Advanced Trajectory Generation Using Rhino3D

Michael E. Aiello
11/7/2015

MEA Consulting

Revision -

Introduction

Numerical methods for the generation of trajectory as applied to machine control can be found to exist as far back as the early 19th century starting with tracing lathes and cams which were controlled manually. With the introduction of hydraulics larger parts could be fabricated using these devices.

The introduction of what was considered to be the first Numerical Controller (NC) based machine was due to John T. Parsons around 1940 using an X/Y table with indexed markers that would allow the table to be manually positioned in two dimensions. A *half axis* (Z) would be fitted with a cutter that would be lowered at predetermined points by an operator reading from a chart.

With the arrival of servo controllers to replace the manual positioning of the X/Y table and the control of the Z axis cutter the term *two-and-a-half-axis* machine controller came to being (see History of numerical control).

Not surprisingly, the first versions of the NC machine utilized *linear interpolation* between programmed points. By the late 1970's due to the advances in micro-electronics, *circular interpolation* between programmed points became feasible and with this the introduction of the *Computer Controlled Numerical* machine (CNC) and *G code interpreter*[1]. Now the machine could combine linear interpolation with circular interpolation. The two axis (vector) velocity could now be made to be continuous by simply making the vector velocities between the segments of the *lines* and *arcs* equal to each other.

By the early 1980's with the availability of greater computer processing power, came the ability to accelerate and decelerate the continuous vector velocity under controlled conditions.

With the emergence of cubic spline interpolation, 3D motion could now be accomplished. One could piece together individual spline and line segments by simply equating the velocities and accelerations at the endpoints of the connecting segments.

Concurrent with the evolution of numerical control, were the development of parametric curves used to model physical shapes. The first of these classes of curves was know as Bezier Curves which appeared in the mid 20th century and was used primarily to design automobile bodies. The nature of how the curves are constructed also made them popular in computer graphics in the mid 70's.

At this point in time there was no direct usage of this type of curve in relation to trajectory control because unlike natural splines (cubic splines), there was no strait forwards means to in which to *bind* the velocity and acceleration at the end of one Bezier curve to the beginning of the next without affecting the shape of both curves. Later, variants of the Bezier curve were developed, notably the B-Spline and NURBS curves.

Like the Bezier, the shape B-Splines and NURB curves are controlled by parameters *local* to each of the segments. It it this characteristic that allows the shape of these curves to be manipulated without effecting the shape of the neighboring segments. It is the intent of this paper to demonstrate that continuous vector velocity, acceleration and jerk can be controlled using these class of curves without affecting the shape of a given segment or it's neighbors.

---

1 G codes (and M) codes are constructs for describing motion profiles and associated actuator control.

B-Spines and NURBS: Generalizations of the Bezier Curve

There is a multitude of documents on the WEB devoted to the theory and practical application of NURBS, B-Splines and Bezier curves. This paper focuses on NURBS which are extensions of B-Spines and NURBS. I have found some sites I believe best explain the concepts of these curves and provide just enough theory to support the information provided in this paper.

First, I would suggest referring to Wikipedia B-Spline Overview for a general explanation of this type of spline again taking note that B-Splines are extensions of the Bezier curve as described in this document

Next, for a more detailed review refer to the explanation of Bezier curves followed by a review of B-Splines and NURBS.

Finally I need to mention the characteristics of the software platform used to generate some of the information in this paper. The tool is Rhino3D (Rhinoceros), a 3D CAD platform.

A detailed discussion of this utility is beyond the scope of this document. The reader would be better served exploring web and the many references (and books) devoted to the use of this software.

There are three points that I would like to mention regarding this tool that makes it a standout relative to CAD development packages such as SolidWorks and AutoCAD.

- The tool has an open connection for Plugins with an SDK (software design kit) that allows user to create custom interfaces.
- An open source NURB processing library called openNURBS that can read, process and write Rhino 3D output files (.3dm) that can be opened, viewed and manipulated using Rhino 3D[2].
- A low price ($1000.0 for Windows, $500 for MAC)[3].

2 The openNURBS library was used to generate some of the information provided in this paper.
3 This is single site price. I am not sure how volume licensing is handled.

Organization and Presentation

As previously stated, some of the information provided in the following sections is based on the use of the Rhino3D tool and the accompanying openNURBS library. The openNURBS library and executable applications **Process** and **Execute** were built within Linux. The following is a block diagram of this Linux project.

Project Block Diagram (Diag. 1)

It should be noted that the blocks **Process** and **Execute** are representations of processes that may be found in an embedded design. **Process** would be part of the operating system environment where as **Execute** would be more closely associated with the control section.

Referring to (Diag. 1), The Rhino3D file to be processed is named *my_curves_Rhino3d.3dm* and is derived from a sample file named *my_curves.3dm* included with openNURBS distribution. The file was modified to include lines, arcs, and free-form NURBS connected together using the *tangent* object snap tool[4]. These objects were drawn on an arbitrary plane created in 3D space. The connected elements within *my_curves_Rhino3d.3dm* are shown in (Diag. 2).

my_curves_Rhino3d.3dm (Diag. 2)

There are eight elements (objects) within *my_curves_Rhino3d.3dm.* Each element is highlighted in the following diagrams. Note that within the object descriptions, the keywords *NURBS*, *Line* and *Arc*. Also note the ID tag for each object description.

The elements of of my_curves_Rhino3d.3dm are (in the order that they are drawn).

Free form NURBS:	Object 7	(Diag. 3)
Arc:	Object 6	(Diag. 4)
Line:	Object 5	(Diag. 5)
Free form NURBS:	Object 4	(Diag. 6)
Line:	Object 3	(Diag. 7)
Arc:	Object 2	(Diag. 8)
Line:	Object 1	(Diag. 9)
Free form NURBS:	Object 0	(Diag. 10)

These ID's can also be viewed from the file *my_curves_Rhino3d_obj_dump* which was generated with the help of the openNURBS library (see block diagram of (Diag.1) above).

my_curves_Rhino3d_obj_dump (Diag. 11)

Note that the objects shown in the object dump file above are not in the order drawn in my_curves_Rhino3d.3dm. Reordering of the objects is discussed in below.

---

4 Note that I will be referring to mechanisms within the Rhino-3D package such as *object snap.* The reader may have to acquaint him or herself with the menu commands within the GUI to gain a better understanding on how object are created and manipulated within the Rhino3D GUI.

One of the first things done in the **Process** block of (Diag. 1) is to use openNURBS to process and dump the objects contained in *my_curves_Rhino3d.3dm.* This is done in three stages

First the object created in *my_curves_Rhino3d.3dm* that are not described by NURBS (e.g., lines and arcs) are converted to NURBS. The file *my_curves_Rhino3d.3dm* is then regenerated by openNURBS to a file named *my_curves_Rhino3d_all_nurbs.3dm.* This new file can then be opened in Rhino3D and viewed. In conjunction with this translation, a new object dump file named *my_curves_Rhino3d_all_nurbs_obj_dump* is also created.

Next, the newly generated NURBS file *my_curves_Rhino3d_all_nurbs.3dm* is processed again to fix the knot vectors for all the NURBS objects[5]. Again, if desired this newly generated 3dm file (*my_curves_Rhino3d_all_nurbs_knots_fixed.3dm*) can be opened and viewed in Rhino3D. Likewise, an accompanying dump file *my_curves_Rhino3d_all_nurbs_knots_fixed_obj_dump* is generated.

Finally, the generated file *my_curves_Rhino3d_all_nurbs_knots_fixed.3dm* is a processed to reorder the object as a final preparation for execution by the **Execute** block. This file is created by the **Process** block and named *my_curves_Rhino3d_all_nurbs_knots_fixed_reordered.3dm* along with the accompanying dump file *my_curves_Rhino3d_all_nurbs_knots_fixed_reordered_obj_dump.*

The generated file *my_curves_Rhino3d_all_nurbs_knots_fixed_reordered.3dm* is opened in Rhino3D[6]

my_curves_Rhino3d_all_nurbs_knots_fixed_reordered.3dm (Diag. 12)

With the dump describing the objects. Comparing to (Diag. 11) above, note that all objects are now defined as NURBS.

my_curves_Rhino3d_all_nurbs_knots_fixed_reordered_obj_dump (Diag. 13)

In addition to this third round of reprocessing an executable file named *my_curves_Rhino3d.3dx* is created. The 3dx file is not part of Rhino3D/openNURBS distribution but instead created by the **Process** block to be used for execution by the trajectory generator code in the **Execute** block. This file is described in detail in the following sections.

For the purposes of evaluation, the **Execute** block contains code to generate a *point*[7] file named *my_curves_Rhino3d.xyz*. This file is used to verify the accuracy of the position information contained in file *my_curves_Rhino3d.3dx*. This is demonstrated in the following sections.

---

5 This is required in order to create an executable file that can be processed by the **Execute** process as discussed in the following sections.

6 The plot is labeled as *curve* to distinguish it from the point file *my_curves_Rhino3d.xyz* described in the following sections.

7 A file that simply contain a three dimensional array of points in space.

Creation of the executable 3DX file format

Looking at a the 3dm dump file *my_curves_Rhino3d_all_nurbs_knots_fixed_reordered_obj_dump* of (Diag. 13) we do not see an order (k) greater then 4. This is a limit typically imposed by Rhino3D (and not the openNURBS library)

In turns out that free form NURBS[8] creationed in Rhino3D results in a NURBS parametrization of Order 4, non-rational.

When transforming lines and arcs into NURBS equations using openNURBS the following is produced.

- For the representation of a line, an Order 2 non-rational NURBS parametrization is produced.
- For the representation of an arc, an Order 3 rational NURBS parametrization is produced.

Again, this can be seen by comparing the dump file of (Diag. 11) with that of (Diag. 13).

We cannot execute the NURBS algorithm in real time because this would require sending down order (k) sets of point and knot structures to the client (or execution engine). In a typical application, the point and knot count may be very large.

Furthermore, a literal execution of the equation would require significant amount of real time division, which would be prohibitive even for a processor like the TI C6000 DSP.

Instead I elect to use a straight forward method of execution that provides a separate algorithm for each "order" ("k") value, since the "order" number is limited to values of 2,3, and 4 as imposed by Rhino3D.

It should be noted that the openNURBS library requires that we add an extra knot at the points where two knots would normally contain the same value. This is termed the *superficial* knot required when using legacy NURBS interpolation.

To visualize the data structures making up the ".3dx" file, I am going to generate an outline for the calculations for order “k” (2, 3 and 4) NURBS along with the corresponding first derivatives. As will be pointed out below, in order to allow the NURBS to be implemented in a trajectory generator that can generate a constant vector velocity along a prescribed path, we need to calculate in addition, the second and third derivatives for each order. The calculations for these can be deduced from that described for the first order derivative.

NURBS Equation Decomposition (Diag. 14)

After reviewing this document, it can be seen that the **P**(t) vector position vs reference (time in this case) that was derived for each order is not a continuous function. For example, **P**(t) for order k = 3

8 NURBS curves that are drawn by hand.

is calculated using three sets of parameters (**P, H** and N) that represent at any given point in space, three adjacent NURBS parametrizations. So even though we applied the stipulation that each NURB segment be joined *tangent* in to it's adjacent segment in (Diag. 2) above, this does not mean that the velocity and acceleration vectors at the connection points of each segment are equal.

This is unlike the typical cubic spline parametrization that guarantees equal velocities and accelerations at the connection points by way of calculations derived from a tri-diagonal matrix.

For NURBS, discontinuous velocity an acceleration at the connecting point for each segment is the price that is paid in order to allow the degree of freedom of moving around the positions within the segment without affecting the positions defined by the parameterizations of the adjacent segment. More on this later.

Evaluating Vector Velocity, Acceleration and Jerk using openNURBS

The analytics for determining NURBS derivatives was outlined in (Diag.14) NURBS Equation Decomposition.

It turns out that the openNURBS library can evaluate higher order derivatives of NURBS contained in 3dm files like that shown in the dump file of (Diag. 13) above.

Referring back to (Diag. 1) Project Block Diagram the **Process** block (with the help of the openNURBS library) provides for the capability of evaluating the 1st, 2nd and 3rd derivative of the NURBS as a vector quantity[9] for the objects described (Diag. 13) my_curves_Rhino3d_all_nurbs_knots_fixed_reordered_obj_dump and plotted in (Diag. 12) my_curves_Rhino3d_all_nurbs_knots_fixed_reordered.3dm. This is shown in the following diagram.

Vector Derivatives of (Diag. 12) (Diag. 15)

The plot of (Diag. 15) represents the vector velocity, acceleration and jerk with respect to a constant reference (time). Note the discontinuities at the connecting points of each NURBS object.

Now using the uncompensated velocity *my_curves_Rhino3d.dat_vel_cnst_ref* shown in (Diag. 15) as a reference, velocity compensation is added.

Vector velocity compensated using a velocity compensated reference (Diag. 16)

Note the improvement in the vector velocity. However, this compensation alone is not enough. The improvement really takes hold when both velocity and acceleration information is added to the reference compensator.

Vector velocity compensated using a velocity and acceleration compensated reference (Diag. 17)

9 This vector is simply the sum-of-squares-squart-root of the component derivatives.

The vector velocity of the path shown in (Diag. 12) is now almost constant.

The characteristics for control of constant vector velocity for curve in (Diag. 12) dictate no major improvement when Jerk compensation is added as shown here.

Vector velocity compensated using a velocity, acceleration and jerk compensated reference (Diag. 18)

Before continuing, it should be noted that the compensated vector velocity shown in (Diag. 16), (Diag. 17) and (Diag. 18) are scaled to 1 PU.

The addition of Jerk compensation would be relevant if the curve of (Diag. 12) possessed more dramatic *bends* along the path.

A closeup look of (Diag. 18) however reveals that the compensation algorithm cannot handle in a precise manor the transition points between NURBS objects[10] as shown in the next diagram.

Transition errors between adjacent NURBS objects (Diag. 19)

Improvements in the algorithm must added to address these conditions. Otherwise, compensation is nearly perfect along the paths of each of the NURBS object. In fact if the entire path of (Diag. 12) where constructed with NURBS object, the vector velocity along the path would nearly perfect.

The important point here is that if a single NURBS curve describes a path or an offset of a path relative to a NURBS patch (NURBS surface), the transition errors are not an issue.

Finally, for a true implementation of a trajectory generator for NURBS in 3D space, controlled *reference* acceleration/deceleration between the starting and ending points is mandatory. The compensated reference generator can be modified to include a *ramping* function that can accelerate the reference from zero at the starting point and decelerate the reference to zero at the end point.

An addition requirement is that the total path distance must be calculated in order to determine the points where acceleration/deceleration must be applied. In the simulated environment, this can easily be determined using the openNURBS library.

A simple trapezoidal ramping function was added to the reference generator as shown in the next diagram.

Trapezoidal ramping function added to reference generator. (Diag. 20)

Because all that we are doing is applying a ramping function to the reference generator, it too is effected by the transition points between the NURBS as shown in this close-up of (Diag. 20).

Detail of Trapezoidal ramping function added to reference generator. (Diag. 21)

10 The spikes shown in (Diag. 19)

It should be emphasized that the (Diag. 20) above represents a trapezoidal vector velocity *ramp* of a 1 PU velocity vector starting at the beginning of the first NURBS object of (Diag. 12) and ending at end of the last NURBS object described in this 3dm file.

Runtime generation of Vector Velocity, Acceleration and Jerk

In the previous section a trajectory generator was constructed in order to execute at a controlled constant vector velocity, the 3D path described in (Diag. 12) above. This was accomplished using the **Process** block and the openNURBS library (see Diag. 1)

For a real life trajectory generator, we need to create an intermediate representation of the path described in (Diag. 12) in order that it may be executed by and embedded controller. The form of this intermediate representation is shown in the next diagram.

3dx Structure Definition (Diag. 22)

The position data points for this new 3dx file format (named *my_curves_Rhino3d.xyz*) is generated by the **Process** block so it can be compared to the design file *my_curves_Rhino3d_all_ nurbs_knots_fixed.3dm* shown in (Diag. 12). It is imported into Rhino3d with *my_curves_Rhino3d_all _nurbs_knots_fixed.3dm* already opened. This point file is superimposed over the 3dm file.

my_curves_Rhino3d.xyz imported into (Diag. 12) (Diag. 23)

A zoomed in shot within Rhino3D clarifies the position accuracy of the generated output file *my_curves_Rhino3d.3dx* relative to the original 3dm file *my_curves_Rhino3d_all_nurbs_ knots_fixed_reordered.3dm.*

Zoomed in section of (Diag. 23) (Diag. 24)

This zoomed in area is near the connection point between Object 6 and Object 7 described in the dump of (Diag. 13). These object are highlighted relative to the imported point file as shown in the following diagrams.

Zoomed in section of (Diag. 23) with Object 6 Highlighed (Diag. 25)

Zoomed in section of (Diag. 23) with Object 7 highlighted (Diag. 26)

The **Execute** block shown in (Diag. 1) is representative of an actual NURBS trajectory generator contained within an embedded controller. As such, without having access to a powerful function library like openNURBS, it must be constructed with a high level of efficiency in order to be able to execute in real time.

The actual code used to generate the point iin my_curves_Rhino3d.xyz shown in (Diag. 23), (Diag. 24), (Diag. 25) and (Diag. 26) above is shown in the following diagram.

Trajectory Generator for producing my_curves_Rhino3d.xyz in (Diag. 23) (Diag. 27)

The algorythm of (Diag. 27) however is incomplete. First it is missing the code necessary to compute the 2nd and 3rd vector derivatives (acceleration and Jerk) of the position path which is necessary for compensating the reference generator like that depicted in the openNURBS assisted simulation plot of (Diag. 18) above.

Also this code lacks the routine to generate a trapezoidal *ramp* of the reference like that shown in the simulation plot of (Diag. 20) above.

However, along with the position generation shown in (Diag. 23) above, the vector path velocity (uncompensated) can be generated. This is shown in the following plot as *my_curves_Rhino3d .dat_xyz_vel_cnst_ref* and is compared with the openNURBS assisted simulated vector path velocity plot *my_curves_Rhino3d.dat_vel_cnst_ref* of (Diag. 15).

my_curves_Rhino3d.dat_xyz_vel_cnst_ref compared with velocity plot of(Diag. 15) (Diag. 28)

As can be seen the real time calculation for vector path velocity is identical to that generated with the help of the openNURBS library.

The NURBS Patch: Representations of surfaces in 3D Space

Given the general mathematical expression for NURBS, if we remove the **N**on-**U**niform and **R**ational characteristics we are left with a **B**-**S**pline equation as defined in (87) of B-Splines. Applying the tensor product of two independent sets of (87) we end up with the B-Spline patch as described in (93) of this same reference. Add the non-uniform and rational characteristics back into the each equation, we have the NURBS patch.

I good document that explains a *tensor product surface* including a NURBS surface (or patch) is here.

Tensor Product Surface (Diag. 29)

For simplicity this document begins with the explanation of the *Bezier surface.* Form this, the construction of the NURBS surface is inferred.

If we refer back to (Diag. 14) NURBS Equation Decomposition and the definition for the run

time implementation of a degree k-1 = 2 (Order k = 3) NURBS curve which involves at any given reference value, three control parameters (segments):

$$\mathbf{P}(c) = \frac{\mathbf{P}1*H1*N1,3 + \mathbf{P}2*H2*N2,3 + \mathbf{P}3*H3*N3,3}{H1*N1,3 + H2*N2,3 + H3*N3,3} \qquad \text{(Diag. 30)}$$

It may be deduced that the summation (i = 0 to m) of **P**ij **B**i in equation (15.5) shown in reference (Diag. 29) above represents the equation of (Diag. 30) for reference **c**. With this solved, **Q**j is known which in turn presents the solution for **P**(c,t) of equation (15.5).

Given this information, we can construct an infinite number of NURBS curves that follow any sets of points on the NURBS surface.

It turns out that with the use of Rhino3D and the openNURBS library, there is no need to embed an algorithm in the runtime implementation of the NURBS generator[11] (the **Execute** block of (Diag. 1)) if the NURBS curve that is to be derived from the NURBS surface can be created off line using Rhino3D.

To illistrate this, we again turn to the open*NURBS distribution for an example of a NURBS surface. The example file is called* v3_Patch.3dm. This file is shown in (Diag. 31) open in Rhino3d.

Surface created with NURBS patches (Diag. 31)

The image is actually created with five NURBS patches.

By using the Rhino3d command *Curve->Free_Form->Interpolate_on_Surface* we draw an arbitrary NURBS curve on the surface of one of the patches. This is shown in (Diag. 32)

NURBS curve drawn on NURBS surface (Diag. 32)

The NURBS surface and the NURBS curve drawn on this surface is illistrated in an object dump of this 3dm file which is edited to show only these two objects.

Object Dump of v3_patch (edited) (Diag. 33)

The objects defining the NURBS surface and curve drawn on this surface are shown in the next two diagrams.

NURBS surface object (Diag. 34)

NURBS curve object on surface (Diag. 35)

Next, using the command *Curve->Offset->Offset_Normal_to_Surface* we can create a NURBS

11 Not to say that there could not be applications where such an implementation would be useful.

curve *lifted off* the surface by 15 cm with all points normal to the original curve. This is shown in (Diag. 36).

NURBS curve offset on NURBS surface (Diag. 36)

To illustrate that this new NURBS curve is normal to the original curve drawn on the surface, we use the *object-snap* feature of Rhino3d to connect the ends and centers of both curves together with lines. This is shown in the next diagram.

Lines connecting both NURBS curves (Diag. 37)

If one were to select (highlight) any of the connecting lines between the two curves, a measurement of 15 cm would be indicated. This simple test demonstrates that if one were to select a reference for each curve that had the same proportion of a selected reference value divided by the the total reference that defines the curve, the two points in space should be separated by 15 cm. This then indicates the two curves are normal to each other relative to points on the surface that make up the original curve.

The ability to create a curve offset from the surface is useful in scanning and milling applications.

Finally, it should be noted that surface(s) contained in a Rhino3D 3dm file are not required to be defined strictly as NURB patches during object creation in order to take advantage of capability of drawing or creating NURBS curves on that surface. This is demonstrated by an example provided by the openNURBS library distribution in which the image is comprised entirely of entities called BREP's[12].

The example is called *v4_PerfumeBottle.3dm*. The next diagram shows that we can use the same example demonstration above for drawing and offsetting of NURBS curves on a surface that is not initially created as a NURBS surface.

BREP image with NURBS curve and its offset on surface (Diag. 38)

As stated, the surface of the bottle was created entirely of BREP objects as highlighted in the next diagram.

Image with BREPS highlighted (Diag. 39)

The entire image of the bottle however is automaticly translated to NURBS surfaces when processing the 3dm file using openNURBS. This is defined by an edited dump of the objects contained in *v4_PerfumeBottle.3dm* defining the base of the bottle.

Object dump of v4_PerfumeBottle (Edited) (Diag. 40)

The main object (Object 2) is defined as a ON_Brep object comprising of a mixture of NURBS curve and surface information along with BREP information that actually produces the image in Rhino3D.

12 See https://en.wikipedia.org/wiki/Boundary_representation for explanation of BREP's.

SUMMARY

A new method for trajectory generation suitable for 3D motion path planning using NURBS has been presented as an alternative to traditional NC machine and G code interpreters.

This new approach lends itself to a variety of new applications such as 3D Machining and surface scanning using an articulated robot.

An approach for executing constant vector velocity along a path derived directly from NURBS curves was also described. This was accomplished by using a 3D CAD system know as Rhino3D and a support library known as openNURBS. The result was the generation of a 3dx that is suitable for execution on an embedded controller.