \vec{n} \times (\vec{d} - \vec{r}_1)$
$c_2 = \vec{n} \times \vec{r}_2$

VELOCITY CONSTRAINT: $V(\vec{v}) = \vec{w} \begin{bmatrix} \vec{n} \times (\vec{v}_2 - \vec{v}_1) + \omega_2 d_2 - \omega_1 d_1 \\ \vec{n} \cdot (\vec{v}_2 - \vec{v}_1) - \omega_2 c_2 + \omega_1 c_1 \end{bmatrix}$

JACOBIAN: $\mathbf{J} = \vec{w} \begin{bmatrix} -[\vec{n}]_\times^T & -d_1 & [\vec{n}]_\times^T & d_2 \\ -\vec{n}^T & c_1 & \vec{n}^T & c_2 \end{bmatrix}$

We normalised $\vec{n}$ so that $\left(\vec{w} \begin{bmatrix} [\vec{n}]_\times^T \\ \vec{n}^T \end{bmatrix}\right) \otimes \left(\vec{w} \begin{bmatrix} [\vec{n}]_\times^T \\ \vec{n}^T \end{bmatrix}\right) = \vec{w}\mathbf{E}_2\vec{w}^T = \begin{pmatrix} 1 & 0 \\ 0 & w^2 \end{pmatrix}$

And $\left(\vec{w} \begin{bmatrix} a \\ b \end{bmatrix}\right) \otimes \left(\vec{w} \begin{bmatrix} a \\ b \end{bmatrix}\right) = \begin{pmatrix} a^2 & wab \\ wab & w^2 b^2 \end{pmatrix}$

EFF-MASS: $\mathbf{K} = \begin{pmatrix} \frac{1}{m_1} + \frac{1}{m_2} + \frac{d_1^2}{i_1} + \frac{d_2^2}{i_2} & -w\left(\frac{d_1 c_1}{i_1} + \frac{d_2 c_2}{i_2}\right) \\ \# & w^2\left(\frac{1}{m_1} + \frac{1}{m_2} + \frac{c_1^2}{i_1} + \frac{c_2^2}{i_2}\right) \end{pmatrix}$

IMPULSES: $\mathbf{J}^T\vec{\lambda} = \begin{pmatrix} -\vec{\gamma} \\ wc_1\lambda_y - d_1\lambda_x \\ \vec{\gamma} \\ d_2\lambda_x - wc_1\lambda_y \end{pmatrix}$ where $\vec{\gamma} = \begin{pmatrix} wn_x\lambda_y - n_y\lambda_x \\ n_x\lambda_x - wn_y\lambda_y \end{pmatrix}$

3.6 MotorJoint

The MotorJoint is a velocity-only constraint (It has no positional constraint) which locks the weighted difference of body angular velocities to be at a given rate.

Using ρ for ratio as previous and recycling ϕ as the rate:

VELOCITY CONSTRAINT: $V(\vec{v}) = \rho\omega_2 - \omega_1 - \phi$. Noticing that ϕ is free and forms the velocity bias.

JACOBIAN: $\mathbf{J} = \begin{bmatrix} \vec{0}^T & -1 & \vec{0}^T & \rho \end{bmatrix}$

EFF-MASS: $\mathbf{K} = \frac{1}{i_1} + \frac{\rho^2}{i_2}$

IMPULSES: $\mathbf{J}^T\vec{\lambda} = \begin{pmatrix} \vec{0} \\ -\lambda \\ \vec{0} \\ \rho\lambda \end{pmatrix}$

4 UserConstraint

The UserConstraint API provides a way to define a customised Nape constraint at a reasonably low-level. Performance is not going to be good as writing one by hand entirely, but is FAR simpler and quicker and unless you have hundreds of them will be perfectly performant.

UserConstraints can automatically be made into soft constraints, have force limits set, breakable just like normal Nape constraints without any user code.

A UserConstraint looks like:

```
typedef TArray<T> = #if flash9 flash.Vector<T> #else Array<T> #end;
class MyConstraint extends UserConstraint {
    public function new() {
        super(/*number of dimensions*/, /*is velocity only*/=false);
    }
    public override function __position(err:TArray<Float>):Void {
        // populate 'err' with values of C(x)
        err[0] = ..;
         ...
        err[dimensions-1] = ..;
    }
    public override function __velocity(err:TArray<Float>):Void {
        // populate 'err' with values of V(v)
        err[0] = ..;
         ...
        err[dimensions-1] = ..;
    }
    public override function __eff_mass(eff:TArray<Float>):Void {
        // populate 'eff' with values of the effective mass.
        // the matrix is compressed abusing mass symmetry and is indexed like
        // K = [eff[0], eff[1], eff[2],
        //              eff[3], eff[4],
        //              eff[5], eff[6]] for a 3-dimensional constraint.
        eff[0] = ..;
         ...
        eff[#] = ..;
    }
    public override function __impulse(imp:TArray<Float>, body:Body, out:Vec3):Void {
        // populate 'out' Vec3 with impulse to be applied to 'body' given
        // constraint space impulse (lambda) 'imp'.
        // Note we do not 'apply' any impulse here, only populate 'out'
        out.x = ..;
        out.y = ..;
        out.z = ..;
    }
    public override function __clamp(jAcc:TArray<Float>):Void {
        // perform any required clamping of accumulated impulses
        // such as for inequality constraints
    }
    public override function __draw(debug:Debug):Void {
        // draw a representation of constraint to Debug object.
    }
    ...
    public override function __copy():UserConstraint {
        // produce and return an exact copy of this UserConstraint
    }
    public override function __broken():Void {
        // called when constraint is broken, before it is removed from the
        // space or deactivated.
    }
    public override function __validate():Void {
        // perform any verification steps on the integrity of the constraint
        // if there are any computations that can be re-used through both
        // the velocity and positional iterations, they can be done here.
    }
    public override function __prepare():Void {
        // perform any computations that are dependent only on the positions
        // of the bodies and remain fixed during velocity iterations.
    }
}
```

A UserConstraint should have an API that is indistinguishable from the other Nape constraints, and to provide this you should use the following models for common property types:

Body properties :

We want to ensure that when we set and change Body type properties that all necessary internal actions are performed, this can be achieved with the following model:

```
//Haxe
public var body1(default, set_body1):Body;
function set_body1(body1:Body) {
    return this.body1 = __registerBody(this.body1, body1);
}

//AS3
var body1:Body;
public function get body1():Body {
    return this.body1;
}
public function set body1(body1:Body):void {
    this.body1 = __registerBody(this.body1, body1);
}
```

Vec2 properties:

We want to ensure that not only does the Constraint be refreshed when the Vec2 property is changed directly, but also when its x/y values are changed indirectly. This can be achieved with the following model:

```
//Haxe
public var anchor1(default, set_anchor1):Vec2;
function set_anchor1(anchor1:Vec2) {
    if (this.anchor1 == null) this.anchor1 = __bindVec2();
    return this.anchor1.set(anchor1);
}

//AS3
var anchor1:Vec2 = __bindVec2();
public function get anchor1():Vec2 {
    return this.anchor1;
}
public function set anchor1(anchor1:Vec2):void {
    return this.anchor1.set(anchor1);
}
```

Other types:

For other parameter types, you need to ensure yourself that mutations of the property invalidate the appropariate constraint the property belongs to.

For basic types, this is trivial:

```
//Haxe
public var ratio(default, set_ratio):Float;
function set_ratio(ratio:Float) {
    if (this.ratio != ratio) __invalidate();
    return this.ratio = ratio;
}

//AS3
var ratio:Number;
public function get ratio():Number {
    return this.ratio;
}
public function set ratio(ratio:Number):void {
    if (this.ratio != ratio) __invalidate();
    this.ratio = ratio;
}
```

4.1 UserConstraint :: PivotJoint

Implementing a PivotJoint with the UserConstraint could then be done like:

```
typedef TArray<T> = #if flash9 flash.Vector<T> #else Array<T> #end;
class UserPivotJoint extends UserConstraint {
    public var body1(default, set_body1):Body;
    public var body2(default, set_body2):Body;
    function set_body1(body1:Body) return this.body1 = __registerBody(this.body1, body1)
    function set_body2(body2:Body) return this.body2 = __registerBody(this.body2, body2)

    public var anchor1(default, set_anchor1):Vec2;
    public var anchor2(default, set_anchor2):Vec2;
    function set_anchor1(anchor1:Vec2) {
        if (this.anchor1 == null) this.anchor1 = __bindVec2();
        return this.anchor1.set(anchor1);
    }
    function set_anchor2(anchor2:Vec2) {
        if (this.anchor2 == null) this.anchor2 = __bindVec2();
        return this.anchor2.set(anchor2);
    }

    public function new(body1:Null<Body>, body2:Null<Body>, anchor1:Vec2, anchor2:Vec2) {
        super(2);
        this.body1 = body1;
        this.body2 = body2;
        this.anchor1 = anchor1;
        this.anchor2 = anchor2;

        rel1 = Vec2.get();
        rel2 = Vec2.get();
    }
    public override function __copy():UserConstraint {
        return new UserPivotJoint(body1, body2, anchor1, anchor2);
    }

    public override function __validate():Void {
        // example:
        if (body1 == null || body2 == null)
            throw "Error: UserPivotJoint cannot be simulated with null bodies!";
    }

    var rel1:Vec2;
    var rel2:Vec2;
    public override function __prepare():Void {
        rel1.set(body1.localVectorToWorld(anchor1, true));
        rel2.set(body2.localVectorToWorld(anchor2, true));
    }
    public override function __position(err:TArray<Float>):Void {
        err[0] = (body2.position.x + rel2.x) - (body1.position.x + rel1.x);
        err[1] = (body2.position.y + rel2.y) - (body1.position.y + rel1.y);
    }
    public override function __velocity(err:TArray<Float>):Void {
        var v1 = body1.constraintVelocity;
        var v2 = body2.constraintVelocity;
        err[0] = (v2.x - rel2.y * v2.z) - (v1.x - rel1.y * v1.z);
        err[1] = (v2.y + rel2.x * v2.z) - (v1.y + rel1.x * v1.z);
    }
    public override function __eff_mass(eff:TArray<Float>):Void {
        var m1 = body1.constraintMass; var i1 = body1.constraintInertia;
        var m2 = body2.constraintMass; var i2 = body2.constraintInertia;
        eff[0] = m1 + m2 + (rel1.y * rel1.y * i1) + (rel2.y * rel2.y * i2);
        eff[1] =         - (rel1.x * rel1.y * i1) - (rel2.x * rel2.y * i2);
        eff[2] = m1 + m2 + (rel1.x * rel1.x * i1) + (rel2.x * rel2.x * i2);
    }
    public override function __impulse(imp:TArray<Float>, body:Body, out:Vec3):Void {
        var scale = if (body == body1) -1.0 else 1.0;
        var relv  = if (body == body1) rel1 else rel2;
        out.x = scale * imp[0];
        out.y = scale * imp[1];
        out.z = scale * relv.cross(Vec2.weak(imp[0], imp[1]));
    }
}
```

4.2 UserConstraint :: AngleJoint

Implementing an AngleJoint is more fun :)

```
typedef TArray<T> = #if flash9 flash.Vector<T> #else Array<T> #end;
class UserAngleJoint extends UserConstraint {
    ... body1, body2 as above

    public var ratio(default, set_ratio):Float;
    public var jointMin(default, set_jointMin):Float;
    public var jointMax(default, set_jointMax):Float;
    ... + the model setters.

    public function new(body1:Null<Body>, body2:Null<Body>, ...) {
        super(1);
        ...
    }
    public override functino __copy():UserConstraint {
        return new UserAngleJoint(body1, body2, ratio, jointMin, jointMax);
    }

    var equality:Bool = false;
    public override function __validate():Void {
        equality = (jointMin == jointMax);
    }
    var weight:Float = 1.0;
    var error:Float;
    public override function __prepare():Void {
        // We compute positional error in here too to avoid repeating
        // any calculations.
        var cx = (ratio * body2.rotation - body1.rotation);
        if (equality) {
            weight = 1;
            error = cx - jointMax;
        }
        else if(cx <= jointMin) {
            error = jointMin - cx;
            weight = -1;
        }
        else if(cx >= jointMax) {
            error = cx - jointMax;
            weight = 1;
        }
        else {
            error = 0;
            weight = 0;
        }
    }
    public override function __position(err:TArray<Float>):Void {
        err[0] = error;
    }
    public override function __velocity(err:TArray<Float>):Void {
        var v1 = body1.constraintVelocity;
        var v2 = body2.constraintVelocity;
        err[0] = (weight * (ratio * v2.w)) - v1.w;
    }
    public override function __eff_mass(eff:TArray<Float>):Void {
        var i1 = body1.constraintInertia;
        var i2 = body2.constraintInertia;
        eff[0] = i1 + (ratio * ratio * i2);
    }
    public override function __impulse(imp:TArray<Float>, body:Body, out:Vec3):Void {
        var scale = if (body == body1) -weight else (weight * ratio);
        out.x = out.y = 0;
        out.z = scale * imp[0];
    }
    public override function __clamp(jAcc:TArray<Float>):Void {
        if (!equality && jAcc[0] > 0) jAcc[0] = 0;
    }
}
```

5 SymbolicConstraint

The SymbolicConstraint class from the nape-symbolic module provides an 'even higher' path to creating custom constraints. It does this through an expressive DSL (Domain Specific Language) to define positional constraints in (nape-symbolic does not at present support velocity only constraints).

These SymbolicConstraints have a slightly more obtuse API for setting properties like:

```
symbolicConstraint.setBody("body1", someBody);
symbolicConstraint.setVector("anchor1", someVec2);
symbolicConstraint.setScalar("ratio", someFloat);
```

Check the SymbolicConstraint API entry for details on the syntax of the DSL, for this manual I will simply give some example usages:

```
var symbolicWeldJoint = new SymbolicConstraint("
    body body1, body2
    vector anchor1, anchor2
    scalar phase

    constraint
        let r1 = relative body1.rotation anchor1 in
        let r2 = relative body2.rotation anchor2 in
        { (body2.position + r2) - (body1.position + r1)
          body2.rotation - body1.rotation - phase }
");
```

```
var symbolicDistanceJoint = new SymbolicConstraint("
    body body1, body2
    vector anchor1, anchor2
    scalar jointMin, jointMax

    limit jointMin 0 jointMax # 0 <= jointMin <= jointMax

    constraint
        let r1 = relative body1.rotation anchor1 in
        let r2 = relative body2.rotation anchor2 in
        |(body2.position + r2) - (body1.position + r1)|

    limit constraint jointMin jointMax # jointMin <= constraint <= jointMax
");
```

```
var symbolicLineJoint = new SymbolicConstraint("
    body body1, body2
    vector anchor1, anchor2, direction
    scalar jointMin, jointMax

    limit jointMin (-inf) jointMax # jointMin <= jointMax
    limit |direction| eps inf  # direction != 0

    constraint
        let r1 = relative body1.rotation anchor1 in
        let r2 = relative body2.rotation anchor2 in
        let dir = unit(relative body1.rotation direction) in
        let del = (body2.position + r2) - (body1.position + r1) in
        { dir  dot  del
          dir cross del }

    # limit first dimension of constraint between jointMin, jointMax
    # and limit second dimension to exactly 0
    limit constraint { jointMin 0 } { jointMax 0 }
");
```