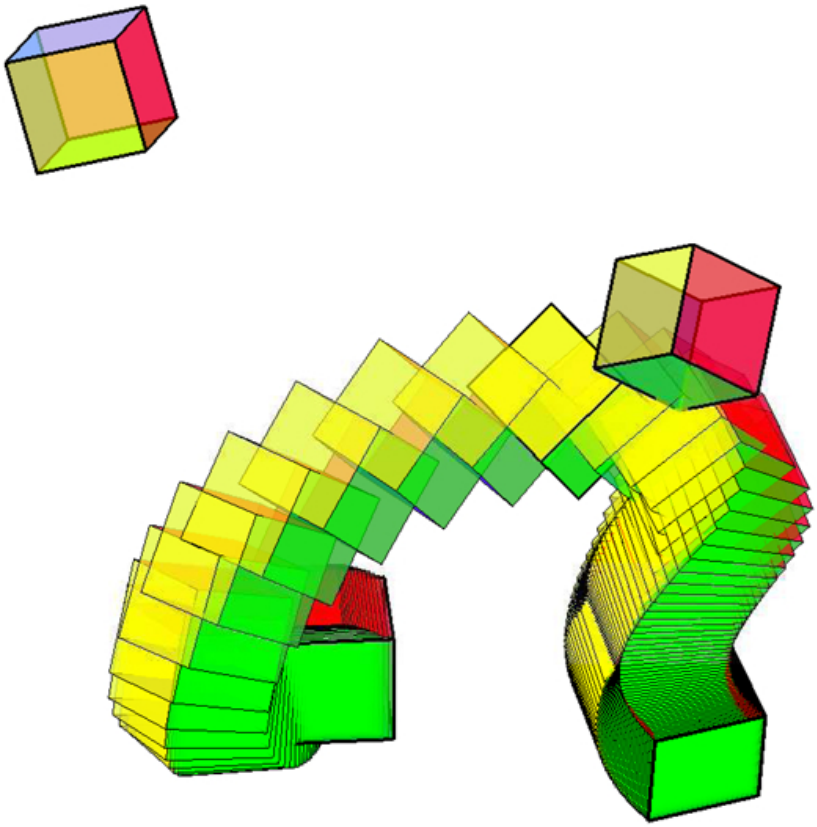
Documentation - Kinematic Toolbox

Daniel Klawitter
Technische Universität Dresden

March 29, 2010



Contents

1 Introduction

This toolbox was developed while the diploma thesis [18] was written. The main goal is to interpolate given positions of an object in $\mathbb{R}^3$ and to generate a continuous motion through these positions. The positions are given in terms of translation and rotation. We consider $\mathcal{R}$ to be the principal-axes space-fixed system and $\mathcal{G}_i$ the system that is derived from $\mathcal{R}$ by a motion A_i, consisting of a translation and a rotation. Via Study's kinematic mapping, one can map these motions to points on Study's quadric S_2^6, a special hyperquadric in 7-dimensional projective space $\mathbb{P}^7$. The Study quadric is a model for the 6-parametric group SE(3) of direct Euclidean displacements in 3-space.

The documentation is organized as follows: In chapter 2, some classes that deal with basic concepts in kinematics are presented. Their use is explained and all functions are listed. Chapter 3 deals with some convenient functions and explains their application. The interpolation functions given in chapter 4 work in an d-dimensional projective space $\mathbb{P}^d$. Examples for the case $d = 3$ (Sphere) and $d = 7$ (Study's quadric) are implemented in the toolbox. You may also change the quadric and dimension to adapt these tools to specific problems.

2 Classes

2.1 The DualNumber Class

The ring of the dual numbers $\mathbb{D} := \left\{a + \epsilon b | a, b \in \mathbb{R}, \epsilon^2 = 0\right\}$ consists of two real numbers a and b, where ϵ is the dual unit.

Properties of the DualNumber Class

- real...the real part of the DualNumber class object.
- dual...the dual part of the DualNumber class object.

Methods in the DualNumber Class

- The constructor

```
>> obj = DualNumber(a,b)
```

creates a DualNumber object. The arguments a and b are reals. If the constructor is called without arguments, then the dual number is set to $0 + \epsilon 0$. For example, the following commands create two DualNumber class objects:

```
>> d1 = DualNumber(1,1);
>> d2 = DualNumber(3,4);
```

Overloaded functions for calculation with DualNumber class objects:

- plus: $d_1 + d_2$ returns the sum of two DualNumber class objects

```
>> d1+d2

ans =

class DualNumber
4 + epsilon * ( 5 )
```

The output for DualNumber class objects in this form is implemented in the disp funtion.

- minus: $d_1 - d_2$ returns the difference of two DualNumber class objects

```
>> d1-d2

ans =

class DualNumber
-2 + epsilon * ( -3 )
```

- mtimes: $d_1 * d_2$ returns the product of two DualNumber class objects

```
>> d1*d2

ans =

class DualNumber
3 + epsilon * ( 7 )
```

- times: $a. * d_1$ returns the product of a real number a and a DualNumber class object d_1

```
>> 0.5.*d1

ans =

class DualNumber
0.5 + epsilon * ( 0.5 )
```

- sqrt: returns the square root of a DualNumber class object

```
>> sqrt(d2)

ans =

class DualNumber
1.7321 + epsilon * ( 1.1547 )
```

- inv: returns the inverse of a DualNumber class object

```
>> inv(d2)

ans =

class DualNumber
0.33333 + epsilon * ( -0.44444 )
```

  To validate this result, one can multiply the inverse dual number with d2

```
>> d2*inv(d2)

ans =

class DualNumber
1 + epsilon * ( 0 )
```

- disp: a function for an intuitive output for DualNumber class objects. For every object of the class DualNumber, the disp function controls the output in the Matlab command line.
- conjugate: returns the conjugate dual number $a - \epsilon b$

```
>> conjugate(d2)

ans =

class DualNumber
3 + epsilon * ( -4 )
```

It is also possible to call class function like we would do in Java

```
>> d2.conjugate

ans =

class DualNumber
3 + epsilon * ( -4 )
```

By typing d2. and then pressing the tabulator key code completion is activated and all available class functions are shown. Notice that the code completion is available for all upcoming objects and classes.

2.2 The Quaternion Class

The quaternions are a construct that extend the complex numbers. The set of quaternions $\mathbb{H}$ is a 4-dimensional vectorspace over $\mathbb{R}$. With addition and multiplication they form a skew field. Addition is defined component-wise and multiplication by the following rules for the imaginary units.

$\circ$	**i**	**j**	**k**
i	-1	**k**	$-$**j**
j	$-$**k**	-1	**i**
k	**j**	$-$**i**	-1

Properties of the Quaternion Class

- scal...the scalar part of the Quaternion class object
- vect...the vector part of the Quaternion class object

Methods in the Quaternion Class

- The constructor

```
>> obj = Quaternion(a,b);
```

creates a Quaternion class object with scalar part a and vector part b, where $b \in \mathbb{R}^3$. Possible inputs are
no input: scalar and vector part of the Quaternion class object are set to zero;
scalar: the vector is set to zero.
scalar and a vector.
The following commands create two Quaternion class objects.

```
>> q1 = Quaternion(1,[1 1 1])

q1 =

class Quaternion
1 + 1i + 1j + 1k
>> q2 = Quaternion(2,[0 0 -1])

q2 =

class Quaternion
2 + 0i + 0j + -1k
```

- plus: $q_1 + q_2$ returns the sum of two Quaternion class objects

```
>>q1+q2

ans =

class Quaternion
3 + 1i + 1j + 0k
```

- minus: $q_1 - q_2$ returns the difference of two Quaternion class objects

```
>>q1-q2

ans =

class Quaternion
-1 + 1i + 1j + 2k
```

- mtimes: $q_1 * q_2$ returns the product of two Quaternion class objects

```
>> q1*q2

ans =

class Quaternion
3 + 1i + 3j + 1k
```

- times: $a. * q_1$ returns the product of a real number a and a Quaternion class object q_1

```
>> 0.5.*q1

ans =

class Quaternion
0.5 + 0.5i + 0.5j + 0.5k
```

- norm: returns the norm of a Quaternion class object

```
>> norm(q1)

ans =

     4
```

- normalize: returns the normalized Quaternion class object

```
>> normalize(q1)

ans =

class Quaternion
0.25 + 0.25i + 0.25j + 0.25k
```

- conjugate: returns the conjugated Quaternion class object

```
>> conjugate(q1)

ans =

class Quaternion
1 + -1i + -1j + -1k
```

- the disp function provides formatted output.

2.3 The Dualquaternion Class

Dual quaternions were invented by Eduard Study, therefore they are also called Study's biquaternions. They parameterize the group of Euclidean spatial displacements. The basis elements are

$$\mathbf{1},\ \mathbf{i},\ \mathbf{j},\ \mathbf{k},\ \epsilon,\ \epsilon\mathbf{i},\ \epsilon\mathbf{j},\ \epsilon\mathbf{k}$$

with $\mathbf{i}\epsilon = \epsilon\mathbf{i}$, $\mathbf{j}\epsilon = \epsilon\mathbf{j}$ and $\mathbf{k}\epsilon = \epsilon\mathbf{k}$. The dual unit satisfies the equation $\epsilon^2 = 0$. The addition is declared component-wise and for the multiplication the following table is given:

$\circ$	$\mathbf{1}$	$\mathbf{i}$	$\mathbf{j}$	$\mathbf{k}$	ϵ	$\epsilon\mathbf{i}$	$\epsilon\mathbf{j}$	$\epsilon\mathbf{k}$
$\mathbf{1}$	$\mathbf{1}$	$\mathbf{i}$	$\mathbf{j}$	$\mathbf{k}$	ϵ	$\epsilon\mathbf{i}$	$\epsilon\mathbf{j}$	$\epsilon\mathbf{k}$
$\mathbf{i}$	$\mathbf{i}$	$-\mathbf{1}$	$\mathbf{k}$	$-\mathbf{j}$	$\epsilon\mathbf{i}$	$-\epsilon$	$-\epsilon\mathbf{k}$	$\epsilon\mathbf{j}$
$\mathbf{j}$	$\mathbf{j}$	$-\mathbf{k}$	$-\mathbf{1}$	$\mathbf{i}$	$\epsilon\mathbf{j}$	$\epsilon\mathbf{k}$	$-\epsilon$	$-\epsilon\mathbf{i}$
$\mathbf{k}$	$\mathbf{k}$	$\mathbf{j}$	$-\mathbf{i}$	$-\mathbf{1}$	$\epsilon\mathbf{k}$	$-\epsilon\mathbf{j}$	$\epsilon\mathbf{i}$	$-\epsilon$
ϵ	ϵ	$-\epsilon\mathbf{i}$	$-\epsilon\mathbf{j}$	$-\epsilon\mathbf{k}$	0	0	0	0
$\epsilon\mathbf{i}$	$\epsilon\mathbf{i}$	ϵ	$-\epsilon\mathbf{k}$	$\epsilon\mathbf{j}$	0	0	0	0
$\epsilon\mathbf{j}$	$\epsilon\mathbf{j}$	$\epsilon\mathbf{k}$	ϵ	$-\epsilon\mathbf{i}$	0	0	0	0
$\epsilon\mathbf{k}$	$\epsilon\mathbf{k}$	$-\epsilon\mathbf{j}$	$\epsilon\mathbf{i}$	ϵ	0	0	0	0

Properties of the Dualquaternion Class

- realpart...the real part is a Quaternion class object with subproperties scal and vect.
- dualpart...the dual part is a Quaternion class object.

Methods in the Dualquaternion Class

- The constructor

```
obj = Dualquaternion(q1,q2)
```

q_1 and q_2 are Quaternion class objects. It is also possible to call the constructor with no input arguments. Then the dual quaternion will be the zero dual quaternion. The following commands create two Dualquaternion objects for futher calculations.

```
Dq1 = Dualquaternion(Quaternion(1,[0 0 0]),...
                     Quaternion(1,[0.5 0 0.5]))
Dq1 =

class Dualquaternion
1+ 0i+ 0j+ 0k+ epsilon*(1+ 0.5i+ 0j+ 0.5k)
>> Dq2 = Dualquaternion(Quaternion(0.9239,[0 0 0.3827]),...
                        Quaternion)
Dq2 =

class Dualquaternion
0.9239+ 0i+ 0j+ 0.3827k+ epsilon*(0+ 0i+ 0j+ 0k)
```

- plus: $Dq_1 + Dq_2$ returns the sum of two Dualquaternions class objects

```
>> Dq1+Dq2

ans =

class Dualquaternion
1.9239+ 0i+ 0j+ 0.3827k+ epsilon*(1+ 0.5i+ 0j+ 0.5k)
```

- minus: $Dq_1 - Dq_2$ returns the difference of two Dualquaternions class objects

```
>> Dq1-Dq2

ans =

class Dualquaternion
0.0761+ 0i+ 0j+ -0.3827k+ epsilon*(1+ 0.5i+ 0j+ 0.5k)
```

- mtimes: $Dq_1 * Dq_2$ returns the product of two Dualquaternions class objects

```
>> Dq1*Dq2

ans =

class Dualquaternion
0.9239+ 0i+ 0j+ 0.3827k
+ epsilon*(0.73255+ 0.46195i -0.19135j+ 0.84465k)
```

- times: $a.*Dq_1$ returns the product of a real number (or a dual number) a with the Dualquaternion class object Dq_1

```
>> 0.5.*Dq1

ans =

class Dualquaternion
0.5+ 0i+ 0j+ 0k+ epsilon*(0.5+ 0.25i+ 0j+ 0.25k)
```

- conjugate: returns the conjugated Dualquaternion class object

```
>> conjugate(Dq1) %Dq1.conjugate

ans =

class Dualquaternion
1+ 0i+ 0j+ 0k+ epsilon*(1+ -0.5i+ 0j+ -0.5k)
```

- dualconjugate: returns the dual conjugated Dualquaternion class object

```
>> Dq1.dualconjugate %dualconjugate(Dq1)

ans =

class Dualquaternion
1+ 0i+ 0j+ 0k+ epsilon*(-1+ -0.5i+ 0j+ -0.5k)
```

- norm: returns the norm, but note that this norm does not define a norm on the set of dual quaternions $\mathbb{H}_d$.

- getTranslation: returns the translational part of the spatial displacement described by the dual quaternion. In the next example, T is a dual quaternion that describes a translation and the dual quaternion R describes a rotation. The product is the dual quaternion representation of the spatial displacement. The getTranslation function applied to the product returns the translational part. Note that this form is homogeneous. That means, the vector in the dualpart of T is the half of the real translation vector and the getTranslation function returns a Dualquaternion class object, where the real translation vector t is contained in the dual part. The scalar part of the realpart of the dual quaternion is the homogeneous factor.

```
>> T = Dualquaternion(Quaternion(1,[0 0 0]),...
                      Quaternion(0,[0.5 4 0.5]));
>> R = Dualquaternion(Quaternion(0.9239,[0 0 0.3827]),...
                      Quaternion);
>> Q = T*R;
>> Q.getTranslation

ans =

class Dualquaternion
2+ 0i+ 0j+ 0k+ epsilon*(0+ 1i+ 8j+ 1k)
```

The vector in the dual part $t = (1, 8, 1)^T$ is the translation vector.

- getRotation: Continuing the example above, the function getRotation applied to the product returns R.

```
>> Q.getRotation

ans =

class Dualquaternion
0.92385+ 0i+ 0j+ 0.38268k+ epsilon*(0+ 0i+ 0j+ 0k)
```

The real part of this dual quaternion is the rotation in *EULER-coordinates*.

- displace(Q,x): returns a Dualquaternion object. Notice the special form of the Dualquaternion object x.

```
>> x = Dualquaternion(Quaternion(1,[0 0 0]),...
                      Quaternion(0,[1 0 0]));
>> displace(Q,x)

ans =

class Dualquaternion
1.0001+ 0i+ 0j+ 0k+ epsilon*(0+ 1.7072i+ 8.7076j+ 1.0001k)
```

The displace function performs the spatial displacement described through the dual quaternion Q. As you can see, this function just operates on the dual part of x. The dual part of x represents a point $(1, 0, 0)^T \in \mathbb{R}^3$ and the image point is $(1.7072, 8.7076, 1.0001)^T$.

- displaceRT(Q,x): this function works like the function above. The only difference is that the dual quaternion gets splitted in a translational and a rotational part. In a first step, the rotation is applied, then the translation follows.

- displaceTR(Q,x): returns a displaced point where first the translation and then the rotation is applied.

- getStudyParameters: returns a STUDYparameter object

```
Q.getStudyParameters

ans =

class STUDYparameter
    0.9239
         0
         0
    0.3827
   -0.1913
    1.9928
    3.5043
    0.4620
```

- normalize: returns the normalized dual quaternion
- latex1: returns the LaTeX-code of the dual quaternion in the following form

$$Q = a_1 + a_2\epsilon + (b_1 + b_2\epsilon)\mathbf{i} + (c_1 + c_2\epsilon)\mathbf{j} + (d_1 + d_2\epsilon)\mathbf{k}.$$

```
>> Q.latex1

ans =

0.9239-0.19135\epsilon+(0+1.9928\epsilon)\mathbf{i}
+(0+3.5043\epsilon)\mathbf{j}
+(0.3827+0.46195\epsilon)\mathbf{k}
```

- latex2: returns the LaTeX-code in the form

$$Q = a_1 + b_1\mathbf{i} + c_1\mathbf{j} + d_1\mathbf{k} + (a_2 + b_2\mathbf{i} + c_2\mathbf{j} + d_2\mathbf{k})\epsilon$$

```
>> Q.latex2

ans =

0.9239+0\mathbf{i}+0\mathbf{j}+0.3827\mathbf{k}
+(-0.19135+1.9928\mathbf{i}+3.5043\mathbf{j}
+0.46195\mathbf{k})\epsilon
```

2.4 The RotationMatrix Class

This class contains several kinds of parametrizations of rotations. It is possible to create rotation matrices in several ways.

Properties of the RotationMatrix Class

- data...a 3×3 matrix containing the entries of the RotationMatrix class object

Methods in the RotationMatrix Class

- The constructor

```
>> obj = RotationMatrix( varargin )
```

creates a RotationMatrix object. For this class, more than one possibility exists to call the constructor. RoationMatrix objects can be generated from *EULER-coordinates*, *RODRIGUES-coordinates*, *EXPONENTIAL-ccordinates* or by giving the rotation axis and the corresponding rotation angle (GENERAL).

```
>> A = RotationMatrix('EULER',[.5 .5 .5 .5])

A =

class RotationMatrix
     0     0     1
     1     0     0
     0     1     0

>> A = RotationMatrix('GENERAL',2,[0; 0; 5])

A =

class RotationMatrix
   -0.4161   -0.9093         0
    0.9093   -0.4161         0
         0         0    1.0000

>> A = RotationMatrix('RODRIGUES',[-1; 2; 1])

A =

class RotationMatrix
   -0.4286   -0.8571    0.2857
   -0.2857    0.4286    0.8571
   -0.8571    0.2857   -0.4286

>> A = RotationMatrix('EXPONENTIAL',[0; 1; 1],pi/4)

A =

class RotationMatrix
    0.7071   -0.5000    0.5000
    0.5000    0.8536    0.1464
   -0.5000    0.1464    0.8536
```

- toHomogeneousMatrix: returns a 4×4 HomogeneousTransformationMatrix class object that contains the rotation matrix

```
>> A.toHomogeneousMatrix

ans =

class HomogeneousTransformationMatrix
    1.0000         0         0         0
         0    0.7071   -0.5000    0.5000
         0    0.5000    0.8536    0.1464
         0   -0.5000    0.1464    0.8536
```

- getAngle: returns the angle of a RotationMatrix class object

```
>> A.getAngle

ans =

    0.7854
```

- getAxis: returns the rotation axis of a RotationMatrix class object

```
>> A.getAxis

ans =

         0
    0.7071
    0.7071
```

- mtimes: overloads the * operator for RotationMatrix class and normal vectors
- disp: for output

2.5 The HomogeneousVector Class

This class is implemented for homogeneous vectors in $\mathbb{P}^3$.

Properties of the HomogeneousVector Class

- lambda...the homogeneous factor of a HomogeneousVector object
- coords...the vectorpart of a HomogeneousVector object

Methods in the HomogeneousVector Class

- The constructor

```
>> obj = HomogeneousVector(vect)
```

The constructor of this class can be called with the vect argument, which is a vector from $\mathbb{R}^4$, or without input argument. In the second case, the coords property of the object is set to $(0,0,0)^T$ and the lambda property is set to 1.

```
>> a = getEulerCoords(pi/4,[0 0 1]);
>> rotation = HomogeneousVector(a);
>> translation = HomogeneousVector([1 1 1 0])

translation =

class HomogeneousVector
[1,1,1,0]
```

- normalize: returns the normalized HomogeneousVector object

```
>> rotation.normalize

ans =

class HomogeneousVector
[1,0,0,0.41421]
```

- times: multiplication with scalars for homogeneous vectors
- power: overload the .^ operator

```
>> power(rotation, 2)

ans =

class HomogeneousVector
[0.85355,0,0,0.14645]

>> rotation.^2

ans =

class HomogeneousVector
[0.85355,0,0,0.14645]
```

- sum: returns the sum of all components
- dualTranslationQuaternion: returns a Dualquaternion object that describes a translation

```
>> dualTranslationQuaternion(translation)

ans =

2+ 0i+ 0j+ 0k+ epsilon*(0+ 2i+ 2j+ 0k)
```

- rotationQuaternion: returns a Quaternion object that describes a rotation

```
>> rotationQuaternion(rotation)

ans =

class Quaternion
0.92388 + 0i + 0j + 0.38268k
```

- dualRotationQuaternion: returns a Dualquaternion object, that describes a rotation

```
>> dualRotationQuaternion(rotation)

ans =

0.92388+ 0i+ 0j+ 0.38268k+ epsilon*(0+ 0i+ 0j+ 0k)
```

- DualSpatialDisplacementQuaternion: returns a Dualquaternion object that contains a spatial displacement described through a translation and a rotation, which are both homogeneous vectors.

```
>> DualSpatialDisplacementQuaternion(translation,rotation)

ans =

1.8478+ 0i+ 0j+ 0.76537k+ epsilon*(0+ 2.6131i+ 1.0824j+ 0k)
```

- double: returns a vector of real numbers

```
>> translation.double

ans =

     1     1     1     0
```

- disp: function for output of HomogeneousVector class objects

2.6 The HomogeneousTransformationMatrix Class

A class for calculations on homogeneous transformation matrices. The matrices have the form

$$M = \left(\begin{array}{c|ccc} 1 & 0 & 0 & 0 \\ \hline t_1 & & & \\ t_2 & & \mathbf{A} & \\ t_3 & & & \end{array}\right),$$

where $t = (t_1, t_2, t_3)^T$ is the translation vector and A a rotation matrix.

Properties of the HomogeneousTransformationMatrix Class

- data...matrix entries
- rotation...HomogeneousVector object containing the rotation in *EULER-coordinates*
- translation...HomogeneousVector object describing a translation

Methods in the HomogeneousTransformationMatrix Class

- The constructor

```
>> rotation=HomogeneousVector([0.92388,0,0,0.38268]);
>> translation = HomogeneousVector([1 1 1 0]);
>> A = HomogeneousTransformationMatrix(translation, rotation)

A =

class HomogeneousTransformationMatrix
    1.0000         0         0         0
    1.0000    0.7071   -0.7071         0
    1.0000    0.7071    0.7071         0
         0         0         0    1.0000

>> B = HomogeneousTransformationMatrix;
```

  If the constructor is called with no input arguments, the homogeneous transformation matrix is the 4×4 identity matrix.

- times: '.*' operator for multiplication with scalars
- mtimes: multiplication of two HomogeneousTransformationMatrix objects or of a HomogeneousTransformationMatrix object and HomogeneousVector object.

```
>> A*B

ans =

class HomogeneousTransformationMatrix
    1.0000         0         0         0
    1.0000    0.7071   -0.7071         0
    1.0000    0.7071    0.7071         0
         0         0         0    1.0000

>>  A*translation

ans =

class HomogeneousVector
[1,1,2.4142,0]
```

- disp: creates the output you can see in the examples given above.

2.7 The STUDYparameter Class

Properties of the STUDYparameter Class

- parameters...the 8 STUDY parameters
- dualQuaternion...a Dualquaternion object containing these STUDY parameter

Methods in the STUDYparameter Class

- The constructor's input argument is a vector with 8 components, containing the STUDY parameters or nothing.

```
>> par = STUDYparameter([1 0 0 0 2 3 1 9])

par =

class STUDYparameter
     1
     0
     0
     0
     2
     3
     1
     9
```

However, in this context it is not clear what the components stand for. Therefore it is better to generate STUDYparamter objects by using the associated dual quaternion and the getStudyParameters function of the class Dualquaternion.

```
>> T = Dualquaternion(Quaternion(1,[0 0 0]),...
                      Quaternion(0,[0.5 0 0.5]));
>> R = Dualquaternion(Quaternion(0.9239,[0 0 0.3827]),...
                      Quaternion);
>> Q = T*R;
>> par = Q.getStudyParameters

par =

class STUDYparameter
    0.9239
         0
         0
    0.3827
   -0.1913
    0.4620
```

```
   -0.1913
    0.4620
```

- getHomogeneousMatrix: returns a HomogeneousTransformationMatrix object that describes the same spatial displacement as the STUDYparameter object.

```
>> par.getHomogeneousMatrix

ans =

class HomogeneousTransformationMatrix
    1.0000         0         0         0
    1.0000    0.7071   -0.7071         0
         0    0.7071    0.7071         0
    1.0000         0         0    1.0000
```

- double: returns a double vector
- disp: specific output for STUDYparameter class objects

3 Utility Functions

3.1 The Function getEulerCoords

function [eulerCoords] = getEulerCoords(phi, d)
This function returns *EULER-coordinates* for a given angle and rotation axis.

```
>> a = getEulerCoords(pi/4,[0 0 1])

a =

    0.9239         0         0    0.3827
```

The first 4 components of the *STUDY-parameters* are the *EULER-coordinates*. They are also useful to describe pure rotations with quaternions.

3.2 The Function getSphereCoords

For some examples, the quadric, on which points are interpolated, is chosen as the unit sphere. The getSphereCoords function returns vectors $\in \mathbb{R}^4$ belonging to the points on the sphere. Input arguments are the azimuth- and the altitude-angles of the point.

```
>> getSphereCoords(pi/4,pi/4)

ans =

    1.0000
    0.5000
    0.5000
    0.7071
```

Due to the fact that the homogeneous factor of the vector, returned by this function, is equal to one, the coordinates of this point can be seen directly.

3.3 The Function plotcube

The goal of this work is to interpolate given rigid body motions by pointinterpolation on Study's quadric. In order to visualize the resulting motion, a cube with differently colored sides is used. The vertices of the non-transformed cube are

$$p_1 = (1,1,1),\ p_2 = (-1,1,1),\ p_3 = (-1,-1,1),\ p_4 = (1,-1,1)$$
$$p_5 = (1,1,-1),\ p_6 = (-1,1,-1),\ p_7 = (-1,-1,-1),\ p_8 = (1,-1,-1).$$

The motion is visualized by performing the transformation to the cube. Examples will be shown later.

4 Interpolation

The image of a k-parametric Euclidean motion under Study's kinematic mapping is a k-parametric submanifold on the Study quadric. Especially one-parametric motions are represented by curves on Study's quadric. With the kinematic mapping of Study, motions can be mapped to points in the 7-dimensional projective space $\mathbb{P}^7$. In this space, these points are situated on a special quadric. There is a need for algorithms that interpolate points on quadrics and generate solution curves on that quadric. The aim is to find a rational interpolating curve $c...x = x(t)$ for the points A_i and the corresponding parameter values t_i that is contained in the given hyperquadric, $c \subset Q^{d-1}$ and $x(t_i) \mathrel{\widehat{=}} A_i$, where d is the dimension of the projective space. The following polynoms help to define rational interpolation in projective space.

$$\begin{aligned} p_J(t) &:= \prod_{k \in J} (t - t_k) \\ p_{J\setminus\{i\}}(t) &:= \prod_{k \in J\setminus\{i\}} (t - t_k),\ i \in J \\ p_{J\setminus I}(t) &:= \prod_{k \in J\setminus I} (t - t_k) \quad \text{for } I := \{i_0, ..., i_l\} \subset J \end{aligned}$$

J is the index set of the points to be interpolated, $J = \{0, ..., n\}$ for $A_0, ..., A_n$ and belonging $t_0, ..., t_n$. The solution curve of an interpolation problem for the points A_i, represented by their homogeneous coordinate vectors, has the form

$$c...x(t) = \sum_{i \in J} p_{J\setminus i}(t) \cdot p_{J\setminus i}(t_i) \cdot \omega_i \cdot \mathbf{a_i},$$

where ω_i are weights.

4.1 The Function interpolateOnQuadric

In this section, conditions for the weights ω_i are given so that the curve is located on the hyperquadric Q^{d-1}. Every hyperquadric $Q^{d-1} \subset \mathbb{P}^d$ can be represented using the equation

$$x^T \cdot M \cdot x = 0,$$

where M is a symmetric $(d+1)\times(d+1)$ matrix with rank$(M) > 0$ and x is a point in homogeneous coordinates that is located on the quadric Q^{d-1}. Every quadric has an associated bilinear form

$$\langle x, y \rangle := x^T \cdot M \cdot y.$$

Therefore the equation of the quadric has the form

$$\langle x, x \rangle = 0.$$

That means, if the solution curve c of the interpolation problem is part of the quadric, the condition

$$\langle c, c \rangle = 0$$

holds for all t.
The parametric representation and the calculation for a point on the solution curve for the corresponding parameter value t_i is implemented in the function interpolateOnQuadric, in "docu.m"an example is given. In the following example, the dimension is 3 and the quadric is the unit sphere. The function interpolateOnQuadric has three input arguments.

```
>> interpolateOnQuadric(controlPoints,t, M)
```

controlPoints is a cell array containing the struct controlPoints. In this struct, for each point $\mathbf{a_i} \mathrel{\widehat{=}} A_i$, the homogeneous coordinates and the corresponding parameter value t_i are stored in the variable controlPoints{i}.coords and controlPoints{i}.value. The number of points one wants to interpolate must be odd. For a more detailed explanation see [11]. The second input argument is the parameter value of the calculated point on the solution curve. The last input is the matrix M of the quadric.

```
a = {};
controlPoints.coords = getSphereCoords(0,0);
controlPoints.value = 0;
a{1} = controlPoints;
controlPoints.coords = getSphereCoords(pi/10,-pi/12);
controlPoints.value = 0.166666666;
a{2} = controlPoints;
controlPoints.coords = getSphereCoords(pi/6,pi/8);
controlPoints.value = 0.333333332;
a{3} = controlPoints;
controlPoints.coords = getSphereCoords(pi/4,pi/4);
controlPoints.value = .5;
a{4} = controlPoints;
controlPoints.coords = getSphereCoords(pi/2,pi/2);
controlPoints.value = 0.66666666;
a{5} = controlPoints;


M = diag([-1 1 1 1]);

point1 = [];
tic
for i = 0:0.001:a{5}.value
     point1 = [point1,interpolateOnQuadric(a,i,M)];
end
toc

% normalize the points to plot them in IR^3
for i=1:size(point1,2)
    point1(:,i)=1/point1(1,i).*point1(:,i);
```

```
end

% plot the curve
sphere;
axis equal;
hold on;
X=point1(2,:);
Y=point1(3,:);
Z=point1(4,:);
plot3(X,Y,Z,'LineWidth',2);

% plot the points that are interpolated
for i=1:length(a)
    p = plot3(a{i}.coords(2),a{i}.coords(3),a{i}.coords(4));
    set(p,'Marker','.');
    set(p,'MarkerSize',30);
end
```

The resulting picture is shown below.

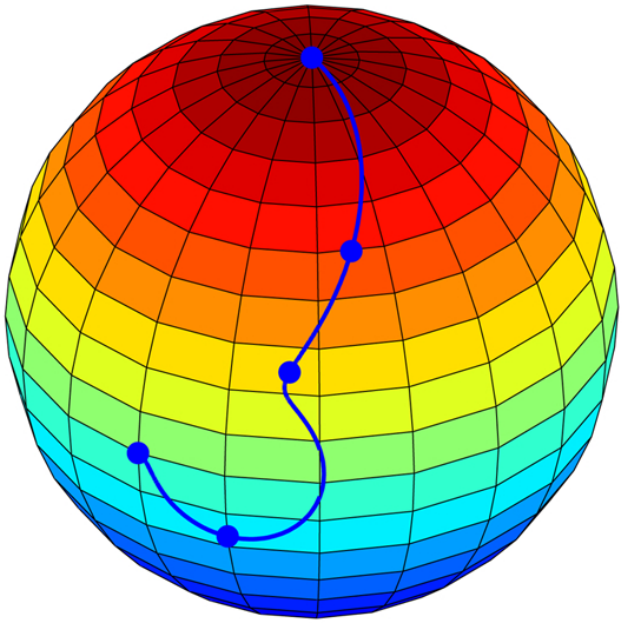


Figure 4.1: Interpolating curve on the unit sphere

This example is fully implemented in the file docu.m. In the next section, a subdivision algorithm that returns the same solution to the problem is presented. Note that this subdivision algorithm works much faster than the direct implementation of the parametrization.

4.2 Aitken Subdivision Algorithm

In this section, a projective extension of the Aitken algorithm is introduced. While the Aitken algorithm in affine space works with ratios on lines, cf. figure 4.2, the projective one works with cross ratios on regular second order curves (conics).

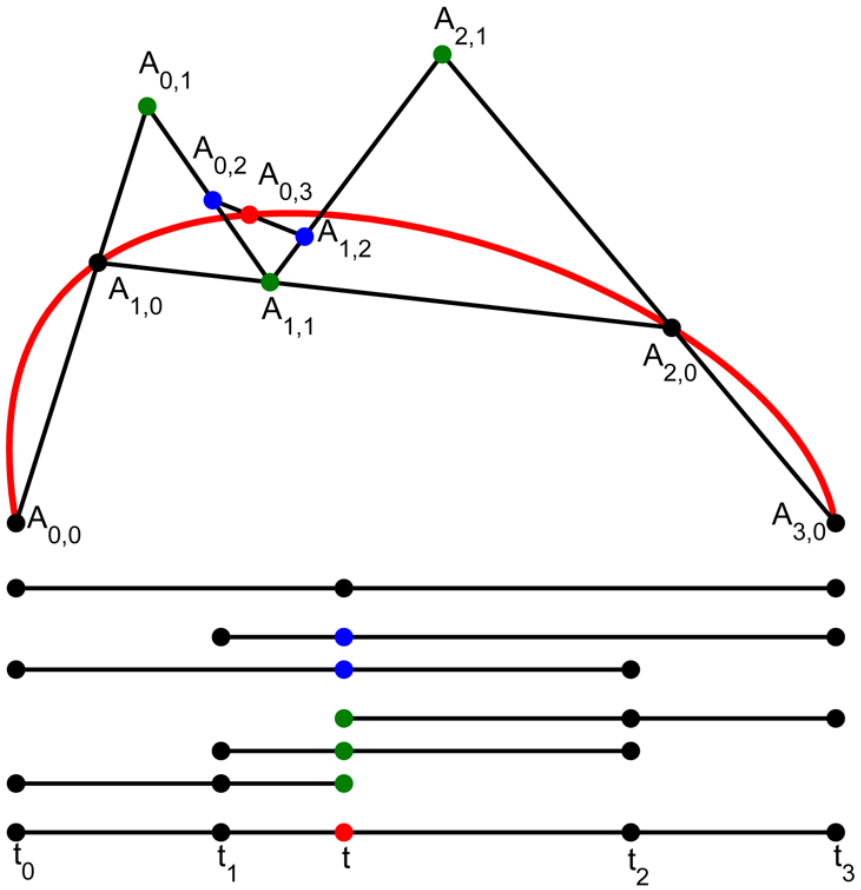


Figure 4.2: Aitken's algorithm in affine space

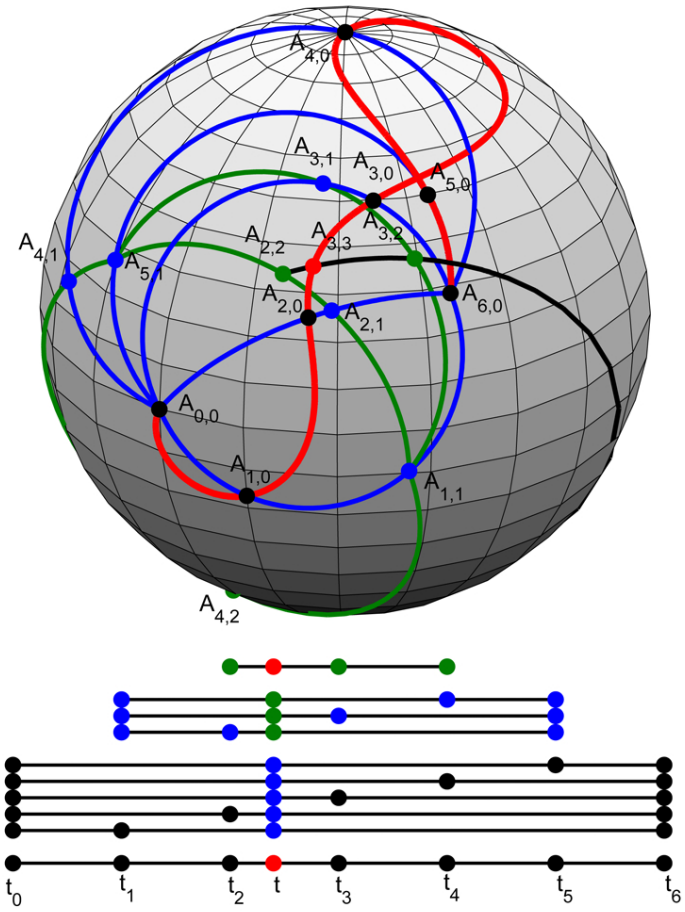


Figure 4.3: Extended Aitken algorithm in $\mathbb{P}^3$

Figure 4.3 illustrates the projective extended Aitken algorithm. Since the quadric is chosen as the sphere, the projective space is $\mathbb{P}^3$. The points to be interpolated are referred to $A_{0,0}, ..., A_{6,0}$. The zero in the second index shows that theses points are the initial points of the algorithm, the zeroth generation. In the first cycle of the algorithm, the planes

$$[A_0, A_1, A_6]_p\,,\ [A_0, A_2, A_6]_p\,, ..., [A_0, A_5, A_6]_p$$

are intersected with the hyperquadric Q^{d-1} (the sphere). In general, the intersection is a regular second order curve. Now, a new point is calculated by the cross ratio of the parameter values of the three points A_0, A_i, A_6 and the parameter value t. This leads to the first generation of points that are colored blue in the figure 4.3 and that are denoted by the one in the second index. Applying this procedure to the first generation points results in the second generation points, colored green. The third generation is the required point that is located on the curve.

In the following an example for the extendend Aitken algorithm is shown which is also contained in the associated m-file docu.m. The function is called as follows:

```
>> aitken(controlPoints,t,M)
```

The input parameters are the same as for the interpolateOnQuadric function. This algorithm creates the same solution curve as interpolateOnQuadric but works much faster. For a picture of the solution curve, see figure 4.1.

```
a = {};
controlPoints.coords = getSphereCoords(0,0);
controlPoints.value = 0;
a{1} = controlPoints;
controlPoints.coords = getSphereCoords(pi/10,-pi/12);
controlPoints.value = 0.166666666;
a{2} = controlPoints;
controlPoints.coords = getSphereCoords(pi/6,pi/8);
controlPoints.value = 0.333333332;
a{3} = controlPoints;
controlPoints.coords = getSphereCoords(pi/4,pi/4);
controlPoints.value = .5;
a{4} = controlPoints;
controlPoints.coords = getSphereCoords(pi/2,pi/2);
controlPoints.value = 0.66666666;
a{5} = controlPoints;


M = diag([-1 1 1 1]);

point1 = [];
tic
for i = 0:0.001:a{5}.value
     point1 = [point1,aitken(a,i,M)];
end
toc

% normalize the points to plot them in IR^3
for i=1:size(point1,2)
    point1(:,i)=1/point1(1,i).*point1(:,i);
end

% plot the curve
sphere;
axis equal;
hold on;
X=point1(2,:);
Y=point1(3,:);
Z=point1(4,:);
plot3(X,Y,Z,'LineWidth',2);

% plot the points that are interpolated
for i=1:length(a)
```

```
    p = plot3(a{i}.coords(2),a{i}.coords(3),a{i}.coords(4));
    set(p,'Marker','.');
    set(p,'MarkerSize',30);
end
```

Feel free to change or extend these examples.

4.3 QB-curves

In affine space, interpolants are constructed with Aitken's algorithm. It is well known that by simply replacing the ratios in this algorithm with constant ratios, one gets a Bèzier curve instead of the Lagrange-interpolant. Now, one could get the idea to replace the cross ratios in the projective version of Aitken's algorithm with constant cross ratios. This leads to QB-curves, Quadric Bèzier curves. For more information see [11]. These curves also have some interesting properties similar to the properties of standard Bèzier curves. Here, these properties are just listed, for details, cf. Anton Gfrerrer [11].

- Projective invariance, the curve is invariantly combined with its control-structure with respect to projective transformations.
- Start-, middle- and endpoint interpolation.
- A change of the interval on which the curve is defined only effects the parametrization of the curve and not the curve itself. In the implemented algorithm, the interval is set to $[0, 1]$.
- The tangent at the startpoint only depends on the two first and the last control points (A_0, A_1, A_n). The tangent at the endpoint only depends on the first and the two last points (A_0, A_{n-1}, A_n).

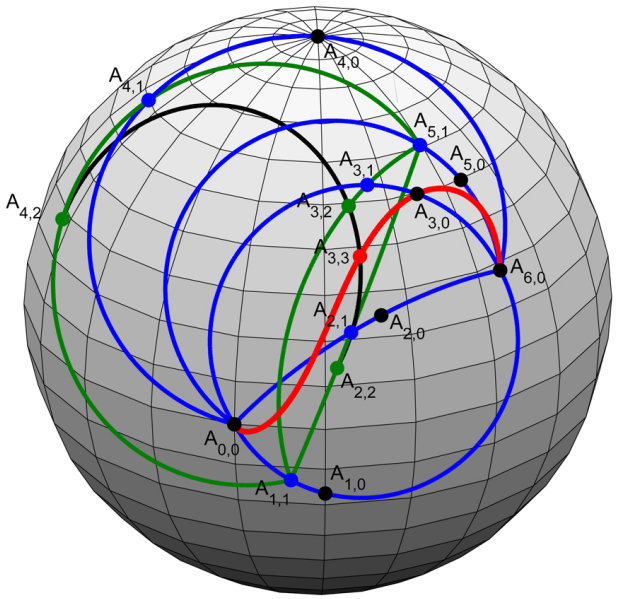


Figure 4.4: QB-curve construction

In figure 4.4, the construction of a QB-curve through a subdivision algorithm is illustrated. The algorithm works like the projective extension of Aitken's algorithm (with constant cross ratios).

```
>> QBinterpolation(a,t,M)
```

An example on the sphere in $\mathbb{P}^3$ is presented below. This example can also be found in docu.m.

```
a = {};
controlPoints.coords = getSphereCoords(0,0);
a{1} = controlPoints;
controlPoints.coords = getSphereCoords(pi/10,-pi/12);
a{2} = controlPoints;
controlPoints.coords = getSphereCoords(pi/6,pi/8);
a{3} = controlPoints;
controlPoints.coords = getSphereCoords(pi/4,pi/4);
a{4} = controlPoints;
controlPoints.coords = getSphereCoords(pi/2,pi/2);
a{5} = controlPoints;


M = diag([-1 1 1 1]);

point1 = [];
tic
for i = 0:0.001:1
     point1 = [point1,QBinterpolation(a,i,M)];
end
toc

% normalize the points to plot them in IR^3
for i=1:size(point1,2)
    point1(:,i)=1/point1(1,i).*point1(:,i);
end

% plot the curve
sphere;
axis equal;
hold on;
X=point1(2,:);
Y=point1(3,:);
Z=point1(4,:);
plot3(X,Y,Z,'LineWidth',2);

% plot the control points
for i=1:length(a)
    p = plot3(a{i}.coords(2),a{i}.coords(3),a{i}.coords(4));
    set(p,'Marker','.');
    set(p,'MarkerSize',30);
end
```

The resulting picture:

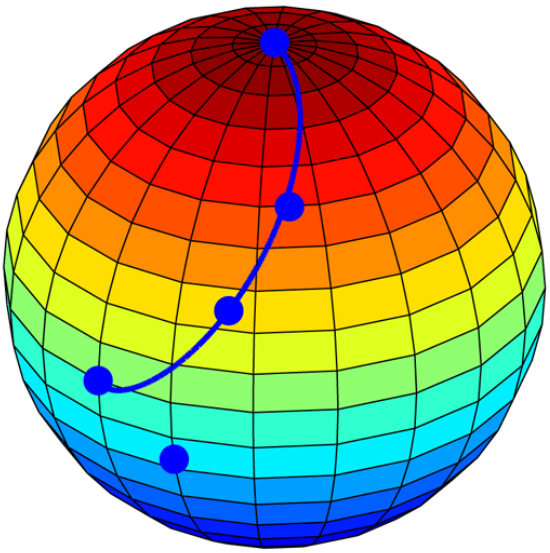


Figure 4.5: QB-curve on the unit sphere

The property that the tangent just depends on three points leads to the idea to construct G^1 continuity between two QB-segments. If the points that are not interpolated are used as design points to change the direction at the common point of both segments, one can get the same tangent direction in this point. For the sphere, this is implemented in the file QBsplines.m

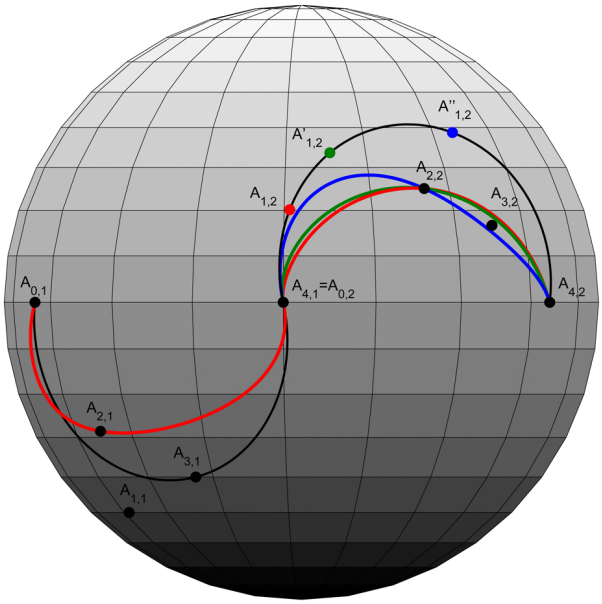


Figure 4.6: G^1 continuity connection

5 Interpolation on Study's Quadric

Hence, all these concepts are extended to Study's quadric. Because all presented algorithms are independent of the dimension of the projective space and the quadric, the concepts can be used for individual problems. An application is for example the construction of ruled surfaces through interpolation on Klein's quadric.

The positions of the moving object are given via dual quaternions representing spatial displacements. Study's quadric has the equation

$$x_1x_5 + x_2x_6 + x_3x_7 + x_4x_8 = 0,$$

therefore the matrix of this quadric has the form:

$$M = \begin{pmatrix} 0 & 0 & 0 & 0 & \frac{1}{2} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & \frac{1}{2} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{2} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{2} \\ \frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{1}{2} & 0 & 0 & 0 & 0 \end{pmatrix}.$$

5.1 Aitken Algorithm on Study's Quadric

The first example interpolates three positions

```
M = diag([.5 .5 .5 .5],-4) + diag([.5 .5 .5 .5],4);
a={};
% first point on the study quadric defined via dual quaternions
T = Dualquaternion(Quaternion(1,[0 0 0]),Quaternion(0,[0 0 0.5]));
Q = T;
controlPoints.coords = double(Q.getStudyParameters);
controlPoints.value = 0;
plotCube(Q.getStudyParameters,2);
hold on;
a{1} = controlPoints;
% second point
T = Dualquaternion(Quaternion(1,[0 0 0]),Quaternion(0,[2 0 3]));
R = Dualquaternion(Quaternion(0.9239,[0 .5 0.3827]),Quaternion);
Q = T*R;
controlPoints.coords = double(Q.getStudyParameters);
controlPoints.value = 0.25;
```

```
plotCube(Q.getStudyParameters,2);
a{2} = controlPoints;
% third point
T = Dualquaternion(Quaternion(1,[0 0 0]),Quaternion(0,[3 -0.5 0]));
R = Dualquaternion(Quaternion( 0.9808 ,[0.1379 0 0.1379]),...
                   Quaternion);
Q = T*R;
controlPoints.coords = double(Q.getStudyParameters);
controlPoints.value = 0.5;
plotCube(Q.getStudyParameters,2);
a{3} = controlPoints;

point1 = [];
for i = 0:0.01:a{end}.value
    point1 = [point1,aitken(a,i,M)];
end

for i = 1:size(point1,2)
    h1 = plotCube(STUDYparameter(point1(:,i)));
    drawnow;
%     delete(h1)
end
```

Here, T is a Dualquaternion object representing a pure translation and R is a Dualquaternion object representing a pure rotation. The product of T and R is the Dualquaternion object Q that represents the spatial displacement. The projective space is $\mathbb{P}^7$ and the entries in the Dualquaternion object Q are the point coordinates of the motions in this space. The resulting motion is shown below:

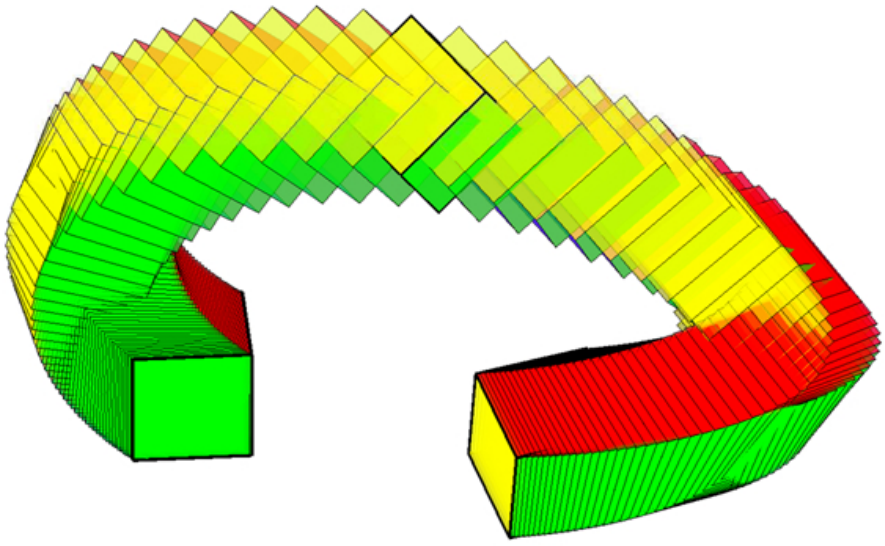


Figure 5.1: Aitken algorithm for 3 positions

5.2 QB-Curves on Study's Quadric

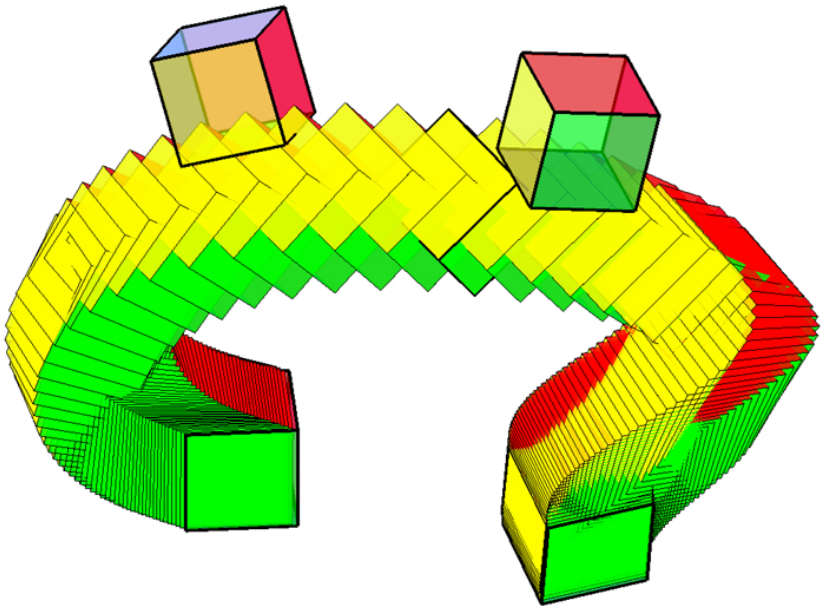
In this section, the same positions as in the last example are interpolated. Instead of the projective extended Aitken algorithm, the QB-algorithm is used. Thus, there are two positions that are not interpolated and that can be used to fit the motion according to the designer's wishes.

```
a={};
% first point on the study quadric defined via dual quaternions
T = Dualquaternion(Quaternion(1,[0 0 0]),Quaternion(0,[0 0 0.5]));
Q = T;
controlPoints.coords = double(Q.getStudyParameters);
plotCube(Q.getStudyParameters,1.5);
hold on;
a{1} = controlPoints;
% second point
T = Dualquaternion(Quaternion(1,[0 0 0]),Quaternion(0,[0 0.5 4]));
R = Dualquaternion(Quaternion(0.7071,[0.4082  0.4082  0.4082]),...
                   Quaternion);
Q = T*R;
controlPoints.coords = double(Q.getStudyParameters);
plotCube(Q.getStudyParameters,1.5);
a{2} = controlPoints;
%---
T = Dualquaternion(Quaternion(1,[0 0 0]),Quaternion(0,[2 0 3]));
R = Dualquaternion(Quaternion(0.9239,[0 .5 0.3827]),Quaternion);
Q = T*R;
controlPoints.coords = double(Q.getStudyParameters);
plotCube(Q.getStudyParameters,1.5);
a{3} = controlPoints;
%---
T = Dualquaternion(Quaternion(1,[0 0 0]),Quaternion(0,[3 -0.5 4]));
R = Dualquaternion(Quaternion(0.9877,[0 0.1106 0.1106]),...
                   Quaternion);
Q = T*R;
controlPoints.coords = double(Q.getStudyParameters);
plotCube(Q.getStudyParameters,1.5);
a{4} = controlPoints;
%---
T = Dualquaternion(Quaternion(1,[0 0 0]),Quaternion(0,[3 -0.5 0]));
R = Dualquaternion(Quaternion( 0.9808 ,[0.1379 0 0.1379]),...
                   Quaternion);
Q = T*R;
controlPoints.coords = double(Q.getStudyParameters);
plotCube(Q.getStudyParameters,1.5);
a{5} = controlPoints;

tic
```

```
point1 = [];
for i = 0:0.01:1
    point1 = [point1,QBinterpolation(a,i,M)];
end
toc

for i = 1:size(point1,2)
    h1 = plotCube(STUDYparameter(point1(:,i)));
    drawnow;
%     delete(h1)
end
```



The next example shows the possible influence of the design points. Only the second and fourth points were changed. The interpolated positions are the same as in the previous two examples.

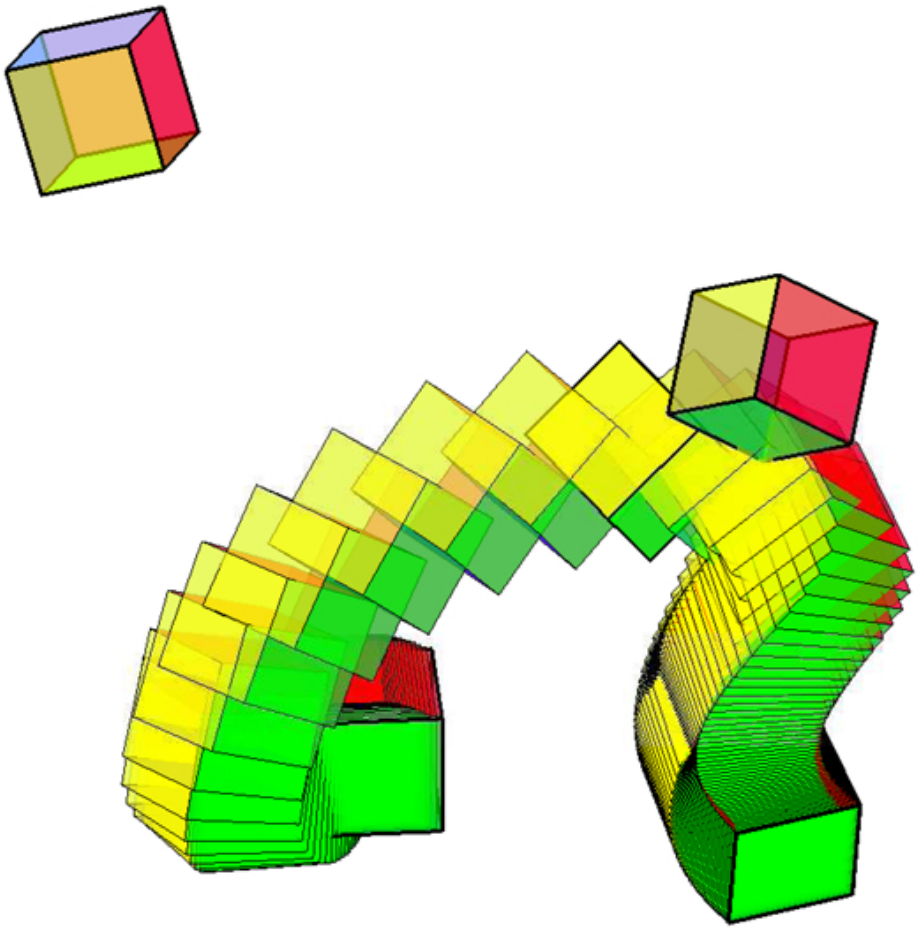
```
M = diag([.5 .5 .5 .5],-4) + diag([.5 .5 .5 .5],4);

a={};
% first point on the study quadric defined via dual quaternions
T = Dualquaternion(Quaternion(1,[0 0 0]),Quaternion(0,[0 0 0.5]));
Q = T;
controlPoints.coords = double(Q.getStudyParameters);
plotCube(Q.getStudyParameters,1.5);
hold on;
a{1} = controlPoints;
% second point
T = Dualquaternion(Quaternion(1,[0 0 0]),Quaternion(0,[-2 0.5 6]));
R = Dualquaternion(Quaternion(0.7071,[0.4082 0.4082 0.4082]),...
```

```
                    Quaternion);
Q = T*R;
controlPoints.coords = double(Q.getStudyParameters);
plotCube(Q.getStudyParameters,1.5);
a{2} = controlPoints;
%---
T = Dualquaternion(Quaternion(1,[0 0 0]),Quaternion(0,[2 0 3]));
R = Dualquaternion(Quaternion(0.9239,[0 .5 0.3827]),Quaternion);
Q = T*R;
controlPoints.coords = double(Q.getStudyParameters);
plotCube(Q.getStudyParameters,1.5);
a{3} = controlPoints;
%---
T = Dualquaternion(Quaternion(1,[0 0 0]),Quaternion(0,[3 -0.5 4]));
R = Dualquaternion(Quaternion(0.8660,[0.3536  0.3536      0   ]),...
                    Quaternion);
Q = T*R;
controlPoints.coords = double(Q.getStudyParameters);
plotCube(Q.getStudyParameters,1.5);
a{4} = controlPoints;
%---
T = Dualquaternion(Quaternion(1,[0 0 0]),Quaternion(0,[3 -0.5 0]));
R = Dualquaternion(Quaternion( 0.9808 ,[0.1379 0 0.1379]),...
                    Quaternion);
Q = T*R;
controlPoints.coords = double(Q.getStudyParameters);
plotCube(Q.getStudyParameters,1.5);
a{5} = controlPoints;

point1 = [];
for i = 0:0.01:1
    point1 = [point1,QBinterpolation(a,i,M)];
end

for i = 1:size(point1,2)
    h1 = plotCube(STUDYparameter(point1(:,i)));
    drawnow;
%     delete(h1)
end
```



For spline constructions on Study's quadric see the files QBspline_Study.m or QBspline_Study_2.m. The bibliography includes suggestions for further reading.

Bibliography

[1] B. Roth B. Ravani. Mappings of spatial kinematics. *Journal of Mechanisms Transmissions and Automation in Design-Transactions of the ASME*, 3:341–347, 1984.

[2] Oene Bottema Bernard Roth. *Theoretical Kinematics*. Dover Publications Inc, 1990.

[3] Wilhelm Blaschke. *Kinematik und Quaternionen*. VEB Deutscher Verlag der Wissenschaften, 1960.

[4] Wolfgang Boehm. Some remarks on quadrics. *Computer Aided Geometric Design*, 10:231–236, 1993.

[5] Kahmann Jürgen Boehm Wolfgang, Farin Gerald. A survey of curve and surface methods in cagd. *Computer Aided Geometric Design*, 1:1–60, 1984.

[6] Gerrit Bol. *Projektive Differentialgeometrie 1.Teil*. Vandenhoeck & Ruprecht, 1950.

[7] Gert Bär. *Eine Einführung für Ingenieure und Naturwissenschaftler*. Teubner, 2001.

[8] Carl de Boor. *A Practical Guide tp Splines*. Springer Verlag New York Heidelberg Berlin, 1978.

[9] Gerald Farin. *Curves and surfaces for CAGD*. Morgan Kaufmann Publishers, 2002.

[10] Gerd Fischer. *Analytische Geometrie*. Vieweg Verlag von R. Oldenburg, 2001.

[11] Anton Gfrerrer. On the construction of rational curves on hyperquadrics. *Grazer Mathematische Berichte*, 1999.

[12] Anton Gfrerrer. Rational interpolation on a hypersphere. *Computer-Aided Geometric Design*, 16:21–37, 1999.

[13] Anton Gfrerrer. Study's kinematic mapping - a tool for motion design. *editors J. Lenarcic and M. M. Stanisic - Recent Advances in Robot Kinematics*, pages 7–16, 2000.

[14] Johannes Wallner Helmut Pottmann. *Computational Line Geometry*. Springer-Verlag Berlin, 2001.

[15] Josef Hoschek. *Liniengeometrie*. Bibliographisches Institut AG Zürich, 1971.

[16] Bert Jüttler. Zur konstruktion rationaler kurven und flächen auf quadriken. *zum 60. Geburtstag von Herrn Professor Dr. Oswald Giering, Darmstadt*.

[17] Bert Jüttler. Visualization of moving objects using dual quaternion curves. *Computers & Graphics*, 18(3):315–326, 1994.

[18] Daniel Klawitter. Bewegungsdesign mit der kinematischen Abbildung. Master's thesis, Technische Universität Dresden, 2010.

[19] Wayne Tiller Les Piegl. *The NURBS Book*. Springer, 1995.

[20] Hans Sachs Waldemar Steinhilper Manfred Husty, Adolf Karger. *Kinematik und Robotik*. Springer-Verlag Berlin Heidelberg, 1997.

[21] J. Michael McCarthy. *An Introduction to Theoretical Kinematics*. The MIT Press, Cambridge, Massachusetts, 1990.

[22] X.-W. Chang M.John D. Hayes, T. Luu. Kinematic mapping application to approximate type and dimension synthesis of planar mechanisms. 2004.

[23] Hans Robert Müller. *Sphärische Kinematik*. VEB Deutscher Verlag der Wissenschaften, 1962.

[24] Hans Robert Müller. Die kinematischen abbildungen im dreidimensionalen raum. *Monatshefte für Mathematik*, pages 252–258, 2005.

[25] Wilhelm Blaschke Müller Hans Robert. *Ebene Kinematik*. Verlag von R. Oldenburg, 1956.

[26] S. Shankar Sastry Murray M. R., Zexiang Li. *A Mathematical Introduction to Robotic Manipulation*. Crc Pr Inc, 1994.

[27] Otto Röschel. Rational motion design - a survey. *Computer-Aided Design*, 30:169–178, 1998.

[28] J.M. Selig. *Geometric fundamentals of robotics*, volume 2. Springer, 2005.

[29] Karl Stubecker. Direkte herleitung der darstellung der bewegungen der ebene durch studysche quaternionen. *zum 70. Geburtstag von Ph. Furtwängler, Wien*.

[30] Eduard Study. Von den bewegungen und umlegungen (i u. ii abhandlung). *Math. Ann.*, 39:441–566, 1869.

[31] Eduard Study. *Geometrie der Dynamen*. Teubner, 1903.

[32] Ernst August Weiss. *Einführung in die Liniengeometrie und Kinematik*. Teubner Verlag Leipzig, 1935.