luadq: Tutorial on using Dual Quaternions in Lua

E. Simo-Serra

May 3, 2013

Abstract

This short tutorial attempts to explain the basics of using unit dual quaternions with the Lua interface of libdq, known as luadq. Unit dual quaternions are widely used in kinematics and can express spatial displacements in $SE(3)$.

1 Introduction

This small tutorial is meant to teach potential users how to use luadq, the Lua front-end of libdq [1] in the context of kinematics. It is not meant to be a in-depth review nor explanation of the mathematics behind dual quaternions. For a more detailed guide on mathematics in robotic kinematics please refer to [2].

Dual quaternions are an efficient representation of spatial displacements that are generally faster and more efficient to use that homogeneous matrices or quaternions with translation vectors. Recently they have been seen widely used in kinematics [3, 4]. However, they also have usage on other fields like 3D graphics [5] or computer vision [6]. While the dual quaternions can be used in many contexts, this tutorial focuses on their application in kinematics.

2 Theoretic Primer

For the purpose of this tutorial it is sufficient to say that dual quaternions are elements of the Clifford even subalgebra $C^+_{0,3,1}$. There are many notations for dual quaternions. This library uses the basis used by McCarthy which is the same as Selig with minor rearrangements,

$$\{1, e_{23}, e_{31}, e_{12}, e_{41}, e_{42}, e_{43}, e_{1234}\} = \{1, i, j, k, i\epsilon, j\epsilon, k\epsilon, \epsilon\} \tag{1}$$

This allows us to write a dual quaternion as,

$$\widehat{Q} = (q_0 + q_1 i + q_2 j + q_3 k) + \epsilon(q_7 + q_4 i + q_5 j + q_6 k) = \widehat{q} + \epsilon \widehat{q}^0 \tag{2}$$

Using vertical notation we would have the following,

$$\widehat{Q} = \left\{ \begin{array}{c} e_{23} \\ e_{31} \\ e_{12} \\ 1 \end{array} \right\} + \left\{ \begin{array}{c} e_{41} \\ e_{42} \\ e_{43} \\ e_{1234} \end{array} \right\} = \left\{ \begin{array}{c} i \\ j \\ k \\ 1 \end{array} \right\} + \epsilon \left\{ \begin{array}{c} i \\ j \\ k \\ 1 \end{array} \right\} \tag{3}$$

In order for the dual quaternion to be able to represent spatial displacements it must be a unit dual quaternion and thus comply with the following restrictions,

$$\widehat{q}\widehat{q}^0 = 1 \tag{4}$$

$$\widehat{q} \cdot \widehat{q}^0 = 0 \tag{5}$$

It is important to note that unit dual quaternions double cover the special Euclidean group $SE(3)$. This means that $\widehat{Q}$ and $-\widehat{Q}$ represent the same spatial displacement.

3 Set-up

Installation is straightforward on linux systems and consists of two steps: compilation and installation. For installation of luadq the only dependency is lua[1] and luarocks[2].

On Ubuntu you can obtain the dependencies by running:

```
apt-get install liblua5.1-0-dev luarocks
```

If you wish to additionally compile the Doxygen documentation (make docs), you will also need to run:

```
apt-get install doxygen doxygen-latex texlive graphviz
```

3.1 Compilation

To compile execute from the root of the libdq directory:

```
make # Compiles libdq
make rock # Compiles luadq
```

3.2 Installation

Once compile it must be installed. The paths can be set by editing the Makefile. By default it installs into system directories and needs root privileges. To install execute from the root of the libdq directory:

```
make install # Installs libdq
make rock-install # Installs luadq
```

3.3 Verification

You can verify the installation is correct by checking if your Lua installation can find luadq. This can be done by running the following in a Lua terminal:

```
Lua 5.1.4  Copyright (C) 1994-2008 Lua.org, PUC-Rio
> require 'luadq'
> =luadq
table: 0x17fbd60
```

You should get similar output.

[1] http://www.lua.org/
[2] http://luarocks.org/

4 Getting Started

The API of luadq is the same as libdq's API. However, the naming scheme is different. Instead of being prepended by "dq_", the functions are prepended by "luadq.". For details of these functions please refer to the libdq manual [1]. Some examples:

- dq_rotation $\longmapsto$ luadq.rotation
- dq_rotation_plucker $\longmapsto$ luadq.rotation_plucker
- dq_rotation_matrix $\longmapsto$ luadq.rotation_matrix
- dq_translation $\longmapsto$ luadq.translation
- dq_translation_vector $\longmapsto$ luadq.translation_vector
- dq_point $\longmapsto$ luadq.point
- dq_line $\longmapsto$ luadq.line
- dq_line_plucker $\longmapsto$ luadq.line_plucker
- dq_homo $\longmapsto$ luadq.homo
- dq_copy $\longmapsto$ luadq.copy
- dq_inv $\longmapsto$ luadq.inv
- dq_f1g $\longmapsto$ luadq.f1g
- dq_f2g $\longmapsto$ luadq.f2g
- dq_f3g $\longmapsto$ luadq.f3g
- dq_f4g $\longmapsto$ luadq.f4g

The differences with the C types are not too large. For 3D vectors you must define them in Lua as tables:

```
{ 1, 2, 3 } -- A vector with components x=1, y=2, z=3
```

For 3×3 matrices you must use a multi-dimensional table:

```
{ { 1, 2, 3 }, -- First row
  { 4, 5, 6 }, -- Second row
  { 7, 8, 9 } } -- Last row
```

For homogeneous matrices, they are represented as a 3×4 table:

```
{ { 1,  2,  3,  4 }, -- First row
  { 5,  6,  7,  8 }, -- Second row
  { 9, 10, 11, 12 } } -- Last row
```

However, some Lua functions do not directly derive from libdq C equivalents. These are:

- Q = luadq.raw(vec) – Takes a table with 8 components and sets the dual quaternion
- vec = luadq.get(Q) – Gets the raw 8 components of the dual quaternion
- rn, dn = luadq.norm2(Q) – Returns two numbers, the real and dual component of the norm
- R,t = luadq.extract(Q) – Gets the rotation matrix and translation vector associated to the dual quaternion
- b = luadq.unit(Q) – Checks to see if the dual quaternion is unitary

Also note that basic math operators $+$, $-$ and $*$ and supported directly.

4.1 First steps

Now we will give some simple examples of using luadq. These are pure Lua programs.

```
require 'luadq'
-- Create the origin
O = luadq.point( { 0, 0, 0 } )
-- Translation 5 units in X direction
T = luadq.translation( 5, {1, 0, 0 } )
-- Rotation around Z axis centered on origin
-- We are using Plucker coordinates to describe the
-- axis of rotation
R = luadq.rotation_plucker( math.pi, {0, 0, 1}, {0, 0, 0} )
-- We compose displacements, translation is applied first
D = R * T
-- We apply the Clifford conjugation for points on the origin O
-- using the displacement
P = D:f4g( O )
-- Display the dual quaternion
P:print( true )
-- As it is a point transformation the point is stored in the
   imaginary component
Q = P:get()
-- Display the new coordinates
print( string.format( "x=%d, y=%d, z=%d", Q[5], Q[6], Q[7] ) )
```

The example is fairly straight forward. We create a point, then we create a spatial transformation composed of both a translation and a rotation, and we finally transform the point and display the new point obtained (which should be (-5,0,0)). This small application can give an idea of the power behind dual quaternions and the simplicity of working with them. Next we'll give an example on a more real forward kinematics situation.

5 Example: Epson E2L Scara Robot

In this example we'll consider the Epson E2L SCARA robot. The joints of this robot have the following Plücker coordinates.

$$\mathsf{S}_1 = (0,0,1) + \epsilon(0,0,0) \tag{6}$$
$$\mathsf{S}_2 = (0,0,1) + \epsilon(0,-300,0) \tag{7}$$
$$\mathsf{S}_3 = (0,0,1) + \epsilon(0,650,0) \tag{8}$$
$$\mathsf{S}_4 = (0,0,-1) + \epsilon(0,650,0) \tag{9}$$

where S_1, S_2 and S_3 are revolute joints while S_4 is a prismatic joint.

We'll also consider the end-effector or Tool Center Point (TCP) to be at,

$$tcp = (650, 0, 318) \tag{10}$$

First we'll write a small function that will perform the forward kinematics, that is,

$$\hat{Q}(\boldsymbol{\Delta\hat{\theta}}) = \left(\prod_{i=1}^{n} e^{\frac{\Delta\hat{\theta}_i}{2}\mathsf{S}_i}\right) Q_{tcp} = \left(\prod_{i=1}^{n} (\cos\frac{\Delta\hat{\theta}_i}{2} + \sin\frac{\Delta\hat{\theta}_i}{2}\mathsf{S}_i)\right) Q_{tcp} \tag{11}$$

where $\Delta\hat{\theta} = \hat{\theta} - \hat{\theta}_{ref}$ with $\hat{\theta}_{ref}$ being the joint parameter in the reference configuration and Q_{tcp} the transformation from the origin to the TCP position.

Note that we have to consider the transformation from the origin to the TCP when doing the forward kinematics. We can write the case of the Epson E2L Scara robot in Lua as the following,

```lua
-- Relative forward kinematics kinematics for the
-- Epson E2L Scara Robot
-- The values of thetas are relative to the reference configuration
function scara_relfk( theta1, theta2, theta3, theta4 )

   -- Here we create the three rotation quaternions
   -- Note that we are directly using the Plucker coordinates of the
   -- joint axes. However, luadq.rotation would work for axes defined
   -- by an orientation and a point belonging to the line
   local S1  = luadq.rotation( theta1, { 0, 0, 1 },  { 0,    0, 0 } )
   local S2  = luadq.rotation( theta2, { 0, 0, 1 },  { 0, -300, 0 } )
   local S3  = luadq.rotation( theta3, { 0, 0, 1 },  { 0, -650, 0 } )
   -- Fourth joint is translation
   local S4  = luadq.translation( theta4, { 0, 0, -1 } )

   -- The end effector position for the reference configuration
   local TCP = luadq.translation_vector( { 650, 0, 318 } )

   -- Here we combine the four displacements into a single one and
   -- return it. Notice we also append the end-effector at the
   -- reference configuration. The order of application of the
   -- transformations is right-most is applied first.
   local S   =  S1 * S2 * S3 * S4 * TCP

   -- Here we create a point at the origin, we'll transform this
   -- point
   local P   = luadq.point( { 0, 0, 0 } )

   -- Here we basically use the action f4g (specified in libdq's
   -- manual) to transform the origin point. This will give us
   -- the forward kinematics of the robot.
   return S:f4g( P )
end
```

This code is straight forward and basically performs the relative transformation. We can see it does indeed work quickly by checking the reference position,

```lua
-- This will give the original TCP
TCP = scara_relfk( 0, 0, 0, 0 )
print( "TCP:" )
TCP:print( true ) -- The parameter makes it print vertically
-- We can extract the actual point value by using :get()
Q = TCP:get()
print( string.format( "x=%d,␣y=%d,␣z=%d", Q[5], Q[6], Q[7] ) )
```

which as expected gives "x=650, y=0, z=318".

Now we can move the 4th joint (prismatic) to lower it to the X-Y plane. This is simply done by setting $\theta_4 = 318$,

```lua
-- We can only do translation to lower it to the bottom plane
TCP = scara_relfk( 0, 0, 0, 318 )
print( "\nLowering␣to␣X-Y␣plane:" )
TCP:print(true)
Q = TCP:get()
print( string.format( "x=%d,␣y=%d,␣z=%d", Q[5], Q[6], Q[7] ) )
```

which gives us “x=650, y=0, z=0”.

You can also move other joints like for example,

```
-- You can also play around with rotations
TCP = scara_relfk( -math.pi/2, math.pi/2, math.pi/2, 318 )
print( "\nRotating:" )
TCP:print(true)
Q = TCP:get()
print( string.format( "x=%d, y=%d, z=%d", Q[5], Q[6], Q[7] ) )
```

which gives us “x=-350, y=-300, z=0”.

As you can see dual quaternions are a powerful and simple way of performing forward kinematics. Note that in these examples we were ignoring the rotations, although internally they were taken into account. To see the rotation you can use the line transform “luadq.f2g” and look at the real component of dual quaternion which will represent the rotation using “luadq.extract“.

6 Conclusions

As shown in this tutorial, unit dual quaternions are a powerful way of expressing robot kinematics. The luadq interface to libdq provides an efficient and simple way of being able to manipulate dual quaternions to be able perform standard robotic kinematics like forward kinematics.

The only reason dual quaternions are as widely used are because of the lack of familiarity most people have with Clifford algebras. This tutorial attempts to dispel a bit of the mystic behind the quaternions and show that they are not as difficult to use as they may appear, especially with tools that simplify the entire process like the one presented.

References

[1] E. Simo-Serra, “libdq: Dual Quaternion Library.” https://github.com/bobbens/libdq, 2011.

[2] J. M. Selig, *Geometric Fundamentals of Robotics (Monographs in Computer Science)*. SpringerVerlag, 2004.

[3] A. Perez and J. M. McCarthy, “Dual quaternion synthesis of constrained robotic systems,” *Journal of Mechanical Design*, vol. 126, no. 3, pp. 425–435, 2004.

[4] A. Perez-Gracia and J. M. McCarthy, “Kinematic synthesis of spatial serial chains using Clifford algebra exponentials,” *Proc. of the Institution of Mechanical Engineers, Part C: Journal of Mechanical Engineering Science*, vol. 220, no. 7, pp. 953–968, 2006.

[5] L. Kavan, S. Collins, J. Zara, and C. O’Sullivan, “Geometric skinning with approximate dual quaternion blending,” vol. 27, (New York, NY, USA), p. 105, ACM Press, 2008.

[6] A. Torsello, E. Rodolà, and A. Albarelli, “Multiview registration via graph diffusion of dual quaternions,” in *CVPR*, pp. 2441–2448, IEEE, 2011.