Kinematics of moving frames in Chrono::Engine

Alessandro Tasora
alessandro.tasora@unipr.it

March 31, 2016

Abstract

The present work introduces a custom $\mathcal{T}(\mathbb{T}, \succ)$ algebra which is used to express kinematic transformations in the **Chrono**::Engine programming interface. Floating frames are considered C++ objects that carry data about position, rotation, speed and acceleration as a whole, and define an algebraic structure: the $\mathsf{T}(\mathcal{T}, \succ)$ group. In this work we present the main properties of such group and then we show how these features are implemented in the **Chrono**::Engine library using C++ operator overloading, so that coordinate transformations can be performed with a very compact syntax.

1. Introduction

One of the most interesting features of C++ and similar languages, is the ability of overloading operators between objects, so that common mathematical operators such as $+$, $-$, $*$, $/$, etc., can be customized in a sense that they can operate over complex objects. This led us to implement a system in **Chrono**::Engine where the operator-overloading capabilities of the C++ language are used to create an algebra between objects that represent moving references in three dimensional space.

Theoretical, applied and computational mechanics often require coordinate transformations, for instance one might need to compute the absolute position of points given their relative position respect to rigid frames which move in space [3]. In this case, the transformation can be expressed as an affine map using matrix algebra or similar formalisms.

Additionally, considerthe case of a kinematic chain of moving frames, where the position of each frame is known respect to the previous frame in the chain: the absolute position of the end of the chain can be expressed as a sequence of affine transformations. This case is often met in robotics, multibody simulation,

kinematics [9]. In literature a common way to express this kind of consecutive transformations is the Denavit-Hartenberg approach [6], where 4x4 matrices are used to express rotations and translations with a single matrix multiplication; these matrices are multiplied to express sequences of coordinate transformations as in robotic arms.

The algebra implemented in **Chrono**::Engine , instead, includes also speeds and accelerations: in fact if in a chain of frames also the relative speed and relative acceleration of each frame is known respect to the previous frame in the chain, an algorithm might also find the absolute position, speed and acceleration of the end of the chain.

We define a $\succ$ operator such that a sequence of $\succ$ operations corresponds to a chain of coordinate transformations. An example can explain this: consider $\chi_{i,(j)}$ as a data that represent the position, speed and acceleration of the frame i respect to coordinate j; therefore in the example of Fig.1 the sequence of transformations can be written as:

$$\chi_{c,(a)} = \chi_{c,(b)} \succ \chi_{b,(a)}. \tag{1}$$

Also, one can use the previous transformation to get $\mathbf{r}_{c,(a)}$, that is the absolute position of the origin of 3 respect to the absolute coordinate system 0, since

$$\chi_{c,(a)} = \{\mathbf{r}_{c,(a)}, \mathbf{q}_{c,(a)}, \dot{\mathbf{r}}_{c,(a)}, \omega_{c,(a)}, \ddot{\mathbf{r}}_{c,(a)}, \alpha_{c,(a)}\}.$$

The interesting fact is that in **Chrono**::Engine the elements $\chi_{i,(j)}$ are C++ objects of the `chrono::ChFrameMoving` class, and the $\succ$ operator becomes the `>>` operator. We recall that such approach also includes speed and acceleration transformations in a single operation. For instance, the absolute speed of c is also contained in $\chi_{c,(a)}$, and depends on angular velocities and speeds of all other systems in the chain.

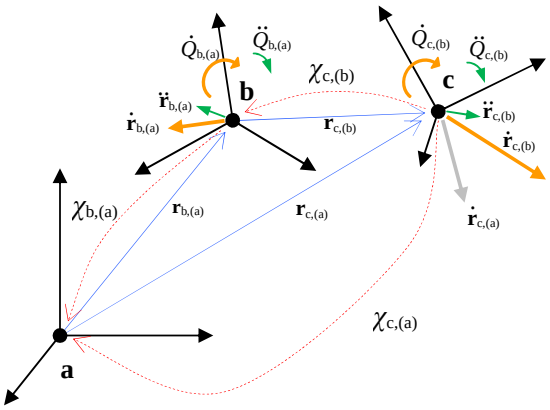

Figure 1: Example of chained transformation

2. The $\mathcal{T}(\mathbb{T}, \succ)$ algebra

It is known [7] [4] that the so called special Euclidean group of rigid transformations in three dimensional space, $\mathsf{SE}(3)$, is the semi-direct product of the two Lie soubgroups: $\mathsf{SO}(3)$, the Lie group of isometries with fixed point [5], and $\mathbb{R}^3$, for translations; that is $\mathsf{SE}(3) = \mathsf{SO}(3) \rtimes \mathbb{R}^3$.

Aiming at a group algebra that succintly transforms also velocities and accelerations, we introduce the group $\mathcal{T}^{\mathsf{SO}}(\mathbb{T}^{\mathsf{SO}}, \succ)$ whose non-abelian operator $\succ$ acts over a 18-dimensional manifold

$$\mathbb{T}^{\mathsf{SO}} = \{\mathbb{R}^3 \rtimes \mathsf{SO}(3) \rtimes \mathbb{R}^6 \rtimes \mathbb{R}^6\}.$$

The $\succ$ operator has arity $\bar{\alpha} = 2$.

This algebra will be used to manage coordinate transformations: each element $\chi \in \mathbb{T}^{\mathsf{SO}}$ can represent a frame in three-dimensional space, including position, rotation information, as well as speed and acceleration information.

Here we remark that, although rotations could be parametrized with a set of three angles, it is well known that, given the topology $\mathsf{SO}(3)$, this could cause problems of singularities because $\mathbb{R}^3$ is not omeomorphic to $\mathsf{SO}(3)$. Therefore we parametrize $\mathsf{SO}(3)$ with unitary quaternions $\mathbf{q} \in \mathsf{S}^3$, i.e. $\mathbf{q} \in \mathbb{H}_1$, whose adoption in the context of computer simulation proved to be very reliable and practical. The only drawback is that S^3 is not exactly isomorphic to $\mathsf{SO}(3)$, being its double cover. Yet this is not a big issue (the only drawback is that there are two distinct quaternions per a single rotation), so we can rather work with an epimorphism of $\mathcal{T}^{\mathsf{SO}}(\mathbb{T}^{\mathsf{SO}}, \succ)$, that is $\mathcal{T}(\mathbb{T}, \succ)$, using quaternions:

$$\mathbb{T} = \{\mathbb{R}^3 \rtimes \mathsf{S}^3 \rtimes \mathbb{R}^6 \rtimes \mathbb{R}^6\}.$$

The speed of the origin of the frame is denoted with $\dot{\mathbf{r}} \in \mathbb{R}^3$, its angular speed is denoted with $\omega \in \mathbb{R}^3$ (expressed in the base of the moving frame). Similarly we denote acceleration with $\ddot{\mathbf{r}}$ and angular acceleration with α, the latter being expressed in the local base of the moving frame. Thus, we define the vector $\chi \in \mathbb{T}$ for $\mathbf{r} \in \mathbb{R}^3$, $\mathbf{q} \in \mathsf{S}^3$, $\dot{\mathbf{r}}, \omega, \ddot{\mathbf{r}}, \alpha \in \mathbb{R}^3$, as :

$$\chi = \{\mathbf{r}, \mathbf{q}, \dot{\mathbf{r}}, \omega, \ddot{\mathbf{r}}, \alpha\}.$$

From now on, assuming that position, rotation, speed and acceleration of a frame a in χ_a are expressed relative to another frame b, we will use the notation $\chi_{a,(b)}$. Also, we will denote the frame b as the *parent frame* of a or, similarly, a as the *child frame* of b. The definition of the $\succ$ operation stems from the requirement that

$$\chi_{a,(c)} = \begin{Bmatrix} \mathbf{r}_{a,(c)} \\ \mathbf{q}_{a,(c)} \\ \dot{\mathbf{r}}_{a,(c)} \\ \omega_{a,(c)} \\ \ddot{\mathbf{r}}_{a,(c)} \\ \alpha_{a,(c)} \end{Bmatrix} = \chi_{a,(b)} \succ \begin{Bmatrix} \mathbf{r}_{a,(b)} \\ \mathbf{q}_{a,(b)} \\ \dot{\mathbf{r}}_{a,(b)} \\ \omega_{a,(b)} \\ \ddot{\mathbf{r}}_{a,(b)} \\ \alpha_{a,(b)} \end{Bmatrix} \succ \chi_{b,(c)} \begin{Bmatrix} \mathbf{r}_{b,(c)} \\ \mathbf{q}_{b,(c)} \\ \dot{\mathbf{r}}_{b,(c)} \\ \omega_{b,(c)} \\ \ddot{\mathbf{r}}_{b,(c)} \\ \alpha_{b,(c)} \end{Bmatrix} \tag{2}$$

By applying expressions for kinematic transformations in three dimensional space, in the following subsections we develop the expressions for the terms in Eq.(2).

Product: position and rotation part

As known [8], the morphism

$$\mathbf{q}_r^o = \mathbf{q}_e \mathbf{q}_r \mathbf{q}_e^*. \tag{3}$$

is $||\cdot||_2$-norm preserving and rotates a quaternion $\mathbf{q}_r \in \mathbb{H}$ by means of an unimodular quaternion $\mathbf{q}_e \in \mathsf{S}^3$ and its conjugate $\mathbf{q}_e^*$.

Expression (3) can be used to rotate points in space, with rotation represented in form of Euler parameters $\mathbf{q}_e$ and $\mathbf{q}_r$ as a so-called *pure* quaternion with imaginary part as the cartesian coordinates of a point:

$$\mathbf{q}_r = \Im(\mathbf{r}) \equiv \{0, r_x, r_y, r_z\}.$$

Therefore, the affine transformation of the point $\mathbf{r}_{a,(b)}$ after rotation $\mathbf{q}_{b,(c)}$ and translation $\mathbf{r}_{b,(c)}$ can be expressed as:

$$\Im(\mathbf{r}_{a,(c)}) = \Im(\mathbf{r}_{b,(c)}) + \mathbf{q}_{b,(c)} \Im(\mathbf{r}_{a,(b)}) \mathbf{q}_{b,(c)}^*. \tag{4}$$

This Eq.(4) has a counterpart in linear algebra:

$$\mathbf{r}_{a,(c)} = \mathbf{r}_{b,(c)} + [A(\mathbf{q}_{b,(c)})] \mathbf{r}_{a,(b)}. \tag{5}$$

where we introduced the rotation matrix $[A(\mathbf{q}_{b,(c)})]$ function of a quaternion $\mathbf{q}_{b,(c)}$ as described, for instance, in [8].

The term $\mathbf{q}_{a,(c)}$, representing the rotation of the reference a respect to the reference c, can be easily obtained with a single quaternion product:

$$\mathbf{q}_{a,(c)} = \mathbf{q}_{b,(c)} \mathbf{q}_{a,(b)} \tag{6}$$

as can be seen applying the affine map (4) twice, for transforming a point p

from reference a to reference b and from reference b to reference c:

$$\begin{aligned}\Im(\mathbf{r}_{p,(c)}) =&\Im(\mathbf{r}_{b,(c)}) + \mathbf{q}_{b,(c)}\left(\Im(\mathbf{r}_{a,(b)})+\right.\\ &\left.\mathbf{q}_{a,(b)}\Im(\mathbf{r}_{p,(a)})\mathbf{q}^*_{a,(b)}\right)\mathbf{q}^*_{b,(c)}\\ =&\Im(\mathbf{r}_{b,(c)}) + \mathbf{q}_{b,(c)}\Im(\mathbf{r}_{a,(b)})\mathbf{q}^*_{b,(c)}+\\ &\mathbf{q}_{b,(c)}\mathbf{q}_{a,(b)}\Im(\mathbf{r}_{p,(a)})\mathbf{q}^*_{a,(b)}\mathbf{q}^*_{b,(c)}\\ =&\Im(\mathbf{r}_{a,(c)}) + \mathbf{q}_{a,(c)}\Im(\mathbf{r}_{p,(a)})\mathbf{q}^*_{a,(c)}.\end{aligned}$$

Product: the velocity part

Speed terms $\dot{\mathbf{r}}_{a,(c)}$ and $\omega_{a,(c)}$ can be obtained from symbolic differentiation of Eq.(4) and Eq.(6). By applying the chain rule to the differentiation of the affine map of Eq.(4):

$$\begin{aligned}\dot{\Im}(\mathbf{r}_{a,(c)}) =&\dot{\Im}(\mathbf{r}_{b,(c)}) + \dot{\mathbf{q}}_{b,(c)}\Im(\mathbf{r}_{a,(b)})\mathbf{q}^*_{b,(c)}+\\ &\mathbf{q}_{b,(c)}\dot{\Im}(\mathbf{r}_{a,(b)})\mathbf{q}^*_{b,(c)} + \mathbf{q}_{b,(c)}\Im(\mathbf{r}_{a,(b)})\dot{\mathbf{q}}^*_{b,(c)}\\ \dot{\Im}(\mathbf{r}_{a,(c)}) =&\dot{\Im}(\mathbf{r}_{b,(c)}) + 2\dot{\mathbf{q}}_{b,(c)}\Im(\mathbf{r}_{a,(b)})\mathbf{q}^*_{b,(c)}+\\ &\mathbf{q}_{b,(c)}\dot{\Im}(\mathbf{r}_{a,(b)})\mathbf{q}^*_{b,(c)}\end{aligned}$$

We note that $\dot{\Im}(\mathbf{r}) = \Im(\dot{\mathbf{r}})$, hence we get:

$$\begin{aligned}\Im(\dot{\mathbf{r}}_{a,(c)}) =&\Im(\dot{\mathbf{r}}_{b,(c)}) + 2\dot{\mathbf{q}}_{b,(c)}\Im(\mathbf{r}_{a,(b)})\mathbf{q}^*_{b,(c)}+\\ &\mathbf{q}_{b,(c)}\Im(\dot{\mathbf{r}}_{a,(b)})\mathbf{q}^*_{b,(c)}.\end{aligned} \tag{7}$$

Similarly, one can perform the time derivative of the expression of Eq.(6):

$$\dot{\mathbf{q}}_{a,(c)} = \dot{\mathbf{q}}_{b,(c)}\mathbf{q}_{a,(b)} + \mathbf{q}_{b,(c)}\dot{\mathbf{q}}_{a,(b)}. \tag{8}$$

The angular velocity $\omega_{a,(c)}$ of $\chi_{a,(c)}$, expressed in the coordinates of reference a, follows immediately the quaternion derivative $\dot{\mathbf{q}}_{a,(c)}$ obtained with Eq.(8) using the following formula, discussed in [8]:

$$\Im(\omega_{a,(c)}) = 2\mathbf{q}^*_{a,(c)}\dot{\mathbf{q}}_{a,(c)}. \tag{9}$$

Product: the acceleration part

The last two parts of the $\chi_{a,(c)}$ vector are the acceleration $\ddot{\mathbf{r}}_{a,(c)}$ and the angular acceleration $\alpha_{a,(c)}$.

By differentiation of Eq.(7):

$$\begin{aligned}\ddot{\Im}(\mathbf{r}_{a,(c)}) =&\ddot{\Im}(\mathbf{r}_{b,(c)})+\\ &2\ddot{\mathbf{q}}_{b,(c)}\Im(\mathbf{r}_{a,(b)})\mathbf{q}^*_{b,(c)}+\\ &2\dot{\mathbf{q}}_{b,(c)}\dot{\Im}(\mathbf{r}_{a,(b)})\mathbf{q}^*_{b,(c)}+\\ &2\dot{\mathbf{q}}_{b,(c)}\Im(\mathbf{r}_{a,(b)})\dot{\mathbf{q}}^*_{b,(c)}+\\ &+\dot{\mathbf{q}}_{b,(c)}\dot{\Im}(\mathbf{r}_{a,(b)})\mathbf{q}^*_{b,(c)}+\\ &\mathbf{q}_{b,(c)}\ddot{\Im}(\mathbf{r}_{a,(b)})\mathbf{q}^*_{b,(c)}+\\ &\mathbf{q}_{b,(c)}\dot{\Im}(\mathbf{r}_{a,(b)})\dot{\mathbf{q}}^*_{b,(c)}\\ =&\ddot{\Im}(\mathbf{r}_{b,(c)})+\\ &2\ddot{\mathbf{q}}_{b,(c)}\Im(\mathbf{r}_{a,(b)})\mathbf{q}^*_{b,(c)}+\\ &4\dot{\mathbf{q}}_{b,(c)}\dot{\Im}(\mathbf{r}_{a,(b)})\mathbf{q}^*_{b,(c)}+\\ &+2\dot{\mathbf{q}}_{b,(c)}\Im(\mathbf{r}_{a,(b)})\dot{\mathbf{q}}^*_{b,(c)}+\\ &\mathbf{q}_{b,(c)}\ddot{\Im}(\mathbf{r}_{a,(b)})\mathbf{q}^*_{b,(c)}\end{aligned}$$

and finally, as $\ddot{\Im}(\mathbf{r}_{a,(c)}) = \Im(\ddot{\mathbf{r}}_{a,(c)})$, it is:

$$\begin{aligned}\Im(\ddot{\mathbf{r}}_{a,(c)}) =&\Im(\ddot{\mathbf{r}}_{b,(c)}) + 2\ddot{\mathbf{q}}_{b,(c)}\Im(\mathbf{r}_{a,(b)})\mathbf{q}^*_{b,(c)}+\\ &4\dot{\mathbf{q}}_{b,(c)}\Im(\dot{\mathbf{r}}_{a,(b)})\mathbf{q}^*_{b,(c)}+\\ &+2\dot{\mathbf{q}}_{b,(c)}\Im(\mathbf{r}_{a,(b)})\dot{\mathbf{q}}^*_{b,(c)}+\\ &\mathbf{q}_{b,(c)}\Im(\ddot{\mathbf{r}}_{a,(b)})\mathbf{q}^*_{b,(c)}\end{aligned} \tag{10}$$

Interms of quaternion derivative, angular acceleration follow from the differentiation of the epression of Eq.(8):

$$\begin{aligned}\ddot{\mathbf{q}}_{a,(c)} =&\ddot{\mathbf{q}}_{b,(c)}\mathbf{q}_{a,(b)} + \dot{\mathbf{q}}_{b,(c)}\dot{\mathbf{q}}_{a,(b)}+\\ &\dot{\mathbf{q}}_{b,(c)}\dot{\mathbf{q}}_{a,(b)} + \mathbf{q}_{b,(c)}\ddot{\mathbf{q}}_{a,(b)}\\ =&\ddot{\mathbf{q}}_{b,(c)}\mathbf{q}_{a,(b)} + 2\dot{\mathbf{q}}_{b,(c)}\dot{\mathbf{q}}_{a,(b)} + \mathbf{q}_{b,(c)}\ddot{\mathbf{q}}_{a,(b)}\end{aligned} \tag{11}$$

As a vector, the classical angular acceleration $\alpha_{a,(c)}$ of $\chi_{a,(c)}$, expressed in the coordinates of reference a, is obtained from the quaternion $\ddot{\mathbf{q}}_{a,(c)}$ of Eq.(11) and from the differentiation of Eq.(9):

$$\Im(\alpha_{a,(c)}) = 2\dot{\mathbf{q}}^*_{a,(c)}\dot{\mathbf{q}}_{a,(c)} + 2\mathbf{q}^*_{a,(c)}\ddot{\mathbf{q}}_{a,(c)} \tag{12}$$

3. Properties of the $\mathcal{T}(\mathbb{T}, \succ)$ algebra

This section presents some interesting properties of the $\mathcal{T}(\mathbb{T}, \succ)$ algebra.

- The manifold $\mathbb{T}$ is a topological space,

- The entire $\mathbb{T}$ is a double cover of $\mathbb{T}^{\mathsf{SO}} = \{\mathbb{R}^3 \rtimes \mathsf{SO}(3) \rtimes \mathbb{R}^{12}\}$ because of the epimorphism $r : \mathsf{S}^3 \to \mathsf{SO}(3, \mathbb{R})$. Also, it is isomorphic to $\mathbb{T}^{\mathsf{SU}} = \{\mathbb{R}^3 \rtimes \mathsf{SU}(2) \rtimes \mathbb{R}^{12}\}$.
- The $\mathcal{T}(\mathbb{T}, \succ)$ algebra is a group with $\dim_{/\mathbb{R}} = 18$, and it has the properties of closure, associativity, identity, existence of inverse element.

Here we demonstrate that $\mathcal{T}(\mathbb{T}, \succ)$ has a neutral element and an inverse element.

THEOREM 1

The $\mathcal{T}(\mathbb{T}, \succ)$ algebra has an identity element $\chi_I \in \mathbb{T}$.

Proof. Let consider an element $\chi_I \in \mathbb{T}$ as $\chi_I = \{\mathbf{0}, \mathbf{q}_I, \mathbf{0}, \mathbf{0}, \mathbf{0}, \mathbf{0}\}$, where $\mathbf{q}_I$ is the unitary quaternion $1 + 0i + 0j + 0k$.

The product $\chi_a = \chi_b \succ \chi_I$ gives $\chi_a = \{\mathbf{r}_a, \mathbf{q}_a, \dot{\mathbf{r}}_a, \omega_a, \ddot{\mathbf{r}}, \alpha_a\}$. According to the definition of Eq.(4), the position term can be made explicit as:

$$\Im(\mathbf{r}_a) = \Im(\mathbf{0}) + \mathbf{q}_I \Im(\mathbf{r}_b) \mathbf{q}_I^*.$$

Since $\mathbf{q}_I \Im(\mathbf{r}_b) \mathbf{q}_I^* = \Im(\mathbf{r}_b)$ by property of quaternion multiplication, it is also

$$\Im(\mathbf{r}_a) = \Im(\mathbf{r}_b) \tag{13}$$

Also, Eq.(6) becomes

$$\mathbf{q}_a = \mathbf{q}_I \mathbf{q}_b = \mathbf{q}_b. \tag{14}$$

Now, expressing the speed and acceleration terms using definitions (7, 8), 10, 11): and using the properties of quaternion multiplications, it simplifies to:

$$\begin{aligned}
\Im(\dot{\mathbf{r}}_a) =& \Im(\mathbf{0}) + 2(\mathbf{0}\Im(\mathbf{r}_b)\mathbf{q}_I^*) + \\
& \mathbf{q}_I \Im(\dot{\mathbf{r}}_b) \mathbf{q}_I^* = \Im(\dot{\mathbf{r}}_b) && (15)\\
\dot{\mathbf{q}}_a =& \mathbf{0}\mathbf{q}_b + \mathbf{q}_I \dot{\mathbf{q}}_b = \dot{\mathbf{q}}_b && (16)\\
\Im(\ddot{\mathbf{r}}_a) =& \mathbf{0} + 2(\mathbf{0}\Im(\mathbf{r}_b)\mathbf{q}_I^*) + \\
& 4(\mathbf{0}\Im(\dot{\mathbf{r}}_b)\mathbf{q}_I^*) + 2(\mathbf{0}\Im(\mathbf{r}_b)\mathbf{0}^*) + \\
& \mathbf{q}_I \Im(\ddot{\mathbf{r}}_b) \mathbf{q}_I^* = \Im(\ddot{\mathbf{r}}_b) && (17)\\
\ddot{\mathbf{q}}_a =& \mathbf{0}\mathbf{q}_b + 2(\mathbf{0}\dot{\mathbf{q}}_b) + \mathbf{q}_I \ddot{\mathbf{q}}_b = \ddot{\mathbf{q}}_b. && (18)
\end{aligned}$$

Given Eq.(13-18) and properties (9),(12), one gets $\chi_a = \chi_b \succ \chi_I = \chi_b$. Thus $\succ \chi_I$ is the right-neutral element of the $\mathcal{T}(\mathbb{T}, \succ)$ algebra.

One can demonstrate in a similar fashion that χ_I is also a left-neutral element in the $\mathcal{T}(\mathbb{T}, \succ)$ algebra.

Give that this magma structure has both right and left-neutral elements, $\mathcal{T}(\mathbb{T}, \succ)$ is also a monoid with $\chi_b \succ \chi_I = \chi_I \succ \chi_b = \chi_b$. QED.

The commutative property does not hold, and this is shown in the following theorem that also provides the explicit expression for the inverse element.

THEOREM 2

The $\mathcal{T}(\mathbb{T}, \succ)$ algebra is a non-abelian group.

Proof.

For the $\mathcal{T}(\mathbb{T}, \succ)$ monoid algebra to be a group, each element χ must have an inverse element χ^{-1} such that $\chi^{-1} \succ \chi = \chi_I$, where χ_I is the neutral elment introduced in Theorem 1.

Recall the product in Eq.(2). If $\chi_{a,(b)} \succ \chi_{b,(c)} = \chi_I$, then $\chi_{a,(b)}$ is the left-inverse of $\chi_{b,(c)}$, and will be denoted as $\chi^{-1L}_{b,(c)}$. Also, $\chi_{b,(c)}$ is the right-inverse of $\chi_{a,(b)}$, and will be denoted as $\chi^{-1R}_{a,(b)}$. If right and left inverses exist,

$$\chi_{a,(b)} \succ \chi^{-1R}_{a,(b)} = \chi_I, \quad \chi^{-1L}_{b,(c)} \succ \chi_{b,(c)} = \chi_I$$

Setting $\chi_{a,(c)} = \chi_I$ in the product of Eq.(2), one can write $\chi_{a,(b)} \succ \chi_{b,(c)} = \chi_I$, then it will be possible to manipulate the definitions in Eq.(4-11) to find the expression of the left-inverse by explicitating the terms belonging to $\chi_{a,(b)}$.

Focusing on transformation of positions, we rewrite Eq.(4) as:

$$\Im(\mathbf{r}_{b,(c)}) + \mathbf{q}_{b,(c)}\Im(\mathbf{r}_{a,(b)})\mathbf{q}^*_{b,(c)} = \Im(\mathbf{0}).$$

By left-multiplying all terms by quaternion $\mathbf{q}^{-1}_{b,(c)}$ and right-multiplying all terms by $\mathbf{q}^{*-1}_{b,(c)}$, and remembering quaternion algebra properties $\mathbf{q}\mathbf{q}^{-1} = \{1,0,0,0\}$ and $\mathbf{q}\{1,0,0,0\} = \mathbf{q}$, $\mathbf{q} \in \mathbb{H}_1$, it is easy to find:

$$\Im(\mathbf{r}_{a,(b)}) = -\mathbf{q}^{-1}_{b,(c)}\Im(\mathbf{r}_{b,(c)})\mathbf{q}^{*-1}_{b,(c)}. \tag{19}$$

This is the first element of the left-inverse vector, that in our proof is $\chi_{a,(b)} = \chi^{-1L}_{b,(c)}$.

For rotations, remember that the rotation part $\mathbf{q}$ in the neutral element χ_I is the unit quaternion $\{1,0,0,0\}$ as demonstrated in Theorem 1, and rewrite Eq.(6) as follows:

$$\mathbf{q}_{b,(c)}\mathbf{q}_{a,(b)} = \{1,0,0,0\}.$$

So, using quaternion inverses, multiplying by $\mathbf{q}^{-1}_{b,(c)}$, one obtains the rotational part of the left-inverse:

$$\mathbf{q}_{a,(b)} = \mathbf{q}^{-1}_{b,(c)}. \tag{20}$$

Set Eq.(7) to zero for computing the speed part of the inverse:

$$\begin{aligned}&\Im(\dot{\mathbf{r}}_{b,(c)}) + 2\dot{\mathbf{q}}_{b,(c)}\Im(\mathbf{r}_{a,(b)})\mathbf{q}^*_{b,(c)} + \\ &\mathbf{q}_{b,(c)}\Im(\dot{\mathbf{r}}_{a,(b)})\mathbf{q}^*_{b,(c)} = \Im(\mathbf{0})\end{aligned}$$

After some algebra:

$$\begin{aligned}\Im(\dot{\mathbf{r}}_{a,(b)}) =& \mathbf{q}^{-1}_{b,(c)}(-\Im(\dot{\mathbf{r}}_{b,(c)}) + \\ & 2\dot{\mathbf{q}}_{b,(c)}\mathbf{q}^{-1}_{b,(c)}\Im(\mathbf{r}_{b,(c)}))\mathbf{q}^{*-1}_{b,(c)}.\end{aligned} \tag{21}$$

Requiring that $\dot{\mathbf{q}}_{a,(c)}$ is null in Eq.(8), it follows:

$$\dot{\mathbf{q}}_{a,(b)} = -\mathbf{q}_{b,(c)}^{-1}\dot{\mathbf{q}}_{b,(c)}\mathbf{q}_{b,(c)}^{-1}. \tag{22}$$

Using other simplifications, recalling $\mathbf{q}^{*-1}\mathbf{q}^* = \{1,0,0,0\}$, one finally gets the acceleration part of the left inverse, obtaining:

$$\begin{aligned}\Im(\ddot{\mathbf{r}}_{a,(b)}) =& \mathbf{q}_{b,(c)}^{-1}[-\Im(\ddot{\mathbf{r}}_{b,(c)}) + 2\ddot{\mathbf{q}}_{b,(c)}\mathbf{q}_{b,(c)}^{-1}\Im(\mathbf{r}_{b,(c)}) \\ &+ 4\dot{\mathbf{q}}_{b,(c)}\mathbf{q}_{b,(c)}^{-1}(\Im(\dot{\mathbf{r}}_{b,(c)}) \\ &- 2\dot{\mathbf{q}}_{b,(c)}\mathbf{q}_{b,(c)}^{-1}\Im(\mathbf{r}_{b,(c)})) \\ &+ 2\dot{\mathbf{q}}_{b,(c)}\mathbf{q}_{b,(c)}^{-1}\Im(\mathbf{r}_{b,(c)})\mathbf{q}_{b,(c)}^{-1}\dot{\mathbf{q}}_{b,(c)}^{*}]\mathbf{q}_{b,(c)}^{*-1}\end{aligned}$$

and

$$\ddot{\mathbf{q}}_{a,(b)} = \mathbf{q}_{b,(c)}^{-1}(\ddot{\mathbf{q}}_{b,(c)} - 2\dot{\mathbf{q}}_{b,(c)}\mathbf{q}_{b,(c)}^{-1}\dot{\mathbf{q}}_{b,(c)})\mathbf{q}_{b,(c)}^{-1}. \tag{23}$$

Finally, using Eq.(9) and Eq.(12), substitute Eq.(19-23) into $\chi_{a,(b)}$, that is the left-inverse $\chi_{b,(c)}^{-1L}$ which satisfies $\chi_{a,(b)} \succ \chi_{b,(c)} = \chi_{b,(c)}^{-1L} \succ \chi_{b,(c)} = \chi_I$.

Similarly, we could solve $\chi_{a,(b)} \succ \chi_{b,(c)} = \chi_I$ for $\chi_{b,(c)}$: after long symbolic manipulation (not reported in these pages in sake of compactness) we would obtain the same results of Eq.(19-23), but with inverted subscripts, that is with $a,(b)$ swapped with $b,(c)$.

Hence we built the right-inverse $\chi_{a,(b)}^{-1R}$ and we conclude that, for a generic element $\chi \in \mathbb{T}$, right- and left-inverses are the same, that is $\chi^{-1R} = \chi^{-1L} = \chi^{-1}$. Given the existence of the inverse, the algebra is a group.

Note that the group is non-abelian since the $\succ$ operation is **non commutative**. QED.

4. Software implementation

The $\mathcal{T}(\mathbb{T}, \succ)$ algebra is implemented in C++ in the **Chrono**::Engine API using operator overloading and object-oriented programming. In our approach, $\chi \in \mathsf{T}$ elements are represented by objects inherited from the `chrono::ChFrameMoving`.

Moreover, a subset of the methods exposed here has been implemented in a simpler C++ parent class called `chrono::ChFrame`, that deals only with the translation and rotation and carries no information about velocities and accelerations. In this way, depending on the problem type, the developer can choose when using a fast `chrono::ChFrame` object or a more powerful but slower `chrono::ChFrameMoving`.

Those classes are templated, so one can have `ChFrameMoving<double>`, or the less precise `ChFrameMoving<float>`, and by default `ChFrameMoving<>` uses `double`.

For these classes, we overloaded C++ binary and unary operators in order to express the Lie group products by a simple syntax. We recall that the usual way of expressing transformations in kinematics is by using a right-to-left concatenation, as expressed in $\chi_{a,(c)} = \chi_{b,(c)} \cdot \chi_{a,(b)}$, but here we also introduced an alternative operator, denoted with $\succ$, for expressing the same result with a left-to-right concatenation: $\chi_{a,(c)} = \chi_{a,(b)} \succ \chi_{b,(c)}$. Both could correspond to overloaded operators in C++; for instance in our implementation they are, respectively, `*` and `>>`.

The following operators have been implemented:

- The $\cdot$ right-to-left transformation is mapped to the `*` operator, so that $\chi_c = \chi_a \cdot \chi_b$ is becomes `c = a * b`.
- The $\succ$ left-to-right transformation is mapped to the `>>` operator, so that $\chi_c = \chi_a \succ \chi_b$ becomes `c = a >> b`.
- The `*=` on-place operator has been used for an efficient implementation of self right-multiplication, so $\chi_a := \chi_a \cdot \chi_b$ becomes `a*=b`,
- The `>>=` on-place operator has been used for an efficient implementation of self left-multiplication, so $\chi_a := \chi_a \succ \chi_b$ becomes `a>>=b`,
- The inversion χ^{-1} is implemented in C++ language by redefining the `!` unary operator,
- The vector-by-frame *heterogeneous* binary operators are implemented, for example as in `v_a = v_b >> frame`, or `v_a = frame * v_b`.

A synoptic table with a schematic view of the most important operators is shown in Table 1.

The last point is also one of the motivation that lead us to implement the `>>` left-to-right transformation as an alternative to the usual right-to-left transformation. In fact, if one must transform a simple point through two (or more) frames in sequence using a single line one could write

```
v_a = frame2 * frame1 * v_b
```

as well as

```
v_a = v_b >> frame1 >> frame2
```

but in the first case, if no parentheses are used, most C++ compilers[1] would start parsing the formula from the left, creating a temporary frame for the result of `frame2 * frame1`, and then they will perform the frame-by-vector operation; whereas in the second case the temporary object would be a vector, that is computationally more efficient.

Nonetheless, in sake of completeness, we also implemented the `*` operator if users prefer to use the right-to-left ordering.

[1] This behavior is not requested by the ISO standard, but it happens most times.

We took the design decision to work with objects where angular speeds and angular accelerations are represented directly with quaternions $\dot{\mathbf{q}}$ and $\ddot{\mathbf{q}}$ rather than with 3D vectors ω and α. Thank to the encapsulation paradigm of OOP, this design does not affect the way the programmer interacts with the data, because custom functions can provide ω and α only when requested, by evaluating Eq.(9) and Eq.(12). Viceversa, the user can provide ω or α, and these are instantly converted to $\dot{\mathbf{q}}$ and $\ddot{\mathbf{q}}$ using inverse formulas.

Since we used this software library in many engineering projects, we were able to make a statistical analysis and we found that, in most cases, an object of `chrono::ChFrameMoving` class is used simply to transform 3D points, and only in few cases one is interested also in speeds and accelerations. This means that the most important function is the one in Eq.(4), which we implemented in different flavours for optimal execution speed. For example, if a single object of `chrono::ChFrameMoving` type must transform many 3D vectors at once, we can use Eq.(5), which is a bit faster than Eq.(4) because rotation by matrix-vector multiplication (after the matrix has been initialized once, with the nine values) takes less time than computing the quaternion endomorphism. However this optimization implies that a 3x3 matrix is stored in the `chrono::ChFrameMoving` object, for easing the case of multiple point transformations; the nine values of the matrix are recomputed when the rotation of the frame changes. The improved performance is worth while the overhead of keeping such matrix updated, and the increased memory requirement.

So far, each `chrono::ChFrameMoving` object contains three vectors `chrono::ChVector`, three quaternions `chrono::ChQuaternion` and an auxiliary 3x3 matrix `chrono::ChMatrix33`:

$$\chi_{c++} = \{\mathbf{r}, \mathbf{q}, [A(\mathbf{q})], \dot{\mathbf{r}}, \dot{\mathbf{q}}, \ddot{\mathbf{r}}, \ddot{\mathbf{q}}\}.$$

The `ChFrame` object contains only $\mathbf{r}$, $\mathbf{q}$, $[A(\mathbf{q})]$.

An useful feature of the C++ language is the *operator overloading*, wich allows a straightforward mapping of the $\mathcal{T}(\mathbb{T}, \succ)$ algebra into a new programming syntax where the $\succ$ operator can be represented as a binary operator between two `chrono::ChFrameMoving` objects. To avoid confusion with default operators `*,+,-,/`, we decided to use the `>>` symbol to represent the $\succ$ operation.

Such operation is implemented in the header of the `chrono::ChFrameMoving` class, using the following binary operator overloading:

```
ChFrameMoving<Real> operator >> (...)
```

In the function above, the formulas in Eq.(4-11) are evaluated, where the return value represents the resulting vector $\chi_{a,(c)}$, the object itself (the `this` pointer) is the left argument $\chi_{a,(b)}$ and parameter `Fb` is the right argument $\chi_{b,(c)}$.

Thus, the example of Eq.(1), shown in Fig.1, could be written with the following source code:

```
ChFrameMoving<> cs_30, cs_32, cs_21, cs_10;
cs_10.coord.pos = ChVector<>(2,4,1);
...  etc ...
cs_30 = cs_32 >> cs_21 >> cs_10;
```

We also implemented the $(\cdot)^{-1}$ operation, with arity $\bar{\alpha} = 1$, by overloading the `!` C++ unary operator. This requires the in-place evaluation of Eq.(19-23). We choose the `!` symbol for readability reasons and because it has higher precedence than `>>` in C++ syntax, so parentheses are rarely needed.

Thank to the implementation of the inversion, it is possible, again in example of Fig.1, to obtain `cs_32` if other frames are known: we multiply both sides by `!cs_10` and `!cs_21`, and remembering that `cs_ij >> !cs_ij` will cancel by Theorems 1 and 2, we simply write

```
cs_32 = cs_30 >> !cs_10 >> !cs_21;
```

Note that the previous statement would require two inversions and two coordinate transformations, but a more efficient approach can be developed. In fact we can implement an *inverse transformation* operator, named `<<`, which requires fewer CPU operations:

```
cs_32 = cs_30 << cs_10 << cs_21;
```

For reasons of space, details about the implementation of the `<<` operator are not given; suffices to say that formulas are not much different from the ones in Eq.(19-23). Also, we remark that such `<<` operator is not associative (although, given that C++ compilers evaluates expressions from left to right if no parentheses are used, this is seldom an issue).

Additionally, thank to further specialized implementations of the `>>` operator, one can mix objects of `chrono::ChFrameMoving` and `chrono::ChFrame` in the same expression. In the same way, it is possible to trasform simple 3D vectors of class `chrono::ChVector` instead than entire frames, for example as in:

```
vect_0 = vect_2 >> cs_21 >> cs_10;
```

This last example shows the reason why we preferred to define the $\mathcal{T}$ algebra such that it concatenates kinematic transformations from left to right instead than from right to left: in fact, in case no parentheses are used and there are multiple operators with the same precedence, C++ compilers evaluate expressions by taking couple of operands from left to right and transforming them into temporary data, up to having a single result; hence when transforming simple vectors, as in the example above, the compiler ends always with a sequence of vector-by-frame products that are quickly computed, whereas if we had chosen an algebra where one has to write instead `vect_0 = cs_10 * cs_21 * vect_2` (for example), the compiler would translate it into a sequence of frame-by-frame products up to the last frame-by-vector product[2]. The result would be the same,

[2]For instance, using Denavit-Hartenberg matrices $M_{a,b}$ transforming rotations and positions from a to b, one would rather write in this order: $\mathbf{v}_0 = M_{1,0}M_{2,1}\mathbf{v}_2$.

but the performance would be much poorer because frame-by-frame products are much slower than frame-by-vector.

All classes are templated and metaprogrammed [1], they can work with floating point in double or single precision.

We remark that we performed benchmarks to obtain the best trade-off between computational efficiency, ease of use and exploitation of OOP features [2].

Finally, we implemented serialization methods for all the classes discussed in this paper, so they can be converted from transient to persistent data schemes, and vice versa, in a platform independent way. This is useful for storing data on disk or on networks.

Table 1: Mapping between Lie group operators and C++ operators

Operator	C++ op.	Example	C++ primitive	
$c = b \cdot a$ $c = R_a b$ $c = L_b a$	`*`	`x_ca = x_ba * x_cb`	`x_ca = x_ba.Transform(x_cb)`	Transform, R-to-L chain
$c = a \succ b$ $c = R_a b$ $c = L_b a$	`>>`	`x_ca = x_cb >> x_ba`	`x_ca = x_ba.Transform(x_cb)`	Transform, L-to-R chain
$c := c \cdot a$ $c := R_a c$	`*=`	`x_ca *= x_cb`	`x_ca = x_ca.Trasform(x_cb)`	In-place transform (append to end of chain)
$c := c \succ a$ $c := L_a c$	`>>=`	`x_ca >>= x_cb`	`x_ca = x_cb.Trasform(x_ca)`	In-place transform (prepend to root of chain)
$a = b^{-1}$	`!`	`x_ab = !x_ba`	`x_ab = x_ba.Inverse()`	Inverse element
$c = b^{-1} * a$	`%`	`x_ba = !x_ca * x_cb  = x_ca % x_cb`	`x_ba = x_ca.InvTransf(x_cb)`	Fast inverse transformation, R-to-L
$c = a^{-1} \succ b$	`<`	`x_ba = x_ca >> !x_cb  = x_ca << x_cb`	`x_ba = x_ca.InvTransf(x_cb)`	Fast inverse transformation, L-to-R

5. Validation of semantics

Having agreed that the second subscript of an element represents the frame to whom the coordinates are expressed, when looking at the examples exposed in these pages one can see that, in order to make sense from a kinematical point of view, the subscripts of the elements in the expressions must be chained.

For instance, in $\chi_{3,(0)} = \chi_{3,(2)} \succ \chi_{2,(1)} \succ \chi_{1,(0)}$ it can be seen that the two operands share respectively the second subscript and the first subscript, and the result has the remaining two subscripts, respectively the first of the first operand, and the second of the second operand.

This fact could be exploited in a planned feature, that could be implemented in future: automatic semantics validation that tells if the programmer is writing transformations that make sense.

More in detail, one can point out set of rules about the subscripts like the following:

- Right to left transformation:
  $\chi_{c,(a)} = \chi_{b,(a)} \cdot \chi_{c,(b)}$
  (the (b) subscript must match the b subscript, and the result gets the remaining two subscripts c and (a), in reverse order).

- Left to right transofrmation:
  $\chi_{c,(a)} = \chi_{c,(b)} \succ \chi_{b,(a)}$
  (the (b) subscript must match the b subscript, and the result gets the remaining two subscripts c and (a), in exact order).

- Inversion:
  $\chi_{b,(a)} = \chi^{-1}_{a,(b)}$
  (the a and (b) subscripts are swapped to b and (a)).

Other rules for the semantics are summarized in Table 2 for easy reference, along with a graphical representation of the subscripts rules.

To support this type of run-time validation, additional data could be added to each `chrono::ChFrameMoving` object, representing the two subscripts. When the `>>` operator or the `!` operator are used, the above mentioned rules are invoked and the correctness of the transformation is checked, so a warning, or an assert, or an exception-throwing can signal the problem. The data for the subscript can be a simple (unique) numeric identifier, or a mnemonic string, although the latter option would be slower in execution.

We remark that, by using template metaprogramming, the validation of the sequences of coordinate transformations could be also checked in compile-time, instead of run-time, and this would mean that there is no overhead on the execution time (although it is not as flexible, because objects cannot be reused with different bases once they are created).

Such optional possibility of validating the syntax of expressions for kinematic consistency is particularly useful for educational purposes.

Table 2: Semantics checking rules, for $\chi_{origin,base}$

Op.	C++ syntax	Semantics	Check	Outcome
$\cdot$	`Res = Op1 * Op2`	$\chi_{c,a} = \chi_{b,a} * \chi_{c,b}$	`Op2.base == Op1.origin`	`Res.base = Op1.base` `Res.origin = Op2.origin`
$\succ$	`Res = Op1 >> Op2`	$\chi_{c,a} = \chi_{c,b} >> \chi_{b,a}$	`Op1.base == Op2.origin`	`Res.base = Op2.base` `Res.origin = Op1.origin`
$\cdot$	`Res *= Op`	$\chi_{bc,a} \mathrel{*}= \chi_{c,b}$	`Res.origin == Op.base`	`Res.origin = Op.origin`
$\succ$	`Res >>= Op`	$\chi_{c,ab} >>= \chi_{b,a}$	`Res.base == Op.base`	`Res.base = Op.origin`
χ^{-1}	`Res = !Op`	$\chi_{a,b} = ! \chi_{b,a}$	`Res.origin == Op.base` `Res.base == Op.origin`	`Res.origin == Op.base` `Res.base == Op.origin`

6. Example

Figure 2 shows an example based on a 4-DOF industrial robot and a conveyor belt, both simulated with our multibody software. One must compute the position, speed and acceleration of an item on a conveyor belt respect to the end effector, assuming that position, speed and acceleration of the item (frame n.8)

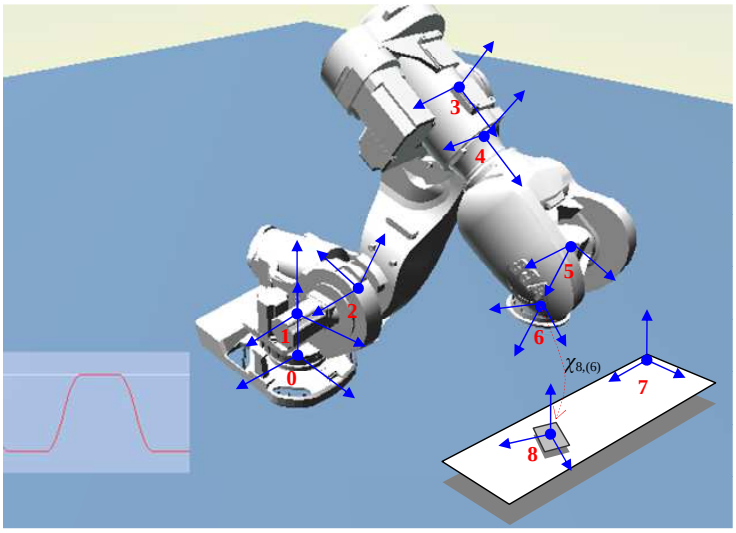

Figure 2: Example. Computing position, speed and acceleration of an item on a conveyor belt, respect to an end effector

is known respect to the conveyor belt (frame n.7), and all frames of the parts of the robot are known respect to the previous joint, up to the base (frame n.0). By using the $\mathcal{T}$ algebra, one can obtain position, rotation, velocity, angular velocity, acceleration, angular acceleration of frame n.8 respect to frame n.6 using a single expression:

$$\chi_{8,(6)} = \chi_{8,(7)} \succ \chi_{7,(0)} \succ \chi^{-1}_{1,(0)} \succ \chi^{-1}_{2,(1)} \succ \chi^{-1}_{3,(2)} \succ \chi^{-1}_{4,(3)} \succ \chi^{-1}_{5,(4)} \succ \chi^{-1}_{6,(5)}$$

Equivalently, one could write instead:

$$\chi_{8,(6)} = \chi_{8,(7)} \succ \chi_{7,(0)} \succ (\chi_{6,(5)} \succ \chi_{5,(4)} \succ \chi_{4,(3)} \succ \chi_{3,(2)} \succ \chi_{2,(1)} \succ \chi_{1,(0)})^{-1}$$

that is, by doing the product of two groups of factors, also:

$$\chi_{8,(6)} = \chi_{8,(0)} \succ \chi^{-1}_{6,(0)} = \chi_{8,(0)} \succ \chi_{0,(6)}$$

7. Conclusion

This paper discusses the $\mathcal{T}(\mathbb{T}, \succ)$ algebra as a compact formal method to represent kinematic transformations in chains of moving frames. This formal framework has been implemented in **Chrono**::Engine using object-oriented programming and the operator-overloading capabilities of the C++ language.

For additional information look at the `chrono::ChFrameMoving` and `chrono::ChFrame` classes.

References

[1] A. Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley, New York, 2001.

[2] A. Alexandrescu H. Sutter. *C++ Coding Standards: 101 Rules, Guidelines, and Best Practices*. Addison-Wesley, New York, 2004.

[3] J.M. Hervé. The mathematical group structure of the set of displacements. *Mechanism and Machine Theory*, 29(1):73 – 81, 1994.

[4] J.M. Hervé. The lie group of rigid body displacements, a fundamental tool for mechanism design. *Mechanism and Machine Theory*, 34(5):719 – 730, 1999.

[5] B. Kolev. Lie groups and mechanics: An introduction. *Journal of Nonlinear Mathematical Physics*, 11:480–498, 2004.

[6] M.W.Spong and M. Vidyasagar. *Robot Dynamics and Control*. Wiley, New York, 1989.

[7] J.M. Selig. *Geometric Fundamentals in Robotics*. Springer, 2005.

[8] A. Shabana. *Multibody Systems*. John Wiley and Sons, New York, 1989.

[9] A. Tasora and P. Righettini. Sliding contact between freeform surfaces. *Multibody System Dynamics*, 10(3):239–262, 2003.