User-defined Constraints in Nape

Luca Deltodesco

January 21, 2012

Contents

1 Notations and results.

Vector notation: For vectors $\vec{u}, \vec{v}$ and scalars s, t:
Let $\vec{u} \cdot \vec{v}$ denote the dot-product $\vec{u} \cdot \vec{v} = u_x v_x + u_y v_y = \|\vec{u}\| \, \|\vec{v}\| \cos\theta$
Let $\vec{u} \times \vec{v}$ denote the perp-dot-product $\vec{u} \times \vec{v} = u_x v_y - u_y v_x = \|\vec{u}\| \, \|\vec{v}\| \sin\theta$
Define $[\vec{u}]_\times = \begin{bmatrix} -u_y \\ u_x \end{bmatrix}$ such that $\vec{u} \times \vec{v} = [\vec{u}]_\times \cdot \vec{v}$
noting that $[\vec{u}]_\times \cdot [\vec{v}]_\times = \vec{u} \cdot \vec{v}$ and $\left[[\vec{u}]_\times\right]_\times = -\vec{u}$
Overloading the $\times$ operator, let $s \times \vec{u} = s\,[\vec{u}]_\times$ and $\vec{u} \times s = -s \times \vec{u}$

Triple products: For vectors $\vec{u}, \vec{v}, \vec{w}$ and scalars s, t:
$\vec{u} \times (\vec{v} \times \vec{w}) = -\,(\vec{v} \times \vec{w})\,[\vec{u}]_\times \leftarrow$ a vector
$(\vec{u} \times \vec{v}) \times \vec{w} = (\vec{u} \times \vec{v})\,[\vec{w}]_\times \leftarrow$ a vector
$\vec{u} \times (s \times \vec{v}) = s\,(\vec{u} \cdot \vec{v})$
$(s \times \vec{u}) \times \vec{v} = -s\,(\vec{u} \cdot \vec{v})$

Outer products: For vectors $\vec{u}, \vec{v}$
$\vec{u}\vec{v}^\top = \vec{u} \otimes \vec{v} = \begin{bmatrix} u_x v_x & u_x v_y \\ u_y v_x & u_y v_y \end{bmatrix}$
$[\vec{u}]_\times \, [\vec{v}]_\times^\top = \vec{u} \odot \vec{v} = \begin{bmatrix} u_y v_y & -u_y v_x \\ -u_x v_y & u_x v_x \end{bmatrix}$

Derivatives: For general a,b:
$\frac{d}{dt}(a \cdot b) = \left(\frac{da}{dt} \cdot b\right) + \left(a \cdot \frac{db}{dt}\right)$
$\frac{d}{dt}(a \times b) = \left(\frac{da}{d} \times b\right) + \left(a \times \frac{db}{dt}\right)$
$\frac{d}{dt}\|a\| = \frac{1}{\|a\|}\left(a \cdot \frac{da}{dt}\right)$

Body defined by:

Position	$\vec{x}$	Velocity	$\vec{v} = \frac{d\vec{x}}{dt}$	Mass	m
Rotation	θ	Angular Velocity	$\omega = \frac{d\theta}{dt}$	Moment of Inertia	i

Body derivatives: Given a vector $\vec{u}$ defined with respect to the coordinate system of a Body as above, we have:
$\frac{d\vec{u}}{dt} = \omega \times \vec{u}$

2 UserConstraint deriviations

A (positional) constraint is defined by a function of all the bodies' positions and rotations which are collectively grouped into a block-vector like:

$\vec{x} = \begin{bmatrix} \vec{x}_1 \\ \theta_1 \\ \vdots \end{bmatrix}$ and related $\vec{v} = \frac{d\vec{x}}{dt}$ for the block-vector of velocities.

for however many bodies there are.

The positional constraint is a function $C : \mathbb{R}^{3n} \to \mathbb{R}^{\text{dimensions}}$ satisftying $C(\vec{x}) = \vec{0}$.

A PivotJoint in nape is defined on two Bodies with two locally defined anchors. These anchors are transform into 'relative' space producing the vectors $\vec{r}_1, \vec{r}_2$ and the positional constraint is:
$C(\vec{x}) = (\vec{x}_2 + \vec{r}_2) - (\vec{x}_1 + \vec{r}_1)$
Noting that the dimension of this constraint is 2.

As well as the positional constraint, nape requires the velocity constraint which is determined by $V(\vec{v}) = \frac{d}{dt}C(\vec{x})$, such that $V(\vec{v}) = \vec{0}$.

The velocity constraint for the PivotJoint in nape is:
$V(\vec{v}) = (\vec{v}_2 + \omega_2 \times \vec{r}_2) - (\vec{v}_1 + \omega_1 \times \vec{r}_1)$

Furthermore, we require (implicitly) the jacobian of the velocity constraint, formed as a block-row vector of each of the partial derivatives of V with respect to the velocities.

$\mathbf{J} = \begin{bmatrix} \frac{\partial V}{\partial \vec{v}_1} & \frac{\partial V}{\partial \omega_1} & \cdots \end{bmatrix}$. Infact the jacobian is a function of the *positions* of the objects.

Such that $V(\vec{v}) = \mathbf{J}\vec{v} + \vec{b}$ for some velocity bias $\vec{b}$ (Generally $\vec{0}$ except for motors).

The Jacobian of the PivotJoint's velocity constraint in nape is:
$\mathbf{J} = \begin{bmatrix} -\mathbf{E}_2 & -[\vec{r}_1]_\times & \mathbf{E}_2 & [\vec{r}_2]_\times \end{bmatrix} = \begin{bmatrix} -1 & 0 & r_{1y} & 1 & 0 & -r_{2y} \\ 0 & -1 & -r_{1x} & 0 & 1 & r_{2x} \end{bmatrix}$
Noting that the number of columns matches the number of velocities, and the number of rows matches the dimension of the constraint.

The Jacobian of the velocity constraint is used for two things:

Firstly, nape requires the effective mass matrix of the constraint. This long named thing is the equivalent of the usual mass matrix in the contraint space.

In the normal nape space, the 'mass matrix' $\mathbf{M}$ is a diagonal matrix composed of all the bodies' masses and moments of inertia:

$\mathbf{M} = m_1\mathbf{E}_2 \oplus i_1 \oplus \cdots = \begin{bmatrix} m_1 & & & \\ & m_1 & & \\ & & i_1 & \\ & & & \ddots \end{bmatrix}$

The effective mass matrix of the constraint is then computed as:
$\mathbf{K} = \mathbf{J}\mathbf{M}^{-1}\mathbf{J}^\top$

This sounds difficult, but there are a lot of simplifications we can make since $\mathbf{M}$ is diagonal, infact what we end up with is:

$\mathbf{K} = \frac{1}{m_1}\left(\frac{\partial V}{\partial \vec{v}_1}\right)\left(\frac{\partial V}{\partial \vec{v}_1}\right)^\top + \frac{1}{i_1}\left(\frac{\partial V}{\partial \omega_1}\right)\left(\frac{\partial V}{\partial \omega_1}\right)^\top + \cdots$

Noting that the effective mass matrix is ALWAYS symmetric. And like the jacobian is also a function of the *positions* of the objects.

Remembering the Jacobian for nape's PivotJoint:
$\mathbf{J} = \begin{bmatrix} -\mathbf{E}_2 & -[\vec{r}_1]_\times & \mathbf{E}_2 & [\vec{r}_2]_\times \end{bmatrix}$

We see that it's effective mass matrix is:
$\mathbf{K} = \frac{1}{m_1}\mathbf{E}_2 + \frac{1}{i_1}(\vec{r}_1 \odot \vec{r}_1) + \frac{1}{m_2}\mathbf{E}_2 + \frac{1}{i_2}(\vec{r}_2 \odot \vec{r}_2) = \begin{bmatrix} \frac{1}{m_1} + \frac{1}{m_2} + \frac{r_{1y}^2}{i_1} + \frac{r_{2y}^2}{i_2} & -\frac{r_{1x}r_{1y}}{i_1} - \frac{r_{2x}r_{2y}}{i_2} \\ \# & \frac{1}{m_1} + \frac{1}{m_2} + \frac{r_{1x}^2}{i_1} + \frac{r_{2x}^2}{i_2} \end{bmatrix}$

Secondly, the Jacobian is used in determining how constraint space impulses are applied to the the bodies.

Quite simply, from the constraint space impulse λ we get a world-space impulse $\mathbf{J}^\top\lambda$

For nape's PivotJoint, this looks quite simply like:

$\mathbf{J}^\top\lambda = \begin{bmatrix} -\lambda \\ -\vec{r}_1 \times \lambda \\ \lambda \\ \vec{r}_2 \times \lambda \end{bmatrix}$

And we see that the impulse received by the first body is $-\lambda$ with rotational impulse $-\vec{r}_1 \times \lambda$.
And we see that the inpulse received by the second body is λ with rotational impulse $\vec{r}_2 \times \lambda$.

3 Constructing a UserConstraint

Clearly, all that is truly needed for a basic positional constraint is the function $C(\vec{x}), V(\vec{v})$ and the jacobian **J** from which everything else can be derived.

However there is a lot of computational shortcuts that can be taken by ignoring the jacobian, and instead asking for the effective mass matrix and a method of converting the constraint space impulse into a specific body impulse which is what nape presently asks for.

These together with various other pieces of logic come together to form a User-defined constraint in nape:

```
class MyConstraint extends UserConstraint {
        public function new() {
                super(/*dimensions*/);
        }
        public override function __position(err:ARRAY<Float>):Void {
                //populate 'err' with value of C(x)
        }
        public override function __velocity(err:ARRAY<Float>):Void {
                //populate 'err' with value of V(v)
        }
        public override function __eff_mass(eff:ARRAY<Float>):Void {
                //populate 'eff' with value of K(x)
                //eff is actually compressed for symmetry of effective mass and looks like:
                //K = [eff[0], eff[1], eff[2],
                //             eff[3], eff[4],
                //       #             eff[5]] for example
        }
        public override function __impulse(imp:ARRAY<Float>,body:Body,out:Vec3):Void {
                //populate 'out' with the impulse to be applied to 'body' given
                //constraint space impulse 'imp'
                //(Note we don't actually apply the impulse here)
        }
}
```

Ontop of these functions are various others:

```
        ...
        public override function __copy():UserConstraint {
                //produce a copy of the user-defined constraint
        }
        public override function __destroy():Void {
                //if there is any extra logic that must be performed when a
                //constraint breaks it can be done here.
        }
        public override function __validate():Void {
                //perform any verifications on the integrity of the constraint
                //for instance, are all bodies defined? Are we trying to
                //simulate a constraint between static bodies which will fail?
                //
                //If there are any computations that can be re-used throughout
                //both velocity and positional iterations, they can be done here also.
        }
        public override function __prepare():Void {
                //any calculations that depend on positional values should be done
                //here to be called before velocity iterations, and before every position iteration.
        }
```

Beyond these, all that remains is to inform the constraint of what bodies it is operating over. This should be done as per the following model:

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

Like this, the constraint will act exactly like all other nape constraints.

You are free to add any other parameters to the constraint, but for a fully compliant constraint which will work in nape like any other there are some things to be noted:

Whenever a parameter of the constraint changes, you must call the invalidate() method of the UserConstraint to inform nape of the change. This is to make sure that the constraint is awoken should it be sleeping.

You should do this internally, so that to the outside the Constraint acts like any other nape constraint (which do not have a need for an invalidate method).

For example, for simple datatypes like Float/Int you can follow this model:

```
...
public var parameter(default,set_parameter):Type;
function set_parameter(parameter:Type) {
        if(this.parameter!=parameter) invalidate();
        return this.parameter = parameter;
}
```

For a complex data type, it is up to you to ensure that any invalidation of that data type produces a call to the constraint invalidate() method (through setters/function handlers etc).

For the specific Vec2 nape type which will be most common for such things as anchors and direction vectors etc, you can follow this model:

```
...
public var vec2param(default,set_vec2param):Vec2;
function set_vec2param(vec2param:Vec2) {
        if(this.vec2param==null) this.vec2param = bindVec2();
        return this.vec2param.set(vec2param);
}
```

Again, this will ensure a consistent API with respect to all other constraints where assignment of a Vec2 value like throughout all of nape consists of a copy-by-value and not copy-by-reference. Instantiating the parameter with *bindVec2* handles invalidation of the constraint when it's x/y values are modified automaticaly.

3.1 Example:

Here is a re-implementation of nape's PivotJoint, but as a custom UserConstraint. The result is a PivotJoint that behaves almost exactly like the internal nape PivotJoint (The differences are of performance, and that the internal nape PivotJoint has extra logic to deal with large positional errors which is not implemented here).

```
typedef ARRAY<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:Body,body2:Body,anchor1:Vec2,anchor2:Vec2) {
                super(2);
                this.body1 = body1;
                this.body2 = body2;
                this.anchor1 = anchor1;
                this.anchor2 = anchor2;
        }

        public override function __copy():UserConstraint {
                return new UserPivotJoint(body1,body2,anchor1,anchor2);
        }

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

        var rel1:Vec2;
        var rel2:Vec2;
        public override function __prepare() {
                rel1 = body1.localToRelative(anchor1);
                rel2 = body2.localToRelative(anchor2);
        }

        public override function __position(err:ARRAY<Float>) {
                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:ARRAY<Float>) {
                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.x + rel1.x*v1.z);
        }
        public override function __eff_mass(eff:ARRAY<Float>) {
                var m1 = body1.constraintMass; var m2 = body2.constraintMass;
                var i1 = body1.constraintInertia; 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:ARRAY<Float>,body:Body,out:Vec3) {
                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(new Vec2(imp[0],imp[1]));
        }
}
```

4 Moar!!

You can additionaly define velocity-only constraints like the MotorJoint in nape. To do this you simply do not implement the __position method and pass TRUE as a second argument to the super constructor call.

You can also define inequality constraints. For this purpose there is one more additional function that can be overriden not listed about which is:

```
        ...
        public override function __clamp(jAcc:ARRAY<Float>) {
        }
```

Which should perform any necessary clamping of the accumulated impulse to handle inequalities in the constraint. For instance in the internal penetration contact constraint of nape, the inequality means that we clamp the one-dimensional jAcc so that it is never negative (Contact only pushes, never pulls).

Lastly, you can also define a way for the Userconstraint to draw itself in Debug drawing

```
        ...
        public override function __draw(debug:Debug) {
        }
```