An introduction to NURBS^{++}

Philippe Lavoie

April 28, 1999

NURBS^{++}is available from the web at http://yukon.genie.uottawa.ca/ lavoie/software/nurbs. It offers classes to represent NURBS curves and surfaces along with many functions to help manipulate them. For interactive design purposes it also has OpenGL wrappers.

This document focuses on how to use the library to perform elementary tasks. It doesn't not discuss the theoretical aspects of NURBS curves and surfaces. From the link above you will be able to find documents which explain more the mathematical and theoretical aspects of NURBS curves and surfaces.

This document is *not* a replacement for the reference manual. Most of the function are shown here with a brief introduction but for a more complete description of the input and output values you should refer to the reference manual. The reference manual can be found from the link mentionned above.

I will also admit that I spend more time on the source code than on the documentation. Therefore for the truly latest documentation, nothing beats the source code. However, I will incorporate into this document any changes proposed by a reader.

1 The basic types

NURBS curves and surfaces are represented using knot vectors and control points. Since the library uses templates, it is up to the user to define if he/she prefers to perform the computation using float or double float precision. The dimension domain can also be changed, i.e. 2D or 3D curves. A the time of writing, no efforts were made to obtain a 2D surface. To specify the precision and the dimension, change the mentions of `<T,D>` that follows in the text with `<float,3>`, with `<double,3>`, with `<float,2>` or with `<double,2>`.

A control point is of type `HPoint_nD<T,D>` which stands for Homogenous point. A knot vector is of type `Vector<T>`. By extension, a vector of control points is of type `Vector<HPoint_nD<T,D> >`. In the case of a surface, a matrix of Control points is used (`Matrix<HPoint_nD<T,D> >`). A NURBS curve or surface can be evaluated in normal space and the result is returns is a `Point_nD<T,D>` Finally, a NURBS curve is referred to as `NurbsCurve<T,D>` and a NURBS surface as `NurbsSurface<T,D>`.

Every classes defined by NURBS^{++}are declared inside the `PLib` namespace. To save you some typing, you can either add a `using namespace PLib;` at the begining of your files or alternatively you can use one of the typedefs provided by the library. These typedefs are the following:

2 NURBS curve

There are different methods to initialize a NURBS curve: from a list of points in 3D or 4D, by copying, or by leaving it empty. If you leave a NURBS curve empty you must realize that it has *not* been initialized properly and therefore shouldn't be used to compute anything useful (any such attempt will result in an error message).

It should be noted that at the initialization stage, the value provided is the value given to the control points. It doesn't mean that the curve will go through these points. If you want the later, you will need to use a routine that interpolates between those points.

Here is an example on how to initialize a NURBS curve with known control points values and a known knot vector.

PLib::Point_nD<float,3>	Point3Df
PLib::Point_nD<double,3>	Point3Dd
PLib::HPoint_nD<float,3>	HPoint3Df
PLib::HPoint_nD<double,3>	HPoint3Dd
PLib::NurbsCurve<float,3>	NurbsCurvef
PLib::NurbsCurve<double,3>	NurbsCurved
PLib::NurbsCurve<float,2>	NurbsCurve_2Df
PLib::NurbsCurve<double,2>	NurbsCurve_2Dd
PLib::NurbsSurface<float,3>	NurbsSurfacef
PLib::NurbsSurface<double,3>	NurbsSurfaced
PLib::Matrix<float>	Matrix_float
PLib::Matrix<double>	Matrix_double
PLib::Vector<float>	Vector_float
PLib::Vector<double>	Vector_double
PLib::Matrix<HPoint3Df>	Matrix_HPoint3Df
PLib::Matrix<HPoint3Dd>	Matrix_HPoint3Dd
PLib::Matrix<Point3Df>	Matrix_Point3Df
PLib::Matrix<Point3Dd>	Matrix_Point3Dd

Table 1: Correspondence between basic types and there typedefs

```
Vector_HPoint3Df pts(4) ;

pts[0] = HPoint3Df(0,0,0,1) ;
pts[1] = HPoint3Df(30,0,0,1) ;
pts[2] = HPoint3Df(60,30,0,1) ;
pts[3] = HPoint3Df(90,30,0,1) ;

Vector_float knot(8) ;
knot[0] = knot[1] = knot[2] = knot[3] = 0 ;
knot[4] = knot[5] = knot[6] = knot[7] = 1 ;

NurbsCurvef curve(pts,knot,3) ;
```

The above creates a NURBS curve with the control points set at the points shown above. The figure 1 shows what the curve looks like.

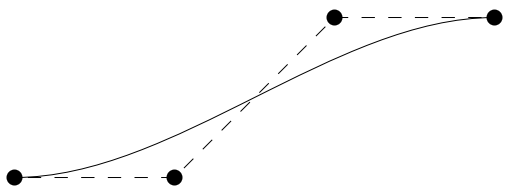


Figure 1: The curve generated by specifying the control points.

2.1 Display and input/output

The library offers different methods to view, save and read a NURBS curve. To display a curve you can generate a Post-Script or a VRML file. It should be noted that Post-Script can not handle true NURBS curves, i.e. if there is a weight associated with a control point different than 1.0. Then the curve in Post-Script will *not* be the same as the one you provided. This is because Post-Script only uses B-Splines.

```
bool cp = true ;
float magf = 1 ;
int dash = 2 ;
curve.writePS("curve1.ps",cp,magf,dash) ;
curve.writePSp("curve2.ps",pts,empty,cp,magf,dash) ;
curve.writeVRML("curve.wrl",radius,K,Color(255,0,0),nu,nv) ;
```

The `cp` value indicates if we want to print the control points. The `magf` value indicates the magnification of the image. If that value is smaller or equal to 0, the library tries to guess an appropriate value such that the curve will fill the page[1]. The `dash` value indicates the size of the dash of the lines joining each control points (if cp is set to true). A value of 0 indicates a plain line.

In the second method to call writePSp, the `pts` are a list of points we want to print with the NURBS curve. the `empty` is a list of normal vectors that are possibly drawn from a point in `pts`. In this case `empty` denotes that there is no such value to print.

Finally when writing a VRML file, the library is actually writing a NURBS surface. It sweeps a circle along the curve to represent the curve in 3D. The radius value specify which value to use as a radius for that circle. The call to the Color constructor specifies the color of the resulting curve. And the last two values (nu,nv) specify the number of points in the U and V parametric space.

2.2 Generating certain types of curves

NURBS^{++}generates two types of standard curves automatically: a circle or a line.

You can creates a circle centered at $(0,0,0)$ of radius 1 and having a starting and ending angle of 0 and 2π respectively.Since a NURBS curve is rational, it can represent *exactly* a circle. Something that a B-Spline can't do.

```
NurbsCurvef curve ;
curve.makeCircle(Point3Df(0,0,0), 1.0f, 0, 2*M_PI) ;
```

To create a line going from $(0,0,0)$ to $(1,1,1)$ with a degree 3, do the following:

```
NurbsCurvef curve ;
curve.makeLine(Point3Df(0,0,0), Point3Df(1,1,1), 3) ;
```

2.3 Point fitting

A good method to generate a NURBS curve is the interpolation of points. Using this technique, the NURBS curve will go *through* the points specified. However, sometimes there are too many points and you only want a NURBS curve which looks like it goes through them but within a certain error bound. For that case you can use the global approximation method. Finally there is a last case where the points are noisy and you want a least square fit of the data.

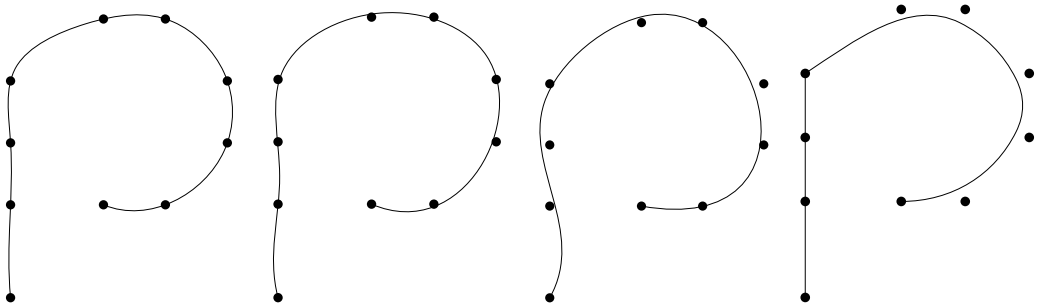
The library can handle all the above cases with the member functions: globalInterp, globalAprroxErrBnd, leastSquare. Example of there use is shown below.

```
Vector_Point3Df pointsB(10) ;

pointsB[0] = Point3Df(20,20,0) ;
pointsB[1] = Point3Df(20,80,0) ;
pointsB[2] = Point3Df(20,120,0) ;
pointsB[3] = Point3Df(20,160,0) ;
pointsB[4] = Point3Df(80,200,0) ;
pointsB[5] = Point3Df(120,200,0) ;
pointsB[6] = Point3Df(160,160,0) ;
```

[1]This feature is not fully operational yet, i.e. don't use it.

```
pointsB[7] = Point3Df(160,120,0) ;
pointsB[8] = Point3Df(120,80,0) ;
pointsB[9] = Point3Df(80,80,0) ;

NurbsCurvef curveA,curveB,curveC,curveD ;

curveA.globalInterp(pointsB,3) ;
curveB.leastSquares(pointsB,3,pointsB.n()-2) ;
curveC.leastSquares(pointsB,3,pointsB.n()-4) ;
curveD.globalApproxErrBnd(pointsB,3,10) ;
```

With the above code, curveA interpolates through all the points, curveB is a leastSquares approximation with 2 control points less than curveA, curveC has 4 control points less than curveB and finally curveD is an approximation of the points with an accepted error of 10. The difference between all of these methods is best shown in figure 2.



(a) Global Interpolation.

(b) Least Square fitting with 8 control points.

(c) Least Square fitting with 5 control points.

(d) Global Approximation with an accepted error of 10.

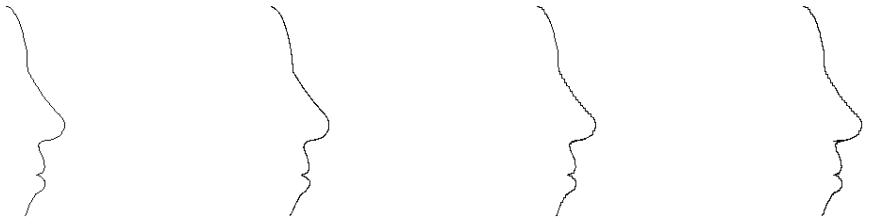
Figure 2: Point fitting functions.

The points in figure 2 are actually hard to fit, in the sense that there is a straight line followed by points in a square like pattern. Reducing the number of points in the least squares method emphasizes the fact that there is a wobble in the straight line, but even the interpolation method can't get rid of this wobble. In contrast, the approximation method captures exactly the points on the line, but it only goes near the other points (within the error of 10 specified).

I think it is important to specify at this point that the approximation method is mostly useful when there is a lot of points close to each others. In those cases, global interpolation is almost useless and least squares fitting can be used but it will most likely smooth the curve you want to fit and add wobble. To emphasize this, you should look at the figure 3. That figure clearly shows that in the case of interpolation and least square fit there is a wobble. Something that isn't present in the approximation case.

When doing global interpolation you can also specify the derivative of the curve at the points to fit. Sometimes this is necessary to ensure that the curve will keep a certain geometry. For example, the code below constructs two curves, one without the derivatives and one with them specified.

```
Vector_Point3Df ptsA(4),ptsAD(4) ;

ptsA[0] = Point3Df(0,0,0) ;
ptsA[1] = Point3Df(30,0,0) ;
ptsA[2] = Point3Df(60,30,0) ;
ptsA[3] = Point3Df(90,30,0) ;
```



(a) Original image. Each black pixel is a point to fit.

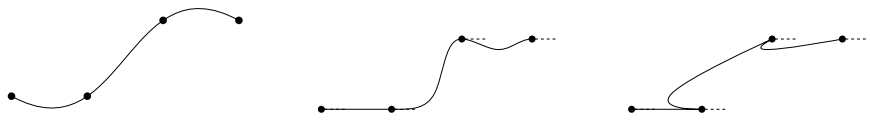
(b) Global approximation with 1.5 error

(c) Global interpolation

(d) Least square fit with 80% of the control points.

Figure 3: Difference between approximation and interpolation when a lot of points are close to each others.

```
ptsAD[0] = Point3Df(1,0,0) ;
ptsAD[1] = Point3Df(1,0,0) ;
ptsAD[2] = Point3Df(1,0,0) ;
ptsAD[3] = Point3Df(1,0,0) ;

NurbsCurvef cA(pts,knot,3) ;

cA.globalInterp(ptsA,3) ;
cA.writePSp("interp.ps",ptsA,empty,0,2.0) ;

cA.globalInterpD(ptsA,ptsAD,3,1) ;
cA.writePSp("interpD.ps",ptsA,10.0*ptsAD,0,2.0) ;

cA.globalInterpD(ptsA,ptsAD,3,0) ;
cA.writePSp("interpDb.ps",ptsA,10.0*ptsAD,0,2.0) ;
```



(a) Global Interpolation

(b) Global Interpolation with unit dervative specified.

(c) Global Interpolation with dervative value specified.

Figure 4: Global interpolation and derivative specification.

If you run this program you will obtain the outputs shown in figure4. From that figure you can notice that changing the last parameter of globalInterpD dramatically changes the look of the curve. This parameters specifies if the derivatives given to the function are unit vectors or not. A unit vector gives the direction of the derivative but not its length.

2.4 Curve evaluation

Once you have an initialized curve you can use it to evaluate the curve at a certain parameter or evaluate the derivative of the curve at that parameter. To evaluate a curve in the 4D homogeneous space system or to evaluate it in the 3D space use one of the function below.

```
HPoint3Df pw ;
pw = curve(u) ;
pw = curve.point4D(u) ;
pw = C(u,curve) ;

Point3Df p ;
p = curve.point3D(u) ;
p = project(curve(u)) ;
p = Cp(u,curve) ;
```

When computing the derivative, the library actually stores all the derivatives in a vector: $[c^{(0)}|_u, c^{(1)}|_u, \ldots, c^{(n)}|_u]$ where $n \leq$ degree. The value of $c^{(0)}|_u$ is actually the same as $c(u)$. For your convenience there is also a call to the derivative function which hides this information. Therefore you can choose how much information you want when computing the derivative. The derivative can be computed in either the homogeneous space or the normal dimension. In the example, 4D is used to indicate the 3D homogenous space.

```
Vector_HPoint3Df dw ;
Vector_Point3Df d ;

curve.deriveAt(u,2,dw) ;// dw[0] == curve(u)
                        // dw[1] = first derivative
                        // dw[2] = second derivative
curve.deriveAt(u,2,d) ; // as above but in 3D

pw = curve.derive(u,1) ; // pw = first derivative in 4D
p = curve.derive3D(u,1) ; // p = first derivative in 3D
```

2.5 Curve modification

Different methods can be used to modify the curve: global transformation, local modification and curve splitting. The global transformation affects every control points. A local modification affects either a single control point or a point on the NURBS curve. In either cases, one or more control points are moved and the NURBS curve is changed locally around that point. Finally curve splitting creates two NURBS curves from a single one.

The global transformation requires you to get familiar with the `MatrixRT<T>` class. This class is used to describe a transformation matrix. The transformation matrix allows you to rotate, scale and translate points. These transformations can be combined to create complex transformations.

```
MatrixRT<float> Tx(rx,ry,rz,tx,ty,tz) ;
MatrixRT<float> Sx ;

Sx.scale(sx,sy,sz) ;

curve.transform(Tx*Sx) ;
```

The above creates a matrix `Tx` which translates a vector by (t_x, t_y, t_z) then rotates it around the z-axis, then the y-axis and finally around the x-axis. The `Sx` matrix scales the result by (s_x, s_y, s_z). The order of the transformation will be first a scaling and afterwards a transformation.

For local modification there are two simple calls and then more complex ones. The simple ones directly modify a control point to either change its value or to change its value by a certain amount. For example,

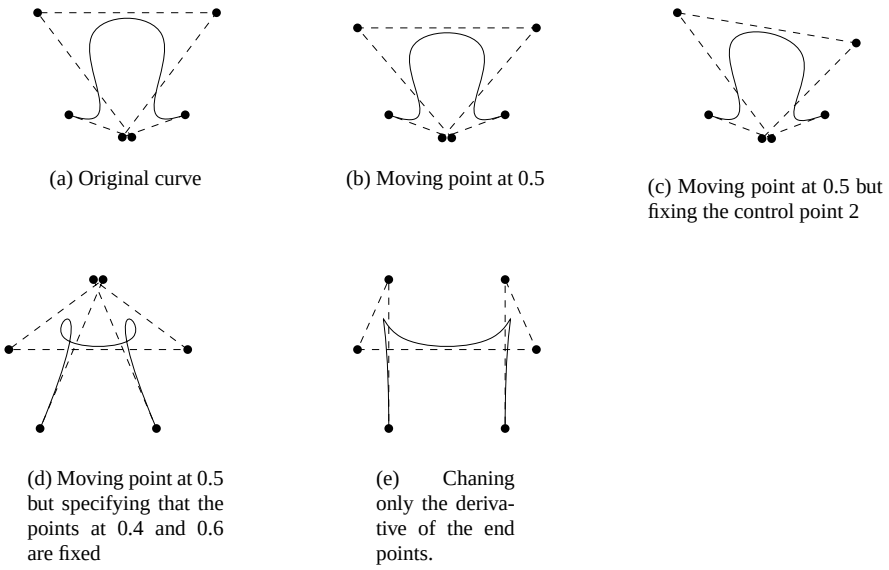
```
curve.modCP(i,cp) ;      // Cp[i] = cp
curve.modCPby(i,cpby) ;  // Cp[i] += cpby
```

The more complex function call modify the curve at a parameter value and only delta values can be specified. However, it allows you to also modify the derivative of the curve at a certain point and specify that some of the control points are fixed (i.e. that they can't be moved).

To explain the different methods, I will use an example (this is the same example as the one found in the file `tst/tnMovePoint.C` from the library)

```
  Vector_Point3Df pts(6) ;
  pts[0] = Point3Df(40,0,0) ;
  pts[1] = Point3Df(60,0,0) ;
  pts[2] = Point3Df(60,60,0) ;
  pts[3] = Point3Df(100,60,0) ;
  pts[4] = Point3Df(100,0,0) ;
  pts[5] = Point3Df(120,0,0) ;

  NurbsCurvef curve,tcurve ;

  curve.globalInterp(pts,3) ; tcurve = curve ; // graphic A
  curve.movePoint(0.5,Point3Df(0,-10,0)) ;  // graphic B

  Vector_Point3Df D(1) ;
  Vector_INT dr(1) ;
  Vector_INT dk(1) ;
  Vector_FLOAT ur(1) ;
  Vector_INT fixCP(1) ;

  D[0] = Point3Df(0,-10,0) ;
  dr[0] = 0 ;
  dk[0] = 0 ;
  ur[0] = 0.5 ;
  fixCP[0] = 2 ;

  curve = tcurve ;
  curve.movePoint(ur,D,dr,dk,fixCP) ; // graphic C

  curve = tcurve ;

  D.resize(3);  dr.resize(3);  dk.resize(3);  ur.resize(3) ;
  ur[0] = 0.4 ;   ur[1] = 0.5 ;   ur[2] = 0.6 ;
  D[0] = Point3Df(0,0,0) ;
  D[1] = Point3Df(0,-10,0) ;
  D[2] = Point3Df(0,0,0) ;
  dr[0]= 0; dr[1]= 1; dr[2]= 2; dk[0]= 0; dk[1]= 0; dk[2]= 0;

  curve.movePoint(ur,D,dr,dk) ; // graphic D

  Point3Df d0 = curve.derive3D(0.0,1) ;
  Point3Df d1 = curve.derive3D(1.0,1) ;
  D.resize(4); dr.resize(4); dk.resize(4); ur.resize(2);

  ur[0] = 0.0 ;   ur[1] = 1.0 ;
  D[0] = D[1] = Point3Df(0,0,0) ;
```

```
D[2] = Point3Df(-d0.x(),0,0) ;
D[3] = Point3Df(-d1.x(),0,0) ;
dr[0]=0; dr[1]= 1;  dr[2]= 0;  dr[3]= 1;  dk[0]= dk[1]= 0;
dk[2]= dk[3]= 1 ;

curve.movePoint(ur,D,dr,dk) ; // graphic E
```



(a) Original curve

(b) Moving point at 0.5

(c) Moving point at 0.5 but fixing the control point 2

(d) Moving point at 0.5 but specifying that the points at 0.4 and 0.6 are fixed

(e) Chaning only the derivative of the end points.

Figure 5: Moving at a parameter

The results of the example code is shown in figure 5. Except on the first call, some vectors need to be specified to properly call the `movePoint` function. These are

ur specifies the parameters that we want to modify

D specifies the deltas

dr specifies to which parameter the delta is applied

dk specifies to which degree the delta applies

fixCP specifies which control points are fixed, i.e. are not allowed to move.

Note that the vectors `D`, `dr` and `dk` must be of the same size since together they form the complete specification for Δ_r^k. For more information about this you should refer to the reference manual.

In some cases, the movement requested is impossible i.e. there is no way to move the control points while keeping the requirement of the caller. In those cases the function returns with 0.

The last modification possible is to split a NURBS curve at a given parameter value. This is done with a call to `splitAt` as in the following example:

```
curve.splitAt(0.3,c1,c2) ;
```

This creates two curves `c1` and `c2`. and the knot vector of `c1` will end at 0.3 and the knot vector of `c2` will start at 0.3. As long as you don't change these values you can later on merge them with a call to mergeAt. However, until a routine to reparameterize a NURBS curve exist, these two functions are of minimal use.

2.6 Knot operations

There are different operators available for knot modification. I will only describe the most used: knot insertion, knot removal and knot refinement. Beside knot removal, these functions do not change the geometrical aspect of the NURBS curve. However they do add control points to the NURBS curve. Because knot removal also removes a control point, it is unsure if the curve will remain the same.

Inserting a knot adds also a control point to the curve. In an interactive session this is often necessary to allow a user more control over the shape of the curves. Knot refinement is like inserting a knot except that it can insert a lot of them at the same time.

Knot removal is only useful in interactive sessions since the change to the geometry of the curve is undefined. However, there is a function named `removeKnotsBound` that only removes a knot if the resulting curve remains within an accepted margin of error. That function is used by the approximation routine.

3 NURBS surface