DOCUMENTATION - Curves

Jason Chemin

April 2019

Contents

1 Curves **2**
- 1.1 Definition of different types of curves . . . 2
- 1.2 curve_abc . . . 2
- 1.3 Bezier curve . . . 3
  - 1.3.1 Definition . . . 3
  - 1.3.2 Template and constructors . . . 6
  - 1.3.3 Bezier in C++ . . . 7
  - 1.3.4 Bezier in Python . . . 8
- 1.4 Polynomial . . . 9
  - 1.4.1 Template and constructors . . . 10
  - 1.4.2 Polynomial in C++ . . . 11
  - 1.4.3 Polynomial in python . . . 12
- 1.5 Cubic Hermite spline . . . 13
  - 1.5.1 Definition . . . 13
  - 1.5.2 Template and constructors . . . 14
  - 1.5.3 Cubic Hermite in C++ . . . 14
  - 1.5.4 Cubic Hermite in Python . . . 16
- 1.6 piecewise curve . . . 16
  - 1.6.1 Template and constructors . . . 17
  - 1.6.2 Piecewise curve in C++ . . . 18
  - 1.6.3 Piecewise curve in Python . . . 20
- 1.7 Exact cubic spline . . . 20
  - 1.7.1 Template and constructors . . . 21
  - 1.7.2 Exact cubic in C++ . . . 22
  - 1.7.3 Exact cubic in Python . . . 23

2 Conversion of cubic curves **24**
- 2.1 Basic cubic curves conversion . . . 24
  - 2.1.1 Conversion of basic cubic curves in C++ . . . 24
  - 2.1.2 Conversion of basic cubic curves in Python . . . 25
- 2.2 Piecewise cubic curves conversion . . . 25
  - 2.2.1 Conversion of piecewise cubic curves in C++ . . . 25
  - 2.2.2 Conversion of piecewise cubic curves in Python . . . 26

3 Serialization **26**
3.0.1 Serialization in C++ 27
3.0.2 Serialization in Python 27

1 Curves

1.1 Definition of different types of curves

Piecewise curve : a curve that is formed from a collection of simple segments strung end-to-end so that their junctions are fairly smooth. There are many kinds of spline that use different mathematics functions, but most of splines consist of polynomial segments where we can add some constraints of continuity.

Bezier curve : Polynomial parametric curve defined by several control points. *Properties* :

- Pass through the first and last control points.
- Curve is contained in the convex hull of its defining control point and smoothly follows the control points.
- The degree of polynomial defining the curve segment with n control points is $n - 1$.

Cubic spline : spline where each polynomial segments has degree 3.

Exact Cubic spline* : Cubic spline optimized as in the paper : "Task-Space Trajectories via Cubic Spline Optimization" By J. Zico Kolter and Andrew Y.ng (ICRA 2009). *Properties* :

- Pass through each waypoint.
- C2 continuous.

Cubic hermite spline : spline where each segment is a third degree polynomial and described by its extremum positions end derivatives.

- Pass through all positions given.
- No convex hull guarantees.
- C1 continuous.

1.2 curve_abc

All curves in our implementation inherits from class curve_abc and have to implement the following functions :

```
    // Evaluation of the cubic spline at time t.
    virtual point_t operator()(const time_t t)
```

```
    // Evaluate the derivative of order N of curve at time t.
    virtual point_t derivate(const time_t t, const std::size_t order)
```

```
// Get dimension of curve.
virtual std::size_t dim()
```

```
// Get the minimum time for which the curve is defined.
virtual time_t min()
```

```
// Get the maximum time for which the curve is defined.
virtual time_t max()
```

To serialize a curve, curve_abc inherits from boost::serialization::Serializable where serialization functions are defined in include/curves/serialization/archive.hpp, so it has to implement the function :

```
template<class Archive>
void serialize(Archive& ar, const unsigned int version)
```

1.3 Bezier curve

A Bezier curve is a parametric curve defined by a set of control points $p_0, p_1, ..., p_n$ where n is its order. The curve passes through its first and last control point. All Bezier curves of order superior to 3 can be represented as Bezier curves of order 3 or less joined end-to-end, where the last control point of one curve coincides with the first control point of the next curve.

If you want to create a bezier using five or more control points, we recommend you instead to create a piecewise bezier curve containing bezier curves of four or less control points (See section piecewise curve).

1.3.1 Definition

A bezier curve with $n+1$ is represented by :

$$x(t) = \sum_{i=0}^{N} B_i^N((t - t_{min})/T) \cdot p_i \tag{1}$$

where

$$B_i^n((t - t_{min})/T) = \binom{n}{i} \cdot ((t - t_{min})/T)^i \cdot (1 - (t - t_{min})/T)^{n-i} \tag{2}$$

are Bernstein polynomials, p_i is the control point at index i, $t \in [t_{min}, t_{max}]$ and $T = t_{max} - t_{min}$.

To add constraints on initial and final velocity on a Bezier curve x(t), we can set position of control points accordingly to its derivative $x'(t)$ and $x''(t)$. For more details, read subsection III.A of paper "C-CROC: Continuous and Convex Resolution of Centroidal dynamic trajectories for legged robots in multi-contact scenarios" by Pierre Fernbach et al. .

Order of Bezier curve $x(t)$	Derivative $x'(t)$	$x'(t=0)$	$x'(t=1)$
1 (Linear)	$(p_1 - p_0)/T$	$(p_1 - p_0)/T$	$(p_1 - p_0)/T$
2 (Quadratic)	$(2(1-t)(p_1 - p_0) + 2t(p_2 - p_1)))/T$	$2(p_1 - p_0)/T$	$2(p_2 - p_1)/T$
3 (Cubic)	$(3(1-t)^2(p_1 - p_0) + 6t(1-t)(p_2 - p_1) + 3t^2(p_3 - p_2))/T$	$3(p_1 - p_0)/T$	$3(p_3 - p_2)/T$

Table 1: Derivative order 1 of bezier curve defined between [0,1] at initial and final time.

Order of Bezier curve $x(t)$	$x''(t=0)$	$x''(t=1)$
1 (Linear)	$(p_1 - p_0)/T^2$	$(p_1 - p_0)/T^2$
2 (Quadratic)	$2(p_1 - p_0)/T^2$	$2(p_2 - p_1)/T^2$
3 (Cubic)	$3(p_1 - p_0)/T^2$	$3(p_3 - p_2)/T^2$

Table 2: Derivative order 2 of bezier curve defined between [0,1] at initial and final time.

Our method to ensure that velocity and acceleration constraints are respected on a bezier curve, consists in adding two control points P_{c1}, P_{c2} after the first one P_0 and two, Pc_{n-1} and Pc_{n-2}, before the last one P_n. The derivatives order 1 and 2 of a Bezier curve for N control points are :

$$\begin{cases} x'(t) = \frac{N}{T} \cdot \sum_{i=0}^{N-1} B_i^{N-1}(t) \cdot (P_{i+1} - P_i) \\ x''(t) = \frac{N \cdot (N-1)}{T^2} \cdot \sum_{i=0}^{N-2} B_i^{N-2}(t) \cdot (P_{i+2} - 2P_{i+1} + P_i) \end{cases} \tag{3}$$

At initial and final time, $t = T_{min}$ and $t = T_{max}$ and $T = T_{max} - T_{min}$, the derivatives are equal to :

$$\begin{cases} x'(0) = \frac{N}{T} \cdot (P_1 - P_0) \\ x''(0) = \frac{N}{T^2} \cdot (N-1) \cdot (P_2 - 2P_1 + P_0) \\ x'(1.0) = \frac{N}{T} \cdot (P_N - P_{N-1}) \\ x''(1.0) = \frac{N}{T^2} \cdot (N-1) \cdot (P_N - 2P_{N-1} + P_{N-2}) \end{cases} \tag{4}$$

By adding Pc_1, Pc_2 right after the first control point P_0 and Pc_{n-1}, Pc_{n-2} before the last control points P_N to bezier curve. We can set them as followed to respect the constraints fixed on acceleration and velocities at initial and final time :

$$\begin{cases} Pc_1 = P_0 + \dfrac{constraints.init_vel}{N-1} \cdot T \\ Pc_2 = 2Pc_1 - P_0 + \dfrac{constraints.init_acc}{(N-1)\cdot(N-2)} \cdot T^2 \\ Pc_{N-1} = P_N - \dfrac{constraints.end_vel}{N-1} \cdot T \\ Pc_{N-2} = 2Pc_{N-1} - P_N + \dfrac{constraints.end_acc}{(N-1)\cdot(N-2)} \cdot T^2 \end{cases} \tag{5}$$

Evaluation of the curve at time t has been implemented in three different ways. First, the evaluation of **Bernstein** which consists in solving the equation (1).

The second one is the evaluation **De Casteljau** which computes the $N-1$ centroids of parameters $(t, 1-t)$ of consecutive N control points of bezier curve, and perform it iteratively until getting one point in the list which will be the evaluation of bezier curve at time t.

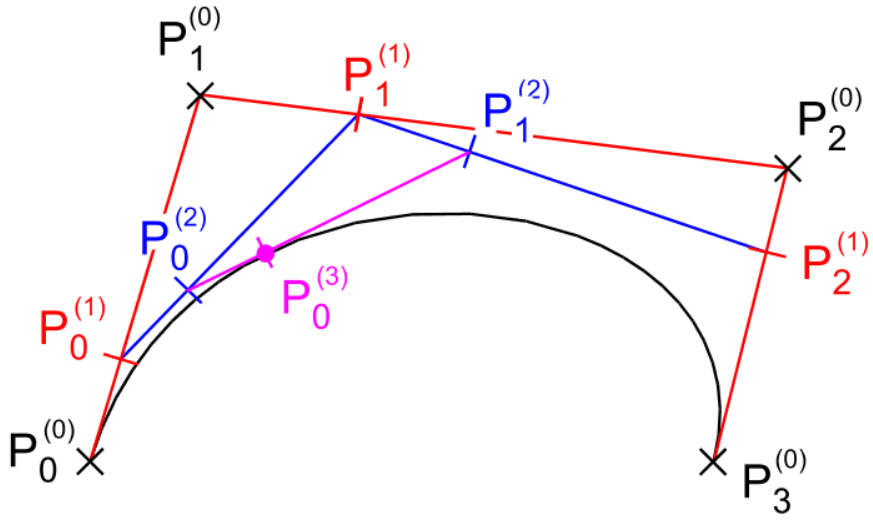

Figure 1: Evaluation of Bezier curve using De Casteljau algorithm.

Finally, the last one implemented is **Horner** evaluation which is the one used by default in our algorithm as it is faster than the two others. We apply the Horner's scheme in order to reduce complexity of calculation of this polynomial. Applying it transforms equation (1) into $x(t) = (1-t)^N(\sum_{i=0}^{N}\binom{N}{i}\frac{1-t}{t}^i P_i)$ reduces computation time.

In this implementation we use the Horner evaluation which is faster for low order bezier curve, but which is less stable with higher orders. Otherwise we

should use De Casteljau which is more stable and better to use with high orders.

1.3.2 Template and constructors

The template for this class is composed of (See applications of bezier_curve.h) :
- typename Time=double
- typename Numeric=Time
- bool Safe=false
- typename Point=Eigen::Matrix<Numeric, Eigen::Dynamic, 1 >
The type of point used by default is of dynamic-size. Size of control points given in the constructor will be of the same size as all points returned by bezier_curve.
The class bezier_curve has the following constructors :

```
bezier_curve() // default constructor

bezier_curve(In PointsBegin, In PointsEnd,
             const time_t T_min=0.0,
             const time_t T_max=1.0,
             const time_t mult_T=1.0)

bezier_curve(In PointsBegin, In PointsEnd,
             const curve_constraints_t& constraints
             const time_t T_min=0.0,
             const time_t T_max=1.0,
             const time_t mult_T=1.0)

bezier_curve(const bezier_curve& other)
```

Where parameters of constructors are :

- ***PointBegin*** and ***PointsEnd*** : iterators pointing to the first and last element of a control point container.;
The actual implementation of bezier curve allows you to enter any number of control points as long as it is superior or equal to one. But in most of cases if the degree is superior to 3, it is prefered to create a composite Bezier curve with segments of degree inferior or equal to 3.

- **T** : Upper bound of time range which is between $[0; T]$ (default $[0; 1]$). This value is then used in evaluation of the curve at time t. In these evaluation, we divide the time by T to bring interval $[0; T]$ to $[0; 1]$;

- ***mult_T*** : By default equals to 1.0. Each time the curve the curve is derived, this value will be multiply by $1/(T_{max} - T_{min})$. The evaluation of curve at time t is multiplied by mult_T. For more details, read subsection III.A of paper "C-CROC: Continuous and Convex Resolution of Centroidal dynamic trajectories for legged robots in multi-contact scenarios" by Pierre Fernbach and Steve Tonneau.

- ***constraints*** : constraints applied on start and end point of the curve. Structure curve_constraint contains 4 instance variables to set : *init_vel*,*init_acc*, *end_vel*, *end_acc* which correspond to initial and final velocities and accelerations.

- ***other*** : the bezier curve to copy.

1.3.3 Bezier in C++

Here are some possible applications using the class bezier_curve.h in C++ :

```
#include "curves/bezier_curve.h"
using namespace curves;
```

```
// Template for bezier curve
// Points in bezier can have any dimension
typedef Eigen::VectorXd pointX_t;
typedef double time_t;
typdef double num_t;
typedef bezier_curve  <time_t, num_t,
                       true, pointX_t> bezier_curve_t;
// We will use points of dimension 3 and 4 for the example
typedef Eigen::Vector3d point3_t;
typedef Eigen::Vector4d point4_t;

// Creation of control points in 3D : P0, P1, P2, P3
point3_t P0(1,2,3);
point3_t P1(2,3,4);
point3_t P2(3,4,5);
point3_t P3(3,6,7);

// Creation of control points in 4D : P4_0, P4_1, P4_2
point4_t P4_0(1,2,3,4);
point4_t P4_1(2,3,4,5);
point4_t P4_2(3,4,5,6);

// Creation of the container of control points
std::vector<point3_t> cp;
std::vector<point4_t> cp4;

// Creation of bezier curve 4D defined between [0.,1.]
cp4.push_back(P4_0);
cp4.push_back(P4_1)
cp4.push_back(P4_2)
bezier_curve_t bc_4D(cp4.begin(),cp4.end());

// Create some 3D bezier curves defined between [0.,1.]
// Two control points = Degree 1 (Linear Bezier curve)
cp.push_back(P0);
cp.push_back(P1);
bezier_curve_t bc1(cp.begin(),cp.end());
// Three control points = Degree 2 (Quadratic Bezier curve)
cp.push_back(P2);
bezier_curve_t bc2(cp.begin(),cp.end());
// Four control points = Degree 3 (Cubic Bezier curve)
```

```
cp.push_back(P3);
bezier_curve_t bc3(cp.begin(),cp.end());

// Value of curve at t=0 and t=1
point3_t res = bc3(0.);
res = bc3(1.);

// Value of derivative curve at t=0.5
res = bc3.derivate(0.5,1); // derivative order 1
res = bc3.derivate(0.5,2); // derivative order 2

// Get the derivative or primitive curve at any time t
// (Recommended if derivative computed several times)
// derivative order 1
bezier_curve_t bc_der1 = bc3.compute_derivate(1);
res = bc_der1(0.5); // equivalent to bc3.derivate(0.5,1)
// primitive order 1
bezier_curve_t bc_prim1 = bc3.compute_primitive(1);

// Time reparameterization
time_t T_min = 1.;
time_t T_max = 3.; // Bezier curve defined between [1.0,3.0]
bezier_curve_t bc3_T_modif(cp.begin(),cp.end(), T_min, T_max);
res = bc3_T_modif(T_min); // value at t=T_min equals to bc3(0.)
res = bc3_T_modif(T_max); // value at t=T_max equals to bc3(1.)

// Add constraints on curve
bezier_curve_t::curve_constraints_t constraints;
constraints.init_vel = point3_t(-1.,-1.,-1.);
constraints.init_acc = point3_t(-2.,-3.,-2.);
constraints.end_vel = point3_t(1.,2.,1.);
constraints.end_acc = point3_t(3.,2.,3.);
bezier_curve_t bc4(beg,end,constraints,T_min,T_max);
bezier_curve_t bc4_der1 = bc4.compute_derivate(1);
bezier_curve_t bc4_der2 = bc4.compute_derivate(2);
res = bc4_der1(T_min); // equal to init_vel
res = bc4_der1(T_max); // equal to end_vel
res = bc4_der2(T_min); // equal to init_acc
res = bc4_der2(T_max); // equal to end_acc

// Upper and Lower bound of definition interval
T_min = bc3_T_modif.min(); // Equal to 1.0
T_max = bc3_T_modif.max(); // Equal to 3.0

// Get waypoints
bezier_curve_t::t_point_t waypoints = bc3.waypoints();
bezier_curve_t::point_t first_waypoint = bc3.waypointAtIndex(0);
```

1.3.4 Bezier in Python

We can use bezier_curve.h through curves.so as followed :

```
from curves import bezier, curve_constraints

# Creation of bezier in 3D with two control points in [0,1]
P3_0 = [1., 2., 3.]
P3_1 = [4., 5., 6.]
```

```
waypoints = matrix([P3_0,P3_1]).transpose()
bc_3dim = bezier(waypoints)
res = bc_3dim(1.) # value at t=1., equal to P3_1

# Creation of bezier in 6D with two control points in [0,1]
P6_0 = [1., 2., 3., 7., 5., 5.]
P6_1 = [4., 5., 6., 4., 5., 6.]
waypoints6 = matrix([P6_0,P6_1]).transpose()
bc_6dof = bezier6(waypoints6)
res = bc_6dof(0.) # value at t=0., equal to P6_0

# Time reparameterization
T_min = 0.0
T_max = 2.0 # Time range will be [0.,2.]
bc_3dim_T_modified = bezier3(waypoints3, T_min, T_max)
res = bc_3dof_T_modified(T_min) # at t=T_min equals to P3_0
res = bc_3dof_T_modified(T_max) # at t=T_max equals to P3_1

# Add constraints on curve
c = curve_constraints()
c.init_vel = matrix([0., 1., 1.]).transpose()
c.end_vel = matrix([0., 1., 1.]).transpose()
c.init_acc = matrix([0., 1., -1.]).transpose()
c.end_acc = matrix([0., 10., 1.]).transpose()
bc_3dim_constraints = bezier3(waypoints, c, T_min, T_max)

# Derivative at time t
res = bc_3dim_constraints.derivate(T_max,1) # Equals to end_vel
res = bc_3dim_constraints.derivate(T_max,2) # Equals to end_acc

# Derivative & Primitive curve
bc_derived = bc_3dim_constraints.compute_derivate(1)
bc_primitive = bc_3dim_constraints.compute_primitive(1)

# Upper and Lower bound of definition interval
Tmin = bc_3dim_T_modified.min() # Equal to T_min
Tmax = bc_3dim_T_modified.max() # Equal to T_max

# Get waypoints
nb_waypoints = bc_3dim_T_modified.nbWaypoints
waypoint_first_index = bc_3dim.waypointAtIndex(0)
waypoint_last_index = bc_3dim.waypointAtIndex(nb_waypoints-1)
```

1.4 Polynomial

Represents a polynomial of an arbitrary order defined on the interval $[t_{min}, t_{max}]$. This polynomial follows the equation :

$$x(t) = a + b \cdot (t - t_{min}) + ... + d \cdot (t - t_{min})^N \tag{6}$$

where N is the order and $t \in [t_{min}, t_{max}]$.

1.4.1 Template and constructors

The template for this class is composed of (See applications of polynomial.h) :
typename Time=double
typename Numeric=Time
bool Safe=false
typename Point=Eigen::Matrix<Numeric, Dim, 1 >
typename T_Point=std::vector<Point,Eigen::aligned_allocator<Point>>
The type of point used by default is of dynamic-size. Dimension of points given in the constructor as coefficients will be the same as all points returned by polynomial.

The class polynomial has the following constructors :

```
polynomial() // Default constructor

polynomial(const coeff_t& coefficients,
           const time_t min, const time_t max)

polynomial(const T_Point& coefficients,
           const time_t min, const time_t max)

template<typename In>
polynomial(In zeroOrderCoefficient, In out,
           const time_t min, const time_t max)

polynomial(const Point& init,const Point& d_init,
           const Point& end, const Point& d_end,
           const time_t min, const time_t max)

polynomial(const Point& init,const Point& d_init,
           const Point& dd_init,
           const Point& end, const Point& d_end,
           const Point& dd_end,
           const time_t min, const time_t max )

polynomial(const polynomial& other)
```

Parameters of constructors are :

- ***coefficients*** : coefficients of polynomial represented in 2 different ways. In the first constructor, *coeff_t* is a reference to an Eigen Matrix where each column is a coefficient, from the zero order coefficient, up to the highest order. Polynomial order is given by the number of columns minus one. In the second constructor, *T_Point&* is a container with all polynomial coefficients, starting with the zero coefficient, up to the highest order. Polynomial order is given by the size of a coefficient minus one.

- ***min*** : lower bound on interval definition of the curve.

- ***max*** : upper bound on interval definition of the curve.

- ***zeroOrderCoefficient*** : iterator pointing to the first element of a structure containing the coefficients. It corresponds to the zero degree coefficient.

- ***out*** : iterator pointing to the last element of a structure containing the coefficients.

- ***init / end*** : initial and ending position of the curve.

- ***d_init / d_end*** : initial and ending velocity of the curve.

- ***dd_init / dd_end*** : initial and ending acceleration of the curve.

- ***other*** : polynomial to copy.

1.4.2 Polynomial in C++

Here is an example of application using the class polynomial.h :

```
#include "curves/polynomial.h"
using namespace curves;
```

```
// Template
// We can create polynomial of any dimension
typedef Eigen::VectorXd pointX_t;
typedef std::vector<pointX_t,
                    Eigen::aligned_allocator<pointX_t>
                   >  t_pointX_t;
typedef double time_t;
typdef double num_t;
typedef polynomial  <time_t, num_t, true,
                     pointX_t, t_pointX_t> polynomial_t;
// Example with vector of dimension 3
typedef Eigen::Vector3d point_t;

// Creation of polynomial coefficients
point_t a(1,2,3);
point_t b(2,3,4);
point_t c(3,4,5);
polynomial_t::num_t T_min = 0.0;
polynomial_t::num_t T_max = 2.0;

// Polynomial order 1 : P(t) = a+b*t in [T_min,T_max]
std::vector<pointX_t> coefs;
coefs.push_back(a);
coefs.push_back(b);
polynomial_t pol1(coefs.begin(),coefs.end(),T_min,T_max);

// value of polynomial at time t=T_min and t=T_max
point_t res = pol1(T_min); // Equal to a
res = pol1(T_max);         // Equal to a+b
```

```
// Polynomial order 2 : P(t) = a+b*t+c*t^2 in [T_min, T_max]
coefs.push_back(c);
polynomial_t pol2(coefs.begin(),coefs.end(),T_min,T_max);

// value of polynomial at time t=T_min and t=T_max
res = pol2(T_min); // Equal to a
res = pol2(T_max); // Equal to a+b+c

// Derivative order one at time t=0.5 in [T_min, T_max]
res = pol2.derivate(0.5, 1);

// compute derivate
polynomial_t pol_der1 = pol2.compute_derivate(1);
res = pol_der1(0.5) // Equal to pol2.derivate(0.5, 1)

// Upper and Lower bound of definition interval
polynomial_t::num_t T_min_check = pol2.min(); // Equal to T_min
polynomial_t::num_t T_max_check = pol2.max(); // Equal to T_max

// get coefficients
Eigen::MatrixXd coeffs = pol2.coeff();
polynomial_t::point_t coeff_order0 = pol2.coeffAtDegree(0);

// Creation of polynomial from discrete points
point_t p0(0.,1.,0.);
point_t p1(1.,2.,-3.);
point_t dp0(-8.,4.,6.);
point_t ddp0(-1.,7.,4.);
point_t dp1(-9.,5.,7.);
point_t ddp1(0.,8.,3.);
// Constraint on initial and final pos
polynomial_t pol_a = polynomial_t(p0,p1,
                                  T_min,T_max);
// Constraint on initial and final pos/vel
polynomial_t pol_b = polynomial_t(p0,dp0,
                                  p1,dp1,
                                  T_min,T_max);
// Constraint on initial and final :pos/vel/acc
polynomial_t pol_c = polynomial_t(p0,dp0,ddp0,
                                  p1,dp1,ddp1,
                                  T_min,T_max);
```

1.4.3 Polynomial in python

We can use polynomial.h through curves.so as followed :

```
from curves import polynomial
```

```
# Creation of polynomial order 2
P0 = [1.,2.,3.]
P1 = [4.,5.,6.]
P2 = [7.,8.,9.]
T_min = 1.0
T_max = 3.0
coeffs = matrix([P0, P1, P2]).transpose()
pol = polynomial(coeffs) # defined in [0.,1.]
pol_time = polynomial(coeffs,T_min,T_max) # defined in [1.,3.]
```

```
# Value at time t=0.5
res = pol2(0.5)
# Derivative order 1 at time t=0.5
res = pol.derivate(0.5, 1)
# Compute derivate order 1
pol_der1 = pol2.compute_derivate(1)
res = pol_der1(0.5) # Equal to pol.derivate(0.5, 1)

# Upper and Lower bound of definition interval
T_min_check = pol_time.min() # Equal to T_min
T_max_check = pol_time.max() # Equal to T_max

# Get coefficients
res_coeff = pol.coeff()
res_coeff_order0 = pol.coeffAtDegree(0)

# Create polynomial from discrete points
p0 = matrix([1.,3.,-2.]).T
p1 = matrix([0.6,2.,2.5]).T
dp0 = matrix([-6.,2.,-1.]).T
dp1 = matrix([10.,10.,10.]).T
ddp0 = matrix([1.,-7.,4.5]).T
ddp1 = matrix([6.,-1.,-4]).T
# Polynomial C0
pol_C0 = polynomial(p0,p1,min,max)
# Polynomial C1
polC1 = polynomial(p0,dp0,p1,dp1,min,max)
# Polynomial C2
polC2 = polynomial(p0,dp0,ddp0,p1,dp1,ddp1,min,max)
```

1.5 Cubic Hermite spline

A cubic hermite spline is a polynomial curve of degree 3, defined by its initial and final positions and velocities : P_i, P_{i+1} and m_i, m_{i+1}.

1.5.1 Definition

A hermite cubic spline has some of the following properties :

- cross each of the waypoints given in its initialization (P_0, P_1,...,P_N).
- its derivatives $p'(t_{P_i}) = m_i$ and $p'(t_{P_{i+1}}) = m_{i+1}$.

A cubic hermite spline is defined by the polynom :

$$p(t) = h_{00}(t) \cdot P_0 + h_{10}(t) \cdot m_0 + h_{01}(t) \cdot P_1 + h_{11}(t) \cdot m_1 \tag{7}$$

where h_{ij} are Hermite basis functions :

- $h_{00} = 2 \cdot t^3 - 3 \cdot t^2 + 1.0$
- $h_{10} = t^3 - 2 \cdot t^2 + t$

- $h_{01} = -2 \cdot t^3 + 3 \cdot t^2$
- $h_{11} = t^3 - t^2$

If you want to create a cubic Hermite spline containing more than two waypoints, we recommend you instead to create a piecewise cubic Hermite curve (See section piecewise curve).

1.5.2 Template and constructors

Template for this class is composed of (See applications of cubic_hermite_spline):
- typename Time=double
- typename Numeric=Time
- bool Safe=false
- typename Point=Eigen::Matrix<Numeric, Eigen::Dynamic, 1 >

The class cubic_hermite_spline has the following constructors :

```
cubic_hermite_spline() // Default constructor

cubic_hermite_spline(In PairsBegin, In PairsEnd,
                     const Vector_time& time_control_points)

cubic_hermite_spline(const cubic_hermite_spline& other)
```

Where parameters of constructors are :

- ***PairsBegin*** : iterator pointing to the first element of a pair(position, derivative) container.

- ***PairsEnd*** : iterator pointing to the last element of a pair(position, derivative) container.

time_control_points : container with time corresponding to each waypoint. To evaluate spline at time t, we find the interval $[T_i, T_{i+1}]$ where t belong to, and we evaluate the segment corresponding using (P_i, m_i) and (P_{i+1}, m_{i+1}).

In our implementation, a cubic Hermite spline can be instantiated with several control points but it is deprecated, because we create in other words a piecewise cubic hermite. The class piecewise_curve exists and should be used instead if we have several control points. In this case, you have create all cubic hermite with 2 pairs only and add them all to the piecewise_curve (See subsection for piecewise_curve).

1.5.3 Cubic Hermite in C++

Here is an example of application using the class cubic_hermite_spline.h :

```
#include "curves/cubic_hermite_spline.h"
using namespace curves;
```

```
// Template
typedef Eigen::VectorXd pointX_t;
typedef double time_t;
typdef double num_t;
typedef std::pair<pointX_t, pointX_t> pair_point_tangent_t;
typedef cubic_hermite_spline <time_t, num_t, true, pointX_t
                              > cubic_hermite_spline_t;
// Example with vector of dimension 3
typedef Eigen::Vector3d point_t;

// Creation of pairs(position, derivative) and times
point_t P0(1,2,3);
point_t P1(4,5,6);
point_t m0(1,2,3);
point_t m1(4,5,6);
pair_point_tangent_t pair0(P0,m0);
pair_point_tangent_t pair1(P1,m1);
time_t T0 = 0.0;
time_t T1 = 2.0;

// Creation of containers
std::vector< pair_point_tangent_t > cp; // Control points
cp.push(pair0);
cp.push(pair1);
cubic_hermite_spline_t::vector_time_t time_control_points;
time_cp.push_back(T0);
time_cp.push_back(T1);

// Creation of cubic hermite spline
cubic_hermite_spline_t chs(cp.begin(), cp.end(), time_cp);

// Value of curve at time T0 and T1
Eigen::Vector3d res = chs(T0); // Equal to P0
res = chs(T1); // Equal to P1

// Value of derivative order 1 at time T0 and T1
res = chs.derivate(T0, 1); // Equal to m0
res = chs.derivate(T1, 1); // Equal to m1

// Upper and Lower bound of definition interval
cubic_hermite_spline_t::num_t Tmin = chs.min(); // Equal to T0
cubic_hermite_spline_t::num_t Tmax = chs.max(); // Equal to T1

// setTime and getTime
cubic_hermite_spline_t::vector_time_t time_cp2;
time_control_points2.push_back(0.);
time_control_points2.push_back(0.5);
cubic_hermite_spline_t chs2(cp.begin(), cp.end(), time_cp2);
res = chs(0.);  # Equal to P0
res = chs(0.5); # Equal to P1

// Get control points and Time
std::vector<pair_point_tangent_t> r_cp = chs.getControlPoints();
std::vector<time_t> r_time = chs.getTime();
```

```
// If we use several control points (Deprecated)
// Get number of Intervals (number of splines)
std::size_t nb_splines = chs.numIntervals(); // Here equal to 1
// Get number of pairs (control points)
std::size_t nb_cp = chs.size() // Here equal to 2
```

1.5.4 Cubic Hermite in Python

We can use cubic_hermite_spline.h through curves.so as followed :

```
from curves import cubic_hermite_spline
```

```
# Create containers for positions, derivatives and times
P0 = [1., 2., 3., 4.]
P1 = [4., 5., 6., 7.]
m0 = [1., 1., 3., 3.]
m1 = [4., 4., 6., 6.]
T0 = 0.0;
T1 = 1.0;
# In python constructors take 2 different containers
# for position and velocities
points = matrix([P0, P1]).transpose()
tangents = matrix([m0, m1]).transpose()
time_points = matrix([0., 1.]).transpose()

# Creation of cubic hermite spline
chs = cubic_hermite_spline(points, tangents, time_points)

# Value of curve at time T0 and T1
res = chs(T0) # Equal to P0
res = chs(T1) # Equal to P1

// Value of derivative order 1 at time T0 and T1
res = chs.derivate(T0, 1) # Equal to m0
res = chs.derivate(T1, 1) # Equal to m1

# Upper and Lower bound of definition interval
Tmin = chs.min() # Equal to T0
Tmax = chs.max() # Equal to T2
```

1.6 piecewise curve

A piecewise curve Pc is made of one or several curves $Cf_0, Cf_1, ...Cf_N$, where :

- Pc defined in $[Tmin, Tmax]$
- $\forall i \subset \{0, 1, ..., N\}$, Cf_i is defined in $[Tmin_i, Tmax_i]$
- $\forall i \subset \{0, 1, ..., N\}$, $Tmax_i = Tmin_{i+1}$
- $Tmin_0 = Tmax$ and $Tmax_N = Tmax$

To evaluate the piecewise curve at time $t \in [Tmin, Tmax]$, we search the curve whose interval of definiton contains t, and we evaluate the curve found at time t.

1.6.1 Template and constructors

Template for this class is composed of (See applications of cubic_hermite_spline):
- typename Time=double
- typename Numeric=Time
- bool Safe=false
- typename Point=Eigen::Matrix<Numeric, Eigen::Dynamic, 1 >
- typename T_Point= std::vector<Point, Eigen::aligned_allocator<Point> >
- typename Curve= curve_abc<Time, Numeric, Safe, Point>

The class piecewise_curve has the following constructors :

```
piecewise_curve() // Default constructor : Need to add a curve

piecewise_curve(const curve_t& cf)

piecewise_curve(const t_curve_t list_curves)

piecewise_curve(const piecewise_curve& other)
```

Where parameters of constructors are :

- ***cf*** : A curve of the same type than the one in template. The piecewise curve created will contain this curve.
- ***list_curves*** : A list of curves of the same type than the one in template. The piecewise curve created will contain these curves. Curves in list needs to respect the conditions (See subsection piecewise_curve).
- ***other*** : The piecewise curve to copy. The type of curve it contains must be of the same type than the one in template.

We also can get a piecewise polynomial from a set of discrete points with templated functions (See how to use them in section Piecewise curve in C++) :

```
template<typename Polynomial>
static piecewise_curve<Time, Numeric, Safe, Point, T_Point,
                       Polynomial>
convert_discrete_points_to_polynomial(T_Point points,
                                      t_time_t time_points)
```

```
template<typename Polynomial>
static piecewise_curve<Time, Numeric, Safe, Point, T_Point,
                       Polynomial>
convert_discrete_points_to_polynomial(T_Point points,
                                      t_time_t time_points)
```

```
template<typename Polynomial>
static piecewise_curve<Time, Numeric, Safe, Point, T_Point,
                       Polynomial>
convert_discrete_points_to_polynomial(T_Point points,
                                      t_time_t time_points)
```

1.6.2 Piecewise curve in C++

Here is an example of application using the class piecewise_curve.h :

```
#include "curves/piecewise_curve.h"
```

```
// You can use one of these three curves with piecewise curve
#include "curves/polynomial.h"
#include "curves/bezier_curve.h"
#include "curves/cubic_hermite_spline.h"
using namespace curves;
```

```
typedef Eigen::VectorXd pointX_t;
typedef std::vector<pointX_t,
                    Eigen::aligned_allocator<pointX_t>
                   >  t_pointX_t;
typedef double time_t;
typdef double num_t;
// Example with vector of dimension 3
typedef Eigen::Vector3d point3_t;

// ***** Curves inheriting from curve_abc *****
// Polynomial
typedef polynomial  <time_t, num_t, true,
                     pointX_t, t_pointX_t> polynomial_t;
// Bezier
typedef bezier_curve  <double, double,
                       true, pointX_t
                      > bezier_curve_t;
// Cubic Hermite
typedef cubic_hermite_spline <double, double,
                              true, pointX_t
                             > cubic_hermite_spline_t;

// ***** Piecewise curves *****
// Piecewise polynomial curve
typedef piecewise_curve <double, double,
                         true, pointX_t,
                         t_pointX_t, polynomial_t
                        > piecewise_polynomial_curve_t;
// Piecewise bezier curve
typedef piecewise_curve <double, double,
                         true, pointX_t,
                         t_pointX_t, bezier_curve_t
                        > piecewise_bezier_curve_t;
// Piecewise cubic hermite curve
typedef piecewise_curve <double, double,
                         true, pointX_t,
                         t_pointX_t, cubic_hermite_spline_t
                        > piecewise_cubic_hermite_curve_t;

// ** Example with piecewise polynomial curve **
// Create polynomials
point3_t a(1,1,1);
point3_t b(2,1,1);
point3_t c(3,1,1);
t_pointX_t vec1, vec2, vec3;
vec1.push_back(a); // x_coef=1, y_coef=1, z_coef=1
```

```
vec2.push_back(b); // x_coef=2, y_coef=1, z_coef=1
vec3.push_back(c); // x_coef=3, y_coef=1, z_coef=1
time_t T0 = 0.0;
time_t T1 = 2.0;
time_t T2 = 3.0;
time_t T3 = 6.0;
polynomial_t pol1(vec1.begin(), vec1.end(), T0, T1);
polynomial_t pol2(vec2.begin(), vec2.end(), T1, T2);
polynomial_t pol3(vec3.begin(), vec3.end(), T2, T3);

// Create an empty piecewise polynomial curve
piecewise_polynomial_curve_t pc();

// Add some polynomial in pc
pc.add_curve(pol1); // pc defined on [T0,T1]
pc.add_curve(pol1); // pc defined on [T0,T2]
pc.add_curve(pol1); // pc defined on [T0,T3]

// Evaluation of piecewise polynomial curve
point3_t res;
// position
res = pc(T0); // Equal to pol1(T0)
res = pc(T1); // Equal to and pol2(T1)
res = pc(T2); // Equal to pol3(T2)
res = pc(T3); // Equal to pol3(T3)
// derivative order 1
res = pc.derivate(T0,1); // Equal to pol1.derivate(T0)
res = pc.derivate(T1,1); // Equal to pol2.derivate(T1)
res = pc.derivate(T2,1); // Equal to pol3.derivate(T2)
res = pc.derivate(T3,1); // Equal to pol3(T3)
// derivative order 2
res = pc.derivate(0.5,2); // Equal to pol1.derivate(0.5,2)

// Continuity at specified order of piecewise curve
bool isC0 = pc.is_continuous(0);
bool isC1 = pc.is_continuous(1);
bool isC2 = pc.is_continuous(2);

// Convert discrete points to piecewise polynomial curves
// We have :
// - t_pointX_t points : list of discrete points.
// - t_pointX_t points_derivative : list of velocity at these points.
// - t_pointX_t points_second_derivative : list of acceleration
//                                         at these points.
// - std::vector<time_t> time_points : list of time corresponding
//                                     to these points.
// Piecewise polynomial curve C0
piecewise_polynomial_curve_t ppc =
    piecewise_polynomial_curve_t::
    convert_discrete_points_to_polynomial<polynomial_t>
    (points,time_points);
// Piecewise polynomial curve C1
piecewise_polynomial_curve_t ppc_C1 =  piecewise_polynomial_curve_t::
    convert_discrete_points_to_polynomial<polynomial_t>
    (points,points_derivative,time_points);
// Piecewise polynomial curve C2
piecewise_polynomial_curve_t ppc_C2 =  piecewise_polynomial_curve_t::
```

```
        convert_discrete_points_to_polynomial<polynomial_t>
        (points, points_derivative, points_second_derivative, time_points);
```

1.6.3 Piecewise curve in Python

We can use piecewise_curve.h through curves.so as followed :

```
    # You can use the three kinds of piecewise curves :
    from curves import ( polynomial, bezier, cubic_hermite_spline,
                         piecewise_polynomial_curve,
                         piecewise_bezier_curve,
                         piecewise_cubic_hermite_curve )
```

```
    # Example with piecewise_bezier_curve
    # Create two bezier curves (with two control points)
    P0 = [1., 1., 1.]
    P1 = [2., 2., 2.]
    P2 = [3., 3., 3.]
    P3 = [4., 4., 4.]
    waypoints0 = matrix([P0, P1]).transpose()
    waypoints1 = matrix([P2, P3]).transpose()
    bc0 = bezier(waypoints0, 0., 1.)
    bc1 = bezier(waypoints1, 1., 3.)

    # Create an empty piecewise bezier curve
    pc = piecewise_bezier_curve()

    # Add two bezier curves
    pc.add_curve(bc0)
    pc.add_curve(bc1)

    # Evaluation
    res = pc(0.) # Equal to P0
    res = pc(3.) # Equal to P3

    # Continuity at specified order of piecewise curve
    isC0 = pc.is_continuous(0)
    isC1 = pc.is_continuous(1)
```

1.7 Exact cubic spline

Spline defined as in the paper "Task-Space Trajectories via Cubic Spline Optimization" by J. Zico Kolter and Andrew Y.ng (ICRA 2009).
Given a set of waypoints x_i and timestep t_i, it provides the unique set of cubic splines fulfilling those four restrictions :

- $x_i(t_i) = x_i$, meaning that the curve passes trough each waypoint;
- $x_i(t_{i+1}) = x_{i+1}$;
- its derivative is continous at t_{i+1};
- its acceleration is continous at t_{i+1}.

Exact cubic class inherits from piecewise_curve and is in fact, a piecewise polynomial curve with all functions of the paper implemented. With N waypoints, we will have $N-1$ polynomial curves :

$$x(t) = \begin{cases} x_0(t) & \text{if } t_0 \leq t < t_1 \\ x_1(t) & \text{if } t_1 \leq t < t_2 \\ ... & \\ x_{N-1}(t) & \text{if } t_{N-2} \leq t < t_{N-1} \end{cases} \tag{8}$$

1.7.1 Template and constructors

The class exact_cubic.h has four constructor with templated types :

```
exact_cubic() // Default constructor :
              // this is a simple piecewise polynomial curve

exact_cubic(In wayPointsBegin, In wayPointsEnd)

exact_cubic(In wayPointsBegin, In wayPointsEnd,
            const spline_constraints& constraints)

exact_cubic(const t_spline_t& subSplines)

exact_cubic(const exact\_cubic\& other)
```

The template for this class is composed of (See applications of exact_cubic.h) :
typename Time=double
typename Numeric=Time
bool Safe=false
typename Point=Eigen::Matrix<Numeric, Eigen::Dynamic, 1 >
typename T_Point=std::vector<Point,Eigen::aligned_allocator<Point>>
typename SplineBase=polynomial<Time, Numeric, Dim, Safe, Point, T_Point>

Parameters of constructors are :

wayPointsBegin : an iterator pointing to the first element of a waypoint container. A waypoint is represented as a std::pair< t_i, P_i > with t_i the time where $x(t_i) = P_i$.
wayPointsEnd :an iterator pointing to the last element of a waypoint container.
spline_constraints : constraints applied on start and end point of the curve. Structure curve_constraint contains 4 instance variables to set : *init_vel*, *init_acc*, *end_vel*, *end_acc* which correspond to initial and final velocities and accelerations. For more details, read paper on how to apply it, read paper "Task-Space Trajectories via Cubic Spline Optimization" by J. Zico Kolter and Andrew Y.ng (ICRA 2009).
other : The exact cubic spline to copy.

1.7.2 Exact cubic in C++

Here is an example of application using the class exact_cubic.h :

```
#include "curves/exact_cubic.h"
using namespace curves;
```

```
// Template
typedef Eigen::VectorXd pointX_t;
typedef std::vector<pointX_t,
                    Eigen::aligned_allocator<pointX_t>
                   >  t_pointX_t;
typedef double time_t;
typdef double num_t;
typedef exact_cubic <double, double,
                     true, pointX_t
                    > exact_cubic_t;
// Example with vector of dimension 3
typedef Eigen::Vector3d point3_t;

// Creation of waypoint container
typedef std::pair<double, pointX_t> Waypoint;
typedef std::vector<Waypoint> T_Waypoint;
T_Waypoint waypoints;
point3_t P0(1,2,3);
point3_t P1(4,5,6);
point3_t P2(7,8,9);
double T0 = 0.0;
double T1 = 1.0;
double T2 = 2.0;
std::pair pair_P0 = std::make_pair(T0, P0);
std::pair pair_P1 = std::make_pair(T1, P1);
std::pair pair_P2 = std::make_pair(T2, P2);
waypoints.push_back(pair_P0);
waypoints.push_back(pair_P1)
waypoints.push_back(pair_P2)

// Creation of exact cubic
exact_cubic_t ec(waypoints.begin(), waypoints.end());

// Get number of splines (Polynomials)
std::size_t numberSegments = ec.getNumberSplines();

// Get spline at index
exact_cubic_t::spline_t first_spline = ec.getSplineAt(0);

// Value of curve at time t=T0, t=T1 and t=T2
Eigen::Vector3d res = ec(T0) // Equal to P0
res = ec(T1)                 // Equal to P1
res = ec(T2)                 // Equal to P2

// Value of derivative order 1 at time t=0.5
res = ec.derivate(0.5, 1)
```

```
// Upper and Lower bound of definition interval
double Tmin = ec.min() // Equal to T0
double Tmax = ec.max() // Equal to T2

// Creation of exact cubic with constraints
exact_cubic_t::spline_constraints constraints;
constraints.end_vel = point3_t(1,2,3);
constraints.init_vel = point3_t(-1,-2,-3);
constraints.end_acc = point3_t(4,5,6);
constraints.init_acc = point3_t(-4,-4,-6);
exact_cubic_t ec2(waypoints.begin(), waypoints.end(),constraints);

// Check derivatives
res = ec2.derivate(T0,1); // Equal to init_vel
res = ec2.derivate(T0,2); // Equal to init_acc
res = ec2.derivate(T2,1); // Equal to end_vel
res = ec2.derivate(T2,2); // Equal to end_acc
```

1.7.3 Exact cubic in Python

We can use exact_cubic.h through curves.so as followed :

```
from curves import exact_cubic, curve_constraints
# Creation of exact cubic
P0 = [1.,2.,3.]
P1 = [4.,5.,6.]
P2 = [7.,8.,9.]
T0 = 0.0;
T1 = 1.0;
T2 = 2.0;
waypoints = matrix([P0, P1, P2]).transpose()
time_waypoints = matrix([T0, T1, T2]).transpose()
ec = exact_cubic(waypoints, time_waypoints)
NumberOfSplines = ec.getNumberSplines() # Get number of splines
FirstSpline = ec.getSplineAt(0) # Get first spline (polynomial)

# Evaluation at t=0.5
res = ec(0.5)
# Derivative order 1 at t=0.5
res = ec.derivate(0.5, 1)

# Upper and lower bound of definition interval
T_min_check = ec.min()
T_min_check = ec.max()

# Creation of exact cubic with constraints
c = curve_constraints()
c.init_vel = matrix([0., 1., 1.]).transpose()
c.end_vel = matrix([0., 2., 2.]).transpose()
c.init_acc = matrix([0., 3., 3.]).transpose()
c.end_acc = matrix([0., 4., 4.]).transpose()
ec2 = exact_cubic(waypoints, time_waypoints, c)

# Derivative at time t
res = ec2.derivate(T0,1) # Equal to init_vel
res = ec2.derivate(T0,2) # Equal to init_acc
res = ec2.derivate(T2,1) # Equal to end_vel
```

```
res = ec2.derivate(T2,2) # Equal to end_acc
```

2 Conversion of cubic curves

In our actual implementation, we can convert our curves between bezier, cubic hermite and polynomial curves of degree 3 (cubic curves) or less. We can also convert piecewise curves containing these types of curves if they are all of degree 3 or less.

2.1 Basic cubic curves conversion

The header curve_conversion.h contains all templated functions to convert all curves of degree 3 or less to a bezier, cubic hermite or polynomial curve :

```
template<typename Polynomial, typename curveTypeToConvert>
Polynomial polynomial_from_curve(const curveTypeToConvert& curve)
```

```
template<typename Bezier, typename curveTypeToConvert>
Bezier bezier_from_curve(const curveTypeToConvert& curve)
```

```
template<typename Hermite, typename curveTypeToConvert>
Hermite hermite_from_curve(const curveTypeToConvert& curve)
```

2.1.1 Conversion of basic cubic curves in C++

```
#include "curves/curve_conversion.h"
```

```
// Example with all 3 basic curves
#include "curves/polynomial.h"
#include "curves/bezier_curve.h"
#include "curves/cubic_hermite_spline.h"
using namespace curves;
```

```
// We take the same typedef than in previous
// examples in C++ for all curves.

// We have 3 curves :
// polynomial_t pol  => Polynomial with Degree 3 or less.
// bezier_curve_t bc => Bezier with 4 or less control points.
// cubic_hermite_spline_t chs => Cubic Hermite spline

// Convert : Polynomial=>Bezier
bezier_curve_t bc_res = bezier_from_curve<bezier_curve_t,
                                          polynomial_t
                                         >(pol);
// Convert : Bezier=>Polynomial
polynomial_t pol_res = polynomial_from_curve<polynomial_t,
                                             bezier_curve_t
                                            >(bc);
// Convert : Bezier=>Cubic Hermite
cubic_hermite_spline_t chs_res = hermite_from_curve
```

```
                                    <cubic_hermite_spline_t,
                                     bezier_curve_t
                                    >(bc)

    // For all t in interval of definition of :
    // - bc_res : bc_res(t) = pol(t)
    // - pol_res : pol_res(t) = bc(t)
    // - chs_res : chs_res(t) = bc(t)
```

2.1.2 Conversion of basic cubic curves in Python

```
    # Select the type of conversion needed
    from curves import (bezier_from_hermite,
                        bezier_from_polynomial,
                        hermite_from_polynomial,
                        hermite_from_bezier,
                        polynomial_from_hermite,
                        polynomial_from_bezier)
```

```
    # Example with a bezier : bc
    pol_res = polynomial_from_bezier(bc)
    chs_res = hermite_from_polynomial(pol_res)
    bc_res = bezier_from_hermite(chs_res) # Same as bc
```

2.2 Piecewise cubic curves conversion

If we need to convert a piecewise curve containing one type of curve of degree 3 or less to another one, three functions are available in piecewise curve that :
- create a new empty piecewise curve which will contain the type wanted;
- convert all curves in the actual one to the type wanted; - add them all to the new piecewise curve returned.

```
    template<typename Bezier>
    piecewise_curve<Time, Numeric, Safe,
                    Point, T_Point, Bezier
                   > convert_piecewise_curve_to_bezier()
```

```
    template<typename Polynomial>
    piecewise_curve<Time, Numeric, Safe,
                    Point, T_Point, Polynomial
                   > convert_piecewise_curve_to_polynomial()
```

```
    template<typename Hermite>
    piecewise_curve<Time, Numeric, Safe,
                    Point, T_Point, Hermite
                   > convert_piecewise_curve_to_cubic_hermite()
```

2.2.1 Conversion of piecewise cubic curves in C++

```
    // We take the same typedef than in previous
    // examples in C++ for all curves.

    // Example : we have a piecewise_polynomial_curve_t : ppc
```

```
// Piecewise Polynomial => Piecewise Bezier
piecewise_bezier_curve_t pc_bezier =
    ppc.convert_piecewise_curve_to_bezier
       <bezier_curve_t>();

// Piecewise Polynomial => Piecewise Cubic Hermite
piecewise_bezier_curve_t pc_bezier =
    ppc.convert_piecewise_curve_to_cubic_hermite
       <cubic_hermite_spline_t>();

// Piecewise Bezier => Piecewise Polynomial
piecewise_polynomial_curve_t pc_pol =
    pc_bezier.convert_piecewise_curve_to_polynomial
       <polynomial_t>();
```

2.2.2 Conversion of piecewise cubic curves in Python

```
from curves import ( piecewise_polynomial_curve,
                     piecewise_bezier_curve,
                     piecewise_cubic_hermite_curve )
```

```
# Piecewise curve with polynomials of degree 3 or less : pc

# Convert to piecewise cubic hermite curve
pc_chs = pc.convert_piecewise_curve_to_cubic_hermite()

# Convert to piecewise bezier curve
pc_bc = pc.convert_piecewise_curve_to_bezier()

# Convert to piecewise polynomial curve
pc_pol = pc_bc.convert_piecewise_curve_to_polynomial()
```

3 Serialization

For serializing curves, we use boost::serialization which allows us to serialize entire classes. Eigen matrices have been serialized in file include/curves/serialization/eigen-matrix.hpp. All curves and piecewise curves can be serialized simply by overriding the function of curve_abc :

```
void serialize(Archive& ar, const unsigned int version)
```

Example in polynomial.h :

```
public:
// Serialization of the class
friend class boost::serialization::access;

template<class Archive>
void serialize(Archive& ar, const unsigned int version){
    if (version) {
      // Do something depending on version ?
    }
    ar & boost::serialization::make_nvp("dim", dim_);
    ar & boost::serialization::make_nvp("coefficients", coefficients_);
    ar & boost::serialization::make_nvp("dim", dim_);
    ar & boost::serialization::make_nvp("degree", degree_);
    ar & boost::serialization::make_nvp("T_min", T_min_);
```

```
        ar & boost::serialization::make_nvp("T_max", T_max_);
    }
```

Functions to call to serialize an object are defined in include/curves/serialization/archive.hpp :

```
    // Save/Load from Text file
    template<class Derived>
    void saveAsText(const std::string & filename)
    template<class Derived>
    void loadFromText(const std::string & filename)

    // Save/Load from XML file
    template<class Derived>
    void saveAsXML(const std::string & filename)
    template<class Derived>
    void loadFromXML(const std::string & filename)

    // Save/Load from Binary file
    template<class Derived>
    void saveAsBinary(const std::string & filename)
    template<class Derived>
    void loadFromBinary(const std::string & filename)
```

3.0.1 Serialization in C++

```
    // We take the same typedef than in previous
    // examples in C++ for all curves.

    // Example with a polynomial_t : pol
    // Serialize in file : pathToFile0
    pol.saveAsText<polynomial_t>(pathToFile0);
    polynomial_t pol_res;
    pol_res.loadFromText<polynomial_t>(pathToFile0); // Equal to pol

    // Example with piecewise_bezier_curve_t : pc
    // Serialize in file : pathToFile1
    pc.saveAsBinary<piecewise_bezier_curve_t>(pathToFile1)
    piecewise_bezier_curve_t pc_res;
    pc_res.loadFromBinary<piecewise_bezier_curve_t>(pathToFile1)
```

3.0.2 Serialization in Python

```
    # We take the same import than in previous
    # examples in Python for all curves

    # Example with bezier curve : bc
    # Serialize in file : pathToFile0
    bc.saveAsText(pathToFile0)
    bc_res = bezier()
    bc_res.loadFromText(pathToFile0)

    # Example with piecewise_polynomial_curve : pc
    # Serialize in file : pathToFile1
    pc.saveAsText(pathToFile1)
    pc_res = piecewise_polynomial_curve()
    pc_res.loadFromText(pathToFile1)
```